The window-management system is first and foremost a resource manager in much the same way that an operating system is a resource manager—only the types of resources differ. It allocates the resource of screen area to various programs that seek to use the screen, and then assists in managing these screen areas so that the programs do not interfere with one another. This aspect of window systems is further discussed in Section 10.3. The window system also allocates the resource of interaction devices to programs that require user input, and then routes the flow of input information from the devices to the event queue of the appropriate program for which the input is destined. Input handling is discussed further in Section 10.4.

Our objective with these three sections is to provide an overview of key window-management concepts: The most comprehensive treatment of the subject is [STEI89], and a historical development overview is given in [TEIT86], in a book of relevant papers [HOPG86b].

A window-management system has two important parts. The first is the *window manager*, with which the end user interacts to request that windows be created, resized, moved, opened, closed, and so on. The second is the underlying functional component, the *window system*,which actually causes windows to be created, resized, moved, opened, closed, and so on.

The window manager is built on top of the window system: The window manager uses services provided by the window system. The window manager is to its underlying window system as a command-line interpreter is to its underlying operating-system kernel. Also built on top of the window system are higher-level graphics packages and application programs. The programs built on the window system are sometimes called *client* programs, which in turn use the capabilities of the window system, itself sometimes called the *server* program. In some server–client window-management systems, such as the X Window System [SCHE88a] and NeWS [SUN87], the window manager itself appears to the window system as just another client program. In other systems, there is a closer relationship between the window manager and window system than there is between a client and server.

Some window systems, including the X Window System and NeWS, are designed to be *policy-free*, meaning that multiple window managers, each with a different look and feel, can be built on top of the window system. The window manager, not the window system, determines how windows look, and how the user interacts with windows. A policy-free window system would support all the window styles of Fig. 9.30, as well as others. Just as many different application programs can be built on top of a graphics package, many different window managers can be built on top of a policy-free window system: The window manager and graphics application program both control external appearance and behavior. For this approach to be possible, the window system must be designed to carry out a wide range of window-manager policies (see Exercise 10.10). Of course, in a specific environment, the window manager and application programs need to have a common user-interface look and feel.

If the programming interface to the window system is cleanly defined and is implemented via an interprocess communication capability, then clients of the window system can reside on computers different from that of the window system, provided the computers are connected by a high-speed network. If the window manager is

Samsung/Google, EX1027, p. 454

itself just another client of the window system, then it too can reside on another computer. The use of interprocess communications in this way allows computation-intensive applications to reside on a powerful computer, while the user interacts with the application from a workstation. In this regard, the server–client model is just a sophisticated instance of a virtual terminal protocol; such protocols in general share this advantage.

A window-management system does not need to be built on the server–client model. For instance, the Macintosh has no well-defined separation between the window manager and window system. Such separation was not necessary for the single-active-process, single-processor design objective of the Macintosh, and would have led to additional run-time overhead.

In window systems that provide for use of interprocess communications between the window manager and window system, such as the X Window System, NeWS, and Andrew [MORR86], the interface must be designed to minimize communications delays. Several strategies can help us to meet this objective. First, asynchronous rather than synchronous communications can be used between the client and server whenever possible, so that, when the client sends a message to the server, the client does not need to wait for a reply before resuming processing and sending another message. For example, when the client program calls DrawLine, a message is sent to the server, and control returns immediately to the client.

We can sometimes realize a modest savings by minimizing the number of separate messages that must be sent between the server and client. Most network communication protocols have a minimum packet size, typically 16 to 32 bytes, and the time to send larger numbers of bytes is often proportionately less than the time needed to send the minimum packet. There is thus an advantage in batching messages, as provided in some systems with a BeginBatchOfUpdates and EndBatchOfUpdates subroutine call; all the calls made between the BeginBatch and EndBatch calls are transmitted in one message. There is also an advantage in designing single messages that replace multiple messages, such as a single message that sets multiple graphics attributes.

A third way to minimize communication is to move more functionality and generality into the server. The commands most clients send to their window system are fairly primitive: draw line, create window, copy pixmap. In the X Window System, for instance, many commands are needed to create a menu or dialogue box. A more robust and powerful strategy is to send commands to the server as programs written in a language that can be interpreted efficiently. Thus, the commands can be very general and can carry out any functionality the language can express. The cost of this generality is, of course, the time taken to execute the programs interpretively, and any additional space needed for the interpreter—a modest price with contemporary computers. The benefit can be a dramatic decrease in communications traffic. The strategy of moving more generality and functionality into the workstation is not new; exactly the same issues were discussed two decades ago when distributed graphics systems were first being built [FOLE71; FOLE76; VAND74].

This third strategy is used in the NeWS window system, which accepts as commands programs written in an extended version of the PostScript language [ADOB85a; ADOB85b]. PostScript combines traditional programming-language constructs (variables,

Samsung/Google, EX1027, p. 455

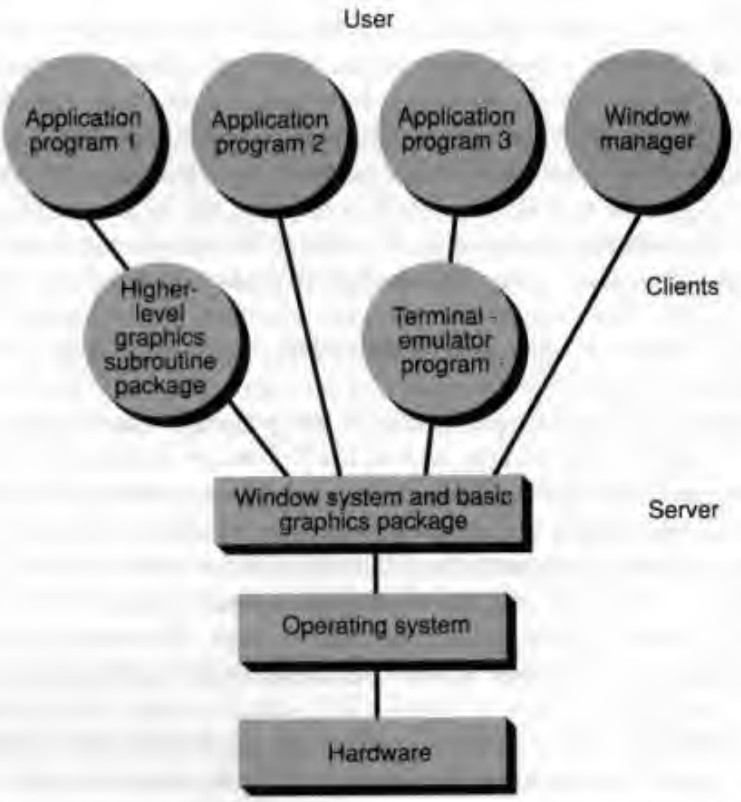


Fig. 10.4 The relationship of the window system to the operating system and application programs.

data structures, expressions, assignments, control flow, I/O) with imbedded graphics capabilities for drawing output primitives, clipping against arbitrary regions, and transforming primitives. NeWS adds extensions for processes, input, and windows. The language is further discussed in Section 19.9.

For a dialogue box to be defined in NeWS, a PostScript program defining the box is sent to the server when a program begins execution; each time the dialogue box is to appear, a short message is sent to invoke the program. This strategy avoids resending the box's definition each time. Similarly, programs to perform time-critical operations, such as rubberband drawing of lines (Chapter 8) or curves (Chapter 11), can be sent to the server, avoiding the time delays involved in each round-trip message between the server and client needed to update the rubberband line.

A graphics package is often integrated with the window system, typically a 2D nonsegmented package with capabilities similar to SRGP of Chapter 2. If the underlying hardware has 3D or segmentation capabilities, then the window-system level might pass on graphics calls to the hardware. Figure 10.4 shows how the window system and its graphics package typically relate to other system components; Fig. 10.5 shows the relationships among windows, clients, and input events. User-generated events involving windows—resizing, repositioning, pushing, popping, scrolling, and so on—are routed by the window system to the window manager; other events are routed to the appropriate application program.

Samsung/Google, EX1027, p. 456

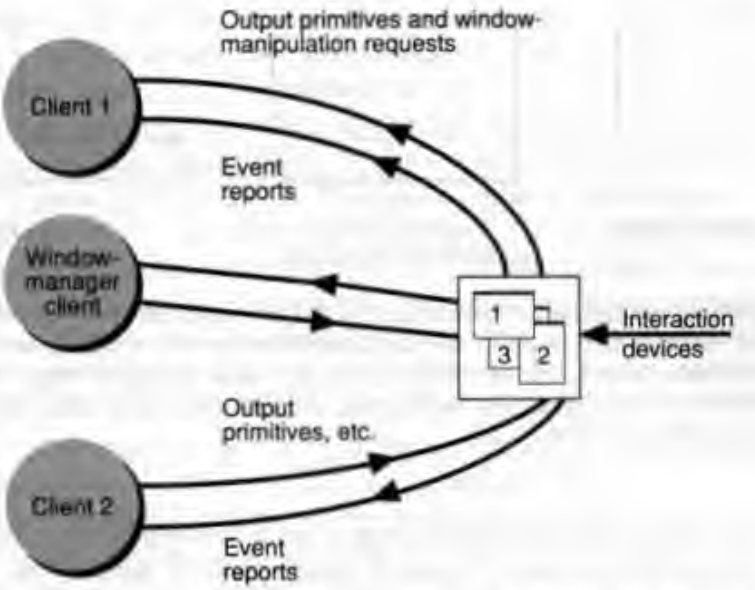


Fig. 10.5 Another view of the relationship among windows, clients, and events. Each client outputs to a window; input events are routed to the appropriate client's event queue.

10.3 OUTPUT HANDLING IN WINDOW SYSTEMS

The output resource allocated by the window system to its client programs is screen space, which must be managed so that clients do not interfere with one another's use of the screen. The strategies by which this allocation is made vary considerably from one window system to another, but fit into three broad categories. The main difference is in how parts of a window that have just been exposed (when the window is made larger, uncovered, or scrolled) are displayed. The strategies place progressively more responsibility for making this decision on the window system itself, such that the client is progressively less aware of the existence of the windows. The system may also have to manage a look-up table so as to avoid conflicts between clients.

A minimal window system takes no responsibility for drawing newly exposed portions of a window; rather, it sends a "window-exposed" event to the client responsible for the window. Such a window system does not save the occluded portions of windows. When the client sends output to a window, output primitives are clipped against the window system's window (which corresponds to the graphics package's viewport). If a line is drawn in a partially visible window, only the visible part of the line is drawn.

When the client responsible for the window receives the window-exposed event, the client can handle the event by establishing a clipping window on the entire portion of the world that is to be displayed in the window system's window, and then processing all the output primitives, or by using a smaller clipping window corresponding to just the newly exposed portion of the window, as shown in Fig. 10.6. Alternatively, the client can save the pixmaps of obscured windows to avoid regenerating the window's contents when the window is next uncovered. The client would simply display some or all of the saved pixmap with a PixBlt from the saved pixmap to the screen pixmap. In some cases, the window system repaints the window backgrounds and borders to provide the illusion of quick

Samsung/Google, EX1027, p. 457



Fig. 10.6 When a window (a) is enlarged to (b), the enlarged window can be updated by using the bitmap that was in the original window (a), clipping world-coordinate output primitives against the clipping region (c), and drawing those within the clip region. Note that the clip region is nonconvex, so clipping a line may result in display of more than one line segment.

response, even though the actual redraw may incur a noticable delay.

This minimal-window-system approach makes sense in systems that support diverse application data models, ranging from pixmaps through structured hierarchies to more complex relational models. The window system itself cannot have a model that is well matched for all applications. Instead, the application is given total responsibility for providing efficient storage and redrawing capabilities.

The Macintosh window system is an example of this minimalist design [APPL85]. Algorithms embedded in its Quickdraw graphics package allow clipping to arbitrary regions, such as those shown in Fig. 10.6. NeWS and X also support arbitrary-shaped windows. Some of the clipping algorithms needed to support arbitrary windows are discussed in Section 19.7.

More memory-rich window systems save the obscured parts of windows, so that the client does not need to display newly exposed portions of a window. Other window systems give the client a choice of whether or not obscured window parts are to be saved. In any case, there is a question of how much of the obscured window is saved. Typically, the maximum possible size of the window is saved, which is usually the size of the entire screen. Some window systems save a pixmap larger than the display itself, although this approach becomes more expensive as the pixmap becomes larger or deeper. Decreasing memory prices, however, are having a dramatic effect on what is cost-effective. The client must be involved in redrawing if the window is scrolled away from the part of the world that has been saved as a pixmap, or if the view must be rescaled.

A slightly different strategy is for the window manager to keep, for each window, an offscreen pixmap containing the entire window. Whenever part of a window is unobscured, the appropriate subarea of the offscreen pixmap is copied to the screen by a PixBlt. This strategy is slow for window updating, because partially obscured windows can be written into by client programs. Thus, after a client program writes into a window (which is the offscreen pixmap), the window system must copy the visible part of the window to the screen. Alternatively, the window system can directly scan convert new output primitives into both the offscreen pixmap and the visible part of the window in the onscreen pixmap, by clipping each output primitive against two clip regions: one for the visible part of the pixmap, the other for the entire offscreen pixmap. Updates to a completely unobscured window can be done faster by updating only the visible, onscreen version of the window;

Samsung/Google, EX1027, p. 458

the window is then copied to its offscreen pixmap only when it is about to be obscured. To avoid this whole issue, some window systems require that the window that is being written into be completely unobscured. This solution, unfortunately, also prevents multiprocessing when an active process needs to write into partially obscured windows. Several special-purpose hardware systems avoid the need to copy offscreen pixmaps to the screen pixmap. In these systems, the hardware knows where each offscreen pixmap is stored and which portion of each is to be made visible on the screen. On each refresh cycle, the video controller picks up the appropriate pixels from the offscreen pixmaps. Exposing more of a window is done not by copying pixmaps, but by giving new window-visibility information to the hardware. These hardware solutions are further discussed in Chapter 18.

A second way to implement this type of window system, developed by Pike [PIKE83] and discussed further in Section 19.8, avoids storing any information twice. Each window is partitioned into rectangular pixmaps. Invisible pixmaps are saved offscreen, whereas visible pixmaps are saved only in the onscreen refresh memory.

Yet another fundamental design strategy is to have the window system maintain a display list for each window, as in the VGTS system developed by Lantz and Nowicki [LANT84]. In essence, the window system maintains a display-list–based graphics package, such as SPHIGS, as part of the window system. Whenever a window needs to be redrawn, the display list is traversed and clipped. Fast scan-conversion hardware is desirable for this approach, so that redraw times do not become prohibitively long. Pixmap-oriented applications, such as paint programs, do not benefit from this approach, although VGTS does include a pixmap primitive.

A frequent concern is the effect of a window-resize operation on the amount of information shown in a window: what happens to the world-coordinate window when the window-manager window is resized? There are two possibilities, and the client program should be able to cause either to occur. The first possibility is that, when the user resizes the window, the world-coordinate window changes size correspondingly. The net effect is that the user sees more or less of the world, according to whether the window was made larger or smaller, but always at the same scale, as depicted in Fig. 10.7(c). The second possibility is that, when the user resizes the window, the world-coordinate window size stays fixed. Thus, as the window is enlarged, the same amount of the world is seen, but at a larger scale. In one approach, a uniform scaling is applied to the world, even if the aspect ratios of the world window and window-system window are different; this can make some part of the window-system window go unused, as in Fig. 10.7(d). Alternatively, a nonuniform scaling can be applied, distorting the contents of the world window to fit the window-system window, as in Fig. 10.7(e).

Several window systems use hierarchical windows—that is, windows that contain subwindows—as shown in Fig. 10.8. Subwindows are generally contained within their parents. Hierarchical windows can be used to implement dialogue boxes and forms of the type shown in Chapter 8. The entire dialogue box is defined as a window, and then each field, radio button, and check box is defined as a separate subwindow, and mouse button-down events are enabled for each one. When the user selects any of the subwindows, the event report contains the name of that subwindow. A typical restriction is that subwindows be contained within their parent window, so if the dialogue box is to be moved outside of the application window, the box cannot be implemented as a subwindow to the application window.

Samsung/Google, EX1027, p. 459

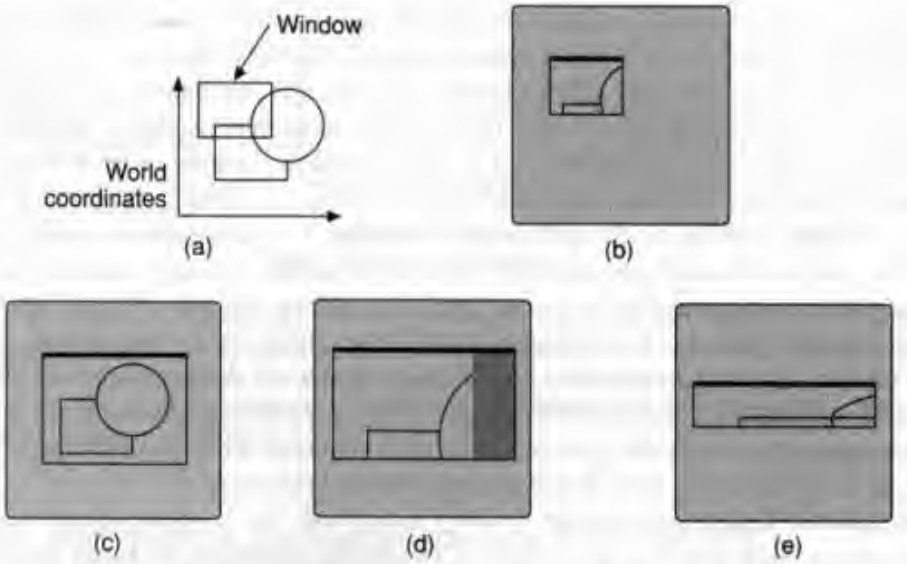


Fig. 10.7 Relationships between world-coordinate window and window-manager window. (a) A world-coordinate scene; (b) its view through a window. In (c), when the window-manager window is enlarged, more of the world is seen: the world-coordinate window is enlarged. In (d), enlarging the window-manager window creates an enlarged view of the contents of the world-coordinate window. The enlargement is done with uniform scaling, so that part of the window-manager window (gray tone) is unused. In (e), enlarging the window-manager window also creates an enlarged view of the contents of the world-coordinate window, but with nonuniform scaling so as to fill the entire window-manager window.

The design of a window-hierarchy system involves many subtleties, such as determining the effect on children of resizing a parent. Also, if a client process spawns a subprocess that then creates windows, the subprocess's windows could be considered subwindows of the spawning process's window, except that the subprocess may have its own event queue to receive input from its windows. See [ROSE83; SCHE86] for a more extensive discussion of hierarchies.

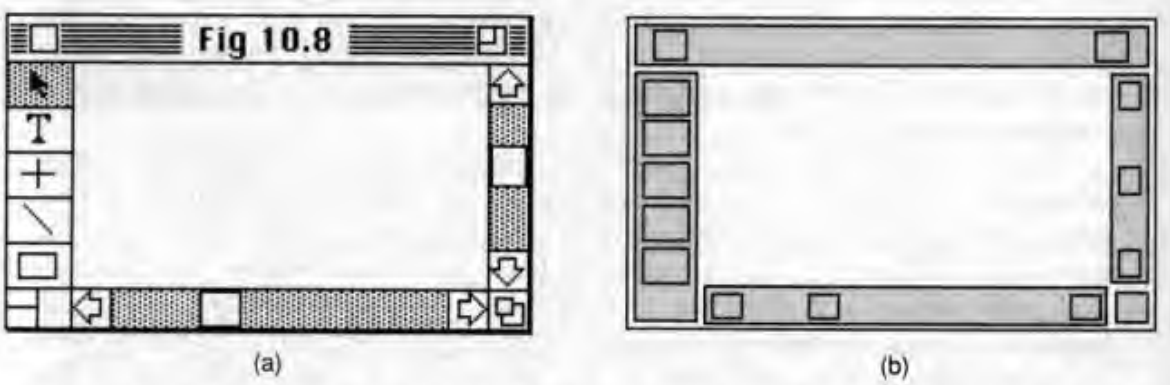


Fig. 10.8 (a) The window for a drawing program; (b) division of the window into a hierarchy of windows. In (b), a contained window is a child of the containing window. (Copyright 1988 Claris Corporation. All rights reserved.)

Samsung/Google, EX1027, p. 460

The subroutine calls with which a typical window system must deal follow:

CreateWindow (*name*)	Create a new window; it becomes the current window
SetPosition (*xmin*, *ymin*)	Set position of current window
SetSize (*height*, *width*)	Set size of current window
SelectWindow (*name*)	Make this the current window
ShowWindow	Make the current window visible, on top of all others
HideWindow	Make the current window invisible; expose any windows the current window was hiding
SetTitle (*char_string*)	Set displayed title of current window to *char_string*
GetPosition (*xmin*, *xmax*)	Get position of the current window
GetSize (*height*, *width*)	Get size of the current window
BringToTop	Put the current window on top of all other windows
SendToBottom	Send the current window to the bottom, behind all others
DeleteWindow	Delete the current window

The other output resource allocated by a window system is look-up table entries. Imagine a window system running on an 8-bit per pixel hardware system, with two window-system clients each wanting to have a look-up table. With two clients, each could be given half the entries (128), but then the number of look-up table entries per client depends on the number of clients. A fixed upper bound on the number of clients could be established to determine the number of entries per client, but if there are in fact fewer clients, then some of the look-up table entries will be wasted. A single client at a time could be given exclusive use of the look-up table—often the client whose window contains the cursor. This solution is viable, but suffers in that the overall screen appearance can change dramatically as the table is given first to one client, then to another, and so on.

Another solution is to allocate not look-up table entries, but rather colors. If a client asks for 100 percent blue, and some entry already contains this color, the client is given the same index to use (but is not allowed to change the contents of the entry). If no entry contains 100 percent blue and there are free entries, one is allocated. Otherwise, the entry with the color closest to that requested is allocated. The danger is that the distance between the requested and actual color might be quite large; not being able to change the look-up table is also a disadvantage. Unfortunately, there is no generally satisfactory solution.

10.4 INPUT HANDLING IN WINDOW SYSTEMS

The input resource being allocated and managed by the window system for that system's clients is the set of input devices and the events the devices generate. The window system must know to which client different types of events are to be routed. The process of routing events to the proper client is sometimes called *demultiplexing*, since events destined for different clients arrive in sequential order from a single source and must then be fanned out to different clients.

The types of events are those discussed in Section 10.1, plus additional events that are specific to window systems. Some window systems generate *window-enter* and *window-leave* events, which allow a user interface to highlight the window containing the cursor without the overhead of constantly sampling the pointer device. Window systems that do not retain a record of what is displayed in each window generate *window-damage* events whenever a window needs to be redrawn. Damage occurs if the window is enlarged,

Samsung/Google, EX1027, p. 461

uncovered, or scrolled. The window-enter and window-leave events are generated by the window system in direct response to user actions, whereas the window-damage event is a secondary event generated when a client requests that a window be changed. All these events are routed to the client's event queue. Some user actions, such as closing a window, can cause damage events to be to sent to a large number of clients. The information in the event report is similar to that discussed in Section 10.1, but is augmented with additional event types and window-specific information.

If a window hierarchy exists, child windows can be manipulated just like the parent window, and can have associated events. The window name associated with an event is that of the lowest-level window in the hierarchy that contains the cursor and for which the event was enabled. This means that different types of events can be associated with different windows. Every event placed in the event queue has as part of its record the name of the window with which it is associated. It is also possible to report an event in a subwindow not just to the subwindow, but also to all the windows that contain the subwindow. The client would do this by enabling the same event for all the windows in the hierarchical path from the outermost containing window to the lowest-level subwindow. Thus, multiple event reports, each with a different window name, will be placed in the client's event queue.

With hierarchical windows, a pop-up menu can be defined as a main window subdivided into as many subwindows as there are menu items. As the cursor moves out of the window of menu item i into the window of menu item $i + 1$, a leave-window event is placed on the event queue with the window name of menu item i, followed by an enter-window event with the window name of menu item $i + 1$. The client program processes the first event by undoing the highlight feedback on menu item i. It processes the second event similarly, creating the highlight feedback on menu item $i + 1$. If the cursor enters some region of the pop-up menu (such as a title area at the top) that is not overlaid by a subwindow, then the enter-window event includes the name of the pop-up menu window. Window hierarchies tend to be used in this way, but user response time can be degraded by the processing time needed to manage the hierarchy and its many events. A NeWS-style window system does not generate such a large number of events to be processed, because the feedback can be handled within the server.

Hierarchical windows can be used for selection of displayed objects in much the same way as SPHIGS structures can be. Of course, subwindows are not as general as structures, but they are convenient for picking rectangularly shaped, hierarchically nested regions (NeWS and X support nonrectangular windows). Other uses for hierarchical windows include causing the cursor shape to change as the cursor moves from one part of the screen to another, and allowing pick detection on the handles sometimes used for manipulating graphics objects (see Fig. 8.42).

Two basic approaches are widely used by window systems to route events to clients: real-estate–based and listener routing. Many window systems actually provide both strategies, and allow the window manager to specify which to use. Some also allow a policy provided by the window manager to be used in place of either of these two basic approaches. *Real-estate–based* routing looks at which window the cursor is in when an event occurs; all events are directed to the client that created the window and include the name of the window as part of the event report.

To do real-estate–based routing, the window system must maintain a data structure that stores the bounds of each window, as shown in Fig. 10.9. When an event occurs, the data

Samsung/Google, EX1027, p. 462

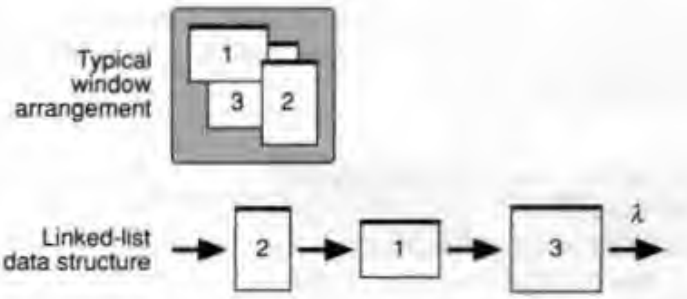


Fig. 10.9 Data structure used by the window manager to determine in which window the cursor is located, by searching for the first window on the list that brackets the cursor position.

structure is searched for the visible window containing the cursor position. If the data structure is ordered, with the most recently fully revealed window always brought to the head, then the search can terminate with the first window that contains the cursor (x, y) position. For hierarchical windows, a more complex data structure and search is needed (see Exercise 10.2).

Listener event routing, also known as *click-to-type* routing, is done when one client tells the window system to route all events of a certain type to another client (the receiving client can be, but does not need to be, the client that makes the request). For instance, the window manager can have a command that allows the user to route all keyboard input to the client that owns a particular window. The window manager implements the command by directing the window system to route keyboard events to the appropriate client program. Keyboard events are those most commonly routed explicitly, but even a mouse button-down event can be redirected.

Event distribution can cause unexpected results for users. Suppose, for instance, that the user accidentally double-clicks on a window's close (also called "go-away") button, although only a single click is needed to close the window. The window system routes the first click to the window manager, which closes the window. The second click is next routed to whatever was underneath the close button of the now-closed window, perhaps in turn selecting a menu command!

Message-transmission delays in a network can also wreak havoc with the user. Consider a drawing program executing in window A. To draw a rubberband line, the user generates a button-down event in window A, and then moves the cursor to wherever the line will end. This point might be outside of window A, if the line is to end in an area of the drawing that is not visible through the window. In anticipation of the cursor going out of its window, the drawing program sends the window system a request that *all* button-up events come to it, no matter where they occur. Now if the user does move the cursor outside window A, we have a *race condition*. Will the mouse-up event occur before or after the window system receives the request that all mouse-up events go to the drawing program? If the mouse-up occurs before the request is received, the event will go to whatever window the cursor is in, causing an unexpected result. If the mouse-up occurs after the request is received, all is well. The only certain way to avoid these race conditions is to require the client to tell the server that event i has been processed before the server will send event $i + 1$ to the client.

Samsung/Google, EX1027, p. 463

Once events come to a client, they enter an event queue of the sort shown in Figure 10.2. The client routes, or dispatches, events to various event-handling routines. The pseudocode for a typical dispatcher is

```
while (!quit) {
    WaitEvent (timeout, deviceClass, deviceId);
    switch (deviceClass) {
        case CLASS1: switch (deviceId) {
                case DEVICE1: ProcedureA (); break;
                case DEVICE2: ProcedureB (); break;
            }
        case CLASS2: switch (deviceId) {
etc.
```

As events occur and the program moves to different states, the procedures called in response to a particular event may change, further complicating the program logic.

The *dispatcher model* (also called the *notifier model*) enhances the input-handling system with a procedure that responds to the user actions, as shown in Fig. 10.10. The application program *registers* a procedure with the notifier and tells the notifier under what conditions the procedure is to be called. Procedures called by the notifier are sometimes called *callback procedures*, because they are called back by the notifier. With hierarchical windows, different callback procedures can be associated with locator events occurring in different parts of a dialogue box or in different parts of a menu. These callback procedures may modify the event before it is reported to the application, in which case they are called *filter procedures*.

The input subroutine calls with which a typical window system must deal include

EnableEvents (*eventList*)	Enable the listed set of events
WaitEvent (*timeout, eventType, windowName, eventRecord*)	Get the next event from the event queue
SetInputFocus (*window, eventList*)	Direct all input events of the type on *eventList* to *window*
CursorShape (*pixmap, x, y*)	*pixmap* defines the cursor shape; *x, y* give the position in the cursor pixmap used for reporting the cursor position

Typical types of events that can be placed on the event queue follow:

KeyPress	Keyboard key pressed
KeyRelease	Keyboard key released
ButtonPress	Locator (such as mouse) button pressed
ButtonRelease	Locator button released
Motion	Cursor has moved
LeaveNotify	Cursor has left window
EnterNotify	Cursor has entered window
WindowExpose	Window has been partially or completely exposed
ResizeRequest	Window resizing has been requested
Timer	Previously specified time or time increment has occurred

Each of these event types has a timestamp (see Section 10.1 to understand why this is needed), the name of the window where the cursor was when the event occurred, and other

Samsung/Google, EX1027, p. 464

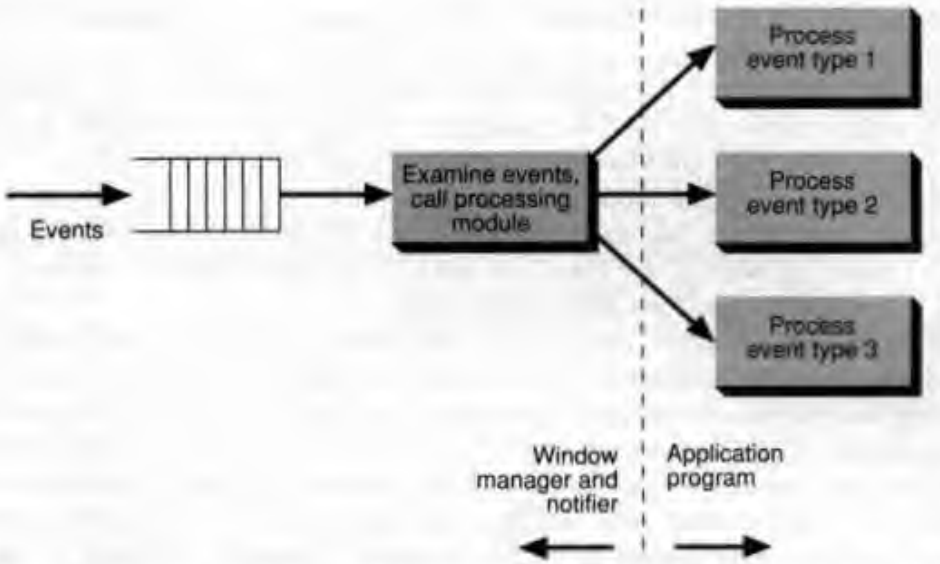


Fig. 10.10 The window manager's notifier examines the event queue and calls the procedure previously registered to process a particular event.

event-specific information, such as the new window size for a ResizeRequest.

This brief overview of window-management systems has excluded important topics, such as ensuring that the window system is sufficiently general that any and all types of window-manager policies can be provided. Also important is whether the window system is a separate process from the clients, is a subroutine library linked in with the clients, or is a part of the operating system. These and other issues are more fully discussed in [LANT87; SCHE86; SCHE88a; STEI89].

10.5 INTERACTION-TECHNIQUE TOOLKITS

The look and feel of a user–computer interface is determined largely by the collection of interaction techniques provided for it. Recall that interaction techniques implement the hardware binding portion of a user–computer interface design. Designing and implementing a good set of interaction techniques is time consuming: Interaction-technique toolkits, which are subroutine libraries of interaction techniques, are mechanisms for making a collection of techniques available for use by application programmers. This approach, which helps to ensure a consistent look and feel among application programs, is clearly a sound software-engineering practice.

Interaction-technique toolkits can be used not only by application programs, but also by the window manager, which is after all just another client program. Using the same toolkit across the board is an important and commonly used approach to providing a look and feel that unifies both multiple applications and the windowing environment itself. For instance, the menu style used to select window operations should be the same style used within applications.

As shown in Fig. 10.1, the toolkit can be implemented on top of the window-management system. In the absence of a window system, toolkits can be implemented

Samsung/Google, EX1027, p. 465

directly on top of a graphics subroutine package; however, because elements of a toolkit include menus, dialogue boxes, scroll bars, and the like, all of which can conveniently be implemented in windows, the window system substrate is normally used. Widely used toolkits include the Andrew window-management system's toolkit [PALA88], the Macintosh toolkit [APPL85], OSF/Motif [OPEN89a] and InterViews [LINT89] for use with the X Window System, several toolkits that implement OPEN LOOK [SUN89] on both X and NeWS, Presentation Manager [MICR89], and the SunView window-management system's toolkit [SUN86]. Color Plates I.26 and I.27 show the OSF/Motif interface. Color Plates I.28 and I.29 show the OPEN LOOK interface.

In the X Window System, interaction techniques are called *widgets*, and we adopt this name for use here. A typical set of widgets includes a dialogue box, file-selection box, alert box, help box, list box, message box, radio-button bank, radio buttons, choice-button bank, choice buttons, toggle-button bank, toggle button, fixed menu, pop-up menu, text input, scroll bar, and application window. Each of these widgets is normally implemented as a window. In X, subwindows may also be used. For instance, a radio-button bank is a window containing a subwindow for each radio button. Complex dialogue boxes can have dozens of subwindows. An application window may have subwindows for scroll bars, resize buttons, and so on, as in Fig. 10.8.

Interaction-technique toolkits typically have notifiers of the type discussed in Section 10.4 to invoke callback procedures when events occur in their subwindows. The procedures are, in some cases, part of the toolkit—for instance, procedures to highlight the current menu item, to select and deselect radio buttons, and to scroll a list or file-selection box. They can also be provided by the application; for instance, there are procedures to carry out a command selected from a menu, to check the validity of each character as it is typed into a text input area, or simply to record the fact that a button has been selected. Figure 10.11

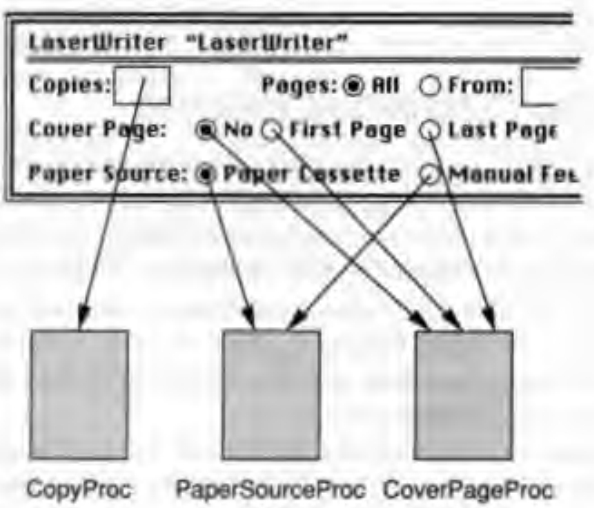


Fig. 10.11 Callback procedures associated with widgets in a dialogue box. CopyProc checks to ensure that each character is numeric and that the total number entered does not exceed some upper bound. PaperSourceProc manages the radio-button bank for the paper source to ensure that one and only one button is on and to maintain the current selection. CoverPageProc performs a similar function for the cover-page radio-button bank. (Screen graphics © Apple Computer, Inc.)

Samsung/Google, EX1027, p. 466

shows part of a dialogue box and some of the procedures that might be associated with the box.

Notice that the previous list of widgets includes both high- and low-level items, some of which are composites of others. For example, a dialogue box might contain several radio-button banks, toggle-button banks, and text-input areas. Hence, toolkits include a means of composing widgets together, typically via subroutine calls. Figure 10.12 shows just some of the code needed to specify the SunView [SUN86] dialogue box of Fig. 10.13. Some toolkits are built using object-oriented programming concepts: Each widget is an instantiation of the widget's definition, possibly with overrides of some of the methods and attributes associated with the definition. A composite consists of multiple instances.

```
print_frame =
  window_create(
    frame,    FRAME,                    {Surrounding box}
    WIN_SHOW,                           TRUE,
    FRAME_NO_CONFIRM,                   TRUE,
    FRAME_SHOW_LABEL,                   TRUE,
    FRAME_LABEL,    "Print",            {Header at top of window}
  0);                                   {Zero means end of list}

print_panel =                           {Panel inside the window}
  window_create(print_frame,            PANEL,
    WIN_ROWS,                           PRINT_WIN_ROWS,
    WIN_COLUMNS,                        PRINT_WIN_COLS,
  0);

print_uickb_name =                      {Header at top of panel}
  panel_create_item(print_panel,        PANEL_MESSAGE,
    PANEL_LABEL_STRING,                 "UICKB: Untitled",
    PANEL_ITEM_X,                       ATTR_COL(PRINT_NAME_COL),
    PANEL_ITEM_Y,                       ATTR_ROW(PRINT_NAME_ROW),
  0);

print_report_choice_item =
  panel_create_item(print_panel,        PANEL_CHOICE,
  {List of mutually exclusive options}
    PANEL_ITEM_X,                       ATTR_COL(PRINT_REPORT_COL),
    PANEL_ITEM_Y,                       ATTR_ROW(PRINT_REPORT_ROW),
    PANEL_LABEL_STRING,                 "Report",
    PANEL_LAYOUT,                       PANEL_VERTICAL,{Or horizontal}
    PANEL_CHOICE_STRINGS,
      "Completeness", "Consistency", "Schema", 0,
    PANEL_NOTIFY_PROC,                  print_report_choice_proc,
    {Name of callback procedure}
  0);
```

Fig. 10.12 Some of the SunView code needed to specify the dialogue box of Fig. 10.13.

Samsung/Google, EX1027, p. 467



Fig. 10.13 Dialogue box created using the SunView window-manager system's toolkit. The code specifying this box is shown in Fig. 10.12. (Courtesy of Kevin Murray, The George Washington University.)

Creating composites by programming, no matter what the mechanism, is tedious. Interactive editors, such as those shown in Figs. 10.14 and 10.15, allow composites to be created and modified quickly, facilitating the rapid prototyping discussed in Section 9.6. Cardelli has developed a sophisticated interactive editor that allows spatial constraints between widgets to be specified [CARD88]. At run time, when the dialogue box's size can be changed by the user, the constraints are used to keep the widgets neatly spaced.



Fig. 10.14 The SmethersBarnes Prototyper dialogue-box editor for the Macintosh. A scrolling-list box is being dragged into position. The menu to the left shows the widgets that can be created; from top to bottom, they are buttons, icons, pictures, static text, text input, check boxes, radio buttons, scrolling lists, rectangles (for visual grouping, as with the radio-button banks), lines (for visual separation), pop-up menus, and scroll bars. (Courtesy of SmethersBarnes, Inc.)

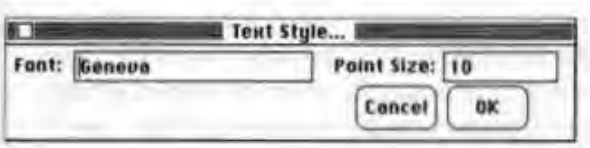
Samsung/Google, EX1027, p. 468



Fig. 10.15 An interactive editor for designing windows. The size, position, border style, title, and presence of the "go away" box can all be controlled. The editor shows the window at its actual size; the text file describing the window is at the upper right, and a dialogue box for controlling the window characteristics is at the upper left. The window's size and position can be modified by direct manipulation, in which case the values in the dialogue box are modified. The text file is written out as the permanent record of the window's design. The I.D. and Ref# form a name by which the application program can refer to the window. (Screen graphics © Apple Computer, Inc.)

The output of these editors is a representation of the composite, either as data structures that can be translated into code, or as code, or as compiled code. In any case, mechanisms are provided for linking the composite into the application program. Programming skills are not needed to use the editors, so the editors are available to user-interface designers and even to sophisticated end users. These editors are typical of the interactive design tools shown in Fig. 10.1.

Another approach to creating menus and dialogue boxes is to use a higher-level programming-language description. In Mickey [OLSE89], an extended Pascal for the Macintosh, a dialogue box is defined by a record declaration. The data type of each record item is used to determine the type of widget used in the dialogue box; enumerated types become radio-button banks, character strings become text inputs, Booleans become checkboxes, and so on. Figure 10.16 shows a dialogue box and the code that creates it. An




```
type
  Str40     = string[40]
  textStyle = record
                font : Str40;
                points ("Name = 'Point Size'") : integer
              end
```

Fig. 10.16 A dialogue box created automatically by Mickey from the extended Pascal record declaration. (Courtesy of Dan Olsen, Jr., Brigham Young University.)

Samsung/Google, EX1027, p. 469




```
procedure NewDrawing (
    (* Menu = File Name = 'New...' Key = N *)
    DrawFile : OutFileDesc);                    {Name of dialogue box to be shown.}
procedure OpenDrawing (
    (* Menu = File Name = 'Open...' Key = O *)
    DrawFile : InFileDesc);                     {Name of dialogue box to be shown.}
procedure CloseDrawing;
    (* Menu = File Name = 'Close' Key = W *)
procedure SaveDrawing;
    (* Menu = File Name = 'Save' Key = S *)
procedure SaveDrawingAs (
    (* Menu = File Name = 'Save As...' *)
    DrawFile : OutFileDesc);                    {Name of dialogue box to be shown.}
```

Fig. 10.17 A menu created automatically by Mickey from the extended Pascal record declaration. (Courtesy of Dan Olsen, Jr., Brigham Young University.)

interactive dialogue-box editor can be used to change the placement of widgets. Figure 10.17 shows a menu and the code from which it is generated.

Peridot [MYER86; MYER88] takes a radically different approach to toolkits. The interface designer creates widgets and composite widgets interactively, by example. Rather than starting with a base set of widgets, the designer works with an interactive editor to create a certain look and feel. Examples of the desired widgets are drawn, and Peridot infers relationships that allow instances of the widget to adapt to a specific situation. For instance, a menu widget infers that its size is to be proportional to the number of items in the menu choice set. To specify the behavior of a widget, such as the type of feedback to be given in response to a user action, the designer selects the type of feedback from a Peridot menu, and Peridot generalizes the example to all menu items.

10.6 USER-INTERFACE MANAGEMENT SYSTEMS

A user-interface management system (UIMS) assists in implementing at least the form of a user interface, and in some cases portions of the meaning as well. All UIMSs provide some means of defining admissible user-action sequences and may in addition support overall screen design, help and error messages, macro definition, undo, and user profiles. Some recent UIMSs also manage the data associated with the application. This is in contrast to interaction technique toolkits, which provide far less support.

UIMSs can increase programmer productivity (in one study, up to 50 percent of the code in interactive programs was user-interface code [SUTT78]), speed up the development

Samsung/Google, EX1027, p. 470

process, and facilitate iterative refinement of a user interface as experience is gained in its use. As suggested in Fig. 10.1, the UIMS is interposed between the application program and the interaction-technique toolkit. The more powerful the UIMS, the less the need for the application program to interact directly with the operating system, window system, and toolkit.

In some UIMSs, user-interface elements are specified in a programming language that has specialized operators and data types. In others, the specification is done via interactive graphical editors, thus making the UIMS accessible to nonprogrammer interface designers.

Applications developed on top of a UIMS are typically written as a set of subroutines, often called *action routines* or *semantic action routines*. The UIMS is responsible for calling appropriate action routines in response to user inputs. In turn, the action routines influence the dialogue—for instance, by modifying what the user can do next on the basis of the outcome of a computation. Thus, the UIMS and the application share control of the dialogue—this is called the *shared-control* model. A UIMS in which the action routines have no influence over the dialogue is said to follow an *external-control* model; control resides solely in the UIMS. External control is not as powerful a model as is shared control.

UIMSs vary greatly in the specific capabilities they provide to the user–interface designer, but the one essential ingredient is a dialogue-sequence specification, to control the order in which interaction techniques are made available to the end user. For this reason, in the next section, we turn our attention to dialogue sequencing; then, in Section 10.6.2, we discuss more advanced UIMS concepts. Further background on UIMSs can be found in [HART89; MYER89; OLSE84b; OLSE87].

10.6.1 Dialogue Sequencing

Permissible sequences of user actions can be defined in a variety of ways: via transition networks (also called state diagrams), recursive transition networks, event languages, or by example, where the designer demonstrates the allowable action sequences to the system and the system "learns" what sequences are possible. Common to all these methods is the concept of a user-interface *state* and associated user actions that can be performed from that state. The notion of state has been discussed in Section 9.3.1 (state diagrams) and in Section 9.4 (modes). Each of the specification methods encodes the user-interface state in a slightly different way, each of which generalizes to the use of one or more *state variables*.

If a context-sensitive user interface is to be created, the system response to user actions must depend on the current state of the interface. System responses to user actions can include invocation of one or several action routines, changes in one or more of the state variables, and enabling, disabling, or modifying interaction techniques or menu items in preparation for the next user action. Help should also be dependent on the current state. Since the outcome of computations performed by the action routines should affect user-interface behavior, the action routines must be able to modify the state variables. Thus, state is at the heart of context-sensitivity, a concept central to contemporary user interfaces.

The simplest and least powerful, but nevertheless useful, sequence specification method is the *transition network* or *state diagram*. Transition networks have a single state variable, an integer indicating the current state. User actions cause transitions from one

Samsung/Google, EX1027, p. 471

state to another; each transition has associated with it zero or more action routines that are called when the transition occurs. In addition, states can have associated action routines executed whenever the state is entered. This shorthand is convenient for actions that are common to all transitions entering a state.

The action routines can affect the current state of the transition network in one of two ways. First, they can place events in the event queue, which in turn drives the interaction handling. This approach implicitly modifies the state, although to ensure that the state change is immediate, the event must be put at the front of the queue, not at the back. Alternatively, the action routines can modify the state more directly by simply setting the state variable to a new value. The first approach is cleaner from a software-engineering view, whereas the second is more flexible but more error-prone.

A number of UIMSs are based on state diagrams [JACO83; JACO85; SCHU85; RUBE83; WASS85]. Some of these provide interactive transition-network editors, which makes the networks simple to specify. The first UIMS, developed by Newman and called *The Reaction Handler*, included such an editor [NEWM68]. A simple transition-network–driven UIMS is easy to implement—see Exercise 10.8.

Transition networks are especially useful for finding sequencing inconsistencies, as discussed in Section 9.3.1, and can easily be used to determine the number of steps required to complete a task sequence. Thus, they also serve as means of predicting how good a particular design will be, even before the complete interface is implemented. Consider, for example, the simple case of explicit versus implicit acceptance of results. Figure 10.18 represents a one-operand command with explicit acceptance and rejection of the results; Fig. 10.19 shows implicit acceptance and explicit rejection. In the first case, three steps are always required: enter command, enter operand, accept. In the second case, only two steps are normally needed: enter command, enter operand. Three steps are needed only when an error has been made. Minimizing steps per task is one goal in interface design, especially for experienced users, since (not surprisingly) the speed with which experienced users can input commands is nearly linearly related to the number of discrete steps (keystrokes, hand movements) required [CARD80].

Transition networks, however, have drawbacks. First, the user-interface state is typically based on a number of state variables, and having to map all possible combinations of values of these variables onto a single state is awkward and nonintuitive for the

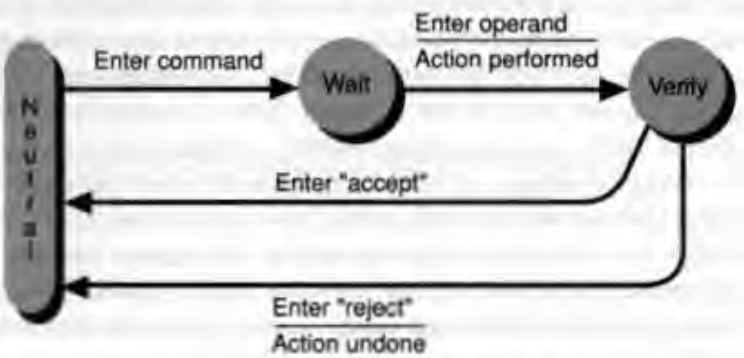


Fig. 10.18 Transition network for a dialogue with explicit acceptance and rejection of results.

Samsung/Google, EX1027, p. 472

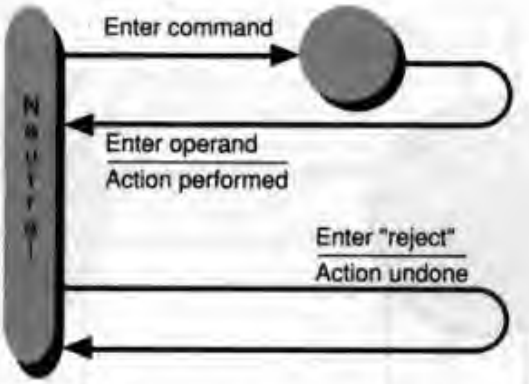


Fig. 10.19 Transition network for a dialogue with implicit acceptance and explicit rejection of results.

user-interface designer. For example, if commands are to behave in one way when there is a currently selected object (CSO) and in another way when there is no CSO, the number of states must be doubled to encode the "CSO exists–does not exist" condition. These types of context-sensitivities can expand the state space and make the transition networks difficult to create and understand. Figure 10.20, for example, shows a transition network for a simple application having the following commands:

- Select an object (establishes a CSO)
- Deselect the CSO (so there is no CSO)
- Create an object (establishes a CSO)
- Delete the CSO (so there is no CSO)
- Copy the CSO to the clipboard (requires a CSO, makes the clipboard full)
- Paste from the clipboard (requires that the clipboard be full, creates a CSO)
- Clear the clipboard (requires that the clipboard be full, empties the clipboard).

Four states are needed to encode the two possible conditions of the clipboard and the CSO. Notice also that whether or not any objects exist at all also should be encoded, since objects must exist for the command Select_object to be available in the starting state. Four more states would be needed to encode whether any objects do or do not exist.

Concurrency creates a similar state-space growth problem. Consider two user-interface elements—say, two concurrently active dialogue boxes—each with its own "state" encoding the selections currently allowable or currently made. If each dialogue-box state can be encoded in 10 states, their combination requires 100 states; for three dialogue boxes, 1000 states are needed; and so on. This exponential growth in state space is unacceptable. Jacob [JACO86] combines transition networks with object-oriented programming concepts to specify complete user interfaces while limiting the state-space explosion. Objects are self-contained entities within the interface, and each object has its own transition network to specify its behavior, which is independent of that of other objects. The UIMS portion of HUTWindows, the Helsinki University of Technology Window Manager and UIMS, uses a similar strategy [KOIV88].

Samsung/Google, EX1027, p. 473

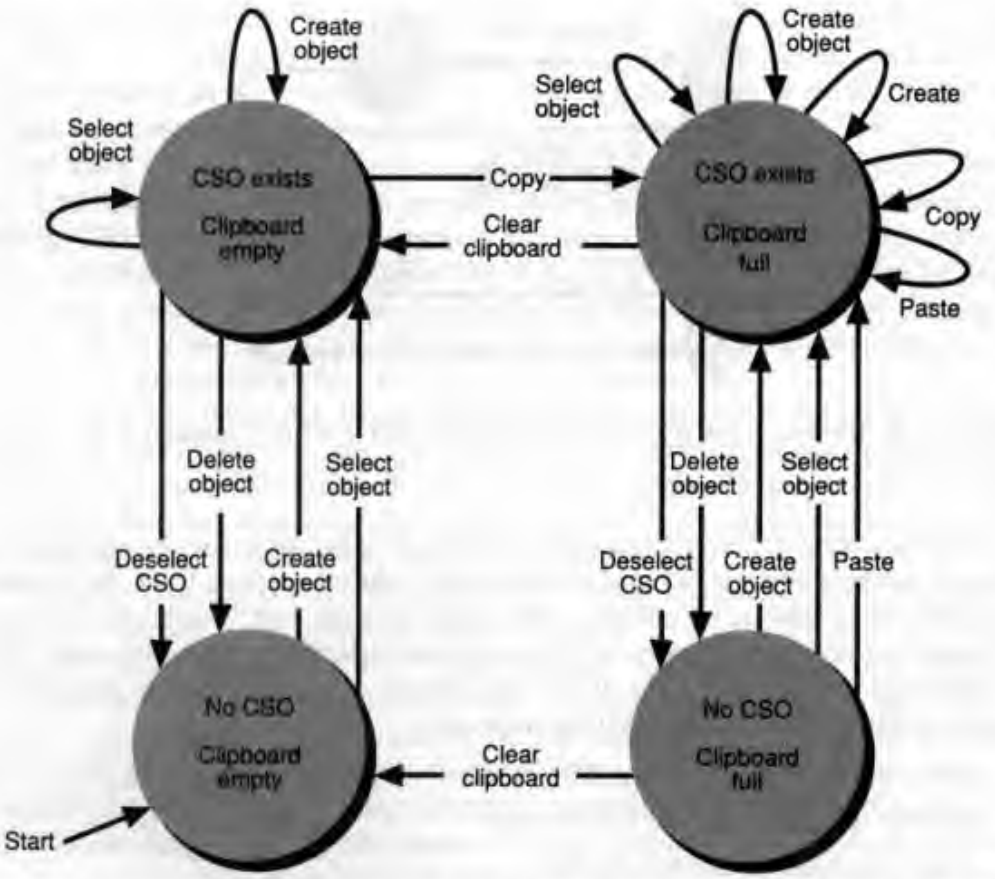


Fig. 10.20 A transition network with four states. Not all commands are available in all states. In general, action routines associated with transitions should appear on diagrams of this sort, with the names of the user actions (user commands in this case); we omit them here because the actions are obvious.

Globally available commands similarly enlarge the transition network. If help is to be globally available, each state must have an associated help state, a transition to the help state, and a reverse transition back to the originating state. This is also needed for the help to be context-sensitive. Undo must be done similarly, except that the transition from an undo state returns to a state different from the one from which it was entered. As the number of transitions relative to the number of states increases, we end up with complex "spaghetti" transition networks.

Various specialized constructs have been developed to simplify transition networks. For instance, we can alleviate the help problem by using subnetworks, in a fashion analogous to subroutines, to hide localized repetitive detail. Transition networks that can call subnetworks recursively are called *recursive transition networks*. The state variables in this case are the entire stack of saved states, plus the state of the currently active transition network. Several other powerful diagramming techniques, all derived from transition networks, are described in [WELL89].

Samsung/Google, EX1027, p. 474

```
<command> ::= <create> | <polyline> | <delete> | <move> | STOP
<create> ::= CREATE + <type> + <position>
<type> ::= SQUARE | TRIANGLE
<position> ::= NUMBER + NUMBER
<polyline> ::= POLYLINE + <vertex list> + END_POLY
<vertex_list> ::= <position> | <vertex_list> + <position>
<delete> ::= DELETE + OBJECT_ID
<move> ::= MOVEA + OBJECT_ID + <position>
```

Fig. 10.21 Backus–Naur form representation of the sequencing rules for a simple user interface.

Backus–Naur form (BNF) can also be used to define sequencing, and is equivalent in representational power to recursive transition networks (both are equivalent to push-down automata). BNF, illustrated in Fig. 10.21, can also be shown diagrammatically as the diagrams of Fig. 10.22. It is difficult to read BNF and to obtain a good overview of the sequencing rules, but BNF form can be processed to provide an evaluation of certain aspects of user-interface quality [BLES82, REIS82], or to generate command-language parsers [JOHN78]. Several older UIMSs were based on BNF specifications [HANA80; OLSE83; OLSE84a].

Transition networks, whether recursive or not, encode user-inteface state in a small number of state variables. *Augmented transition networks* (ATNs), a more flexible derivative of transition networks, encode user-interface state by which node of the ATN is active and by what the values of explicit state variables are. Responses can be the calling of action routines, the setting of these explicit state variables, or the changing of the node of the ATN that is active. Of course, the state variables can also be set by action routines. Figure 10.23 shows an ATN in which the Boolean state variable *cb*, set by several of the transitions, is used to affect flow of control from one state to another. The variable *cb* is **true** if the clipboard is full.

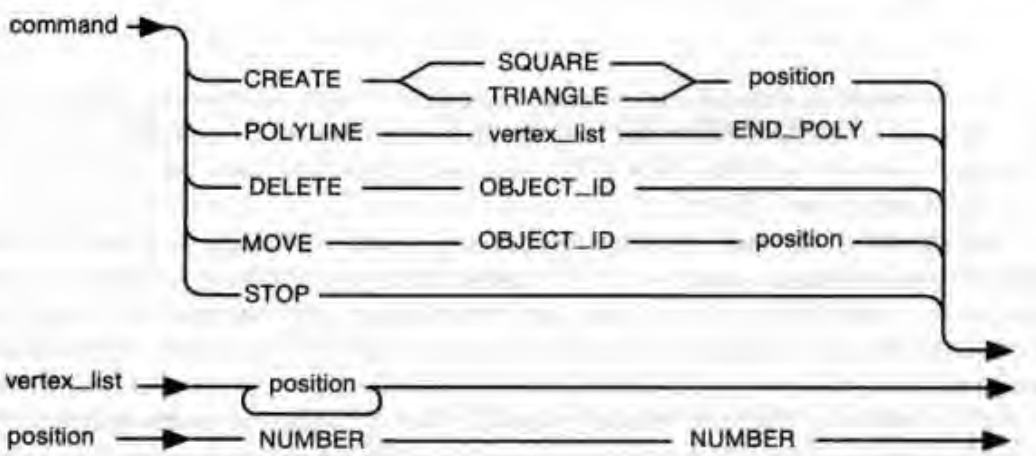


Fig. 10.22 A diagrammatic representation of the Backus–Naur form equivalent to that in Fig. 10.21.

Samsung/Google, EX1027, p. 475

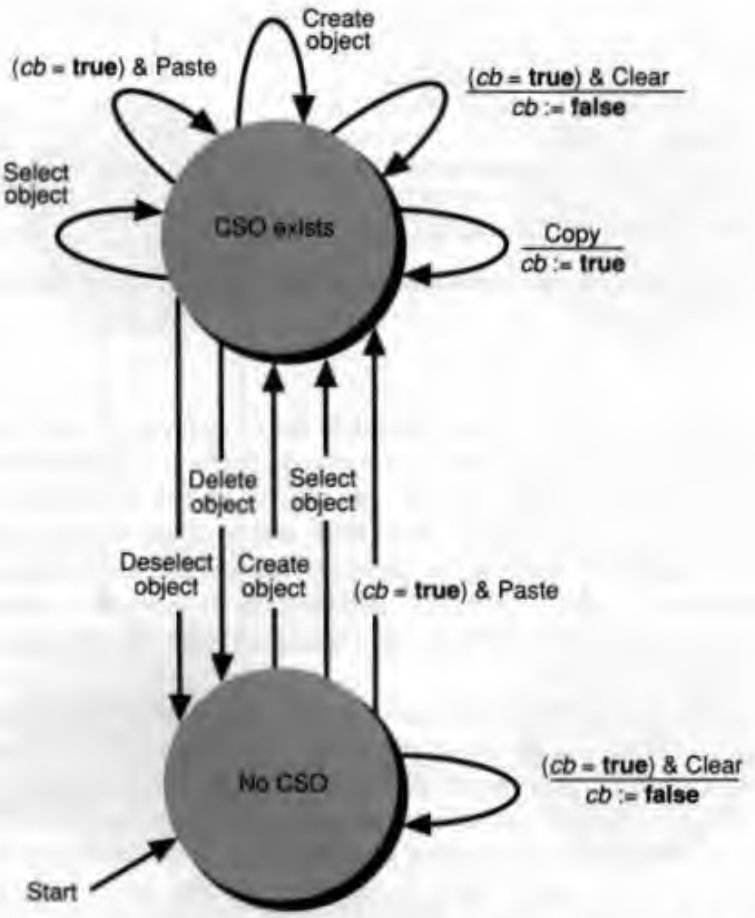


Fig. 10.23 An augmented transition network (ATN) representing the same user interface as that in Fig. 10.20. Transitions can be conditionally dependent on the value of explicit state variables (the Boolean *cb* in this case), and can also set state variables. In general, the application program can also set state variables.

Just as transition networks can be made more general with subnetworks, so too ATNs can call lower-level ATNs. ATNs that can recursively call other ATNs are called *augmented recursive transition networks* [WOOD70], and researchers have used these networks to model user interfaces [KIER85].

As transition networks become more complicated, with logical expressions on transition and subroutine calls, we are led toward more programlike specifications. After all, programming languages are the most powerful way yet developed of specifying sequencing and the multiple conditions often associated with transitions. Several *event languages* have been developed specifically for user-interface specification [CARD85; FLEC87; GARR82; GREE85a; HILL87; KASI82; SIOC89]. The user interface depicted in Figs. 10.20 and 10.23 can be described in a typical event language, as shown in Fig. 10.24. Note that event languages, unlike traditional programming languages, have no explicit flow of control. Instead, whenever an **if** condition becomes true, the associated actions are executed. Thus, the event language is a production-rule system.

Samsung/Google, EX1027, p. 476

```
if Event = SelectObject then
  begin
    cso := true
    perform action routine name
  end
if Event = DeselectCSO and cso = true then
  begin
    cso := false
    perform action routine name
  end
if Event = CreateObject then
  begin
    cso := true
    perform action routine name
  end
if Event = DeleteCSO and cso = true then
  begin
    cso := false
    perform action routine name
  end
if Event = CopyCSO and cso = true then
  begin
    cb := true
    perform action routine name
  end
if Event = PasteClipboard and cb = true then
  begin
    cso := true
    perform action routine name
  end
if Event = ClearClipboard and cb = true then
  begin
    cb := false
    perform action routine name
  end
```

Fig. 10.24 A typical event language, with a Pascal-like syntax.

Green [GREE87] surveys event languages and all the other sequence-specification methods we have mentioned, and shows that general event languages are more powerful than are transition networks, recursive transition networks, and grammars; he also provides algorithms for converting these forms into an event language. ATNs that have general computations associated with their arcs are also equivalent to event languages.

If event languges are so powerful, why do we bother with the various types of transition networks? Because, for simple cases, it is easier to work with diagrammatic representations. One of the goals of UIMSs is to allow nonprogrammers who specialize in user-interface design to be directly involved in creating an interface. This goal is probably best met with transition-network–oriented tools that are easier to use, although somewhat less powerful. Networks provide a useful, time-proven tool for laying out a dialogue, and

Samsung/Google, EX1027, p. 477

they appear to help the designer to document and understand the design. The diagrammatic representations are especially compelling if user actions are performed on interaction objects such as menus, dialogue boxes, and other visible objects. Then diagrams of the type shown in Fig. 10.25 can be created interactively to define dialogue sequencing. If needed, conditions (such as the *cb* = **true** in Fig. 10.23) can be associated with the arcs. Figure 10.26 shows one way to establish a link on such a diagram.

A quite different way to define syntax is by example. Here, the user-interface designer places the UIMS into a "learning" mode, and then steps through all acceptable sequences of actions (a tedious process in complex applications, unless the UIMS can infer general rules from the examples). The designer might start with a main menu, select an item from it, and then go through a directory to locate the submenu, dialogue box, or application-specific object to be presented to the user in response to the main menu selection. The object appears on the screen, and the designer can indicate the position, size, or other attributes that the object should have when the application is actually executed. The designer goes on to perform some operation on the displayed object and again shows what object should appear next, or how the displayed object is to respond to the operation; the designer repeats this process until all actions on all objects have been defined. This technique works for sequencing through items that have already been defined by the interface designer, but is not sufficiently general to handle arbitrary application functionality. User-interface software tools with some degree of by-example sequencing specification include Menulay [BUXT83], TAE Plus [MILL88c] and the SmethersBarnes Prototyper [COSS89]. Peridot, mentioned earlier, builds interaction techniques, (i.e., hardware bindings) by example.

10.6.2 Advanced UIMS Concepts

UIMSs have tended to focus on sequence control and visual design. Transition networks provide a good basis for sequencing, and interactive editors are just right for visual design.

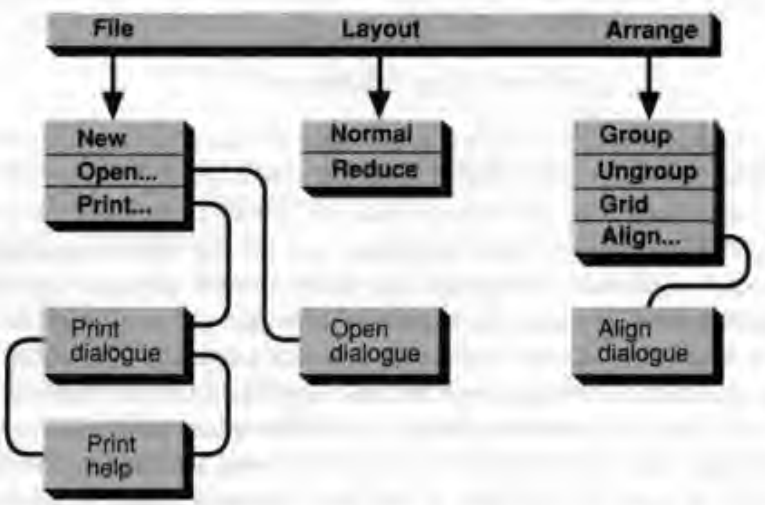


Fig. 10.25 Several menus and dialogue boxes linked together. The return paths from dialogue boxes to the main menu are not shown.

Samsung/Google, EX1027, p. 478

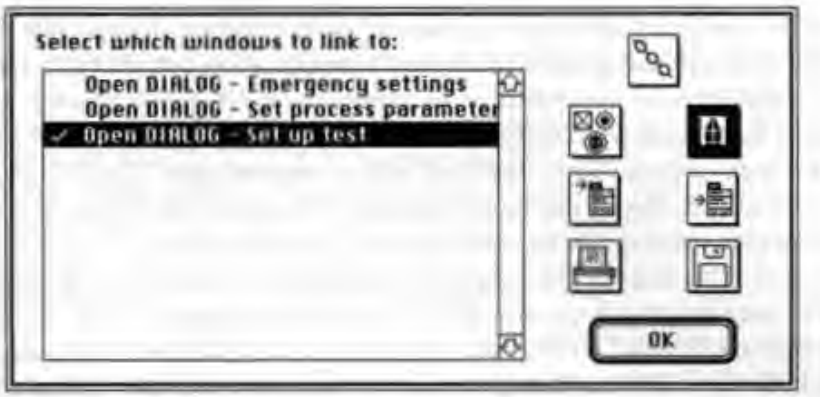


Fig. 10.26 Linking together different interaction techniques using the SmethersBarnes Prototyper. A menu item is being linked to the dialogue box "Set up test" checked in the list. The icons on the right are used to select the class of responses to be linked to the menu selection. Reading left to right underneath the "link" icon, the possibilities are enable or disable check boxes, radio buttons, and buttons in dialogue boxes; open a window or dialogue box (the class of response selected); enable or disable a menu; enable or disable a menu item; open a print dialogue box; or open a file dialogue box. (Courtesy of SmethersBarnes, Inc.)

As discussed in Chapter 9, however, a user-interface design includes conceptual, functional, sequencing, and hardware-binding levels. Much recent UIMS development has begun to address the functional and conceptual designs as well. Thus, there has come to be more focus on combining sequencing control with a higher-level model of objects and commands, and also on integrating intelligent help systems into the UIMS.

Representations at a higher level than that of transition networks are clearly needed. Consider how difficult it would be to add to the transition network of Fig. 10.20 new states to record whether any objects have yet been created. It would also be difficult to apply some of the dialogue modifications, such as CO, currently selected command, and factored attributes, discussed in Section 9.4. And the sequencing specifications provide no information about what operations can be performed on what objects, and certainly give no glimmer of whart parameters are needed to perform an operation.

The first step away from a sequencing orientation and toward higher levels of abstraction was taken by COUSIN [HAYE83; HAYE84], which automatically generates menus and dialogue boxes from a specification of commands, parameters, and parameter data types. The innovation of COUSIN is in defining all the parameters needed by a command as an integral unit. COUSIN has enough information that a prefix or postfix syntax could also be generated. Green took a similar approach, adding preconditions and postconditions to specify the semantics of user commands [GREE85b]. Olsen's MIKE system [OLSE86] declares commands and parameters, also generating a user interface in a fashion similar to COUSIN. In addition, MIKE supports direct manipulation of objects to specify positions, and can cause commands to be executed when a button-down event occurs in a window or subwindow.

Samsung/Google, EX1027, p. 479

All these significant advances are focused on commands. If a UIMS is to mediate between the user and the application in direct-manipulation interfaces, however, it must have some knowledge of the objects to be manipulated. HIGGINS was the first UIMS to incorporate a data model [HUDS86; HUDS87; HUDS88], one that is based on objects and relations between objects. The UIMS and action routines share the data model, so that changes made to data objects can be immediately reflected in the display. *Active values* are used to propogate changes among interdependent objects and from objects to their visual representations. The George Washington University User Interface Management System (GWUIMS) uses active values and object-oriented programming concepts to achieve a similar objective [SIBE86]. GWUIMS II also uses a data model [HURL89], as does the Serpent UIMS [BASS88]. Although the details vary, all the data models make use of object-oriented programming concepts and active values, and are closely related to developments in database-management systems in the area of semantic data models [HULL87].

The User Interface Design Environment (UIDE) project [FOLE89] has developed a new user-interface specification method integrating some elements of these recent developments to include a data model, the commands that can be applied to each type of object in the data model, the parameters needed by the commands, the parameter data types, the conditions under which commands are available for use (that is, command preconditions), and the changes that occur to state variables when a command is executed (that is, command postconditions) [FOLE87; FOLE88]. To illustrate the method, we start with the sample application developed in Section 10.6.1. We add a data model, which is here a single class of objects with two subclasses, square and triangle. In addition, there are two distinguished instances of objects, the CSO and the clipboard object, both of which may or may not exist at any given time. The specification is shown in Fig. 10.27. The preconditions are the conditions on state variables that must be satisfied for a command to be invoked, whereas the postconditions are changes in state variables.

Not only is this specification sufficient to create automatically an operational interface to the application's action routines, but also it is represented such that

- Menu items can be enabled and disabled, using preconditions
- Users can be told why a command is disabled, again using preconditions
- Users can be told what to do to enable a command, by back chaining to determine what commands must be invoked to satisfy the preconditions of the command in question
- Users can be given a partial explanation of what a command does, using the postconditions
- Some user-interface design-consistency rules can be checked
- Different interaction techniques can be assigned for use in specifying commands and command parameters
- Speed of use of the interface can be predicted for various task sequences and for various interaction techniques.

Another way to define user interfaces consisting of interconnected processing modules is with data-flow diagrams. For instance, the NeXT Interface Builder, shown in Fig. 10.28,

Samsung/Google, EX1027, p. 480

```
class object                                    {First the data model}
  subclasses triangle, square;
  actions CreateObject, SelectObject;
  attributes position range [0..10] x [0..10]   {Attribute name and data type}
class triangle, square;
  superclass object;
  inherits actions
  inherits attributes
instance CSO
  of object
  actions DeselectCSO, DeleteCSO, CopyCSO
  inherits attributes
instance CB
  of object
  actions ClearClipboard, Paste
  inherits attributes

{Initial values for state variables}
initial Number (object) := 0; csoExists := false; cbFull := false;

{Actions on objects, with preconditions, postconditions, and parameters}
precondition Number (object) ≠ 0;
SelectObject (object);
postcondition csoExists := true;

precondition csoExists := true;
DeselectCSO (CSO);
postcondition csoExists := false;

precondition;
CreateObject (position, object);
postcondition Number (object) := Number (object) +1; csoExists := true;

precondition csoExists := true;
DeleteCSO (CSO);
postcondition Number (object) = Number (object) - 1; csoExists := false;

precondition csoExists := true;
CopyCSO (CSO);
postcondition cbFull := true;

precondition cbFull := true;
Paste (CB);
postcondition csoExists := true;

precondition cbFull := true;
ClearClipboard (CB);
postcondition cbFull := false;
```

Fig. 10.27 A high-level specification of a user interface incorporating a data model, sequencing information, and command parameters.

Samsung/Google, EX1027, p. 481

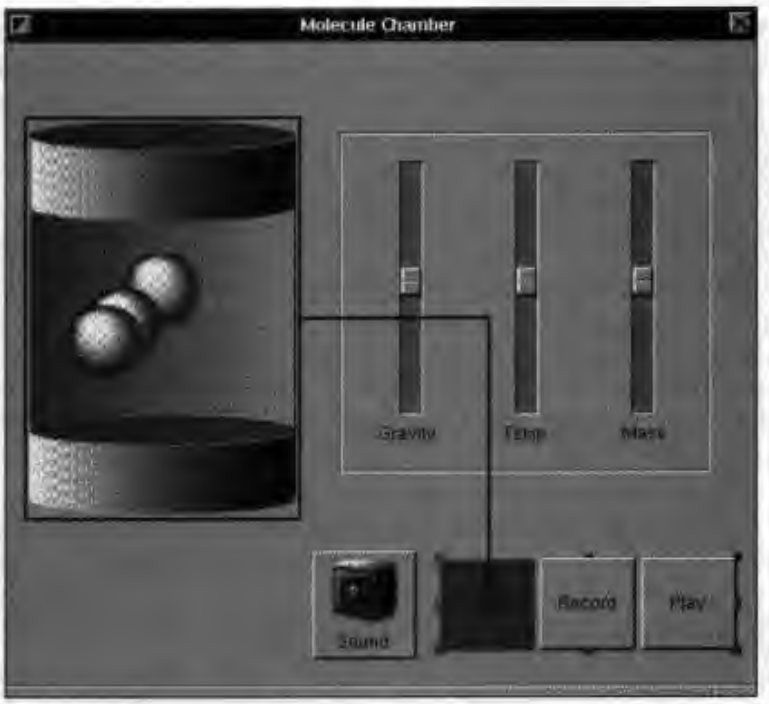


Fig. 10.28 The NeXT Interface Builder, showing a connection being made. The user has already selected the type of message to be sent from the Stop button, and has drawn the connection to the cylindrical chamber to indicate the destination of the message. (Courtesy of NeXT, Inc. © 1989 NeXT, Inc.)

allows objects to be interconected so that output messages from one object are input to another object. Type checking is used to ensure that only compatible messages are sent and received.

Data-flow diagrams can also be used to specify the detailed behavior of some or all of a user interface, although doing so takes on considerable programming flavor and suffers the same problems of scale seen with flowcharts and transition networks. Work is this area is surveyed in [BORN86a]; a more recent project is described in [SMIT88]. A specialized system, for scientific data visualization, is shown in Color Plate I.30.

UIMSs are finding their way into regular use. Early UIMSs suffered from rigid interaction styles that did not allow custom-tailoring to suit users' needs and were overly dependent on transition networks. Commercial UIMSs are now used on a large scale, and are becoming as essential to developing interactive graphics application programs as are graphics subroutine packages, window managers, and interaction-technique toolkits.

EXERCISES

10.1 Study the user interfaces to two different window systems. Categorize each with respect to the design issues discussed in Section 9.3.

10.2 Devise the search mechanism for real-estate–based event routing with overlapping main windows, in which each main window can contain a hierarchy of spatially nested subwindows.

Samsung/Google, EX1027, p. 482

10.3 Survey three window-management systems to determine whether they
   a. Have hierarchical windows
   b. Implement a server–client model and, if they do, whether the implementation allows the server and client to be distributed in a network
   c. Provide real-estate or listener input event dispatching, or some combination thereof
   d. Integrate the graphics subroutine package with the window-management system, or pass graphics calls directly through to graphics hardware.

10.4 Write an interactive dialogue box or menu editor.

10.5 Implement the concepts demonstrated in MICKY [OLSE89] with a programming language and toolkit available to you.

10.6 Examine several user interfaces with which you are familiar. Identify a set of user-interface state variables used in implementing each user interface. How many of these state variables are used in the user interface to provide context-sensitive menus, help, and so on?

10.7 Document the dialogue of a user interface to a paint or drawing program. Do this (a) with state diagrams, and (b) with the specialized language introduced in Section 10.6.1. Which method did you find easier? Why? Compare your opinions with those of your classmates. Which method is easier for answering questions such as "How do I draw a circle?" "What do I do after an error message appears?"

10.8 Write a transition-network–based UIMS. Every transition in the state diagram should be represented by a state-table entry with the following information:

- Current state number
- Next state
- Event which causes the transition
- Name of procedure to call when the transition occurs.

Events should include selection of a command from a menu, typing of a command name, mouse movement, mouse button-down, and mouse button-up. You should automatically display a menu containing all possible commands (derive this list from the events in the state table), enabling only those choices available from the current state.

10.9 For each of the extensions to state diagrams discussed in Section 9.3, determine whether the modifications create a push-down automaton (is it bounded or unbounded?) or a Turing machine.

10.10 Carefully study a window-management system that includes a policy-free window system. Examine several window managers to determine whether they can be implemented with the window system. For instance, some window systems provide for borders around windows for scroll bars, a heading, and perhaps selection buttons. For the window system to be completely policy-free, you must be able to specify separately the width of the borders on the four sides of the window.

Samsung/Google, EX1027, p. 483

11
Representing Curves and Surfaces

Smooth curves and surfaces must be generated in many computer graphics applications. Many real-world objects are inherently smooth, and much of computer graphics involves modeling the real world. Computer-aided design (CAD), high-quality character fonts, data plots, and artists' sketches all contain smooth curves and surfaces. The path of a camera or object in an animation sequence (Chapter 21) is almost always smooth; similarly, a path through intensity or color space (Chapters 16 and 13) must often be smooth.

The need to represent curves and surfaces arises in two cases: in modeling existing objects (a car, a face, a mountain) and in modeling "from scratch," where no preexisting physical object is being represented. In the first case, a mathematical description of the object may be unavailable. Of course, one can use as a model the coordinates of the infinitely many points of the object, but this is not feasible for a computer with finite storage. More often, we merely approximate the object with pieces of planes, spheres, or other shapes that are easy to describe mathematically, and require that points on our model be close to corresponding points on the object.

In the second case, when there is no preexisting object to model, the user creates the object in the modeling process; hence, the object matches its representation exactly, because its only embodiment is the representation. To create the object, the user may sculpt the object interactively, describe it mathematically, or give an approximate description to be "filled in" by some program. In CAD, the computer representation is used later to generate physical realizations of the abstractly designed object.

This chapter introduces the general area of *surface modeling*. The area is quite broad, and only the three most common representations for 3D surfaces are presented here: polygon mesh surfaces, parametric surfaces, and quadric surfaces. We also discuss

Samsung/Google, EX1027, p. 484



Fig. 11.1 A 3D object represented by polygons.

parametric curves, both because they are interesting in their own right and because parametric surfaces are a simple generalization of the curves.

Solid modeling, introduced in the next chapter, is the representation of volumes completely surrounded by surfaces, such as a cube, an airplane, or a building. The surface representations discussed in this chapter can be used in solid modeling to define each of the surfaces that bound the volume.

A *polygon mesh* is a set of connected polygonally bounded planar surfaces. Open boxes, cabinets, and building exteriors can be easily and naturally represented by polygon meshes, as can volumes bounded by planar surfaces. Polygon meshes can be used, although less easily, to represent objects with curved surfaces, as in Fig. 11.1; however, the representation is only approximate. Figure 11.2 shows the cross-section of a curved shape and the polygon mesh representing it. The obvious errors in the representation can be made arbitrarily small by using more and more polygons to create a better piecewise linear approximation, but this increases space requirements and the execution time of algorithms processing the representation. Furthermore, if the image is enlarged, the straight edges again become obvious. (Forrest calls this problem "geometric aliasing" [FORR80], by analogy to the general notion of aliasing discussed in Chapters 3 and 14.)

Parametric polynomial curves define points on a 3D curve by using three polynomials in a parameter t, one for each of x, y, and z. The coefficients of the polynomials are selected such that the curve follows the desired path. Although various degrees of polynomials can be used, we present only the most common case, cubic polynomials (that have powers of the parameter up through the third). The term *cubic curve* will often be used for such curves.

Parametric bivariate (two-variable) *polynomial surface patches* define the coordinates of points on a curved surface by using three bivariate polynomials, one for each of x, y, and z. The boundaries of the patches are parametric polynomial curves. Many fewer bivariate polynomial surface patches than polygonal patches are needed to approximate a curved



Fig. 11.2 A cross-section of a curved object and its polygonal representation.

Samsung/Google, EX1027, p. 485

surface to a given accuracy. The algorithms for working with bivariate polynomials, however, are more complex than are those for polygons. As with curves, polynomials of various degrees can be used, but we discuss here only the common case of polynomials that are cubic in both parameters. The surfaces are accordingly called *bicubic surfaces*.

Quadric surfaces are those defined implicitly by an equation $f(x, y, z) = 0$, where f is a quadric polynomial in x, y, and z. Quadric surfaces are a convenient representation for the familiar sphere, ellipsoid, and cylinder.

The next chapter, on solid modeling, incorporates these representations into systems to represent not just surfaces, but also bounded (solid) volumes. The surface representations described in this chapter are used, sometimes in combination with one another, to bound a 3D volume.

11.1 POLYGON MESHES

A *polygon mesh* is a collection of edges, vertices, and polygons connected such that each edge is shared by at most two polygons. An edge connects two vertices, and a polygon is a closed sequence of edges. An edge can be shared by two adjacent polygons, and a vertex is shared by at least two edges. A polygon mesh can be represented in several different ways, each with its advantages and disadvantages. The application programmer's task is to choose the most appropiate representation. Several representations can be used in a single application: one for external storage, another for internal use, and yet another with which the user interactively creates the mesh.

Two basic criteria, space and time, can be used to evaluate different representations. Typical operations on a polygon mesh are finding all the edges incident to a vertex, finding the polygons sharing an edge or a vertex, finding the vertices connected by an edge, finding the edges of a polygon, displaying the mesh, and identifying errors in representation (e.g., a missing edge, vertex, or polygon). In general, the more explicitly the relations among polygons, vertices, and edges are represented, the faster the operations are and the more space the representation requires. Woo [WOO85] has analyzed the time complexity of nine basic access operations and nine basic update operations on a polygon-mesh data structure.

In the rest of this section, several issues concerning polygon meshes are discussed: representing polygon meshes, ensuring that a given representation is correct, and calculating the coefficients of the plane of a polygon.

11.1.1 Representing Polygon Meshes

In this section, we discuss three polygon-mesh representations: explicit, pointers to a vertex list, and pointers to an edge list. In the *explicit* representation, each polygon is represented by a list of vertex coordinates:

$$P = ((x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_n, y_n, z_n)).$$

The vertices are stored in the order in which they would be encountered traveling around the polygon. There are edges between successive vertices in the list and between the last and first vertices. For a single polygon, this is space-efficient; for a polygon mesh, however,

Samsung/Google, EX1027, p. 486

much space is lost because the coordinates of shared vertices are duplicated. Even worse, there is no explicit representation of shared edges and vertices. For instance, to drag a vertex and all its incident edges interactively, we must find all polygons that share the vertex. This requires comparing the coordinate triples of one polygon with those of all other polygons. The most efficient way to do this would be to sort all N coordinate triples, but this is at best an $N\log_2 N$ process, and even then there is the danger that the same vertex might, due to computational roundoff, have slightly different coordinate values in each polygon, so a correct match might never be made.

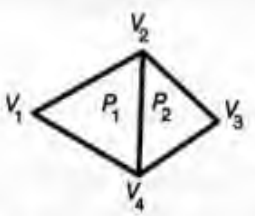
With this representation, displaying the mesh either as filled polygons or as polygon outlines necessitates transforming each vertex and clipping each edge of each polygon. If edges are being drawn, each shared edge is drawn twice; this causes problems on pen plotters, film recorders, and vector displays due to the overwriting. A problem may also be created on raster displays if the edges are drawn in opposite directions, in which case extra pixels may be intensified.

Polygons defined with *pointers to a vertex list*, the method used by SPHIGS, have each vertex in the polygon mesh stored just once, in the vertex list $V = ((x_1, y_1, z_1), \ldots, (x_n, y_n, z_n))$. A polygon is defined by a list of indices (or pointers) into the vertex list. A polygon made up of vertices 3, 5, 7, and 10 in the vertex list would thus be represented as $P = (3, 5, 7, 10)$.

This representation, an example of which is shown in Fig. 11.3, has several advantages over the explicit polygon representation. Since each vertex is stored just once, considerable space is saved. Furthermore, the coordinates of a vertex can be changed easily. On the other hand, it is still difficult to find polygons that share an edge, and shared polygon edges are still drawn twice when all polygon outlines are displayed. These two problems can be eliminated by representing edges explicitly, as in the next method.

When defining polygons by *pointers to an edge list*, we again have the vertex list V, but represent a polygon as a list of pointers not to the vertex list, but rather to an edge list, in which each edge occurs just once. In turn, each edge in the edge list points to the two vertices in the vertex list defining the edge, and also to the one or two polygons to which the edge belongs. Hence, we describe a polygon as $P = (E_1, \ldots, E_n)$, and an edge as $E = (V_1, V_2, P_1, P_2)$. When an edge belongs to only one polygon, either P_1 or P_2 is null. Figure 11.4 shows an example of this representation.

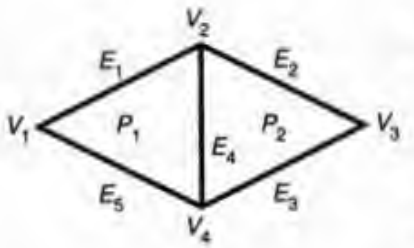
Polygon outlines are shown by displaying all edges, rather than by displaying all polygons; thus, redundant clipping, transformation, and scan conversion are avoided. Filled polygons are also displayed easily. In some situations, such as the description of a 3D honeycomblike sheet-metal structure, some edges are shared by three polygons. In such



$V = (V_1, V_2, V_3, V_4) = ((x_1, y_1, z_1), \ldots, (x_4, y_4, z_4))$
$P_1 = (1, 2, 4)$
$P_2 = (4, 2, 3)$

Fig. 11.3 Polygon mesh defined with indexes into a vertex list.

Samsung/Google, EX1027, p. 487



$V = (V_1, V_2, V_3, V_4) = ((x_1, y_1, z_1), \ldots, (x_4, y_4, z_4))$
$E_1 = (V_1, V_2, P_1, \lambda)$
$E_2 = (V_2, V_3, P_2, \lambda)$
$E_3 = (V_3, V_4, P_2, \lambda)$
$E_4 = (V_4, V_2, P_1, P_2)$
$E_5 = (V_4, V_1, P_1, \lambda)$
$P_1 = (E_1, E_4, E_5)$
$P_2 = (E_2, E_3, E_4)$

Fig. 11.4 Polygon mesh defined with edge lists for each polygon (λ represents null).

cases, the edge descriptions can be extended to include an arbitrary number of polygons: $E = (V_1, V_2, P_1, P_2, \ldots, P_n)$.

In none of these three representations (i.e., explicit polygons, pointers to vertices, pointers to an edge list), is it easy to determine which edges are incident to a vertex: All edges must be inspected. Of course, information can be added explicitly to permit determining such relationships. For instance, the winged-edge representation used by Baumgart [BAUM75] expands the edge description to include pointers to the two adjoining edges of each polygon, whereas the vertex description includes a pointer to an (arbitrary) edge incident on the vertex, and thus more polygon and vertex information is available. This representation is discussed in Chapter 12.

11.1.2 Consistency of Polygon-Mesh Representations

Polygon meshes are often generated interactively, such as by operators digitizing drawings, so errors are inevitable. Thus, it is appropriate to make sure that all polygons are closed, all edges are used at least once but not more than some (application-defined) maximum, and each vertex is referenced by at least two edges. In some applications, we would also expect the mesh to be completely connected (any vertex can be reached from any other vertex by moving along edges), to be topologically planar (the binary relation on vertices defined by edges can be represented by a planar graph), or to have no holes (there exists just one boundary—a connected sequence of edges each of which is used by one polygon).

Of the three representations discussed, the explicit-edge scheme is the easiest to check for consistency, because it contains the most information. For example, to make sure that all edges are part of at least one but no more than some maximum number of polygons, the code in Fig. 11.5 can be used.

This procedure is by no means a complete consistency check. For example, an edge used twice in the same polygon goes undetected. A similar procedure can be used to make sure that each vertex is part of at least one polygon; we check whether at least two different edges of the same polygon refer to the vertex. Also, it should be an error for the two vertices of an edge to be the same, unless edges with zero length are allowed.

The relationship of "sharing an edge" between polygons is a binary equivalence relation and hence partitions a mesh into equivalence classes called *connected components*.

Samsung/Google, EX1027, p. 488

```
for (each edge E_j in set of edges)
    use_count_j = 0;
for (each polygon P_i in set of polygons)
    for (each edge E_j of polygon P)
        use_count_j++;
for (each edge E_j in set of edges) {
    if (use_count_j == 0)
        Error ();
    if (use_count_j > maximum)
        Error ();
}
```

Fig. 11.5 Code to ensure that all edges of explicit polygon representation are used between 1 and *maximum* times.

One usually expects a polygon mesh to have a single connected component. Algorithms for determining the connected components of a binary relation are well known [SEDG88].

More detailed testing is also possible; one can check, for instance, that each polygon referred to by an edge E_i refers in turn back to the edge E_i. This ensures that all references from polygons to edges are complete. Similarly, we can check that each edge E_i referred to by a polygon P_i also refers back to polygon P_i, which ensures that the references from edges to polygons are complete.

11.1.3 Plane Equations

When working with polygons or polygon meshes, we frequently need to know the equation of the plane in which the polygon lies. In some cases, of course, the equation is known implicitly through the interactive construction methods used to define the polygon. If it is not known, we can use the coordinates of three vertices to find the plane. Recall the plane equation

$$Ax + By + Cz + D = 0. \tag{11.1}$$

The coefficients A, B, and C define the normal to the plane, $[A \quad B \quad C]$. Given points P_1, P_2, and P_3 on the plane, that plane's normal can be computed as the vector cross-product $P_1P_2 \times P_1P_3$ (or $P_2P_3 \times P_2P_1$, etc.). If the cross-product is zero, then the three points are collinear and do not define a plane. Other vertices, if any, can be used instead. Given a nonzero cross-product, D can be found by substituting the normal $[A \quad B \quad C]$ and any one of the three points into Eq. (11.1).

If there are more than three vertices, they may be nonplanar, either for numerical reasons or because of the method by which the polygons were generated. Then another technique for finding the coefficients A, B, and C of a plane that comes close to all the vertices is better. It can be shown that A, B, and C are proportional to the signed areas of the projections of the polygon onto the (y, z), (x, z), and (x, y) planes, respectively. For example, if the polygon is parallel to the (x, y) plane, then $A = B = 0$, as expected: The

Samsung/Google, EX1027, p. 489

projections of the polygon onto the (y, z) and (x, z) planes have zero area. This method is better because the areas of the projections are a function of the coordinates of all the vertices and so are not sensitive to the choice of a few vertices that might happen not to be coplanar with most or all of the other vertices, or that might happen to be collinear. For instance, the area (and hence coefficient) C of the polygon projected onto the (x, y) plane in Fig. 11.6 is just the area of the trapezoid A_3, minus the areas of A_1 and A_2. In general,

$$C = \frac{1}{2}\sum_{i=1}^{n}(y_i + y_{i\oplus 1})(x_{i\oplus 1} - x_i), \tag{11.2}$$

where the operator $\oplus$ is normal addition except that $n \oplus 1 = 1$. The areas for A and B are given by similar formulae, except the area for B is negated (see Exercise 11.1).

Eq. (11.2) gives the sum of the areas of all the trapezoids formed by successive edges of the polygons. If $x_{i\oplus 1} < x_i$, the area makes a negative contribution to the sum. The sign of the sum is also useful: if the vertices have been enumerated in a clockwise direction (as projected onto the plane), then the sign is positive; otherwise, it is negative.

Once we determine the plane equation by using all the vertices, we can estimate how nonplanar the polygon is by calculating the perpendicular distance from the plane to each vertex. This distance d for the vertex at (x, y, z) is

$$d = \frac{Ax + By + Cz + D}{\sqrt{A^2 + B^2 + C^2}}. \tag{11.3}$$

This distance is either positive or negative, depending on which side of the plane the point is located. If the vertex is on the plane, then $d = 0$. Of course, to determine only on which side of a plane a point is, only the sign of d matters, so division by the square root is not needed.

The plane equation is not unique; any nonzero multiplicative constant k changes the equation, but not the plane. It is often convenient to store the plane coefficients with a normalized normal; this can be done by choosing

$$k = \frac{1}{\sqrt{A^2 + B^2 + C^2}}, \tag{11.4}$$

which is the reciprocal of the length of the normal. Then, distances can be computed with Eq. (11.3) more easily, since the denominator is 1.

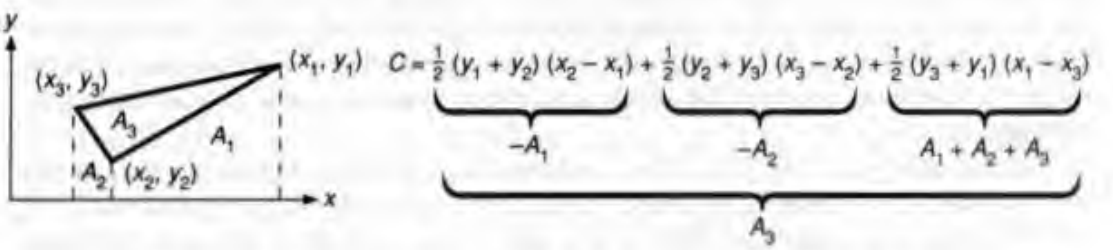


Fig. 11.6 Calculating the area C of a triangle using Eq. (11.2).

Samsung/Google, EX1027, p. 490

11.2 PARAMETRIC CUBIC CURVES

Polylines and polygons are first-degree, piecewise linear approximations to curves and surfaces, respectively. Unless the curves or surfaces being approximated are also piecewise linear, large numbers of endpoint coordinates must be created and stored to achieve reasonable accuracy. Interactive manipulation of the data to approximate a shape is tedious, because many points have to be positioned precisely.

In this section, a more compact and more manipulable representation of piecewise smooth curves is developed; in the following section, the mathematical development is generalized to surfaces. The general approach is to use functions that are of a higher degree than are the linear functions. The functions still generally only approximate the desired shapes, but use less storage and offer easier interactive manipulation than do linear functions.

The higher-degree approximations can be based on one of three methods. First, one can express y and z as *explicit* functions of x, so that $y = f(x)$ and $z = g(x)$. The difficulties with this are that (1) it is impossible to get multiple values of y for a single x, so curves such as circles and ellipses must be represented by multiple curve segments; (2) such a definition is not rotationally invariant (to describe a rotated version of the curve requires a great deal of work and may in general require breaking a curve segment into many others); and (3) describing curves with vertical tangents is difficult, because a slope of infinity is difficult to represent.

Second, we can choose to model curves as solutions to *implicit* equations of the form $f(x, y, z) = 0$; this is fraught with its own perils. First, the given equation may have more solutions than we want. For example, in modeling a circle, we might want to use $x^2 + y^2 = 1$, which is fine. But how do we model a half circle? We must add constraints such as $x \geq 0$, which cannot be contained within the implicit equation. Furthermore, if two implicitly defined curve segments are joined together, it may be difficult to determine whether their tangent directions agree at their join point. Tangent continuity is critical in many applications.

These two mathematical forms do permit rapid determination of whether a point lies on the curve or on which side of the curve the point lies, as was done in Chapter 3. Normals to the curve are also easily computed. Hence, we shall briefly discuss the implicit form in Section 11.4.

The *parametric representation* for curves, $x = x(t)$, $y = y(t)$, $z = z(t)$ overcomes the problems caused by functional or implicit forms and offers a variety of other attractions that will become clear in the remainder of this chapter. Parametric curves replace the use of geometric slopes (which may be infinite) with parametric tangent vectors (which, we shall see, are never infinite). Here a curve is approximated by a *piecewise polynomial* curve instead of the piecewise linear curve used in the preceding section. Each segment Q of the overall curve is given by three functions, x, y, and z, which are cubic polynomials in the parameter t.

Cubic polynomials are most often used because lower-degree polynomials give too little flexibility in controlling the shape of the curve, and higher-degree polynomials can introduce unwanted wiggles and also require more computation. No lower-degree representation allows a curve segment to interpolate (pass through) two specified endpoints with specified derivatives at each endpoint. Given a cubic polynomial with its four coefficients,

Samsung/Google, EX1027, p. 491

four knowns are used to solve for the unknown coefficients. The four knowns might be the two endpoints and the derivatives at the endpoints. Similarly, the two coefficients of a first-order (straight-line) polynomial are determined by the two endpoints. For a straight line, the derivatives at each end are determined by the line itself and cannot be controlled independently. With quadratic (second-degree) polynomials, and hence three coefficients, two endpoints and one other condition, such as a slope or additional point, can be specified.

Also, parametric cubics are the lowest-degree curves that are nonplanar in 3D. You can see this by recognizing that a second-order polynomial's three coefficients can be completely specified by three points and that three points define a plane in which the polynomial lies.

Higher-degree curves require more conditions to determine the coefficients and can "wiggle" back and forth in ways that are difficult to control. Despite this, higher-degree curves are used in applications—such as the design of cars and planes—in which higher-degree derivatives must be controlled to create surfaces that are aerodynamically efficient. In fact, the mathematical development for parametric curves and surfaces is often given in terms of an arbitrary degree n. In this chapter, we fix n at 3.

The cubic polynomials that define a curve segment $Q(t) = [x(t)\ y(t)\ z(t)]$ are of the form

$$\begin{aligned} x(t) &= a_x t^3 + b_x t^2 + c_x t + d_x, \\ y(t) &= a_y t^3 + b_y t^2 + c_y t + d_y, \\ z(t) &= a_z t^3 + b_z t^2 + c_z t + d_z, \qquad 0 \le t \le 1. \end{aligned} \tag{11.5}$$

To deal with finite segments of the curve, without loss of generality, we restrict the parameter t to the [0, 1] interval.

With $T = [t^3 \quad t^2 \quad t \quad 1]$, and defining the matrix of coefficients of the three polynomials as

$$C = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix}, \tag{11.6}$$

we can rewrite Eq. (11.5) as

$$Q(t) = [x(t) \quad y(t) \quad z(t)] = T \cdot C. \tag{11.7}$$

This provides a compact way to express the Eq. (11.5).

Figure 11.7 shows two joined parametric cubic curve segments and their polynomials; it also illustrates the ability of parametrics to represent easily multiple values of y for a single value of x with polynomials that are themselves single valued. (This figure of a curve, like all others in this section, shows 2D curves represented by $[x(t) \quad y(t)]$.)

The derivative of $Q(t)$ is the parametric *tangent vector* of the curve. Applying this definition to Eq. (11.7), we have

$$\begin{aligned} \frac{d}{dt}Q(t) = Q'(t) &= \left[\frac{d}{dt}x(t) \quad \frac{d}{dt}y(t) \quad \frac{d}{dt}z(t)\right] = \frac{d}{dt}T \cdot C = [3t^2 \quad 2t \quad 1 \quad 0] \cdot C \\ &= [3a_x t^2 + 2b_x t + c_x \quad 3a_y t^2 + 2b_y t + c_y \quad 3a_z t^2 + 2b_z t + c_z]. \end{aligned} \tag{11.8}$$

Samsung/Google, EX1027, p. 492

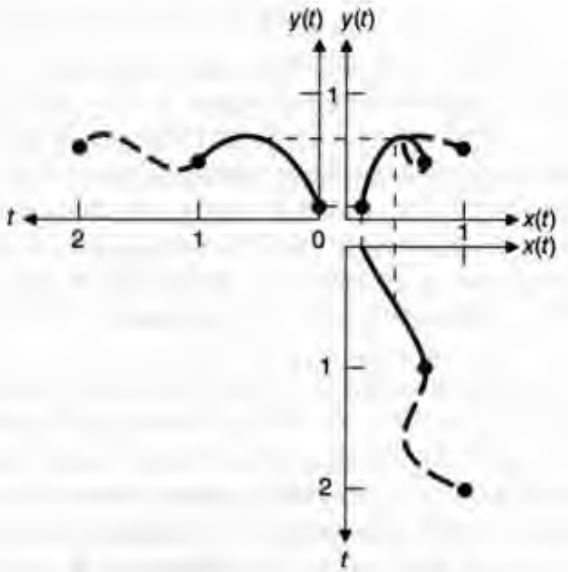


Fig. 11.7 Two joined 2D parametric curve segments and their defining polynomials. The dashed lines between the (x,y) plot and the $x(t)$ and $y(t)$ plots show the correspondence between the points on the (x,y) curve and the defining cubic polynomials. The $x(t)$ and $y(t)$ plots for the second segment have been translated to begin at $t = 1$, rather than at $t = 0$, to show the continuity of the curves at their join point.

If two curve segments join together, the curve has G^0 *geometric continuity*. If the directions (but not necessarily the magnitudes) of the two segments' tangent vectors are equal at a join point, the curve has G^1 geometric continuity. In computer-aided design of objects, G^1 continuity between curve segments is often required. G^1 continuity means that the geometric slopes of the segments are equal at the join point. For two tangent vectors TV_1 and TV_2 to have the same direction, it is necessary that one be a scalar multiple of the other: $TV_1 = k \cdot TV_2$, with $k > 0$ [BARS88].

If the tangent vectors of two cubic curve segments are equal (i. e., their directions *and* magnitudes are equal) at the segments' join point, the curve has first-degree continuity in the parameter t, or *parametric continuity*, and is said to be C^1 continuous. If the direction and magnitude of $d^n/dt^n[Q(t)]$ through the nth derivative are equal at the join point, the curve is called C^n *continuous*. Figure 11.8 shows curves with three different degrees of continuity. Note that a parametric curve segment is itself everywhere continuous; the continuity of concern here is at the join points.

The tangent vector $Q'(t)$ is the velocity of a point on the curve with respect to the parameter t. Similarly, the second derivative of $Q(t)$ is the acceleration. If a camera is moving along a parametric cubic curve in equal time steps and records a picture after each step, the tangent vector gives the velocity of the camera along the curve. The camera velocity and acceleration at join points should be continuous, to avoid jerky movements in the resulting animation sequence. It is this continuity of acceleration across the join point in Fig. 11.8 that makes the C^2 curve continue farther to the right than the C^1 curve, before bending around to the endpoint.

Samsung/Google, EX1027, p. 493

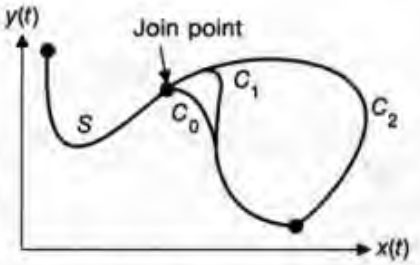


Fig. 11.8 Curve segment S joined to segments C_0, C_1, and C_2 with the 0, 1, and 2 degrees of parametric continuity, respectively. The visual distinction between C_1 and C_2 is slight at the join, but obvious away from the join.

In general, C^1 continuity implies G^1, but the converse is generally not true. That is, G^1 continuity is generally less restrictive than is C^1, so curves can be G^1 but not necessarily C^1. However, join points with G^1 continuity will appear just as smooth as those with C^1 continuity, as seen in Fig. 11.9.

There is a special case in which C^1 continuity does *not* imply G^1 continuity: When both segments' tangent vectors are [0 0 0] at the join point. In this case, the tangent vectors are indeed equal, but their directions can be different (Fig. 11.10). Figure 11.11 shows this concept in another way. Think again of a camera moving along the curve; the camera velocity slows down to zero at the join point, the camera changes direction while its velocity is zero, and the camera accelerates in the new direction.

The plot of a parametric curve is distinctly different from the plot of an ordinary function, in which the independent variable is plotted on the x axis and the dependent variable is plotted on the y axis. In parametric curve plots, the independent variable t is

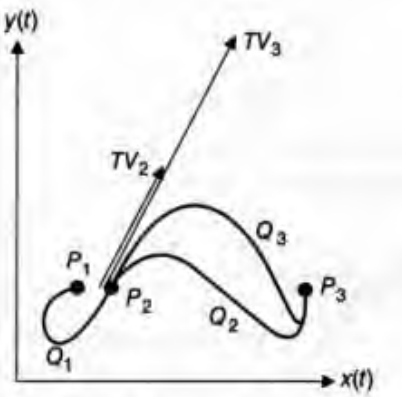


Fig. 11.9 Curve segments Q_1, Q_2, and Q_3 join at the point P_2 and are identical except for their tangent vectors at P_2. Q_1 and Q_2 have equal tangent vectors, and hence are both G^1 and C^1 continuous at P_2. Q_1 and Q_3 have tangent vectors in the same direction, but Q_3 has twice the magnitude, so they are only G^1 continuous at P_2. The larger tangent vector of Q_3 means that the curve is pulled more in the tangent-vector direction before heading toward P_3. Vector TV_2 is the tangent vector for Q_2, TV_3 is that for Q_3.

Samsung/Google, EX1027, p. 494

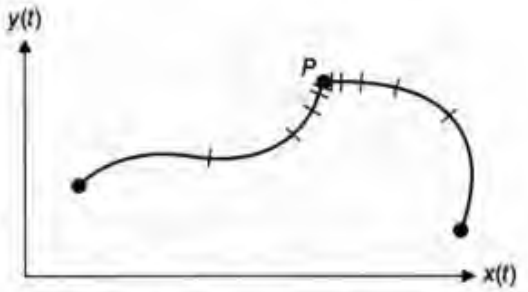


Fig. 11.10 The one case for which C^1 continuity does not imply G^1 continuity: the tangent vector (i.e., the parametric velocity along the curve) is zero at the join joint P. Each tick mark shows the distance moved along the curve in equal time intervals. As the curve approaches P, the velocity goes to zero, then increases past P.

never plotted at all. This means that we cannot determine, just by looking at a parametric curve plot, the tangent vector to the curve. It is possible to determine the direction of the vector, but not its magnitude. This can be seen as follows: if $\gamma(t)$, $0 \le t \le 1$ is a parametric curve, its tangent vector at time 0 is $\gamma'(0)$. If we let $\eta(t) = \gamma(2t)$, $0 \le t \le \frac{1}{2}$, then the parametric plots of γ and η are identical. On the other hand, $\eta'(0) = 2\,\gamma'(0)$. Thus, two curves that have identical plots can have different tangent vectors. This is the motivation for the definition of geometric continuity: For two curves to join smoothly, we require only that their tangent-vector directions match, not that their magnitudes match.

A curve segment $Q(t)$ is defined by constraints on endpoints, tangent vectors, and continuity between curve segments. Each cubic polynomial of Eq. (11.5) has four coefficients, so four constraints will be needed, allowing us to formulate four equations in the four unknowns, then solving for the unknowns. The three major types of curves discussed in this section are *Hermite*, defined by two endpoints and two endpoint tangent vectors; *Bézier*, defined by two endpoints and two other points that control the endpoint tangent vectors; and several kinds of *splines*, each defined by four control points. The

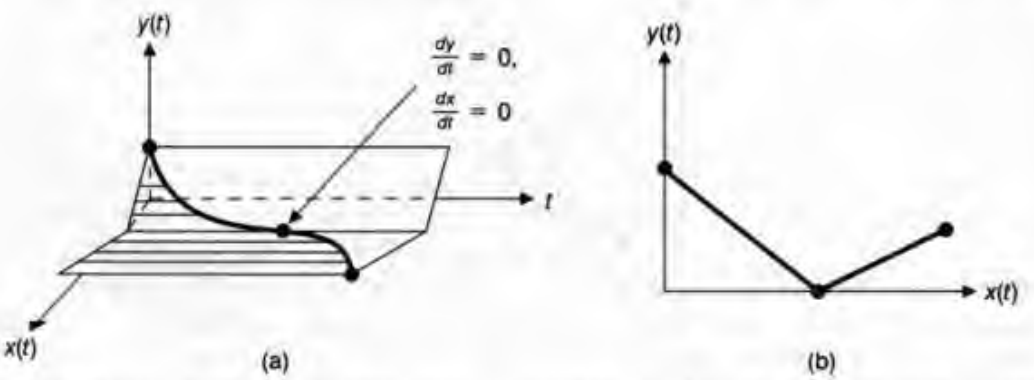


Fig. 11.11 (a) View of a 2D parametric cubic curve in 3D (x, y, t) space, and (b) the curve in 2D. At the join, the velocity of both parametrics is zero; that is, $dy/dt = 0$ and $dx/dt = 0$. You can see this by noting that, at the join, the curve is parallel to the t axis, so there is no change in either x or y. Yet at the join point, the parametrics are C^1 continuous, but are not G^1 continuous.

Samsung/Google, EX1027, p. 495

splines have C^1 and C^2 continuity at the join points and come close to their control points, but generally do not interpolate the points. The types of splines are uniform B-splines, nonuniform B-splines, and β-splines.

To see how the coefficients of Eq. (11.5) can depend on four constraints, we recall that a parametric cubic curve is defined by $Q(t) = T \cdot C$. We rewrite the coefficient matrix as $C = M \cdot G$, where M is a 4×4 *basis matrix*, and G is a four-element column vector of geometric constraints, called the *geometry vector*. The geometric constraints are just the conditions, such as endpoints or tangent vectors, that define the curve. We use G_x to refer to the column vector of just the x components of the geometry vector. G_y and G_z have similar definitions. M or G, or both M and G, differ for each type of curve.

The elements of M and G are constants, so the product $T \cdot M \cdot G$ is just three cubic polynomials in t. Expanding the product $Q(t) = T \cdot M \cdot G$ gives

$$Q(t) = [x(t) \quad y(t) \quad z(t)] = [t^3 \quad t^2 \quad t \quad 1] \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} \begin{bmatrix} G_1 \\ G_2 \\ G_3 \\ G_4 \end{bmatrix}. \quad (11.9)$$

Multiplying out just $x(t) = T \cdot M \cdot G_x$ gives

$$\begin{aligned} x(t) = {} & (t^3 m_{11} + t^2 m_{21} + t\, m_{31} + m_{41}) g_{1x} + (t^3 m_{12} + t^2 m_{22} + t\, m_{32} + m_{42}) g_{2x} \\ & + (t^3 m_{13} + t^2 m_{23} + t\, m_{33} + m_{43}) g_{3x} + (t^3 m_{14} + t^2 m_{24} + t\, m_{34} + m_{44}) g_{4x}. \end{aligned} \quad (11.10)$$

Equation (11.10) emphasizes that the curve is a weighted sum of the elements of the geometry matrix. The weights are each cubic polynomials of t, and are called *blending functions*. The blending functions B are given by $B = T \cdot M$. Notice the similarity to a piecewise linear approximation, for which only two geometric constraints (the endpoints of the line) are needed, so each curve segment is a straight line defined by the endpoints G_1 and G_2:

$$\begin{aligned} x(t) &= g_{1x}(1 - t) + g_{2x}(t), \\ y(t) &= g_{1y}(1 - t) + g_{2y}(t), \\ z(t) &= g_{1z}(1 - t) + g_{2z}(t). \end{aligned} \quad (11.11)$$

Parametric cubics are really just a generalization of straight-line approximations.

To see how to calculate the basis matrix M, we turn now to specific forms of parametric cubic curves.

11.2.1 Hermite Curves

The Hermite form (named for the mathematician) of the cubic polynomial curve segment is determined by constraints on the endpoints P_1 and P_4 and tangent vectors at the endpoints R_1 and R_4. (The indices 1 and 4 are used, rather than 1 and 2, for consistency with later sections, where intermediate points P_2 and P_3 will be used instead of tangent vectors to define the curve.)

To find the *Hermite basis matrix* M_H, which relates the *Hermite geometry vector* G_H to the polynomial coefficients, we write four equations, one for each of the constraints, in the four unknown polynomial coefficients, and then solve for the unknowns.

Samsung/Google, EX1027, p. 496

Defining G_{H_x}, the x component of the Hermite geometry matrix, as

$$G_{H_x} = \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}_x, \tag{11.12}$$

and rewriting $x(t)$ from Eqs. (11.5) and (11.9) as

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x = T \cdot C_x = T \cdot M_H \cdot G_{H_x} \cdot = [t^3 \quad t^2 \quad t \quad 1]\, M_H \cdot G_{H_x}, \tag{11.13}$$

the constraints on $x(0)$ and $x(1)$ are found by direct substitution into Eq. (11.13) as

$$x(0) = P_{1_x} = [0 \quad 0 \quad 0 \quad 1]\, M_H \cdot G_{H_x}, \tag{11.14}$$

$$x(1) = P_{4_x} = [1 \quad 1 \quad 1 \quad 1]\, M_H \cdot G_{H_x}. \tag{11.15}$$

Just as in the general case we differentiated Eq. (11.7) to find Eq. (11.8), we now differentiate Eq. (11.13) to get $x'(t) = [3t^2 \quad 2t \quad 1 \quad 0]M_H \cdot G_{H_x}$. Hence, the tangent-vector–constraint equations can be written as

$$x'(0) = R_{1_x} = [0 \quad 0 \quad 1 \quad 0]\, M_H \cdot G_{H_x}, \tag{11.16}$$

$$x'(1) = R_{4_x} = [3 \quad 2 \quad 1 \quad 0]\, M_H \cdot G_{H_x}. \tag{11.17}$$

The four constraints of Eqs. (11.14), (11.15), (11.16), and (11.17) can be rewritten in matrix form as

$$\begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}_x = G_{H_x} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix} M_H \cdot G_{H_x} \tag{11.18}$$

For this equation (and the corresponding expressions for y and z) to be satisfied, M_H must be the inverse of the 4×4 matrix in Eq. (11.18)

$$M_H = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}. \tag{11.19}$$

M_H, which is of course unique, can now be used in $x(t) = T \cdot M_H \cdot G_{H_x}$ to find $x(t)$ based on the geometry vector G_{H_x}. Similarly, $y(t) = T \cdot M_H \cdot G_{H_y}$ and $z(t) = T \cdot M_H \cdot G_{H_z}$, so we can write

$$Q(t) = [x(t) \quad y(t) \quad z(t)] = T \cdot M_H \cdot G_H, \tag{11.20}$$

where G_H is the column vector

$$\begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}.$$

Samsung/Google, EX1027, p. 497

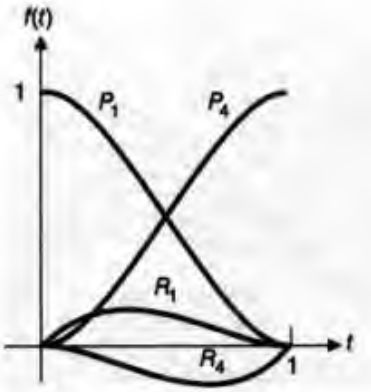


Fig. 11.12 The Hermite blending functions, labeled by the elements of the geometry vector that they weight.

Expanding the product $T \cdot M_H$ in $Q(t) = T \cdot M_H \cdot G_H$ gives the *Hermite blending functions* B_H as the polynomials weighting each element of the geometry vector:

$$Q(t) = T \cdot M_H \cdot G_H = B_H \cdot G_H$$
$$= (2t^3 - 3t^2 + 1)P_1 + (-2t^3 + 3t^2)P_4 + (t^3 - 2t^2 + t)R_1 + (t^3 - t^2)R_4. \quad (11.21)$$

Figure 11.12 shows the four blending functions. Notice that, at $t = 0$, only the function labeled P_1 is nonzero: only P_1 affects the curve at $t = 0$. As soon as t becomes greater than zero, R_1, P_4, and R_4 begin to have an influence. Figure 11.13 shows the four functions weighted by the y components of a geometry vector, their sum $y(t)$, and the curve $Q(t)$.

Figure 11.14 shows a series of Hermite curves. The only difference among them is the length of the tangent vector R_1: the directions of the tangent vectors are fixed. The longer the vectors, the greater their effect on the curve. Figure 11.15 is another series of Hermite curves, with constant tangent-vector lengths but with different directions. In an interactive

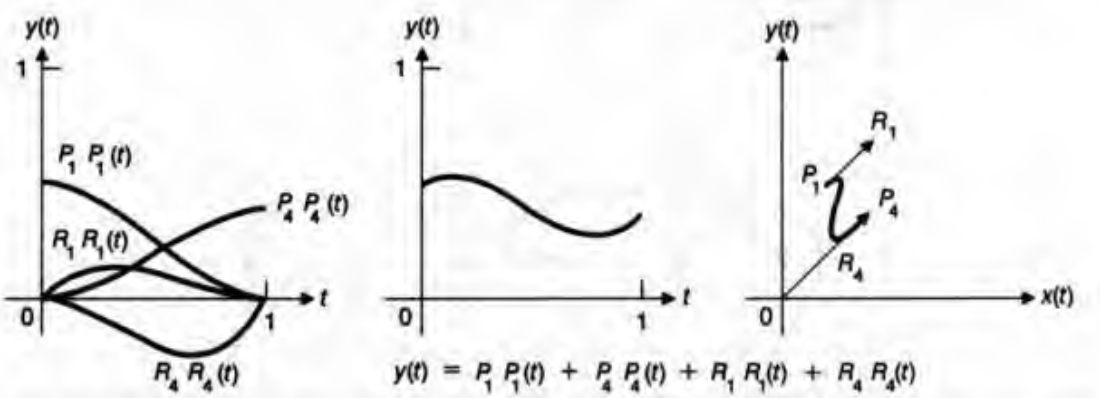


Fig. 11.13 A Hermite curve showing the four elements of the geometry vector weighted by the blending functions (leftmost four curves), their sum *y(t)*, and the 2D curve itself (far right). *x(t)* is defined by a similar weighted sum.

Samsung/Google, EX1027, p. 498

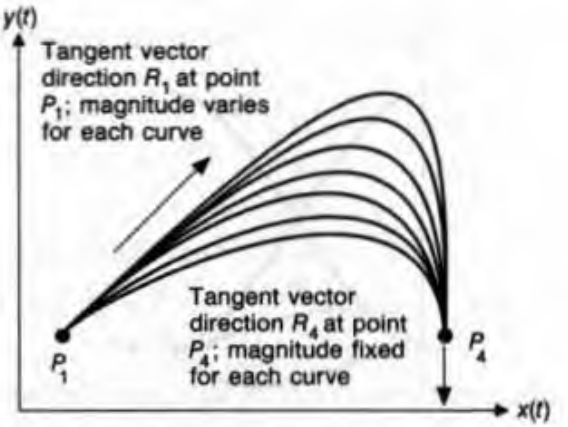


Fig. 11.14 Family of Hermite parametric cubic curves. Only R_1, the tangent vector at P_1, varies for each curve, increasing in magnitude for the higher curves.

graphics system, the endpoints and tangent vectors of a curve are manipulated interactively by the user to shape the curve. Figure 11.16 shows one way of doing this.

For two Hermite cubics to share a common endpoint with G^1 (geometrical) continuity, as in Fig. 11.17, the geometry vectors must have the form

$$\begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix} \text{ and } \begin{bmatrix} P_4 \\ P_7 \\ kR_4 \\ R_7 \end{bmatrix}, \text{ with } k > 0. \tag{11.22}$$

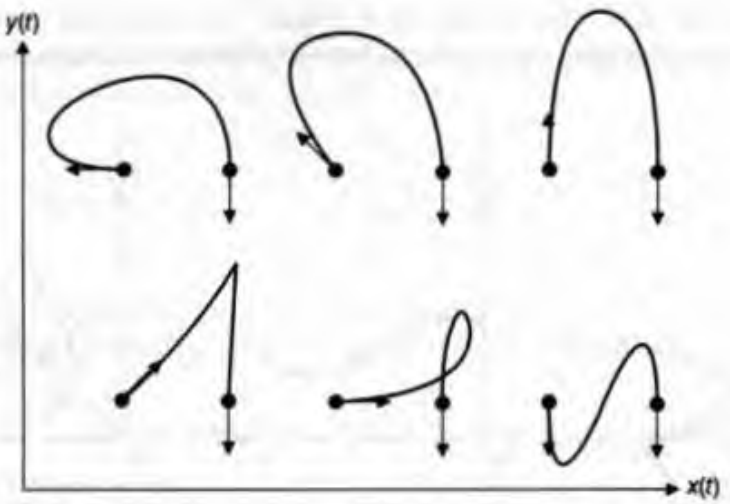


Fig. 11.15 Family of Hermite parametric cubic curves. Only the direction of the tangent vector at the left starting point varies; all tangent vectors have the same magnitude. A smaller magnitude would eliminate the loop in the one curve.

Samsung/Google, EX1027, p. 499

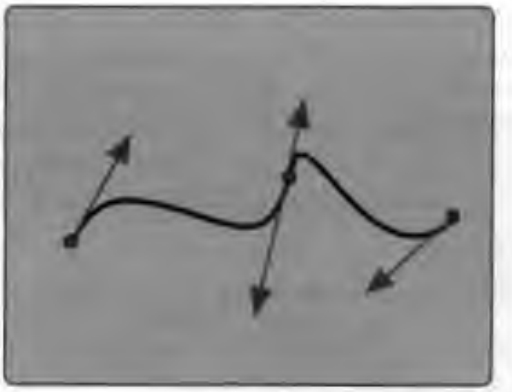


Fig. 11.16 Two Hermite cubic curve segments displayed with controls to facilitate interactive manipulation. The endpoints can be repositioned by dragging the dots, and the tangent vectors can be changed by dragging the arrowheads. The tangent vectors at the join point are constrained to be collinear (to provide C^1 continuity): The user is usually given a command to enforce C^0, C^1, G^1, or no continuity. The tangent vectors at the $t = 1$ end of each curve are drawn in the reverse of the direction used in the mathematical formulation of the Hermite curve, for clarity and more convenient user interaction.

That is, there must be a shared endpoint (P_4) and tangent vectors with at least equal directions. The more restrictive condition of C^1 (parametric) continuity requires that $k = 1$, so the tangent vector direction and magnitude must be equal.

Hermite and other similar parametric cubic curves are simple to display: We evaluate Eq. (11.5) at n successive values of t separated by a step size δ. Figure 11.18 gives the code. The evaluation within the **begin** . . . **end** takes 11 multiplies and 10 additions per 3D point. Use of Horner's rule for factoring polynomials,

$$f(t) = at^3 + bt^2 + ct + d = ((at + b)t + c)t + d, \tag{11.23}$$

reduces the effort slightly to nine multiplies and 10 additions per 3D point. In Section 11.2.9, we shall examine much more efficient ways to display these curves.

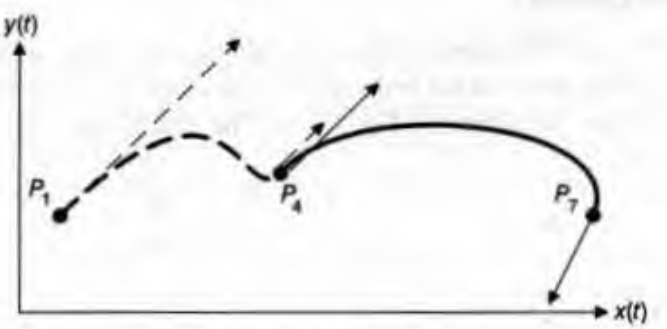


Fig. 11.17 Two Hermite curves joined at P_4. The tangent vectors at P_4 have the same direction but different magnitudes, yielding G^1 but not C^1 continuity.

Samsung/Google, EX1027, p. 500

```
typedef double CoefficientArray[4];
void DrawCurve (
        CoefficientArray cx,          /* Coefficients for x(t): C_x = M • G_x */
        CoefficientArray cy,          /* Coefficients for y(t): C_y = M • G_y */
        CoefficientArray cz,          /* Coefficients for z(t): C_z = M • G_z */
        int n)                        /* Number of steps */
{
    int i;
    double δ = 1.0 / n;
    double t = 0;

    MoveAbs3 (cx[3], cy[3], cz[3]);   /* t = 0: start at x(0), y(0), z(0) */
    for (i = 0; i < n; i++) {
        double t2, t3, x, y, z;

        t += δ;
        t2 = t * t;
        t3 = t2 * t;
        x = cx[0] * t3 + cx[1] * t2 + cx[2] * t + cx[3];
        y = cy[0] * t3 + cy[1] * t2 + cy[2] * t + cy[3];
        z = cz[0] * t3 + cz[1] * t2 + cz[2] * t + cz[3];
        DrawAbs3 (x, y, z);
    }
} /* DrawCurve */
```

Fig. 11.18 Program to display a cubic parametric curve.

Because the cubic curves are linear combinations (weighted sums) of the four elements of the geometry vector, as seen in Eq. (11.10), we can transform the curves by transforming the geometry vector and then using it to generate the transformed curve, which is equivalent to saying that the curves are invariant under rotation, scaling, and translation. This strategy is more efficient than is generating the curve as a series of short line segments and then transforming each individual line. The curves are *not* invariant under perspective projection, as will be discussed in Section 11.2.5.

11.2.2 Bézier Curves

The Bézier [BEZI70; BEZI74] form of the cubic polynomial curve segment, named after Pierre Bézier, indirectly specifies the endpoint tangent vector by specifying two intermediate points that are not on the curve; see Fig. 11.19. The starting and ending tangent vectors

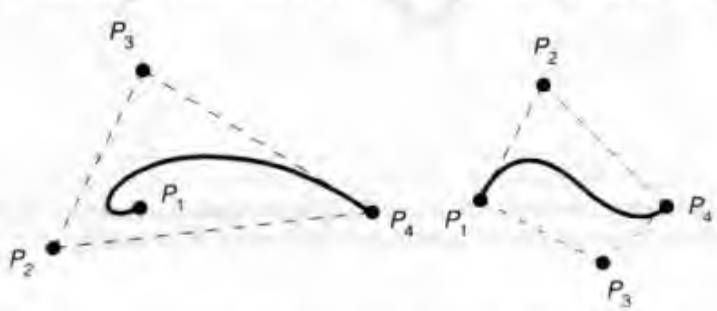


Fig. 11.19 Two Bézier curves and their control points. Notice that the convex hulls of the control points, shown as dashed lines, do not need to touch all four control points.

Samsung/Google, EX1027, p. 501

are determined by the vectors P_1P_2 and P_3P_4 and are related to R_1 and R_4 by

$$R_1 = Q'(0) = 3(P_2 - P_1),\ R_4 = Q'(1) = 3(P_4 - P_3). \tag{11.24}$$

The Bézier curve interpolates the two end control points and approximates the other two. See Exercise 11.12 to understand why the constant 3 is used in Eq. (11.24).

The *Bézier geometry vector* G_B, consisting of four points, is

$$G_B = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}. \tag{11.25}$$

Then the matrix M_{HB} that defines the relation $G_H = M_{HB} \cdot G_B$ between the Hermite geometry vector G_H and the Bézier geometry vector G_B is just the 4×4 matrix in the following equation, which rewrites Eq. (11.24) in matrix form:

$$G_H = \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix} = M_{HB} \cdot G_B. \tag{11.26}$$

To find the *Bézier basis matrix* M_B, we use Eq. (11.20) for the Hermite form, substitute $G_H = M_{HB} \cdot G_B$, and define $M_B = M_H \cdot M_{HB}$:

$$Q(t) = T \cdot M_H \cdot G_H = T \cdot M_H \cdot (M_{HB} \cdot G_B) = T \cdot (M_H \cdot M_{HB}) \cdot G_B = T \cdot M_B \cdot G_B. \tag{11.27}$$

Carrying out the multiplication $M_B = M_H \cdot M_{HB}$ gives

$$M_B = M_H \cdot M_{HB} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \tag{11.28}$$

and the product $Q(t) = T \cdot M_B \cdot G_B$ is

$$Q(t) = (1 - t)^3P_1 + 3t(1 - t)^2P_2 + 3t^2(1 - t)P_3 + t^3P_4. \tag{11.29}$$

The four polynomials $B_B = T \cdot M_B$, which are the weights in Eq. (11.29), are called the *Bernstein polynomials*, and are shown in Fig. 11.20.

Figure 11.21 shows two Bézier curve segments with a common endpoint. G^1 continuity is provided at the endpoint when $P_3 - P_4 = k(P_4 - P_5)$, $k > 0$. That is, the three points P_3, P_4, and P_5 must be distinct and collinear. In the more restrictive case when $k = 1$, there is C^1 continuity in addition to G^1 continuity.

If we refer to the polynomials of two curve segments as x^l (for the left segment) and x^r (for the right segment), we can find the conditions for C^0 and C^1 continuity at their join point:

$$x^l(1) = x^r(0), \quad \frac{d}{dt}x^l(1) = \frac{d}{dt}x^r(0). \tag{11.30}$$

Samsung/Google, EX1027, p. 502

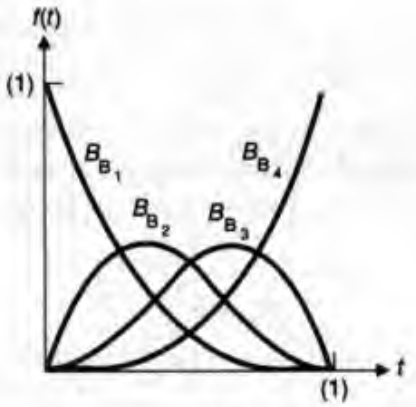


Fig. 11.20 The Bernstein polynomials, which are the weighting functions for Bézier curves. At $t = 0$, only B_{B_1} is nonzero, so the curve interpolates P_1; similarly, at $t = 1$, only B_{B_4} is nonzero, and the curve interpolates P_4.

Working with the x component of Eq. (11.29), we have

$$x^1(1) = x^r(0) = P_{4_x}, \frac{d}{dt}x^1(1) = 3(P_{4_x} - P_{3_x}), \frac{d}{dt}x^r(0) = 3(P_{5_x} - P_{4_x}). \quad (11.31)$$

As always, the same conditions are true of y and z. Thus, we have C^0 and C^1 continuity when $P_4 - P_3 = P_5 - P_4$, as expected.

Examining the four B_B polynomials in Eq. (11.29), we note that their sum is everywhere unity and that each polynomial is everywhere nonnegative for $0 \leq t < 1$. Thus, $Q(t)$ is just a weighted average of the four control points. This condition means that each curve segment, which is just the sum of four control points weighted by the polynomials, is completely contained in the *convex hull* of the four control points. The convex hull for 2D curves is the convex polygon formed by the four control points: Think of it as the polygon formed by putting a rubberband around the points (Fig. 11.19). For 3D curves, the convex hull is the convex polyhedron formed by the control points: Think of it as the polyhedron formed by stretching a rubber sheet around the four points.

This convex-hull property holds for all cubics defined by weighted sums of control points if the blending functions are nonnegative and sum to one. In general, the weighted average of n points falls within the convex hull of the n points; this can be seen intuitively

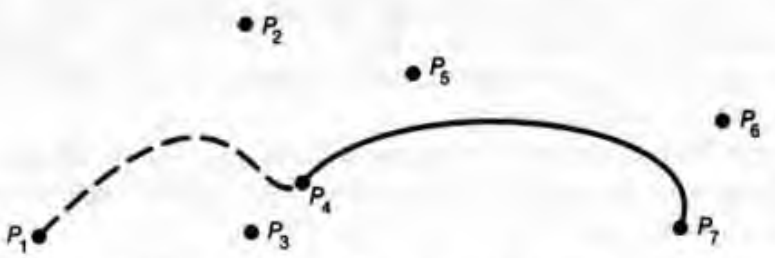


Fig. 11.21 Two Bézier curves joined at P_4. Points P_3, P_4, and P_5 are collinear. Curves are the same as those used in Fig. 11.17.

Samsung/Google, EX1027, p. 503

for $n = 2$ and $n = 3$, and the generalization follows. Another consequence of the fact that the four polynomials sum to unity is that the value of the fourth polynomial for any value of t can be found by subtracting the first three from unity.

The convex-hull property is also useful for clipping curve segments: Rather than clip each short line piece of a curve segment to determine its visibility, we first apply a polygonal clip algorithm to clip the convex hull or its extent against the clip region. If the convex hull (extent) is completely within the clip region, so is the entire curve segment. If the convex hull (extent) is completely outside the clip region, so is the curve segment. Only if the convex hull (extent) intersects the clip region does the curve segment itself need to be examined.

11.2.3 Uniform Nonrational B-Splines

The term *spline* goes back to the long flexible strips of metal used by draftspersons to lay out the surfaces of airplanes, cars, and ships. "Ducks," weights attached to the splines, were used to pull the spline in various directions. The metal splines, unless severely stressed, had second-order continuity. The mathematical equivalent of these strips, the *natural cubic spline*, is a C^0, C^1, and C^2 continuous cubic polynomial that interpolates (passes through) the control points. This is 1 more degree of continuity than is inherent in the Hermite and Bézier forms. Thus, splines are inherently smoother than are the previous forms.

The polynomial coefficients for natural cubic splines, however, are dependent on all n control points; their calculation involves inverting an $n + 1$ by $n + 1$ matrix [BART87]. This has two disadvantages: moving any one control point affects the entire curve, and the computation time needed to invert the matrix can interfere with rapid interactive reshaping of a curve.

B-splines, discussed in this section, consist of curve segments whose polynomial coefficients depend on just a few control points. This is called *local control*. Thus, moving a control point affects only a small part of a curve. In addition, the time needed to compute the coefficients is greatly reduced. B-splines have the same continuity as natural splines, but do not interpolate their control points.

In the following discussion we change our notation slightly, since we must discuss an entire curve consisting of several curve segments, rather than its individual segments. A curve segment need not pass through its control points, and the two continuity conditions on a segment come from the adjacent segments. This is achieved by sharing control points between segments, so it is best to describe the process in terms of all the segments at once.

Cubic B-splines approximate a series of $m + 1$ control points $P_0, P_1, \ldots P_m$, $m \geq 3$, with a curve consisting of $m - 2$ cubic polynomial curve segments $Q_3, Q_4, \ldots Q_m$. Although such cubic curves might be defined each on its own domain $0 \leq t < 1$, we can adjust the parameter (making a substitution of the form $t = t + k$) so that the parameter domains for the various curve segments are sequential. Thus, we say that the parameter range on which Q_i is defined is $t_i \leq t < t_{i+1}$, for $3 \leq i \leq m$. In the particular case of $m = 3$, there is a single curve segment Q_3 that is defined on the interval $t_3 \leq t < t_4$ by four control points, P_0 to P_3.

For each $i \geq 4$, there is a join point or *knot* between Q_{i-1} and Q_i at the parameter value t_i; the parameter value at such a point is called a *knot value*. The initial and final points at t_3

Samsung/Google, EX1027, p. 504

and t_{m+1} are also called knots, so that there is a total of $m - 1$ knots. Figure 11.22 shows a 2D B-spline curve with its knots marked. A closed B-spline curve is easy to create: The control points P_0, P_1, P_2 are repeated at the end of the sequence—P_0, P_1, . . . P_m, P_0, P_1, P_2.

The term *uniform* means that the knots are spaced at equal intervals of the parameter t. Without loss of generality, we can assume that $t_3 = 0$ and the interval $t_{i+1} - t_i = 1$. Nonuniform nonrational B-splines, which permit unequal spacing between the knots, are discussed in Section 11.2.4. (In fact, the concept of knots is introduced in this section to set the stage for nonuniform splines.) The term *nonrational* is used to distinguish these splines from rational cubic polynomial curves, discussed in Section 11.2.5, where $x(t)$, $y(t)$, and $z(t)$ are each defined as the ratio of two cubic polynomials. The "B" stands for basis, since the splines can be represented as weighted sums of polynomial basis functions, in contrast to the natural splines, for which this is not true.

Each of the $m - 2$ curve segments of a B-spline curve is defined by four of the $m + 1$ control points. In particular, curve segment Q_i is defined by points P_{i-3}, P_{i-2}, P_{i-1}, and P_i. Thus, the *B-spline geometry vector* G_{Bs_i} for segment Q_i is

$$G_{Bs_i} = \begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix}, \quad 3 \le i \le m. \tag{11.32}$$

The first curve segment, Q_3, is defined by the points P_0 through P_3 over the parameter range $t_3 = 0$ to $t_4 = 1$, Q_4 is defined by the points P_1 through P_4 over the parameter range $t_4 = 1$ to $t_5 = 2$, and the last curve segment, Q_m, is defined by the points P_{m-3}, P_{m-2}, P_{m-1}, and P_m over the parameter range $t_m = m - 3$ to $t_{m+1} = m - 2$. In general, curve segment Q_i begins somewhere near point P_{i-2} and ends somewhere near point P_{i-1}. We shall see that the B-spline blending functions are everywhere nonnegative and sum to unity, so the curve segment Q_i is constrained to the convex hull of its four control points.

Just as each curve segment is defined by four control points, each control point (except for those at the beginning and end of the sequence P_0, P_1, . . . P_m) influences four curve

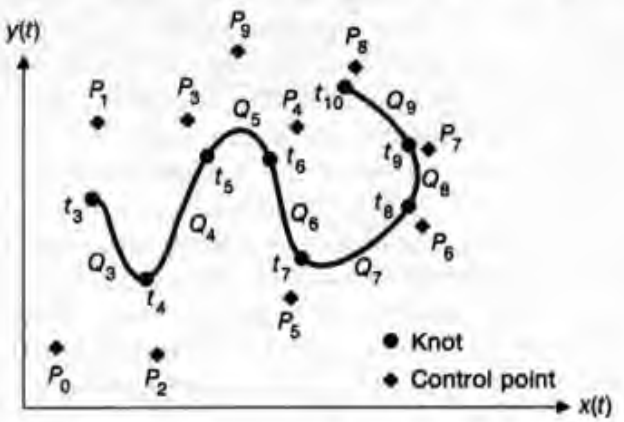


Fig. 11.22 A B-spline with curve segments Q_3 through Q_9. This and many other figures in this chapter were created with a program written by Carles Castellsaqué.

Samsung/Google, EX1027, p. 505

segments. Moving a control point in a given direction moves the four curve segments it affects in the same direction; the other curve segments are totally unaffected (see Fig. 11.23). This is the local control property of B-splines and of all the other splines discussed in this chapter.

If we define T_i as the row vector $[(t - t_i)^3 \quad (t - t_i)^2 \quad (t - t_i) \quad 1]$, then the B-spline formulation for curve segment i is

$$Q_i(t) = T_i \cdot M_{Bs} \cdot G_{Bs_i}, \quad t_i \le t < t_{i+1}. \tag{11.33}$$

The entire curve is generated by applying Eq. (11.33) for $3 \le i \le m$.

The *B-spline basis matrix*, M_{Bs}, relates the geometrical constraints G_{Bs} to the blending functions and the polynomial coefficients:

$$M_{Bs} = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}. \tag{11.34}$$

This matrix is derived in [BART87].

The B-spline blending functions B_{Bs} are given by the product $T_i \cdot M_{Bs}$, analogously to the previous Bézier and Hermite formulations. Note that the blending functions for each curve segment are exactly the same, because for each segment i the values of $t - t_i$ range from 0 at $t = t_i$ to 1 at $t = t_{i+1}$. If we replace $t - t_i$ by t, and replace the interval $[t_i, t_{i+1}]$ by $[0, 1]$, we have

$$\begin{aligned} B_{Bs} = T \cdot M_{Bs} &= [B_{Bs_{-3}} \quad B_{Bs_{-2}} \quad B_{Bs_{-1}} \quad B_{Bs_0}] \\ &= \frac{1}{6}[-t^3 + 3t^2 - 3t + 1 \quad 3t^3 - 6t^2 + 4 \quad -3t^3 + 3t^2 + 3t + 1 \quad t^3] \\ &= \frac{1}{6}[(1 - t)^3 \quad 3t^3 - 6t^2 + 4 \quad -3t^3 + 3t^2 + 3t + 1 \quad t^3], \ 0 \le t < 1. \end{aligned} \tag{11.35}$$

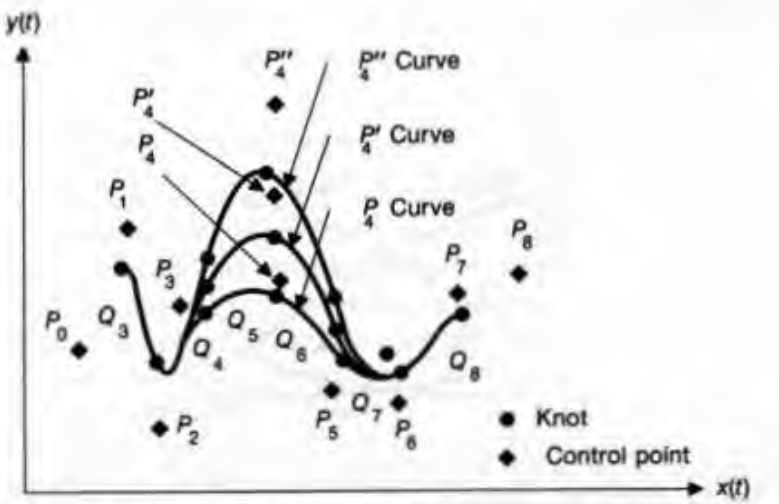


Fig. 11.23 A B-spline with control point P_4 in several different locations.

Samsung/Google, EX1027, p. 506

Figure 11.24 shows the B-spline blending functions B_{Bs_i}. Because the four functions sum to 1 and are nonnegative, the convex-hull property holds for each curve segment of a B-spline. See [BART87] to understand the relation between these blending functions and the Bernstein polynomial basis functions.

Expanding Eq. (11.33), again replacing $t - t_i$ with t at the second equals sign, we have

$$\begin{aligned} Q_i(t - t_i) &= T_i \cdot M_{Bs} \cdot G_{Bs_i} = T \cdot M_{Bs} \cdot G_{Bs_i} \\ &= B_{Bs} \cdot G_{Bs_i} = B_{Bs_{-3}} \cdot P_{i-3} + B_{Bs_{-2}} \cdot P_{i-2} + B_{Bs_{-1}} \cdot P_{i-1} + B_{Bs_0} \cdot P_i \\ &= \frac{(1-t)^3}{6} P_{i-3} + \frac{3t^3 - 6t^2 + 4}{6} P_{i-2} + \frac{-3t^3 + 3t^2 + 3t + 1}{6} P_{i-1} \\ &\quad + \frac{t^3}{6} P_i, \; 0 \le t < 1. \end{aligned} \tag{11.36}$$

It is easy to show that Q_i and Q_{i+1} are C^0, C^1, and C^2 continuous where they join. When we consider the x components of the adjacent segments, which are $x_i(t - t_i)$ and $x_{i+1}(t - t_{i+1})$ (y and z, as always, are analogous), it is necessary to show only that, at the knot t_{i+1} where they join,

$$x_i(t_{i+1}) = x_{i+1}(t_{i+1}), \quad \frac{d}{dt} x_i(t_{i+1}) = \frac{d}{dt} x_{i+1}(t_{i+1}), \quad \text{and} \quad \frac{d^2}{dt^2} x_i(t_{i+1}) = \frac{d^2}{dt^2} x_{i+1}(t_{i+1}). \tag{11.37}$$

Recalling the substitution of t for $t - t_i$, Eq. (11.37) is equivalent to showing that

$$\begin{aligned} x_i|_{t-t_i=1} &= x_{i+1}|_{t-t_{i+1}=0}, \\ \frac{d}{dt} x_i|_{t-t_i=1} &= 1 \, \frac{d}{dt} x_{i+1}|_{t-t_{i+1}=0}, \\ \frac{d^2}{dt^2} x_i|_{t-t_i=1} &= \frac{d^2}{dt^2} \, x_{i+1}|_{t-t_{i+1}=0}. \end{aligned} \tag{11.38}$$

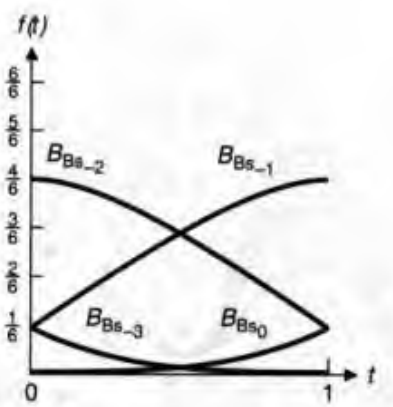


Fig. 11.24 The four B-spline blending functions from Eq. (11.35). At $t = 0$ and $t = 1$, just three of the functions are nonzero.

Samsung/Google, EX1027, p. 507

We demonstrate the equivalence by working with the x component of Eq. (11.36), and its first and second derivatives, to yield

$$x_i|_{t-t_i=1} = x_{i+1}|_{t-t_i=1} = \frac{P_{i-2_x} + 4P_{i-1_x} + P_{i_x}}{6}, \tag{11.39}$$

$$\frac{d}{dt}x_i|_{t-t_i=1} = \frac{d}{dt}x_{i+1}|_{t-t_{i+1}=0} = \frac{-P_{i-2_x} + P_{i_x}}{2} \tag{11.40}$$

$$\frac{d^2}{dt^2}x_i|_{t-t_i=1} = \frac{d^2}{dt^2}x_{i+1}|_{t-t_{i+1}=0} = P_{i-2_x} - 2P_{i-1_x} + P_{i_x}. \tag{11.41}$$

The additional continuity afforded by B-splines is attractive, but it comes at the cost of less control of where the curve goes. The curve can be forced to interpolate specific points by replicating control points; this is useful both at endpoints and at intermediate points on the curve. For instance, if $P_{i-2} = P_{i-1}$, the curve is pulled closer to this point because curve segment Q_i is defined by just three different points, and the point $P_{i-2} = P_{i-1}$ is weighted twice in Eq. (11.36)—once by $B_{Bs_{-2}}$ and once by $B_{Bs_{-1}}$.

If a control point is used three times—for instance, if $P_{i-2} = P_{i-1} = P_i$—then Eq. (11.36) becomes

$$Q_i(t) = B_{Bs_{-3}} \cdot P_{i-3} + (B_{Bs_{-2}} + B_{Bs_{-1}} + B_{Bs_0}) \cdot P_i. \tag{11.42}$$

Q_i is clearly a straight line. Furthermore, the point P_{i-2} is interpolated by the line at $t = 1$, where the three weights applied to P_i sum to 1, but P_{i-3} is not in general interpolated at $t = 0$. Another way to think of this is that the convex hull for Q_i is now defined by just two distinct points, so Q_i has to be a line. Figure 11.25 shows the effect of multiple control points at the interior of a B-spline. The price of interpolating the points in part (*c*) is loss of G^1 continuity, even though Eq. (11.40) shows that C^1 continuity is preserved (but with a value of zero). This is a case where C^1 continuity does not imply G^1 continuity, as discussed in Section 11.2.

Another technique for interpolating endpoints, *phantom vertices*, is discussed in [BARS83; BART87]. We shall see that, with nonuniform B-splines, discussed in the next section, endpoints and internal points can be interpolated in a more natural way than they can with the uniform B-splines.

11.2.4 Nonuniform, Nonrational B-Splines

Nonuniform, nonrational B-splines differ from the uniform, nonrational B-splines discussed in the previous section in that the parameter interval between successive knot values need not be uniform. The nonuniform knot-value sequence means that the blending functions are no longer the same for each interval, but rather vary from curve segment to curve segment.

These curves have several advantages over uniform B-splines. First, continuity at selected join points can be reduced from C^2 to C^1 to C^0 to none. If the continuity is reduced to C^0, then the curve interpolates a control point, but without the undesirable effect of uniform B-splines, where the curve segments on either side of the interpolated control point are straight lines. Also, starting and ending points can be easily interpolated exactly,

Samsung/Google, EX1027, p. 508

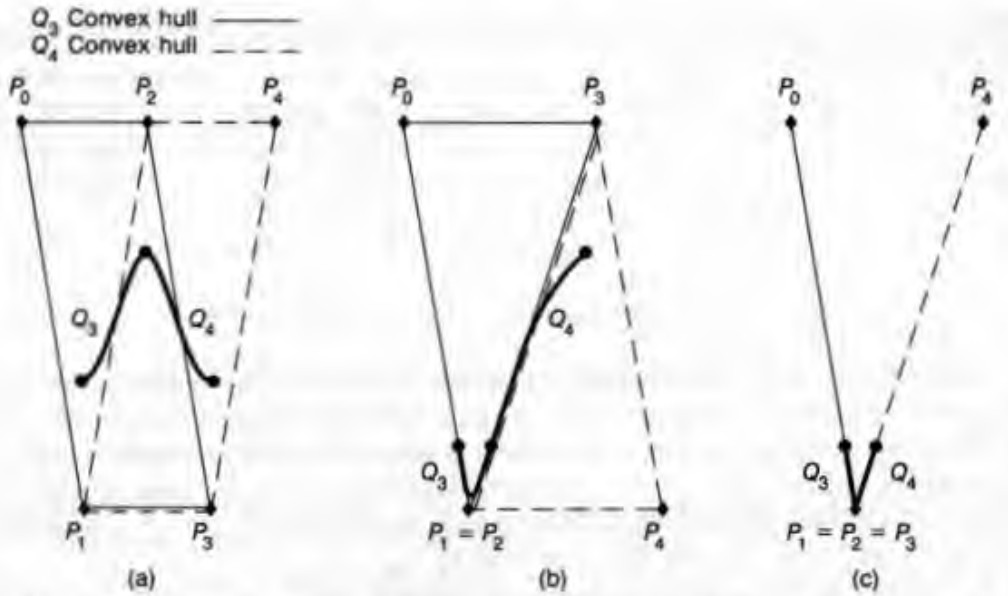


Fig. 11.25 The effect of multiple control points on a uniform B-spline curve. In (a), there are no multiple control points. The convex hulls of the two curves overlap; the join point between Q_3 and Q_4 is in the region shared by both convex hulls. In (b), there is a double control point, so the two convex hulls share edge P_2P_3; the join point is therefore constrained to lie on this edge. In (c), there is a triple control point, and the two convex hulls are straight lines that share the triple point; hence, the join point is also at the triple point. Because the convex hulls are straight lines, the two curve segments must also be straight lines. There is C^2 but only G^0 continuity at the join.

without at the same time introducing linear segments. As is further discussed in Section 11.2.7, it is possible to add an additional knot and control point to nonuniform B-splines, so the resulting curve can be easily reshaped, whereas this cannot be done with uniform B-splines.

The increased generality of nonuniform B-splines requires a slightly different notation than that used for uniform B-splines. As before, the spline is a piecewise continuous curve made up of cubic polynomials, approximating the control points P_0 through P_m. The *knot-value sequence* is a nondecreasing sequence of knot values t_0 through t_{m+4} (that is, there are four more knots than there are control points). Because the smallest number of control points is four, the smallest knot sequence has eight knot values and the curve is defined over the parameter interval from t_3 to t_4.

The only restriction on the knot sequence is that it be nondecreasing, which allows successive knot values to be equal. When this occurs, the parameter value is called a *multiple knot* and the number of identical parameter values is called the *multiplicity* of the knot (a single unique knot has multiplicity of 1). For instance, in the knot sequence (0, 0, 0, 0, 1, 1, 2, 3, 4, 4, 5, 5, 5, 5), the knot value 0 has multiplicity four; value 1 has multiplicity 2; values 2 and 3 have multiplicity 1; value 4 has multiplicity 2; and value 5 has multiplicity 4.

Samsung/Google, EX1027, p. 509

Curve segment Q_i is defined by control points P_{i-3}, P_{i-2}, P_{i-1}, P_i and by blending functions $B_{i-3,4}(t)$, $B_{i-2,4}(t)$, $B_{i-1,4}(t)$, $B_{i,4}(t)$, as the weighted sum

$$Q_i(t) = P_{i-3} \cdot B_{i-3,4}(t) + P_{i-2} \cdot B_{i-2,4}(t) + P_{i-1} \cdot B_{i-1,4}(t) + P_i \cdot B_{i,4}(t)$$
$$3 \le i \le m, \quad t_i \le t < t_{i+1}. \tag{11.43}$$

The curve is not defined outside the interval t_3 through t_{m+1}. When $t_i = t_{i+1}$ (a multiple knot), curve segment Q_i is a single point. It is this notion of a curve segment reducing to a point that provides the extra flexibility of nonuniform B-splines.

There is no single set of blending functions, as there was for other types of splines. The functions depend on the intervals between knot values and are defined recursively in terms of lower-order blending functions. $B_{i,j}(t)$ is the jth-order blending function for weighting control point P_i. Because we are working with fourth-order (that is, third-degree, or cubic) B-splines, the recursive definition ends with $B_{i,4}(t)$ and can be easily presented in its "unwound" form. The recurrence for cubic B-splines is

$$B_{i,1}(t) = \begin{cases} 1, & t_i \le t < t_{i+1} \\ 0, & \text{otherwise,} \end{cases}$$

$$B_{i,2}(t) = \frac{t - t_i}{t_{i+1} - t_i} B_{i,1}(t) + \frac{t_{i+2} - t}{t_{i+2} - t_{i+1}} B_{i+1,1}(t),$$

$$B_{i,3}(t) = \frac{t - t_i}{t_{i+2} - t_i} B_{i,2}(t) + \frac{t_{i+3} - t}{t_{i+3} - t_{i+1}} B_{i+1,2}(t),$$

$$B_{i,4}(t) = \frac{t - t_i}{t_{i+3} - t_i} B_{i,3}(t) + \frac{t_{i+4} - t}{t_{i+4} - t_{i+1}} B_{i+1,3}(t). \tag{11.44}$$

Figure 11.26 shows how Eq. (11.44) can be used to find the blending functions, using the knot vector (0, 0, 0, 0, 1, 1, 1, 1) as an example. The figure also makes clear why eight knot vectors are needed to compute four blending functions. $B_{3,1}(t)$ is unity on the interval $0 \le t < 1$. All other $B_{i,1}(t)$ are zero. $B_{2,2}(t)$ and $B_{3,2}(t)$ are linear ramps, and are the blending functions for linear interpolation between two points. Similarly, $B_{1,3}(t)$, $B_{2,3}(t)$, and $B_{3,3}(t)$ are quadratics, and are the blending functions for quadratic interpolation. For this particular knot vector, the $B_{i,4}(t)$ are the Bernstein polynomials, that is, the Bezier blending functions; compare them to those in Fig. 11.20. Also, for this knot vector, the curve interpolates the control points P_0 and P_3, and is in fact a Bézier curve, with the tangent vector at the endpoints determined by the vectors P_0P_1 and P_2P_3.

Computing the blending functions takes time. By restricting B-spline knot sequences to have intervals that are either 0 or 1, it is possible to store just a small number of matrices corresponding to Eq. (11.44), which covers all such possible knot configurations. This eliminates the need to reevaluate Eq. (11.44) for each curve segment.

It can be shown that the blending functions are nonnegative and sum to one, so nonuniform B-spline curve segments lie within the convex hulls of their four control points. For knots of multiplicity greater than one, the denominators can be zero because successive knot values can be equal: division by zero is defined to yield zero.

Samsung/Google, EX1027, p. 510

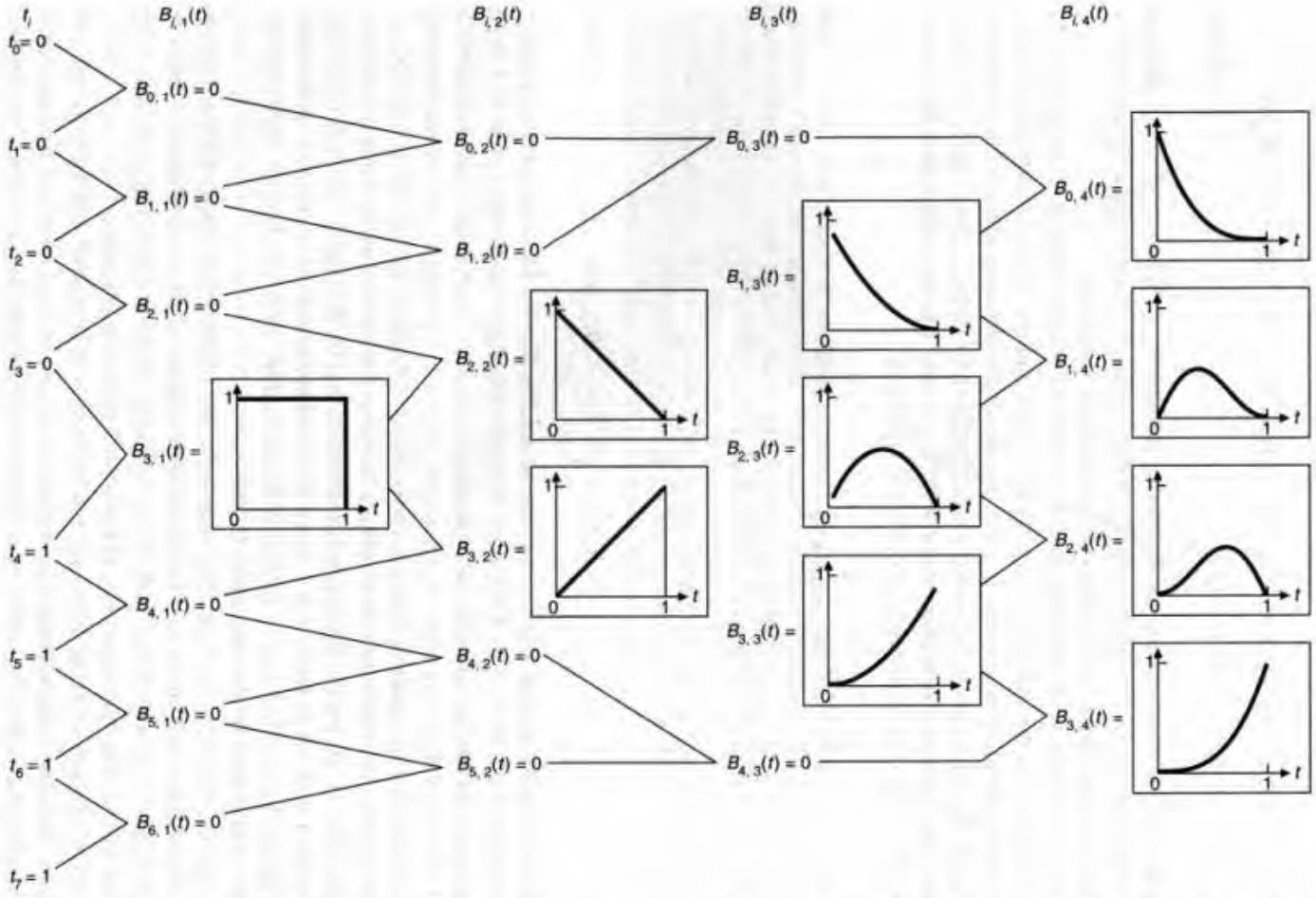


Fig. 11.26 The relationships defined by Eq. (11.44) between the knot vector (0, 0, 0, 0, 1, 1, 1, 1) and the blending functions $B_{i,1}(t)$, $B_{i,2}(t)$, $B_{i,3}(t)$, and $B_{i,4}(t)$.

Samsung/Google, EX1027, p. 511

Increasing knot multiplicity has two effects. First, each knot value t_i will automatically lie within the convex hull of the points P_{i-3}, P_{i-2}, and P_{i-1}. If t_i and t_{i+1} are equal, they must lie in the convex hull of P_{i-3}, P_{i-2}, and P_{i-1}, *and* in the convex hull of P_{i-2}, P_{i-1}, and P_i. This means they must actually lie on the line segment between P_{i-2} and P_{i-1}. In the same way, if $t_i = t_{i+1} = t_{i+2}$, then this knot must lie *at* P_{i-1}. If $t_i = t_{i+1} = t_{i+2} = t_{i+3}$, then the knot must lie both at P_{i-1} and at P_i—the curve becomes broken. Second, the multiple knots will reduce parametric continuity: from C^2 to C^1 continuity for one extra knot (multiplicity 2); from C^1 to C^0 continuity for two extra knots (multiplicity 3); from C^0 to no continuity for three extra knots (multiplicity 4).

Figure 11.27 provides further insight for a specific case. Part (a) shows the case when all knots have multiplicity 1. Each curve segment is defined by four control points and four blending functions, and adjacent curve segments each share three control points. For instance, curve segment Q_3 is defined by points P_0, P_1, P_2, and P_3; curve segment Q_4 is defined by points P_1, P_2, P_3 and P_4; and curve segment Q_5 is defined by points P_2, P_3, P_4, and P_5. Part (b) shows a double knot, $t_4 = t_5$, for which the curve segment Q_4 has zero length. Segments Q_3 and Q_5 are thus adjacent but share only two control points, P_2 and P_3; the two curve segments hence have less "in common," as implied by the loss of 1 degree of continuity. For the triple knot in part (c), only control point P_3 is shared in common: the one that the two curve segments now interpolate. Because only one control point is shared, we can expect only one constraint, C^0 continuity, to be satisfied at the join. The knot of multiplicity 4, shown in part (d), causes a discontinuity, or break, in the curve. Hence, several disjoint splines can be represented by a single knot sequence and set of control points. Figure 11.28 provides additional understanding of the relations among knots, curve segments, and control points. Table 11.1 summarizes the effects of multiple control points and multiple knots.

TABLE 11.1 COMPARISON OF THE EFFECTS OF MULTIPLE CONTROL POINTS AND OF MULTIPLE KNOTS

Multiplicity	Multiple control points	Multiple knots
1	C^2 G^{2*}	C^2 G^{2*}
2	C^2 G^1 Knots constrained to a smaller convex hull.	C^1 G^1 Knots constrained to a smaller convex hull of fewer control points.
3	C^2 G^0 Curve interpolates the triple control point. Curve segments on either side of the join are linear.	C^0 G^0 Curve interpolates control point. Can control shape of curve segments on either side of the join.
4	C^2 G^0 Curve interpolates the quadruple control points. Curve segments on either side of the join are linear and interpolate the control points on either side of the join.	There is a discontinuity in the curve. Curve stops on one control point, resumes at next. Can control shape of curve segments on either side of the discontinuity.

*Except for special case discussed in Section 11.2.

Samsung/Google, EX1027, p. 512

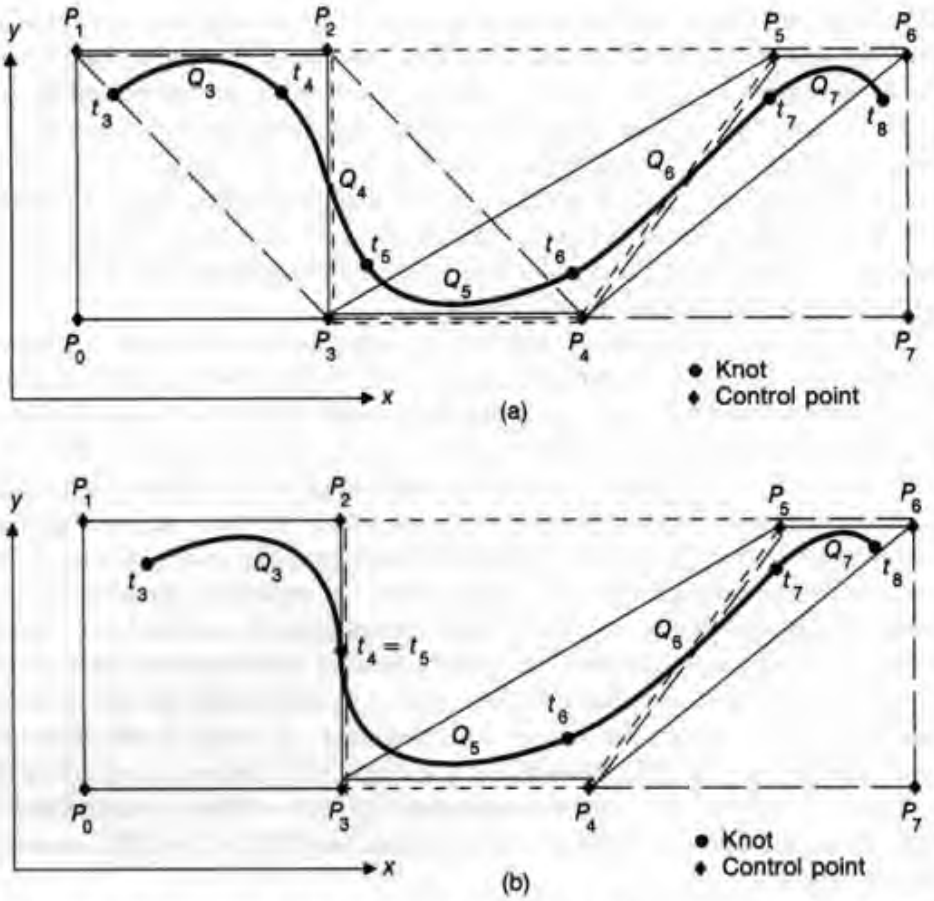
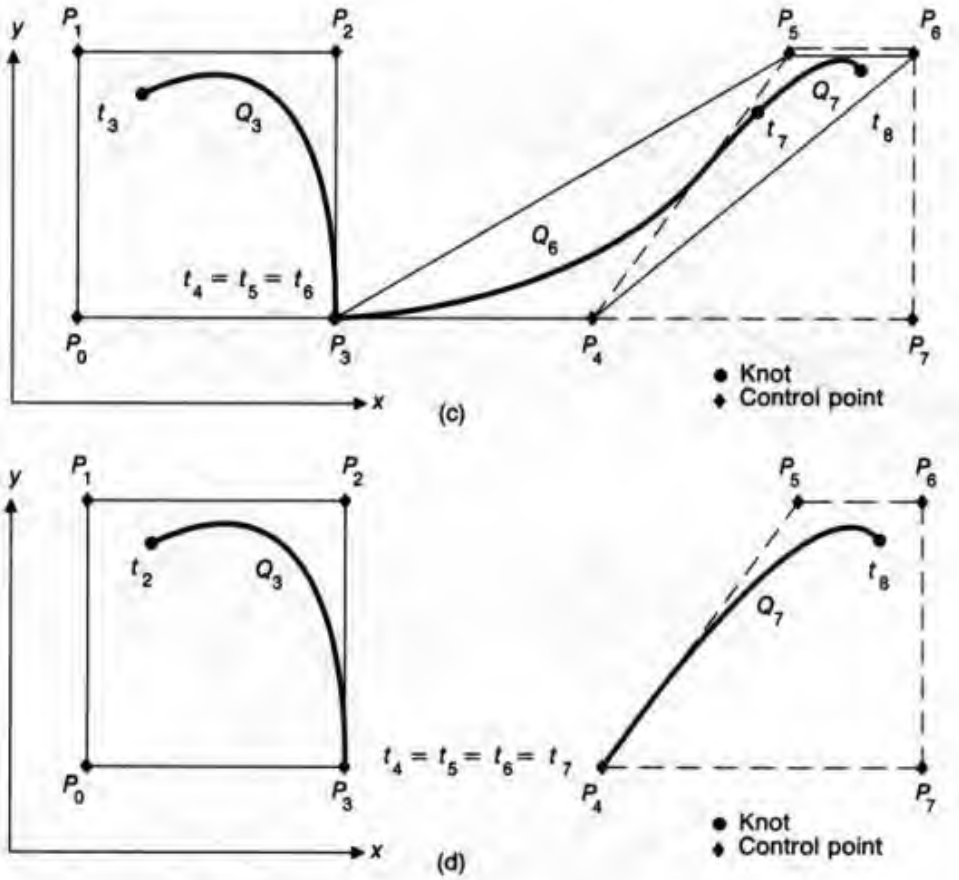


Fig. 11.27 The effect of multiple knots. In (a), with knot sequence (0, 1, 2, 3, 4, 5), there are no multiple knots; all the curve segments join with C^2 and G^2 continuity. The convex hulls containing each curve segment are also shown. In (b), with knot sequence (0, 1, 1, 2, 3, 4), there is a double knot, so curve segment Q_4 degenerates to a point. The convex hulls containing Q_3 and Q_5 meet along the edge P_2P_3, on which the join point is forced to lie. The join has C^1 and G^2 continuity. In (c), with knot sequence (0, 1, 1, 1,

Figure 11.29 illustrates the complexity of shapes that can be represented with this technique. Notice part (a) of the figure, with knot sequence (0, 0, 0, 0, 1, 1, 1, 1): The curve interpolates the endpoints but not the two intermediate points, and is a Bézier curve. The other two curves also start and stop with triple knots. This causes the tangent vectors at the endpoints to be determined by the vectors P_0P_1 and $P_{m-1}P_m$, giving Bézier-like control to the curves at the start and stop points.

Interactive creation of nonuniform splines typically involves pointing at control points, with multiple control points indicated simply by successive selection of the same point. Figure 11.30 shows a way of specifying knot values interactively. Another way is to point directly at the curve with a multibutton mouse: A double click on one button can indicate a double control point; a double click on another button, a double knot.

Samsung/Google, EX1027, p. 513



2, 3), there is a triple knot, so curve segments Q_4 and Q_5 degenerate to points. The convex hulls containing Q_3 and Q_6 meet only at P_3, where the join point is forced to be located. The two curve segments share only control point P_3, with C_0 continuity. In (d), with knot sequence (0, 1, 1, 1, 1, 2), there is a quadruple knot, which causes a discontinuity in the curve because curve segments Q_3 and Q_7 have no control points in common.

11.2.5 Nonuniform, Rational Cubic Polynomial Curve Segments

General rational cubic curve segments are ratios of polynomials:

$$x(t) = \frac{X(t)}{W(t)}, \quad y(t) = \frac{Y(t)}{W(t)}, \quad z(t) = \frac{Z(t)}{W(t)}, \tag{11.45}$$

where $X(t)$, $Y(t)$, $Z(t)$, and $W(t)$ are all cubic polynomial curves whose control points are defined in homogeneous coordinates. We can also think of the curve as existing in homogeneous space as $Q(t) = [X(t) \quad Y(t) \quad Z(t) \quad W(t)]$. As always, moving from homogeneous space to 3-space involves dividing by $W(t)$. Any nonrational curve can be transformed to a rational curve by adding $W(t) = 1$ as a fourth element. In general, the

Samsung/Google, EX1027, p. 514

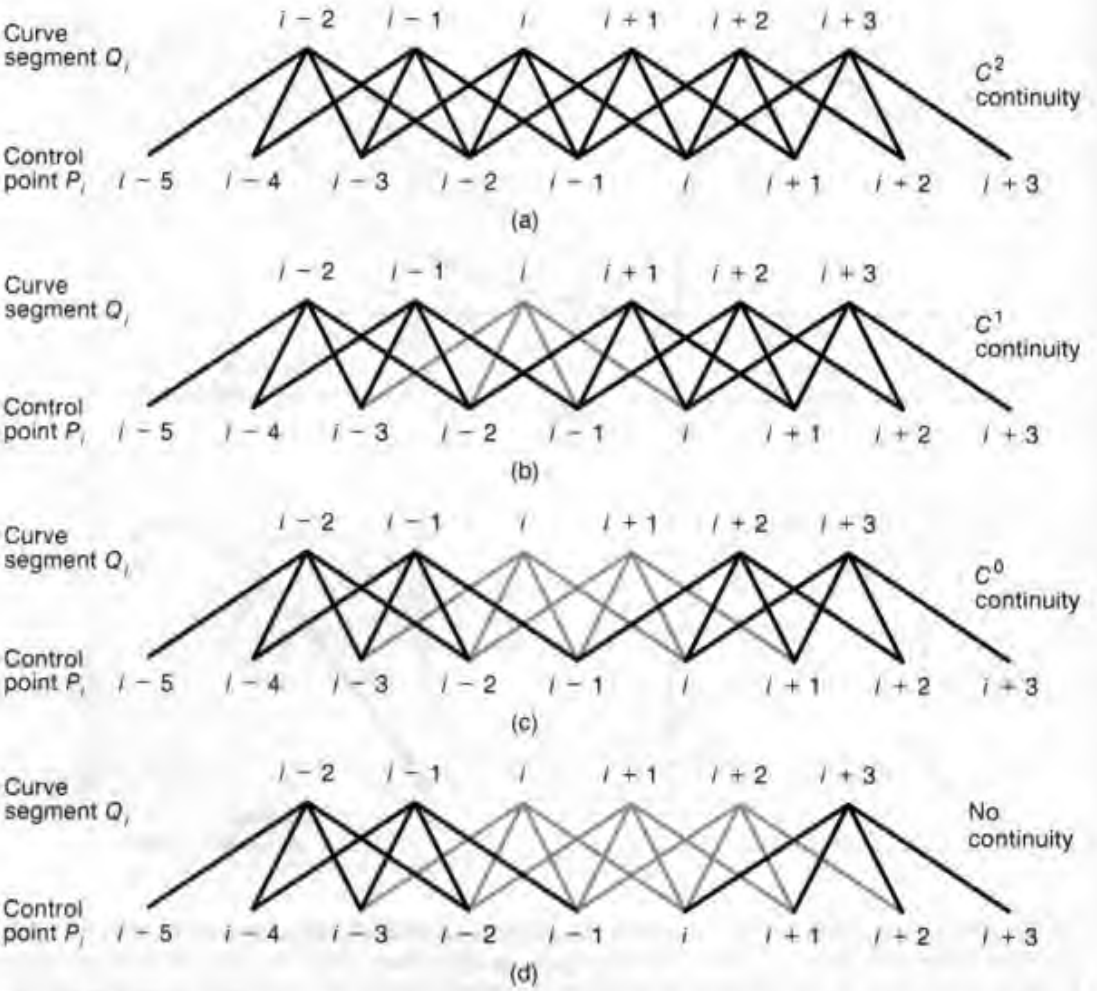


Fig. 11.28 The relationship among curve segments, control points, and multiple knots for nonuniform B-splines. Lines connect curve segments to their control points; gray lines are used for curve segments that do not appear because their knot interval is zero (i.e., the knot multiplicity is greater than one), causing them to have zero length. In (a), all knots are single. In (b), there is a double knot, so segment i is not drawn. In (c), there is a triple knot, so two segments are not drawn; thus, the single point, $i - 1$, is held in common between adjacent segments $i - 1$ and $i + 2$. In (d), with a quadruple knot, segments $i - 1$ and $i + 3$ have no points in common, causing the curve to be disconnected between points $i - 1$ and i.

polynomials in a rational curve can be Bézier, Hermite, or any other type. When they are B-splines, we have nonuniform rational B-splines, sometimes called *NURBS* [FORR80].

Rational curves are useful for two reasons. The first and most important reason is that they are invariant under rotation, scaling, translation and perspective transformations of the control points (nonrational curves are invariant under only rotation, scaling, and translation). This means that the perspective transformation needs to be applied to only the control

Samsung/Google, EX1027, p. 515

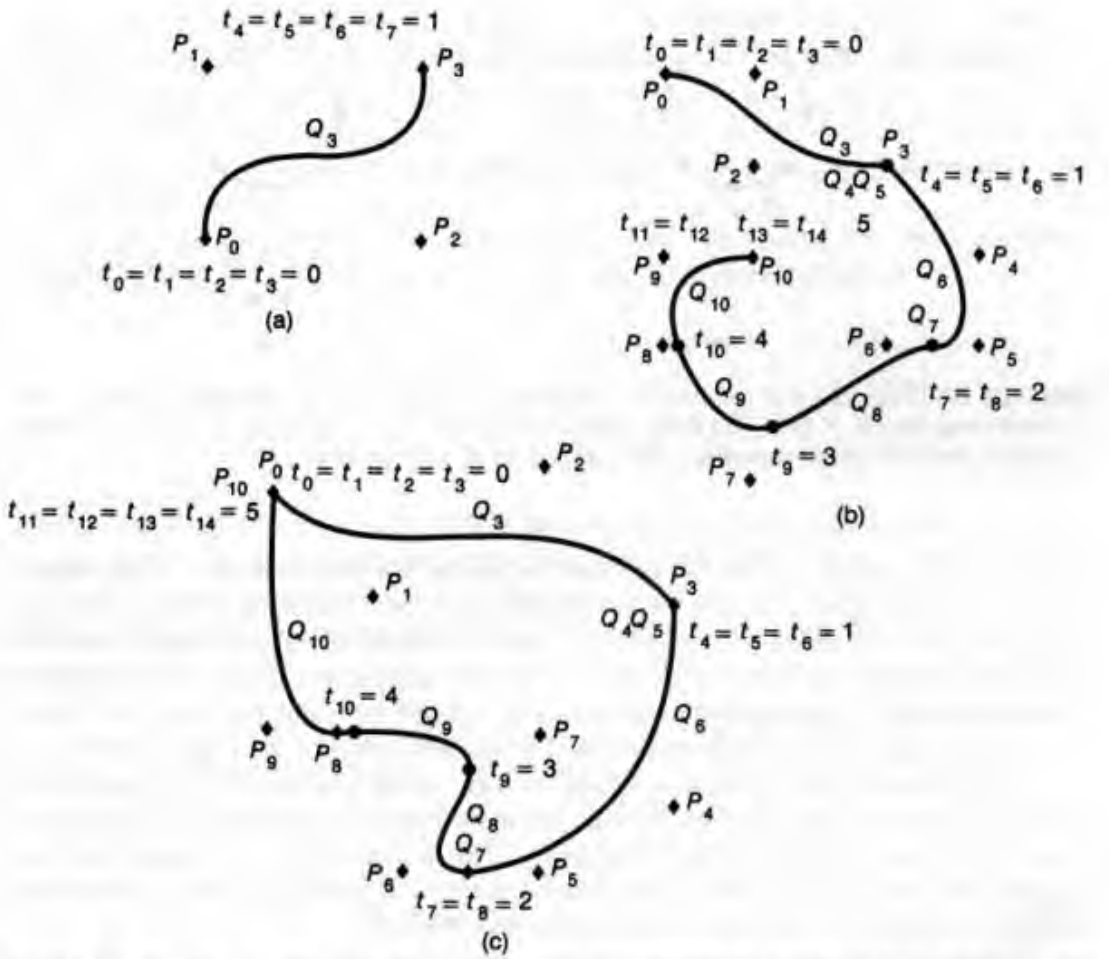


Fig. 11.29 Examples of shapes defined using nonrational B-splines and multiple knots. Part (a) is just a Bézier curve segment, with knot sequence (0, 0, 0, 0, 1, 1, 1, 1), and hence just one curve segment, Q_3. Parts (b) and (c) have the same knot sequence, (0, 0, 0, 0, 1, 1, 1, 2, 2, 3, 4, 5, 5, 5, 5) but different control points. Each curve has curve segments Q_3 to Q_{10}. Segments Q_4, Q_5, and Q_7 are located at multiple knots and have zero length.

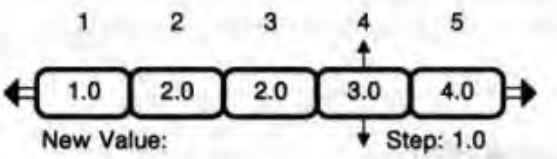


Fig. 11.30 An interaction technique developed by Carles Castellsagué for specifying knot values. The partial knot sequence is shown, and can be scrolled left and right with the horizontal arrows. One knot value, selected with the cursor, can be incremented up and down using the vertical arrows, in increments specified by value Step. The selected knot value can also be replaced with a new typed-in value.

Samsung/Google, EX1027, p. 516

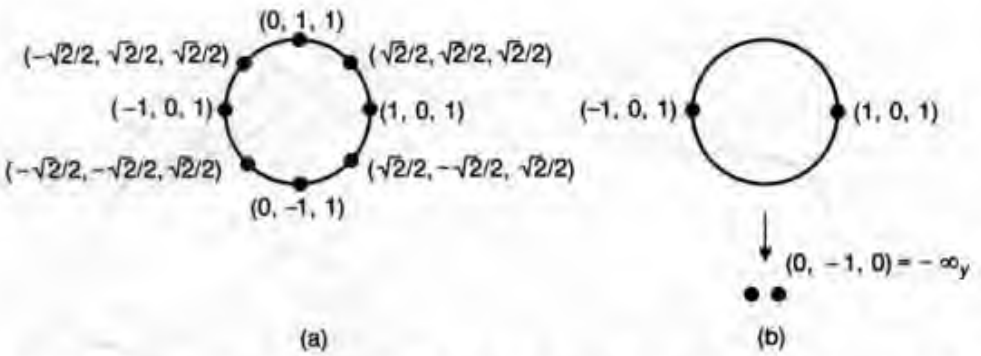


Fig. 11.31 The control points for defining a circle as a rational spline in 2D. Coordinates are (X, Y, W). The knot vector is (0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4), with the first and last control points repeated. The choice of P_0 is arbitrary.

points, which can then be used to generate the perspective transformation of the original curve. The alternative to converting a nonrational curve to a rational curve prior to a perspective transformation is first to generate points on the curve itself and then to apply the perspective transformation to *each* point, a far less efficient process. This is analogous to the observation that the perspective transformation of a sphere is not the same as a sphere whose center and radius are the transformed center and radius of the original sphere.

A second advantage of rational splines is that, unlike nonrationals, they can define precisely any of the conic sections. A conic can be only approximated with nonrationals, by using many control points close to the conic. This second property is useful in those applications, particularly CAD, where general curves and surfaces as well as conics are needed. Both types of entities can be defined with NURBS.

Defining conics requires only quadratic, not cubic, polynomials. Thus, the $B_{i,3}(t)$ blending functions from the recurrence Eq. (11.44) are used in the curve of the form

$$Q_i(t) = P_{i-2}B_{i-2,3}(t) + P_{i-1}B_{i-1,3}(t) + P_iB_{i,3}(t)$$
$$2 \le i \le m, \quad t_i \le t < t_{i+1}. \tag{11.46}$$

Two ways of creating a unit circle centered at the origin are shown in Fig. 11.31. Note that, with quadratic B-splines, a double knot causes a control point to be interpolated, and triple knots fix the starting and ending points of the curve on control points.

For further discussion of conics and NURBS, see [FAUX79; BÖHM84; TILL83].

11.2.6 Other Spline Forms

Very often, we have a series of positions and want a curve smoothly to interpolate (pass through) them. This might arise with a series of points read from a data tablet or mouse, or a series of 3D points through which a curve or camera path is to pass. The Catmull–Rom family of interpolating or approximating splines [CATM74a], also called *Overhauser*

Samsung/Google, EX1027, p. 517

splines [BREW77], are useful for this situation. One member of this spline family is able to interpolate the points P_1 to P_{m-1} from the sequence of points P_0 to P_m. In addition, the tangent vector at point P_i is parallel to the line connecting points P_{i-1} and P_{i+1}, as shown in Fig. 11.32. However, these splines do not posess the convex-hull property. The natural (interpolating) splines also interpolate points, but without the local control afforded by the Catmull–Rom splines.

Designating M_{CR} as the Catmull-Rom basis matrix and using the same geometry matrix G_{Bsi} of Eq. (11.32) as was used for B-splines, the representation is

$$Q(t) = T \cdot M_{CR} \cdot G_{Bsi} = \frac{1}{2} \cdot T \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix}. \tag{11.47}$$

A method for rapidly displaying Catmull–Rom curves is given in [BARR88b].

Another spline is the *uniformly shaped β-spline* [BARS88; BART87], which has two additional parameters, β_1 and β_2, to provide further control over shape. It uses the same geometry matrix G_{Bsi} as the B-spline, and has the convex hull property. The *β-spline basis matrix* M_β is

$$M_\beta = \frac{1}{\delta} \begin{bmatrix} -2\beta_1^3 & 2(\beta_2 + \beta_1^3 + \beta_1^2 + \beta_1) & -2(\beta_2 + \beta_1^2 + \beta_1 + 1) & 2 \\ 6\beta_1^3 & -3(\beta_2 + 2\beta_1^3 + 2\beta_1^2) & 3(\beta_2 + 2\beta_1^2) & 0 \\ -6\beta_1^3 & 6(\beta_1^3 - \beta_1) & 6\beta_1 & 0 \\ 2\beta_1^3 & \beta_2 + 4(\beta_1^2 + \beta_1) & 2 & 0 \end{bmatrix},$$

$$\delta = \beta_2 + 2\beta_1^3 + 4\beta_1^2 + 4\beta_1 + 2. \tag{11.48}$$

The first parameter, β_1, is called the *bias* parameter; the second parameter, β_2, is called the *tension* parameter. If $\beta_1 = 1$ and $\beta_2 = 0$, M_β reduces to M_{Bs} (Eq. (11.34)) for B-splines. As β_1 is increased beyond 1, the spline is "biased," or influenced more, by the tangent vector on the parameter-increasing side of the points, as shown in Fig. 11.33. For values of β_1 less than 1, the bias is in the other direction. As the tension parameter β_2 increases, the spline is pulled closer to the lines connecting the control points, as seen in Fig. 11.34.

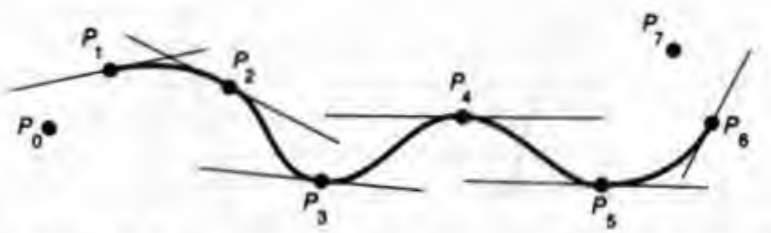


Fig. 11.32 A Catmull–Rom spline. The points are interpolated by the spline, which passes through each point in a direction parallel to the line between the adjacent points. The straight line segments indicate these directions.

Samsung/Google, EX1027, p. 518

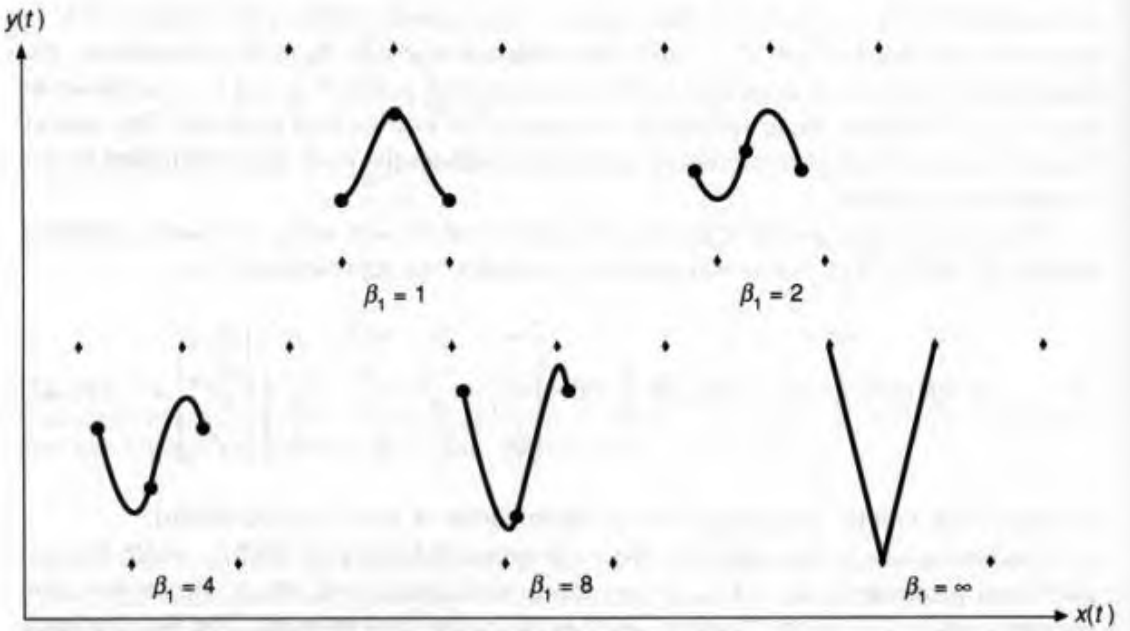


Fig. 11.33 Effect on a uniformly shaped β-spline of increasing the bias parameter β_1 from 1 to infinity.

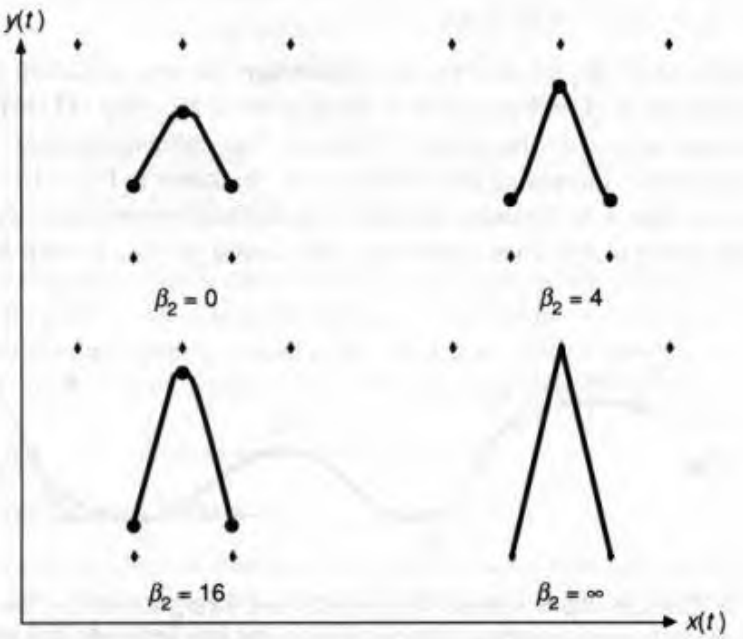


Fig. 11.34 Effect on a uniformly shaped β-spline of increasing the tension parameter β_2 from 0 to infinity.

Samsung/Google, EX1027, p. 519

These β-splines are called *uniformly shaped* because β_1 and β_2 apply uniformly, over all curve segments. This global effect of β_1 and β_2 violates the spirit of local control. *Continuously shaped β-splines* and *discretely shaped β splines* associate distinct values of β_1 and β_2 with each control point, providing a local, rather than a global, effect [BARS83; BART87].

Although β-splines provide more shape generality than do B-splines, they are G^2 but only C^0 continuous at the join points. This is not a drawback in geometric modeling, but can introduce a velocity discontinuity into an animation's motion path. In the special case $\beta_1 = 1$, the β-splines are C^1 continuous.

A variation on the Hermite form useful for controlling motion paths in animation was developed by Kochanek and Bartels [KOCH84]. The tangent vector R_i at point P_i is

$$R_i = \frac{(1 - a_i)(1 + b_i)(1 + c_i)}{2}(P_i - P_{i-1}) + \frac{(1 - a_i)(1 - b_i)(1 - c_i)}{2}(P_{i+1} - P_i), \tag{11.49}$$

and the tangent vector R_{i+1} at point P_{i+1} is

$$R_{i+1} + \frac{(1 - a_{i+1})(1 + b_{i+1})(1 - c_{i+1})}{2}(P_i - P_{i-1}) + \frac{(1 - a_{i+1})(1 - b_{i+1})(1 + c_{i+1})}{2}(P_{i+1} - P_i). \tag{11.50}$$

See Exercise 11.16 for how to find the basis matrix M_{KB}.

The parameters a_i, b_i, and c_i control different aspects of the path in the vicinity of point P_i: a_i controls how sharply the curve bends, b_i is comparable to β_1, the bias of β-splines, and c_i controls continuity at P_i. This last parameter is used to model the rapid change in direction of an object that bounces off another object.

11.2.7 Subdividing Curves

Suppose you have just created a connected series of curve segments to approximate a shape you are designing. You manipulate the control points, but you cannot quite get the shape you want. Probably, there are not enough control points to achieve the desired effect. There are two ways to increase the number of control points. One is the process of *degree elevation*: The degree of the splines is increased from 3 to 4 or more. This adjustment is sometimes necessary, especially if higher orders of continuity are needed, but is generally undesirable because of the additional inflection points allowed in a single curve and the additional computational time needed to evaluate the curve. In any case, the topic is beyond the scope of our discussion; for more details, see [FARI86].

The second, more useful way to increase the number of control points is to *subdivide* one or more of the curve segments into two segments. For instance, a Bézier curve segment with its four control points can be subdivided into two segments with a total of seven control points (the two new segments share a common point). The two new segments exactly match the one original segment until any of the control points are actually moved; then, one or both of the new segments no longer matches the original. For nonuniform B-splines, a more general process know as *refinement* can be used to add an arbitrary number of control points

Samsung/Google, EX1027, p. 520

to a curve. Another reason for subdividing is to display a curve or surface. We elaborate on this in Section 11.2.9, where we discuss tests for stopping the subdivision.

Given a Bézier curve $Q(t)$ defined by points P_1, P_2, P_3, and P_4, we want to find a left curve defined by points L_1, L_2, L_3, and L_4, and a right curve defined by points R_1, R_2, R_3, and R_4, such that the left curve is coincident with Q on the interval $0 \le t < \frac{1}{2}$ and the right curve is coincident with Q on the interval $\frac{1}{2} \le t < 1$. The subdivision can be accomplished using a geometric construction technique developed by de Casteljau [DECA59] to evaluate a Bézier curve for any value of t. The point on the curve for a parameter value of t is found by drawing the construction line L_2H so that it divides P_1P_2 and P_2P_3 in the ratio of t:$(1 - t)$, HR_3 so that it similarly divides P_2P_3 and P_3P_4, and L_3R_2 to likewise divide L_2H and HR_3. The point L_4 (which is also R_1) divides L_3R_2 by the same ratio and gives the point $Q(t)$. Figure 11.35 shows the construction for $t = \frac{1}{2}$, the case of interest.

All the points are easy to compute with an add and shift to divide by 2:

$$L_2 = (P_1 + P_2)/2,\ H = (P_2 + P_3)/2,\ L_3 = (L_2 + H)/2,\ R_3 = (P_3 + P_4)/2,$$
$$R_2 = (H + R_3)/2,\ L_4 = R_1 = (L_3 + R_2)/2. \tag{11.51}$$

These results can be reorganized into matrix form, so as to give the left Bézier division matrix D_B^L and the right Bézier division matrix D_B^R These matrices can be used to find the geometry matrices G_B^L of points for the left Bézier curve and G_B^R for the right Bézier curve:

$$G_B^L = D_B^L \cdot G_B = \frac{1}{8}\begin{bmatrix} 8 & 0 & 0 & 0 \\ 4 & 4 & 0 & 0 \\ 2 & 4 & 2 & 0 \\ 1 & 3 & 3 & 1 \end{bmatrix}\begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix},$$

$$G_B^R = D_B^R \cdot G_B = \frac{1}{8}\begin{bmatrix} 1 & 3 & 3 & 1 \\ 0 & 2 & 4 & 2 \\ 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 8 \end{bmatrix}\begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}. \tag{11.52}$$

Notice that each of the new control points L_i and R_i is a weighted sum of the points P_i, with the weights positive and summing to 1. Thus, each of the new control points is in the convex hull of the set of original control points. Therefore, the new control points are no farther from the curve $Q(t)$ than are the original control points, and in general are closer

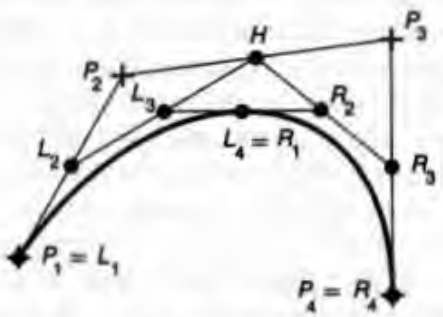


Fig. 11.35 The Bézier curve defined by the points P_i is divided at $t = \frac{1}{2}$ into a left curve defined by the points L_i and a right curve defined by the points R_i.

Samsung/Google, EX1027, p. 521

than the original points. This *variation-diminishing* property is true of all splines that have the convex-hull property. Also notice the symmetry between D_B^L and D_B^R, a consequence of the symmetry in the geometrical construction.

Dividing the Bézier curve at $t = \frac{1}{2}$ often gives the interactive user the control needed, but it might be better to allow the user to indicate a point on the curve at which a split is to occur. Given such a point, it is easy to find the approximate corresponding value of t. The splitting then proceeds as described previously, except that the construction lines are divided in the t:$(1 - t)$ ratio (see Exercise 11.22).

The corresponding matrixes D_{Bs}^L and D_{Bs}^R for B-spline subdivision are

$$G_{Bs_i}^L = D_{Bs}^L \cdot G_{Bs_i} = \frac{1}{8}\begin{bmatrix} 4 & 4 & 0 & 0 \\ 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 1 & 6 & 1 \end{bmatrix}\begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix},$$

$$G_{Bs_i}^R = D_{Bs}^R \cdot G_{Bs_i} = \frac{1}{8}\begin{bmatrix} 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 1 & 6 & 1 \\ 0 & 0 & 4 & 4 \end{bmatrix}\begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix}. \tag{11.53}$$

Careful examination of these two equations shows that the four control points of G_{Bs_i} are replaced by a total of five new control points, shared between $G_{Bs_i}^L$ and $G_{Bs_i}^R$. However, the spline segments on either side of the one being divided are still defined by some of the control points of G_{Bs_i}. Thus, changes to any of the five new control points or four old control points cause the B-spline to be no longer connected. This problem is avoided with nonuniform B-splines.

Subdividing nonuniform B-splines is not as is simple as specifying two splitting matrices, because there is no explicit basis matrix: The basis is defined recursively. The basic approach is to add knots to the knot sequence. Given a nonuniform B-spline defined by the control points $P_0, \ldots, P_n$ and a value of $t = t'$ at which to add a knot (and hence to add a control point), we want to find the new control points $Q_0, \ldots, Q_{n+1}$ that define the same curve. (The value t' might be determined via user interaction—see Exercise 11.21.) The value t′ satisfies $t_j < t' \le t_{j+1}$. Böhm [BÖHM80] has shown that the new control points are given by

$$\begin{aligned} Q_0 &= P_0 \\ Q_i &= (1 - a_i)P_{i-1} + a_iP_i, \quad 1 \le i \le n \\ Q_{n+1} &= P_n \end{aligned} \tag{11.54}$$

where a_i is given by:

$$\begin{aligned} a_i &= 1, && 1 \le i \le j - 3 \\ a_i &= \frac{t' - t_i}{t_{i+3} - t_i}, && j - 2 \le i \le j \quad \text{(division by zero is zero)} \\ a_i &= 0, && j + 1 \le i \le n. \end{aligned} \tag{11.55}$$

Samsung/Google, EX1027, p. 522

This algorithm is a special case of the more general Oslo algorithm [COHE80], which inserts any number of knots into a B-spline in a single set of computations. If more than a few knots are being inserted at once, the Oslo algorithm is more efficient than is the Böhm algorithm.

As an example of Böhm subdivision, consider the knot sequence (0, 0, 0, 0, 1, 1, 1, 1), the four x coordinates of the control point vector (5.0, 8.0, 9.0, 6.0), and a new knot at $t = 0.5$ (i.e., $n = 3$, $j = 3$). The a_i values defined by Eq. (11.55) are (0.5, 0.5, 0.5). Applying Eq. (11.54) we find that the x coordinates of the new Q_i control points are (5.0, 6.5, 8.5, 7.5, 6.0).

Notice that adding a knot causes two old control points to be replaced by three new control points. Furthermore, segments adjacent to the subdivided segment are defined only in terms of the new control points. Contrast this to the less attractive case of uniform–B-spline subdivision, in which four old control points are replaced by five new ones and adjacent segments are defined with the old control points, which are no longer used for the two new segments.

Hierarchical B-spline refinement is another way to gain finer control over the shape of a curve [FORS89]. Additional control points are added to local regions of a curve, and a hierarchical data structure is built relating the new control points to the original points. Use of a hierarchy allows further additional refinement. Storing the hierarchy rather than replacing the old control points with the larger number of new control points means that the original control points can continue to be used for coarse overall shaping of the curve, while at the same time the new control points can be used for control of details.

11.2.8 Conversion Between Representations

It is often necessary to convert from one representation to another. That is, given a curve represented by geometry vector G_1 and basis matrix M_1, we want to find the equivalent geometry matrix G_2 for basis matrix M_2 such that the two curves are identical: $T \cdot M_2 \cdot G_2 = T \cdot M_1 \cdot G_1$. This equality can be rewritten as $M_2 \cdot G_2 = M_1 \cdot G_1$. Solving for G_2, the unknown geometry matrix, we find

$$M_2^{-1} \cdot M_2 \cdot G_2 = M_2^{-1} \cdot M_1 \cdot G_1, \text{ or } G_2 = M_2^{-1} \cdot M_1 \cdot G_1 = M_{1,2} \cdot G_1. \quad (11.56)$$

That is, the matrix that converts a known geometry vector for representation 1 into the geometry vector for representation 2 is just $M_{1,2} = M_2^{-1} \cdot M_1$.

As an example, in converting from B-spline to Bézier form, the matrix $M_{\mathrm{Bs,B}}$ is

$$M_{\mathrm{Bs,B}} = M_{\mathrm{B}}^{-1} \cdot M_{\mathrm{Bs}} = \frac{1}{6}\begin{bmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix}. \quad (11.57)$$

The inverse is

$$M_{\mathrm{B,Bs}} = M_{\mathrm{Bs}}^{-1} \cdot M_{\mathrm{B}} = \begin{bmatrix} 6 & -7 & 2 & 0 \\ 0 & 2 & -1 & 0 \\ 0 & -1 & 2 & 0 \\ 0 & 2 & -7 & 6 \end{bmatrix}. \quad (11.58)$$

Samsung/Google, EX1027, p. 523

Nonuniform B-splines have no explicit basis matrix; recall that a nonuniform B-spline over four points with a knot sequence of (0, 0, 0, 0, 1, 1, 1, 1) is just a Bézier curve. Thus, a way to convert a nonuniform B-spline to any of the other forms is first to convert to Bézier form by adding multiple knots using either the Böhm or Oslo algorithms mentioned in Section 11.2.7, to make all knots have multiplicity 4. Then the Bézier form can be converted to any of the other forms that have basis matrices.

11.2.9 Drawing Curves

There are two basic ways to draw a parametric cubic. The first is by iterative evaluation of $x(t)$, $y(t)$, and $z(t)$ for incrementally spaced values of t, plotting lines between successive points. The second is by recursive subdivision that stops when the control points get sufficiently close to the curve itself.

In the introduction to Section 11.2, simple brute-force iterative evaluation display was described, where the polynomials were evaluated at successive values of t using Horner's rule. The cost was nine multiplies and 10 additions per 3D point. A much more efficient way repeatedly to evaluate a cubic polynomial is with *forward differences*. The forward difference $\Delta f(t)$ of a function $f(t)$ is

$$\Delta f(t) = f(t + \delta) - f(t), \delta > 0, \tag{11.59}$$

which can be rewritten as

$$f(t + \delta) = f(t) + \Delta f(t). \tag{11.60}$$

Rewriting Eq. (11.60) in iterative terms, we have

$$f_{n+1} = f_n + \Delta f_n, \tag{11.61}$$

where f is evaluated at equal intervals of size δ, so that $t_n = n\delta$ and $f_n = f(t_n)$.

For a third-degree polynomial,

$$f(t) = at^3 + bt^2 + ct + d = T \cdot C, \tag{11.62}$$

so the forward difference is

$$\begin{aligned} \Delta f(t) &= a(t + \delta)^3 + b(t + \delta)^2 + c(t + \delta) + d - (at^3 + bt^2 + ct + d) \\ &= 3at^2\delta + t(3a\delta^2 + 2b\delta) + a\delta^3 + b\delta^2 + c\delta. \end{aligned} \tag{11.63}$$

Thus, $\Delta f(t)$ is a second-degree polynomial. This is unfortunate, because evaluating Eq. (11.61) still involves evaluating $\Delta f(t)$, plus an addition. But forward differences can be applied to $\Delta f(t)$ to simplify its evaluation. From Eq. (11.61), we write

$$\Delta^2 f(t) = \Delta(\Delta f(t)) = \Delta f(t + \delta) - \Delta f(t). \tag{11.64}$$

Applying this to Eq. (11.63) gives

$$\Delta^2 f(t) = 6a\delta^2 t + 6a\delta^3 + 2b\delta^2. \tag{11.65}$$

This is now a first-degree equation in t. Rewriting (11.64) and using the index n, we obtain

$$\Delta^2 f_n = \Delta f_{n+1} - \Delta f_n. \tag{11.66}$$

Samsung/Google, EX1027, p. 524

Reorganizing and replacing n by $n - 1$ yields

$$\Delta f_n = \Delta f_{n-1} + \Delta^2 f_{n-1}. \tag{11.67}$$

Now, to evaluate Δf_n for use in Eq. (11.61), we evaluate $\Delta^2 f_{n-1}$ and add it to Δf_{n-1}. Because $\Delta^2 f_{n-1}$ is linear in t, this is less work than is evaluating Δf_n directly from the second-degree polynomial Eq. (11.63).

The process is repeated once more to avoid direct evaluation of Eq. (11.65) to find $\Delta^2 f(t)$:

$$\Delta^3 f(t) = \Delta(\Delta^2 f(t)) = \Delta^2 f(t + \delta) - \Delta^2 f(t) = 6a\delta^3. \tag{11.68}$$

The third forward difference is a constant, so further forward differences are not needed. Rewriting Eq. (11.68) with n, and with $\Delta^3 f_n$ as a constant yields

$$\Delta^2 f_{n+1} = \Delta^2 f_n + \Delta^3 f_n = \Delta^2 f_n + 6a\delta^3. \tag{11.69}$$

One further rewrite, replacing n with $n - 2$, completes the development:

$$\Delta^2 f_{n-1} = \Delta^2 f_{n-2} + 6a\delta^3. \tag{11.70}$$

This result can be used in Eq. (11.67) to calculate Δf_n, which is then used in Eq. (11.61) to find f_{n+1}.

To use the forward differences in an algorithm that iterates from $n = 0$ to $n\delta = 1$, we compute the initial conditions with Eqs. (11.62), (11.63), (11.65), and (11.68) for $t = 0$. They are

$$f_0 = d,\ \Delta f_0 = a\delta^3 + b\delta^2 + c\delta,\ \Delta^2 f_0 = 6a\delta^3 + 2b\delta^2,\ \Delta^3 f_0 = 6a\delta^3. \tag{11.71}$$

These initial condition calculations can be done by direct evaluation of the four equations. Note that, however, if we define the vector of initial differences as D, then

$$D = \begin{bmatrix} f_0 \\ \Delta f_0 \\ \Delta^2 f_0 \\ \Delta^3 f_0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ \delta^3 & \delta^2 & \delta & 0 \\ 6\delta^3 & 2\delta^2 & 0 & 0 \\ 6\delta^3 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}. \tag{11.72}$$

Rewriting, with the 4×4 matrix represented as $E(\delta)$, yields $D = E(\delta)C$. Because we are dealing with three functions, $x(t)$, $y(t)$, and $z(t)$, there are three corresponding sets of initial conditions, $D_x = E(\delta)C_x$, $D_y = E(\delta)C_y$, and $D_z = E(\delta)C_z$.

Based on this derivation, the algorithm for displaying a parametric cubic curve is given in Fig. 11.36. This procedure needs just nine additions and no multiplies per 3D point, and just a few adds and multiplies are used for initialization! This is considerably better than the 10 additions *and* nine multiplies needed for the simple brute-force iterative evaluation using Horner's rule. Notice, however, that error accumulation can be an issue with this algorithm, and sufficient fractional precision must be carried to avoid it. For instance, if $n = 64$ and integer arithmetic is used, then 16 bits of fractional precision are needed; if $n = 256$, 22 bits are needed [BART87].

Recursive subdivision is the second way to display a curve. Subdivision stops adaptively, when the curve segment in question is flat enough to be approximated by a line.

Samsung/Google, EX1027, p. 525

```
void DrawCurveFwdDif (
    int n,                          /* number of steps used to draw a curve */
    double x, double Δx, double Δ²x, double Δ³x,
    /* initial values for x(t) polynomial at t = 0, computed as D_x = E(δ)C_x. */
    double y, double Δy, double Δ²y, double Δ³y,
    /* initial values for y(t) polynomial at t = 0, computed as D_y = E(δ)C_y. */
    double z, double Δz, double Δ²z, double Δ³z
    /* initial values for z(t) polynomial at t = 0, computed as D_z = E(δ)C_z. */
    /* The step size δ used to calculate D_x, D_y, and D_z is 1/n . */
)

{
    int i;

    MoveAbs3 (x, y, z);             /* Go to start of curve */
    for (i = 0; i < n; i++) {
        x += Δx;  Δx += Δ²x;  Δ²x += Δ³x;
        y += Δy;  Δy += Δ²y;  Δ²y += Δ³y;
        z += Δz;  Δz += Δ²z;  Δ²z += Δ³z;
        LineAbs3 (x, y, z);         /* Draw a short line segment */
    }
} /* DrawCurveFwdDif */
```

Fig. 11.36 Procedure to display a parametric curve using forward differences.

Details vary for each type of curve, because the subdivision process is slightly different for each curve, as is the flatness test. The general algorithm is given in Fig. 11.37.

The Bézier representation is particularly appropriate for recursive subdivision display. Subdivision is fast, requiring only six shifts and six adds in Eq. (11.51). The test for "straightness" of a Bézier curve segment is also simple, being based on the convex hull formed by the four control points; see Fig. 11.38. If the larger of distances d_2 and d_3 is less than some threshold ε, then the curve is approximated by a straight line drawn between the endpoints of the curve segment. For the Bézier form, the endpoints are P_1 and P_4, the first and last control points. Were some other form used, the flatness test would be more complicated and the endpoints might have to be calculated. Thus, conversion to Bézier form is appropriate for display by recursive subdivision.

More detail on recursive-subdivision display can be found in [BARS85; BARS87; LANE80]. If the subdivision is done with integer arithmetic, less fractional precision is

```
void DrawCurveRecSub (curve, ε)
{
    if (Straight (curve, ε))        /* Test control points to be within ε of a line */
        DrawLine (curve);
    else {
        SubdivideCurve (curve, leftCurve, rightCurve);
        DrawCurveRecSub (leftCurve, ε);
        DrawCurveRecSub (rightCurve, ε);
    }
} /* DrawCurveRecSub */
```

Fig. 11.37 Procedure to display a curve by recursive subdivision. Straight is a procedure that returns **true** if the curve is sufficiently flat.

Samsung/Google, EX1027, p. 526

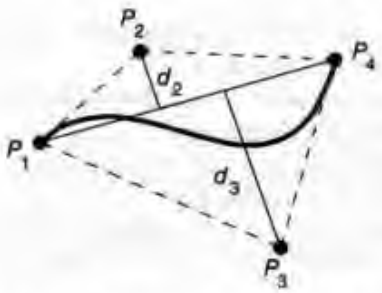


Fig. 11.38 Flatness test for a curve segment. If d_2 and d_3 are both less than some ε, the segment is declared to be flat and is approximated by the line segment P_1P_4.

needed than in the forward difference method. If at most eight levels of recursion are needed (a reasonable expectation), only 3 bits of fractional precision are necessary; if 16, 4 bits.

Recursive subdivision is attractive because it avoids unnecessary computation, whereas forward differencing uses a fixed subdivision. The forward-difference step size δ must be small enough that the portion of the curve with the smallest radius of curvature is approximated satisfactorily. For nearly straight portions of the curve, a much larger step size would be acceptable. [LANE80a] gives a method for calculating the step size δ to obtain a given maximum deviation from the true curve. On the other hand, recursive subdivision takes time to test for flatness. An alternative is to do recursive subdivision down to a fixed depth, avoiding the flatness test at the cost of some extra subdivision (see Exercise 11.30).

A hybrid approach, adaptive forward differencing [LIEN87; SHAN87; SHAN89], uses the best features of both the forward differencing and subdivision methods. The basic strategy is forward differencing, but an adaptive step size is used. Computationally efficient methods to double or halve the step size are used to keep it close to 1 pixel. This means that essentially no straight-line approximations are used between computed points.

11.2.10 Comparison of the Cubic Curves

The different types of parametric cubic curves can be compared by several different criteria, such as ease of interactive manipulation, degree of continuity at join points, generality, and speed of various computations using the representations. Of course, it is not necessary to choose a single representation, since it is possible to convert between all representations, as discussed in Section 11.2.8. For instance, nonuniform rational B-splines can be used as an internal representation, while the user might interactively manipulate Bézier control points or Hermite control points and tangent vectors. Some interactive graphics editors provide the user with Hermite curves while representing them internally in the Bézier form supported by PostScript [ADOB85a]. In general, the user of an interactive CAD system may be given several choices, such as Hermite, Bézier, uniform B-splines, and nonuniform B-splines. The nonuniform rational B-spline representation is likely to be used interally, because it is the most general.

Table 11.2 compares most of the curve forms mentioned in this section. Ease of interactive manipulation is not included explicitly in the table, because the latter is quite

Samsung/Google, EX1027, p. 527

TABLE 11.2 COMPARISON OF SEVEN DIFFERENT FORMS OF PARAMETRIC CUBIC CURVES

	Hermite	Bézier	Uniform B-spline	Uniformly shaped β-spline	Nonuniform B-spline	Catmull–Rom	Kochanek–Bartels
Convex hull defined by control points	N/A	Yes	Yes	Yes	Yes	No	No
Interpolates some control points	Yes	Yes	No	No	No	Yes	Yes
Interpolates all control points	Yes	No	No	No	No	Yes	Yes
Ease of subdivision	Good	Best	Avg	Avg	High	Avg	Avg
Continuities inherent in representation	C^0 G^0	C^0 G^0	C^2 G^2	C^0 G^2	C^2 G^2	C^1 G^1	C^1 G^1
Continuities easily achieved	C^1 G^1	C^1 G^1	C^2 G^{2*}	C^1 G^{2*}	C^2 G^{2*}	C^1 G^1	C^1 G^1
Number of parameters controlling a curve segment	4	4	4	6†	5	4	7

*Except for special case discussed in Section 11.2.
†Four of the parameters are local to each segment, two are global to the entire curve.

Samsung/Google, EX1027, p. 528

application specific. "Number of parameters controlling a curve segment" is the four geometrical constraints plus other parameters, such as knot spacing for nonuniform splines, β_1 and β_2 for β-splines, or a, b, or c for the Kochanek–Bartels case. "Continuity easily achieved" refers to constraints such as forcing control points to be collinear to allow G^1 continuity. Because C^n continuity is more restrictive than G^n, any form that can attain C^n can by definition also attain at least G^n.

When only geometric continuity is required, as is often the case for CAD, the choice is narrowed to the various types of splines, all of which can achieve both G^1 and G^2 continuity. Of the three types of splines in the table, uniform B-splines are the most limiting. The possibility of multiple knots afforded by nonuniform B-splines gives more shape control to the user, as does the use of the β_1 and β_2 shape parameters of the β-splines. Of course, a good user interface that allows the user to exploit this power easily is important.

To interpolate the digitized points of a camera path or shape contour, Catmull–Rom or Kochanek–Bartels splines are preferable. When a combination of interpolation and tangent vector control is desired, the Bézier or Hermite form is best.

It is customary to provide the user with the ability to drag control points or tangent vectors interactively, continually displaying the updated spline. Figure 11.23 shows such a sequence for B-splines. One of the disadvantages of B-splines in some applications is that the control points are not on the spline itself. It is possible, however, not to display the control points, allowing the user instead to interact with the knots (which must be marked so they can be selected). When the user selects a knot and moves it by some $(\Delta x, \Delta y)$, the control point weighted most heavily in determining the position of the join point is also moved by $(\Delta x, \Delta y)$. The join does not move the full $(\Delta x, \Delta y)$, because it is a weighted sum of several control points, only one of which was moved. Therefore, the cursor is repositioned on the join. This process is repeated in a loop until the user stops dragging.

11.3 PARAMETRIC BICUBIC SURFACES

Parametric bicubic surfaces are a generalization of parametric cubic curves. Recall the general form of the parametric cubic curve $Q(t) = T \cdot M \cdot G$, where G, the geometry vector, is a constant. First, for notational convenience, we replace t with s, giving $Q(s) = S \cdot M \cdot G$. If we now allow the points in G to vary in 3D along some path that is parameterized on t, we have

$$Q(s, t) = S \cdot M \cdot G(t) = S \cdot M \cdot \begin{bmatrix} G_1(t) \\ G_2(t) \\ G_3(t) \\ G_4(t) \end{bmatrix}. \tag{11.73}$$

Now, for a fixed t_1, $Q(s, t_1)$ is a curve because $G(t_1)$ is constant. Allowing t to take on some new value—say, t_2—where $t_2 - t_1$ is very small, $Q(s, t)$ is a slightly different curve. Repeating this for arbitrarily many other values of t_2 between 0 and 1, an entire family of curves is defined, each arbitrarily close to another curve. The set of all such curves defines a surface. If the $G_i(t)$ are themselves cubics, the surface is said to be a *parametric bicubic surface*.

Samsung/Google, EX1027, p. 529

Continuing with the case that the $G_i(t)$ are cubics, each can be represented as $G_i(t) = T \cdot M \cdot \boldsymbol{G}_i$, where $\boldsymbol{G}_i = [\boldsymbol{g}_{i1} \quad \boldsymbol{g}_{i2} \quad \boldsymbol{g}_{i3} \quad \boldsymbol{g}_{i4}]^T$ (the $\boldsymbol{G}$ and $\boldsymbol{g}$ are used to distinguish from the G used for the curve). Hence, $\boldsymbol{g}_{i1}$ is the first element of the geometry vector for curve $G_i(t)$, and so on.

Now let us transpose the equation $G_i(t) = T \cdot M \cdot \boldsymbol{G}_i$, using the identity $(A \cdot B \cdot C)^T = C^T \cdot B^T \cdot A^T$. The result is $G_i(t) = \boldsymbol{G}_i^T \cdot M^T \cdot T^T = [\boldsymbol{g}_{i1} \quad \boldsymbol{g}_{i2} \quad \boldsymbol{g}_{i3} \quad \boldsymbol{g}_{i4}] \cdot M^T \cdot T^T$. If we now substitute this result in Eq. (11.73) for each of the four points, we have

$$Q(s, t) = S \cdot M \cdot \begin{bmatrix} \boldsymbol{g}_{11} & \boldsymbol{g}_{12} & \boldsymbol{g}_{13} & \boldsymbol{g}_{14} \\ \boldsymbol{g}_{21} & \boldsymbol{g}_{22} & \boldsymbol{g}_{23} & \boldsymbol{g}_{24} \\ \boldsymbol{g}_{31} & \boldsymbol{g}_{32} & \boldsymbol{g}_{33} & \boldsymbol{g}_{34} \\ \boldsymbol{g}_{41} & \boldsymbol{g}_{42} & \boldsymbol{g}_{43} & \boldsymbol{g}_{44} \end{bmatrix} \cdot M^T \cdot T^T \tag{11.74}$$

or

$$Q(s, t) = S \cdot M \cdot \boldsymbol{G} \cdot M^T \cdot T^T, \; 0 \le s, t \le 1. \tag{11.75}$$

Written separately for each of x, y, and z, the form is

$$\begin{aligned} x(s, t) &= S \cdot M \cdot \boldsymbol{G}_x \cdot M^T \cdot T^T, \\ y(s, t) &= S \cdot M \cdot \boldsymbol{G}_y \cdot M^T \cdot T^T, \\ z(s, t) &= S \cdot M \cdot \boldsymbol{G}_z \cdot M^T \cdot T^T, \end{aligned} \tag{11.76}$$

Given this general form, we now move on to examine specific ways to specify surfaces using different geometry matrixes.

11.3.1 Hermite Surfaces

Hermite surfaces are completely defined by a 4×4 geometry matrix $\boldsymbol{G}_H$. Derivation of $\boldsymbol{G}_H$ follows the same approach used to find Eq. (11.75). We further elaborate the derivation here, applying it just to $x(s, t)$. First, we replace t by s in Eq. (11.13), to get $x(s) = S \cdot M_H \cdot G_{H_x}$. Rewriting this further so that the Hermite geometry vector G_{H_x} is not constant, but is rather a function of t, we obtain

$$x(s, t) = S \cdot M_H \cdot G_{H_x}(t) = S \cdot M_H \begin{bmatrix} P_{1_x}(t) \\ P_{4_x}(t) \\ R_{1_x}(t) \\ R_{4_x}(t) \end{bmatrix}. \tag{11.77}$$

The functions $P_{1_x}(t)$ and $P_{4_x}(t)$ define the x components of the starting and ending points for the curve in parameter s. Similarly, $R_{1_x}(t)$ and $R_{4_x}(t)$ are the tangent vectors at these points. For any specific value of t, there are two specific endpoints and tangent vectors. Figure 11.39 shows $P_1(t)$, $P_4(t)$, and the cubic in s that is defined when $t = 0.0, 0.2, 0.4, 0.6, 0.8$, and 1.0. The surface patch is essentially a cubic interpolation between $P_1(t) = Q(0, t)$ and $P_4(t) = Q(1, t)$ or, alternatively, between $Q(s, 0)$ and $Q(s, 1)$.

In the special case that the four interpolants $Q(0, t)$, $Q(1, t)$, $Q(s, 0)$, and $Q(s, 1)$ are straight lines, the result is a *ruled surface*. If the interpolants are also coplanar, then the surface is a four-sided planar polygon.

Samsung/Google, EX1027, p. 530

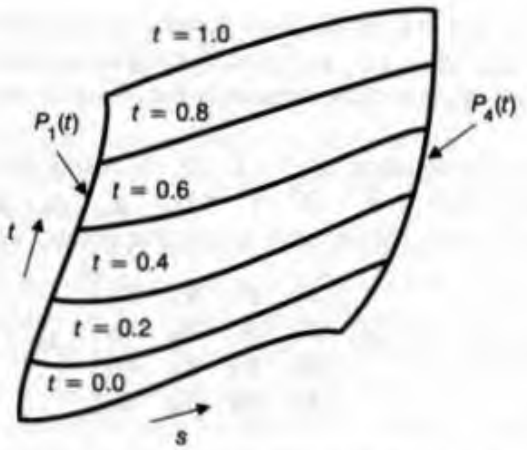


Fig. 11.39 Lines of constant parameter values on a bicubic surface: $P_1(t)$ is at $s = 0$, $P_4(t)$ is at $s = 1$.

Continuing with the derivation, let each of $P_1(t)$, $P_4(t)$, $R_1(t)$, and $R_4(t)$ be represented in Hermite form as

$$P_{1_x}(t) = T \cdot M_H \begin{bmatrix} g_{11} \\ g_{12} \\ g_{13} \\ g_{14} \end{bmatrix}_x, \qquad P_{4_x}(t) = T \cdot M_H \begin{bmatrix} g_{21} \\ g_{22} \\ g_{23} \\ g_{24} \end{bmatrix}_x, \tag{11.78}$$

$$R_{1_x}(t) = T \cdot M_H \begin{bmatrix} g_{31} \\ g_{32} \\ g_{33} \\ g_{34} \end{bmatrix}_x, \qquad R_{4_x}(t) = T \cdot M_H \begin{bmatrix} g_{41} \\ g_{42} \\ g_{43} \\ g_{44} \end{bmatrix}_x.$$

These four cubics can be rewritten together as a single equation:

$$[P_1(t) \quad P_4(t) \quad R_1(t) \quad R_4(t)]_x = T \cdot M_H \cdot G_{H_x}^T, \tag{11.79}$$

where

$$G_{H_x} = \begin{bmatrix} g_{11} & g_{12} & g_{13} & g_{14} \\ g_{21} & g_{22} & g_{23} & g_{24} \\ g_{31} & g_{32} & g_{33} & g_{34} \\ g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix}_x. \tag{11.80}$$

Transposing both sides of Eq. (11.79) results in

$$\begin{bmatrix} P_1(t) \\ P_4(t) \\ R_1(t) \\ R_4(t) \end{bmatrix}_x = \begin{bmatrix} g_{11} & g_{12} & g_{13} & g_{14} \\ g_{21} & g_{22} & g_{23} & g_{24} \\ g_{31} & g_{32} & g_{33} & g_{34} \\ g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix}_x M_H^T \cdot T^T = G_{H_x} \cdot M_H^T \cdot T^T. \tag{11.81}$$

Samsung/Google, EX1027, p. 531

Substituting Eq. (11.81) into Eq. (11.77) yields

$$x(s, t) = S \cdot M_H \cdot G_{H_x} \cdot M_H^T \cdot T^T; \tag{11.82}$$

similarly,

$$y(s, t) = S \cdot M_H \cdot G_{H_y} \cdot M_H^T \cdot T^T, \; z(s, t) = S \cdot M_H \cdot G_{H_z} \cdot M_H^T \cdot T^T. \tag{11.83}$$

The three 4×4 matrixes G_{H_x}, G_{H_y}, and G_{H_z} play the same role for Hermite surfaces as did the single matrix G_H for curves. The meanings of the 16 elements of G_H can be understood by relating them back to Eqs. (11.77) and (11.78). The element g_{11_x} is $x(0, 0)$ because it is the starting point for $P_{1_x}(t)$, which is in turn the starting point for $x(s, 0)$. Similarly, g_{12_x} is $x(0, 1)$ because it is the ending point of $P_{1_x}(t)$, which is in turn the starting point for $x(s, 1)$. Furthermore, g_{13_x} is $\partial x/\partial t(0, 0)$ because it is the starting tangent vector for $P_{1_x}(t)$, and g_{33_x} is $\partial^2 x/\partial s \partial t(0, 0)$ because it is the starting tangent vector of $R_{1_x}(t)$, which in turn is the starting slope of $x(s, 0)$.

Using these interpretations, we can write G_{H_x} as

$$\mathbf{G}_{H_x} = \begin{bmatrix} x(0,0) & x(0,1) & \frac{\partial}{\partial t}x(0,0) & \frac{\partial}{\partial t}x(0,1) \\ x(1,0) & x(1,1) & \frac{\partial}{\partial t}x(1,0) & \frac{\partial}{\partial t}x(1,1) \\ \frac{\partial}{\partial s}x(0,0) & \frac{\partial}{\partial s}x(0,1) & \frac{\partial^2}{\partial s \partial t}x(0,0) & \frac{\partial^2}{\partial s \partial t}x(0,1) \\ \frac{\partial}{\partial s}x(1,0) & \frac{\partial}{\partial s}x(1,1) & \frac{\partial^2}{\partial s \partial t}x(1,0) & \frac{\partial^2}{\partial s \partial t}x(1,1) \end{bmatrix}. \tag{11.84}$$

The upper-left 2×2 portion of G_{H_x} contains the x coordinates of the four corners of the patch. The upper-right and lower-left 2×2 areas give the x coordinates of the tangent vectors along each parametric direction of the patch. The lower-right 2×2 portion has at its corners the partial derivatives with respect to both s and t. These partials are often called the *twists*, because the greater they are, the greater the corkscrewlike twist at the corners. Figure 11.40 shows a patch whose corners are labeled to indicate these parameters.

This Hermite form of bicubic patches is an alternative way to express a restricted form of the *Coons patch* [COON67]. These more general patches permit boundary curves and slopes to be any curves. (The Coons patch was developed by the late Steven A. Coons [HERZ80], an early pioneer in CAD and computer graphics after whom SIGGRAPH's prestigious *Steven A. Coons Award for Outstanding Contributions to Computer Graphics* is named.) When the four twist vectors in the Hermite form are all zero, the patches are also called Ferguson surfaces after another early developer of surface representations [FERG64; FAUX79].

Just as the Hermite cubic permits C^1 and G^1 continuity from one curve segment to the next, so too the Hermite bicubic permits C^1 and G^1 continuity from one patch to the next. First, to have C^0 continuity at an edge, the matching curves of the two patches must be identical, which means the control points for the two surfaces must be identical along the edge. The necessary conditions for C^1 continuity are that the control points along the edge and the tangent and twist vectors across the edge be equal. For G^1 continuity, the tangent

Samsung/Google, EX1027, p. 532

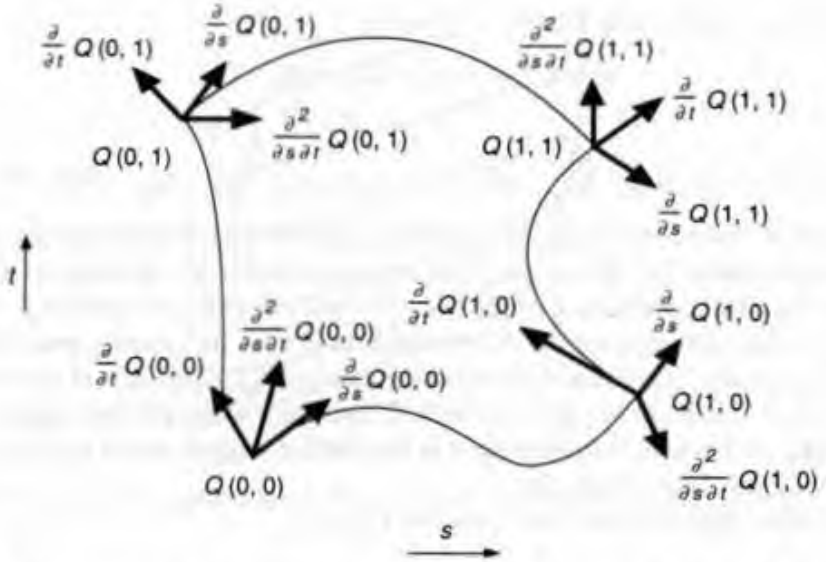


Fig. 11.40 Components of the geometry matrix for a Hermite surface. Each vector is a 3-tuple, the x component of which is given by Eq. (11.84).

vector requirement is relaxed so that the vectors must be in the same direction, but do not need to have the same magnitude. If the common edge for patch 1 is at $s = 1$ and that for patch 2 is at $s = 0$, as in Fig. 11.41, then the values in some rows of the geometry matrices for the two patches must reflect the G^1 conditions, as indicated here:

$$\text{Patch 1} \qquad \text{Patch 2}$$

$$\begin{bmatrix} - & - & - & - \\ g_{21} & g_{22} & g_{23} & g_{24} \\ - & - & - & - \\ g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix} \quad \begin{bmatrix} g_{21} & g_{22} & g_{23} & g_{24} \\ - & - & - & - \\ kg_{41} & kg_{42} & kg_{43} & kg_{44} \\ - & - & - & - \end{bmatrix} \quad k > 0. \tag{11.85}$$

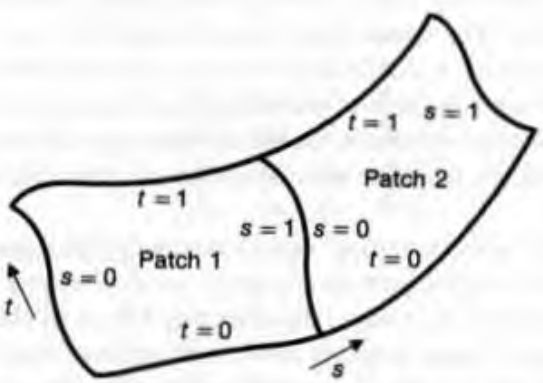


Fig. 11.41 Two joined surface patches.

Samsung/Google, EX1027, p. 533

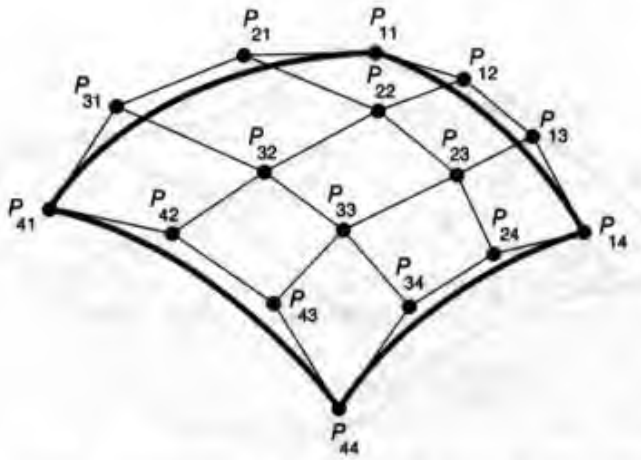


Fig. 11.42 Sixteen control points for a Bézier bicubic patch.

Entries marked with a dash can have any value. If four patches joined at a common corner and along the edges emanating from that corner are to have G^1 continuity, then the relationships are more complex; see Exercise 11.25.

11.3.2 Bézier Surfaces

The Bézier bicubic formulation can be derived in exactly the same way as the Hermite cubic. The results are

$$\begin{aligned} x(s, t) &= S \cdot M_B \cdot G_{B_x} \cdot M_B^T \cdot T^T, \\ y(s, t) &= S \cdot M_B \cdot G_{B_y} \cdot M_B^T \cdot T^T, \\ z(s, t) &= S \cdot M_B \cdot G_{B_z} \cdot M_B^T \cdot T^T. \end{aligned} \tag{11.86}$$

The Bézier geometry matrix $\boldsymbol{G}$ consists of 16 control points, as shown in Fig. 11.42. Bézier surfaces are attractive in interactive design for the same reason as Bézier curves are: Some of the control points interpolate the surface, giving convenient precise control, whereas tangent vectors also can be controlled explicitly. When Bézier surfaces are used as an internal representation, their convex-hull property and easy subdivision are attractive.

C^0 and G^0 continuity across patch edges is created by making the four common control points equal. G^1 continuity occurs when the two sets of four control points on either side of the edge are collinear with the points on the edge. In Fig. 11.43, the following sets of control points are collinear and define four line segments whose lengths all have the same ratio k: (P_{13}, P_{14}, P_{15}), (P_{23}, P_{24}, P_{25}), (P_{33}, P_{34}, P_{35}), and (P_{43}, P_{44}, P_{45}).

An alternative way to maintain interpatch continuity, described in more detail in [FAUX79; BEZI70], requires that, at each corner of an edge across which continuity is desired, the corner control points and the control points immediately adjacent to the corner be coplanar.

Samsung/Google, EX1027, p. 534

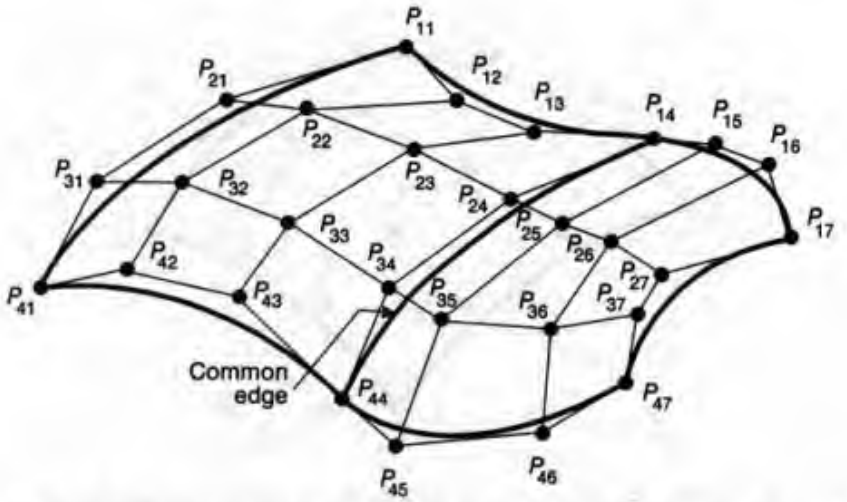


Fig. 11.43 Two Bézier patches joined along the edge P_{14}, P_{24}, P_{34}, and P_{44}.

11.3.3 B-Spline Surfaces

B-spline patches are represented as

$$\begin{aligned} x(s, t) &= S \cdot M_{Bs} \cdot \boldsymbol{G}_{Bs_x} \cdot M_{Bs}{}^{T} \cdot T^{T}, \\ y(s, t) &= S \cdot M_{Bs} \cdot \boldsymbol{G}_{Bs_y} \cdot M_{Bs}{}^{T} \cdot T^{T}, \\ z(s, t) &= S \cdot M_{Bs} \cdot \boldsymbol{G}_{Bs_z} \cdot M_{Bs}{}^{T} \cdot T^{T}. \end{aligned} \tag{11.87}$$

C^2 continuity across boundaries is automatic with B-splines; no special arrangements of control points are needed except to avoid duplicate control points, which create discontinuities.

Bicubic nonuniform and rational B-spline surfaces and other rational surfaces are similarly analogous to their cubic counterparts. All the techniques for subdivision and display carry over directly to the bicubic case.

11.3.4 Normals to Surfaces

The normal to a bicubic surface, needed for shading (Chapter 16), for performing interference detection in robotics, for calculating offsets for numerically controlled machining, and for doing other calculations, is easy to find. From Eq. (11.75), the s tangent vector of the surface $Q(s, t)$ is

$$\begin{aligned} \frac{\partial}{\partial s} Q(s, t) &= \frac{\partial}{\partial s} (S \cdot M \cdot G \cdot M^{T} \cdot T^{T}) = \frac{\partial}{\partial s} (S) \cdot M \cdot G \cdot M^{T} \cdot T^{T} \\ &= [3s^2 \quad 2s \quad 1 \quad 0] \cdot M \cdot G \cdot M^{T} \cdot T^{T}, \end{aligned} \tag{11.88}$$

Samsung/Google, EX1027, p. 535

and the t tangent vector is

$$\frac{\partial}{\partial t}Q(s, t) = \frac{\partial}{\partial t}(S \cdot M \cdot G \cdot M^{\mathrm{T}} \cdot T^{\mathrm{T}}) = S \cdot M \cdot G \cdot M^{\mathrm{T}} \cdot \frac{\partial}{\partial t}(T^{\mathrm{T}})$$

$$= S \cdot M \cdot G \cdot M^{\mathrm{T}} \cdot [3t^2 \quad 2t \quad 1 \quad 0]^{\mathrm{T}}. \tag{11.89}$$

Both tangent vectors are parallel to the surface at the point (s, t) and, therefore, their cross-product is perpendicular to the surface. Notice that, if both tangent vectors are zero, the cross-product is zero, and there is no meaningful surface normal. Recall that a tangent vector can go to zero at join points that have C^1 but not G^1 continuity.

Each of the tangent vectors is of course a 3-tuple, because Eq. (11.75) represents the x, y, and z components of the bicubic. With the notation x_s for the x component of the s tangent vector, y_s for the y component, and z_s for the z component, the normal is

$$\frac{\partial}{\partial s}Q(s, t) \times \frac{\partial}{\partial t}Q(s, t) = [y_s z_t - y_t z_s \quad z_s x_t - z_t x_s \quad x_s y_t - x_t y_s]. \tag{11.90}$$

The surface normal is a biquintic (two-variable, fifth-degree) polynomial and hence is fairly expensive to compute. [SCHW82] gives a bicubic approximation that is satisfactory as long as the patch itself is relatively smooth.

11.3.5 Displaying Bicubic Surfaces

Like curves, surfaces can be displayed either by iterative evaluation of the bicubic polynomials or by subdivision, which is essentially an adaptive evaluation of the bicubic polynomials. We consider first iterative evaluation, and then subdivision.

Iterative evaluation is best suited for displaying bicubic patches in the style of Fig. 11.44. Each of the curves of constant s and constant t on the surface is itself a cubic, so display of each of the curves is straightforward, as in Fig. 11.45.

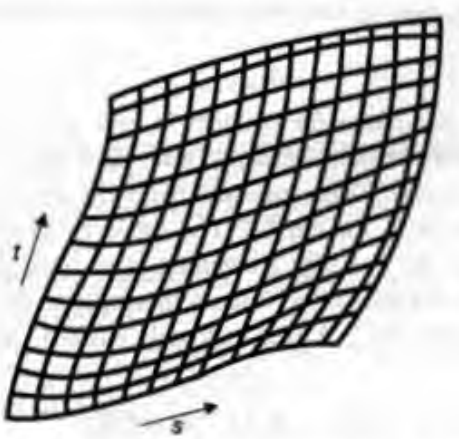


Fig. 11.44 A single surface patch displayed as curves of constant s and constant t.

Samsung/Google, EX1027, p. 536

```
typedef double Coeffs[4][4][3];
void DrawSurface (
        Coeffs coefficients,          /* Coefficients for Q(s,t) */
        int n_s,                      /* No. of curves of constant s to draw, typically 5–10 */
        int n_t,                      /* No. of curves of constant t to draw, typically 5–10 */
        int n)                        /* No. of steps used to draw each curve, typically 20–100 */
{
    double δ = 1.0 / n;               /* Step size to use in drawing each curve */
    double δ_s = 1.0 / (n_s−1);       /* Step size in s to increment to next curve of constant t */
    double δ_t = 1.0 / (n_t−1);       /* Step size in t to increment to next curve of constant s */
    int i, j; double s, t;
    /* Draw n_s curves of constant s, for s = 0, δ_s, 2δ_s, ... 1 */
    for (i = 0, s = 0.0; i < n_s; i++, s += δ_s) {
        /* Draw a curve of constant s, varying t from 0 to 1. */
        /* X, Y, and Z are functions to evaluate the bicubics. */
        MoveAbs3 (X (s, 0.0), Y (s, 0.0), Z (s, 0.0));
        for (j = 1, t = δ; j < n; j++, t += δ) {
            /* n−1 steps are used as t varies from δ to 1 for each curve. */
            LineAbs3 (X (s, t), Y (s, t), Z (s, t));
        }
    }
    /* Draw n_t curves of constant t, for t = 0, δ_t, 2δ_t, ... 1 */
    for (i = 0, t = 0.0; i < n_t; i++, t += δ_t) {
        /* Draw a curve of constant t, varying s from 0 to 1. */
        MoveAbs3 (X (0.0, t), Y (0.0, t), Z (0.0, t));
        for (j = 1, s = δ; j < n; j++, s += δ) {
            /* n−1 steps are used as s varies from δ to 1 for each curve. */
            LineAbs3 (X (s, t), Y (s, t), Z (s, t));
        }
    }
} /* DrawSurface */
```

Fig. 11.45 Procedure to display bicubic patch as a grid. Procedures X(*s*, *t*), Y(*s*, *t*), and Z(*s*, *t*) evaluate the surface using the coefficient matrix *coefficients*.

Brute-force iterative evaluation for surfaces is even more expensive than for curves, because the surface equations must be evaluated about $2/\delta^2$ times. For $\delta = 0.1$, this value is 200; for $\delta = 0.01$, it is 20,000. These numbers make the alternative, forward differencing even more attractive than it is for curves. The basic forward-differencing method was developed in Section 11.2.9. The step remaining is to understand how to calculate the initial forward differences for curve $i + 1$ from the forward differences for curve i.

The derivation in Section 11.2.9 used to find $D = E(\delta) \cdot C$ for curves can be used to find

$$DD_x = E(\delta_s) \cdot A_x \cdot E(\delta_t)^T \tag{11.91}$$

where δ_s is the step size in s, δ_t is the step size in t, and A_x is the 4×4 matrix of coefficients for $x(s, t)$. The 4×4 matrix DD_x has as its first row the values $x(0, 0)$, $\Delta_t x(0, 0)$, $\Delta_t^2 x(0, 0)$,

Samsung/Google, EX1027, p. 537

and $\Delta_t^3 x(0, 0)$ (the notation Δ_t means forward difference on t, as opposed to s). Thus, the first row can be used to calculate $x(0, t)$ in increments of δ_t.

After $x(0, t)$ has been computed, how can $x(\delta_s, t)$ be computed, to draw another curve of constant s? (This is the step of calculating the initial forward differences for curve $i + 1$ from the forward differences for curve i.) The other rows of DD_x are the first, second, and third forward differences on s of the first row's forward differences. Therefore, applying the following equations to the rows of DD_x

$$\begin{aligned} &\text{row 1} := \text{row 1} + \text{row 2} \\ &\text{row 2} := \text{row 2} + \text{row 3} \\ &\text{row 3} := \text{row 3} + \text{row 4} \end{aligned} \tag{11.92}$$

```
typedef double Coeffs[4][4][3];

void DrawSurfaceFwdDif (
        Coeffs A,   /* Coefficients for Q(s, t) */
        int n_s,    /* Number of curves of constant s to be drawn, typically 5 to 10. */
        int n_t,    /* Number of curves of constant t to be drawn, typically 5 to 10. */
        int n)      /* Number of steps to use in drawing each curve, typically 20 to 100. */
{
    /* Initialize */
    double δ_s = 1.0 / (n_s - 1.0); double δ_t = 1.0 / (n_t - 1.0);
    /* "*" indicates matrix multiplication */
    DD_x = E(δ_s) * A_x * E(δ_t)^T;
    DD_y = E(δ_s) * A_y * E(δ_t)^T;
    DD_z = E(δ_s) * A_z * E(δ_t)^T;

    /* Draw n_s curves of constant s, for s = 0, δ_s, 2δ_s, ... 1 */
    for (i = 0; i < n_s; i++) {
        /* Procedure from Section 11.2.9 to draw a curve */
        DrawCurveFwdDif (n, First row of DD_x, First row of DD_y, First row of DD_z);
        /* Prepare for next iteration */
        Apply equation 11.92 to DD_x, DD_y, and DD_z;
    }
    /* Transpose DD_x, DD_y, DD_z so can continue working with rows */
    DD_x = DD_x^T; DD_y = DD_y^T; DD_z = DD_z^T;
    /* Draw n_t curves of constant t, for t = 0, δ_t, 2δ_t, 3δ_t, ... 1 */
    for (i = 0; i < n_t; i++) {
        DrawCurveFwdDif (n, First row of DD_x, First row of DD_y, First row of DD_z);
        /* Prepare for next iteration */
        Apply equation 11.92 to DD_x, DD_y, and DD_z;
    }
} /* DrawSurfaceFwdDif */
```

Fig. 11.46 Procedure to display a parametric surface as lines of constant s and t.

Samsung/Google, EX1027, p. 538

```
void DrawSurfaceRecSub (surface, ε)
{
    /* Test whether control points are within ε of a plane */
    if (Flat (surface, ε))
        DrawQuadrilateral (surface);      /* If within ε, draw the surface as a quadrilateral */

    else {                                /* If not flat enough, subdivide into four patches */
        SubdivideSurface (surface, &surfaceLL, &surfaceLR, &surfaceRL, &surfaceRR);
        DrawSurfaceRecSub (surfaceLL, ε);
        DrawSurfaceRecSub (surfaceLR, ε);
        DrawSurfaceRecSub (surfaceRL, ε);
        DrawSurfaceRecSub (surfaceRR, ε);
    }
} /* DrawSurfaceRecSub */
```

Fig. 11.47 Procedure to display a parametric surface as a shaded surface. DrawQuadrilateral renders individual, nearly flat quadrilaterals. Flat returns **true** if the surface is within ε of being flat.

yields in the first row of DD_x the terms $x(\delta_s, 0)$, $\Delta_t x(\delta_s, 0)$, $\Delta_t^2 x(\delta_s, 0)$, and $\Delta_t^3 x(\delta_s, 0)$. These quantities are now used to compute $x(\delta_s, t)$, using forward differences, as before. Then the steps of Eq. (11.92) are again repeated, and the first row of D is used to compute $x(2\delta_s, t)$, and so on.

Drawing the curves of constant t is done similarly. To calculate $x(s, 0)$, we apply Eq. (11.92) to DD_x but substitute *column* for *row*. Alternatively, DD_x^T can be used in place of DD_x, so that rows are still used to calculate $x(s, 0)$, $x(s, \delta_t)$, $x(s, 2\delta_t)$, and so on.

The algorithm for displaying a surface as lines of constant s and t using forward differences is shown in Fig. 11.46.

Surface display using recursive subdivision is also a simple extension from the procedure DrawCurveRecSub of Section 11.2.9, and is presented in Fig. 11.47. Nearly planar quadrilaterals are created, then are displayed by procedure DrawQuadrilateral as shaded flat quadrilaterals using the methods described in Chapters 15 and 16. The process is again simplest if the Bézier form is used.

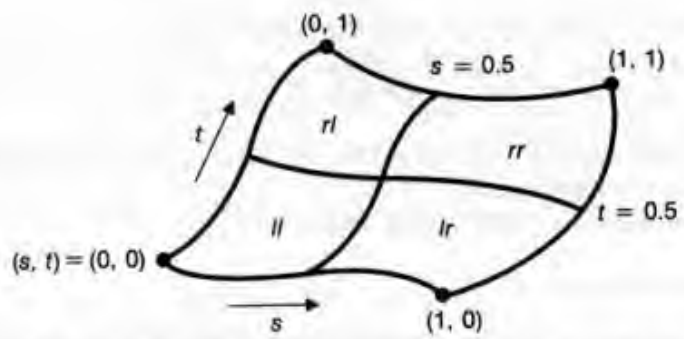


Fig. 11.48 A surface subdivided into four surfaces, *ll*, *lr*, *rl*, and *rr*, each of which is 'flatter' than the original surface.

Samsung/Google, EX1027, p. 539



Fig. 11.49 A subdivided surface whose three approximating quadrilaterals introduce a crack into the approximation.

Flatness is tested by computing the plane passing through three of the four corner control points and finding the distance from each of the other 13 control points to the plane; the maximum distance must be less than ε. This is just a generalization of the flatness test used for curves. [LANE79] discusses a different way to perform a more efficient flatness test. Of course, the recursion can again be done to a constant depth, eliminating the flatness test at the price of needless subdivisions.

Surface subdivision is done by splitting the surface along one parameter, say s, and then splitting each of the two resulting surfaces along t. The curve-splitting methods discussed in Section 11.2.7 are applied to each set of four control points running in the direction of the parameter along which the split is being made. Figure 11.48 shows the idea, with the resulting four surfaces labeled *ll*, *lr*, *rl*, and *rr* as an extension of the notation used in Fig. 11.35. Alternatively, the patch can be split along only one parameter, if the patch is locally flat along the other parameter. [LANE80a] gives a thorough theoretical account of the subdivision process and also describes how it can be used to find curve–curve and surface–surface intersections.

One critical problem with adaptive subdivision is the possibility of cracks between approximating quadrilaterals, as seen in Fig. 11.49. The cracks occur because of the different levels of subdivision applied to adjoining patches. They can be avoided by subdividing to a fixed depth or by making the flatness threshold ε very small. Both solutions, however, cause needless subdivisions. Another way to avoid cracks is to modify adjoining approximating quadrilaterals, as suggested in Fig. 11.50. This basic strategy is

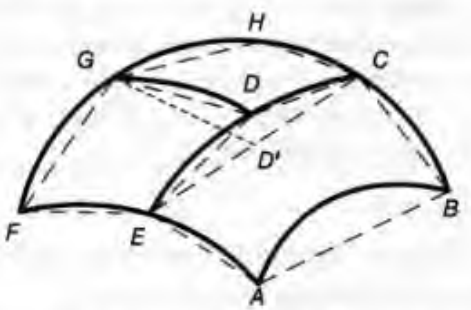


Fig. 11.50 Crack elimination in recursive subdivision. A naive algorithm displays quadrilaterals *ABCE*, *EFGD*, and *GDCH*. A more sophisticated algorithm displays quadrilaterals *ABCE*, *EFGD'*, *and GD'CH*. Vertex *D'* is used instead of *D*.

Samsung/Google, EX1027, p. 540

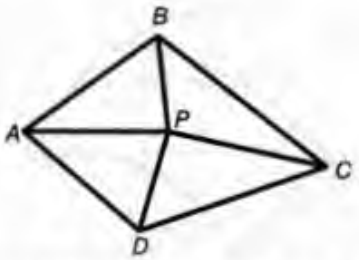


Fig. 11.51 A quadrilateral *ABCD* to be displayed is first subdivided into four triangles. The new point *P* is the average of points *A*, *B*, *C*, and *D*.

used by Clark [CLAR79] and by Barsky, DeRose, and Dippé [BARS87].

Procedure DrawQuadrilateral is given a nearly flat quadrilateral to display. The best way to display the quadrilateral as a shaded surface is to subdivide it further into four triangles, as shown in Fig. 11.51. This strategy avoids visual artifacts.

It is sometimes desirable to display only a portion of a bicubic surface. For instance, there may be a hole in a patch caused by a pipe going through the surface. Displaying only a portion can be done with *trimming curves*, which are just spline curves defined in (s, t) parameter space instead of in (x, y, z) space on the bicubic surface. When displaying a surface that is constrained by a trim curve using DrawSurfaceFwdDif (see Fig. 11.46), the (s, t) values of the trim curve are used to start and stop the iteration.

Other useful ways to display bicubic surfaces are presented in [FORR79].

11.4 QUADRIC SURFACES

The implicit surface equation of the form

$$f(x, y, z) = ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fxz + 2gx + 2hy + 2jz + k = 0 \quad (11.93)$$

defines the family of quadric surfaces. For example, if $a = b = c = -k = 1$ and the remaining coefficients are zero, a unit sphere is defined at the origin. If a through f are zero, a plane is defined. Quadric surfaces are particularly useful in specialized applications such as molecular modeling [PORT79; MAX79], and have also been integrated into solid-modeling systems. Recall, too, that rational cubic curves can represent conic sections; similarly, rational bicubic surfaces can represent quadrics. Hence, the implicit quadratic equation is an alternative to rational surfaces, *if* only quadric surfaces are being represented. Other reasons for using quadrics include ease of

- Computing the surface normal
- Testing whether a point is on the surface (just substitute the point into Eq. (11.93), evaluate, and test for a result within some ε of zero)
- Computing z given x and y (important in hidden-surface algorithms—see Chapter 15) and

Samsung/Google, EX1027, p. 541

- Calculating intersections of one surface with another.

An alternative representation of Eq. (11.93) is:

$$P^T \cdot Q \cdot P = 0, \tag{11.94}$$

$$\text{with} \quad Q = \begin{bmatrix} a & d & f & g \\ d & b & e & h \\ f & e & c & j \\ g & h & j & k \end{bmatrix} \quad \text{and} \quad P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{11.95}$$

The surface represented by Q can be easily translated and scaled. Given a 4×4 transformation matrix M of the form developed in Chapter 5, the transformed quadric surface Q' is given by

$$Q' = (M^{-1})^T \cdot Q \cdot M^{-1}. \tag{11.96}$$

The normal to the implicit surface defined by $f(x, y, z) = 0$ is the vector $[df/dx \quad df/dy \quad df/dz]$. This surface normal is much easier to evaluate than is the surface normal to a bicubic surface discussed in Section 11.3.4.

11.5 SUMMARY

This chapter has only touched on some of the important ideas concerning curve and surface representation, but it has given sufficient information so that you can implement interactive systems using these representations. Theoretical treatments of the material can be found in texts such as [BART87; DEBO78; FARI88; FAUX79; MORT85].

Polygon meshes, which are piecewise linear, are well suited for representing flat-faced objects but are seldom satisfactory for curve-faced objects. Piecewise continuous parametric cubic curves and bicubic surfaces are widely used in computer graphics and CAD to represent curve-faced objects because they

- Permit multiple values for a single value of x or y
- Represent infinite slopes
- Provide local control, such that changing a control point affects only a local piece of the curve
- Can be made either to interpolate or to approximate control points, depending on application requirements
- Are computationally efficient
- Permit refinement by subdivision and knot addition, facilitating display and interactive manipulation
- Are easily transformed by transformation of control points.

Although we have discussed only cubics, higher- and lower-order surfaces can also be used. The texts mentioned previously generally develop parametric curves and surfaces for the general case of order n.

Samsung/Google, EX1027, p. 542

EXERCISES

11.1 Develop the equations, similar to Eq. (11.2), for the coefficients A and B of the plane equation. Assume that the polygon vertices are enumerated counterclockwise as viewed toward the plane from the positive side of the plane. The surface normal—given by A, B, and C—points toward the positive side of the plane (which accounts for the need to negate the area computed for B, as discussed in Section 11.1.3).

11.2 Write a program to calculate the plane equation coefficients, given n vertices of a polygon that is approximately planar. The vertices are enumerated in a counterclockwise direction, as defined in Exercise 11.1. Test the program with $n = 3$ for several known planes; then test it for larger n.

11.3 Find the geometry matrix and basis matrix for the parametric representation of a straight line given in Eq. (11.11).

11.4 Implement the procedure DrawCurve given in Fig 11.18. Display a number of curves, varying the coefficients cx, cy, and cz. Try to make the curve correspond to some of the curve segments shown in figures in this chapter. Why it this difficult to do?

11.5 Show that, for a 2D curve $[x(t) \quad y(t)]$, G^1 continuity means that the geometric slope dy/dx is equal at the join points between segments.

11.6 Let $\gamma(t) = (t, t^2)$ for $0 \le t \le 1$, and let $\eta(t) = (2t + 1, t^3 + 4t + 1)$ for $0 \le t \le 1$. Notice that $\gamma(1) = (1, 1) = \eta(0)$, so γ and η join with C^0 continuity.

a. Plot $\eta(t)$ and $\gamma(t)$ for $0 \le t \le 1$.

b. Do $\eta(t)$ and $\gamma(t)$ meet with C^1 continuity at the join point? (You will need to compute the vectors $\frac{d\gamma}{dt}(1)$ and $\frac{d\eta}{dt}(0)$ to check this.)

c. Do $\eta(t)$ and $\gamma(t)$ meet with G^1 continuity at the join point? (You will need to check ratios from part (b) to determine this).

11.7 Consider the paths

$$\gamma(t) = (t^2 - 2t + 1, t^3 - 2t^2 + t) \quad \text{and} \quad \eta(t) = (t^2 + 1, t^3),$$

both defined on the interval $0 \le t \le 1$. The curves join, since $\gamma(1) = (1, 0) = \eta(0)$. Show that they meet with C^1 continuity, but not with G^1 continuity. Plot both curves as functions of t to demonstrate exactly why this happens.

11.8 Show that the two curves $\gamma(t) = (t^2 - 2t, t)$ and $\eta(t) = (t^2 + 1, t + 1)$ are both C^1 and G^1 continuous where they join at $\gamma(1) = \eta(0)$.

11.9 Analyze the effect on a B-spline of having in sequence four collinear control points.

11.10 Write a program to accept an arbitrary geometry matrix, basis matrix, and list of control points, and to draw the corresponding curve.

11.11 Find the conditions under which two joined Hermite curves have C^1 continuity.

11.12 Suppose the equations relating the Hermite geometry to the Bézier geometry were of the form $R_1 = \beta(P_2 - P_1)$, $R_4 = \beta(P_4 - P_3)$. Consider the four equally spaced Bézier control points $P_1 = (0, 0)$, $P_2 = (1, 0)$, $P_3 = (2, 0)$, $P_4 = (3, 0)$. Show that, for the parametric curve $Q(t)$ to have constant velocity from P_1 to P_4, the coefficient β must be equal to 3.

11.13 Write an interactive program that allows the user to create and refine piecewise continuous cubic curves. Represent the curves internally as B-splines. Allow the user to specify how the curve is to be interactively manipulated—as Hermite, Bézier, or B-splines.

11.14 Show that duplicate interior control points on a B-spline do not affect the C^2 continuity at the

Samsung/Google, EX1027, p. 543

join point. Do this by writing out the equations for the two curve segments formed by the control points $P_{i-1}, P_i, P_{i+1}, P_{i+2} = P_{i+1}, P_{i+3}$. Evaluate the second derivative of the first segment at $t = 1$, and that of the second segment at $t = 0$. They should be the same.

11.15 Find the blending functions for the Catmull–Rom splines of Eq. (11.47). Do they sum to 1, and are they everyone nonzero? If not, the spline is not contained in the convex hull of the points.

11.16 Using Eqs. (11.49), (11.50), and (11.19), find the basis matrix M_{KB} for the Kochanek-Bartels spline, using the geometry matrix G_{Bs_i} of Eq. (11.32).

11.17 Write an interactive program that allows the user to create, to manipulate interactively, and to refine piecewise continuous β-spline curves. Experiment with the effect of varying β_1 and β_2.

11.18 Write an interactive program that allows the user to create, to manipulate interactively, and refine piecewise continuous Kochanek–Bartels curves. Experiment with the effect of varying a, b, and c.

11.19 Implement both the forward-difference and recursive-subdivision curve-display procedures. Compare execution times for displaying various curves such that the curves are equally smooth to the eye.

11.20 Why is Eq. (11.36) for uniform B-splines written as $Q_i(t - t_i)$, whereas Eq. (11.43) for nonuniform B-splines is written as $Q_i(t)$?

11.21 Given a 2D nonuniform B-spline and an (x, y) value on the curve, write a program to find the corresponding value of t. Be sure to consider the possibility that, for a given value of x (or y), there may be multiple values of y (or x).

11.22 Given a value t at which a Bézier curve is to be split, use the de Casteljau construction shown in Fig. 11.35 to find the division matrices $D_B^L(t)$ and $D_B^R(t)$.

11.23 Apply the methodology used to derive Eq. (11.82) for Hermite surfaces to derive Eq. (11.86) for Bézier surfaces.

11.24 Write programs to display parametric cubic curves and surfaces using forward differences and recursive subdivision. Vary the step size δ and error measure ε to determine the effects of these parameters on the appearance of the curve.

11.25 Given four Hermite patches joined at a common corner and along the edges emanating from that corner, show the four geometry matrices and the relations that must hold between elements of the matrices.

11.26 Let $t_0 = 0$, $t_1 = 1$, $t_2 = 3$, $t_3 = 4$, $t_4 = 5$. Using these values, compute $B_{0,4}$ and each of the functions used in its definition. Then plot these functions on the interval $-3 \le t \le 8$.

11.27 Expand the recurrence relation of Eq. (11.44) into an explicit expression for $B_{i,4}(t)$. Use Fig. 11.26 as a guide.

11.28 Write a program that displays a nonuniform, non-rational B-spline, given as input a knot sequence and control points. Provide a user-controllable option to calculate the $B_{i,4}(t)$ in two ways: (a) using the recurrence relations of Eq. (11.44), and (b) using the explicit expression found in Exercise 11.27. Measure how much time is taken by each method. Is the faster method necessarily the better one?

11.29 Expand the program from Exercise 11.28 to allow interactive input and modification of the B-splines.

11.30 Write procedures to subdivide a curve recursively in two ways: adaptively, with a flatness test, and fixed, with a uniform level of subdivision. Draw a curve first with the adaptive subdivision, noting the deepest level of subdivision needed. Now draw the same curve with fixed subdivision as deep as that needed for the adaptive case. Compare the execution time of the two procedures for a variety of curves.

Samsung/Google, EX1027, p. 544

12
Solid Modeling

The representations discussed in Chapter 11 allow us to describe curves and surfaces in 2D and 3D. Just as a set of 2D lines and curves does not need to describe the boundary of a closed area, a collection of 3D planes and surfaces does not necessarily bound a closed volume. In many applications, however, it is important to distinguish between the inside, outside, and surface of a 3D object and to be able to compute properties of the object that depend on this distinction. In CAD/CAM, for example, if a solid object can be modeled in a way that adequately captures its geometry, then a variety of useful operations can be performed before the object is manufactured. We may wish to determine whether two objects interfere with each other; for example, whether a robot arm will bump into objects in its environment, or whether a cutting tool will cut only the material it is intended to remove. In simulating physical mechanisms, such as a gear train, it may be important to compute properties such as volume and center of mass. Finite-element analysis is applied to solid models to determine response to factors such as stress and temperature through finite-element modeling. A satisfactory representation for a solid object may even make it possible to generate instructions automatically for computer-controlled machine tools to create that object. In addition, some graphical techniques, such as modeling refractive transparency, depend on being able to determine where a beam of light enters and exits a solid object. These applications are all examples of *solid modeling*. The need to model objects as solids has resulted in the development of a variety of specialized ways to represent them. This chapter provides a brief introduction to these representations.

Samsung/Google, EX1027, p. 545

12.1 REPRESENTING SOLIDS

A representation's ability to encode things that *look* like solids does not by itself mean that the representation is adequate for representing solids. Consider how we have represented objects so far, as collections of straight lines, curves, polygons, and surfaces. Do the lines of Fig. 12.1(a) define a solid cube? If each set of four lines on each side of the object is assumed to bound a square face, then the figure is a cube. However, there is nothing in the representation given that requires the lines to be interpreted this way. For example, the same set of lines would be used to draw the figure if any or all of the faces were missing. What if we decide that each planar loop of connected lines in the drawing by definition determines a polygonal face? Then, Fig. 12.1(b) would consist of all of the faces of Fig. 12.1(a), plus an extra "dangling" face, producing an object that does not bound a volume. As we shall see in Section 12.5, some extra constraints are needed if we want to ensure that a representation of this sort models only solids.

Requicha [REQU80] provides a list of the properties desirable in a solid representation scheme. The *domain* of representation should be large enough to allow a useful set of physical objects to be represented. The representation should ideally be *unambiguous*: There should be no question as to what is being represented, and a given representation should correspond to one and only one solid, unlike the one in Fig. 12.1(a). An unambiguous representation is also said to be *complete*. A representation is *unique* if it can be used to encode any given solid in only one way. If a representation can ensure uniqueness, then operations such as testing two objects for equality are easy. An *accurate* representation allows an object to be represented without approximation. Much as a graphics system that can draw only straight lines forces us to create approximations of smooth curves, some solid modeling representations represent many objects as approximations. Ideally, a representation scheme should make it *impossible to create an invalid representation* (i.e., one that does not correspond to a solid), such as Fig. 12.1(b). On the other hand, it should be *easy to create a valid representation*, typically with the aid of an interactive solid modeling system. We would like objects to maintain *closure* under

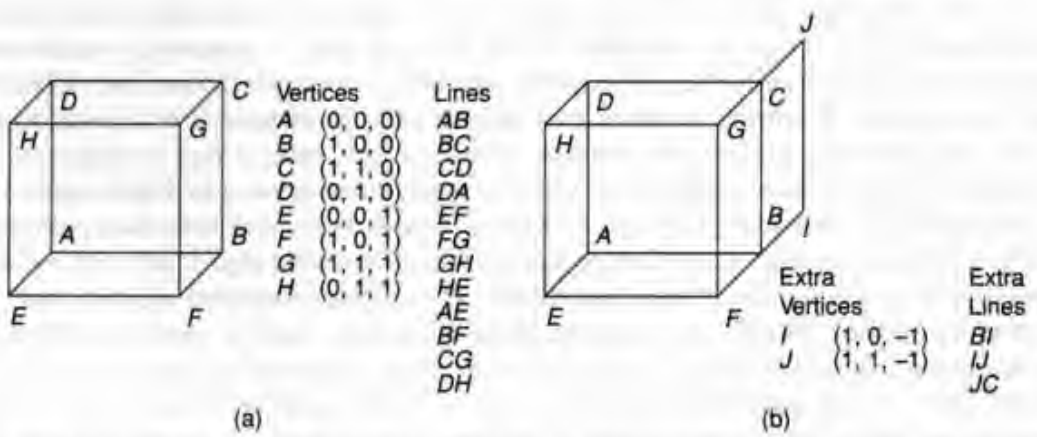


Fig. 12.1 (a) A wireframe cube composed of 12 straight lines. (b) A wireframe cube with an extra face.

Samsung/Google, EX1027, p. 546

rotation, translation, and other operations. This means that performing these operations on valid solids should yield only valid solids. A representation should be *compact* to save space, which in turn may save communication time in a distributed system. Finally, a representation should allow the use of *efficient* algorithms for computing desired physical properties, and, most important for us, for creating pictures.

Designing a representation with all these properties is difficult indeed, and compromises are often necessary. As we discuss the major representations in use today, our emphasis will be on providing enough detail to be able to understand how these representations can be interfaced to graphics software. More detail, with an emphasis on the solid modeling aspects, can be found in [REQU80; MORT85; MANT88].

12.2 REGULARIZED BOOLEAN SET OPERATIONS

No matter how we represent objects, we would like to be able to combine them in order to make new ones. One of the most intuitive and popular methods for combining objects is by *Boolean set operations*, such as union, difference, and intersection, as shown in Fig. 12.2. These are the 3D equivalents of the familiar 2D Boolean operations. Applying an ordinary Boolean set operation to two solid objects, however, does not necessarily yield a solid object. For example, the ordinary Boolean intersections of the cubes in Fig. 12.3(a) through (e) are a solid, a plane, a line, a point, and the null object, respectively.

Rather than using the ordinary Boolean set operators, we will instead use the *regularized Boolean set operators* [REQU77], denoted $\cup^*$, $\cap^*$, and $-^*$, and defined such that operations on solids always yield solids. For example, the regularized Boolean intersection of the objects shown in Fig. 12.3 is the same as their ordinary Boolean intersection in cases (a) and (e), but is empty in (b) through (d).

To explore the difference between ordinary and regularized operators, we can consider any object to be defined by a set of points, partitioned into interior points and boundary points, as shown in Fig. 12.4(a). *Boundary* points are those points whose distance from the object and the object's complement is zero. Boundary points need not be part of the object. A *closed* set contains all its boundary points, whereas an *open* set contains none. The union of a set with the set of its boundary points is known as the set's *closure*, as shown in Fig. 12.4(b), which is itself a closed set. The *boundary* of a closed set is the set of its boundary points, whereas the *interior*, shown in Fig. 12.4(c), consists of all of the set's other points, and thus is the complement of the boundary with respect to the object. The *regularization* of

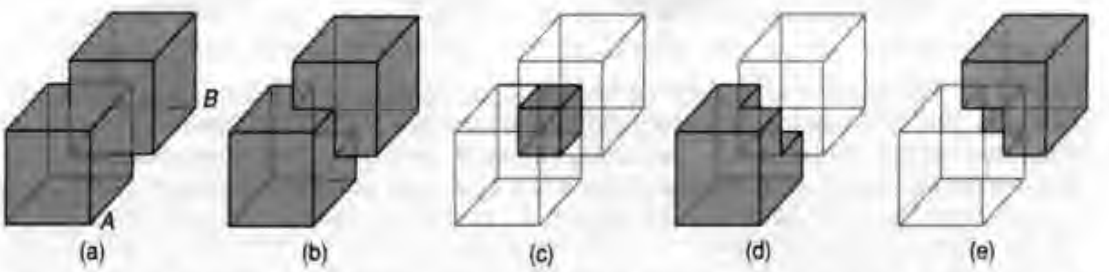


Fig. 12.2 Boolean operations. (a) Objects A and B, (b) $A \cup B$, (c) $A \cap B$, (d) $A - B$, and (e) $B - A$.

Samsung/Google, EX1027, p. 547

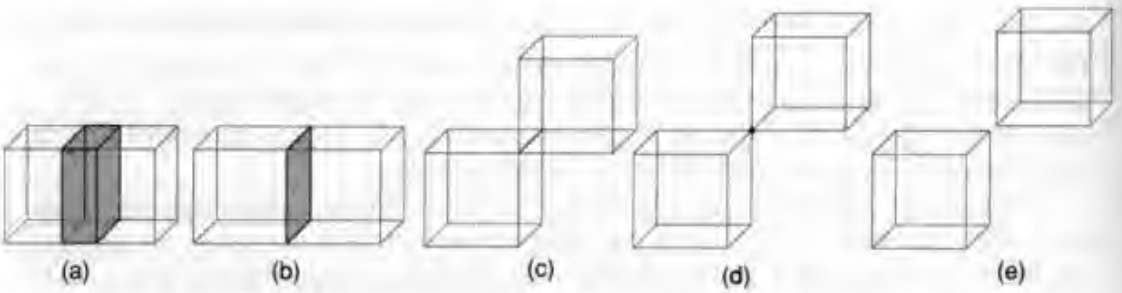


Fig. 12.3 The ordinary Boolean intersection of two cubes may produce (a) a solid, (b) a plane, (c) a line, (d) a point, or (e) the null set.

a set is defined as the closure of the set's interior points. Figure 12.4(d) shows the closure of the object in Fig. 12.4(c) and, therefore, the regularization of the object in Fig. 12.4(a). A set that is equal to its own regularization is known as a *regular set*. Note that a regular set can contain no boundary point that is not adjacent to some interior point; thus, it can have no "dangling" boundary points, lines, or surfaces. We can define each regularized Boolean set operator in terms of the corresponding ordinary Boolean set operator as

$$A\ op^*\ B = \text{closure}(\text{interior}(A\ op\ B)), \tag{12.1}$$

where *op* is one of $\cup$, $\cap$, or $-$. The regularized Boolean set operators produce only regular sets when applied to regular sets.

We now compare the ordinary and regularized Boolean set operations as performed on regular sets. Consider the two objects of Fig. 12.5(a), positioned as shown in Fig. 12.5(b). The ordinary Boolean intersection of two objects contains the intersection of the interior and boundary of each object with the interior and boundary of the other, as shown in Fig. 12.5(c). In contrast, the regularized Boolean intersection of two objects, shown in Fig. 12.5(d), contains the intersection of their interiors and the intersection of the interior of each with the boundary of the other, but only a subset of the intersection of their boundaries. The criterion used to define this subset determines how regularized Boolean

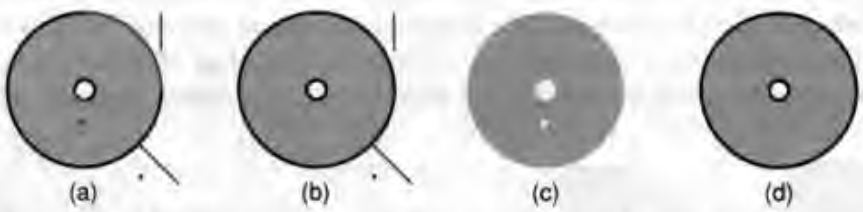


Fig. 12.4 Regularizing an object. (a) The object is defined by interior points, shown in light gray, and boundary points. Boundary points that are part of the object are shown in black; the rest of the boundary points are shown in dark gray. The object has dangling and unattached points and lines, and there is a boundary point in the interior that is not part of the object. (b) Closure of the object. All boundary points are part of the object. The boundary point embedded in the interior of (a) is now part of the interior. (c) Interior of the object. Dangling and unattached points and lines have been eliminated. (d) Regularization of the object is the closure of its interior.

Samsung/Google, EX1027, p. 548

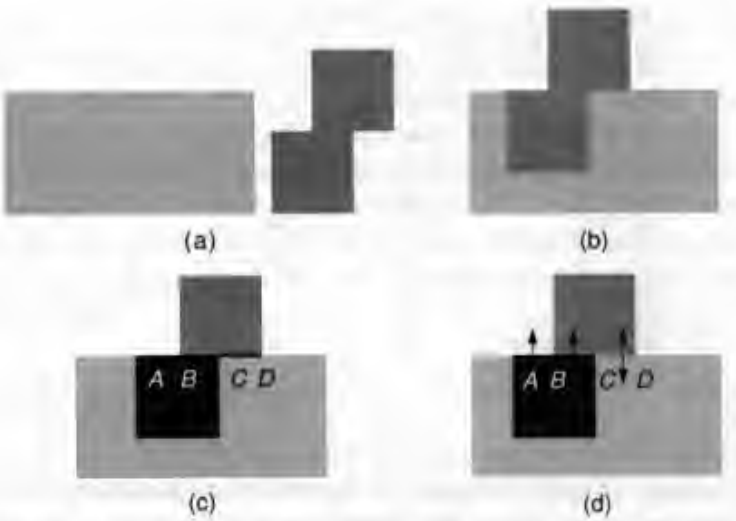


Fig. 12.5 Boolean intersection. (a) Two objects, shown in cross-section. (b) Positions of object prior to intersection. (c) Ordinary Boolean intersection results in a dangling face, shown as line *CD* in cross-section. (d) Regularized Boolean intersection includes a piece of shared boundary in the resulting boundary if both objects lie on the same side of it (*AB*), and excludes it if the objects lie on opposite sides (*CD*). Boundary–interior intersections are always included (*BC*).

intersection differs from ordinary Boolean intersection, in which all parts of the intersection of the boundaries are included.

Intuitively, a piece of the boundary–boundary intersection is included in the regularized Boolean intersection if and only if the interiors of both objects lie on the same side of this piece of shared boundary. Since the interior points of both objects that are directly adjacent to that piece of boundary are in the intersection, the boundary piece must also be included to maintain closure. Consider the case of a piece of shared boundary that lies in coplanar faces of two polyhedra. Determining whether the interiors lie on the same side of a shared boundary is simple if both objects are defined such that their surface normals point outward (or inward). The interiors are on the same side if the normals point in the same direction. Thus, segment *AB* in Fig. 12.5(d) is included. Remember that those parts of one object's boundary that intersect with the other object's interior, such as segment *BC*, are always included.

Consider what happens when the interiors of the objects lie on opposite sides of the shared boundary, as is the case with segment *CD*. In such cases, none of the interior points adjacent to the boundary are included in the intersection. Thus, the piece of shared boundary is not adjacent to any interior points of the resulting object and therefore is not included in the regularized intersection. This additional restriction on which pieces of shared boundary are included ensures that the resulting object is a regular set. The surface normal of each face of the resulting object's boundary is the surface normal of whichever surface(s) contributed that part of the boundary. (As we shall see in Chapter 16, surface normals are important in shading objects.) Having determined which faces lie in the

Samsung/Google, EX1027, p. 549

TABLE 12.1 REGULARIZED BOOLEAN SET OPERATIONS

Set	$A \cup^* B$	$A \cap^* B$	$A -^* B$
$A_i \cap B_i$	●	●	
$A_i - B$	●		●
$B_i - A$	●		
$A_b \cap B_i$		●	
$B_b \cap A_i$		●	●
$A_b - B$	●		●
$B_b - A$	●		
$A_b \cap B_b$ *same*	●	●	
$A_b \cap B_b$ *diff*			●

boundary, we include an edge or vertex of the boundary–boundary intersection in the boundary of the intersection if it is adjacent to one of these faces.

The results of each regularized operator may be defined in terms of the ordinary operators applied to the boundaries and interiors of the objects. Table 12.1 shows how the regularized operators are defined for any objects A and B; Fig. 12.6 shows the results of performing the operations. A_b and A_i are A's boundary and interior, respectively. $A_b \cap B_b$ *same* is that part of the boundary shared by A and B for which A_i and B_i lie on the same side. This is the case for some point b on the shared boundary if at least one point i adjacent to it is a member of both A_i and B_i. $A_b \cap B_b$ *diff* is that part of the boundary shared by A and B for which A_i and B_i lie on opposite sides. This is true for b if it is adjacent to no such point i. Each regularized operator is defined by the union of the sets associated with those rows that have a ● in the operator's column.

Note that, in all cases, each piece of the resulting object's boundary is on the boundary of one or both of the original objects. When computing $A \cup^* B$ or $A \cap^* B$, the surface normal of a face of the result is inherited from the surface normal of the corresponding face of one or both original objects. In the case of $A -^* B$, however, the surface normal of each

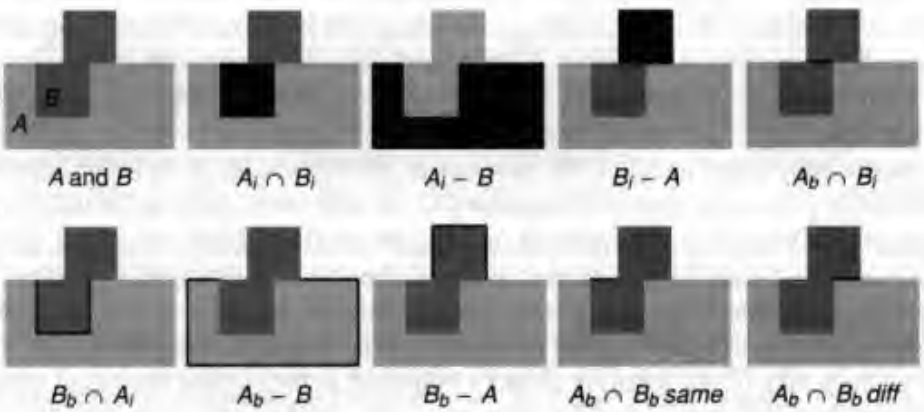


Fig. 12.6 Ordinary Boolean operations on subsets of two objects.

Samsung/Google, EX1027, p. 550

face of the result at which B has been used to excavate A must point in the *opposite* direction from B's surface normal at that face. This corresponds to the boundary pieces $A_b \cap B_b$ *diff* and $B_b \cap A_s$. Alternatively, $A -^* B$ may be rewritten as $A \cap^* \overline{B}$. We can obtain $\overline{B}$ (the complement of B) by complementing B's interior and reversing the normals of its boundary.

The regularized Boolean set operators have been used as a user–interface technique to build complex objects from simple ones in most of the representation schemes we shall discuss. They are also included explicitly in one of the schemes, constructive solid geometry. In the following sections, we shall describe a variety of ways to represent solid objects unambiguously.

12.3 PRIMITIVE INSTANCING

In *primitive instancing*, the modeling system defines a set of primitive 3D solid shapes that are relevant to the application area. These primitives are typically parameterized not just in terms of the transformations of Chapter 7, but on other properties as well. For example, one primitive object may be a regular pyramid with a user-defined number of faces meeting at the apex. Primitive instances are similar to parameterized objects, such as the menus of Chapter 2, except that the objects are solids. A parameterized primitive may be thought of as defining a family of parts whose members vary in a few parameters, an important CAD concept known as *group technology*. Primitive instancing is often used for relatively complex objects, such as gears or bolts, that are tedious to define in terms of Boolean combinations of simpler objects, yet are readily characterized by a few high-level parameters. For example, a gear may be parameterized by its diameter or number of teeth, as shown in Fig. 12.7.

Although we can build up a hierarchy of primitive instances, each leaf-node instance is still a separately defined object. In primitive instancing, no provisions are made for combining objects to form a new higher-level object, using, for example, the regularized Boolean set operations. Thus, the only way to create a new kind of object is to write the code that defines it. Similarly, the routines that draw the objects or determine their mass properties must be written individually for each primitive.

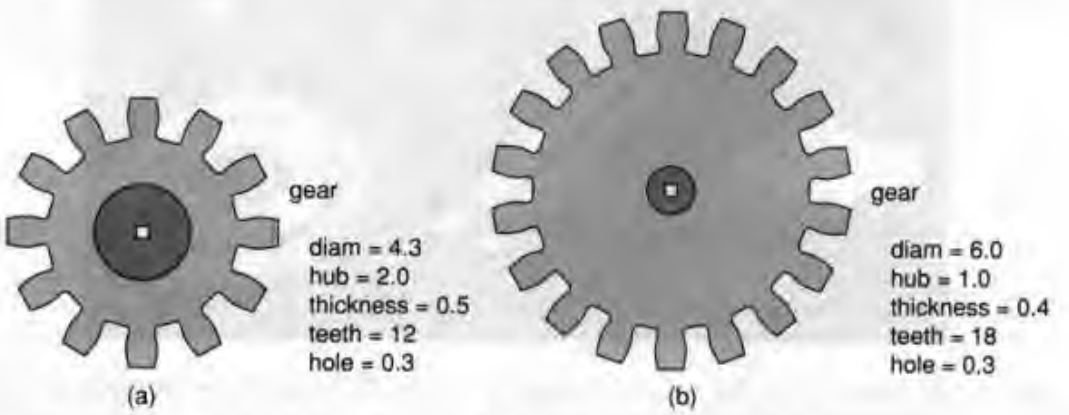


Fig. 12.7 Two gears defined by primitive instancing.

Samsung/Google, EX1027, p. 551

12.4 SWEEP REPRESENTATIONS

Sweeping an object along a trajectory through space defines a new object, called a *sweep*. The simplest kind of sweep is defined by a 2D area swept along a linear path normal to the plane of the area to create a volume. This is known as a *translational sweep* or *extrusion* and is a natural way to represent objects made by extruding metal or plastic through a die with the desired cross-section. In these simple cases, each sweep's volume is simply the swept object's area times the length of the sweep. Simple extensions involve scaling the cross-section as it is swept to produce a tapered object or sweeping the cross-section along a linear path that is not normal to it. *Rotational sweeps* are defined by rotating an area about an axis. Figure 12.8 shows two objects and simple translational and rotational sweeps generated using them.

The object being swept does not need to be 2D. Sweeps of solids are useful in modeling the region swept out by a machine-tool cutting head or robot following a path, as shown in Fig. 12.9. Sweeps whose generating area or volume changes in size, shape, or orientation as they are swept and that follow an arbitrary curved trajectory are called *general sweeps*. General sweeps of 2D cross-sections are known as *generalized cylinders* in computer vision [BINF71] and are usually modeled as parameterized 2D cross-sections swept at right angles along an arbitrary curve. General sweeps are particularly difficult to model efficiently. For example, the trajectory and object shape may make the swept object intersect itself, making volume calculations complicated. As well, general sweeps do not always generate solids. For example, sweeping a 2D area in its own plane generates another 2D area.

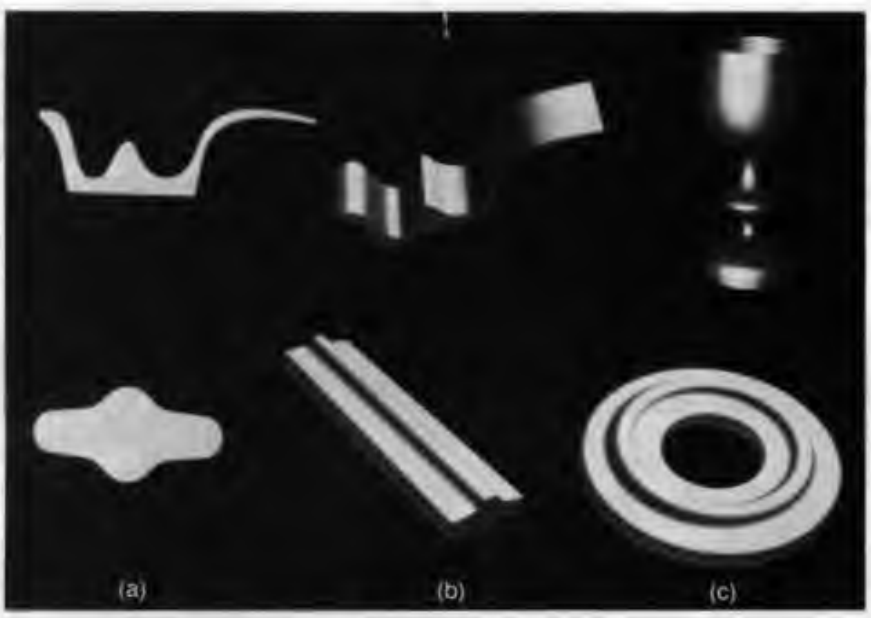


Fig. 12.8 Sweeps. (a) 2D areas are used to define (b) translational sweeps and (c) rotational sweeps. (Created using the Alpha_1 modeling system. Courtesy of the University of Utah.)

Samsung/Google, EX1027, p. 552

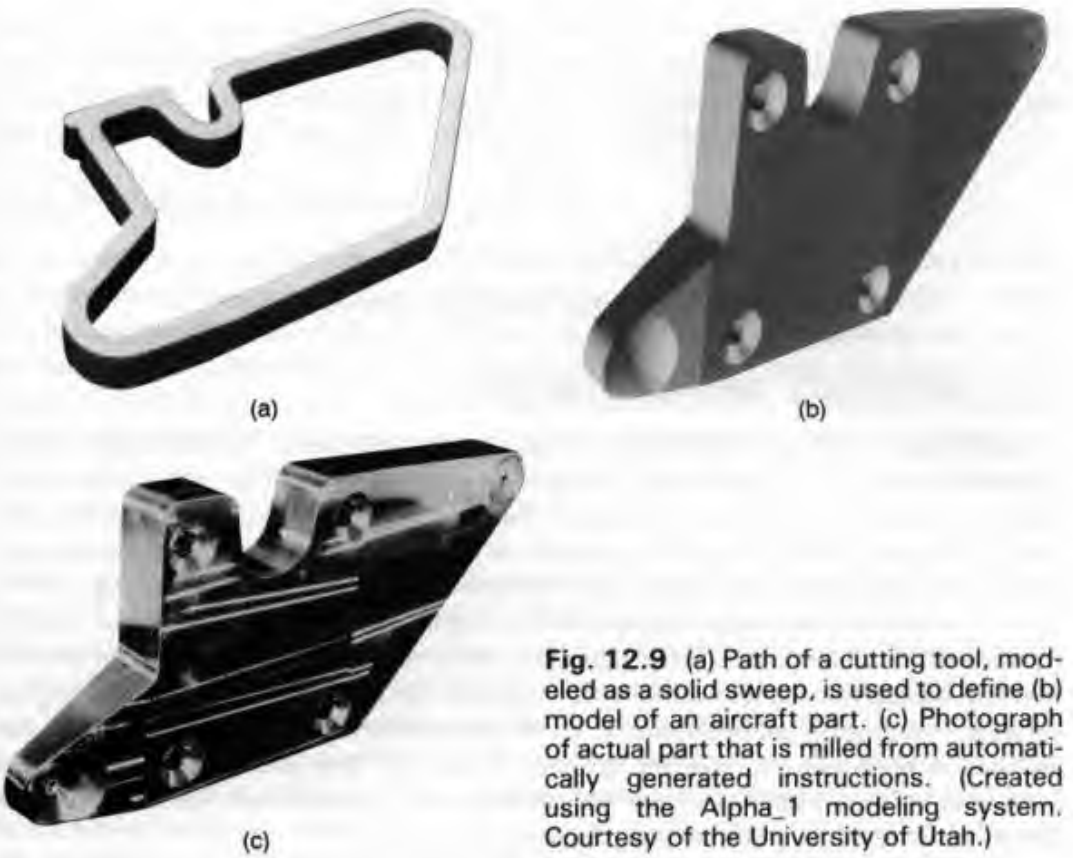


Fig. 12.9 (a) Path of a cutting tool, modeled as a solid sweep, is used to define (b) model of an aircraft part. (c) Photograph of actual part that is milled from automatically generated instructions. (Created using the Alpha_1 modeling system. Courtesy of the University of Utah.)

In general, it is difficult to apply regularized Boolean set operations to sweeps without first converting to some other representation. Even simple sweeps are not closed under regularized Boolean set operations. For example, the union of two simple sweeps is in general not a simple sweep, as shown in Fig. 12.10. Despite problems of closure and calculation, however, sweeps are a natural and intuitive way to construct a variety of objects. For this reason, many solid modeling systems allow users to construct objects as sweeps, but store the objects in one of the other representations that we shall discuss.

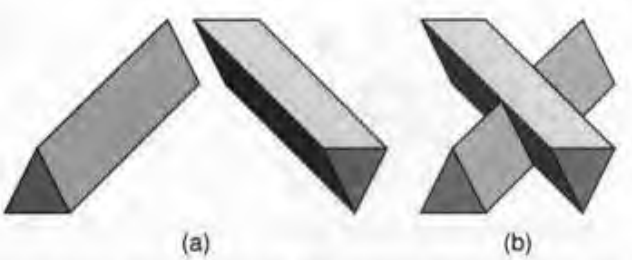


Fig. 12.10 (a) Two simple sweeps of 2D objects (triangles). (b) The union of the sweeps shown in (a) is not itself a simple sweep of a 2D object.

Samsung/Google, EX1027, p. 553

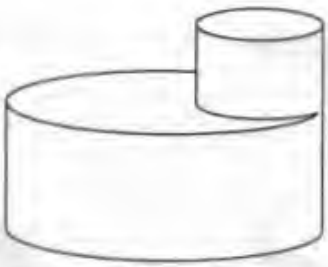


Fig. 12.11 How many faces does this object have?

12.5 BOUNDARY REPRESENTATIONS

Boundary representations (also known as *b-reps*) resemble the naive representations that we discussed in Section 12.1 in that they describe an object in terms of its surface boundaries: vertices, edges, and faces. Some b-reps are restricted to planar, polygonal boundaries, and may even require faces to be convex polygons or triangles. Determining what constitutes a face can be particularly difficult if curved surfaces are allowed, as shown in Fig. 12.11. Curved faces are often approximated with polygons. Alternatively, they can also be represented as surface patches if the algorithms that process the representation can treat the resulting intersection curves, which will, in general, be of higher order than the original surfaces. B-reps grew out of the simple vector representations used in earlier chapters and are used in many current modeling systems. Because of their prevalence in graphics, a number of efficient techniques have been developed to create smooth shaded pictures of polygonal objects; many of these are discussed in Chapter 16.

Many b-rep systems support only solids whose boundaries are *2-manifolds*. By definition, every point on a 2-manifold has some arbitrarily small neighborhood of points around it that can be considered topologically the same as a disk in the plane. This means that there is a continuous one-to-one correspondence between the neighborhood of points and the disk, as shown in Fig. 12.12(a) and (b). For example, if more than two faces share an edge, as in Fig. 12.12(c), any neighborhood of a point on that edge contains points from each of those faces. It is intuitively obvious that there is no continuous one-to-one

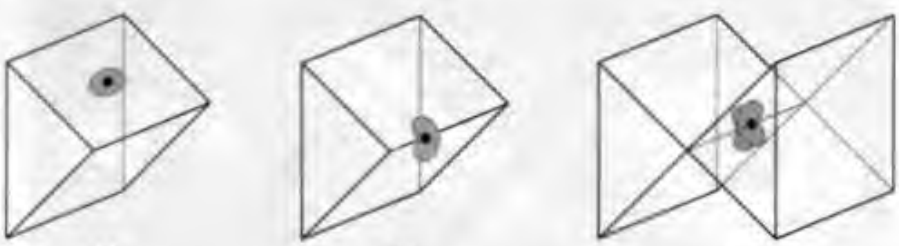


Fig. 12.12 On a 2-manifold, each point, shown as a black dot, has a neighborhood of surrounding points that is a topological disk, shown in gray in (a) and (b). (c) If an object is not a 2-manifold, then it has points that do not have a neighborhood that is a topological disk.

Samsung/Google, EX1027, p. 554

correspondence between this neighborhood and a disk in the plane, although the mathematical proof is by no means trivial. Thus, the surface in Fig. 12.12(c) is not a 2-manifold. Although some current systems do not have this restriction, we limit our discussion of b-reps to 2-manifolds, except where we state otherwise.

12.5.1 Polyhedra and Euler's Formula

A *polyhedron* is a solid that is bounded by a set of polygons whose edges are each a member of an even number of polygons (exactly two polygons in the case of 2-manifolds) and that satisfies some additional constraints (discussed later). A *simple polyhedron* is one that can be deformed into a sphere; that is, a polyhedron that, unlike a torus, has no holes. The b-rep of a simple polyhedron satisfies Euler's formula, which expresses an invariant relationship among the number of vertices, edges, and faces of a simple polyhedron:

$$V - E + F = 2, \tag{12.2}$$

where V is the number of vertices, E is the number of edges, and F is the number of faces. Figure 12.13 shows some simple polyhedra and their numbers of vertices, edges, and faces. Note that the formula still applies if curved edges and nonplanar faces are allowed. Euler's formula by itself states necessary but not sufficient conditions for an object to be a simple polyhedron. One can construct objects that satisfy the formula but do not bound a volume, by attaching one or more dangling faces or edges to an otherwise valid solid, as in Fig. 12.1(b). Additional constraints are needed to guarantee that the object is a solid: each edge must connect two vertices and be shared by exactly two faces, at least three edges must meet at each vertex, and faces must not interpenetrate.

A generalization of Euler's formula applies to 2-manifolds that have faces with holes:

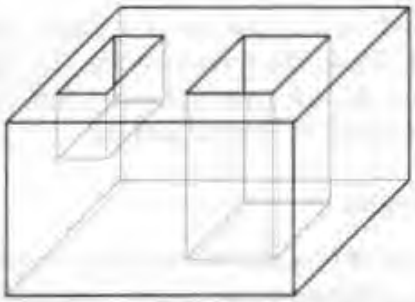
$$V - E + F - H = 2(C - G), \tag{12.3}$$

where H is the number of holes in the faces, G is the number of holes that pass through the object, and C is the number of separate components (parts) of the object, as shown in Fig. 12.14. If an object has a single component, its G is known as its *genus*; if it has multiple components, then its G is the sum of the genera of its components. As before, additional constraints are also needed to guarantee that the objects are solids.



Fig. 12.13 Some simple polyhedra with their V, E, and F values. In each case $V - E + F = 2$.

Samsung/Google, EX1027, p. 555



$$V - E + F - H = 2(C - G)$$
$$24 \quad 36 \quad 15 \quad 3 \qquad 1 \quad 1$$

Fig. 12.14 A polyhedron classified according to Eq. (12.3), with two holes in its top face and one hole in its bottom face.

Baumgart introduced the notion of a set of *Euler operators* that operate on objects satisfying Euler's formula to transform the objects into new objects that obey the formula as well, by adding and removing vertices, edges, and faces [BAUM74]. Braid, Hillyard, and Stroud [BRAI78] show how a small number of Euler operators can be composed to transform objects, provided that intermediate objects are not required to be valid solids, whereas Mäntylä [MANT88] proves that all valid b-reps can be constructed by a finite sequence of Euler operators. Other operators that do not affect the number of vertices, edges, or faces may be defined that *tweak* an object by moving the existing vertices, edges, or faces, as shown in Fig. 12.15.

Perhaps the simplest possible b-rep is a list of polygonal faces, each represented by a list of vertex coordinates. To represent the direction in which each polygon faces, we list a polygon's vertices in clockwise order, as seen from the exterior of the solid. To avoid replicating coordinates shared by faces, we may instead represent each vertex of a face by an index into a list of coordinates. In this representation, edges are represented implicitly by

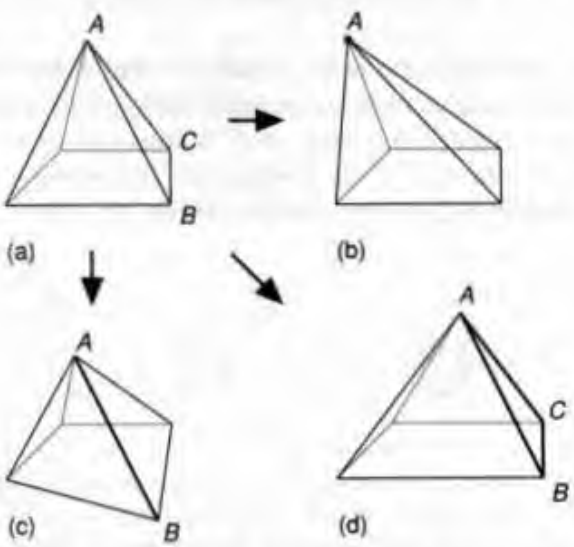


Fig. 12.15 (a) An object on which tweaking operations are performed to move (b) vertex *A*, (c) edge *AB*, and (d) face *ABC*.

Samsung/Google, EX1027, p. 556

the pairs of adjacent vertices in the polygon vertex lists. Edges may instead be represented explicitly as pairs of vertices, with each face now defined as a list of indices into the list of edges. These representations were discussed in more detail in Section 11.1.1.

12.5.2 The Winged-Edge Representation

Simple representations make certain computations quite expensive. For example, discovering the two faces shared by an edge (e.g., to help prove that a representation encodes a valid solid) requires searching the edge lists of all the faces. More complex b-reps have been designed to decrease the cost of these computations. One of the most popular is the *winged-edge* data structure developed by Baumgart [BAUM72; BAUM75]. As shown in Fig. 12.16, each edge in the winged-edge data structure is represented by pointers to its two vertices, to the two faces sharing the edge, and to four of the additional edges emanating from its vertices. Each vertex has a backward pointer to one of the edges emanating from it, whereas each face points to one of its edges. Note that we traverse the vertices of an edge in opposite directions when following the vertices of each of its two faces in clockwise order. Labeling the edge's vertices n and p, we refer to the face to its right when traversing the edge from n to p as its p face, and the face to its right when traversing the edge from p to n as its n face. For edge $E1$ in Fig. 12.16, if n is $V1$ and p is $V2$, then $F1$ is $E1$'s p face, and $F2$ is its n face. The four edges to which each edge points can be classified as follows. The two edges that share the edge's n vertex are the next (clockwise) edge of the n face, and the previous (counterclockwise) edge of the p face, $E3$ and $E2$, respectively. The two edges that share the edge's p vertex are the next (clockwise) edge of the p face, and the previous (counterclockwise) edge of the n face, $E4$ and $E5$, respectively. These four edges are the "wings" from which the winged-edge data structure gets its name.

Note that the data structure described here handles only faces that have no holes. This limitation can be removed by representing each face as a set of edge loops—a clockwise outer loop and zero or more counterclockwise inner loops for its holes—as described in

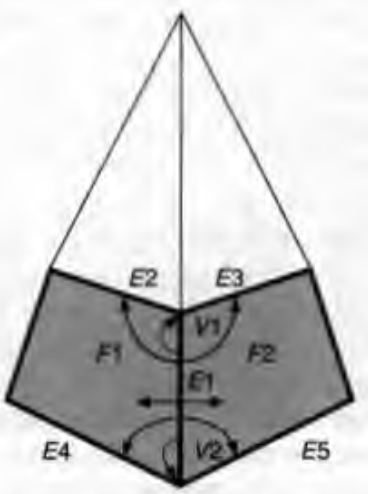


Fig. 12.16 Winged-edge data structure for $E1$. Each of $V1$, $V2$, $F1$, and $F2$ also have a backward pointer to one of their edges (not shown).

Samsung/Google, EX1027, p. 557

Section 19.1. Alternatively, a special auxiliary edge can be used to join each hole's boundary to the outer boundary. Each auxiliary edge is traversed twice, once in each direction, when a circuit of its face's edges is completed. Since an auxiliary edge has the same face on both of its sides, it can be easily identified because its two face pointers point to the same face.

A b-rep allows us to query which faces, edges, or vertices are adjacent to each face, edge, or vertex. These queries correspond to nine kinds of *adjacency relationships*. The winged-edge data structure makes it possible to determine in constant time which vertices or faces are associated with an edge. It takes longer to compute other adjacency relationships. One attractive property of the winged edge is that the data structures for the edges, faces, and vertices are each of a small, constant size. Only the number of instances of each data structure varies among objects. Weiler [WEIL85] and Woo [WOO85] discuss the space–time efficiency of the winged edge and a variety of alternative b-rep data structures.

12.5.3 Boolean Set Operations

B-reps may be combined, using the regularized Boolean set operators, to create new b-reps [REQU85]. Sarraga [SARR83] and Miller [MILL87] discuss algorithms that determine the intersections between quadric surfaces. Algorithms for combining polyhedral objects are presented in [TURN84; REQU85; PUTN86; LAID86], and Thibault and Naylor [THIB87] describe a method based on the binary space-partitioning tree representation of solids discussed in Section 12.6.4.

One approach [LAID86] is to inspect the polygons of both objects, splitting them if necessary to ensure that the intersection of a vertex, edge, or face of one object with any vertex, edge, or face of another, is a vertex, edge, or face of both. The polygons of each object are then classified relative to the other object to determine whether they lie inside, outside, or on its boundary. Referring back to Table 12.1, we note that since this is a b-rep, we are concerned with only the last six rows, each of which represents some part of one or both of the original object boundaries, A_b and B_b. After splitting, each polygon of one object is either wholly inside the other object ($A_b \cap B_i$ or $B_b \cap A_i$), wholly outside the other object ($A_b - \mathrm{B}$ or $B_b - \mathrm{A}$), or part of the shared boundary ($A_b \cap B_b$ *same* or $A_b \cap B_b$ *diff*).

A polygon may be classified by the ray-casting technique discussed in Section 15.10.1. Here, we construct a vector in the direction of the polygon's surface normal from a point in the polygon's interior, and then find the closest polygon that intersects the vector in the other object. If no polygon is intersected, the original polygon is outside the other object. If the closest intersecting polygon is coplanar with the original polygon, then this is a boundary–boundary intersection, and comparing polygon normals indicates what kind of intersection it is ($A_b \cap B_b$ *same* or $A_b \cap B_b$ *diff*). Otherwise, the dot product of the two polygons' normals is inspected. A positive dot product indicates that the original polygon is inside the other object, whereas a negative dot product indicates that it is outside. A zero dot product occurs if the vector is in the plane of the intersected polygon; in this case, the vector is perturbed slightly and is intersected again with the other object's polygons.

Vertex-adjacency information can be used to avoid the overhead of classifying each polygon in this way. If a polygon is adjacent to (i.e., shares vertices with) a classified

Samsung/Google, EX1027, p. 558

polygon and does not meet the surface of the other object, then it is assigned the same classification. All vertices on the common boundary between objects can be marked during the initial polygon-splitting phase. Whether or not a polygon meets the other object's surface can be determined by checking whether or not it has boundary vertices.

Each polygon's classification determines whether it is retained or discarded in the operation creating the composite object, as described in Section 12.2. For example, in forming the union, any polygon belonging to one object that is inside the other object is discarded. Any polygon from either object that is not inside the other is retained, except in the case of coplanar polygons. Coplanar polygons are discarded if they have opposite surface normals, and only one of a pair is retained if the directions of the surface normals are the same. Deciding which polygon to retain is important if the objects are made of different materials. Although $A \cup^* B$ has the same geometric meaning as $B \cup^* A$, the two may have visibly different results in this case, so the operation may be defined to favor one of its operands in the case of coplanar polygons.

12.5.4 Nonpolyhedral b-Reps

Unfortunately, polyhedral representations can only approximate objects that are not themselves polyhedral, and can require large amounts of data to approximate objects with curved surfaces acceptably. Consider the problem of representing a cylindrical object in a cylindrical hole with polyhedral b-reps, as shown in Fig. 12.17. If the boundaries of the actual objects touch, then even if the boundaries of the two polyhedral approximations are initially coincident as well, the approximations will intersect if one is slowly rotated, no matter how many polygons are used in the approximation.

One promising approach to exact b-reps allows sculpted surfaces defined by curves. The Alpha_1 [COHE83] and Geomod [TILL83] modeling systems model such free-form surfaces as tensor products of NURBS (see Section 11.2.5). Since each individual surface may not itself be closed, Thomas [THOM84] has developed an algorithm for Alpha_1 that performs regularized Boolean set operations on objects whose boundaries are only partially specified, as shown in Fig. 12.18. The objects in Color Plate I.31 were modeled with Alpha_1.

Because b-reps tile an object's surface, they do not provide a unique representation of a solid. In addition, as mentioned previously, many b-rep–based systems handle only objects whose surfaces are 2-manifolds. Note that 2-manifolds are not closed under regularized

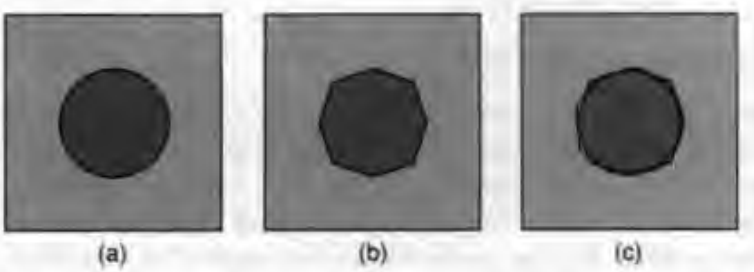


Fig. 12.17 (a) Cross-section of a cylinder in a round hole. (b) Polygonal approximation of hole and cylinder. (c) Interference occurs if approximated cylinder is turned relative to hole.

Samsung/Google, EX1027, p. 559

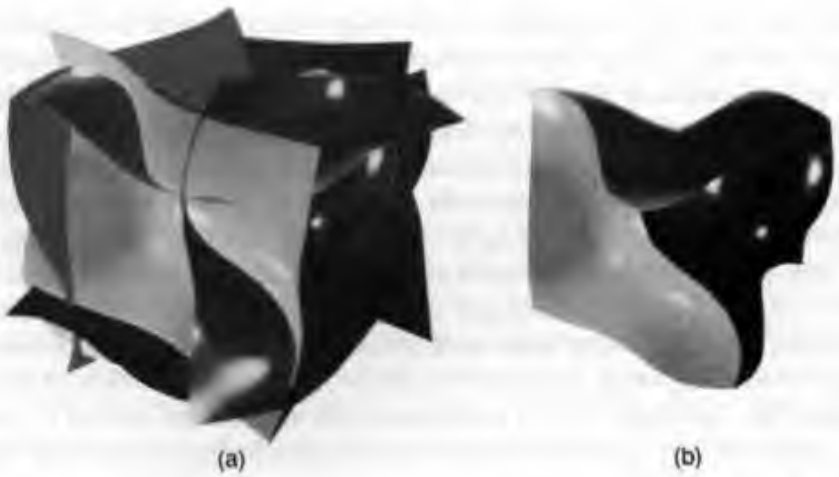


Fig. 12.18 Boolean set operations on partially bounded objects. (a) Six partially bounded sets. (b) Intersection of sets defines a wavy cube. (Courtesy of Spencer W. Thomas, University of Utah.)

Boolean set operations. The regularized union of two b-rep cubes positioned such that they share exactly one common edge, for example, should produce four faces sharing that common edge. This configuration is not allowed in some systems, however, such as those based on the winged-edge representation. Weiler [WEIL88] describes a nonmanifold, boundary-based modeling system that can handle wireframe objects, in addition to surfaces and solids, as illustrated in Color Plate I.32.

12.6 SPATIAL-PARTITIONING REPRESENTATIONS

In *spatial-partitioning* representations, a solid is decomposed into a collection of adjoining, nonintersecting solids that are more primitive than, although not necessarily of the same type as, the original solid. Primitives may vary in type, size, position, parameterization, and orientation, much like the different-shaped blocks in a child's block set. How far we decompose objects depends on how primitive the solids must be in order to perform readily the operations of interest.

12.6.1 Cell Decomposition

One of the most general forms of spatial partitioning is called *cell decomposition*. Each cell-decomposition system defines a set of primitive cells that are typically parameterized and are often curved. Cell decomposition differs from primitive instancing in that we can compose more complex objects from simple, primitive ones in a bottom-up fashion by "gluing" them together. The *glue* operation can be thought of as a restricted form of union in which the objects must not intersect. Further restrictions on gluing cells often require that two cells share a single point, edge, or face. Although cell-decomposition representation of an object is unambiguous, it is not necessarily unique, as shown in Fig. 12.19. Cell

Samsung/Google, EX1027, p. 560

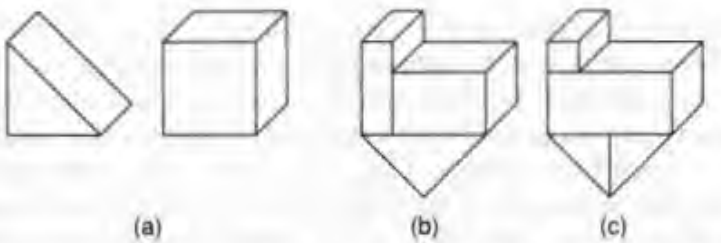


Fig. 12.19 The cells shown in (a) may be transformed to construct the same object shown in (b) and (c) in different ways. Even a single cell type is enough to cause ambiguity.

decompositions are also difficult to validate, since each pair of cells must potentially be tested for intersection. Nevertheless, cell decomposition is an important representation for use in finite element analysis.

12.6.2 Spatial-Occupancy Enumeration

Spatial-occupancy enumeration is a special case of cell decomposition in which the solid is decomposed into identical cells arranged in a fixed, regular grid. These cells are often called *voxels* (volume elements), in analogy to pixels. Figure 12.20 shows an object represented by spatial-occupancy enumeration. The most common cell type is the cube, and the representation of space as a regular array of cubes is called a *cuberille*. When representing an object using spatial-occupancy enumeration, we control only the presence or absence of a single cell at each position in the grid. To represent an object, we need only to decide which cells are occupied and which are not. The object can thus be encoded by a unique and unambiguous list of occupied cells. It is easy to find out whether a cell is inside or outside of the solid, and determining whether two objects are adjacent is simple as well. Spatial-

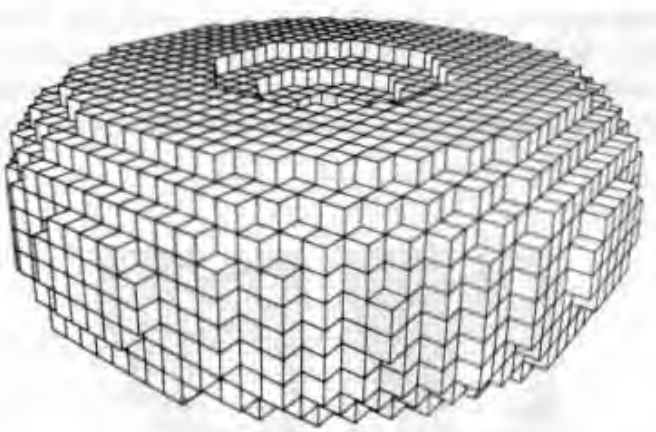


Fig. 12.20 Torus represented by spatial-occupancy enumeration. (By AHJ Christensen, SIGGRAPH '80 Conference Proceedings, *Computer Graphics* (14)3, July 1980. Courtesy of Association for Computing Machinery, Inc.)

Samsung/Google, EX1027, p. 561

occupancy enumeration is often used in biomedical applications to represent volumetric data obtained from sources such as computerized axial tomography (CAT) scans.

For all of its advantages, however, spatial-occupancy enumeration has a number of obvious failings that parallel those of representing a 2D shape by a 1-bit-deep bitmap. There is no concept of "partial" occupancy. Thus, many solids can be only approximated; the torus of Fig. 12.20 is an example. If the cells are cubes, then the only objects that can be represented exactly are those whose faces are parallel to the cube sides and whose vertices fall exactly on the grid. Like pixels in a bitmap, cells may in principle be made as small as desired to increase the accuracy of the representation. Space becomes an important issue, however, since up to n^3 occupied cells are needed to represent an object at a resolution of n voxels in each of three dimensions.

12.6.3 Octrees

Octrees are a hierarchical variant of spatial-occupancy enumeration, designed to address that approach's demanding storage requirements. Octrees are in turn derived from *quadtrees*, a 2D representation format used to encode images (see Section 17.7). As detailed in Samet's comprehensive survey [SAME84], both representations appear to have been discovered independently by a number of researchers, quadtrees in the late 1960s to early 1970s [e.g., WARN69; KLIN71] and octrees in the late 1970s to early 1980s [e.g., HUNT78; REDD78; JACK80; MEAG80; MEAG82a].

The fundamental idea behind both the quadtree and octree is the divide-and-conquer power of binary subdivision. A quadtree is derived by successively subdividing a 2D plane in both dimensions to form quadrants, as shown in Fig. 12.21. When a quadtree is used to represent an area in the plane, each quadrant may be full, partially full, or empty (also called black, gray, and white, respectively), depending on how much of the quadrant intersects the area. A partially full quadrant is recursively subdivided into subquadrants. Subdivision continues until all quadrants are homogeneous (either full or empty) or until a predetermined cutoff depth is reached. Whenever four sibling quadrants are uniformly full or empty, they are deleted and their partially full parent is replaced with a full or empty node. (A bottom-up approach can be used instead to avoid this deletion and merging process [SAME90b].) In Fig. 12.21, any partially full node at the cutoff depth is classified as full. The successive subdivisions can be represented as a tree with partially full quadrants at the internal nodes and full and empty quadrants at the leaves, as shown in Fig. 12.22.

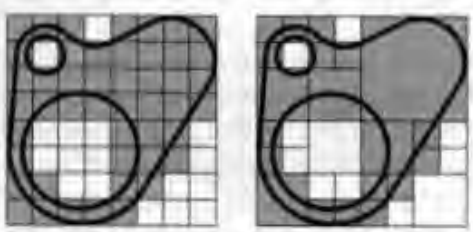


Fig. 12.21 An object represented using (a) spatial-occupancy enumeration (b) a quadtree.

Samsung/Google, EX1027, p. 562

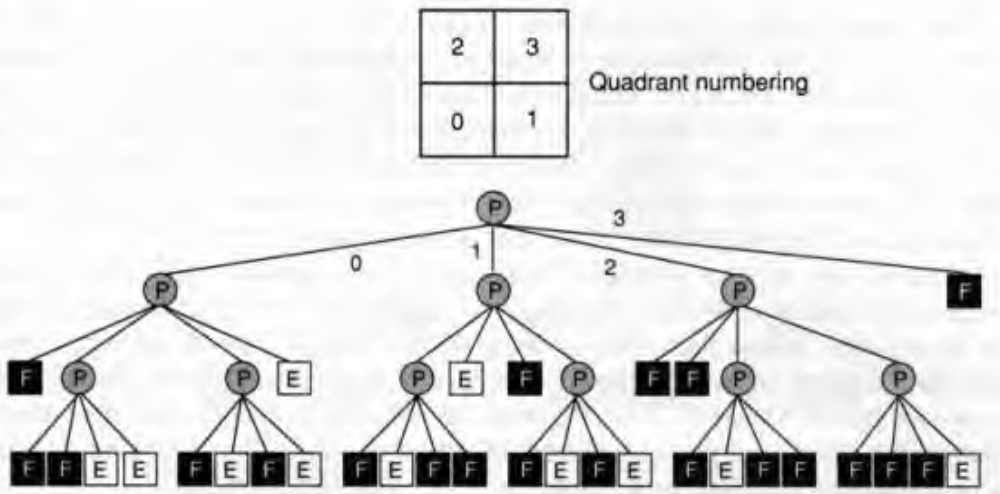


Fig. 12.22 Quadtree data structure for the object in Fig. 12.21. F = full, P = partially full, E = empty.

This idea can be compared to the Warnock area-subdivision algorithm discussed in Section 15.7.1. If the criteria for classifying a node as homogeneous are relaxed, allowing nodes that are above or below some threshold to be classified as full or empty, then the representation becomes more compact, but less accurate. The octree is similar to the quadtree, except that its three dimensions are recursively subdivided into octants, as shown in Fig. 12.23.

Quadrants are often referred to by the numbers 0 to 3, and octants by numbers from 0 to 7. Since no standard numbering scheme has been devised, mnemonic names are also used. Quadrants are named according to their compass direction relative to the center of their parent: NW, NE, SW, and SE. Octants are named similarly, distinguishing between left (L) and right (R), up (U) and down (D), and front (F) and back (B): LUF, LUB, LDF, LDB, RUF, RUB, RDF, and RDB.

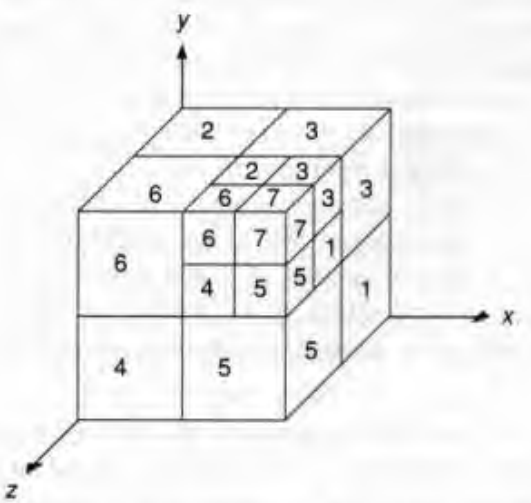


Fig. 12.23 Octree enumeration. Octant 0 is not visible.

Samsung/Google, EX1027, p. 563

With the exception of a few worst cases, it can be shown that the number of nodes in a quadtree or octree representation of an object is proportional to the object's perimeter or surface, respectively [HUNT78; MEAG80]. This relation holds because node subdivision arises only from the need to represent the boundary of the object being encoded. The only internal nodes that are split are those through which part of the boundary passes. Thus, any operation on one of these data structures that is linear in the number of nodes it contains also executes in time proportional to the size of its perimeter or area.

Although the divide-and-conquer approach of the quadtree and octree can be generalized to objects of arbitrary dimension, it is also possible to represent objects using only binary trees. Rather than dividing along all axes at each node of the tree, a *bintree* partitions space into equal halves about a single axis at each node, cycling through a new axis at each level [TAMM84]. A bintree often has more nodes than its equivalent quadtree or octree, but has at most the same number of leaves; as well, many processing algorithms can be formulated more simply for bintrees.

Boolean set operations and transformations. Much work has been done on developing efficient algorithms for storing and processing quadtrees and octrees [SAME84; SAME90a; SAME90b]. For example, Boolean set operations are straightforward for both quadtrees and octrees [HUNT79]. To compute the union or intersection U of two trees, S and T, we traverse both trees top-down in parallel. Figure 12.24 shows the operations for quadtrees; the generalization to octrees is straightforward. Each matching pair of nodes is examined. Consider the case of union. If either of the nodes in the pair is black, then a corresponding black node is added to U. If one of the pair's nodes is white, then the corresponding node is created in U with the value of the other node in the pair. If both nodes of the pair are gray, then a gray node is added to U, and the algorithm is applied recursively to the pair's children. In this last case, the children of the new node in U must be inspected after the algorithm has been applied to them. If they are all black, they are deleted and their parent in U is changed from gray to black. The algorithm for performing intersection is similar, except the roles of black and white are interchanged. If either of the nodes in a pair is white, then a corresponding white node is added to U. If one of the pair's nodes is black, then the corresponding node is created in U with the value of the other node in the pair. If both nodes of the pair are gray, then a gray node is added to U, and the algorithm is applied recursively to the pair's children. As in the union algorithm, if both nodes are gray, then after the algorithm has been applied to the children of the new node in U, the children must be inspected. In this case, if they are all white, they are deleted and their parent in U must be changed from gray to white.

It is easy to perform simple transformations on quadtrees and octrees. For example, rotation about an axis by multiples of 90° is accomplished by recursively rotating the children at each level. Scaling by powers of 2 and reflections are also straightforward. Translations are somewhat more complex, as are general transformations. In addition, as in spatial-occupancy enumeration in general, the problem of aliasing under general transformations is severe.

Neighbor finding. One important operation in quadtrees and octrees is finding a node's *neighbor*; that is, finding a node that is adjacent to the original node (sharing a face, edge, or vertex) and of equal or greater size. A quadtree node has neighbors in eight possible

Samsung/Google, EX1027, p. 564

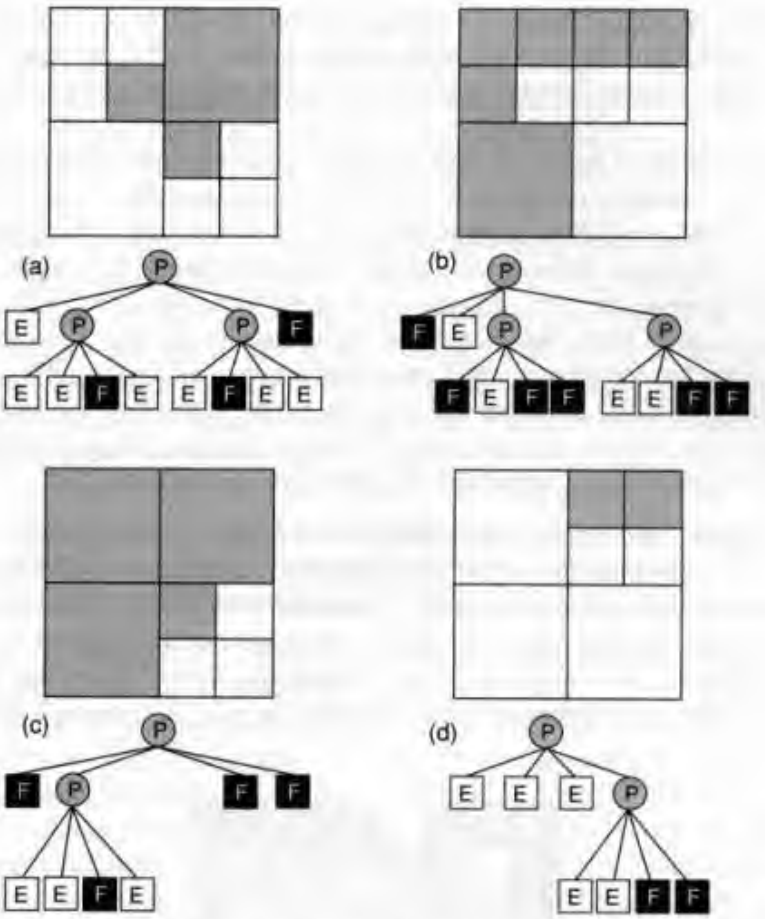


Fig. 12.24 Performing Boolean set operations on quadtrees. (a) Object S and its quadtree. (b) Object T and its quadtree. (c) $S \cup T$. (d) $S \cap T$.

directions. Its N, S, E, W neighbors are neighbors along a common edge, whereas its NW, NE, SW, and SE neighbors are neighbors along a common vertex. An octree node has neighbors in 26 possible directions: 6 neighbors along a face, 12 neighbors along an edge, and 8 neighbors along a vertex.

Samet [SAME89a] describes a way to find a node's neighbor in a specified direction. The method starts at the original node and ascends the quadtree or octree until the first common ancestor of the original node and neighbor is found. The tree is then traversed downward to find the desired neighbor. Two problems must be solved efficiently here: finding the common ancestor and determining which of its descendants is the neighbor. The simplest case is finding an octree node's neighbor in the direction d of one of its faces: L, R, U, D, F, or B. As we ascend the tree starting at the original node, the common ancestor will be the first node that is not reached from a child on the node's d side. For example, if the search is for an L neighbor, then the first common ancestor is the first node that is not reached from an LUF, LUB, LDF, or LDB child. This is true because a node that has been reached from one of these children cannot have any child that is left of (is an L neighbor of)

Samsung/Google, EX1027, p. 565

the original node. When the common ancestor is found, its subtree is descended in a mirror image of the path from the original node to the ancestor, reflected about the common border. Only part of the reflected path is followed if the neighbor is larger than the original node.

A similar method can be used to find a quadtree node's neighbor in the direction of one of its edges. For example, to find the N neighbor of node *A* of Fig. 12.25, we begin at *A*, which is a NW child, and follow the path depicted by the thick line in the figure. We ascend from the NW to its parent, from the NW again to its grandparent, and finally from the SW to its great grandparent, the root, stopping because we have approached it from an S node, rather than from an N node. We then follow the mirror-image path downward (reflected about the N–S border), to the root's NW child, and finally to this node's SW child, which is a leaf. Samet [SAME89a] describes the more elaborate algorithms for finding edge and vertex neighbors in octrees, and provides an elegant recursive implementation that uses table lookup to perform such operations as computing the reflected path.

Linear notations. Although a tree data structure with pointers might at first seem necessary to represent a quadtree or octree, pointerless notations are possible. In the *linear quadtree* or *linear octree* notation [GARG82], each full node is represented as a sequence of digits that represents its fully qualified address. There are as many digits as there are levels. Only black leaf nodes need to be stored to represent the object. Nodes that are not at the lowest level include an additional padding character (e.g., "X") in each of their trailing

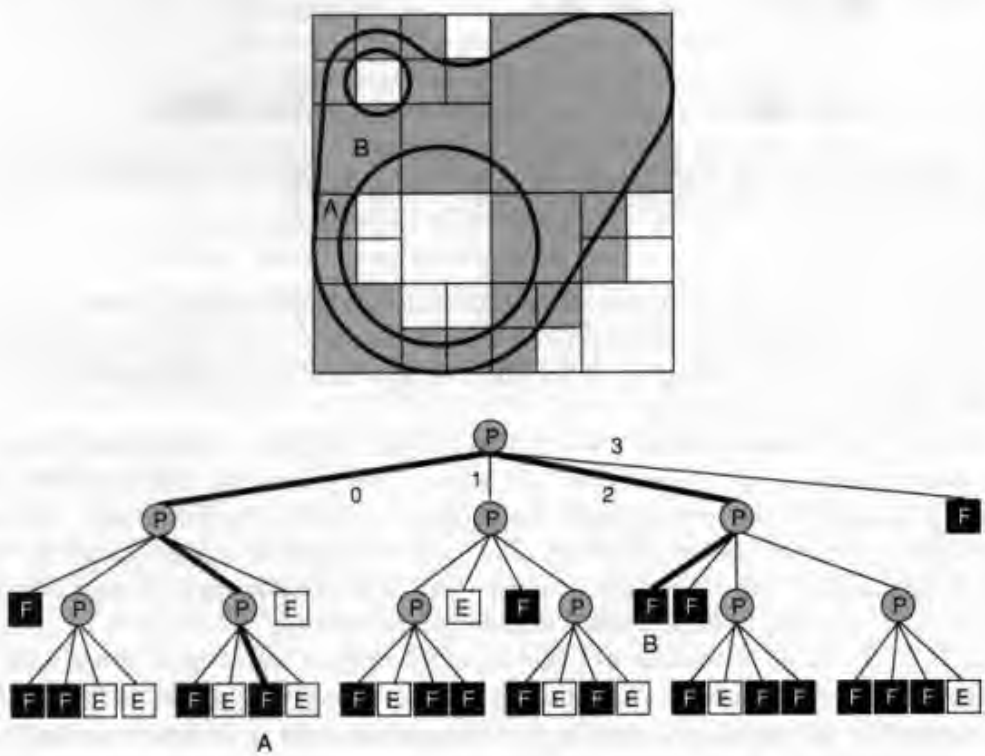


Fig. 12.25 Finding the neighbor of a quadtree node.

Samsung/Google, EX1027, p. 566

digits. For example, a linear octree can be encoded compactly using base-9 numbers (conveniently represented with 4 bits per digit), with 0 through 7 specifying octants and 8 indicating padding. The nodes in the linear octree are stored in sorted order, which represents a postorder traversal of the octree. For example, the linear quadtree representation of the object in Fig. 12.21 is 00X, 010, 011, 020, 022, 100, 102, 103, 12X, 130, 132, 20X, 21X, 220, 222, 223, 230, 231, 232, 3XX.

A number of operations can be performed efficiently with the linear-quadtree or linear-octree representation. For example, Atkinson, Gargantini, and Ramanath [ATKI84] present an algorithm for determining the voxels that form an octree's border by making successive passes over a list of the octree's nodes. Full nodes of the largest size are considered first. Each such node that abuts full nodes of the same size on all six sides is internal to the object and is therefore not part of the border; it is eliminated from the list. (Each neighbor's code may be derived by simple arithmetic manipulation of the node's code.) Any other node of this size may contain voxels that are part of the border; each of these nodes is broken into its eight constituent nodes, which replace it in the list. The algorithm is repeated for successively smaller node sizes, stopping after voxel-sized nodes are considered. Those nodes remaining are all the voxels on the object's border.

PM octrees. A number of researchers have developed hybrid representations that combine octrees and b-reps to maintain the precise geometry of the original b-rep from which the object is derived [HUNT81; QUIN82; AYAL85; CARL85; FUJI85]. These *PM octrees* (*PM* stands for Polygonal Map) expand on a similar quadtree variant [SAME90a]. The octree is recursively divided into nodes until the node is one of five different leaf types. In addition to full and empty, three new leaf nodes are introduced that are actually special kinds of partially full nodes: vertex nodes, which contain a single vertex and its connected faces and edges; edge nodes, which contain part of a single edge and its faces; and surface nodes, which are cut by a piece of a single face. Restricting the new leaf types to a set of simple geometries, each of which divides the node into exactly two parts, simplifies the algorithms that manipulate the representation, such as Boolean set operations [CARL87; NAVA89].

Section 18.11.4 discusses a number of architectures based on voxel and octree models. Section 15.8 discusses visible-surface algorithms for octrees.

12.6.4 Binary Space-Partitioning Trees

Octrees recursively divide space by planes that are always mutually perpendicular and that bisect all three dimensions at each level of the tree. In contrast, *binary space-partitioning* (BSP) *trees* recursively divide space into pairs of subspaces, each separated by a plane of arbitrary orientation and position. The binary-tree data structure created was originally used in determining visible surfaces in graphics, as described in Section 15.5.2. Thibault and Naylor [THIB87] later introduced the use of BSP trees to represent arbitrary polyhedra. Each internal node of the BSP tree is associated with a plane and has two child pointers, one for each side of the plane. Assuming that normals point out of an object, the left child is behind or inside the plane, whereas the right child is in front of or outside the plane. If the half-space on a side of the plane is subdivided further, then its child is the root of a subtree; if the half-space is homogeneous, then its child is a leaf, representing a region either

Samsung/Google, EX1027, p. 567

entirely inside or entirely outside the polyhedron. These homogeneous regions are called "in" cells and "out" cells. To account for the limited numerical precision with which operations are performed, each node also has a "thickness" associated with its plane. Any point lying within this tolerance of the plane is considered to be "on" the plane.

The subdivision concept behind BSP trees, like that underlying octrees and quadtrees, is dimension-independent. Thus, Fig. 12.26(a) shows a concave polygon in 2D, bordered by black lines. "In" cells are shaded light gray, and the lines defining the half-spaces are shown in dark gray, with normals pointing to the outside. The corresponding BSP tree is shown in Fig. 12.26(b). In 2D, the "in" and "out" regions form a convex polygonal tessellation of the plane; in 3D, the "in" and "out" regions form a convex polyhedral tessellation of 3-space. Thus, a BSP tree can represent an arbitrary concave solid with holes as a union of convex "in" regions. Unlike octrees, but like b-reps, an arbitrary BSP tree does not necessarily represent a bounded solid. For example, the 3D BSP tree consisting of a single internal node, with "in" and "out" nodes as children, defines an object that is a half-space bounded by only one plane.

Consider the task of determining whether a point lies inside, outside, or on a solid, a problem known as *point classification* [TILO80]. A BSP tree may be used to classify a point by filtering that point down the tree, beginning at the root. At each node, the point is substituted into the node's plane equation and is passed recursively to the left child if it lies behind (inside) the plane, or to the right child if it lies in front of (outside) the plane. If the node is a leaf, then the point is given the leaf's value, either "out" or "in." If the point lies on a node's plane, then it is passed to both children, and the classifications are compared. If they are the same, then the point receives that value; if they are different, then the point lies on the boundary between "out" and "in" regions and is classified as "on." This approach can be extended to classify lines and polygons. Unlike a point, however, a line or polygon may lie partially on both sides of a plane. Therefore, at each node whose plane intersects the line or polygon, the line or polygon must be divided (clipped) into those parts that are in front of, in back of, or on the plane, and the parts classified separately.

Thibault and Naylor describe algorithms for building a BSP tree from a b-rep, for performing Boolean set operations to combine a BSP tree with a b-rep, and for determining those polygonal pieces that lie on a BSP tree's boundary [THIB87]. These algorithms operate on BSP trees whose nodes are each associated with a list of polygons embedded in

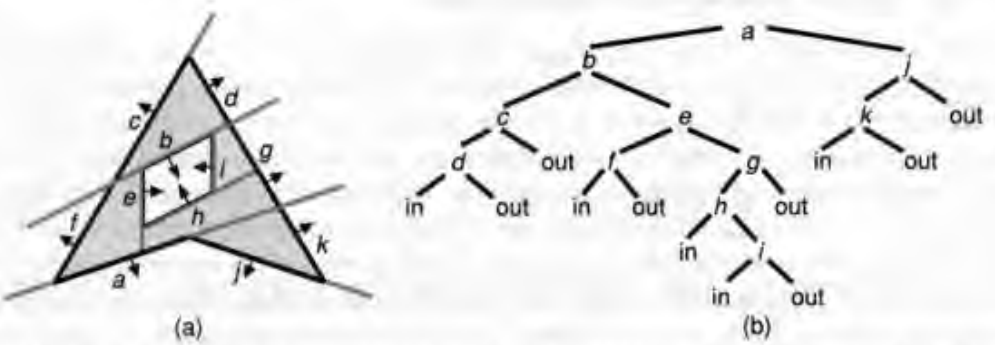


Fig. 12.26 A BSP tree representation in 2D. (a) A concave polygon bounded by black lines. Lines defining the half-spaces are dark gray, and "in" cells are light gray. (b) The BSP tree.

Samsung/Google, EX1027, p. 568

the node's plane. Polygons are inserted into the tree using a variant of the BSP tree building algorithm presented in Section 15.5.2.

Although BSP trees provide an elegant and simple representation, polygons are subdivided as the tree is constructed and as Boolean set operations are performed, making the notation potentially less compact than other representations. By taking advantage of the BSP tree's inherent dimension-independence, however, we can develop a closed Boolean algebra for 3D BSP trees that recursively relies on representing polygons as 2D trees, edges as 1D trees, and points as 0D trees [NAYL90].

12.7 CONSTRUCTIVE SOLID GEOMETRY

In *constructive solid geometry* (CSG), simple primitives are combined by means of regularized Boolean set operators that are included directly in the representation. An object is stored as a tree with operators at the internal nodes and simple primitives at the leaves (Fig. 12.27). Some nodes represent Boolean operators, whereas others perform translation, rotation, and scaling, much like the hierarchies of Chapter 7. Since Boolean operations are not, in general, commutative, the edges of the tree are ordered.

To determine physical properties or to make pictures, we must be able to combine the properties of the leaves to obtain the properties of the root. The general processing strategy is a depth-first tree walk, as in Chapter 7, to combine nodes from the leaves on up the tree. The complexity of this task depends on the representation in which the leaf objects are stored and on whether a full representation of the composite object at the tree's root must actually be produced. For example, the regularized Boolean set operation algorithms for b-reps, discussed in Section 12.5, combine the b-reps of two nodes to create a third b-rep and are difficult to implement. The much simpler CSG algorithm discussed in Section 15.10.3, on the other hand, produces a picture by processing the representations of the leaves without explicitly combining them. Other algorithms for creating pictures of CSG representations include [ATHE83; OKIN84; JANS85]; architectures that support CSG are discussed in Sections 18.9.2, 18.10.2 and 18.11.4.

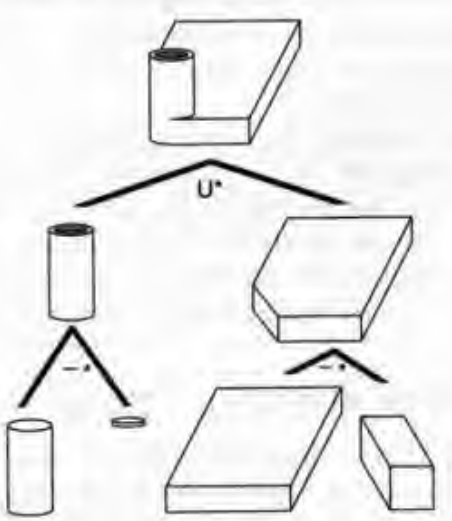


Fig. 12.27 An object defined by CSG and its tree.

Samsung/Google, EX1027, p. 569

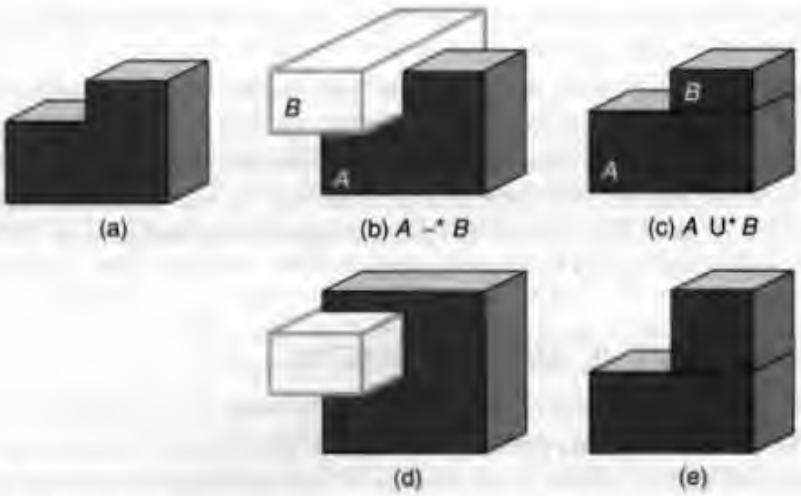


Fig. 12.28 The object shown in (a) may be defined by different CSG operations, as shown in (b) and (c). Tweaking the top face of (b) and (c) upward yields different objects, shown in (d) and (e).

In some implementations, the primitives are simple solids, such as cubes or spheres, ensuring that all regularized combinations are valid solids as well. In other systems, primitives include half-spaces, which themselves are not bounded solids. For example, a cube can be defined as the intersection of six half-spaces, or a finite cylinder as an infinite cylinder that is capped off at the top and bottom by planar half-spaces. Using half-spaces introduces a validity problem, since not all combinations produce solids. Half-spaces are useful, however, for operations such as slicing an object by a plane, which might otherwise be performed by using the face of another solid object. Without half-spaces, extra overhead is introduced, since the regularized Boolean set operations must be performed with the full object doing the slicing, even if only a single slicing face is of interest.

We can think of the cell-decomposition and spatial-occupancy enumeration techniques as special cases of CSG in which the only operator is the implicit glue operator: the union of two objects that may touch, but must have disjoint interiors (i.e., the objects must have a null regularized Boolean intersection).

CSG does not provide a unique representation. This can be particularly confusing in a system that lets the user manipulate the leaf objects with tweaking operators. Applying the same operation to two objects that are initially the same can yield two different results, as shown in Fig. 12.28. Nevertheless, the ability to edit models by deleting, adding, replacing, and modifying subtrees, coupled with the relatively compact form in which models are stored, have made CSG one of the dominant solid modeling representations.

12.8 COMPARISON OF REPRESENTATIONS

We have discussed five main kinds of representations: primitive instancing, sweeps, b-reps, spatial partitioning (including cell decomposition, spatial-occupancy enumeration, octrees, and BSP trees), and CSG. Let us compare them on the basis of the criteria introduced in Section 12.1.

Samsung/Google, EX1027, p. 570

- *Accuracy*. Spatial-partitioning and polygonal b-rep methods produce only approximations for many objects. In some applications, such as finding a path for a robot, this is not a drawback, as long as the approximation is computed to an adequate (often relatively coarse) resolution. The resolution needed to produce visually pleasing graphics or to calculate object interactions with sufficient accuracy, however, may be too high to be practical. The smooth shading techniques discussed in Chapter 16 do not fix the visual artifacts caused by the all-too-obvious polygonal edges. Therefore, systems that support high-quality graphics often use CSG with nonpolyhedral primitives and b-reps that allow curved surfaces. Primitive instancing also can produce high-quality pictures, but does not allow two simpler objects to be combined with Boolean set operators.
- *Domain*. The domain of objects that can be represented by both primitive instancing and sweeps is limited. In comparison, spatial-partitioning approaches can represent any solid, although often only as an approximation. By providing other kinds of faces and edges in addition to polygons bounded by straight lines, b-reps can be used to represent a very wide class of objects. Many b-rep systems, however, are restricted to simple surface types and topologies. For example, they may be able to encode only combinations of quadrics that are 2-manifolds.
- *Uniqueness*. Only octree and spatial-occupancy–enumeration approaches guarantee the uniqueness of a representation: There is only one way to represent an object with a specified size and position. In the case of octrees, some processing must be done to ensure that the representation is fully reduced (i.e., that no gray node has all black children or all white children). Primitive instancing does not guarantee uniqueness in general: for example, a sphere may be represented by both a spherical and an elliptical primitive. If the set of primitives is chosen carefully, however, uniqueness can be ensured.
- *Validity*. Among all the representations, b-reps stand out as being the most difficult to validate. Not only may vertex, edge, and face data structures be inconsistent, but also faces or edges may intersect. In contrast, any BSP tree represents a valid spatial set, but not necessarily a bounded solid. Only simple local syntactic checking needs to be done to validate a CSG tree (which is always bounded, if its primitives are bounded) or an octree, and no checking is needed for spatial-occupancy enumeration.
- *Closure*. Primitives created using primitive instancing cannot be combined at all, and simple sweeps are not closed under Boolean operations. Therefore, neither is typically used as an internal representation in modeling systems. Although particular b-reps may suffer from closure problems under Boolean operations (e.g., the inability to represent other than 2-manifolds), these problem cases can often be avoided.
- *Compactness and efficiency*. Representation schemes are often classified by whether they produce "evaluated" or "unevaluated" models. *Unevaluated* models contain information that must be further processed (or evaluated) in order to perform basic operations, such as determining an object's boundary. With regard to the use of Boolean operations, CSG creates unevaluated models, in that each time computations are performed, we must walk the tree, evaluating the expressions. Consequently, the advantages of CSG are its compactness and the ability to record Boolean operations and changes of transformations quickly, and to undo all of these quickly since they involve only tree-node building. Octrees and BSP trees can also be considered

Samsung/Google, EX1027, p. 571

unevaluated models, as can a sequence of Euler operators that creates a b-rep. B-reps and spatial-occupancy enumeration, on the other hand, are often considered *evaluated* models insofar as any Boolean operations used to create an object have already been performed. Note that the use of these terms is relative; if the operation to be performed is determining whether a point is inside an object, for example, more work may be done evaluating a b-rep than evaluating the equivalent CSG tree.

As discussed in Chapter 15, a number of efficient algorithms exist for generating pictures of objects encoded using b-reps and CSG. Although spatial-occupancy enumeration and octrees can provide only coarse approximations for most objects, the algorithms used to manipulate them are in general simpler than the equivalents for other representations. They have thus been used in hardware-based solid modeling systems intended for applications in which the increased speed with which Boolean set operations can be performed on them outweighs the coarseness of the resulting images.

Some systems use multiple representations because some operations are more efficient with one representation than with another. For example, GMSOLID [BOYS82] uses CSG for compactness and a b-rep for quick retrieval of useful data not explicitly specified in CSG, such as connectivity. Although GMSOLID's CSG representation always reflects the current state of the object being modeled, its b-rep is updated only when the operations that require it are executed. Updating may be done by a background process, so that the user can perform other operations while waiting for the result. In addition to systems that maintain two completely separate representations, deriving one from the other when needed, there are also hybrid systems that go down to some level of detail in one scheme, then switch to another, but never duplicate information. The PM octrees discussed in Section 12.6.3 that combine octrees with b-reps provide examples. Some of the issues raised by the use of multiple representations and hybrid representations are addressed in [MILL89].

It is relatively easy to convert all the representations we have discussed to spatial-occupancy–enumeration or octree representations, but only as approximations. Such conversions are not invertible, because they lose information. In addition, it is easy to convert all representations exactly to b-reps and PM octrees. An algorithm for performing Boolean operations on b-reps, such as the one described in Section 12.5, can be used to convert CSG to b-rep by successive application to each level of the CSG tree, beginning with the polyhedral descriptions of the leaf primitives. Rossignac and Voelcker [ROSS89] have implemented an efficient CSG–to–b-rep conversion algorithm that identifies what they call the *active zone* of a CSG node—that part of the node that, if changed, will affect the final solid; only the parts of a solid within a node's active zone need be considered when Boolean operations are performed. On the other hand, conversion from b-rep into CSG is difficult, especially if an encoding into a minimal number of CSG operations is desired. Vossler [VOSS85b] describes a method for converting sweeps to CSG by automatically recognizing patterns of simpler sweeps that can be combined with Boolean operations to form the more complex sweep being converted.

As pointed out in Section 12.1, wireframe representations containing only vertex and edge information, with no reference to faces, are inherently ambiguous. Markowsky and Wesley, however, have developed an algorithm for deriving all polyhedra that could be represented by a given wireframe [MARK80] and a companion algorithm that generates all polyhedra that could produce a given 2D projection [WESL81].

Samsung/Google, EX1027, p. 572

12.9 USER INTERFACES FOR SOLID MODELING

Developing the user interface for a solid modeling system provides an excellent opportunity to put into practice the interface design techniques discussed in Chapter 9. A variety of techniques lend themselves well to graphical interfaces, including the direct application of regularized Boolean set operators, tweaking, and Euler operators. In CSG systems the user may be allowed to edit the object by modifying or replacing one of the leaf solids or subtrees. Blending and chamfering operations may be defined to smooth the transition from one surface to another. The user interfaces of successful systems are largely independent of the internal representation chosen. Primitive instancing is an exception, however, since it encourages user to think of objects in terms of special-purpose parameters.

In Chapter 11, we noted that there are many equivalent ways to describe the same curve. For example, the user interface to a curve-drawing system can let the user enter curves by controlling Hermite tangent vectors or by specifying Bezier control points, while storing curves only as Bezier control points. Similarly, a solid modeling system may let the user create objects in terms of several different representations, while storing them in yet another. As with curve representations, each different input representation may have some expressive advantage that makes it a natural choice for creating the object. For example, a b-rep system may allow an object to be defined as a translational or rotational sweep. The user interface may also provide different ways to define the same object within a single representation. For example, two of the many ways to define a sphere are to specify its center and a point on its surface, or to specify the two endpoints of a diameter. The first may be more useful for centering a sphere at a point, whereas the second may be better for positioning the sphere between two supports.

The precision with which objects must be specified often dictates that some means be provided to determine measurements accurately; for example, through a locator device or through numeric entry. Because the position of one object often depends on those of others, interfaces often provide the ability to constrain one object by another. A related technique is to give the user the ability to define grid lines to constrain object positions, as discussed in Section 8.2.1.

Some of the most fundamental problems of designing a solid modeling interface are those caused by the need to manipulate and display 3D objects with what are typically 2D interaction devices and displays. These general issues were discussed in more detail in Chapters 8 and 9. Many systems address some of these problems by providing multiple display windows that allow the user to view the object simultaneously from different positions.

12.10 SUMMARY

As we have seen, solid modeling is important in both CAD/CAM and graphics. Although useful algorithms and systems exist that handle the objects described so far, many difficult problems remain unsolved. One of the most important is the issue of robustness. Solid modeling systems are typically plagued by numerical instabilities. Commonly used algorithms require more precision to hold intermediate floating-point results than is available in hardware. For example, Boolean set operation algorithms may fail when presented with two objects, one of which is a very slightly transformed copy of the first.

Samsung/Google, EX1027, p. 573

Representations are needed for nonrigid, flexible, jointed objects. Work on transformations that bend and twist objects is described in Chapter 20. Many objects cannot be specified with total accuracy; rather, their shapes are defined by parameters constrained to lie within a range of values. These are known as "toleranced" objects, and correspond to real objects turned out by machines such as lathes and stampers [REQU84]. New representations are being developed to encode toleranced objects [GOSS88].

Common to all designed objects is the concept of "features," such as holes and chamfers, that are designed for specific purposes. One current area of research is exploring the possibility of recognizing features automatically and inferring the designer's intent for what each feature should accomplish [PRAT84]. This will allow the design to be checked to ensure that the features perform as intended. For example, if certain features are designed to give a part strength under pressure, then their ability to perform this function could be validated automatically. Future operations on the object could also be checked to ensure that the features' functionality was not compromised.

EXERCISES

12.1 Define the results of performing $\cup^*$ and $-^*$ for two polyhedral objects in the same way as the result of performing $\cap^*$ was defined in Section 12.2. Explain how the resulting object is constrained to be a regular set, and specify how the normal is determined for each of the object's faces.

12.2 Consider the task of determining whether or not a legal solid is the null object (which has no volume). How difficult is it to perform this test in each of the representations discussed?

12.3 Consider a system whose objects are represented as sweeps and can be operated on using the regularized Boolean set operators. What restrictions must be placed on the objects to ensure closure?

12.4 Implement the algorithms for performing Boolean set operations on quadtrees or on octrees.

12.5 Explain why an implementation of Boolean set operations on quadtrees or octrees does not need to address the distinction between the ordinary and regularized operations described in Section 12.2.

12.6 Although the geometric implications of applying the regularized Boolean set operators are unambiguous, it is less clear how object properties should be treated. For example, what properties should be assigned to the intersection of two objects made of different materials? In modeling actual objects, this question is of little importance, but in the artificial world of graphics, it is possible to intersect any two materials. What solutions do you think would be useful?

12.7 Explain how a quadtree or octree could be used to speed up 2D or 3D picking in a graphics package.

12.8 Describe how to perform point classification in primitive instancing, b-rep, spatial occupancy enumeration, and CSG.

Samsung/Google, EX1027, p. 574

13 Achromatic and Colored Light

The growth of raster graphics has made color and gray scale an integral part of contemporary computer graphics. Color is an immensely complex subject, one that draws on concepts and results from physics, physiology, psychology, art, and graphic design. Many researchers' careers have been fruitfully devoted to developing theories, measurement techniques, and standards for color. In this chapter, we introduce some of the areas of color that are most relevant to computer graphics.

The color of an object depends not only on the object itself, but also on the light source illuminating it, on the color of the surrounding area, and on the human visual system. Furthermore, some objects reflect light (wall, desk, paper), whereas others also transmit light (cellophane, glass). When a surface that reflects only pure blue light is illuminated with pure red light, it appears black. Similarly, a pure green light viewed through glass that transmits only pure red will also appear black. We postpone some of these issues by starting our discussion with achromatic sensations—that is, those described as black, gray, and white.

13.1 ACHROMATIC LIGHT

Achromatic light is what we see on a black-and-white television set or display monitor. An observer of achromatic light normally experiences none of the sensations we associate with red, blue, yellow, and so on. Quantity of light is the only attribute of achromatic light. Quantity of light can be discussed in the physics sense of energy, in which case the terms *intensity* and *luminance* are used, or in the psychological sense of perceived intensity, in which case the term *brightness* is used. As we shall discuss shortly, these two concepts are

Samsung/Google, EX1027, p. 575

related but are not the same. It is useful to associate a scalar with different intensity levels, defining 0 as black and 1 as white; intensity levels between 0 and 1 represent different grays.

A black-and-white television can produce many different intensities at a single pixel position. Line printers, pen plotters, and electrostatic plotters produce only two levels: the white (or light gray) of the paper and the black (or dark gray) of the ink or toner deposited on the paper. Certain techniques, discussed in later sections, allow such inherently *bilevel* devices to produce additional intensity levels.

13.1.1 Selecting Intensities—Gamma Correction

Suppose we want to display 256 different intensities. Which 256 intensity levels should we use? We surely do not want 128 in the range of 0 to 0.1 and 128 more in the range of 0.9 to 1.0, since the transition from 0.1 to 0.9 would certainly appear discontinuous. We might initially distribute the levels evenly over the range 0 to 1, but this choice ignores an important characteristic of the eye: that it is sensitive to ratios of intensity levels rather than to absolute values of intensity. That is, we perceive the intensities 0.10 and 0.11 as differing just as much as the intensities 0.50 and 0.55. (This nonlinearity is easy to observe: Cycle through the settings on a three-way 50–100–150-watt lightbulb; you will see that the step from 50 to 100 seems much greater than the step from 100 to 150.) On a brightness (that is, perceived intensity) scale, the differences between intensities of 0.10 and 0.11 and between intensities of 0.50 and 0.55 are equal. Therefore, the intensity levels should be spaced logarithmically rather than linearly, to achieve equal steps in brightness.

To find 256 intensities starting with the lowest attainable intensity I_0 and going to a maximum intensity of 1.0, with each intensity r times higher than the preceding intensity, we use the following relations:

$$I_0 = I_0,\ I_1 = rI_0,\ I_2 = rI_1 = r^2I_0,\ I_3 = rI_2 = r^3I_0,\ \ldots,\ I_{255} = r^{255}I_0 = 1. \quad (13.1)$$

Therefore,

$$r = (1/I_0)^{1/255},\ I_j = r^jI_0 = (1/I_0)^{j/255}\,I_0 = I_0^{(255-j)/255} \qquad \text{for } 0 \le j \le 255, \quad (13.2)$$

and in general for $n + 1$ intensities,

$$r = (1/I_0)^{1/n},\ I_j = I_0^{(n-j)/n} \qquad \text{for } 0 \le j \le n. \quad (13.3)$$

With just four intensities ($n = 3$) and an I_0 of $\frac{1}{8}$ (an unrealistically large value chosen for illustration only), Eq. (13.3) tells us that $r = 2$, yielding intensity values of $\frac{1}{8}$, $\frac{1}{4}$, $\frac{1}{2}$, and 1.

The minimum attainable intensity I_0 for a CRT is anywhere from about $\frac{1}{200}$ up to $\frac{1}{40}$ of the maximum intensity of 1.0. Therefore, typical values of I_0 are between 0.005 and 0.025. The minimum is not 0, because of light reflection from the phosphor within the CRT. The ratio between the maximum and minimum intensities is called the *dynamic range*. The exact value for a specific CRT can be found by displaying a square of white on a field of black and measuring the two intensities with a photometer. This measurement is taken in a completely darkened room, so that reflected ambient light does not affect the intensities. With an I_0 of 0.02, corresponding to a dynamic range of 50, Eq. (13.2) yields $r = 1.0154595\ldots$, and the first few and last two intensities of the 256 intensities from Eq. (13.1) are 0.0200, 0.0203, 0.0206, 0.0209, 0.0213, 0.0216, . . ., 0.9848, 1.0000.

Samsung/Google, EX1027, p. 576

Displaying the intensities defined by Eq. (13.1) on a CRT is a tricky process, and recording them on film is even more difficult, because of the nonlinearities in the CRT and film. For instance, the intensity of light output by a phosphor is related to the number of electrons N in the beam by

$$I = kN^{\gamma} \tag{13.4}$$

for constants k and γ. The value of γ is in the range 2.2 to 2.5 for most CRTs. The number of electrons N is proportional to the control-grid voltage, which is in turn proportional to the value V specified for the pixel. Therefore, for some other constant K,

$$I = KV^{\gamma}, \text{ or } V = (I/K)^{1/\gamma}. \tag{13.5}$$

Now, given a desired intensity I, we first determine the nearest I_j by searching through a table of the available intensities as calculated from Eq. (13.1) or from its equivalent:

$$j = ROUND(\log_r(I/I_0)). \tag{13.6}$$

After j is found, we calculate

$$I_j = r^j I_0. \tag{13.7}$$

The next step is to determine the pixel value V_j needed to create the intensity I_j, by using Eq. (13.5):

$$V_j = ROUND\ ((I_j/K)^{1/\gamma}). \tag{13.8}$$

If the raster display has no look-up table, then V_j is placed in the appropriate pixels. If there is a look-up table, then j is placed in the pixel and V_j is placed in entry j of the table.

The values of K, γ, and I_0 depend on the CRT in use, so in practice the look-up table is loaded by a method based on actual measurement of intensities [CATM79; COWA83; HALL89]. Use of the look-up table in this general manner is called *gamma correction*, so named for the exponent in Eq. (13.4). If the display has hardware gamma correction, then I_j rather than V_j is placed in either the refresh buffer or look-up table.

Without the use of ratio-based intensity values and gamma correction, quantization errors (from approximating a true intensity value with a displayable intensity value) will be more conspicuous near black than near white. For instance, with 4 bits and hence 16 intensities, a quantization round-off error of as much as $\frac{1}{32} = 0.031$ is possible. This is 50 percent of intensity value $\frac{1}{16}$, and only 3 percent of intensity value 1.0. Using the ratio-based intensities and gamma correction, the maximum quantization error as a percentage of brightness (perceived intensity) is constant.

A natural question is, "How many intensities are enough?" By "enough," we mean the number needed to reproduce a continuous-tone black-and-white image such that the reproduction appears to be continuous. This appearance is achieved when the ratio r is 1.01 or less (below this ratio, the eye cannot distinguish between intensities I_j and I_{j+1}) [WYSZ82, p. 569]. Thus, the appropriate value for n, the number of intensity levels, is found by equating r to 1.01 in Eq. (13.3):

$$r = (1/I_0)^{1/n} \quad \text{or} \quad 1.01 = (1/I_0)^{1/n}. \tag{13.9}$$

Samsung/Google, EX1027, p. 577

TABLE 13.1 DYNAMIC RANGE $(1/I_0)$ AND NUMBER OF REQUIRED INTENSITIES $n = \log_{1.01}(1/I_0)$ FOR SEVERAL DISPLAY MEDIA

Display media	Typical dynamic range	Number of intensities, n
CRT	50–200	400–530
Photographic prints	100	465
Photographic slides	1000	700
Coated paper printed in B/W*	100	465
Coated paper printed in color	50	400
Newsprint printed in B/W	10	234

*B/W = black and white.

Solving for n gives

$$n = \log_{1.01}(1/I_0), \tag{13.10}$$

where $1/I_0$ is the dynamic range of the device.

The dynamic range $1/I_0$ for several display media, and the corresponding n, which is the number of intensity levels needed to maintain $r = 1.01$ and at the same time to use the full dynamic range, are shown in Table 13.1. These are theoretical values, assuming perfect reproduction processes. In practice, slight blurring due to ink bleeding and small amounts of random noise in the reproduction decreases n considerably for print media. For instance, Fig. 13.1 shows a continuous-tone photograph; and the succeeding five Figs. 13.2 to 13.6 reproduce the same photograph at 4, 8, 16, 32, and 64 intensity levels. With four and eight levels, the transitions or contours between one intensity level and the next are quite conspicuous, because the ratio r between successive intensities is considerably greater that the ideal 1.01. Contouring is barely detectable with 32 levels, and for these particular



Fig. 13.1 A continuous-tone photograph.



Fig. 13.2 A continuous-tone photograph reproduced with four intensity levels. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

Samsung/Google, EX1027, p. 578



Fig. 13.3 A continuous-tone photograph reproduced with eight intensity levels. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)



Fig. 13.4 A continuous-tone photograph reproduced with 16 intensity levels. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)



Fig. 13.5 A continuous-tone photograph reproduced with 32 intensity levels. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)



Fig. 13.6 A continuous-tone photograph reproduced with 64 intensity levels. Differences from the picture in Fig. 13.5 are quite subtle. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

images disappears with 64. This suggests that 64 intensity levels is the absolute minimum needed for contour-free printing of continuous-tone black-and-white images on paper such as is used in this book. For a well-adjusted CRT in a perfectly black room, however, the higher dynamic range means that many more levels are demanded.

Samsung/Google, EX1027, p. 579



Fig. 13.7 Enlarged halftone pattern. Dot sizes vary inversely with intensity of original photograph. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

13.1.2 Halftone Approximation

Many displays and hardcopy devices are bilevel—they produce just two intensity levels—and even 2- or 3-bit-per-pixel raster displays produce fewer intensity levels than we might desire. How can we expand the range of available intensities? The answer lies in the *spatial integration* that our eyes perform. If we view a very small area from a sufficiently large viewing distance, our eyes average fine detail within the small area and record only the overall intensity of the area.

This phenomenon is exploited in printing black-and-white photographs in newspapers, magazines, and books, in a technique called *halftoning* (also called *clustered-dot ordered dither*[1] in computer graphics). Each small resolution unit is imprinted with a circle of black ink whose area is proportional to the blackness $1 - I$ (where I = intensity) of the area in the original photograph. Figure 13.7 shows part of a halftone pattern, greatly enlarged. Note that the pattern makes a 45° angle with the horizontal, called the *screen angle*. Newspaper halftones use 60 to 80 variable-sized and variable-shaped areas [ULIC87] per inch, whereas halftones in magazines and books use 110 to 200 per inch.

Graphics output devices can approximate the variable-area circles of halftone reproduction. For example, a 2×2 pixel area of a bilevel display can be used to produce five different intensity levels at the cost of halving the spatial resolution along each axis. The patterns shown in Fig. 13.8 can be used to fill the 2×2 areas with the number of "on" pixels that is proportional to the desired intensity. Figure 13.9 shows a face digitized as a 351×351 image array and displayed with 2×2 patterns.

An $n \times n$ group of bilevel pixels can provide $n^2 + 1$ intensity levels. In general, there is a tradeoff between spatial resolution and intensity resolution. The use of a 3×3 pattern

[1]The "ordered dither" contrasts with "random dither," an infrequently used technique.

Samsung/Google, EX1027, p. 580

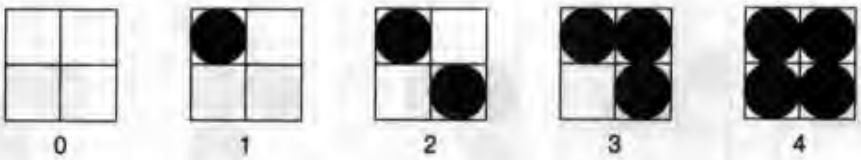


Fig. 13.8 Five intensity levels approximated with four 2 × 2 dither patterns.

cuts spatial resolution by one-third on each axis, but provides 10 intensity levels. Of course, the tradeoff choices are limited by our visual acuity (about 1 minute of arc in normal lighting), the distance from which the image is viewed, and the dots-per-inch resolution of the graphics device.

One possible set of patterns for the 3 × 3 case is shown in Fig. 13.10. Note that these patterns can be represented by the *dither matrix*

$$\begin{bmatrix} 6 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 2 & 7 \end{bmatrix}. \tag{13.11}$$

To display an intensity I, we turn on all pixels whose values are less than I.

The $n \times n$ pixel patterns used to approximate the halftones must be designed not to introduce visual artifacts in an area of identical intensity values. For instance, if the pattern in Fig. 13.11 were used, rather than the one in Fig. 13.10, horizontal lines would appear in any large area of the image of intensity 3. Second, the patterns must form a *growth sequence* so that any pixel intensified for intensity level j is also intensified for all levels $k > j$. This minimizes the differences in the patterns for successive intensity levels, thereby minimizing the contouring effects. Third, the patterns must grow outward from the center, to create the effect of increasing dot size. Fourth, for hardcopy devices such as laser printers and film recorders that are poor at reproducing isolated "on" pixels, all pixels that are "on" for a



Fig. 13.9 A continuous-tone photograph, digitized to a resolution of 351 × 351 and displayed using the 2 × 2 patterns of Fig. 13.8. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

Samsung/Google, EX1027, p. 581

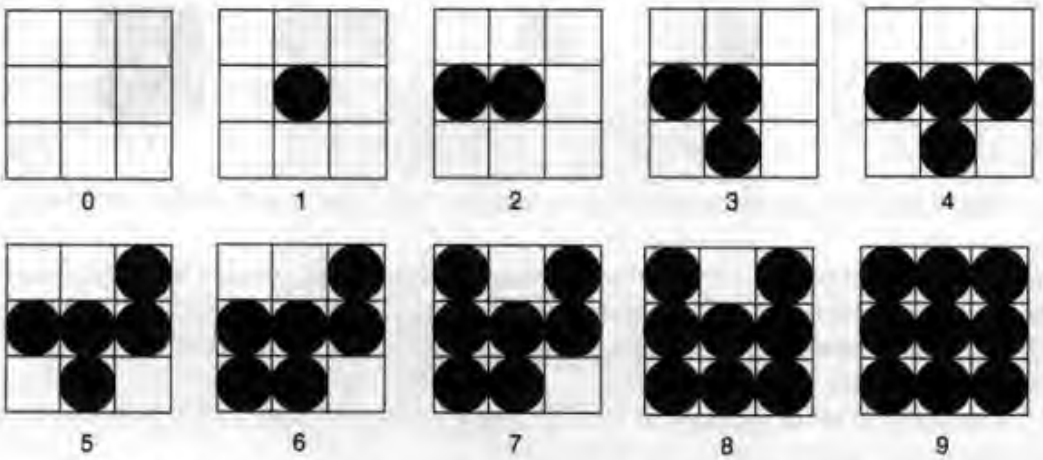


Fig. 13.10 Ten intensity levels approximated with 3 × 3 dither patterns.

particular intensity must be adjacent to other "on" pixels; a pattern such as that in Fig. 13.12 is not acceptable. This is the meaning of the term *clustered-dot* in "clustered-dot ordered dither." Holladay [HOLL80] has developed a widely used method for defining dither matrices for clustered-dot ordered dither. For high-quality reproduction of images, n must be 8 to 10, theoretically allowing 65 to 101 intensity levels. To achieve an effect equivalent to the 150-circle-per-inch printing screen, we thus require a resolution of from $150 \times 8 = 1200$ up to $150 \times 10 = 1500$ pixels per inch. High-quality film recorders can attain this resolution, but cannot actually show all the intensity levels because patterns made up of single black or white pixels may disappear.

Halftone approximation is not limited to bilevel displays. Consider a display with 2 bits per pixel and hence four intensity levels. The halftone technique can be used to increase the number of intensity levels. If we use a 2 × 2 pattern, we have a total of 4 pixels at our disposal, each of which can take on three values besides black; this allows us to display $4 \times 3 + 1 = 13$ intensities. One possible set of growth sequence patterns in this situation is shown in Fig. 13.13.

Unlike a laser printer, a CRT display is quite able to display individual dots. Hence, the clustering requirement on the patterns discussed previously can be relaxed and *dispersed-dot ordered dither* can be used. There are many possible dither matrices: Bayer [BAYE73] has developed dither matrices that minimize the texture introduced into the displayed images. For the 2 × 2 case, the dither matrix, called $D^{(2)}$, is

$$D^{(2)} = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}. \tag{13.12}$$

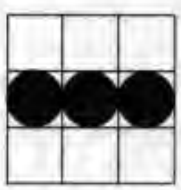


Fig. 13.11 A 3 × 3 dither pattern inappropriate for halftoning.

Samsung/Google, EX1027, p. 582

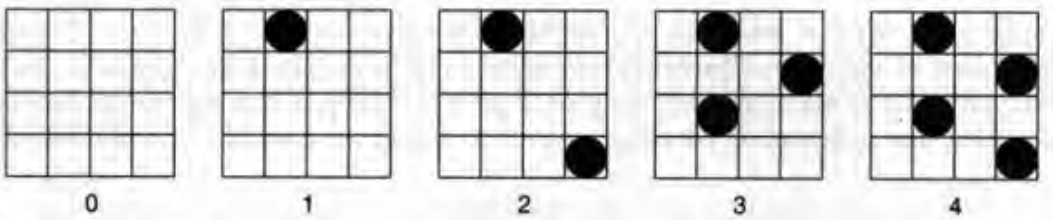


Fig. 13.12 Part of a 4 × 4 ordered dither dot pattern in which several of the patterns have single, nonadjacent dots. Such disconnected patterns are unacceptable for halftoning on laser printers and for printing presses.

This represents the set of patterns of Figure 13.8.

Larger dither matrices can be found using a recurrence relation [JUDI74] to compute $D^{(2n)}$ from $D^{(n)}$. With $U^{(n)}$ defined as an $n \times n$ matrix of 1s, that is,

$$U^{(n)} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ . & . & . & . & . \\ 1 & 1 & 1 & \dots & 1 \end{bmatrix}, \tag{13.13}$$

the recurrence relation is

$$D^{(n)} = \begin{bmatrix} 4D^{(n/2)} + D^{(2)}_{00}U^{(n/2)} & 4D^{(n/2)} + D^{(2)}_{01}U^{(n/2)} \\ 4D^{(n/2)} + D^{(2)}_{10}U^{(n/2)} & 4D^{(n/2)} + D^{(2)}_{11}U^{(n/2)} \end{bmatrix}. \tag{13.14}$$

Applying this relation to $D^{(2)}$ produces

$$D^{(4)} = \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix}. \tag{13.15}$$

The techniques presented thus far have assumed that the image array being shown is smaller than the display device's pixel array, so that multiple display pixels can be used for one

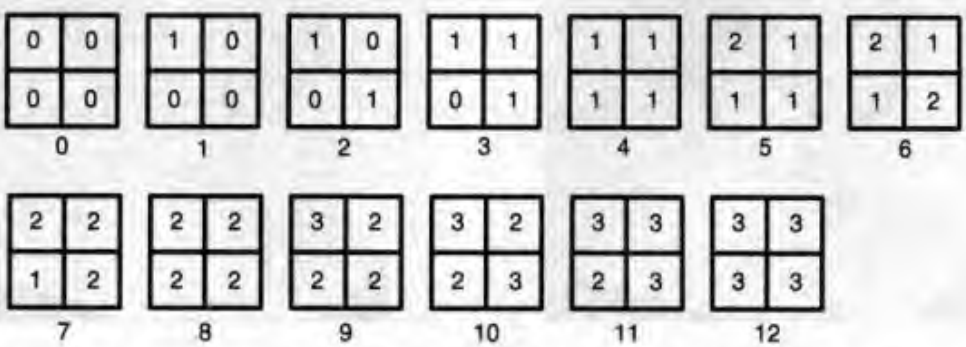


Fig. 13.13 Dither patterns for intensity levels 0 to 13 approximated using 2 × 2 patterns of pixels, with 2 bits (four intensity levels) per pixel. The intensities of the individual pixels sum to the intensity level for each pattern.

Samsung/Google, EX1027, p. 583

image pixel. What if the image and display device arrays are the same size? A simple adaptation of the ordered-dither (either clustered-dot or dispersed-dot) approach can be used. Whether or not to intensify the pixel at point (x, y) depends on the desired intensity $S(x, y)$ at that point and on the dither matrix. To display the point at (x, y), we compute

$$\begin{aligned} i &= x \text{ modulo } n, \\ j &= y \text{ modulo } n. \end{aligned} \tag{13.16}$$

Then, if

$$S(x, y) > D^{(n)}_{ij}, \tag{13.17}$$

the point at (x, y) is intensified; otherwise, it is not. That is, 1 pixel in the image array controls 1 pixel in the display array. Notice that large areas of fixed image intensity are displayed exactly as when the image-array size is less than the display-array size, so the effect of equal image and display arrays is apparent only in areas where intensity varies.

Figure 13.14(a) is a face digitized at 512 × 512 and shown on a 512 × 512 bilevel display using $D^{(8)}$. Compare this bilevel result to the multilevel pictures shown earlier in this section. Further pictures displayed by means of ordered dither appear in [JARV76a; JARV76b; ULIC87], where more ways to display continuous-tone images on bilevel displays are described.

Error diffusion, another way to handle the case when the image and display array sizes are equal, was developed by Floyd and Steinberg [FLOY75]; its visual results are often satisfactory. The error (i.e., the difference between the exact pixel value and the approximated value actually displayed) is added to the values of the four image-array pixels to the right of and below the pixel in question: $\frac{7}{16}$ of the error to the pixel to the right, $\frac{3}{16}$ to the pixel below and to the left, $\frac{5}{16}$ to the pixel immediately below, and $\frac{1}{16}$ to the pixel below



(a) (b)

Fig. 13.14 A continuous-tone photograph reproduced with (a) $D^{(8)}$ ordered dither, and (b) Floyd-Steinberg error diffusion. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

Samsung/Google, EX1027, p. 584

and to the right. This has the effect of spreading, or diffusing, the error over several pixels in the image array. Figure 13.14(b) was created using this method.

Given a picture S to be displayed in the intensity matrix I, the modified values in S and the displayed values in I are computed for pixels in scan-line order, working downward from the topmost scanline.

```
double error;
K = Approximate (S[x][y]);      /* Approximate S to nearest displayable intensity */
I[x][y] = K;                    /* Draw the pixel at (x, y). */
error = S[x][y] - K;            /* Error term */

/* Step 1: spread 7/16 of error into the pixel to the right, at (x + 1, y). */
S[x + 1][y] += 7 * error/16;

/* Step 2: spread 3/16 of error into pixel below and to the left. */
S[x - 1][y - 1] += 3 * error/16;

/* Step 3: spread 5/16 of error into pixel below. */
S[x][y - 1] += 5 * error/16;

/* Step 4: spread 1/16 of error below and to the right. */
S[x + 1][y - 1] += error/16;
```

To avoid introducing visual artifacts into the displayed image, we must ensure that the four errors sum exactly to *error*; no roundoff errors can be allowed. This can be done by calculating the step 4 error term as *error* minus the error terms from the first three steps. The function *Approximate* returns the displayable intensity value closest to the actual pixel value. For a bilevel display, the value of S is simply rounded to 0 or 1.

Even better results can be obtained by alternately scanning left to right and right to left; on a right-to-left scan, the left–right directions for errors in steps 1, 2, and 4 are reversed. For a detailed discussion of this and other error-diffusion methods, see [ULIC87]. Other approaches are discussed in [KNUT87].

Suppose the size of the image array is less than the size of the display array, the number of intensities in the image and display are equal, and we wish to display the image at the size of the display array. A simple case of this is an 8-bit-per-pixel, 512 × 512 image and an 8-bit-per-pixel, 1024 × 1024 display. If we simply replicate image pixels horizontally and vertically in the display array, the replicated pixels on the display will form squares that are quite obvious to the eye. To avoid this problem, we can interpolate intensities to calculate the pixel values. For instance, if an image S is to be displayed at double resolution, then the intensities I to display (with $x = 2x'$ and $y = 2y'$) are

$$I[x][y] = S[x'][y'];$$

$$I[x+1][y] = \tfrac{1}{2}(S[x'][y'] + S[x'+1][y']);$$

$$I[x][y+1] = \tfrac{1}{2}(S[x'][y'] + S[x'][y'+1]);$$

$$I[x+1][y+1] = \tfrac{1}{4}(S[x'][y'] + S[x'+1][y'] + S[x'][y'+1] + S[x'+1][y'+1]);$$

See Section 17.4 for a discussion of two-pass scaling transformations of images, and Section 3.17 for a description of the filtering and image-reconstruction techniques that are applicable to this problem.

Samsung/Google, EX1027, p. 585

13.2 CHROMATIC COLOR

The visual sensations caused by colored light are much richer than those caused by achromatic light. Discussions of color perception usually involve three quantities, known as hue, saturation, and lightness. *Hue* distinguishes among colors such as red, green, purple, and yellow. *Saturation* refers to how far color is from a gray of equal intensity. Red is highly saturated; pink is relatively unsaturated; royal blue is highly saturated; sky blue is relatively unsaturated. Pastel colors are relatively unsaturated; unsaturated colors include more white light than do the vivid, saturated colors. *Lightness* embodies the achromatic notion of perceived intensity of a reflecting object. *Brightness*, a fourth term, is used instead of lightness to refer to the perceived intensity of a self-luminous (i.e., emitting rather than reflecting light) object, such as a light bulb, the sun, or a CRT.

It is necessary to specify and measure colors if we are to use them precisely in computer graphics. For reflected light, we can do this by visually comparing a sample of unknown color against a set of "standard" samples. The unknown and sample colors must be viewed under a standard light source, since the perceived color of a surface depends both on the surface and on the light under which the surface is viewed. The widely used Munsell color-order system includes sets of published standard colors [MUNS76] organized in a 3D space of hue, value (what we have defined as lightness), and chroma (saturation). Each color is named, and is ordered so as to have an equal perceived "distance" in color space (as judged by many observers) from its neighbors. [KELL76] gives an extensive discussion of standard samples, charts depicting the Munsell space, and tables of color names.

In the printing industry and graphic design profession, colors are typically specified by matching to printed color samples. Color Plate I.33 is taken from a widely used color-matching system.

Artists often specify color as different tints, shades, and tones of strongly saturated, or pure, pigments. A *tint* results from adding white pigment to a pure pigment, thereby decreasing saturation. A *shade* comes from adding a black pigment to a pure pigment, thereby decreasing lightness. A *tone* is the consequence of adding both black and white pigments to a pure pigment. All these steps produce different colors of the same hue, with varying saturation and lightness. Mixing just black and white pigments creates grays. Figure 13.15 shows the relationship of tints, shades, and tones. The percentage of pigments that must be mixed to match a color can be used as a color specification. The Ostwald [OSTW31] color-order system is similar to the artist's model.

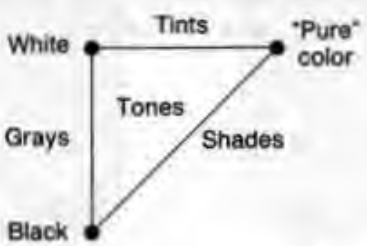


Fig. 13.15 Tints, tones, and shades.

Samsung/Google, EX1027, p. 586

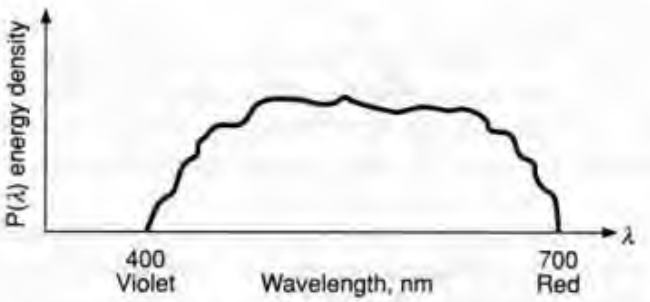


Fig. 13.16 Typical spectral energy distribution $P(\lambda)$ of a light.

13.2.1 Psychophysics

The Munsell and artists' pigment-mixing methods are subjective: They depend on human observers' judgments, the lighting, the size of the sample, the surrounding color, and the overall lightness of the environment. An objective, quantitative way of specifying colors is needed, and for this we turn to the branch of physics known as *colorimetry*. Important terms in colorimetry are dominant wavelength, excitation purity, and luminance.

Dominant wavelength is the wavelength of the color we "see" when viewing the light, and corresponds to the perceptual notion of hue[2]; *excitation purity* corresponds to the saturation of the color; *luminance* is the amount or intensity of light. The excitation purity of a colored light is the proportion of pure light of the dominant wavelength and of white light needed to define the color. A completely pure color is 100 percent saturated and thus contains no white light, whereas mixtures of a pure color and white light have saturations somewhere between 0 and 100 percent. White light and hence grays are 0 percent saturated, containing no color of any dominant wavelength. The correspondences between these perceptual and colorimetry terms are as follows:

Perceptual term	Colorimetry
Hue	Dominant wavelength
Saturation	Excitation purity
Lightness (reflecting objects)	Luminance
Brightness (self-luminous objects)	Luminance

Fundamentally, light is electromagnetic energy in the 400- to 700-nm wavelength part of the spectrum, which is perceived as the colors from violet through indigo, blue, green, yellow, and orange to red. Color Plate I.34 shows the colors of the spectrum. The amount of energy present at each wavelength is represented by a spectral energy distribution $P(\lambda)$, such as shown in Fig. 13.16 and Color Plate I.34. The distribution represents an infinity of numbers, one for each wavelength in the visible spectrum (in practice, the distribution is represented by a large number of sample points on the spectrum, as measured by a

[2]Some colors, such as purple, have no true dominant wavelength, as we shall see later.

Samsung/Google, EX1027, p. 587

spectroradiometer). Fortunately, we can describe the visual effect of any spectral distribution much more concisely by the triple (dominant wavelength, excitation purity, luminance). This implies that many different spectral energy distributions produce the same color: They "look" the same. Hence the relationship between spectral distributions and colors is many-to-one. Two spectral energy distributions that look the same are called *metamers*.

Figure 13.17 shows one of the infinitely many spectral distributions $P(\lambda)$, or metamers, that produces a certain color sensation. At the dominant wavelength, there is a spike of energy of level e_2. White light, the uniform distribution of energy at level e_1, is also present. The excitation purity depends on the relation between e_1 and e_2: when $e_1 = e_2$, excitation purity is 0 percent; when $e_1 = 0$, excitation purity is 100 percent. Brightness, which is proportional to the integral of the product of the curve and the luminous efficiency function (defined later), depends on both e_1 and e_2. In general, spectral distributions may be more complex than the one shown, and it is not possible to determine the dominant wavelength merely by looking at the spectral distributions. In particular, the dominant wavelength may *not* be the one whose component in the spectral distribution is largest.

How does this discussion relate to the red, green, and blue phosphor dots on a color CRT? And how does it relate to the *tristimulus theory* of color perception, which is based on the hypothesis that the retina has three kinds of color sensors (called cones), with peak sensitivity to red, green, or blue lights? Experiments based on this hypothesis produce the spectral-response functions of Fig. 13.18. The peak blue response is around 440 nm; that for green is about 545 nm; that for red is about 580 nm. (The terms "red" and "green" are somewhat misleading here, as the 545 nm and 580 nm peaks are actually in the yellow range.) The curves suggest that the eye's response to blue light is much less strong than is its response to red or green.

Figure 13.19 shows the *luminous-efficiency function*, the eye's response to light of constant luminance, as the dominant wavelength is varied: our peak sensitivity is to yellow-green light of wavelength around 550 nm. There is experimental evidence that this curve is just the sum of the three curves shown in Fig. 13.18.

The tristimulus theory is intuitively attractive because it corresponds loosely to the notion that colors can be specified by positively weighted sums of red, green, and blue (the so-called primary colors). This notion is almost true: The three color-matching functions in

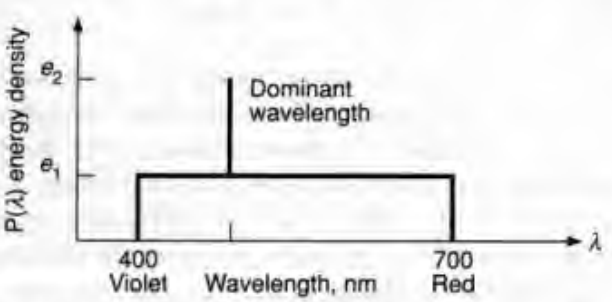


Fig. 13.17 Spectral energy distribution, $P(\lambda)$, illustrating dominant wavelength, excitation purity, and luminance.

Samsung/Google, EX1027, p. 588

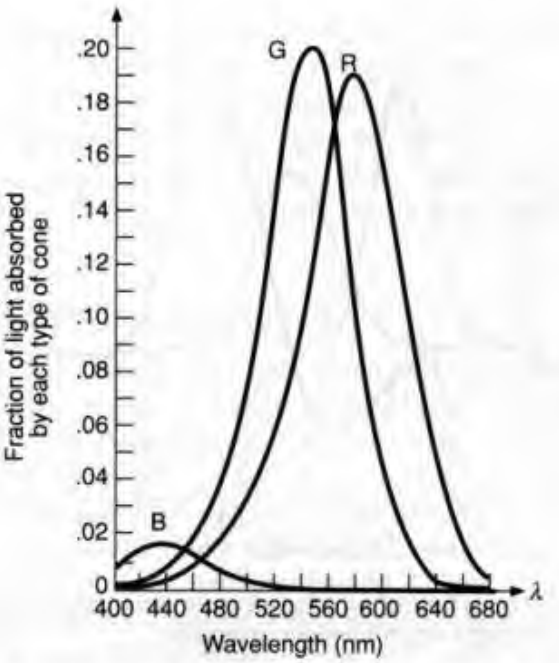


Fig. 13.18 Spectral-response functions of each of the three types of cones on the human retina.

Fig. 13.20 show the amounts of red, green, and blue light needed by an average observer to match a color of constant luminance, for all values of dominant wavelength in the visible spectrum.

A negative value in Fig. 13.20 means that we cannot match the color by adding together the primaries. However, if one of the primaries is added to the color sample, the sample can then be matched by a mixture of the other two primaries. Hence, negative values in Fig. 13.20 indicate that the primary was added to the color being matched. The need for negative values does not mean that the notion of mixing red, green, and blue to obtain other

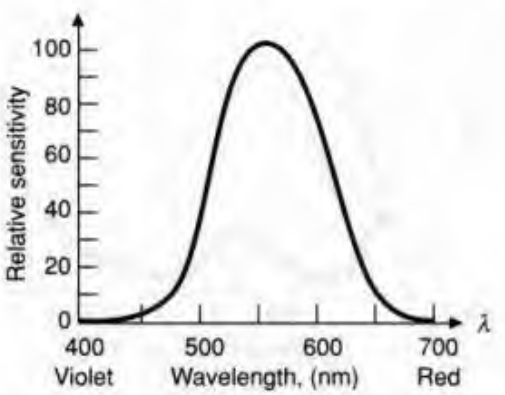


Fig. 13.19 Luminous-efficiency function for the human eye.

Samsung/Google, EX1027, p. 589

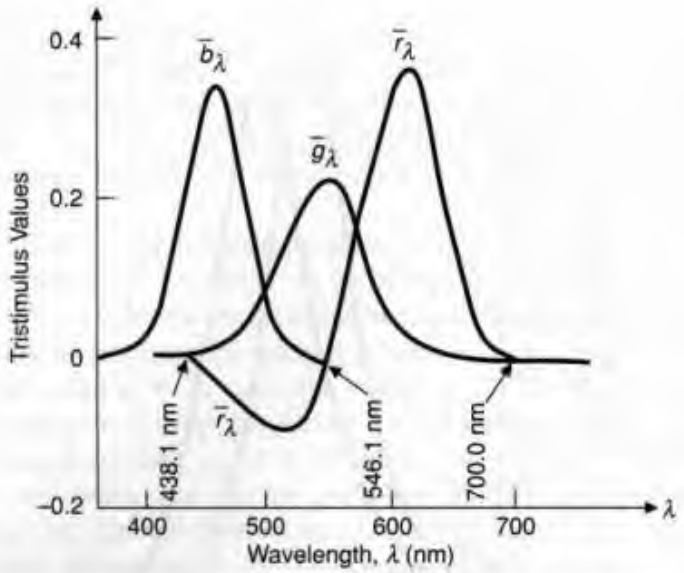


Fig. 13.20 Color-matching functions, showing the amounts of three primaries needed to match all the wavelengths of the visible spectrum.

colors is invalid; on the contrary, a huge range of colors can be matched by positive amounts of red, green, and blue. Otherwise, the color CRT would not work! It does mean, however, that certain colors cannot be produced by RGB mixes, and hence cannot be shown on an ordinary CRT.

The human eye can distinguish hundreds of thousands of different colors in color space, when different colors are judged side by side by different viewers who state whether the colors are the same or different. As shown in Fig. 13.21, when colors differ only in hue, the wavelength between just noticably different colors varies from more than 10 nm at the extremes of the spectrum to less than 2 nm around 480 nm (blue) and 580 nm (yellow).

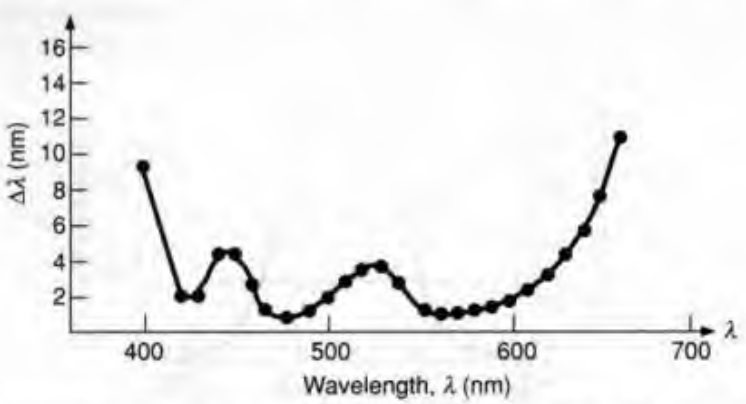


Fig. 13.21 Just-noticable color differences as a function of wavelength λ. The ordinate indicates the minimum detectable difference in wavelength between adjacent color samples. (Source: Data are from [BEDF58].)

Samsung/Google, EX1027, p. 590

Except at the spectrum extremes, however, most distinguished hues are within 4 nm. Altogether about 128 fully saturated hues can be distinguished.

The eye is less sensitive to hue changes in less saturated light. This is not surprising: As saturation tends to 0 percent, all hues tend to white. Sensitivity to changes in saturation for a fixed hue and lightness is greater at the extremes of the visible spectrum, where about 23 distinguishable steps exist. Around 575 nm, only 16 saturation steps can be distinguished [JONE26].

13.2.2 The CIE Chromaticity Diagram

Matching and therefore defining a colored light with a mixture of three fixed primaries is a desirable approach to specifying color, but the need for negative weights suggested by Fig. 13.20 is awkward. In 1931, the *Commission Internationale de l'Éclairage* (*CIE*) defined three standard primaries, called **X**, **Y**, and **Z**, to replace red, green, and blue in this matching process. The three corresponding color-matching functions, $\bar{x}_\lambda$, $\bar{y}_\lambda$, and $\bar{z}_\lambda$, are shown in Fig. 13.22. The primaries can be used to match, with only positive weights, all the colors we can see. The **Y** primary was intentionally defined to have a color-matching function $\bar{y}_\lambda$ that exactly matches the luminous-efficiency function of Fig. 13.19. Note that $\bar{x}_\lambda$, $\bar{y}_\lambda$, and $\bar{z}_\lambda$ are not the spectral distributions of the **X**, **Y**, and **Z** colors, just as the curves in Fig. 13.20 are not the spectral distributions of red, green, and blue. They are merely auxilliary functions used to compute how much of **X**, **Y**, and **Z** should be mixed together to generate a metamer of any visible color.

The color-matching functions are defined tabularly at 1-nm intervals, and are found in texts such as [WYSZ82; BILL81]. The distributions were defined for color samples that

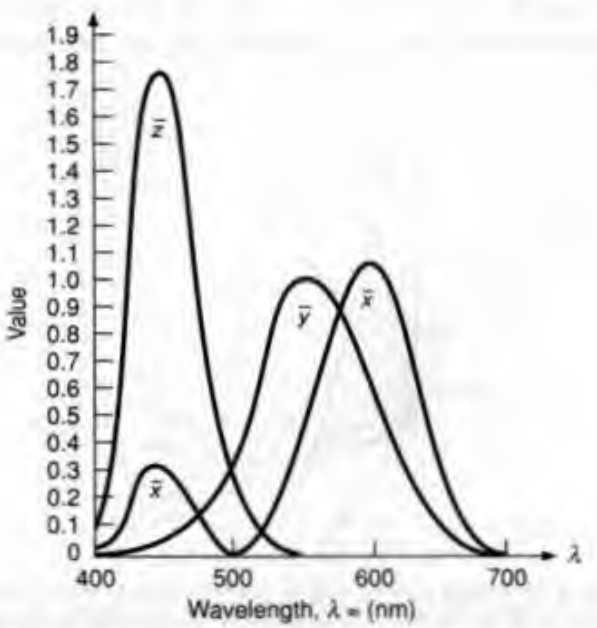


Fig. 13.22 The color-matching functions $\bar{x}_\lambda$, $\bar{y}_\lambda$, and $\bar{z}_\lambda$, for the 1931 CIE **X**, **Y**, **Z** primaries.

Samsung/Google, EX1027, p. 591

subtend a 2° field of view on the retina. The original 1931 tabulation is normally used in work relevant to computer graphics. A later 1964 tabulation for a 10° field of view is not generally useful, because it emphasizes larger areas of constant color than are normally found in graphics.

The three CIE color-matching functions are linear combinations of the color-matching functions from Fig. 13.20. This means that the definition of a color with red, green, and blue lights can be converted via a linear transformation into its definition with the CIE primaries, and vice versa.

The amounts of **X**, **Y**, and **Z** primaries needed to match a color with a spectral energy distribution $P(\lambda)$, are:

$$X = k \int P(\lambda)\, \bar{x}_\lambda\, d\lambda, \quad Y = k \int P(\lambda)\, \bar{y}_\lambda\, d\lambda, \quad Z = k \int P(\lambda)\, \bar{z}_\lambda\, d\lambda. \tag{13.18}$$

For self-luminous objects like a CRT, k is 680 lumens/watt. For reflecting objects, k is usually selected such that bright white has a Y value of 100; then other Y values will be in the range of 0 to 100. Thus,

$$k = \frac{100}{\int P_w(\lambda)\bar{y}_\lambda\, d\lambda} \tag{13.19}$$

where $P_w(\lambda)$ is the spectral energy distribution for whatever light source is chosen as the standard for white. In practice, these integrations are performed by summation, as none of the energy distributions are expressed analytically.

Figure 13.23 shows the cone-shaped volume of XYZ space that contains visible colors. The volume extends out from the origin into the postive octant, and is capped at the smooth curved line terminating the cone.

Let (X, Y, Z) be the weights applied to the CIE primaries to match a color **C**, as found using Eq. (13.18). Then $\mathbf{C} = X\,\mathbf{X} + Y\,\mathbf{Y} + Z\,\mathbf{Z}$. We define *chromaticity* values (which depend only on dominant wavelength and saturation and are independent of the amount of

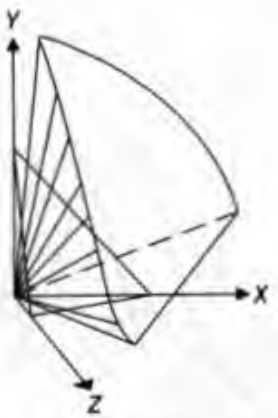


Fig. 13.23 The cone of visible colors in CIE color space is shown by the lines radiating from the origin. The $X + Y + Z = 1$ plane is shown. (Courtesy of Gary Meyer, Program of Computer Graphics, Cornell University, 1978.)

Samsung/Google, EX1027, p. 592

luminous energy) by normalizing against $X + Y + Z$, which can be thought of as the total amount of light energy:

$$x = \frac{X}{(X + Y + Z)}, y = \frac{Y}{(X + Y + Z)}, z = \frac{Z}{(X + Y + Z)}. \tag{13.20}$$

Notice that $x + y + z = 1$. That is, x, y, and z are on the $(X + Y + Z = 1)$ plane of Fig. 13.23. Color plate II.1 shows the $X + Y + Z = 1$ plane as part of CIE space, and also shows an orthographic view of the plane along with the projection of the plane onto the (X, Y) plane. This latter projection is just the CIE chromaticity diagram.

If we specify x and y, then z is determined by $z = 1 - x - y$. We cannot recover X, Y, and Z from x and y, however. To recover them, we need one more piece of information, typically Y, which carries luminance information. Given (x, y, Y), the transformation to the corresponding (X, Y, Z) is

$$X = \frac{x}{y}Y, \quad Y = Y, Z = \frac{1 - x - y}{y}Y. \tag{13.21}$$

Chromaticity values depend only on dominant wavelength and saturation and are independent of the amount of luminous energy. By plotting x and y for all visible colors, we obtain the CIE chromaticity diagram shown in Fig. 13.24, which is the projection onto the (X, Y) plane of the $(X + Y + Z = 1)$ plane of Fig. 13.23. The interior and boundary of the horseshoe-shaped region represent all visible chromaticity values. (All perceivable colors with the same chromaticity but different luminances map into the same point within this region.) The 100 percent spectrally pure colors of the spectrum are on the curved part of the boundary. A standard white light, meant to approximate sunlight, is formally defined by a

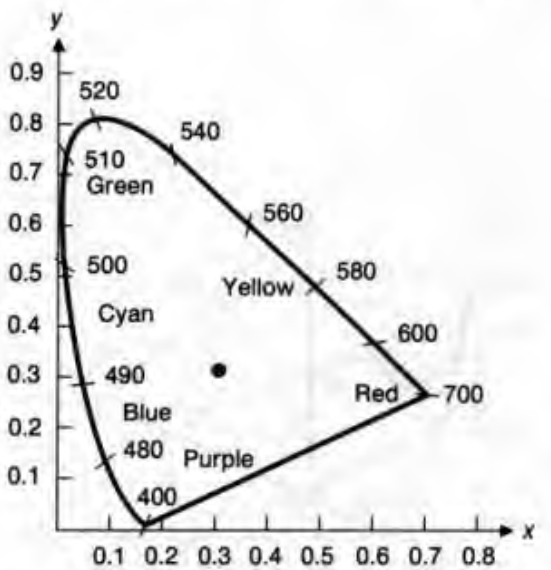


Fig. 13.24 The CIE chromaticity diagram. Wavelengths around the periphery are in nanometers. The dot marks the position of illuminant C.

Samsung/Google, EX1027, p. 593

light source *illuminant C*, marked by the center dot. It is near but not at the point where $x = y = z = \frac{1}{3}$. Illuminant C was defined by specifying a spectral power distribution that is close to daylight at a correlated color temperature of 6774° Kelvin.

The CIE chromaticity diagram is useful in many ways. For one, it allows us to measure the dominant wavelength and excitation purity of any color by matching the color with a mixture of the three CIE primaries. (Instruments called *colorimeters* measure tristimulus X, Y, and Z values, from which chromaticity coordinates are computed using Eq. (13.20). *Spectroradiometers* measure both the spectral energy distribution and the tristimulus values.) Now suppose the matched color is at point A in Fig. 13.25. When two colors are added together, the new color lies somewhere on the straight line in the chromaticity diagram connecting the two colors being added. Therefore, color A can be thought of as a mixture of "standard" white light (illuminant C) and the pure spectral light at point B. Thus, B defines the dominant wavelength. The ratio of length AC to length BC, expressed as a percentage, is the excitation purity of A. The closer A is to C, the more white light A includes and the less pure it is.

The chromaticity diagram factors out luminance, so color sensations that are luminance-related are excluded. For instance, brown, which is an orange-red chromaticity at very low luminance relative to its surrounding area, does not appear. It is thus important to remember that the chromaticity diagram is not a full color palette. There is an infinity of planes in (X, Y, Z) space, each of which projects onto the chromaticity diagram and each of which loses luminance information in the process. The colors found on each such plane are all different.

Complementary colors are those that can be mixed to produce white light (such as D and E in Fig. 13.25). Some colors (such as F in Fig. 13.25) cannot be defined by a

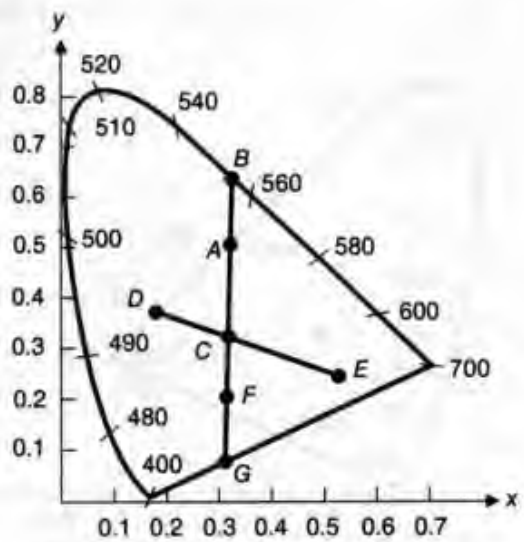


Fig. 13.25 Colors on the chromaticity diagram. The dominant wavelength of color A is that of color B. Colors D and E are complementary colors. The dominant wavelength of color F is defined as the complement of the dominant wavelength of color A.

Samsung/Google, EX1027, p. 594

dominant wavelength and are thus called *nonspectral*. In this case, the dominant wavelength is said to be the complement of the wavelength at which the line through F and C intersects the horseshoe part of the curve at point B, and is designated by a "c" (here about 555 nm c). The excitation purity is still defined by the ratio of lengths (here CF to CG). The colors that must be expressed by using a complementary dominant wavelength are the purples and magentas; they occur in the lower part of the CIE diagram. Complementary colors still can be made to fit the dominant wavelength model of Fig. 13.17, in the sense that if we take a flat spectral distribution and delete some of the light at frequency B, the resulting color will be perceived as F.

Another use of the CIE chromaticity diagram is to define *color gamuts*, or color ranges, that show the effect of adding colors together. Any two colors, say I and J in Fig. 13.26, can be added to produce any color along their connecting line by varying the relative amounts of the two colors being added. A third color K (see Fig. 13.26) can be used with various mixtures of I and J to produce the gamut of all colors in triangle IJK, again by varying relative amounts. The shape of the diagram shows why visible red, green, and blue cannot be additively mixed to match all colors: No triangle whose vertices are within the visible area can completely cover the visible area.

The chromaticity diagram is also used to compare the gamuts available on various color display and hardcopy devices. Color Plate II.2 shows the gamuts for a color television monitor, film, and print. The chromaticity coordinates for the phosphors in two typical monitors are these:

	Short-persistence phosphors			**Long-persistence phosphors**		
	Red	Green	Blue	Red	Green	Blue
x	0.61	0.29	0.15	0.62	0.21	0.15
y	0.35	0.59	0.063	0.33	0.685	0.063

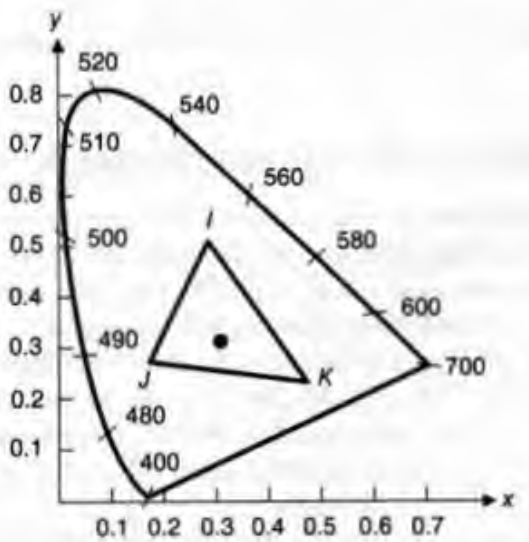


Fig. 13.26 Mixing colors. All colors on the line IJ can be created by mixing colors I and J; all colors in the triangle IJK can be created by mixing colors I, J, and K.

Samsung/Google, EX1027, p. 595

The smallness of the print gamut with respect to the color-monitor gamut suggests that, if images originally seen on a monitor must be faithfully reproduced by printing, a reduced gamut of colors should be used with the monitor. Otherwise, accurate reproduction will not be possible. If, however, the goal is to make a pleasing rather than an exact reproduction, small differences in color gamuts are less important. A discussion of color-gamut compression can be found in [HALL89].

There is a problem with the CIE system. Consider the distance from color $C_1 = (X_1, Y_1, Z_1)$ to color $C_1 + \Delta C$, and the distance from color $C_2 = (X_2, Y_2, Z_2)$ to color $C_2 + \Delta C$, where $\Delta C = (\Delta X, \Delta Y, \Delta Z)$. Both distances are equal to ΔC, yet in general they will not be perceived as being equal. This is because of the variation, throughout the spectrum, of the just noticable differences discussed in Section 13.2.1. A *perceptually uniform* color space is needed, in which two colors that are equally distant are *perceived* as equally distant by viewers.

The 1976 CIE LUV uniform color space was developed in response to this need. With (X_n, Y_n, Z_n) as the coordinates of the color that is to be defined as white, the space is defined by

$$
\begin{aligned}
L^* &= 116\,(Y/Y_n)^{1/3} - 16, \quad Y/Y_n > 0.01 \\
u^* &= 13\,L^*\,(u' - u'_n), \\
v^* &= 13\,L^*\,(v' - v'_n), \\
u' &= \frac{4X}{X + 15Y + 3Z}, \quad v' = \frac{9Y}{X + 15Y + 3Z}, \\
u'_n &= \frac{4X_n}{X_n + 15Y_n + 3Z_n}, \quad v'_n = \frac{9Y_n}{X_n + 15Y_n + 3Z_n}.
\end{aligned}
\tag{13.22}
$$

The shape of the 3D volume of visible colors defined by these equations is of course different from that for CIE (X, Y, Z) space itself (Fig. 13.23).

With this background on color, we now turn our attention to color in computer graphics.

13.3 COLOR MODELS FOR RASTER GRAPHICS

A color model is a specification of a 3D color coordinate system and a visible subset in the coordinate system within which all colors in a particular color gamut lie. For instance, the RGB color model is the unit cube subset of the 3D Cartesian coordinate system.

The purpose of a color model is to allow convenient specification of colors within some color gamut. Our primary interest is the gamut for color CRT monitors, as defined by the RGB (red, green, blue) primaries in Color Plate II.2. As we see in this color plate, a color gamut is a subset of all visible chromaticities. Hence, a color model cannot be used to specify all visible colors. This is emphasized in Fig. 13.27, which shows that the gamut of a CRT monitor is a subset of (X, Y, Z) space.

Three hardware-oriented color models are RGB, used with color CRT monitors, YIQ, the broadcast TV color system, and CMY (cyan, magenta, yellow) for some color-printing devices. Unfortunately, none of these models are particularly easy to use, because they do not relate directly to intuitive color notions of hue, saturation, and brightness. Therefore,

Samsung/Google, EX1027, p. 596

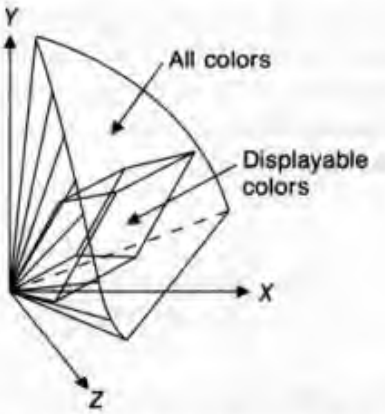


Fig. 13.27 The color gamut for a typical color monitor within the CIE color space. The range of colors that can be displayed on the monitor is clearly smaller than that of all colors visible in CIE space. (Courtesy of Gary Meyer, Program of Computer Graphics, Cornell University, 1978.)

another class of models has been developed with ease of use as a goal. Several such models are described in [GSPC79; JOBL78; MEYE80; SMIT78]. We discuss three, the HSV (sometimes called HSB), HLS, and HVC models.

With each model is given a means of converting to some other specification. For RGB, this conversion is to CIE's (X, Y, Z) space. This conversion is important, because CIE is the worldwide standard. For all of the other models, the conversion is to RGB; hence, we can convert from, say, HSV to RGB to the CIE standard.

13.3.1 The RGB Color Model

The red, green, and blue (RGB) color model used in color CRT monitors and color raster graphics employs a Cartesian coordinate system. The RGB primaries are *additive* primaries; that is, the individual contributions of each primary are added together to yield the result, as suggested in Color Plate II.3. The subset of interest is the unit cube shown in Fig. 13.28. The main diagonal of the cube, with equal amounts of each primary, represents the gray levels: black is (0, 0, 0); white is (1, 1, 1). Color Plates II.4 and II.5 show several views of the RGB color model.

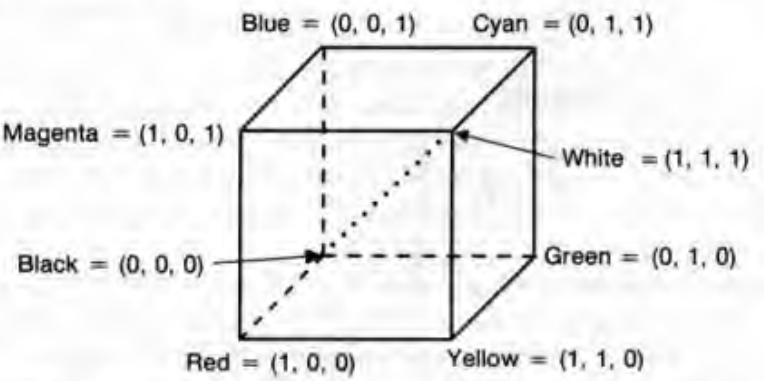


Fig. 13.28 The RGB cube. Grays are on the dotted main diagonal.

Samsung/Google, EX1027, p. 597

The color gamut covered by the RGB model is defined by the chromaticities of a CRT's phosphors. Two CRTs with different phosphors will cover different gamuts. To convert colors specified in the gamut of one CRT to the gamut of another CRT, we can use the transformations M_1 and M_2 from the RGB color space of each monitor to the (X, Y, Z) color space. The form of each transformation is:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}, \tag{13.23}$$

where X_r, X_g, and X_b are the weights applied to the monitor's RGB colors to find X, and so on.

Defining M as the 3×3 matrix of color-matching coefficients, we write Eq. (13.23) as

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = M \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{13.24}$$

With M_1 and M_2 the matricies that convert from each of the two monitor's gamuts to CIE, $M_2^{-1}M_1$ converts from the RGB of monitor 1 to the RGB of monitor 2. This matrix product is all that is needed if the color in question lies in the gamuts of both monitors. What if a color C_1 is in the gamut of monitor 1 but is not in the gamut of monitor 2? The corresponding color $C_2 = M_2^{-1}M_1C_1$ will be outside the unit cube and hence will not be displayable. A simple but not very satisfactory solution is to clamp the color values—that is, to replace values of R, G, or B that are less than 0 with 0, and values that are greater than 1 with 1. More satisfactory but also more complex approaches are described in [HALL89].

The chromaticity coordinates for the RGB phosphors are usually available from CRT manufacturers' specifications. If not, a colorimeter can also be used to measure the chromaticity coordinates directly, or a spectroradiometer can be used to measure $P(\lambda)$, which can then be converted to chromaticity coordinates using Eqs. (13.18), (13.19), and (13.20). Denoting the coordinates by (x_r, y_r) for red, (x_g, y_g) for green, and (x_b, y_b) for blue, and defining C_r as

$$C_r = X_r + Y_r + Z_r, \tag{13.25}$$

we can write, for the red primary;

$$\begin{aligned} x_r &= \frac{X_r}{X_r + Y_r + Z_r} = \frac{X_r}{C_r}, \; X_r = x_r C_r, \\ y_r &= \frac{Y_r}{X_r + Y_r + Z_r} = \frac{Y_r}{C_r}, \; Y_r = y_r C_r, \\ z_r &= (1 - x_r - y_r) = \frac{Z_r}{X_r + Y_r + Z_r} = \frac{Z_r}{C_r}, \; Z_r = z_r C_r. \end{aligned} \tag{13.26}$$

Samsung/Google, EX1027, p. 598

With similar definitions for C_g and C_b, Eq. (13.23) can be rewritten as:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} x_r C_r & x_g C_g & x_b C_b \\ y_r C_r & y_g C_g & y_b C_b \\ (1 - x_r - y_r)C_r & (1 - x_g - y_g)C_g & (1 - x_b - y_b)C_b \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \quad (13.27)$$

The unknowns C_r, C_g, and C_b can be found in one of two ways [MEYE83]. First, the luminances Y_r, Y_g, and Y_b of maximum-brightness red, green, and blue may be known or can be measured with a high-quality photometer (inexpensive meters can be off by factors of 2 to 10 on the blue reading). These measured luminances can be combined with the known y_r, y_g, and y_b to yield

$$C_r = Y_r/y_r,\ C_g = Y_g/y_g,\ C_b = Y_b/y_b. \quad (13.28)$$

These values are then substituted into Eq. (13.27), and the conversion matrix M is thus expressed in terms of the known quantities (x_r, y_r), (x_g, y_g), (x_b, y_b), Y_r, Y_g, and Y_b.

We can also remove the unknown variables from Eq. (13.27) if we know or can measure the X_w, Y_w, and Z_w for the white color produced when $R = G = B = 1$. For this case, Eq. (13.27) can be rewritten as

$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} = \begin{bmatrix} x_r & x_g & x_b \\ y_r & y_g & y_b \\ (1 - x_r - y_r) & (1 - x_g - y_g) & (1 - x_b - y_b) \end{bmatrix} \begin{bmatrix} C_r \\ C_g \\ C_b \end{bmatrix}, \quad (13.29)$$

solved for C_r, C_g, and C_b (the only unknowns), and the resulting values substituted into Eq. (13.27). Alternatively, it may be that the white color is given by x_w, y_w, and Y_w; in this case, before solving Eq. (13.29), we first find

$$X_w = x_w \frac{Y_w}{y_w},\ Z_w = z_w \frac{Y_w}{y_w}. \quad (13.30)$$

13.3.2 The CMY Color Model

Cyan, magenta, and yellow are the complements of red, green, and blue, respectively. When used as filters to subtract color from white light, they are called *subtractive primaries*. The subset of the Cartesian coordinate system for the CMY model is the same as that for RGB except that white (full light) instead of black (no light) is at the origin. Colors are specified by what is removed or subtracted from white light, rather than by what is added to blackness.

A knowledge of CMY is important when dealing with hardcopy devices that deposit colored pigments onto paper, such as electrostatic and ink-jet plotters. When a surface is coated with cyan ink, no red light is reflected from the surface. Cyan subtracts red from the reflected white light, which is itself the sum of red, green, and blue. Hence, in terms of the

Samsung/Google, EX1027, p. 599

additive primaries, cyan is white minus red, that is, blue plus green. Similarly, magenta absorbs green, so it is red plus blue; yellow absorbs blue, so it is red plus green. A surface coated with cyan and yellow absorbs red and blue, leaving only green to be reflected from illuminating white light. A cyan, yellow, and magenta surface absorbs red, green, and blue, and therefore is black. These relations, diagrammed in Fig. 13.29, can be seen in Color Plate II.6 and are represented by the following equation:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{13.31}$$

The unit column vector is the RGB representation for white and the CMY representation for black.

The conversion from RGB to CMY is then

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}. \tag{13.32}$$

These straightforward transformations can be used for converting the eight colors that can be achieved with binary combinations of red, green, and blue into the eight colors achievable with binary combinations of cyan, magenta, and yellow. This conversion is relevant for use on ink-jet and xerographic color printers.

Another color model, CMYK, uses black (abbreviated as K) as a fourth color. CMYK is used in the four-color printing process of printing presses and some hard-copy devices.

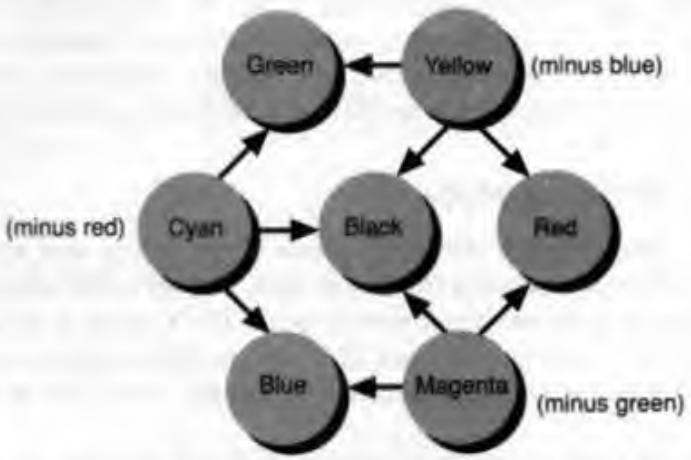


Fig. 13.29 Subtractive primaries (cyan, magenta, yellow) and their mixtures. For instance, cyan and yellow combine to green.

Samsung/Google, EX1027, p. 600

Given a CMY specification, black is used in place of equal amounts of C, M, and Y, according to the relations:

$$\begin{aligned} K &= \min(C, M, Y); \\ C &= C - K; \\ M &= M - K; \\ Y &= Y - K; \end{aligned} \tag{13.34}$$

This is discussed further in Section 13.4 and in [STON88].

13.3.3 The YIQ Color Model

The YIQ model is used in U.S. commercial color television broadcasting and is therefore closely related to color raster graphics. YIQ is a recoding of RGB for transmission efficiency and for downward compatibility with black-and-white television. The recoded signal is transmitted using the National Television System Committee (NTSC) [PRIT77] system.

The Y component of YIQ is not yellow but luminance, and is defined to be the same as the CIE Y primary. Only the Y component of a color TV signal is shown on black-and-white televisions: the chromaticity is encoded in I and Q. The YIQ model uses a 3D Cartesian coordinate system, with the visible subset being a convex polyhedron that maps into the RGB cube.

The RGB-to-YIQ mapping is defined as follows:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{13.33}$$

The quantities in the first row reflect the relative importance of green and red and the relative unimportance of blue in brightness. The inverse of the RGB-to-YIQ matrix is used for the YIQ-to-RGB conversion.

Equation (13.33) assumes that the RGB color specification is based on the standard NTSC RGB phosphor, whose CIE coordinates are

	Red	Green	Blue
x	0.67	0.21	0.14
y	0.33	0.71	0.08

and for which the white point, illuminant C, is $x_w = 0.31$, $y_w = 0.316$, and $Y_w = 100.0$.

Specifying colors with the YIQ model solves a potential problem with material being prepared for broadcast television: Two different colors shown side by side on a color monitor will appear to be different, but, when converted to YIQ and viewed on a monochrome monitor, they may appear to be the same. This problem can be avoided by

Samsung/Google, EX1027, p. 601

specifying the two colors with different Y values in the YIQ color model space (i.e., by adjusting only the Y value to disambiguate them).

The YIQ model exploits two useful properties of our visual system. First, the system is more sensitive to changes in luminance than to changes in hue or saturation; that is, our ability to discriminate spatially color information is weaker than our ability to discriminate spatially monochrome information. This suggests that more bits of bandwidth should be used to represent Y than are used to represent I and Q, so as to provide higher resolution in Y. Second, objects that cover a very small part of our field of view produce a limited color sensation, which can be specified adequately with one rather than two color dimensions. This suggests that either I or Q can have a lower bandwidth than the other. The NTSC encoding of YIQ into a broadcast signal uses these properties to maximize the amount of information transmitted in a fixed bandwidth: 4 MHz is assigned to Y, 1.5 to I, and 0.6 to Q. Further discussion of YIQ can be found in [SMIT78; PRIT77].

13.3.4 The HSV Color Model

The RGB, CMY, and YIQ models are hardware-oriented. By contrast, Smith's HSV (hue, saturation, value) model [SMIT78] (also called the HSB model, with B for brightness) is user-oriented, being based on the intuitive appeal of the artist's tint, shade, and tone. The coordinate system is cylindrical, and the subset of the space within which the model is defined is a hexcone, or six-sided pyramid, as in Fig. 13.30. The top of the hexcone corresponds to $V = 1$, which contains the relatively bright colors. The colors of the $V = 1$ plane are *not* all of the same perceived brightness, however. Color Plates II.7 and II.8 show the color model.

Hue, or H, is measured by the angle around the vertical axis, with red at 0°, green at 120°, and so on (see Fig. 13.30). Complementary colors in the HSV hexcone are 180°

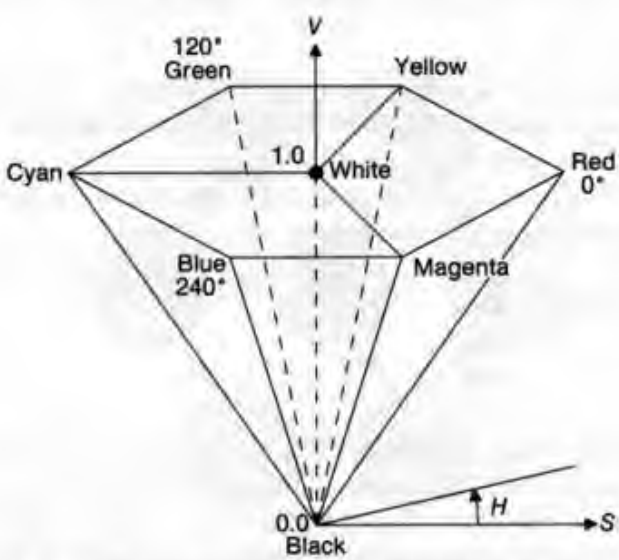


Fig. 13.30 Single-hexcone HSV color model. The $V = 1$ plane contains the RGB model's $R = 1$, $G = 1$, and $B = 1$ planes in the regions shown.

Samsung/Google, EX1027, p. 602

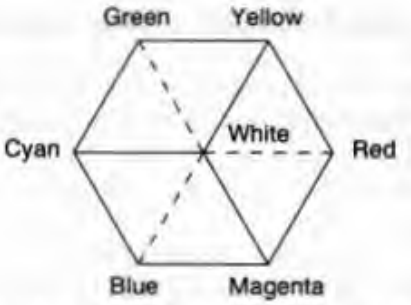


Fig. 13.31 RGB color cube viewed along the principal diagonal. Visible edges of the cube are solid; invisible edges are dashed.

opposite one another. The value of S is a ratio ranging from 0 on the center line (V axis) to 1 on the triangular sides of the hexcone. Saturation is measured relative to the color gamut represented by the model, which is, of course, a subset of the entire CIE chromaticity diagram. Therefore, saturation of 100 percent in the model is less than 100 percent excitation purity.

The hexcone is one unit high in V, with the apex at the origin. The point at the apex is black and has a V coordinate of 0. At this point, the values of H and S are irrelevant. The point $S = 0$, $V = 1$ is white. Intermediate values of V for $S = 0$ (on the center line) are the grays. When $S = 0$, the value of H is irrelevant (called by convention UNDEFINED). When S is not zero, H is relevant. For example, pure red is at $H = 0$, $S = 1$, $V = 1$. Indeed, any color with $V = 1$, $S = 1$ is akin to an artist's pure pigment used as the starting point in mixing colors. Adding white pigment corresponds to decreasing S (without changing V). Shades are created by keeping $S = 1$ and decreasing V. Tones are created by decreasing both S and V. Of course, changing H corresponds to selecting the pure pigment with which to start. Thus, H, S, and V correspond to concepts from the artists' color system, and are not exactly the same as the similar terms introduced in Section 13.2.

The top of the HSV hexcone corresponds to the projection seen by looking along the principal diagonal of the RGB color cube from white toward black, as shown in Fig. 13.31. The RGB cube has subcubes, as illustrated in Fig. 13.32. Each subcube, when viewed along

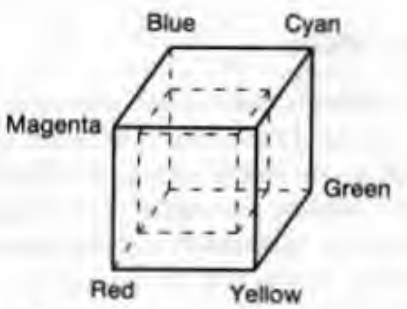


Fig. 13.32 RGB cube and a subcube.

Samsung/Google, EX1027, p. 603

```
void RGB_To_HSV (double r, double g, double b, double *h, double *s, double *v)
/* Given: r, g, b, each in [0,1]. */
/* Desired: h in [0,360), s and v in [0,1] except if s = 0, then h = UNDEFINED. */
/* which is some constant defined with a value outside the interval [0,360]. */
{
    double max = Maximum (r, g, b);
    double min = Minimum (r, g, b);
    *v = max;                                   /* This is the value v. */
    /* Next calculate saturation, s. Saturation is 0 if red, green and blue are all 0 */
    *s = (max != 0.0) ? ((max - min) / max) : 0.0;
    if (*s == 0.0)
        *h = UNDEFINED;
    else {                                      /* Chromatic case: Saturation is not 0, */
        double delta = max - min;               /* so determine hue. */
        if (r == max)
            *h = (g - b) / delta;               /* Resulting color is between yellow and magenta */
        else if (g == max)
            *h = 2.0 + (b - r) / delta;         /* Resulting color is between cyan and yellow */
        else if (b == max)
            *h = 4.0 + (r - g) / delta;         /* Resulting color is between magenta and cyan */
        *h *= 60.0;                             /* Convert hue to degrees */
        if (*h < 0.0)
            *h += 360.0;                        /* Make sure hue is nonnegative */
    } /* Chromatic case */
} /* RGB_To_HSV */
```

Fig. 13.33 Algorithm for converting from RGB to HSV color space.

its main diagonal, is like the hexagon in Fig. 13.31, except smaller. Each plane of constant V in HSV space corresponds to such a view of a subcube in RGB space. The main diagonal of RGB space becomes the V axis of HSV space. Thus, we can see intuitively the correspondence between RGB and HSV. The algorithms of Figs. 13.33 and 13.34 define the correspondence precisely by providing conversions from one model to the other.

13.3.5 The HLS Color Model

The HLS (hue, lightness, saturation) color model is defined in the double-hexcone subset of a cylindrical space, as seen in Fig. 13.35. Hue is the angle around the vertical axis of the double hexcone, with red at 0° (some discussions of HLS have blue at 0°; we place red at 0° for consistency with the HSV model). The colors occur around the perimeter in the same order as in the CIE diagram when its boundary is traversed counterclockwise: red, yellow, green, cyan, blue, and magenta. This is also the same order as in the HSV single-hexcone model. In fact, we can think of HLS as a deformation of HSV, in which white is pulled

Samsung/Google, EX1027, p. 604

```
void HSV_To_RGB (double *r, double *g, double *b, double h, double s, double v)
/* Given: h in [0,360] or UNDEFINED, s and v in [0,1]. */
/* Desired: r, g, b, each in [0,1]. */
{
    if (s == 0.0) {          /* The color is on the black-and-white center line. */
        /* Achromatic color: There is no hue. */
        if (h == UNDEFINED) {
            *r = v;          /* This is the achromatic case. */
            *g = v;
            *b = v;
        } else
            Error();         /* By our convention, error if s = 0 and h has a value. */
    } else {
        double f,p,q,t;      /* Chromatic color: s != 0, so there is a hue. */
        int i;

        if (h == 360.0)      /* 360 degrees is equivalent to 0 degrees. */
            h = 0.0;
        h /= 60.0;           /* h is now in [0,6). */
        i = floor (h);       /* Floor returns the largest integer <= h */
        f = h - i;           /* f is the fractional part of h. */
        p = v * (1.0 - s);
        q = v * (1.0 - (s * f));
        t = v * (1.0 - (s * (1.0 - f)));
        switch(i) {
            case 0: *r = v; *g = t; *b = p; break;
            case 1: *r = q; *g = v; *b = p; break;
            case 2: *r = p; *g = v; *b = t; break;
            case 3: *r = p; *g = q; *b = v; break;
            case 4: *r = t; *g = p; *b = v; break;
            case 5: *r = v; *g = p; *b = q; break;
        }
    }  /* Chromatic case */
}  /* HSV_To_RGB */
```

Fig. 13.34 Algorithm for converting from HSV to RGB color space.

upward to form the upper hexcone from the $V = 1$ plane. As with the single-hexcone model, the complement of any hue is located 180° farther around the double hexcone, and saturation is measured radially from the vertical axis, from 0 on the axis to 1 on the surface. Lightness is 0 for black (at the lower tip of the double hexcone) to 1 for white (at the upper tip). Color Plate II.9 shows a view of the HLS model. Again, the terms hue, lightness, and saturation in this model are similar to, but are not exactly identical to, the same terms as they were introduced in an earlier section. Color Plate II.10 shows a different view of the space.

The procedures of Figs. 13.36 and 13.37 perform the conversions between HLS and RGB. They are modified from those given by Metrick [GSPC79] to leave H UNDEFINED when $S = 0$, and to have $H = 0$ for red rather than for blue.

Samsung/Google, EX1027, p. 605

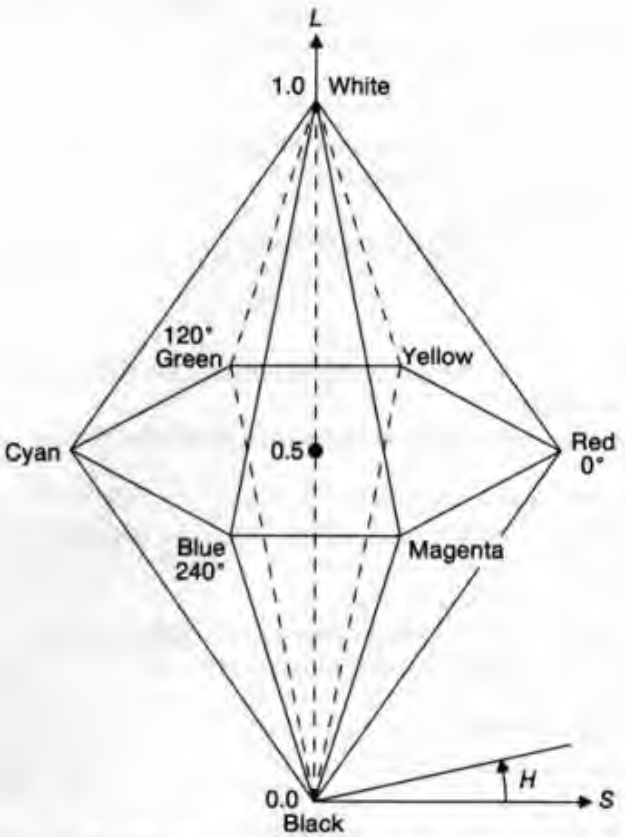


Fig. 13.35 Double-hexcone HLS color model.

The HLS model, like the HSV model, is easy to use. The grays all have $S = 0$, but the maximally saturated hues are at $S = 1, L = 0.5$. If potentiometers are used to specify the color-model parameters, the fact that L must be 0.5 to get the strongest possible colors is a disadvantage over the HSV model, in which $S = 1$ and $V = 1$ achieve the same effect. However, analogously to HSV, the colors of the $L = 0.5$ plane are *not* all of the same perceived brightness. Hence two different colors of equal perceived brightness will generally have different values of L. Furthermore, neither HLS nor any of the other models discussed thus far in this section are perceptually uniform, in the sense discussed in Section 13.2.

The recently developed Tektronix TekHVC (hue, value, chroma) color system, a modification of the CIE LUV perceptually uniform color space discussed in Section 13.2, does provide a color space in which measured and perceived distances between colors are approximately equal. This is an important advantage of both the CIE LUV and TekHVC models. Figure 13.38 shows one view of the HVC color model, and Color Plate II.11 shows another view. Details of the transformations from CIE to TekHVC have not been released. However, we see from Eq. (13.22) that the transformation from CIE XYZ to CIE LUV is computationally straightforward, with the cube root being the most computationally intense element. Thus, we expect that perceptually uniform color spaces will come to be more widely used in the future.

Samsung/Google, EX1027, p. 606

```
void RGB_To_HLS (double r, double g, double b, double *h, double *l, double *s)
/* Given: r, g, b each in [0,1]. */
/* Desired: h in [0,360), l and s in [0,1], except if s = 0, then h = UNDEFINED. */
{
    double max = Maximum (r, g, b);
    double min = Minimum (r, g, b);
    *l = (max + min) / 2.0;                    /* This is the lightness. */
    /* Next calculate saturation */
    if (max == min) {                          /* Achromatic case, because r = g = b */
        *s = 0;
        *h = UNDEFINED;
    } else {                                   /* Chromatic case */
        double delta = max - min;

        /* First calculate the saturation. */
        *s = (*l <= 0.5) ? (delta / (max + min)) : (delta / (2.0 - (max + min)));
        /* Next calculate the hue. */
        if (r == max)
            *h = (g - b) / delta;              /* Resulting color is between yellow and magenta */
        else if (g == max)
            *h = 2.0 + (b - r) / delta;        /* Resulting color is between cyan and yellow */
        else if (b == max)
            *h = 4.0 + (r - g) / delta;        /* Resulting color is between magenta and cyan */
        *h *= 60.0;                            /* Convert to degrees */
        if (h < 0.0)
            *h += 360.0;                       /* Make degrees be nonnegative */
    }  /* Chromatic case */
}  /* RGB_To_HLS */
```

Fig. 13.36 Algorithm for converting from RGB to HLS color space.

13.3.6 Interactive Specification of Color

Many application programs allow the user to specify colors of areas, lines, text, and so on. If only a small set of colors is provided, menu selection from samples of the available colors is appropriate. But what if the set of colors is larger than can reasonably be displayed in a menu?

The basic choices are to use English-language names, to specify the numeric coordinates of the color in a color space (either by typing or with slider dials), or to interact directly with a visual representation of the color space. Naming is in general unsatisfactory because it is ambiguous and subjective ("a light navy blue with a touch of green"), and it is also the antithesis of graphic interaction. On the other hand, [BERK82] describes CNS, a fairly well-defined color-naming scheme that uses terms such as "greenish-yellow," "green-yellow," and "yellowish-green" to distinguish three hues between green and

Samsung/Google, EX1027, p. 607

```
void HLS_To_RGB (double *r, double *g, double *b, double h, double l, double s)
/* Given: h in [0,360] or UNDEFINED, l and s in [0,1]. */
/* Desired: r, g, b each in [0,1] */
{
    double m1, m2;

    m2 = (l <= 0.5) ? (l * (l + s)) : (l + s - l * s);
    m1 = 2.0 * l - m2;
    if (s == 0.0) {                  /* Achromatic: There is no hue. */
        if (h == UNDEFINED)
            *r = *g = *b = l;        /* This is the achromatic case. */
        else Error ();               /* Error if s = 0 and h has a value */
    } else {      /* Chromatic case, so there is a hue */
        *r = Value (m1, m2, h + 120.0);
        *g = Value (m1, m2, h);
        *b = Value (m1, m2, h - 120.0);
    }
} /* HLS_To_RGB */

static double Value (double n1, double n2, double hue)
{
    if (hue > 360.0)
        hue -= 360.0;
    else if (hue < 0.0)
        hue += 360.0;
    if (hue < 60.0)
        return n1 + (n2 - n1) * hue / 60.0;
    else if (hue < 180.0)
        return n2;
    else if (hue < 240.0)
        return n1 + (n2 - n1) * (240.0 - hue) / 60.0;
    else
        return n1;
} /* Value */
```

Fig. 13.37 Algorithm for converting from HLS to RGB color space.

yellow. In an experiment, users of CNS were able to specify colors more precisely than were users who entered numeric coordinates in either RGB or HSV space.

Coordinate specification can be done with slider dials, as in Fig. 13.39, using any of the color models. If the user understands how each dimension affects the color, this technique works well. Probably the best interactive specification method is to let the user interact directly with a representation of the color space, as shown in Fig. 13.40. The line on the circle (representing the $V = 1$ plane) can be dragged around to determine which slice of the HSV volume is displayed in the triangle. The cursor on the triangle can be moved around to specify saturation and value. As the line or the cursor is moved, the numeric readouts change value. When the user types new values directly into the numeric readouts, the line and cursor are repositioned. The color sample box shows the currently selected

Samsung/Google, EX1027, p. 608

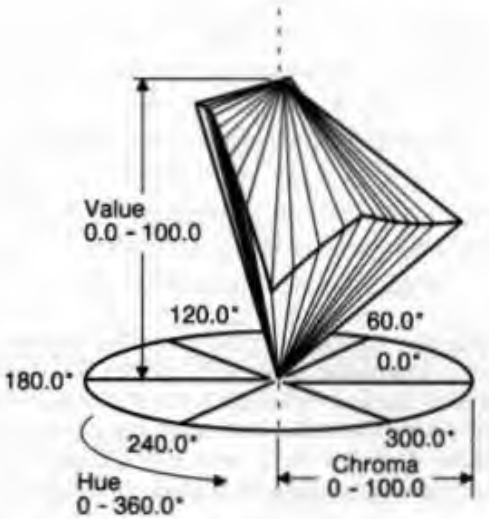


Fig. 13.38 The TekHVC color model. (Courtesy of Tektronix, Inc.)

color. Color Plate II.12 shows a similar method used for the HSV model.

[SCHW87] describes color-matching experiments in which subjects used a data tablet to specify colors in several models including RGB, YIQ, LAB [WYSZ82], and HSV. HSV was found to be slow but accurate, whereas RGB was faster but less accurate. There is a widespread belief that the HSV model is especially tractable, usually making it the model of choice.

Many interactive systems that permit user specification of color show the user the current color settings as a series of adjacent patches of color, as in part (a) of Fig. 13.39, or as the color of the single pixel value currently being set, as in part (b). As the user

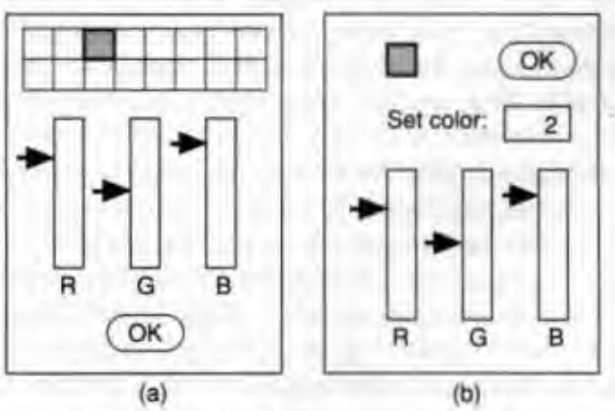


Fig. 13.39 Two common ways of setting colors. In (a), the user selects one of 16 colors to set; the selected color is designated with the thick border. The RGB slider dials control the color; OK is selected to dismiss the color control panel. In (b), the number of the color to be set is typed, and the current color is displayed in the gray-toned box. In both cases, the user must simultaneously see the actual display in order to understand the effects of the color change.

Samsung/Google, EX1027, p. 609

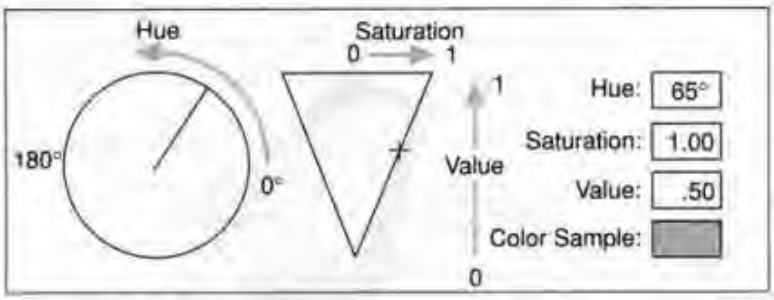


Fig. 13.40 A convenient way to specify colors in HSV space. Saturation and value are shown by the cursor in the triangular area, and hue by the line in the circular area. The user can move the line and cursor indicators on the diagrams, causing the numeric readouts to be updated. Alternatively, the user can type new values, causing the indicators to change. Slider dials for *H*, *S*, and *V* could also be added, giving the user accurate control over a single dimension at a time, without the need to type values.

manipulates the slider dials, the color sample changes. However, a person's perception of color is affected by surrounding colors and the sizes of colored areas, as shown in Color Plate II.13; hence, the color as perceived in the feedback area will probably differ from the color as perceived in the actual display. It is thus important that the user also see the actual display while the colors are being set.

13.3.7 Interpolating in Color Space

Color interpolation is necessary in at least three situations: for Gouraud shading (Section 16.2.4), for antialiasing (Section 3.17), and in blending two images together as for a fade-in, fade-out sequence. The results of the interpolation depend on the color model in which the colors are interpolated; thus, care must be taken to select an appropriate model.

If the conversion from one color model to another transforms a straight line (representing the interpolation path) in one color model into a straight line in the other color model, then the results of linear interpolation in both models will be the same. This is the case for the RGB, CMY, YIQ, and CIE color models, all of which are related by simple affine transformations. However, a straight line in the RGB model does *not* in general transform into a straight line in either the HSV or HLS models. Color Plate II.14 shows the results of linearly interpolating between the same two colors in the HSV, HSL, RGB, and YIQ color spaces. Consider the interpolation between red and green. In RGB, red = (1, 0, 0) and green = (0, 1, 0). Their interpolation (with both weights equal to 0.5 for convenience) is (0.5, 0.5, 0). Applying algorithm RGB_To_HSV (Fig. 13.33) to this result, we have (60°, 1, 0.5). Now, representing red and green in HSV, we have (0°, 1, 1) and (120°, 1, 1). But interpolating with equal weights in HSV, we have (60°, 1, 1); thus, the value differs by 0.5 from the same interpolation in RGB.

As a second example, consider interpolating red and cyan in the RGB and HSV models. In RGB, we start with (1, 0, 0) and (0, 1, 1), respectively, and interpolate to (0.5, 0.5, 0.5), which in HSV is represented as (UNDEFINED, 0, 0.5). In HSV, red and cyan are (0°, 1, 1) and (180°, 1, 1). Interpolating, we have (90°, 1, 1); a new hue at maximum value and saturation has been introduced, whereas the "right" result of combining equal amounts

Samsung/Google, EX1027, p. 610

of complementary colors is a gray value. Here, again, interpolating and then transforming gives different results from transforming and then interpolating.

For Gouraud shading, any of the models can be used, because the two interpolants are generally so close together that the interpolation paths between the colors are close together as well. When two images are blended—as in a fade-in, fade-out sequence or for antialiasing—the colors may be quite distant, and an additive model, such as RGB, is appropriate. If, on the other hand, the objective is to interpolate between two colors of fixed hue (or saturation) and to maintain the fixed hue (or saturation) for all interpolated colors, then HSV or HLS is preferable. But note that a fixed-saturation interpolation in HSV or HLS is *not* seen as having exactly fixed saturation by the viewer [WARE87].

13.4 REPRODUCING COLOR

Color images are reproduced in print in a way similar to that used for monochrome images, but four sets of halftone dots are printed, one for each of the subtractive primaries, and another for black. In a process called *undercolor removal*, black replaces equal amounts of cyan, magenta, and yellow. This creates a darker black than is possible by mixing the three primaries, and hastens drying by decreasing the amounts of cyan, magenta, and yellow ink needed. The orientation of each of the grids of dots is different, so that interference patterns are not created. Color Plate II.15 shows an enlarged halftone color pattern. Our eyes spatially integrate the light reflected from adjacent dots, so that we see the color defined by the proportions of primaries in adjacent dots. This spatial integration of different colors is the same phenomenon we experience when viewing the triads of red, green, and blue dots on a color monitor.

We infer, then, that color reproduction in print and on CRTs depends on the same spatial integration used in monochrome reproduction. The monochrome dithering techniques discussed in Section 13.1.2 can also be used with color to extend the number of available colors, again at the expense of resolution. Consider a color display with 3 bits per pixel, one each for red, green, and blue. We can use a 2×2 pixel pattern area to obtain 125 different colors: each pattern can display five intensities for each of red, green, and blue, by using the halftone patterns in Fig. 13.8. This results in $5 \times 5 \times 5 = 125$ color combinations.

Not all color reproduction depends exclusively on spatial integration. For instance, xerographic color copiers, ink-jet plotters, and thermal color printers actually mix subtractive pigments on the paper's surface to obtain a small set of different colors. In xerography, the colored pigments are first deposited in three successive steps, then are heated and melted together. The inks sprayed by the plotter mix before drying. Spatial integration may be used to expand the color range further.

A related quantization problem occurs when a color image with n bits per pixel is to be displayed on a display of $m < n$ bits per pixel with no loss of spatial resolution. Here, color resolution *must* be sacrificed. In this situation, there are two key questions: Which 2^m colors should be displayed? What is the mapping from the set of 2^n colors in the image to the smaller set of 2^m colors being displayed?

The simple answers are to use a predefined set of display colors and a fixed mapping from image colors to display colors. For instance, with $m = 8$, a typical assignment is 3 bits

Samsung/Google, EX1027, p. 611

to red and green and 2 to blue (because of the eye's lower sensitivity to blue). Thus, the 256 displayable colors are all the combinations of eight reds, eight greens, and four blues. The specific values for the red, green, and blue colors would be spaced across the range of 0.0 to 1.0 range on a ratio scale, as discussed in Section 13.1.1. For an image with 6 bits per color and hence 64 levels for each color, the 64 red colors are mapped into one of the eight displayable reds, and similarly for the greens. The 64 blue colors are mapped into just four blues.

With this solution, however, if all the blue image colors are clustered together in color space, they might all be displayed as the same blue, whereas the other three displayable blues might go unused. An adaptive scheme would take this possibility into account and would divide up the blue-value range on the basis of the distribution of values in the range. Heckbert [HECK82] describes two approaches of this type: the popularity algorithm and the median-cut algorithm.

The *popularity algorithm* creates a histogram of the image's colors, and uses the 2^m most frequent colors in the mapping. The *median-cut algorithm* recursively fits a box around the colors used in the image, splitting the box along its longer dimension at the median in that dimension. The recursion ends when 2^m boxes have been created; the centroid of each box is used as the display color for all image colors in the box. The median-cut algorithm is slower than is the popularity algorithm, but produces better results. Another way of splitting the boxes is described in [WAN88].

Creating an *accurate* color reproduction is much more difficult than is approximating colors. Two display monitors can be calibrated to create the same tristimulus values; the multistep process is described in detail in [COWA83; MEYE83]. The steps are to measure the chromaticity coordinates of the monitor phosphors, then to adjust the brightness and contrast controls of each of the monitor guns so that the same white chromaticity is produced whenever $R = G = B$, and to determine the appropriate gamma correction for each gun.

Making slides or movie film that look exactly like the image on a display is difficult, because many variables are involved. They include the gamma correction of the display and of the CRT used in the film recorder; the color of light emitted by the CRT in the film recorder; the filters used in the film recorder; the type of film used; the quality and temperature of the developing chemicals, the length of time the film is in the chemicals, and the color of light emitted by the bulb in the slide or film projector. Fortunately, all these variables can be quantified and controlled, albeit with considerable difficulty.

Controlling the color match on printed materials is also difficult; the printing process, with its cyan, magenta, yellow, and black primaries, requires careful quality control to maintain registration and ink flow. The paper texture, absorbancy, and gloss also affect the result. Complicating matters further, the simple subtractive CMY color model discussed in Section 13.3.2 cannot be used directly, because it does not take into account these complications of the printing process. More detail can be found in [STON88].

Even if extreme care is taken in color reproduction, the results may not seem to match the original. Lighting conditions and reflections from the display can cause colors with the same measured chromaticity coordinates to appear to be different. Fortunately, the purpose of the reproduction is usually (although not always) to maintain color relationships between different parts of the image, rather than to make an exact copy.

Samsung/Google, EX1027, p. 612

13.5 USING COLOR IN COMPUTER GRAPHICS

We use color for aesthetics, to establish a tone or mood, for realism, as a highlight, to identify associated areas as being associated, and for coding. With care, color can be used effectively for these purposes. In addition, users tend to like color, even when there is no quantitative evidence that it helps their performance. Although cost-conscious buyers may scoff at color monitors, we believe that anything that encourages people to use computers is important!

Careless use of color can make the display less useful or less attractive than a corresponding monochrome presentation. In one experiment, introduction of meaningless color reduced user performance to about one-third of what it was without color [KREB79]. Color should be employed conservatively. Any decorative use of color should be subservient to the functional use, so that the color cannot be misinterpreted as having some underlying meaning. Thus, the use of color, like all other aspects of a user–computer interface, must be tested with real users to identify and remedy problems. Of course, some individuals may have other preferences, so it is common practice to provide defaults chosen on the basis of color usage rules, with some means for the user to change the defaults. A conservative approach to color selection is to design first for a monochrome display, to ensure that color use is purely redundant. This avoids creating problems for color-deficient users and also means that the application can be used on a monochrome display. Additional information on color deficiencies is given in [MEYE88]. The color choices used in the window managers shown in Color Plates I.26 through I.29 are quite conservative. Color is not used as a unique code for button status, selected menu item, and so forth.

Many books have been written on the use of color for aesthetic purposes, including [BIRR61]; we state here just a few of the simpler rules that help to produce color harmony. The most fundamental rule of color aesthetics is to select colors according to some method, typically by traversing a smooth path in a color model or by restricting the colors to planes or hexcones in a color space. This might mean using colors of constant lightness or value. Furthermore, colors are best spaced at equal *perceptual* distances (this is not the same as being at equally spaced increments of a coordinate, and can be difficult to implement). Recall too that linear interpolation (as in Gouraud shading) between two colors produces different results in different color spaces (see Exercise 13.10 and Color Plate II.14).

A random selection of different hues and saturations is usually quite garish. Alvy Ray Smith performed an informal experiment in which a 16×16 grid was filled with randomly generated colors. Not unexpectedly, the grid was unattractive. Sorting the 256 colors according to their H, S, and V values and redisplaying them on the grid in their new order improved the appearance of the grid remarkably.

More specific instances of these rules suggest that, if a chart contains just a few colors, the complement of one of the colors should be used as the background. A neutral (gray) background should be used for an image containing many different colors, since it is both harmonious and inconspicuous. If two adjoining colors are not particularly harmonious, a thin black border can be used to set them apart. This use of borders is also more effective for the achromatic (black/white) visual channel, since shape detection is facilitated by the black outline. Some of these rules are encoded in ACE (A Color Expert), an expert system for selecting user-interface colors [MEIE88]. In general, it is good to minimize the number of

Samsung/Google, EX1027, p. 613

different colors being used (except for shading of realistic images).

Color can be used for coding, as discussed in Chapter 9 and illustrated by Color Plate II.16. However, several cautions are in order. First, color codes can easily carry unintended meanings. Displaying the earnings of company A as red and those of company B as green might well suggest that company A is in financial trouble, because of our learned associations of colors with meanings. Bright, saturated colors stand out more strongly than do dimmer, paler colors, and may give unintended emphasis. Two elements of a display that have the same color may be seen as related by the same color code, even if they are not.

This problem often arises when color is used both to group menu items and to distinguish display elements, such as different layers of a printed circuit board or VLSI chip; for example, green display elements tend to be associated with menu items of the same color. This is one of the reasons that use of color in user-interface elements, such as menus, dialogue boxes, and window borders, should be restrained. (Another reason is to leave as many colors as possible free for the application program itself.)

A number of color usage rules are based on physiological rather than aesthetic considerations. For example, because the eye is more sensitive to spatial variation in intensity than it is to variation in chromaticity, lines, text, and other fine detail should vary from the background not just in chromaticity, but in brightness (perceived intensity) as well—especially for colors containing blue, since relatively few cones are sensitive to blue. Thus, the edge between two equal-brightness colored areas that differ only in the amount of blue will be fuzzy. On the other hand, blue-sensitive cones spread out farther on the retina than do red- and green-sensitive ones, so our peripheral color vision is better for blue (this is why many police-car flashers are now blue instead of red).

Blue and black differ very little in brightness, and are thus a particularly bad combination. Similarly, yellow on white is relatively hard to distinguish, because both colors are quite bright (see Exercise 13.11). Color plates I.28 and I.29 show a very effective use of yellow to highlight black text on a white background. The yellow contrasts very well with the black text and also stands out. In addition, the yellow highlight is not as overpowering as a black highlight with reversed text (that is, with the highlighted text in white on a black highlight), as is common on monochrome displays.

White text on a blue background provides a good contrast that is less harsh than white on black. It is good to avoid reds and greens with low saturation and luminance, as these are the colors confused by those of us who are red–green color blind, the most common form of color-perception deficiency. Meyer and Greenberg describe effective ways to choose colors for color-blind viewers [MEYE88].

The eye cannot distinguish the color of very small objects, as already remarked in connection with the YIQ NTSC color model, so color coding should not be applied to small objects. In particular, judging the color of objects subtending less than 20 to 40 minutes of arc is error-prone [BISH60, HAEU76]. An object 0.1 inches high, viewed from 24 inches (a typical viewing distance) subtends this much arc, which corresponds to about 7 pixels of height on a 1024-line display with a vertical height of 15 inches. It is clear that the color of a single pixel is quite difficult to discern (see Exercise 13.18).

The perceived color of a colored area is affected by the color of the surrounding area, as is very evident in Color Plate II.13; this effect is particularly problematic if colors are used to encode information. The effect is minimized when the surrounding areas are some shade of gray or are relatively unsaturated colors.

Samsung/Google, EX1027, p. 614

The color of an area can actually affect its perceived size. Cleveland and McGill discovered that a red square is perceived as larger than is a green square of equal size [CLEV83]. This effect could well cause the viewer to attach more importance to the red square than to the green ones.

If a user stares at a large area of highly saturated color for several seconds and then looks elsewhere, an afterimage of the large area will appear. This effect is disconcerting, and causes eye strain. Use of large areas of saturated colors is hence unwise. Also, large areas of different colors can appear to be at different distances from the viewer, because the index of refraction of light depends on wavelength. The eye changes its focus as the viewer's gaze moves from one colored area to another, and this change in focus gives the impression of differing depths. Red and blue, which are at opposite ends of the spectrum, have the strongest depth-disparity effect, with red appearing closer and blue more distant. Hence, simultaneously using blue for foreground objects and red for the background is unwise; the converse is fine.

With all these perils and pitfalls of color usage, is it surprising that one of our first-stated rules was to apply color conservatively?

13.6 SUMMARY

The importance of color in computer graphics will continue to increase as color monitors and color hardcopy devices become the norm in many applications. In this chapter, we have introduced those color concepts most relevant to computer graphics; for more information, see the vast literature on color, such as [BILL81; BOYN79; GREG66; HUNT87; JUDD75; WYSZ82]. More background on artistic and aesthetic issues in the use of color in computer graphics can be found in [FROM84; MARC82; MEIE88; MURC85]. The difficult problems of precisely calibrating monitors and matching the colors appearing on monitors with printed colors are discussed in [COWA83; STON88].

EXERCISES

13.1 Derive an equation for the number of intensities that can be represented by $m \times m$ pixel patterns, where each pixel has w bits.

13.2 Write the programs needed to gamma-correct a black-and-white display through a look-up table. Input parameters are γ, I_0, m, the number of intensities desired, and K, the constant in Eq. (13.5).

13.3 Write an algorithm to display a pixel array on a bilevel output device. The inputs to the algorithm are an $m \times m$ array of pixel intensities, with w bits per pixel, and an $n \times n$ growth sequence matrix. Assume that the output device has resolution of $m \cdot n \times m \cdot n$.

13.4 Repeat Exercise 13.3 by using ordered dither. Now the output device has resolution $m \times m$, the same as the input array of pixel intensities.

13.5 Write an algorithm to display a filled polygon on a bilevel device by using an $n \times n$ filling pattern.

13.6 When certain patterns are used to fill a polygon being displayed on an interlaced raster display, all of the "on" bits fall on either the odd or the even scan lines, introducing a slight amount of flicker. Revise the algorithm from Exercise 13.5 to permute rows of the $n \times n$ pattern so that alternate

Samsung/Google, EX1027, p. 615



Fig. 13.41 Results obtained by using intensity level 1 from Fig. 13.8 in two ways: (a) with alternation (intensified pixels are on both scan lines), and (b) without alternation (all intensified pixels are on the same scan line).

replications of the pattern will alternate use of the odd and even scan lines. Figure 13.41 shows the results obtained by using intensity level 1 from Fig. 13.8, with and without this alternation.

13.7 Given a spectral energy distribution, how would you find the dominant wavelength, excitation purity, and luminance of the color it represents?

13.8 Plot the locus of points of the constant luminance values 0.25, 0.50, and 0.75, defined by $Y = 0.30R + 0.59G + 0.11B$, on the RGB cube, the HLS double hexcone, and the HSV hexcone.

13.9 Why are the opposite ends of the spectrum in the CIE diagram connected by a *straight* line?

13.10 Express, in terms of R, G, and B: the I of YIQ, the V of HSV, and the L of HSL. Note that I, V, and L are not the same.

13.11 Calculate in YIQ color space the luminances of the additive and subtractive primaries. Rank the primaries by luminance. This ranking gives their relative intensities, both as displayed on a black-and-white television and as perceived by our eyes.

13.12 Discuss the design of a raster display that uses HSV or HLS, instead of RGB, as its color specification.

13.13 In which color models are the rules of color harmony most easily applied?

13.14 Verify that Eq. (13.27) can be rewritten as Eq. (13.29) when $R = G = B = 1$.

13.15 If the white color used to calculate C_r, C_g, and C_b in Eq. (13.29) is given by x_w, y_w, and Y_w rather than by X_w, Y_w, and Z_w, what are the algebraic expressions for C_r, C_g, and C_b?

13.16 Rewrite the HSV-to-RGB conversion algorithm to make it more efficient. Replace the assignment statements for p, q, and t with: $vs = v * s$; $vsf = vs * f$; $p = v - vs$; $q = v - vsf$; $t = p + vsf$. Also assume that R, G, and B are in the interval [0, 255], and see how many of the computations can be converted to integer.

13.17 Write a program that displays, side by side, two 16×16 grids. Fill each grid with colors. The left grid will have 256 colors randomly selected from HSV color space (created by using a random-number generator to choose one out of 10 equally spaced values for each of H, S, and V). The right grid contains the same 256 colors, sorted on H, S, and V. Experiment with the results obtained by varying which of H, S, and V is used as the primary sort key.

13.18 Write a program to display on a gray background small squares colored orange, red, green, blue, cyan, magenta, and yellow. Each square is separated from the others and is of size $n \times n$ pixels, where n is an input variable. How large must n be so that the colors of each square can be unambiguously judged from distances of 24 and of 48 inches? What should be the relation between the two values of n? What effect, if any, do different background colors have on this result?

13.19 Calculate the number of bits of look-up–table accuracy needed to store 256 different intensity levels given dynamic intensity ranges of 50, 100, and 200.

13.20 Write a program to interpolate linearly between two colors in RGB, HSV, and HSL. Accept the two colors as input, allowing them to be specified in any of these three models.

Samsung/Google, EX1027, p. 616

14
The Quest for Visual Realism

In previous chapters, we discussed graphics techniques involving simple 2D and 3D primitives. The pictures that we produced, such as the wireframe houses of Chapter 6, represent objects that in real life are significantly more complex in both structure and appearance. In this chapter, we introduce an increasingly important application of computer graphics: creating realistic images of 3D scenes.

What is a *realistic* image? In what sense a picture, whether painted, photographed, or computer-generated, can be said to be "realistic" is a subject of much scholarly debate [HAGE86]. We use the term rather broadly to refer to a picture that captures many of the effects of light interacting with real physical objects. Thus, we treat realistic images as a continuum and speak freely of pictures, and of the techniques used to create them, as being "more" or "less" realistic. At one end of the continuum are examples of what is often called *photographic realism* (or *photorealism*). These pictures attempt to synthesize the field of light intensities that would be focused on the film plane of a camera aimed at the objects depicted. As we approach the other end of the continuum, we find images that provide successively fewer of the visual cues we shall discuss.

You should bear in mind that a more realistic picture is not necessarily a more desirable or useful one. If the ultimate goal of a picture is to convey information, then a picture that is free of the complications of shadows and reflections may well be more successful than a *tour de force* of photographic realism. In addition, in many applications of the techniques outlined in the following chapters, reality is intentionally altered for aesthetic effect or to fulfill a naive viewer's expectations. This is done for the same reasons that science-fiction films feature the sounds of weapon blasts in outer space—an impossibility in a vacuum. For example, in depicting Uranus in Color Plate II.20, Blinn shined an extra light on the

Samsung/Google, EX1027, p. 617

night side of the planet and stretched the contrast to make all features visible simultaneously—the night side of the planet would have been black otherwise. Taking liberties with physics can result in attractive, memorable, and useful pictures!

Creating realistic pictures involves a number of stages that are treated in detail in the following chapters. Although these stages are often thought of as forming a conceptual pipeline, the order in which they are performed can vary, as we shall see, depending on the algorithms used. First, models of the objects are generated using methods discussed in Chapters 11, 12, and 20. Next, a viewing specification (as developed in Chapter 6) and lighting conditions are selected. Those surfaces visible to the viewer are then determined by algorithms discussed in Chapter 15. The color assigned to each pixel in a visible surface's projection is a function of the light reflected and transmitted by the objects and is determined by methods treated in Chapter 16. The resulting picture can then be combined with previously generated ones (e.g., to reuse a complex background) by using the compositing techniques of Chapter 17. Finally, if we are producing an animated sequence, time-varying changes in the models, lighting, and viewing specifications must be defined, as discussed in Chapter 21. The process of creating images from models is often called *rendering*. The term *rasterization* is also used to refer specifically to those steps that involve determining pixel values from input geometric primitives.

This chapter presents realistic rendering from a variety of perspectives. First, we look at some of the applications in which realistic images have been used. Then, we examine, in roughly historical progression, a series of techniques that make it possible to create successively more realistic pictures. Each technique is illustrated by a picture of a standard scene with the new technique applied to it. Next, we examine the problems caused by aliasing, which must be dealt with when images are represented as discrete arrays of pixels. Finally, we conclude with suggestions about how to approach the following chapters.

14.1 WHY REALISM?

The creation of realistic pictures is an important goal in fields such as simulation, design, entertainment and advertising, research and education, and command and control.

Simulation systems present images that not only are realistic, but also change dynamically. For example, a flight simulator shows the view that would be seen from the cockpit of a moving plane. To produce the effect of motion, the system generates and displays a new, slightly different view many times per second. Simulators such as those shown in Color Plate I.5 have been used to train the pilots of spacecraft, airplanes, and boats—and, more recently, drivers of cars.

Designers of 3D objects such as automobiles, airplanes, and buildings want to see how their preliminary designs look. Creating realistic computer-generated images is often an easier, less expensive, and more effective way to see preliminary results than is building models and prototypes, and also allows more alternative designs to be considered. If the design work itself is also computer-based, a digital description of the object may already be available to use in creating the images. Ideally, the designer can also interact with the displayed image to modify the design. Color Plate II.17 shows an image produced by an automotive-design system to determine what a car will look like under a variety of lighting conditions. Realistic graphics is often coupled with programs that analyze other aspects of

Samsung/Google, EX1027, p. 618

the object being designed, such as its mass properties or its response to stress.

Computer-generated imagery is used extensively in the entertainment world, both in traditional animated cartoons and in realistic and surrealistic images for logos, advertisements, and science-fiction movies (see Color Plates D, F, I.11, I.12, and II.18). Computer-generated cartoons can mimic traditional animation, but can also transcend manual techniques by introducing more complicated motion and richer or more realistic imagery. Some complex realistic images can be produced at less cost than filming them from physical models of the objects. Other images have been generated that would have been extremely difficult or impossible to stage with real models. Special-purpose hardware and software created for use in entertainment include sophisticated paint systems and real-time systems for generating special effects and for combining images. As technology improves, home and arcade video games generate increasingly realistic images.

Realistic images are becoming an essential tool in research and education. A particularly important example is the use of graphics in molecular modeling, as shown in Color Plate II.19. It is interesting how the concept of realism is stretched here: The realistic depictions are not of "real" atoms, but rather of stylized ball-and-stick and volumetric models that allow larger structures to be built than are feasible with physical models, and that permit special effects, such as animated vibrating bonds and color changes representing reactions. On a macroscopic scale, movies made at JPL show NASA space-probe missions, depicted in Color Plate II.20.

Another application for realistic imagery is in command and control, in which the user needs to be informed about and to control the complex process represented by the picture. Unlike simulations, which attempt to mimic what a user would actually see and feel in the simulated situation, command and control applications often create symbolic displays that emphasize certain data and suppress others to aid in decision making.

14.2 FUNDAMENTAL DIFFICULTIES

A fundamental difficulty in achieving total visual realism is the complexity of the real world. Observe the richness of your environment. There are many surface textures, subtle color gradations, shadows, reflections, and slight irregularities in the surrounding objects. Think of patterns on wrinkled cloth, the texture of skin, tousled hair, scuff marks on the floor, and chipped paint on the wall. These all combine to create a "real" visual experience. The computational costs of simulating these effects can be high: Creating pictures such as those of Color Plates A–H can take many minutes or even hours on powerful computers.

A more easily met subgoal in the quest for realism is to provide sufficient information to let the viewer understand the 3D spatial relationships among several objects. This subgoal can be achieved at a significantly lower cost and is a common requirement in CAD and in many other application areas. Although highly realistic images convey 3D spatial relationships, they usually convey much more as well. For example, Fig. 14.1, a simple line drawing, suffices to persuade us that one building is partially behind the other. There is no need to show building surfaces filled with shingles and bricks, or shadows cast by the buildings. In fact, in some contexts, such extra detail may only distract the viewer's attention from more important information being depicted.

Samsung/Google, EX1027, p. 619

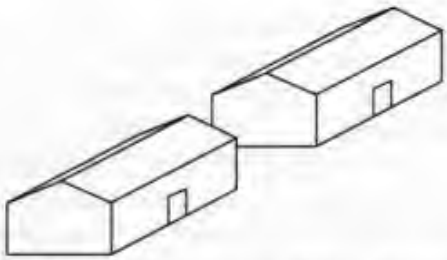


Fig. 14.1 Line drawing of two houses.

One long-standing difficulty in depicting spatial relationships is that most display devices are 2D. Therefore, 3D objects must be projected into 2D, with considerable attendant loss of information—which can sometimes create ambiguities in the image. Some of the techniques introduced in this chapter can be used to add back information of the type normally found in our visual environment, so that human depth-perception mechanisms resolve the remaining ambiguities properly.

Consider the Necker cube illusion of Fig. 14.2(a), a 2D projection of a cube; we do not know whether it represents the cube in part (b) or that in part (c) of this figure. Indeed, the viewer can easily "flip-flop" between the alternatives, because Fig. 14.2(a) does not contain enough visual information for an unambiguous interpretation.

The more the viewers know about the object being displayed, the more readily they can form what Gregory calls an *object hypothesis* [GREG70]. Figure 14.3 shows the Schröder stairway illusion—are we looking down a stairway, or looking up from underneath it? We are likely to choose the former interpretation, probably because we see stairways under our feet more frequently than over our heads and therefore "know" more about stairways viewed from above. With a small stretch of the imagination, however, we can visualize the alternative interpretation of the figure. Nevertheless, with a blink of the eye, a reversal occurs for most viewers and the stairway again appears to be viewed from above. Of course, additional context, such as a person standing on the steps, will resolve the ambiguity.

In the following sections, we list some of the steps along the path toward realistic images. The path has actually been a set of intertwined trails, rather than a single straight road, but we have linearized it for the sake of simplicity, providing a purely descriptive introduction to the detailed treatment in subsequent chapters. We mention first techniques applicable to static line drawings. These methods concentrate on ways to present the 3D spatial relationships among several objects on a 2D display. Next come techniques for

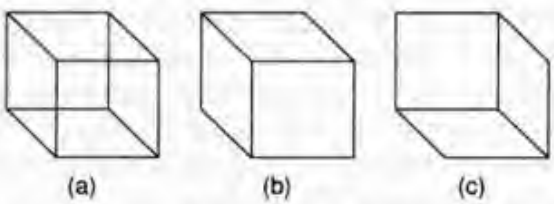


Fig. 14.2 The Necker cube illusion. Is the cube in (a) oriented like the cube in (b) or like that in (c)?

Samsung/Google, EX1027, p. 620

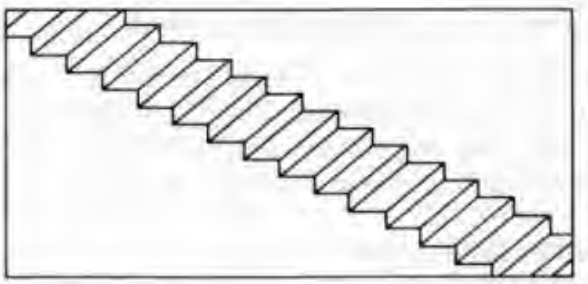


Fig. 14.3 The Schröder stairway illusion. Is the stairway being viewed from above or from below?

shaded images, made possible by raster graphics hardware, that concentrate on the interaction of objects with light. These are followed by the issues of increased model complexity and dynamics, applicable to both line and shaded pictures. Finally, we discuss the possibilities of true 3D images, advances in display hardware, and the future place of picture generation in the context of full, interactive environmental synthesis.

14.3 RENDERING TECHNIQUES FOR LINE DRAWINGS

In this section, we focus on a subgoal of realism: showing 3D depth relationships on a 2D surface. This goal is served by the planar geometric projections defined in Chapter 6.

14.3.1 Multiple Orthographic Views

The easiest projections to create are parallel orthographics, such as plan and elevation views, in which the projection plane is perpendicular to a principal axis. Since depth information is discarded, plan and elevations are typically shown together, as with the top, front, and side views of a block letter "L" in Fig. 14.4. This particular drawing is not difficult to understand; however, understanding drawings of complicated manufactured parts from a set of such views may require many hours of study. Training and experience sharpen one's interpretive powers, of course, and familiarity with the types of objects being represented hastens the formulation of a correct object hypothesis. Still, scenes as complicated as that of our "standard scene" shown in Color Plate II.21 are often confusing when shown in only three such projections. Although a single point may be unambiguously located from three mutually perpendicular orthographics, multiple points and lines may conceal one another when so projected.

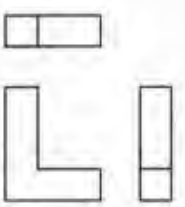


Fig. 14.4 Front, top, and side orthographic projections of the block letter "L."

Samsung/Google, EX1027, p. 621

14.3.2 Axonometric and Oblique Projections

In axonometric and oblique projections, a point's z coordinate influences its x and y coordinates in the projection, as exemplified by Color Plate II.22. These projections provide constant foreshortening, and therefore lack the convergence of parallel lines and the decreasing size of objects with increasing distance that perspective projection provides.

14.3.3 Perspective Projections

In perspective projections, an object's size is scaled in inverse proportion to its distance from the viewer. The perspective projection of a cube shown in Fig. 14.5 reflects this scaling. There is still ambiguity, however; the projection could just as well be a picture frame, or the parallel projection of a truncated pyramid, or the perspective projection of a rectangular parallelepiped with two equal faces. If one's object hypothesis is a truncated pyramid, then the smaller square represents the face closer to the viewer; if the object hypothesis is a cube or rectangular parallelepiped, then the smaller square represents the face farther from the viewer.

Our interpretation of perspective projections is often based on the assumption that a smaller object is farther away. In Fig. 14.6, we would probably assume that the larger house is nearer to the viewer. However, the house that appears larger (a mansion, perhaps) may actually be more distant than the one that appears smaller (a cottage, for example), at least as long as there are no other cues, such as trees and windows. When the viewer knows that the projected objects have many parallel lines, perspective further helps to convey depth, because the parallel lines seem to converge at their vanishing points. This convergence may actually be a stronger depth cue than the effect of decreasing size. Color Plate II.23 shows a perspective projection of our standard scene.

14.3.4 Depth Cueing

The depth (distance) of an object can be represented by the intensity of the image: Parts of objects that are intended to appear farther from the viewer are displayed at lower intensity (see Color Plate II.24). This effect is known as *depth cueing*. Depth cueing exploits the fact that distant objects appear dimmer than closer objects, especially if seen through haze. Such effects can be sufficiently convincing that artists refer to the use of changes in intensity (as well as in texture, sharpness, and color) to depict distance as *aerial perspective*. Thus, depth cueing may be seen as a simplified version of the effects of atmospheric attenuation.

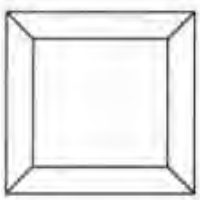


Fig. 14.5 Perspective projection of a cube.

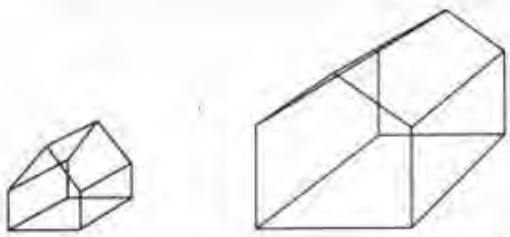


Fig. 14.6 Perspective projection of two houses.

Samsung/Google, EX1027, p. 622

In vector displays, depth cueing is implemented by interpolating the intensity of the beam along a vector as a function of its starting and ending z coordinates. Color graphics systems usually generalize the technique to support interpolating between the color of a primitive and a user-specified depth-cue color, which is typically the color of the background. To restrict the effect to a limited range of depths, PHIGS+ allows the user to specify front and back depth-cueing planes between which depth cueing is to occur. A separate scale factor associated with each plane indicates the proportions of the original color and the depth-cue color to be used in front of the front plane and behind the back plane. The color of points between the planes is linearly interpolated between these two values. The eye's intensity resolution is lower than its spatial resolution, so depth cueing is not useful for accurately depicting small differences in distance. It is quite effective, however, in depicting large differences, or as an exaggerated cue in depicting small ones.

14.3.5 Depth Clipping

Further depth information can be provided by *depth clipping*. The back clipping plane is placed so as to cut through the objects being displayed, as shown in Color Plate II.25. Partially clipped objects are then known by the viewer to be cut by the clipping plane. A front clipping plane may also be used. By allowing the position of one or both planes to be varied dynamically, the system can convey more depth information to the viewer. Back-plane depth clipping can be thought of as a special case of depth cueing: In ordinary depth cueing, intensity is a smooth function of z; in depth clipping, it is a step function. Color Plate II.25 combines both techniques. A technique related to depth clipping is highlighting all points on the object intersected by some plane. This technique is especially effective when the slicing plane is shown moving through the object dynamically, and has even been used to help illustrate depth along a fourth dimension [BANC77].

14.3.6 Texture

Simple vector textures, such as cross-hatching, may be applied to an object. These textures follow the shape of an object and delineate it more clearly. Texturing one of a set of otherwise identical faces can clarify a potentially ambiguous projection. Texturing is especially useful in perspective projections, as it adds yet more lines whose convergence and foreshortening may provide useful depth cues.

14.3.7 Color

Color may be used symbolically to distinguish one object from another, as in Color Plate II.26, in which each object has been assigned a different color. Color can also be used in line drawings to provide other information. For example, the color of each vector of an object may be determined by interpolating colors that encode the temperatures at the vector's endpoints.

14.3.8 Visible-Line Determination

The last line-drawing technique we mention is *visible-line determination* or *hidden-line removal*, which results in the display of only visible (i.e., unobscured) lines or parts of

Samsung/Google, EX1027, p. 623

lines. Only surfaces, bounded by edges (lines), can obscure other lines. Thus, objects that are to block others must be modeled either as collections of surfaces or as solids.

Color Plate II.27 shows the usefulness of hidden-line removal. Because hidden-line–removed views conceal *all* the internal structure of opaque objects, they are not necessarily the most effective way to show depth relations. Hidden-line–removed views convey less depth information than do exploded and cutaway views. Showing hidden lines as dashed lines can be a useful compromise.

14.4 RENDERING TECHNIQUES FOR SHADED IMAGES

The techniques mentioned in Section 14.3 can be used to create line drawings on both vector and raster displays. The techniques introduced in this section exploit the ability of raster devices to display shaded areas. When pictures are rendered for raster displays, problems are introduced by the relatively coarse grid of pixels on which smooth contours and shading must be reproduced. The simplest ways to render shaded pictures fall prey to the problem of aliasing, first encountered in Section 3.17. In Section 14.10, we introduce the theory behind aliasing, and explain how to combat aliasing through antialiasing. Because of the fundamental role that antialiasing plays in producing high-quality pictures, all the pictures in this section have been created with antialiasing.

14.4.1 Visible-Surface Determination

By analogy to visible-line determination, *visible-surface determination* or *hidden-surface removal*, entails displaying only those parts of surfaces that are visible to the viewer. As we have seen, simple line drawings can often be understood without visible-line determination. When there are few lines, those in front may not seriously obstruct our view of those behind them. In raster graphics, on the other hand, if surfaces are rendered as opaque areas, then visible-surface determination is essential for the picture to make sense. Color Plate II.28 shows an example in which all faces of an object are painted the same color.

14.4.2 Illumination and Shading

A problem with Color Plate II.28 is that each object appears as a flat silhouette. Our next step toward achieving realism is to shade the visible surfaces. Ultimately, each surface's appearance should depend on the types of light sources illuminating it, its properties (color, texture, reflectance), and its position and orientation with respect to the light sources, viewer, and other surfaces.

In many real visual environments, a considerable amount of *ambient light* impinges from all directions. Ambient light is the easiest kind of light source to model, because in a simple lighting model it is assumed to produce constant illumination on all surfaces, regardless of their position or orientation. Using ambient light by itself produces very unrealistic images, however, since few real environments are illuminated solely by uniform ambient light. Color Plate II.28 is an example of a picture shaded this way.

A *point source*, whose rays emanate from a single point, can approximate a small incandescent bulb. A *directional source*, whose rays all come from the same direction, can be used to represent the distant sun by approximating it as an infinitely distant point source.

Samsung/Google, EX1027, p. 624

Modeling these sources requires additional work because their effect depends on the surface's orientation. If the surface is *normal* (perpendicular) to the incident light rays, it is brightly illuminated; the more oblique the surface is to the light rays, the less its illumination. This variation in illumination is, of course, a powerful cue to the 3D structure of an object. Finally, a *distributed* or *extended source*, whose surface area emits light, such as a bank of fluorescent lights, is even more complex to model, since its light comes from neither a single direction nor a single point. Color Plate II.29 shows the effect of illuminating our scene with ambient and point light sources, and shading each polygon separately.

14.4.3 Interpolated Shading

Interpolated shading is a technique in which shading information is computed for each polygon vertex and interpolated across the polygons to determine the shading at each pixel. This method is especially effective when a polygonal object description is intended to approximate a curved surface. In this case, the shading information computed at each vertex can be based on the surface's actual orientation at that point and is used for all of the polygons that share that vertex. Interpolating among these values across a polygon approximates the smooth changes in shade that occur across a curved, rather than planar, surface.

Even objects that are supposed to be polyhedral, rather than curved, can benefit from interpolated shading, since the shading information computed for each vertex of a polygon may differ, although typically much less dramatically than for a curved object. When shading information is computed for a true polyhedral object, the value determined for a polygon's vertex is used only for that polygon and not for others that share the vertex. Color Plate II.30 shows Gouraud shading, a kind of interpolated shading discussed in Section 16.2.

14.4.4 Material Properties

Realism is further enhanced if the *material properties* of each object are taken into account when its shading is determined. Some materials are dull and disperse reflected light about equally in all directions, like a piece of chalk; others are shiny and reflect light only in certain directions relative to the viewer and light source, like a mirror. Color Plate II.31 shows what our scene looks like when some objects are modeled as shiny. Color Plate II.32 uses Phong shading, a more accurate interpolated shading method (Section 16.2).

14.4.5 Modeling Curved Surfaces

Although interpolated shading vastly improves the appearance of an image, the object geometry is still polygonal. Color Plate II.33 uses object models that include curved surfaces. Full shading information is computed at each pixel in the image.

14.4.6 Improved Illumination and Shading

One of the most important reasons for the "unreal" appearance of most computer graphics images is the failure to model accurately the many ways that light interacts with objects.

Samsung/Google, EX1027, p. 625

Color Plate II.34 uses better illumination models. Sections 16.7–13 discuss progress toward the design of efficient, physically correct illumination models, resulting in pictures such as Color Plates III.19–III.29 and the jacket of this book (Color Plate I.9).

14.4.7 Texture

Object texture not only provides additional depth cues, as discussed in Section 14.3.6, but also can mimic the surface detail of real objects. Color Plates II.35 and II.36 show a variety of ways in which texture may be simulated, ranging from varying the surface's color (as was done with the patterned ball), to actually deforming the surface geometry (as was done with the striated torus and crumpled cone in Color Plate II.36).

14.4.8 Shadows

We can introduce further realism by reproducing shadows cast by objects on one another. Note that this technique is the first we have met in which the appearance of an object's visible surfaces is affected by other objects. Color Plate II.36 shows the shadows cast by the lamp at the rear of the scene. Shadows enhance realism and provide additional depth cues: If object A casts a shadow on surface B, then we know that A is between B and a direct or reflected light source. A point light source casts sharp shadows, because from any point it is either totally visible or invisible. An extended light source casts "soft" shadows, since there is a smooth transition from those points that see all of the light source, through those that see only part of it, to those that see none of it.

14.4.9 Transparency and Reflection

Thus far, we have dealt with opaque surfaces only. Transparent surfaces can also be useful in picture making. Simple models of transparency do not include the refraction (bending) of light through a transparent solid. Lack of refraction can be a decided advantage, however, if transparency is being used not so much to simulate reality as to reveal an object's inner geometry. More complex models include refraction, diffuse translucency, and the attenuation of light with distance. Similarly, a model of light reflection may simulate the sharp reflections of a perfect mirror reflecting another object or the diffuse reflections of a less highly polished surface. Color Plate II.37 shows the effect of reflection from the floor and teapot; Color Plates III.7 and III.10 show transparency.

Like modeling shadows, modeling transparency or reflection requires knowledge of other surfaces besides the surface being shaded. Furthermore, refractive transparency is the first effect we have mentioned that requires objects actually to be modeled as solids rather than just as surfaces! We must know something about the materials through which a light ray passes and the distance it travels to model its refraction properly.

14.4.10 Improved Camera Models

All the pictures shown so far are based on a camera model with a pinhole lens and an infinitely fast shutter: All objects are in sharp focus and represent the world at one instant in time. It is possible to model more accurately the way that we (and cameras) see the world.

Samsung/Google, EX1027, p. 626

For example, by modeling the focal properties of lenses, we can produce pictures, such as Color Plates II.38 and II.39, that show *depth of field*: Some parts of objects are in focus, whereas closer and farther parts are out of focus. Other techniques allow the use of special effects, such as fish-eye lenses. The lack of depth-of-field effects is responsible in part for the surreal appearance of many early computer-generated pictures.

Moving objects look different from stationary objects in a picture taken with a regular still or movie camera. Because the shutter is open for a finite period of time, visible parts of moving objects are blurred across the film plane. This effect, called *motion blur*, is simulated in Color Plates III.16 and IV.14. Motion blur not only captures the effects of motion in stills, but is of crucial importance in producing high-quality animation, as described in Chapter 21.

14.5 IMPROVED OBJECT MODELS

Independent of the rendering technology used, the search for realism has concentrated in part on ways of building more convincing models, both static and dynamic. Some researchers have developed models of special kinds of objects such as gases, waves, mountains, and trees; see, for example, Color Plates IV.11–IV.21. Other investigators have concentrated on automating the positioning of large numbers of objects, such as trees in a forest, which would be too tedious to do by hand (Color Plate IV.25). These techniques are covered in Chapter 20.

14.6 DYNAMICS

By *dynamics*, we mean changes that spread across a sequence of pictures, including changes in position, size, material properties, lighting, and viewing specification—indeed, changes in any parts of the scene or the techniques applied to it. The benefits of dynamics can be examined independently of the progression toward more realistic static images.

Perhaps the most popular kind of dynamics is motion dynamics, ranging from simple transformations performed under user control to complex animation, as described in Chapter 21. Motion has been an important part of computer graphics since the field's inception. In the early days of slow raster graphics hardware, motion capability was one of the strong competitive selling points of vector graphics systems. If a series of projections of the same object, each from a slightly different viewpoint around the object, is displayed in rapid succession, then the object appears to rotate. By integrating the information across the views, the viewer creates an object hypothesis.

A perspective projection of a rotating cube, for instance, provides several types of information. There is the series of different projections, which are themselves useful. This is supplemented by the motion effect, in which the maximum linear velocity of points near the center of rotation is lower than that of points distant from the center of rotation. This difference can help to clarify the relative distance of a point from the center of rotation. Also, the changing sizes of different parts of the cube as they change distance under perspective projection provide additional cues about the depth relationships. Motion

Samsung/Google, EX1027, p. 627

becomes even more powerful when it is under the interactive control of the viewer. By selectively transforming an object, viewers may be able to form an object hypothesis more quickly.

In contrast to the use of simple transformations to clarify complex models, surprisingly simple models look extremely convincing if they move in a realistic fashion. For example, just a few points positioned at key parts of a human model, when moved naturally, can provide a convincing illusion of a person in motion. The points themselves do not "look like" a person, but they do inform the viewer that a person is present. It is also well known that objects in motion can be rendered with less detail than is needed to represent static objects, because the viewer has more difficulty picking out details when an object is moving. Television viewers, for example, are often surprised to discover how poor and grainy an individual television frame appears.

14.7 STEREOPSIS

All the techniques we have discussed thus far present the same image to both eyes of the viewer. Now conduct an experiment: Look at your desk or table top first with one eye, then with the other. The two views differ slightly because our eyes are separated from each other by a few inches, as shown in Fig. 14.7. The *binocular disparity* caused by this separation provides a powerful depth cue called *stereopsis* or *stereo vision*. Our brain fuses the two separate images into one that is interpreted as being in 3D. The two images are called a *stereo pair*; stereo pairs were used in the stereo viewers popular around the turn of the century, and are used today in the common toy, the View-Master. Color Plate II.19 shows a stereo pair of a molecule. You can fuse the two images into one 3D image by viewing them such that each eye sees only one image; you can do this, for example, by placing a stiff piece of paper between the two images perpendicular to the page. Some people can see the effect without any need for the piece of paper, and a small number of people cannot see it at all.

A variety of other techniques exists for providing different images to each eye, including glasses with polarizing filters and holography. Some of these techniques make possible true 3D images that occupy space, rather than being projected on a single plane. These displays can provide an additional 3D depth cue: Closer objects actually are closer,

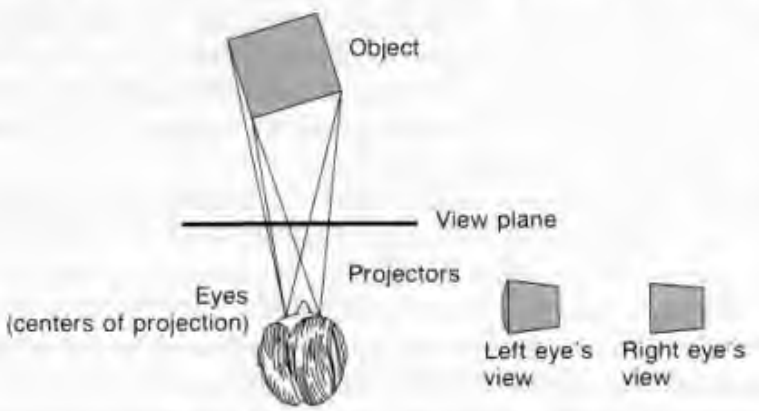


Fig. 14.7 Binocular disparity.

Samsung/Google, EX1027, p. 628

just as in real life, so the viewer's eyes focus differently on different objects, depending on each object's proximity. Methods for producing and viewing stereo images are examined in more detail in Section 18.11.5; the mathematics of stereo projection is described in Exercise 6.27.

14.8 IMPROVED DISPLAYS

In addition to improvements in the software used to design and render objects, improvements in the displays themselves have heightened the illusion of reality. The history of computer graphics is in part that of a steady improvement in the visual quality achieved by display devices. Still, a modern monitor's color gamut and its dynamic intensity range are both a small subset of what we can see. We have a long way to go before the image on our display can equal the crispness and contrast of a well-printed professional photograph! Limited display resolution makes it impossible to reproduce extremely fine detail. Artifacts such as a visible phosphor pattern, glare from the screen, geometric distortion, and the stroboscopic effect of frame-rate flicker are ever-present reminders that we are viewing a display. The display's relatively small size, compared with our field of vision, also helps to remind us that the display is a window on a world, rather than a world itself.

14.9 INTERACTING WITH OUR OTHER SENSES

Perhaps the final step toward realism is the integration of realistic imagery with information presented to our other senses. Computer graphics has a long history of programs that rely on a variety of input devices to allow user interaction. Flight simulators are a current example of the coupling of graphics with realistic engine sounds and motion, all offered in a mocked-up cockpit to create an entire environment. The head-worn simulator of Color Plate I.16 monitors head motion, making possible another important 3D depth cue called *head-motion parallax*: when the user moves her head from side to side, perhaps to try to see more of a partially hidden object, the view changes as it would in real life. Other active work on head-mounted displays centers on the exploration of *virtual worlds*, such as the insides of molecules or of buildings that have not yet been constructed [CHUN89].

Many current arcade games feature a car or plane that the player rides, moving in time to a simulation that includes synthesized or digitized images, sound, and force feedback, as shown in Color Plate I.7. This use of additional output and input modalities points the way to systems of the future that will provide complete immersion of all the senses, including hearing, touch, taste, and smell.

14.10 ALIASING AND ANTIALIASING

In Section 3.17, we introduced the problem of aliasing and discussed some basic techniques for generating antialiased 2D primitives. Here we examine aliasing in more detail so that we can understand when and why it occurs, laying the groundwork for incorporating antialiasing into the visible-surface and shading algorithms covered in the following chapters. Additional material may be found in [CROW77b; CROW81]; an excellent set of examples is included in [BLIN89a; BLIN89b].

Samsung/Google, EX1027, p. 629

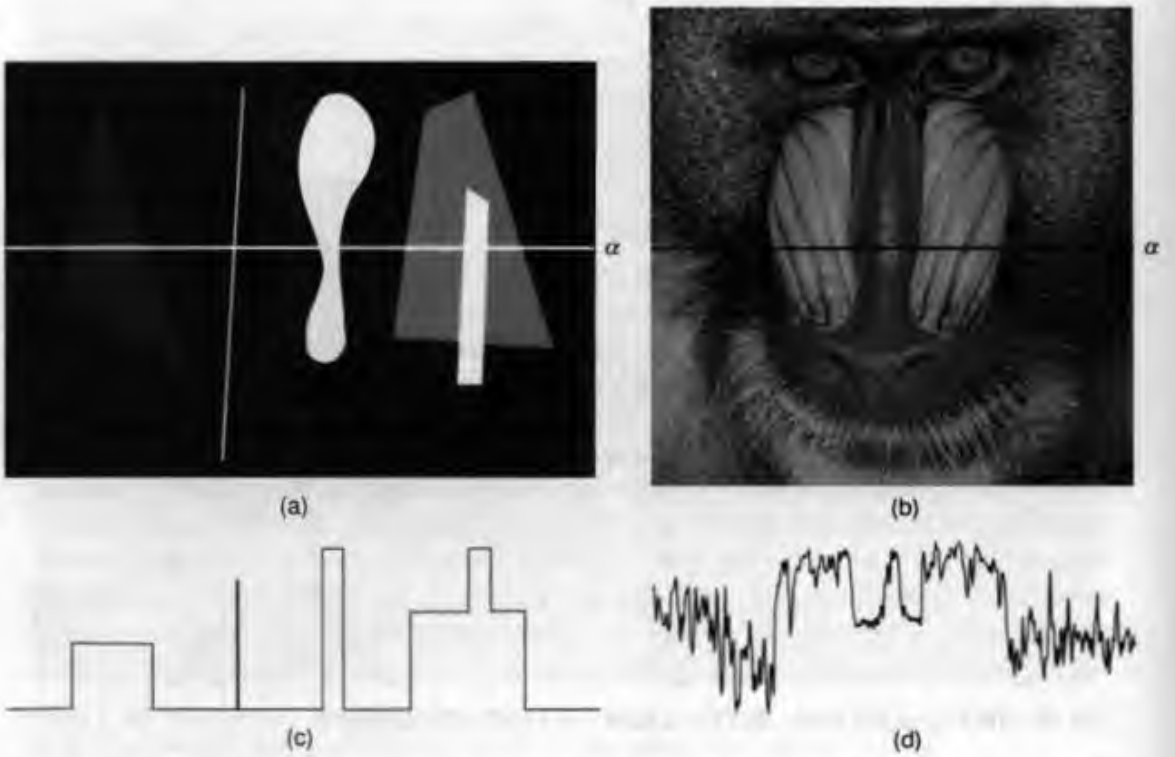


Fig. 14.8 Image. (a) Graphical primitives. (b) Mandrill. (c) Intensity plot of scan line α in (a). (d) Intensity plot of scan line α in (b). (Part d is courtesy of George Wolberg, Columbia University.)

To understand aliasing, we have to introduce some basic concepts from the field of signal processing. We start with the concept of a *signal*, which is a function that conveys information. Signals are often thought of as functions of time, but can equally well be functions of other variables. Since we can think of images as intensity variations over space, we will refer to signals in the *spatial domain* (as functions of spatial coordinates), rather than in the *temporal domain* (as functions of time). Although images are 2D functions of two independent spatial variables (x and y), for convenience our examples will often use the 1D case of a single spatial variable x. This case can be thought of as an infinitesimally thin slice through the image, representing intensity along a single horizontal line. Figure 14.8(a) and (b) show 2D signals, and parts (c) and (d) of the figure show plots of the intensity along the horizontal line α.

Signals can be classified by whether or not they have values at all points in the spatial domain. A *continuous signal*[1] is defined at a continuum of positions in space; a *discrete signal* is defined at a set of discrete points in space. Before scan conversion, the projection of our 3D objects onto the view plane may be treated as a continuous 2D signal whose value

[1]Not to be confused with the definition of continuity in calculus.

Samsung/Google, EX1027, p. 630

at each infinitesimal point in the plane indicates the intensity at that point. In contrast, the array of pixel values in the graphics system's frame buffer is a discrete 2D signal whose value is defined only at the positions in the array. Our rendering algorithms must determine the intensities of the finite number of pixels in the array so that they best represent the continuous 2D signal defined by the projection. The precise meaning of "best represent" is not at all obvious, however. We shall discuss this problem further.

A continuous signal may contain arbitrarily fine detail in the form of very rapid (high-frequency) variations in its value as its continuous parameter is changed. Since a discrete signal can change value only at discrete points, it clearly has a maximum rate of variation. Therefore, it should be clear that converting a continuous signal to a finite array of values may result in a loss of information. Our goal is to ensure that as little information as possible is lost, so that the resulting pixel array can be used to display a picture that looks as much as possible like the original signal would look if we were able to display it directly. The process of selecting a finite set of values from a signal is known as *sampling*, and the selected values are called *samples*. Once we have selected these samples, we must then display them using a process, known as *reconstruction*, that attempts to recreate the original continuous signal from the samples. The array of pixels in the frame buffer is reconstructed by the graphics system's display hardware, which converts these discrete intensity values to continuous, analog voltages that are applied to the CRT's electron gun (see Chapter 4). An idealized version of this pipeline is shown in Fig. 14.9. Signal-processing theory [GONZ87] establishes the minimum frequency at which samples must be selected from a given signal to reconstruct an exact copy of the signal, and specifies how to perform the reconstruction process. As we show later, however, this minimum sampling frequency will be infinite for many kinds of signals in which we are interested, so perfect reconstruction will often be impossible. Furthermore, as described in Section 14.10.5, the reconstruction method typically used by the display hardware differs from the approach prescribed by theory. Therefore, even properly sampled signals will not be reconstructed perfectly.

14.10.1 Point Sampling

The most straightforward way to select each pixel's value is known as *point sampling*. In point sampling, we select one point for each pixel, evaluate the original signal at this point, and assign its value to the pixel. The points that we select are typically arranged in a regular grid, as shown in Fig. 14.10. Unlike the scan-conversion algorithms of Chapter 3, projected vertices are not constrained to lie on integer grid points. Because the signal's values at a finite set of points are sampled, however, important features of the signal may be missed. For example, objects A and C in Fig. 14.10 are represented by the samples, whereas objects B and D are not. To make matters worse, if the viewing specification changes slightly or if the objects move, objects may pop in or out of visibility. What if we sample at a higher rate? The more samples we collect from the signal, the more we know about it. For example, we can see easily that, by increasing sufficiently the number of samples taken horizontally and vertically in Fig. 14.10, we can make sure that no object is missed in that particular picture. This is a necessary, but *not* a sufficient condition for adequate sampling. Nevertheless, sampling at a higher rate, we can generate images with more pixels representing each portion of the picture. We can also generate an image with

Samsung/Google, EX1027, p. 631

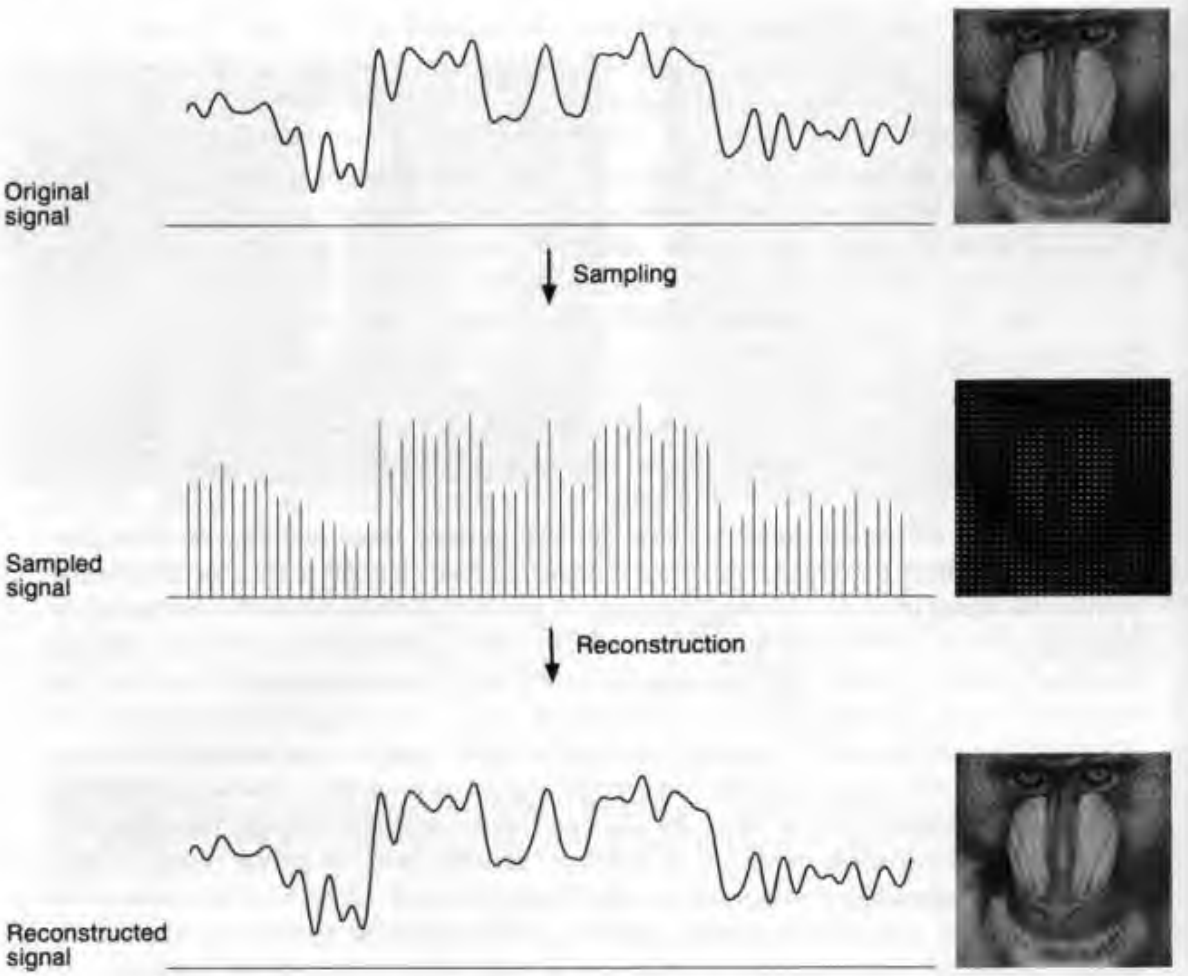


Fig. 14.9 The original signal is sampled, and the samples are used to reconstruct the signal. (Sampled 2D image is an approximation, since point samples have no area.) (Courtesy of George Wolberg, Columbia University.)

fewer pixels, by combining several adjacent samples (e.g., by averaging) to determine the value of each pixel of the smaller image. This means that all the features that would be present in the larger image at least contribute to the smaller one.

The approach of taking more than one sample for each pixel and combining them is known as *supersampling*. It actually corresponds to reconstructing the signal and resampling the reconstructed signal. For reasons described later, sampling the reconstructed signal is often better than sampling the original signal. This technique is popular in computer graphics precisely because it is so easy and often achieves good results, despite the obvious increase in computation. But, how many samples are enough? How do we know that there are no features that our samples are missing? Merely testing whether every object's projection is sampled is not sufficient. The projection may have a complex shape or variations in shading intensity that the samples do not reflect. We would like some way to guarantee that the samples we take are spaced close enough to reconstruct the original

Samsung/Google, EX1027, p. 632

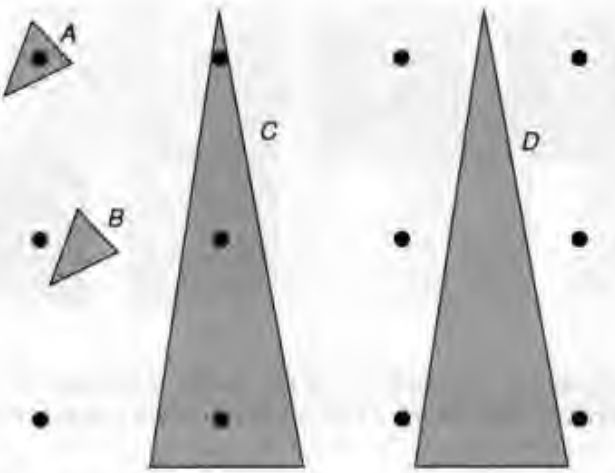


Fig. 14.10 Point-sampling problems. Samples are shown as black dots (•). Objects *A* and *C* are sampled, but corresponding objects *B* and *D* are not.

signal. As we shall see, sampling theory tells us that, on the basis of a particular signal's properties, we can compute a minimum sampling rate that will be adequate. Unfortunately, the rate turns out to be infinite for certain kinds of signals, including the signal shown in Fig. 14.10! We shall explain the reason for this in more detail later; for now, we can see that taking a finite number of samples cannot guarantee to capture the exact x coordinate at which the intensity jumps from one value to another in the figure. Furthermore, even if we find a finite sampling rate at which all of the current objects are sampled, we can always imagine adding just one more object positioned between samples that will be missed entirely.

14.10.2 Area Sampling

The problem of objects "falling between" samples and being missed suggests another approach: integrating the signal over a square centered about each grid point, dividing by the square's area, and using this average intensity as that of the pixel. This technique, called *unweighted area sampling*, was introduced in Chapter 3. The array of nonoverlapping squares is typically thought of as representing the pixels. Each object's projection, no matter how small, contributes to those pixels that contain it, in strict proportion to the amount of each pixel's area it covers, and without regard to the location of that area in the pixel, as shown by the equal weighting function of Fig. 14.11(a). No objects are missed, as may happen with point sampling. The definition of the definite integral requires evaluating a function at many points of an interval, and then taking the limit as the number of points increases. Thus, integrating amounts to a kind of infinite sampling process.

Unweighted area sampling has drawbacks caused by this evenhandedness with which objects are treated. Consider a small black object wholly contained inside of one of the pixels and surrounded by a white background, as in Fig. 14.11(b). This small object may move freely inside the pixel, and for each position the value computed for the pixel (shown as the pixel's shade) remains the same. As soon as the object crosses over into an adjoining pixel, however, the values of the original pixel and the adjoining pixel are both affected.

Samsung/Google, EX1027, p. 633

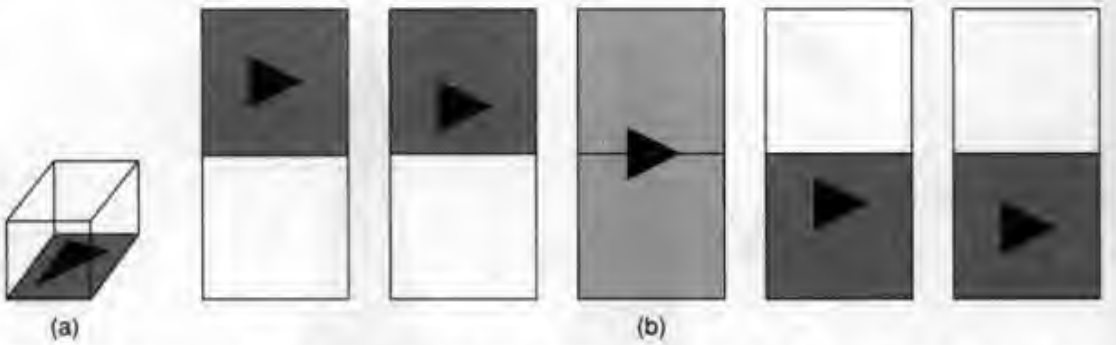


Fig. 14.11 Unweighted area sampling. (a) All points in the pixel are weighted equally. (b) Changes in computed intensities as an object moves between pixels.

Thus, the object causes the image to change only when it crosses pixel boundaries. As the object moves farther from the center of one pixel and closer to the center of another, however, we would like this change to be represented in the image. In other words, we would like the object's contribution to the pixel's intensity to be *weighted* by its distance from the pixel's center: the farther away it is, the less it should contribute.

In Chapter 3, we noted that *weighted area sampling* allows us to assign different weights to different parts of the pixel, and we suggested that the weighting functions of adjacent pixels should overlap. To see why the overlap is needed, we consider a weighting function consisting of an upright pyramid erected over a single pixel, as shown in Fig. 14.12(a). Under this weighting, as desired, an object contributes less to a pixel as it moves away from the pixel's center. But a drawback of unweighted area sampling still remains: An object contributes to only the single pixel that contains it. Consider a subpixel-sized black object moving over a white background from the center of one pixel to the center of an adjacent pixel, as shown in Fig. 14.12(b). As the object moves away from the center of the first pixel, its contribution to the first pixel decreases as it nears its edge. It begins to contribute to the pixel it enters only after it has crossed its border, and reaches its maximum contribution when it reaches the center of the new pixel. Thus, even though the black object has constant intensity, the first pixel increases in intensity before the second pixel decreases in intensity.

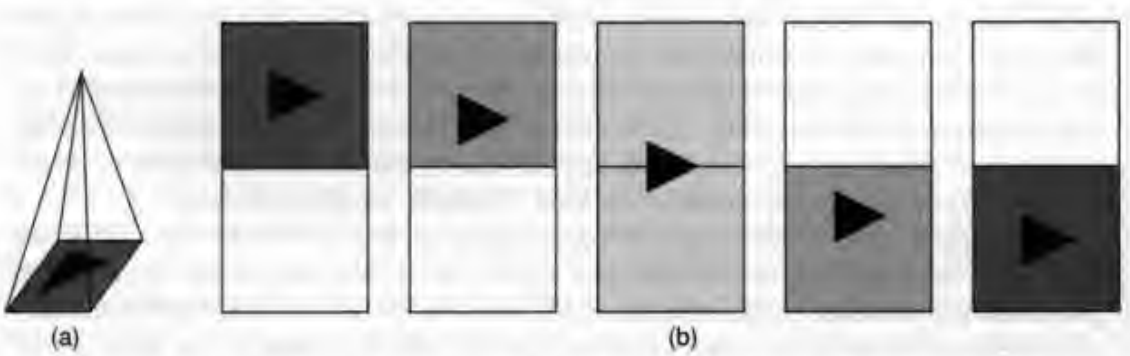


Fig. 14.12 Weighted area sampling. (a) Points in the pixel are weighted differently. (b) Changes in computed intensities as an object moves between pixels.

Samsung/Google, EX1027, p. 634

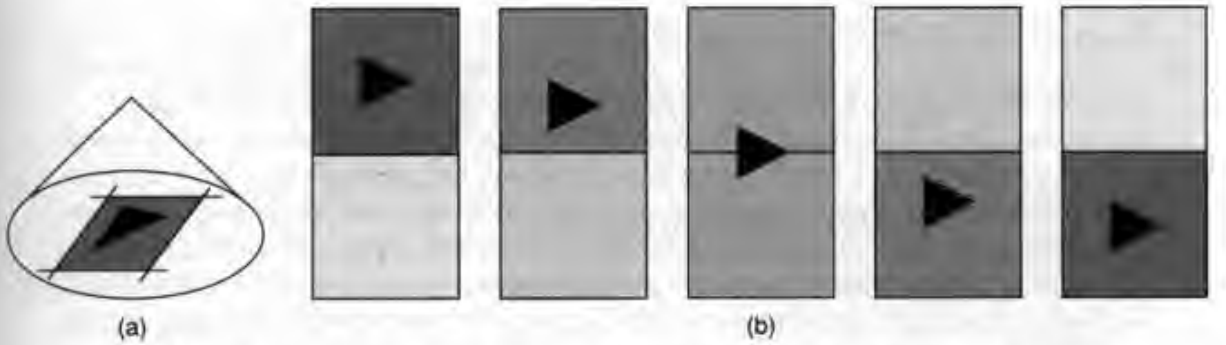


Fig. 14.13 Weighted area sampling with overlap. (a) Typical weighting function. (b) Changes in computed intensities as an object moves between pixels.

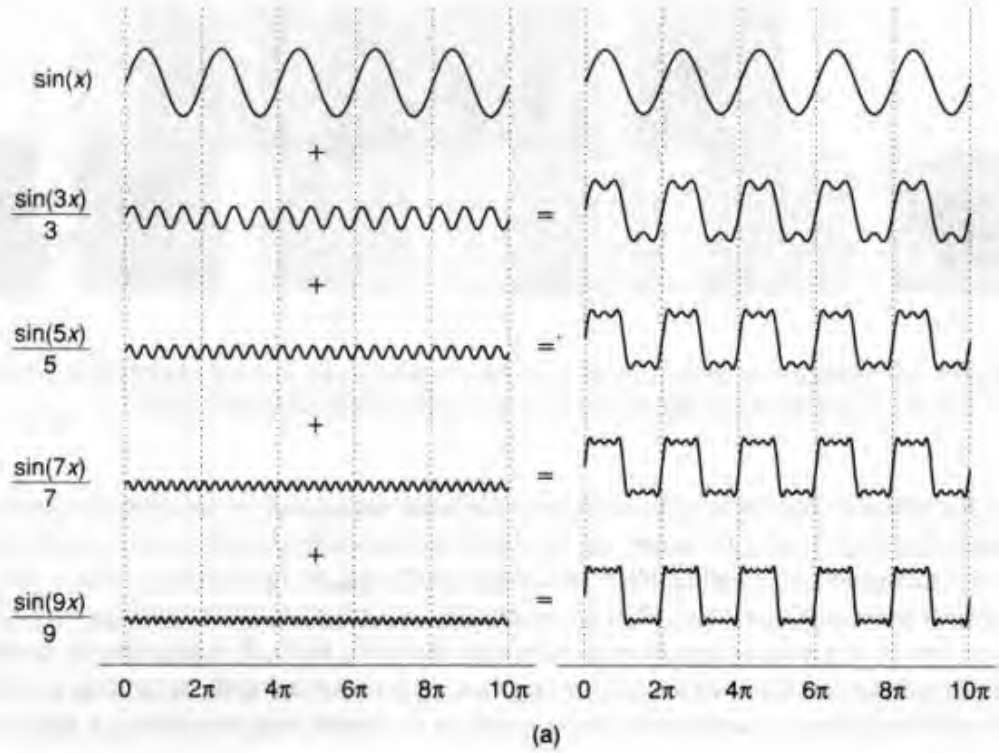
The net effect is that the display changes in intensity depending on the object's position, a change that gives rise to flickering as the object moves across the screen. It is clear that, to correct this problem, we must allow our weighting functions to overlap, so that a point on an object can simultaneously influence more than one pixel, as shown in Fig. 14.13. This figure also uses a radially symmetric weighting function. Here, it is appropriate to turn to sampling theory to discover the underlying reasons for increasing the weighting function's size, and to find out exactly what we need to do to sample and reconstruct a signal.

14.10.3 Sampling Theory

Sampling theory provides an elegant mathematical framework to describe the relationship between a continuous signal and its samples. So far, we have considered signals in the *spatial domain*; that is, we have represented each of them as a plot of amplitude against spatial position. A signal may also be considered in the *frequency domain*; that is, we may represent it as a sum of sine waves, possibly offset from each other (the offset is called *phase shift*), and having different frequencies and amplitudes. Each sine wave represents a component of the signal's *frequency spectrum*. We sum these components in the spatial domain by summing their values at each point in space.

Periodic signals, such as those shown in Fig. 14.14, can each be represented as the sum of phase-shifted sine waves whose frequencies are integral multiples (*harmonics*) of the signal's *fundamental* frequency. But what of nonperiodic signals such as images? Since an image is of finite size, we can define its signal to have a value of zero outside the area of the image. Such a signal, which is nonzero over a finite domain, and, more generally, any signal $f(x)$ that tapers off sufficiently fast (faster than $1/x$ for large values of x) can also be represented as a sum of phase-shifted sine waves. Its frequency spectrum, however, will not consist of integer multiples of some fundamental frequency, but may contain any frequency at all. The original signal cannot be represented as a sum of countably many sine waves, but instead must be represented by an integral over a continuum of frequencies. It is often the case, however, that an image (perhaps padded with surrounding zeros) is treated as one cycle of a periodic signal. This was done in Fig. 14.14(b), which shows the first ten components of Fig. 14.8(d). Each signal in the spatial domain has one representation in the frequency domain, and vice versa. As we shall see later, using two representations for a

Samsung/Google, EX1027, p. 635



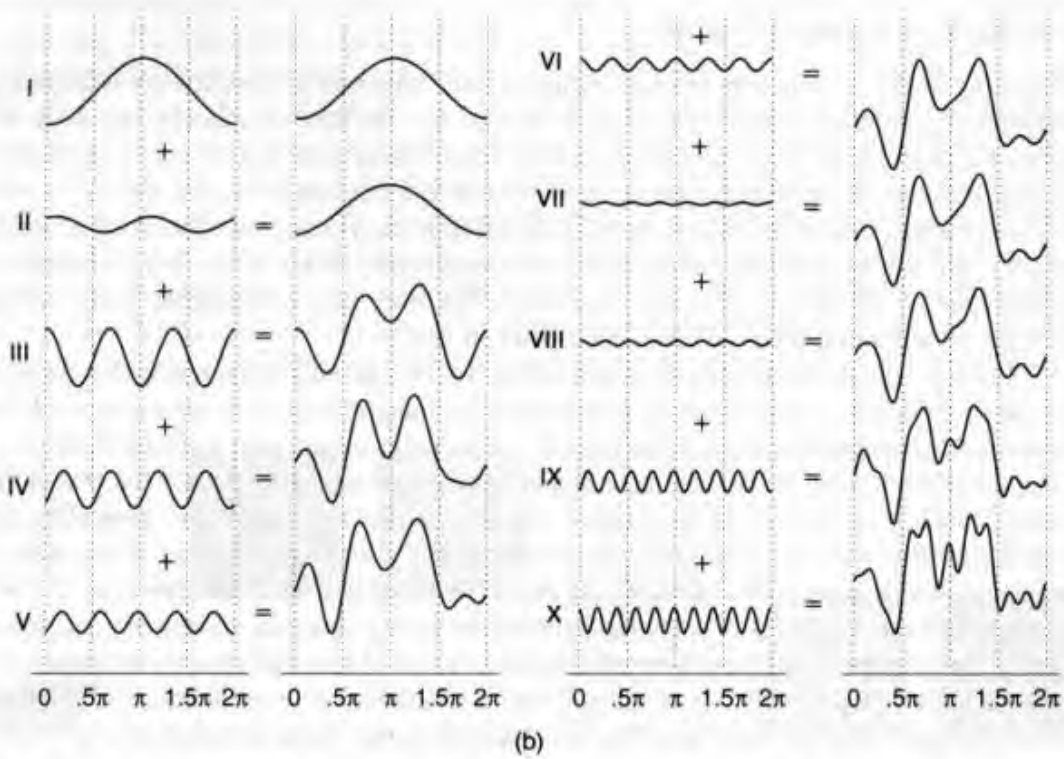


Fig. 14.14 A signal in the spatial domain is the sum of phase-shifted sines. Each component is shown with its effect on the signal shown at its right. (a) Approximation of a square wave. (b) Approximation of Fig. 14.8(d). (Courtesy of George Wolberg, Columbia University.)

Samsung/Google, EX1027, p. 636

signal is advantageous, because some useful operations that are difficult to carry out in one domain are relatively easy to do in the other.

Determining which sine waves must be used to represent a particular signal is the central topic of *Fourier analysis* [GONZ87]. Starting from an original signal, $f(x)$, we can generate a different function, the *Fourier transform* of f, called $F(u)$, whose argument u represents frequency. The value $F(u)$, for each frequency u, tells how much (i.e., the amplitude) of the frequency u appears in the original signal $f(x)$. The function $F(u)$ is therefore also called the *representation of* f (or of *the signal*) *in the frequency domain; f(x)* itself is called the representation of the signal in the spatial domain. The Fourier transform of a continuous, integrable signal $f(x)$ from the spatial domain to the frequency domain is defined by

$$F(u) = \int_{-\infty}^{+\infty} f(x)[\cos 2\pi ux - i\sin 2\pi ux]dx, \tag{14.1}$$

where $i = \sqrt{-1}$ and u represents the frequency of a sine and cosine pair. (Note that this applies only to functions that taper off sufficiently fast.) Recall that the cosine is just the sine, phase shifted by $\pi/2$. Together they can be used to determine the amplitude and phase shift of their frequency's component. For each u, the value of $F(u)$ is therefore a complex number. This is a clever way of encoding the phase shift and amplitude of the frequency u component of the signal: The value $F(u)$ may be written as $R(u) + iI(u)$, where $R(u)$ and $I(u)$ are the real and imaginary parts, respectively. The amplitude (or magnitude) of $F(u)$ is defined by

$$|F(u)| = \sqrt{R^2(u) + I^2(u)}, \tag{14.2}$$

and the phase shift (also known as the *phase angle*) is given by

$$\phi(u) = \tan^{-1}\left[\frac{I(u)}{R(u)}\right]. \tag{14.3}$$

In turn, an integrable signal $F(u)$ may be transformed from the frequency domain to the spatial domain by the *inverse Fourier transform*

$$f(x) = \int_{-\infty}^{+\infty} F(u)[\cos 2\pi ux + i\sin 2\pi ux]du. \tag{14.4}$$

The Fourier transform of a signal is often plotted as magnitude against frequency, ignoring phase angle. Figure 14.15 shows representations of several signals in both domains. In the spatial domain, we label the abscissa with numbered pixel centers; in the frequency domain, we label the abscissa with cycles per pixel (or more precisely, cycles per interval between pixel centers). In each case, the spike at $u = 0$ represents the DC (direct current) component of the spectrum. Substituting $\cos 0 = 1$ and $\sin 0 = 0$ in Eq. (14.1) reveals that this corresponds to integrating $f(x)$. If .5 were subtracted from each value of $f(x)$ in Fig. 14.15 (a) or (b), the magnitude of the signal's DC component would be 0.

Most of the figures in this chapter that show signals and their Fourier transforms were actually computed using discrete versions of Eqs. (14.1) and (14.4) that operate on signals represented by N regularly spaced samples. The *discrete Fourier transform* is

$$F(u) = \sum_{0 \le x \le N-1} f(x)[\cos(2\pi ux/N) - i\sin(2\pi ux/N)], \quad 0 \le u \le N-1, \tag{14.5}$$

Samsung/Google, EX1027, p. 637

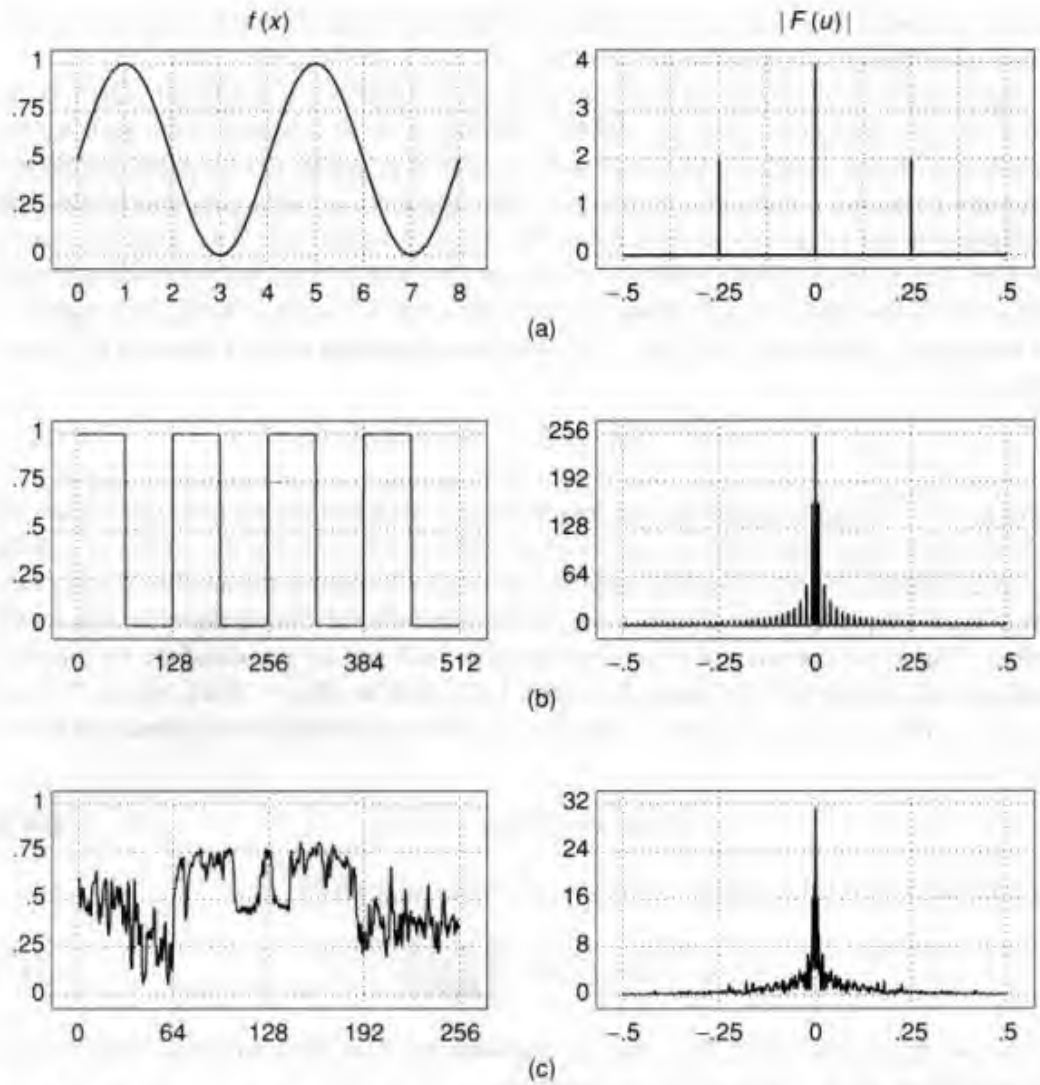


Fig. 14.15 Signals in the spatial and frequency domains. (a) Sine. (b) Square Wave. (c) Mandrill. The DC value in the frequency domain is truncated to make the other values legible and should be 129. (Courtesy of George Wolberg, Columbia Univeristy.)

and the *inverse discrete Fourier transform* is

$$f(x) = \frac{1}{N} \sum_{0 \le u \le N-1} F(u)[\cos(2\pi ux/N) + i\sin(2\pi ux/N)],\ 0 \le x \le N-1. \quad (14.6)$$

By choosing a sufficiently high sampling rate, a good approximation to the behavior of the continuous Fourier transform is obtained for most signals. (The discrete Fourier transform may also be computed more efficiently than Eqs. (14.5) and (14.6) would imply, by using a clever reformulation known as the *fast Fourier transform* [BRIG74].) The discrete Fourier transform always yields a finite spectrum. Note that, if a signal is symmetric about the origin, then $I(u) = 0$. This is true because the contribution of each sine term on one side of the origin is canceled by its equal and opposite contribution on the other side. In this case,

Samsung/Google, EX1027, p. 638

following [BLIN89a], we will plot the signed function $R(u)$, instead of the magnitude $|F(u)|$.

Sampling theory tells us that a signal can be properly reconstructed from its samples if the original signal is sampled at a frequency that is greater than twice f_h, the highest-frequency component in its spectrum. This lower bound on the sampling rate is known as the *Nyquist rate*. Although we do not give the formal proof of the adequacy of sampling above the Nyquist rate, we can provide an informal justification. Consider one cycle of a signal whose highest-frequency component is at frequency f_h. This component is a sine wave with f_h maxima and f_h minima, as shown in Fig. 14.16. Therefore, at least $2f_h$ samples are required to capture the overall shape of the signal's highest-frequency component. Note that exactly $2f_h$ samples is, in fact, a special case that succeeds only if the samples are taken precisely at the maxima and minima (Fig. 14.16a). If they are taken anywhere else, then the amplitude will not be represented correctly (Fig. 14.16b) and may even be determined to be zero if the samples are taken at the zero crossings (Fig. 14.16c). If we sample below the Nyquist rate, the samples we obtain may be identical to what would have been obtained from sampling a lower-frequency signal, as demonstrated in Fig. 14.17. This phenomenon of high frequencies masquerading as low frequencies in the reconstructed signal is known as *aliasing*: The high-frequency components appear as though they were actually lower-frequency components. Another example of aliasing is demonstrated in Fig. 14.18. Figure 14.18(a) shows an image and a plot of its intensity across a horizontal line, representing a set of intensity fluctuations that increase in spatial frequency from left to right. The image in Fig. 14.18(b) was created by selecting every 8th pixel from each line of Fig. 14.18(a) and replicating it eight times. It shows aliasing as the bands increase in spatial frequency.

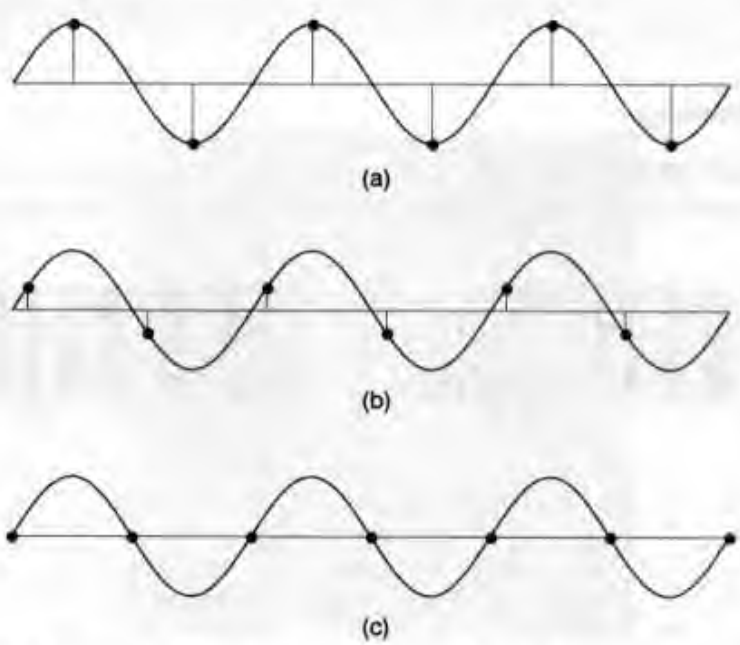


Fig. 14.16 Sampling at the Nyquist rate (a) at peaks, (b) between peaks, (c) at zero crossings. (Courtesy of George Wolberg, Columbia University.)

Samsung/Google, EX1027, p. 639

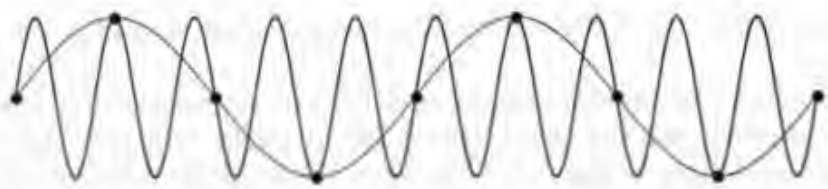


Fig. 14.17 Sampling below the Nyquist rate. (Courtesy of George Wolberg, Columbia University.)

A signal's shape is determined by its frequency spectrum. The sharper and more angular a waveform is, the richer it is in high-frequency components; signals with discontinuities have an infinite frequency spectrum. Figure 14.10 reveals the sharp edges of the objects' projections that our algorithms attempt to represent. This signal has an infinite frequency spectrum, since the image intensity changes discontinuously at object boundaries. Therefore, the signal cannot be represented properly with a finite number of samples. Computer graphics images thus exhibit two major kinds of aliasing. First, "jaggies" along edges are caused by discontinuities at the projected edges of objects: a point sample either does or does not lie in an object's projection. Even the presence of a single such edge in an environment's projection means that the projection has an infinite frequency spectrum. The frequency spectrum tapers off quite rapidly, however, like those of Fig. 14.15(b) and (c). Second, textures and objects seen in perspective may cause arbitrarily many discontinuities and fluctuations in the environment's projection, making it possible for objects whose projections are too small and too close together to be alternately missed and sampled, as in the right hand side of Fig. 14.18(b). The high frequency components representing the frequency at which these projections cross a scan line may have high amplitude (e.g., alternating black and white checkerboard squares). This often affects picture quality more seriously than jaggies do.

14.10.4 Filtering

There is a partial solution to the problems discussed in the previous section. If we could create a new signal by removing the offending high frequencies from the original signal,

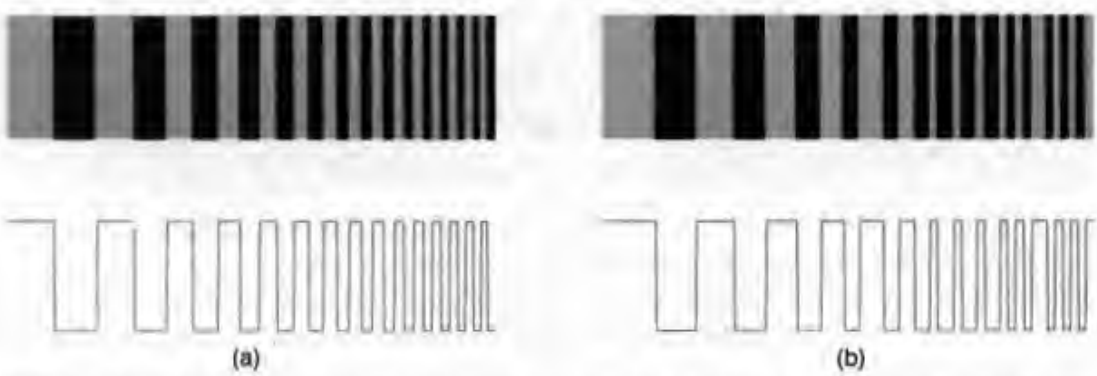


Fig. 14.18 Aliasing. (a) Image and intensity plot of a scan line. (b) Sampled image and intensity plot of a scan line. (Courtesy of George Wolberg, Columbia University.)

Samsung/Google, EX1027, p. 640

then the new signal could be reconstructed properly from a finite number of samples. The more high frequencies we remove, the lower the sampling frequency needed, but the less the signal resembles the original. This process is known as *bandwidth limiting* or *band limiting* the signal. It is also known as *low-pass filtering*, since filtering a signal changes its frequency spectrum; in this case, high frequencies are filtered out and only low frequencies are allowed to pass. Low-pass filtering causes blurring in the spatial domain, since fine visual detail is captured in the high frequencies that are attenuated by low-pass filtering, as shown in Fig. 14.19. We shall revise the pipeline of Fig. 14.9 to include an optional filter, as shown in Fig. 14.20.

A perfect low-pass filter completely suppresses all frequency components above some specified cut-off point, and lets those below the cut-off point pass untouched. We can easily do this filtering in the frequency domain by multiplying the signal's spectrum by a *pulse function*, as shown in Fig. 14.21. We can multiply two signals by taking their product at each point along the paired signals. The pulse function

$$S(u) = \begin{cases} 1, \text{ when } -k \leq u \leq k, \\ 0, \text{ elsewhere.} \end{cases} \tag{14.7}$$

cuts off all components of frequency higher than k. Therefore, if we were to low-pass filter the signal so as to remove all variation, we would be left with only its DC value.

So far, it would seem that a recipe for low-pass filtering a signal in the spatial domain would involve transforming the signal into the frequency domain, multiplying it by an appropriate pulse function, and then transforming the product back into the spatial domain. Some important relationships between signals in the two domains, however, make this procedure unnecessary. It can be shown that multiplying two Fourier transforms in the frequency domain corresponds exactly to performing an operation called *convolution* on their inverse Fourier transforms in the spatial domain. The *convolution* of two signals $f(x)$



Fig. 14.19 Figure 14.8(b) after low-pass filtering. (Courtesy of George Wolberg, Columbia University.)

Samsung/Google, EX1027, p. 641

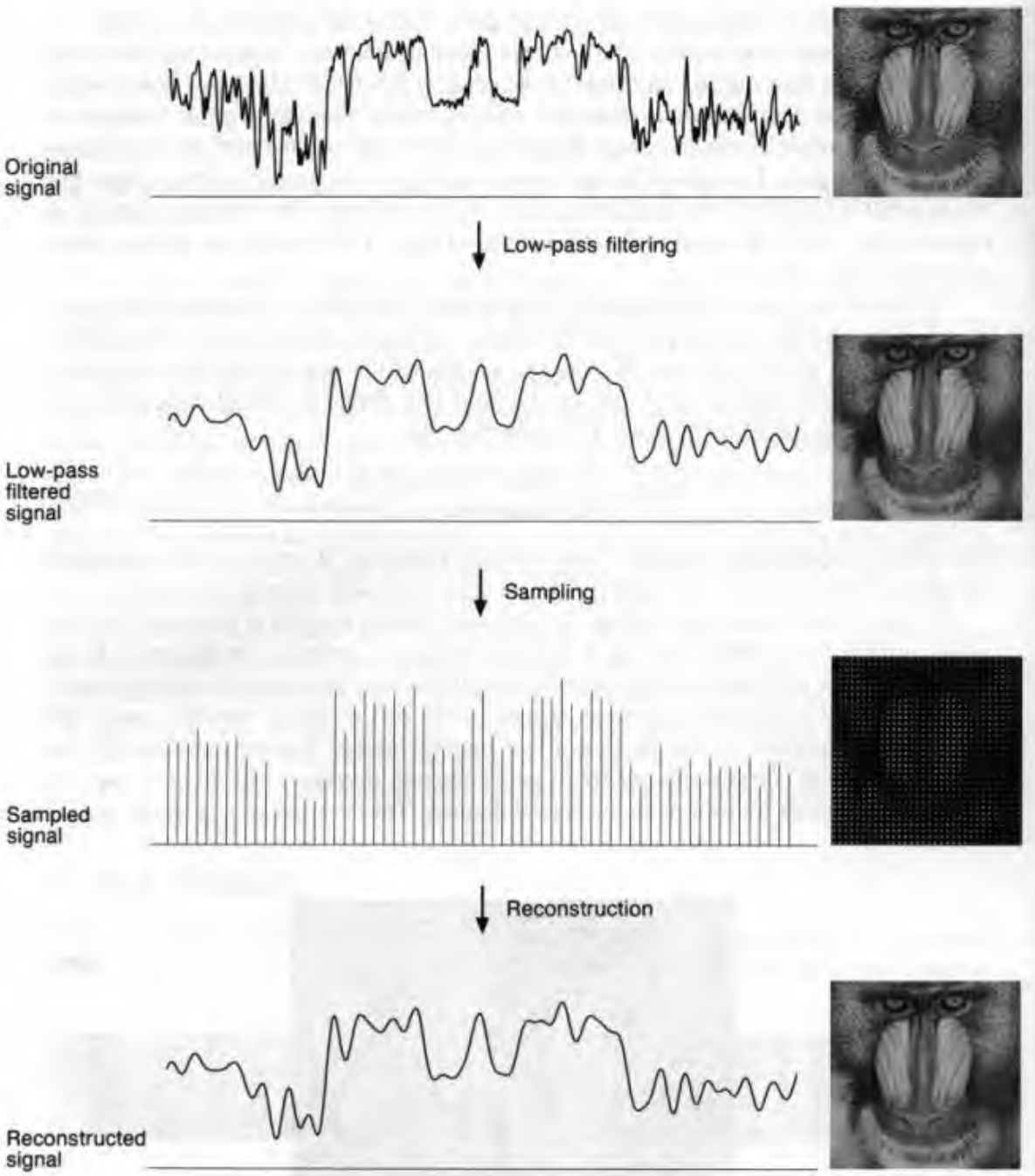


Fig. 14.20 The sampling pipeline with filtering. (Courtesy of George Wolberg, Columbia University.)

and $g(x)$, written as $f(x) * g(x)$, is a new signal $h(x)$ defined as follows. The value of $h(x)$ at each point is the integral of the product of $f(x)$ with the filter function $g(x)$ flipped about its vertical axis and shifted such that its origin is at that point. This corresponds to taking a weighted average of the neighborhood around each point of the signal $f(x)$—weighted by a flipped copy of filter $g(x)$ positioned at the point—and using it for the value of $h(x)$ at the

Samsung/Google, EX1027, p. 642

point. The size of the neighborhood is determined by the size of the domain over which the filter is nonzero. This is known as the filter's *support*, and a filter that is nonzero over a finite domain is said to have *finite support*. We use τ as a dummy variable of integration when defining the convolution. Thus,

$$h(x) = f(x) * g(x) \triangleq \int_{-\infty}^{+\infty} f(\tau)g(x - \tau)d\tau. \tag{14.8}$$

Conversely, convolving two Fourier transforms in the frequency domain corresponds exactly to multiplying their inverse Fourier transforms in the spatial domain. The filter function is often called the *convolution kernel* or *filter kernel*.

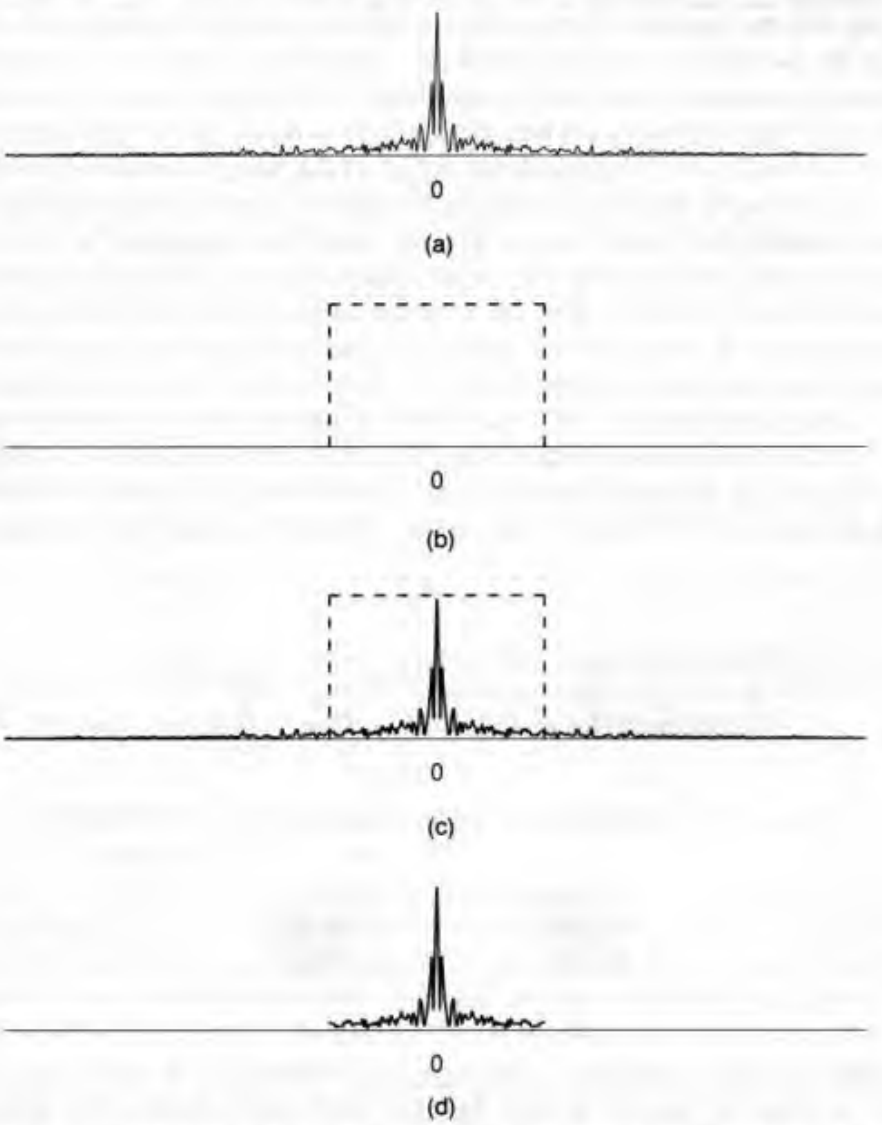


Fig. 14.21 Low-pass filtering in the frequency domain. (a) Original spectrum. (b) Low-pass filter. (c) Spectrum with filter. (d) Filtered spectrum. (Courtesy of George Wolberg, Columbia University.)

Samsung/Google, EX1027, p. 643

Convolution can be illustrated graphically. We will convolve the function $f(x) = 1, 0 \leq x \leq 1$, with the filter kernel $g(x) = c, 0 \leq x \leq 1$, shown in Figs. 14.22(a) and (b). By using functions of τ, we can vary x to move the filter relative to the signal being filtered. To create the function $g(x - \tau)$, we first flip $g(\tau)$ about the origin to yield $g(-\tau)$, and then offset it by x to form $g(x - \tau)$, as depicted in Figs. 14.22(c) and (d). The integral, with respect to τ, of the product $f(\tau)g(x - \tau)$, which is the area of the shaded portions of the figures, is 0 for $-\infty \leq x < 0$, xc for $0 \leq x \leq 1$ (Fig. 14.22e), $(2 - x)c$ for $1 \leq x \leq 2$ (Fig. 14.22f), and 0 for $2 < x \leq \infty$. The convolution $f(x) * g(x)$ is illustrated in Fig. 14.22(g). Note how convolution with this kernel smooths the discontinuities of $f(x)$ while it widens the area over which $f(x)$ is nonzero.

Multiplying by a pulse function in the frequency domain has the same effect as convolving with the signal that corresponds to the pulse in the spatial domain. This signal is known as the *sinc* function, which is defined as $\sin(\pi x)/\pi x$. Figure 14.23 shows the sinc function and an example of the result of convolving it with another signal. Convolving with a sinc function therefore low-pass filters the signal. How do we choose the height and width of the sinc used in Fig. 14.23(c)? As shown in Fig. 14.24, there is a relationship (that we do not prove) between the height and width of the perfect low-pass filter in the spatial and frequency domains. In Fig. 14.24(a), if W is the cutoff frequency and A is the amplitude, then it must be the case that $A/2W = 1$ for all frequencies up to the cutoff frequency to be passed unattenuated. Therefore, $A = 2W$. Both the amplitude and width of the sinc in Fig. 14.24(a) vary with W. When $W = .5$ cycles per pixel (the highest frequency that can be represented when sampling once per pixel), $A = 1$ and the sinc has zero crossings at pixel centers. As the cutoff frequency W is made lower or higher, the sinc becomes shorter and broader, or taller and narrower, respectively. This makes sense because we would like the integral of a filter in the spatial domain to be 1, a necessary restriction if the filter is to maintain the gray level (DC value) of the image, neither brightening nor dimming it. (We

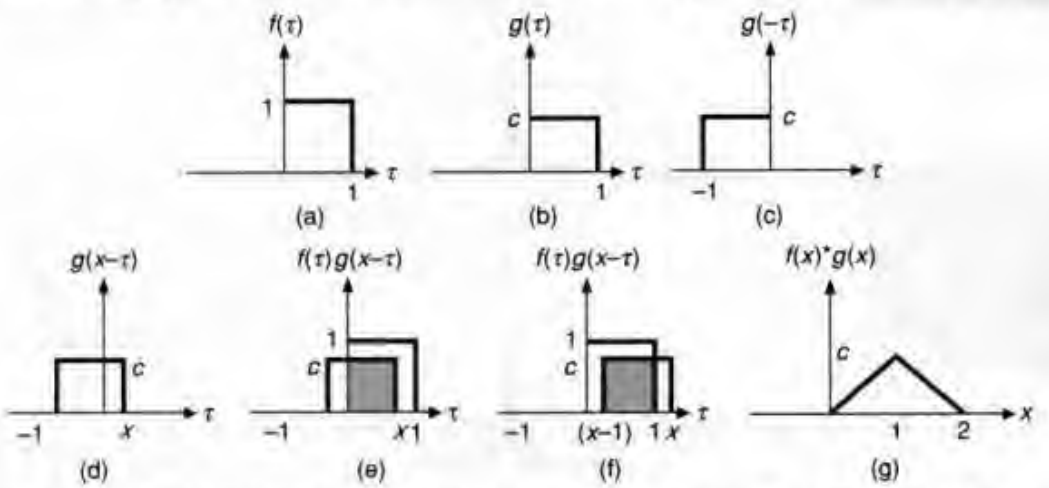


Fig. 14.22 Graphical convolution. (a) Function $f(\tau) = 1, 0 \leq \tau \leq 1$. (b) Filter kernel $g(\tau) = c, 0 \leq \tau \leq 1$. (c) $g(-\tau)$. (d) $g(x - \tau)$. (e) $\int_{-\infty}^{+\infty} f(\tau)g(x - \tau)d\tau = xc, 0 \leq x \leq 1$. (f) $\int_{-\infty}^{+\infty} f(\tau)g(x - \tau)d\tau = (2 - x)c, 1 \leq x \leq 2$. (g) $f(x) * g(x)$.

Samsung/Google, EX1027, p. 644

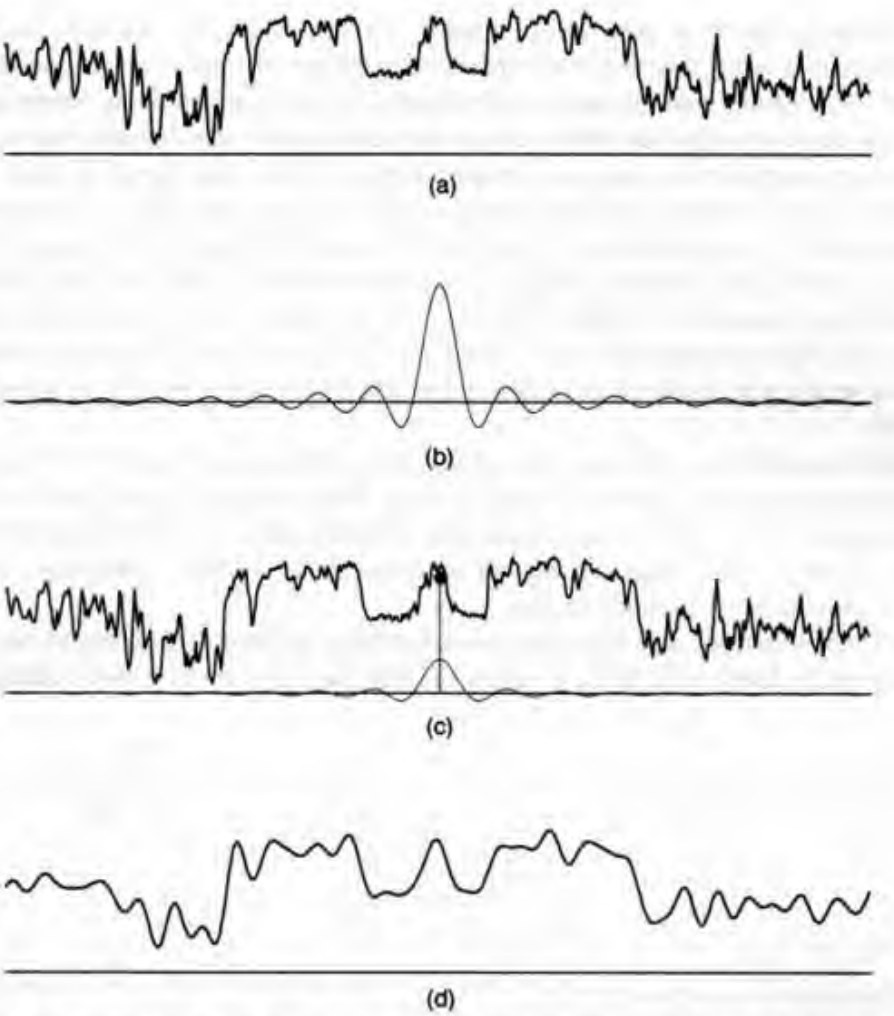


Fig. 14.23 Low-pass filtering in the spatial domain. (a) Original signal. (b) Sinc filter. (c) Signal with filter, with value of filtered signal shown as a black dot (•) at filter's origin. (d) Filtered signal. (Courtesy of George Wolberg, Columbia University.)

can see that this is true by considering the convolution of a filter with a signal that has the same value c at each point.)

The sinc function has the unfortunate property that it is nonzero on points arbitrarily far from the origin (i.e., it has *infinite support* since it is infinitely wide). If we truncate the sinc function, by multiplying it by a pulse function, we can restrict the support, as shown in Fig. 14.24(b). This is a special case of a *windowed* sinc function that has been restricted to a finite window. We might reason that we are throwing away only those parts of the filter where the value is very small anyhow, so it should not influence the result too much. Unfortunately, the truncated version of the filter has a Fourier transform that suffers from *ringing* (also called the *Gibbs phenomenon*): A truncated sinc in the spatial domain no

Samsung/Google, EX1027, p. 645

longer corresponds to a pure pulse function in the frequency domain, but instead corresponds to a pulse function with ripples near the cutoff frequency, as shown in Fig. 14.24(b). This causes some frequency components to pass that should be suppressed, and both attenuates and amplifies others around the cutoff point; the domain over which this effect occurs decreases in size as a greater portion of the sinc signal is used, but the amplitude of the ringing does not decrease as long as the sinc is truncated. The approximation to a square wave in Fig.14.14(a) exhibits ringing in the spatial domain, which appears as little intensity "ripples" at discontinuities. A truncated sinc is obtained by multiplying the sinc by a pulse function. An alternative is to use a windowed sinc function that has been multiplied by a shape that, unlike the pulse, is not discontinuous, which allows the sinc to fall off smoothly. Blinn [BLIN89b] describes the derivation of one such filter.

One final problem is that the sinc, along with windowed filters derived from it, has parts that dip below zero, known as *negative lobes*. When a signal is convolved with a filter that has negative lobes, the resulting signal may itself dip below zero. If the signal represents intensity values, these values correspond to unrealizable negative intensities, and must therefore ultimately be clamped to zero.

Although windowed sinc functions are useful, they are relatively expensive because the window must be fairly wide; thus, a variety of other functions is often used instead. Filters

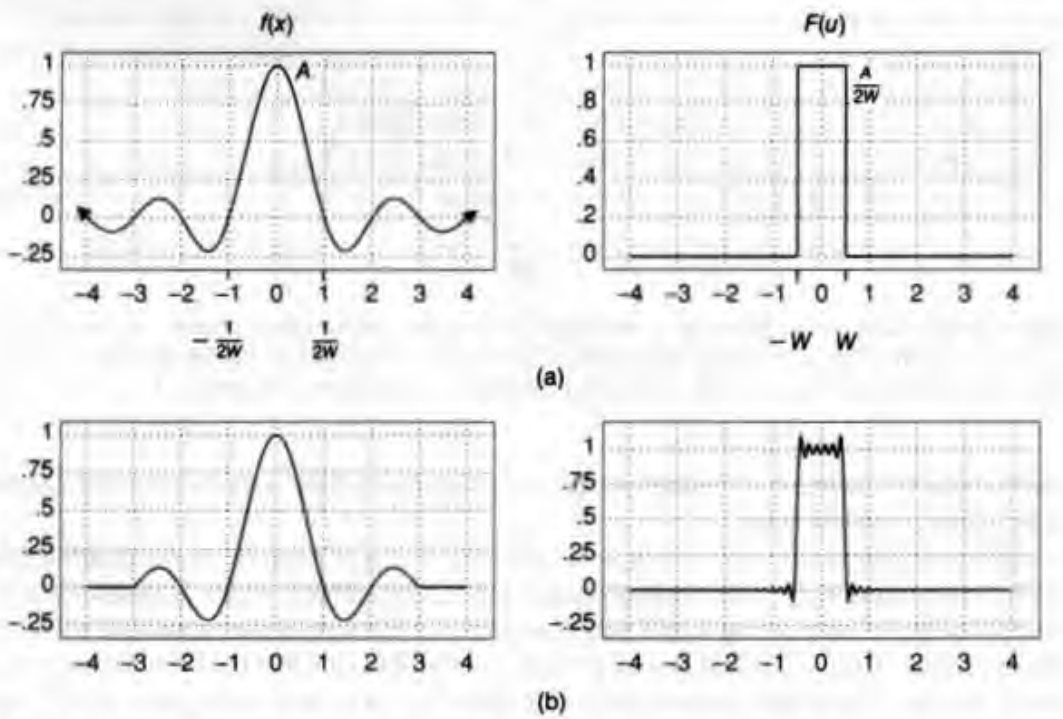


Fig. 14.24 (a) Sinc in spatial domain corresponds to pulse in frequency domain. (b) Truncated sinc in spatial domain corresponds to ringing pulse in frequency domain. (Courtesy of George Wolberg, Columbia University.)

Samsung/Google, EX1027, p. 646

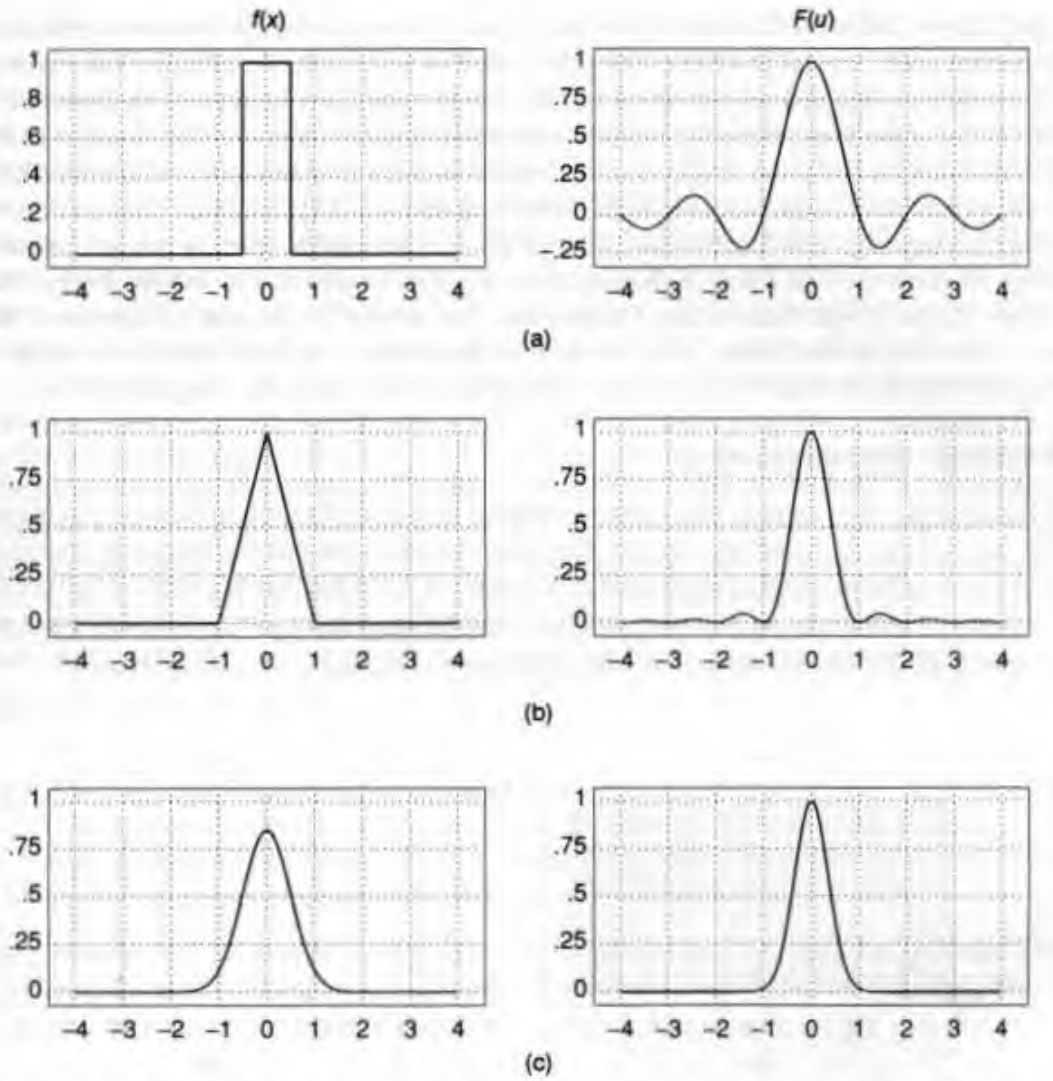


Fig. 14.25 Filters in spatial and frequency domains. (a) Pulse—sinc. (b) Triangle—sinc^2. (c) Gaussian—Gaussian. (Courtesy of George Wolberg, Columbia University.)

with finite support are known as finite impulse-response (FIR) filters, in contrast to the untruncated sinc filter, which is an infinite impulse-response (IIR) filter. Figure 14.25 shows several popular filters in both spatial and frequency domains.

We have now reduced the sampling problem to one of convolving the signal with a suitable filter and then sampling the filtered signal. Notice, however, that if our only use of the filtered signal is to sample it, then the work done filtering the signal anywhere but at the sample points is wasted. If we know in advance exactly where the samples will be taken, we need only to evaluate the convolution integral (Eq. 14.8) at each sample point to determine the sample's value. This is precisely how we perform the weighting operation in using area

Samsung/Google, EX1027, p. 647

sampling to determine the intensity of each pixel. The weighting distribution constructed over each pixel's center is a filter. The pulse function with which we convolve the signal in performing unweighted area sampling is often called a *box filter*, because of its appearance. Just as the pulse function in the frequency domain corresponds to the sinc function in the spatial domain, the pulse function in the spatial domain (the box filter's 1D equivalent) corresponds to the sinc function in the frequency domain (Fig. 14.25a). This correspondence underscores how badly a box filter or pulse filter approximates a perfect low-pass filter. Multiplying with a sinc in the frequency domain not only fails to cut off sharply, but passes infinitely high frequencies. Furthermore, the pulse filter attenuates frequencies that are within the desired range, since its Fourier transform—the sinc function—begins to trail off before the ideal low-pass filter. Therefore, it also blurs the image excessively.

14.10.5 Reconstruction

At this point, let us assume that we have sampled the signal $f(x)$ at a frequency f_s to obtain the sampled signal, which we call $\hat{f}(x)$. Sampling theory shows that the frequency spectrum of $\hat{f}(x)$ looks like that of $f(x)$, replicated at multiples of f_s. To see that this relationship holds, we note that sampling a signal corresponds to multiplying it in the spatial domain by a *comb* function, so named because of its appearance, as shown in Fig. 14.26(a). The comb

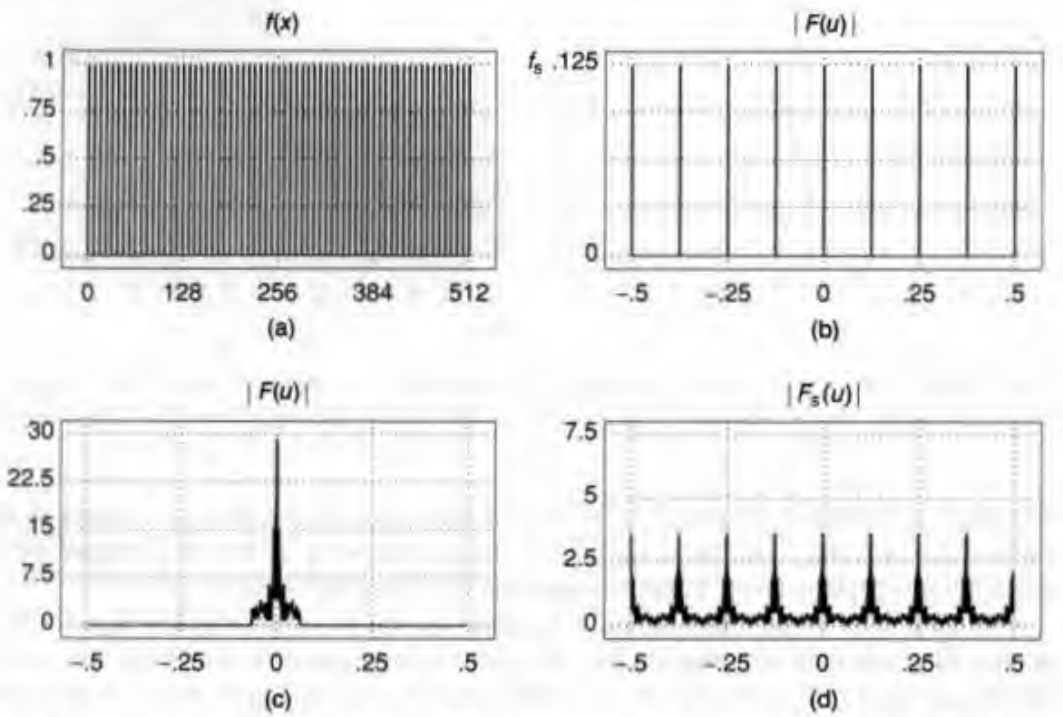


Fig. 14.26 (a) Comb function and (b) its Fourier transform. Convolving the comb's Fourier transform with (c) a signal's Fourier transform in the frequency domain yields (d) the replicated spectrum of the sampled signal. (Courtesy of George Wolberg, Columbia University.)

Samsung/Google, EX1027, p. 648

function has a value of 0 everywhere, except at regular intervals, corresponding to the sample points, where its value is 1. The (discrete) Fourier transform of a comb turns out to be just another comb with teeth at multiples of f_s (Fig. 14.26b). The height of the teeth in the comb's Fourier transform is f_s in cycles/pixel. Since multiplication in the spatial domain corresponds to convolution in the frequency domain, we obtain the Fourier transform of the sampled signal by convolving the Fourier transforms of the comb function and the original signal (Fig. 14.26c). By inspection, the result is the replicated spectrum (Fig. 14.26d). Try performing graphical convolution with the comb to verify this, but note that $F(u)$, not $|F(u)|$ must actually be used. A sufficiently high f_s yields spectra that are replicated far apart from each other. In the limiting case, as f_s approaches infinity, a single spectrum results.

Recall that *reconstruction* is recreation of the original signal from its samples. The result of sampling a signal (Fig. 14.27a) at a finite sampling frequency is a signal with an infinite frequency spectrum (Fig. 14.27b). If once again we deal with the signal in the frequency domain, the familiar operation of multiplying a signal by a pulse function can be used to eliminate these replicated spectra (Fig. 14.27c), leaving only a single copy of the original spectrum (Fig. 14.27d). Thus, we can reconstruct the signal from its samples by multiplying the Fourier transform of the samples by a pulse function in the frequency domain or by convolving the samples with a sinc with $A = 1$ in the spatial domain.

To make the Fourier transforms of signals and filters easier to see in Figs. 14.27–29, we have taken several liberties:

- The DC value of the Fourier transform in part (a) of each figure has been truncated. This corresponds to a signal in the spatial domain with the same shape as shown, but with a negative DC offset. (Such a signal cannot be displayed as an image without further processing, because it contains negative intensity values.)
- Filters in the frequency domain have not been drawn with the correct magnitude. Their heights should be 1 in Fig. 14.27 and 2 in Figs. 14.28–29 to restore the single copy of the spectrum to its original magnitude.

Figure 14.27(e) and (f) show the result of reconstructing the samples with a triangle filter (also known as a Bartlett filter). Convolving with this filter is equivalent to linearly interpolating the samples.

If the sampling frequency is too low, the replicated copies of the frequency spectra overlap, as in Fig. 14.28. In this case, the reconstruction process will fail to remove those parts of the replicated spectra that overlapped the original signal's spectrum. High-frequency components from the replicated spectra are mixed in with low-frequency components from the original spectrum, and therefore are treated like low frequencies during the reconstruction process. Note how an inadequate sampling rate causes aliasing by making a higher frequency appear identical to a lower one before and after reconstruction. There are two ways to resolve this problem. We may choose to sample at a high enough frequency, an approach that is sufficient only if the signal does not have an infinite spectrum. Alternatively, we may filter the signal before sampling to remove all components above $f_s/2$, as shown in Fig. 14.29.

Samsung/Google, EX1027, p. 649

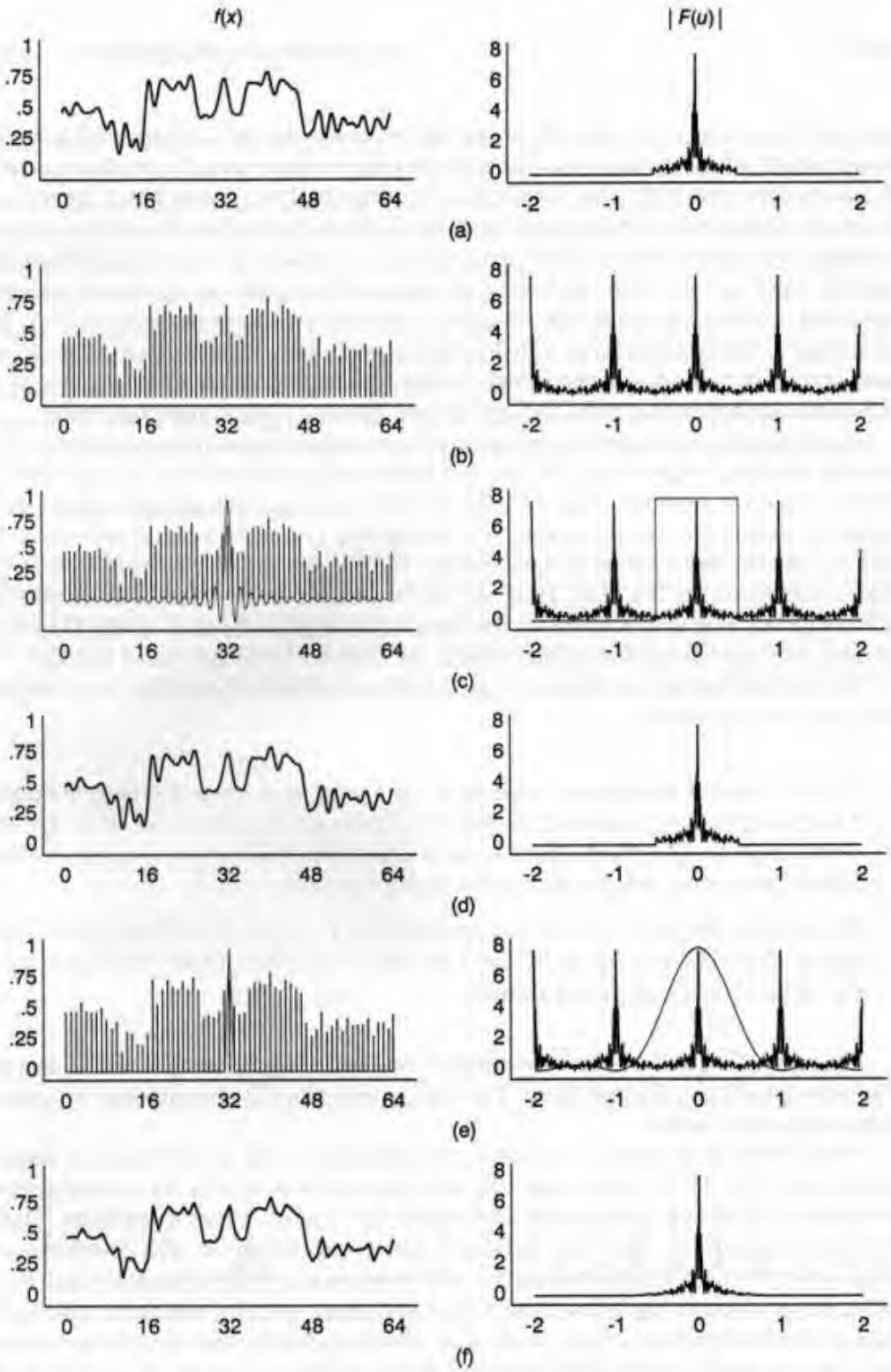


Fig. 14.27 Sampling and reconstruction: Adequate sampling rate. (a) Original signal. (b) Sampled signal. (c) Sampled signal ready to be reconstructed with sinc. (d) Signal reconstructed with sinc. (e) Sampled signal ready to be reconstructed with triangle. (f) Signal reconstructed with triangle. (Courtesy of George Wolberg, Columbia University.)

Samsung/Google, EX1027, p. 650

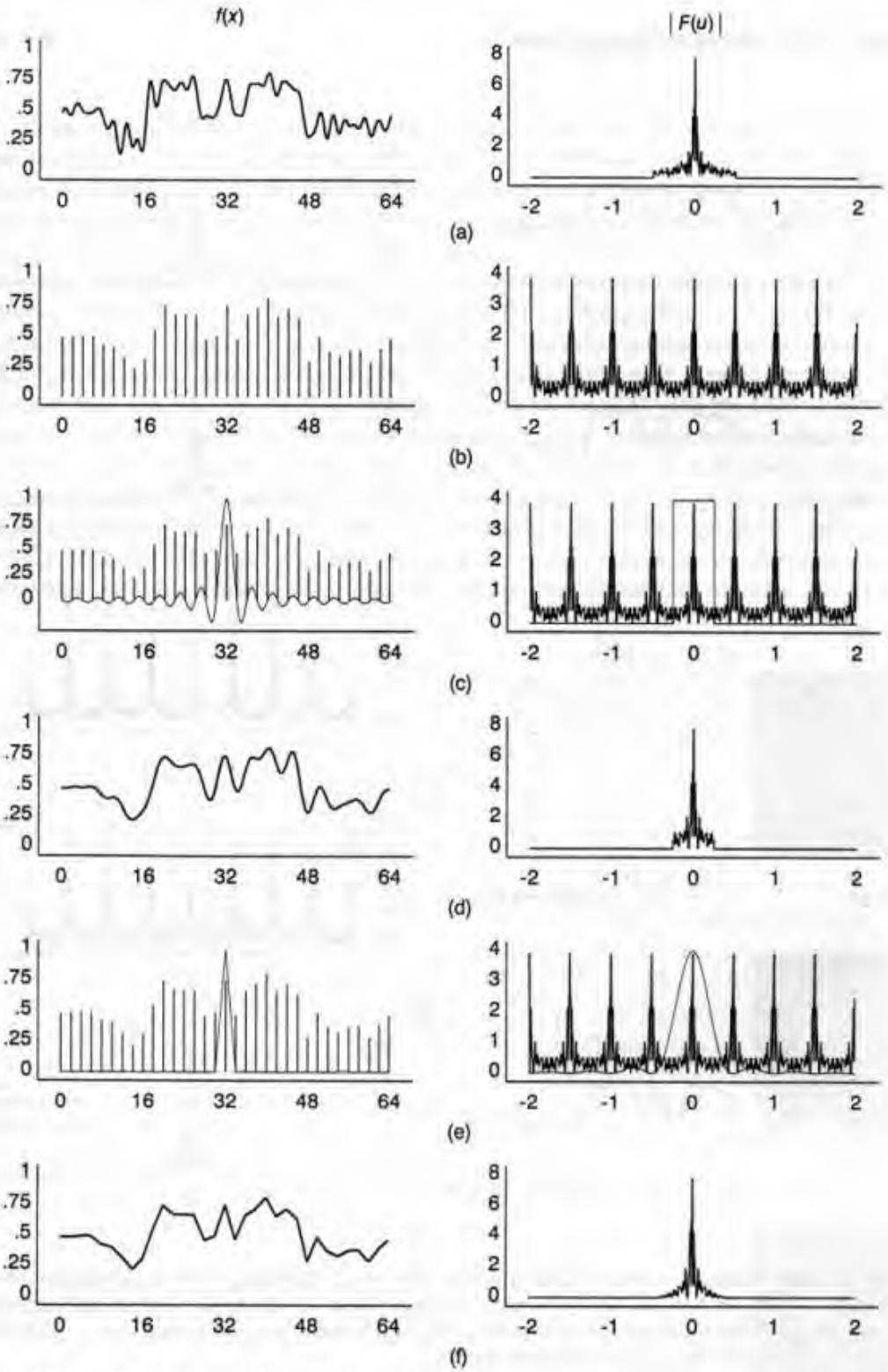


Fig. 14.28 Sampling and reconstruction: Inadequate sampling rate. (a) Original signal. (b) Sampled signal. (c) Sampled signal ready to be reconstructed with sinc. (d) Signal reconstructed with sinc. (e) Sampled signal ready to be reconstructed with triangle. (f) Signal reconstructed with triangle. (Courtesy of George Wolberg, Columbia University.)

Samsung/Google, EX1027, p. 651

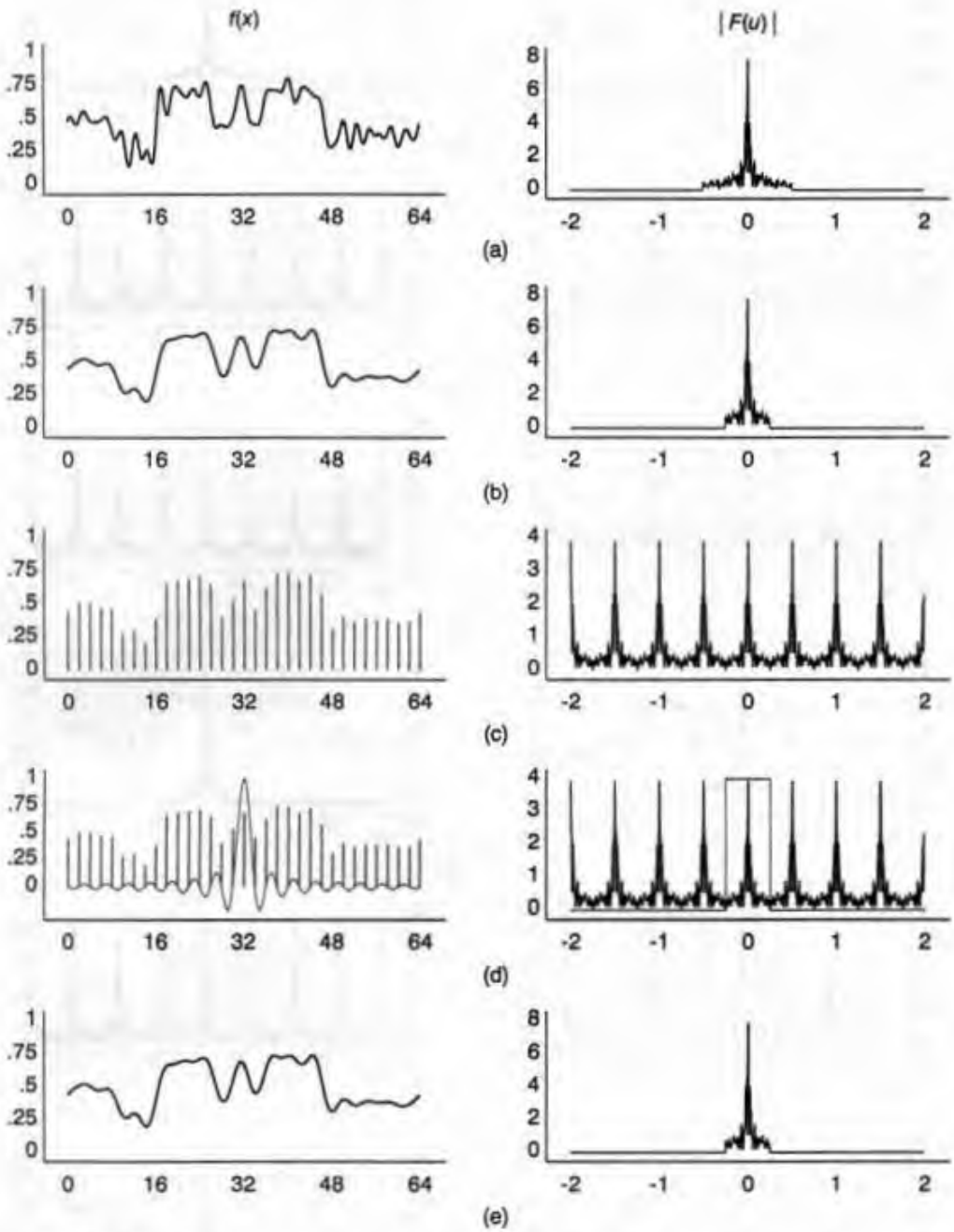


Fig. 14.29 Filtering, sampling, and reconstruction: Sampling rate is adequate after filtering. (a) Original signal. (b) Low-pass filtered signal. (c) Sampled signal. (d) Sampled signal ready to be reconstructed with sinc. (e) Signal reconstructed with sinc. (Courtesy of George Wolberg, Columbia University.)

Samsung/Google, EX1027, p. 652

What happens if reconstruction is done by convolving with some signal other than a sinc? Samples in the frame buffer are translated into a continuous video signal, by a process known as *sample and hold*: for the signal to be reconstructed, the value of each successive sample is simply held for the duration of a pixel. This process corresponds to convolving the samples with a 1-pixel-wide box filter, as shown in Fig. 14.30, and gives rise to our common conception of a pixel as one of a set of square boxes tiling the display. The resulting signal has sharp transitions between pixels, corresponding to high-frequency components that are not represented by the samples. This effect is often known as *rastering*. Although the video hardware nominally samples and holds each pixel's intensity, the circuitry that generates the analog voltages applied to the CRT and the CRT itself are generally not fast enough to produce discontinuous jumps in intensity between pixels. The Gaussian distribution of the CRT spot also reduces this problem. Thus, the sampled signal is reconstructed by the equivalent of convolution with a box filter, followed by convolution with a Gaussian. Rastering is especially easy to see, however, when pixel-replicating zoom is used in raster CRT displays, increasing the amount of screen space allocated to an individual pixel. Rastering is also more evident in printer, digital film recorder, and LCD

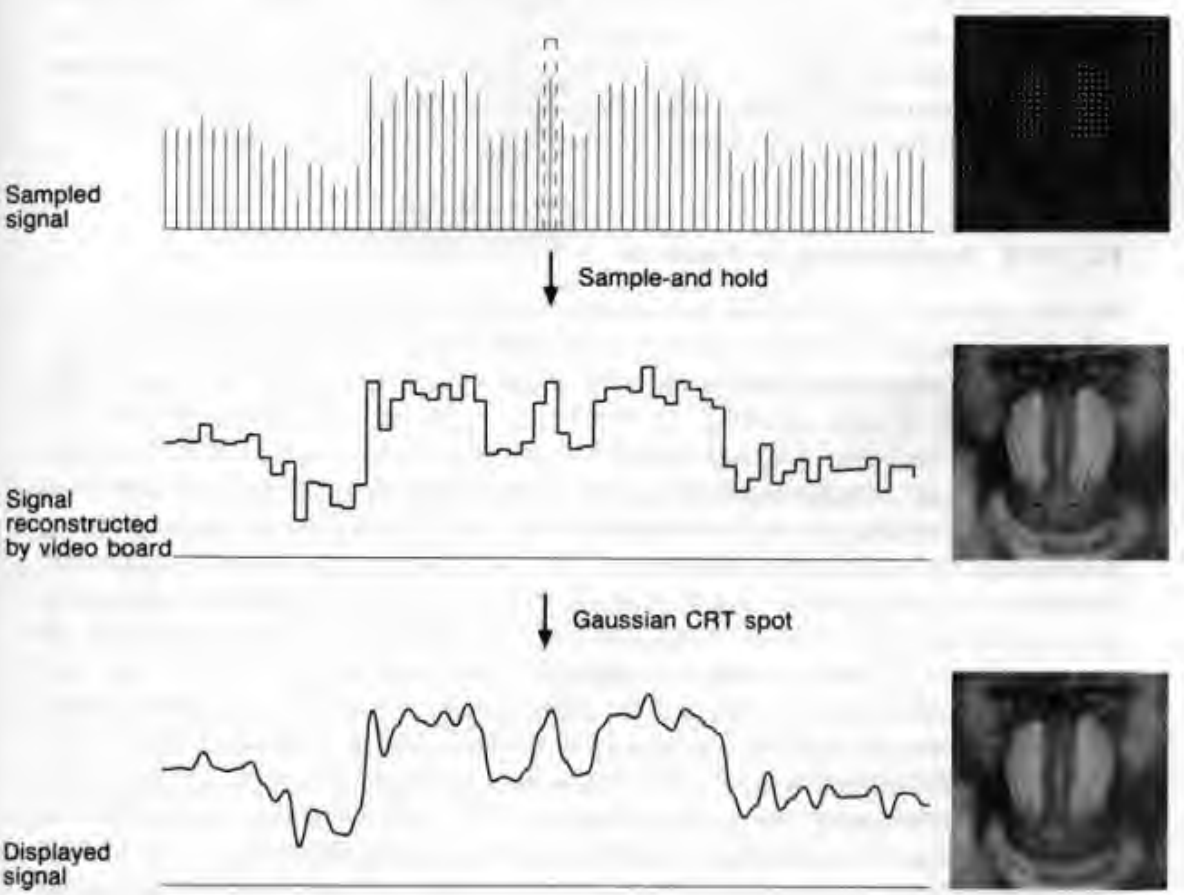


Fig. 14.30 Reconstruction by sample and hold and Gaussian CRT spot. (Courtesy of George Wolberg, Columbia University.)

Samsung/Google, EX1027, p. 653

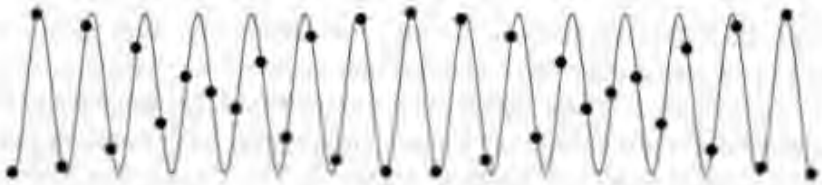


Fig. 14.31 A signal sampled at slightly over the Nyquist rate. (Courtesy of George Wolberg, Columbia University.)

technologies, in which pixel-to-pixel transitions are much sharper and produce relatively hard-edged square pixels of constant intensity.

We noted earlier that a signal must be sampled at a frequency greater than $2f_h$ to make perfect reconstruction possible. If the filter used to reconstruct the samples is not an untruncated sinc, as is always the case when displaying an image, then the sampling frequency must be even higher! Consider, for example, a sampling frequency slightly greater than $2f_h$. The resulting samples trace out the original signal modulated by (multiplied by) a low-frequency sine wave, as shown in Fig. 14.31. The low-frequency amplitude modulation remains, compounded by rastering, if the signal is reconstructed with a 1-pixel-wide box filter. If convolution is performed with an untruncated sinc, however, the original signal is recovered. The inevitable use of nonideal filters before and after sampling therefore mandates higher sampling rates. Mitchell and Netravali [MITC88] discuss some of the problems involved in doing a good job of reconstruction.

14.10.6 Antialiasing in Practice

We have seen that image synthesis involves sampling and reconstruction, noting that there is little that we can do (in software) about the reconstruction approach employed in hardware. Rendering algorithms that perform antialiasing use either point sampling or an analytic approach, such as area sampling. In either case, a single value must ultimately be determined for each pixel. Catmull's algorithm, discussed in Section 15.7.3, is an example of an analytic (and expensive) approach using unweighted area sampling. It corresponds to filtering at object precision before calculating the value of each pixel's sample. Filtering before sampling is often called *prefiltering*. When supersampling is used, the samples are combined according to a filter weighting in a discrete version of the continuous convolution and sampling that we discussed earlier. The filter is represented by an array of values. As shown in Fig. 14.32, the filter array is positioned over the array of supersampled values and the sum of the products of values in corresponding positions determines a single sample taken at the center of the filter. The filter array is then moved to the position at which the next sample will be taken, with the number of samples corresponding to the pixel resolution of the filtered image being created. This approach is often called *postfiltering*, since filtering is performed after point sampling. It actually corresponds to reconstructing the signal from its samples only at selected points in space. These reconstructed values are then used as new samples. Supersampling thus performs a discrete approximation to weighted area sampling.

Samsung/Google, EX1027, p. 654

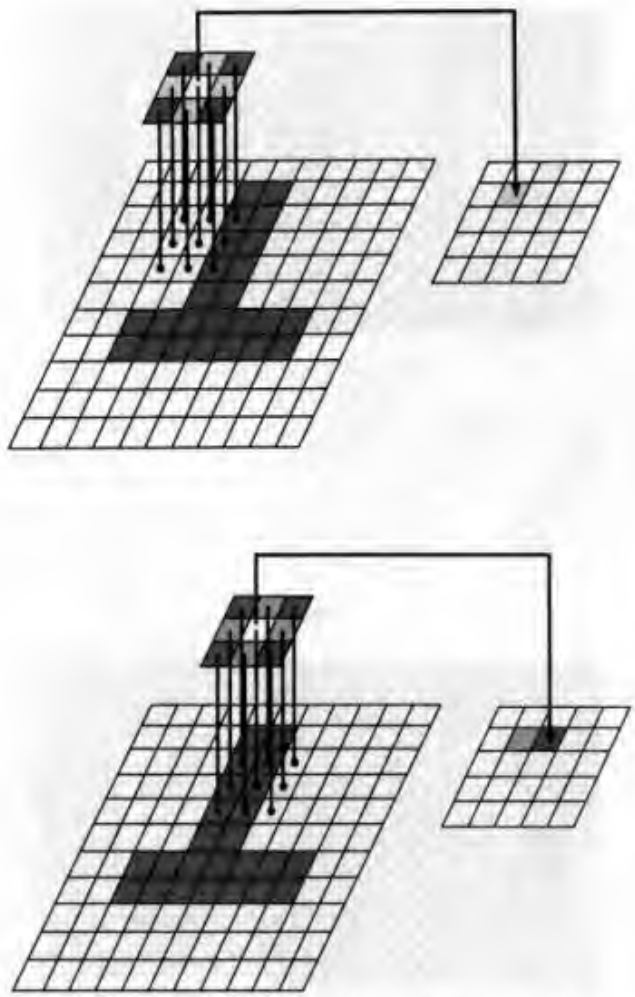


Fig. 14.32 Digital filtering. Filter is used to combine samples to create a new sample.

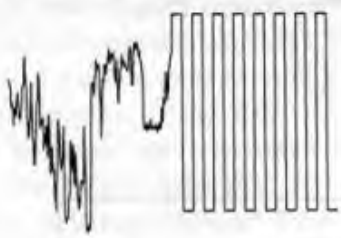
Although it is computationally attractive to use a 1-pixel-wide box filter that averages all subpixel samples, better filters can produce better results, as demonstrated in Fig. 14.33. Note that, no matter what filter is used to postfilter the samples, damage caused by an inadequate initial sampling rate will not be repaired. A rule of thumb is that supersampling four times in each of x and y often will be satisfactory [WHIT85]. This works because the high frequencies in most graphics images are caused by discontinuities at edges, which have a Fourier transform that tapers off rapidly (like the Fourier transform of a pulse—the sinc). In contrast, images with textures and distant objects viewed in perspective have a Fourier transform that is richer in high frequencies and that may be arbitrarily difficult to filter.

Although it is easy to increase the sampling rate, this approach is limited in its usefulness by corresponding increases in both processing time and storage. A number of variations on point sampling have been implemented to address these issues without sacrificing the conceptually simple mechanism of point sampling itself. In *adaptive*

Samsung/Google, EX1027, p. 655



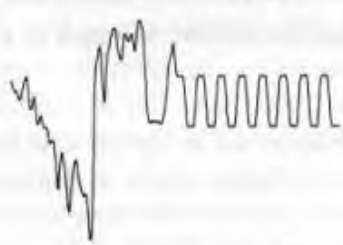
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0



(a)



1 2 3 4 3 2 1
2 4 6 8 6 4 2
3 6 9 12 9 6 3
4 8 12 16 12 8 4
3 6 9 12 9 6 3
2 4 6 8 6 4 2
1 2 3 4 3 2 1



(c)

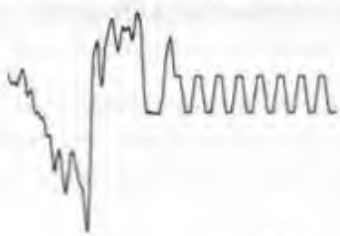
Fig. 14.33 *(Cont'd.)*

supersampling, an example of which is discussed in Section 15.10.4, the sampling rate is varied across the image, with additional samples taken when the system determines that they are needed. *Stochastic supersampling*, discussed in Section 16.12.4, places samples at stochastically determined positions, rather than in a regular grid. This approach produces

Samsung/Google, EX1027, p. 656



1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1



(b)



1 4 8 10 8 4 1
4 12 25 29 25 12 4
8 25 49 58 49 25 8
10 29 58 67 58 29 10
8 25 49 58 49 25 8
4 12 25 29 25 12 4
1 4 8 10 8 4 1



(d)

Fig. 14.33 Filtered images with intensity plot of middle scan line and filter kernel. (a) Original image. (b) Box filter. (c) Bartlett filter. (d) Gaussian filter. Images are 512 × 512, and filters are 7 × 7. Middle scan line is at bottom of a checkerboard row. Because 2D filter covers light and dark squares above and below scan line, amplitude of filtered checkerboard signal along middle scan line is greatly diminished. (Courtesy of George Wolberg, Columbia University.)

aliasing in the form of noise, which our visual system finds less irritating than the clearly defined frequency components of regular aliasing. These two approaches can be combined, allowing the determination of where to place new samples to be based on the statistical properties of those that have already been obtained.

Samsung/Google, EX1027, p. 657

When the original source signal is itself a sampled image, postfiltering followed by resampling may be used to create a new image that has been scaled, rotated, or distorted in a variety of ways. These *image transformations* are discussed in Chapter 17.

14.11 SUMMARY

In this chapter, we provided a high-level introduction to the techniques used to produce realistic images. We then examined the causes of and cures for aliasing. In the following chapters, we discuss in detail how these techniques can be implemented. There are five key questions that you should bear in mind when you read about the algorithms presented in later chapters:

1. *Is the algorithm general or special purpose?* Some techniques work best only in specific circumstances; others are designed to be more general. For example, some algorithms assume that all objects are convex polyhedra and derive part of their speed and relative simplicity from this assumption.
2. *Can antialiasing be incorporated?* Some algorithms may not accommodate antialiasing as easily as others do.
3. *What is the algorithm's space–time performance?* How is the algorithm affected by factors such as the size or complexity of the database, or the resolution at which the picture is rendered?
4. *How convincing are the effects generated?* For example, is refraction modeled correctly, does it look right only in certain special cases, or is it not modeled at all? Can additional effects, such as shadows or specular reflection, be added? How convincing will they be? Sacrificing the accuracy with which an effect is rendered may make possible significant improvements in a program's space or time requirements.
5. *Is the algorithm appropriate, given the purpose for which the picture is created?* The philosophy behind many of the pictures in the following chapters can be summed up by the credo, "If it looks good, do it!" This directive can be interpreted two ways. A simple or fast algorithm may be used if it produces attractive effects, even if no justification can be found in the laws of physics. On the other hand, a shockingly expensive algorithm may be used if it is the only known way to render certain effects.

EXERCISES

14.1 Suppose you had a graphics system that could draw any of the color plates referenced in this chapter in real time. Consider several application areas with which you are (or would like to be) familiar. For each area, list those effects that would be most useful, and those that would be least useful.

14.2 Show that you cannot infer the direction of rotation from orthographic projections of a monochrome, rotating, wireframe cube. Explain how additional techniques can help to make the direction of rotation clear without changing the projection.

14.3 Consider the pulse function $f(x) = 1$ for $-1 \leq x \leq 1$, and $f(x) = 0$ elsewhere. Show that the Fourier transform of $f(x)$ is a multiple of the sinc function. Hint: The Fourier transform of $f(x)$ can be

Samsung/Google, EX1027, p. 658

computed as

$$F(u) = \int_{-1}^{+1} f(x)[\cos 2\pi ux - i\sin 2\pi ux]dx,$$

because the regions where $f(x) = 0$ contribute nothing to the integral, and $f(x) = 1$ in the remaining region. (Apply the inverse Fourier transform to your answer. You should get the original function.)

14.4 Prove that reconstructing a signal with a triangle filter of width 2 corresponds to linearly interpolating its samples. What happens if the filter is wider?

14.5 Write a program that allows you to convolve an image with a filter kernel. Use different filter kernels to create images of the same size from original images with 2, 4, and 8 times the number of pixels in x or y as the new images. You can obtain original images by saving the frame buffer generated by the graphics packages that you have been using. Do your filtered images look better than original images of the same resolution? Does your experience corroborate the rule of thumb mentioned in Section 14.10.6?

Samsung/Google, EX1027, p. 659

15 Visible-Surface Determination

Given a set of 3D objects and a viewing specification, we wish to determine which lines or surfaces of the objects are visible, either from the center of projection (for perspective projections) or along the direction of projection (for parallel projections), so that we can display only the visible lines or surfaces. This process is known as *visible-line* or *visible-surface determination*, or *hidden-line* or *hidden-surface elimination*. In visible-line determination, lines are assumed to be the edges of opaque surfaces that may obscure the edges of other surfaces farther from the viewer. Therefore, we shall refer to the general process as *visible-surface determination*.

Although the statement of this fundamental idea is simple, its implementation requires significant processing power, and consequently involves large amounts of computer time on conventional machines. These requirements have encouraged the development of numerous carefully structured visible-surface algorithms, many of which are described in this chapter. In addition, many special-purpose architectures have been designed to address the problem, some of which are discussed in Chapter 18. The need for this attention can be seen from an analysis of two fundamental approaches to the problem. In both cases, we can think of each object as comprising one or more polygons (or more complex surfaces).

The first approach determines which of n objects is visible at each pixel in the image. The pseudocode for this approach looks like this:

```
for (each pixel in the image) {
    determine the object closest to the viewer that is pierced by
        the projector through the pixel;
    draw the pixel in the appropriate color;
}
```


Samsung/Google, EX1027, p. 660

A straightforward, brute-force way of doing this for 1 pixel requires examining all n objects to determine which is closest to the viewer along the projector passing through the pixel. For p pixels, the effort is proportional to np, where p is over 1 million for a high-resolution display.

The second approach is to compare objects directly with each other, eliminating entire objects or portions of them that are not visible. Expressed in pseudocode, this becomes

```
for (each object in the world) {
    determine those parts of the object whose view is unobstructed
        by other parts of it or any other object;
    draw those parts in the appropriate color;
}
```

We can do this naively by comparing each of the n objects to itself and to the other objects, and discarding invisible portions. The computational effort here is proportional to n^2. Although this second approach might seem superior for $n < p$, its individual steps are typically more complex and time consuming, as we shall see, so it is often slower and more difficult to implement.

We shall refer to these prototypical approaches as *image-precision* and *object-precision* algorithms, respectively. Image-precision algorithms are typically performed at the resolution of the display device, and determine the visibility at each pixel. Object-precision algorithms are performed at the precision with which each object is defined, and determine the visibility of each object.[1] Since object-precision calculations are done without regard to a particular display resolution, they must be followed by a step in which the objects are actually displayed at the desired resolution. Only this final display step needs to be repeated if the size of the finished image is changed; for example, to cover a different number of pixels on a raster display. This is because the geometry of each visible object's projection is represented at the full object database resolution. In contrast, consider enlarging an image created by an image-precision algorithm. Since visible-surface calculations were performed at the original lower resolution, they must be done again if we wish to reveal further detail. Recalling our discussion of sampling in Chapter 14, we can think of object-precision algorithms as operating on the original continuous object data, and image-precision algorithms as operating on sampled data; thus, image-precision algorithms fall prey to aliasing in computing visibility, whereas object-precision algorithms do not.

Object-precision algorithms were first developed for vector graphics systems. On these devices, hidden-line removal was most naturally accomplished by turning the initial list of

[1]The terms *image space* and *object space*, popularized by Sutherland, Sproull, and Schumacker [SUTH74a], are often used to draw the same distinction. Unfortunately, these terms have also been used quite differently in computer graphics. For example, *image-space* has been used to refer to objects after perspective transformation [CATM75] or after projection onto the view plane [GILO78], but still at their original precision. To avoid confusion, we have opted for our slightly modified terms. We refer explicitly to an object's perspective transformation or projection, when appropriate, and reserve the terms *image precision* and *object precision* to indicate the precision with which computations are performed. For example, intersecting two objects' projections on the view plane is an object-precision operation if the precision of the original object definitions is maintained.

Samsung/Google, EX1027, p. 661

lines into one in which lines totally hidden by other surfaces were removed, and partially hidden lines were clipped to one or more visible line segments. All processing was performed at the precision of the original list and resulted in a list in the same format. In contrast, image-precision algorithms were first written for raster devices to take advantage of the relatively small number of pixels for which the visibility calculations had to be performed. This was an understandable partitioning. Vector displays had a large address space (4096 by 4096 even in early systems) and severe limits on the number of lines and objects that could be displayed. Raster displays, on the other hand, had a limited address space (256 by 256 in early systems) and the ability to display a potentially unbounded number of objects. Later algorithms often combine both object- and image-precision calculations, with object-precision calculations chosen for accuracy, and image-precision ones chosen for speed.

In this chapter, we first introduce a subset of the visible-surface problem, displaying single-valued functions of two variables. Next, we discuss a variety of ways to increase the efficiency of general visible-surface algorithms. Then, we present the major approaches to determining visible surfaces.

15.1 FUNCTIONS OF TWO VARIABLES

One of the most common uses of computer graphics has been to plot single-valued continuous functions of two variables, such as $y = f(x, z)$. These functions define surfaces like that shown in Fig. 15.1(a). They present an interesting special case of the hidden-surface problem, for which especially fast solutions are possible.

A function of x and z may be approximated by an m by n array Y of values. Each array element may be thought of as representing a height at one point in a regular grid of samples. We assume that each column of the array corresponds to a single x coordinate, and that each row of the array corresponds to a single z coordinate. In other words, the rectangular array is aligned with the x and z axes. The indices of an array element and the element's value together specify the coordinates of a 3D point. A wireframe drawing of the surface may be constructed as a piecewise linear approximation by drawing one set of polylines through the points defined by each row of the array Y (polylines of constant z) and an orthogonal set of polylines through the points defined by each column (polylines of constant x). A

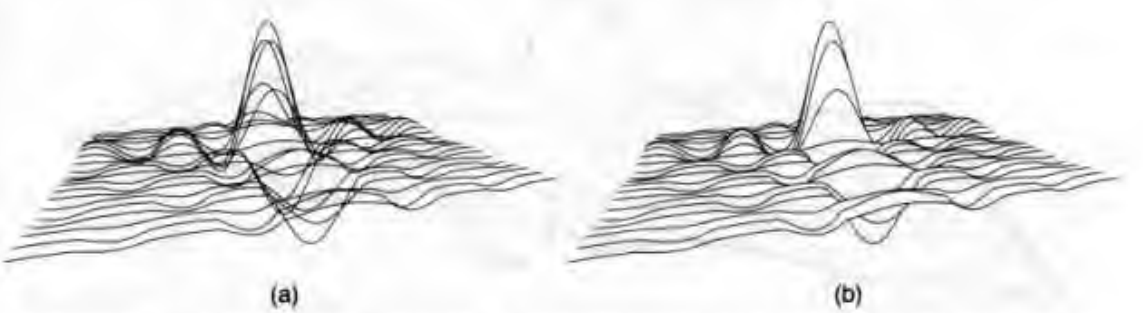


Fig. 15.1 A single-valued function of two variables: (a) without and (b) with hidden lines removed. (By Michael Hatzitheodorou, Columbia University.)

Samsung/Google, EX1027, p. 662

hidden-line algorithm must suppress all parts of the lines that would be obscured from the viewpoint by other parts of the surface, as shown in Fig. 15.1(b).

To see how we might accomplish this, we first consider the problem of plotting only polylines of constant z, assuming that the closest polyline to the viewpoint is an edge of the surface. An efficient class of solutions to this problem is based on the recognition that each polyline of constant z lies in a separate, parallel plane of constant z [WILL72; WRIG73; BUTL79]. Since none of these planes (and hence none of the polylines) can intersect any of the others, each polyline cannot be obscured by any polyline of constant z farther from the viewpoint. Therefore, we will draw polylines of constant z in order of increasing distance from the viewpoint. This establishes a *front-to-back* order. Drawing each polyline of constant z correctly requires only that we not draw those parts of the polyline that are obscured by parts of the surface already drawn.

Consider the "silhouette boundary" of the polylines drawn thus far on the view plane, shown as thick lines in Fig. 15.2. When a new polyline is drawn, it should be visible only where its projection rises above the top or dips below the bottom of the old silhouette. Since each new polyline has a constant z that is farther than that of any of the preceding polylines, it cannot pierce any part of the surface already drawn. Therefore, to determine what parts are to be drawn, we need only to compare the current polyline's projected y values with those of the corresponding part of the surface's silhouette computed thus far. When only enough information is encoded to represent a minimum and maximum silhouette y for each x, the algorithm is known as an *horizon line algorithm*.

One way to represent this silhouette, implemented by Wright [WRIG73], uses two 1D arrays, *YMIN* and *YMAX*, to hold the minimum and maximum projected y values for a finite set of projected x values. These are image-precision data structures because they have a finite number of entries. *YMIN* and *YMAX* are initialized with y values that are, respectively, above and below all the projected y values of the surface. When a new polyline is drawn, the projected y values of each pair of adjacent vertices are compared with the values at the corresponding locations in the silhouette arrays. As shown in Fig. 15.3, a vertex whose value is above that in the corresponding position in *YMAX* (A, B, G) or below that in *YMIN* (E, F) is visible; otherwise, it is invisible (C, D). If both vertices are invisible, then the line segment is wholly invisible (CD) and the silhouette arrays remain unchanged.

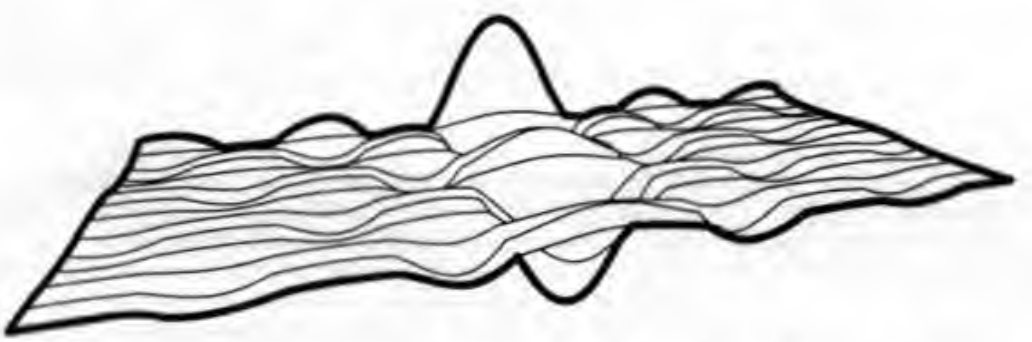


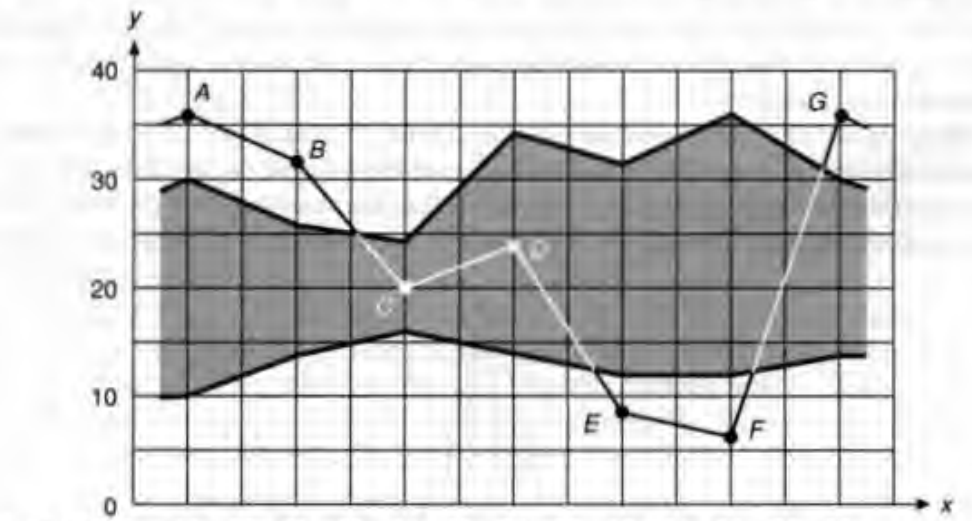
Fig. 15.2 Silhouette of lines drawn. (By Michael Hatzitheodorou, Columbia University.)

Samsung/Google, EX1027, p. 663

If both vertices are visible with regard to the same silhouette array (*AB*, *EF*), then the line segment is wholly visible and should be drawn in its entirety, and that silhouette array should be updated. The x coordinates of two adjacent vertices in a polyline of constant z usually map to nonadjacent locations in the silhouette arrays. In this situation, values of y to insert in the intervening silhouette array locations can be obtained by linearly interpolating between the projected y values of the two adjacent elements of Y.

Finally, we must consider the case of a partially visible line, in which both vertices are not visible with regard to the same silhouette array. Although this typically means that one of the vertices is visible and the other is invisible (*BC*, *DE*), it may be the case that both vertices are visible, one above *YMAX* and the other below *YMIN* (*FG*). Interpolated y values can be compared with those at the intervening locations in the silhouette arrays to determine the point(s) of intersection. The line should not be visible at those places where an interpolated y value falls inside the silhouette. Only the visible parts of the line segment outside the silhouette should be drawn, and the silhouette array should be updated, as shown in Fig. 15.3. When the two adjacent silhouette elements are found between which the line changes visibility, the line can be intersected with the line defined by the x and y values of those elements to determine an endpoint for a line-drawing algorithm.

Unfortunately, the image-precision *YMIN* and *YMAX* silhouette arrays make this algorithm prone to aliasing problems. If the line-drawing primitive used has higher resolution than do the silhouette arrays, aliasing can manifest itself as hidden segments that are drawn or as visible segments that are not. For example, Fig. 15.4 shows three polylines being drawn in front-to-back order. The two polylines of parts (a) and (b) form a gap when



New *YMAX*	36	34	32	26	24	29	34	33	32	34	36	33	36
New *YMIN*	10	12	14	15	16	15	14	13	8	7	6	13	14
Line's *Y*	**36**	34	**32**	26	**20**	22	**24**	16	**8**	7	**6**	21	**36**
Old *YMAX*	30	28	26	25	24	29	34	33	32	34	36	33	30
Old *YMIN*	10	12	14	15	16	15	14	13	12	12	12	13	14

Fig. 15.3 The y values for positions on the line between the endpoints must be interpolated and compared with those in the silhouette arrays. Endpoint values are shown in boldface type.

Samsung/Google, EX1027, p. 664

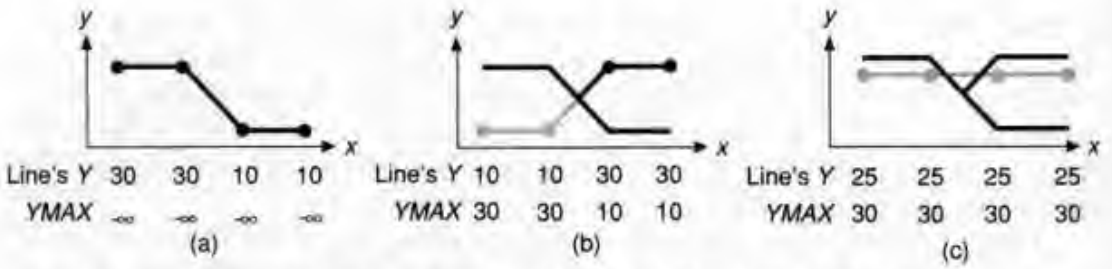


Fig. 15.4 Aliasing problem arising from the use of image-precision silhouette arrays.

they cross. The second line can be correctly hidden by the first by interpolating the values in *YMAX* to determine the intersection of the two lines. After the second line has been drawn, however, both values in *YMAX* are the same, so the third line, drawn in part (c), incorrectly remains hidden when it passes over their intersection. Using higher-resolution silhouette arrays can reduce such problems at the expense of increased processing time.

An alternative to image-precision *YMIN* and *YMAX* arrays is to use object-precision data structures [WILL72]. The *YMIN* and *YMAX* arrays can be replaced by two object-precision polylines representing the silhouette boundaries. As each segment of a polyline of constant z is inspected, only those parts projecting above the *YMAX* polyline or below the *YMIN* polyline are drawn. The projected lines representing these visible parts are linked into the silhouette polyline beginning at the point at which they intersect it, replacing the subsequent parts of the silhouette polyline until the next intersection. The accuracy gained by this approach must be weighed against the extra run-time overhead incurred by searching and maintaining the silhouette polylines and the added difficulties of coding a more complex algorithm.

Suppose we wish to draw polylines of constant x, instead of polylines of constant z, to produce the view shown in Fig. 15.5. In this case, the polyline of constant x closest to the observer does not form an edge of the surface. It is the seventh (the most nearly vertical) polyline from the left. To render the surface correctly, we must render polylines to the right

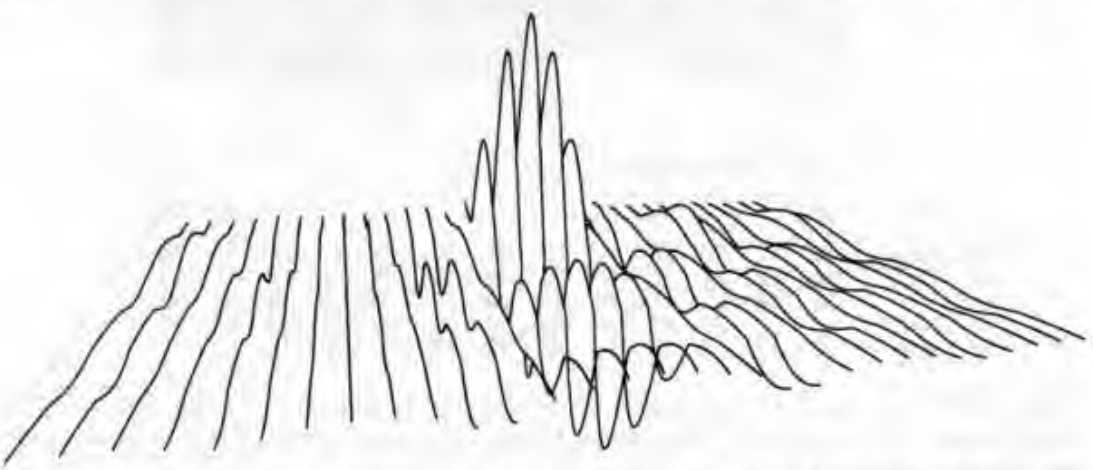


Fig. 15.5 Surface drawn using polylines of constant x, instead of polylines of constant z. (By Michael Hatzitheodorou, Columbia University.)

Samsung/Google, EX1027, p. 665



Fig. 15.6 Surface of Fig. 15.5 with lines of constant x and z. (By Michael Hatzitheodorou, Columbia University.)

of the closest one from left to right, and those to the left of the closest one from right to left. In both cases, polylines are rendered in front-to-back order relative to the observer.

The image-precision algorithm can be easily extended to plot lines of constant x as well as of constant z, as shown in Fig. 15.6. Although it is tempting to assume that superimposing the correctly plotted polylines of constant x and polylines of constant z would work, this does not allow lines from each set to hide those in the other, as shown in Fig. 15.7. Wright [WRIG73] points out that the correct solution can be obtained by interleaving the processing of the two sets of polylines. The set of those polylines that are most nearly parallel to the view plane (e.g., those of constant z) is processed in the same order as before. After each polyline of constant z is processed, the segments of each polyline of constant x between the just-processed polyline of z and the next polyline of z are drawn. The line segments of x must be drawn using the same copy of the silhouette data structure as was used for drawing the polylines of constant z. In addition, they too must be processed in front-to-back order. Figure 15.8(a) shows lines of constant x as processed in the correct order, from left to right in this case. The lines of constant x in Fig. 15.8(b) have been processed in the opposite, incorrect order, from right to left. The incorrect drawing order causes problems, because each successive line is shadowed in the *YMAX* array by the lines drawn previously.

Although the algorithms described in this section are both useful and efficient, they fail for any viewing specification for which the silhouette of the object is not a function of x when projected on the view plane. An example of such a viewing specification is shown in Fig. 15.9, which was rendered with an algorithm developed by Anderson [ANDE82] that uses more complex data structures to handle arbitrary viewing specifications.

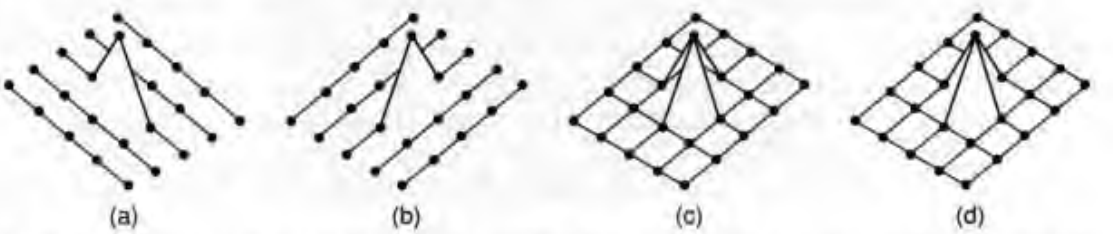


Fig. 15.7 (a) Lines of constant z. (b) Lines of constant x. (c) Superposition of parts (a) and (b). (d) The correct solution. (Based on [WRIG73].)

Samsung/Google, EX1027, p. 666

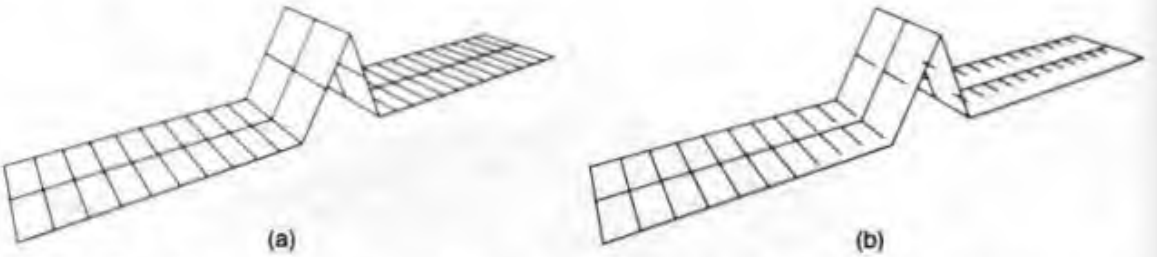


Fig. 15.8 Polylines of constant x, like those of constant z, must be processed in the correct order. (a) Correctly processed lines. (b) Incorrectly processed lines. (By Michael Hatzitheodorou, Columbia University.)

15.2 TECHNIQUES FOR EFFICIENT VISIBLE-SURFACE ALGORITHMS

As we have just seen, a restricted version of the visible-line problem for functions of two variables can be solved efficiently by using clever data structures. What can be done for the general problem of visible-surface determination? The simple formulations of prototypical image-precision and object-precision algorithms given at the beginning of this chapter require a number of potentially costly operations. These include determining for a projector and an object, or for two objects' projections, whether or not they intersect and where they intersect. Then, for each set of intersections, it is necessary to compute the object that is closest to the viewer and therefore visible. To minimize the time that it takes to create a picture, we must organize visible-surface algorithms so that costly operations are performed

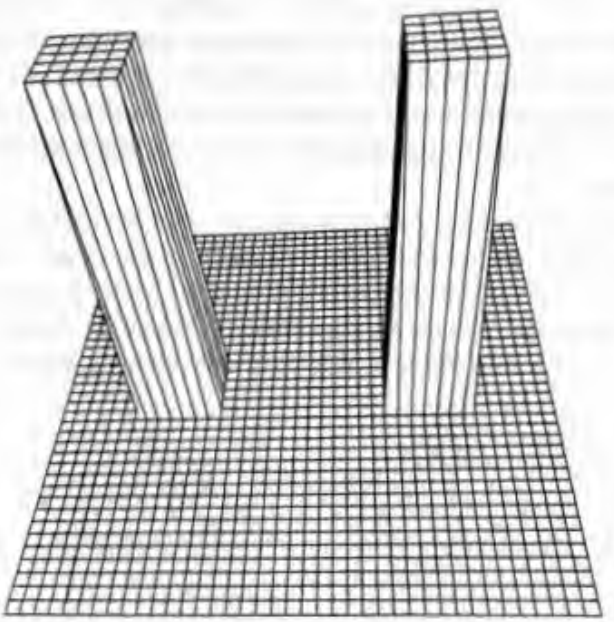


Fig. 15.9 A projection not handled by the algorithm discussed here. (Courtesy of David P. Anderson, University of California, Berkeley.)

Samsung/Google, EX1027, p. 667

as efficiently and as infrequently as possible. The following sections describe some general ways to do this.

15.2.1 Coherence

Sutherland, Sproull, and Schumacker [SUTH74a] point out how visible-surface algorithms can take advantage of *coherence*—the degree to which parts of an environment or its projection exhibit local similarities. Environments typically contain objects whose properties vary smoothly from one part to another. In fact, it is the less frequent discontinuities in properties (such as depth, color, and texture) and the effects that they produce in pictures, that let us distinguish between objects. We exploit coherence when we reuse calculations made for one part of the environment or picture for other nearby parts, either without changes or with incremental changes that are more efficient to make than recalculating the information from scratch. Many different kinds of coherence have been identified [SUTH74a], which we list here and refer to later:

- *Object coherence*. If one object is entirely separate from another, comparisons may need to be done only between the two objects, and not between their component faces or edges. For example, if all parts of object A are farther from the viewer than are all parts of object B, none of A's faces need be compared with B's faces to determine whether they obscure B's faces.
- *Face coherence*. Surface properties typically vary smoothly across a face, allowing computations for one part of a face to be modified incrementally to apply to adjacent parts. In some models, faces can be guaranteed not to interpenetrate.
- *Edge coherence*. An edge may change visibility only where it crosses behind a visible edge or penetrates a visible face.
- *Implied edge coherence*. If one planar face penetrates another, their line of intersection (the implied edge) can be determined from two points of intersection.
- *Scan-line coherence*. The set of visible object spans determined for one scan line of an image typically differs little from the set on the previous line.
- *Area coherence*. A group of adjacent pixels is often covered by the same visible face. A special case of area coherence is *span coherence*, which refers to a face's visibility over a span of adjacent pixels on a scan line.
- *Depth coherence*. Adjacent parts of the same surface are typically close in depth, whereas different surfaces at the same screen location are typically separated farther in depth. Once the depth at one point of the surface is calculated, the depth of points on the rest of the surface can often be determined by a simple difference equation.
- *Frame coherence*. Pictures of the same environment at two successive points in time are likely to be quite similar, despite small changes in objects and viewpoint. Calculations made for one picture can be reused for the next in a sequence.

15.2.2 The Perspective Transformation

Visible-surface determination clearly must be done in a 3D space prior to the projection into 2D that destroys the depth information needed for depth comparisons. Regardless of

Samsung/Google, EX1027, p. 668

the kind of projection chosen, the basic depth comparison at a point can be typically reduced to the following question: Given points $P_1 = (x_1, y_1, z_1)$ and $P_2 = (x_2, y_2, z_2)$, does either point obscure the other? This question is the same: Are P_1 and P_2 on the same projector (see Fig. 15.10)? If the answer is yes, z_1 and z_2 are compared to determine which point is closer to the viewer. If the answer is no, then neither point can obscure the other.

Depth comparisons are typically done after the normalizing transformation (Chapter 6) has been applied, so that projectors are parallel to the z axis in parallel projections or emanate from the origin in perspective projections. For a parallel projection, the points are on the same projector if $x_1 = x_2$ and $y_1 = y_2$. For a perspective projection, we must unfortunately perform four divisions to determine whether $x_1 / z_1 = x_2 / z_2$ and $y_1 / z_1 = y_2 / z_2$, in which case the points are on the same projector, as shown in Fig. 15.10. Moreover, if P_1 is later compared against some P_3, two more divisions are required.

Unnecessary divisions can be avoided by first transforming a 3D object into the 3D screen-coordinate system, so that the parallel projection of the transformed object is the same as the perspective projection of the untransformed object. Then the test for one point obscuring another is the same as for parallel projections. This perspective transformation distorts the objects and moves the center of projection to infinity on the positive z axis, making the projectors parallel (see Fig. 6.56). Figure 15.11 shows the effect of this transformation on the perspective view volume; Fig. 15.12 shows how a cube is distorted by the transformation.

The essence of such a transformation is that it preserves relative depth, straight lines, and planes, and at the same time performs the perspective foreshortening. As discussed in Chapter 6, the division that accomplishes the foreshortening is done just once per point, rather than each time two points are compared. The matrix from Eq. (6.48)

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{1}{1 + z_{min}} & \dfrac{-z_{min}}{1 + z_{min}} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \quad z_{min} \neq -1 \tag{15.1}$$

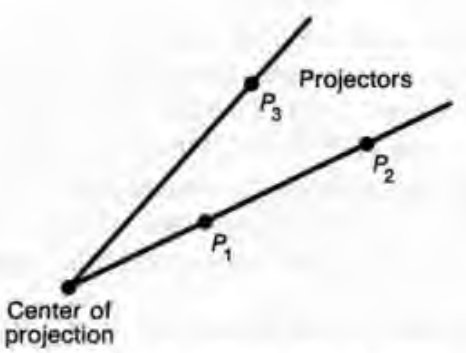


Fig. 15.10 If two points P_1 and P_2 are on the same projector, then the closer one obscures the other; otherwise, it does not (e.g., P_1 does not obscure P_3).

Samsung/Google, EX1027, p. 669

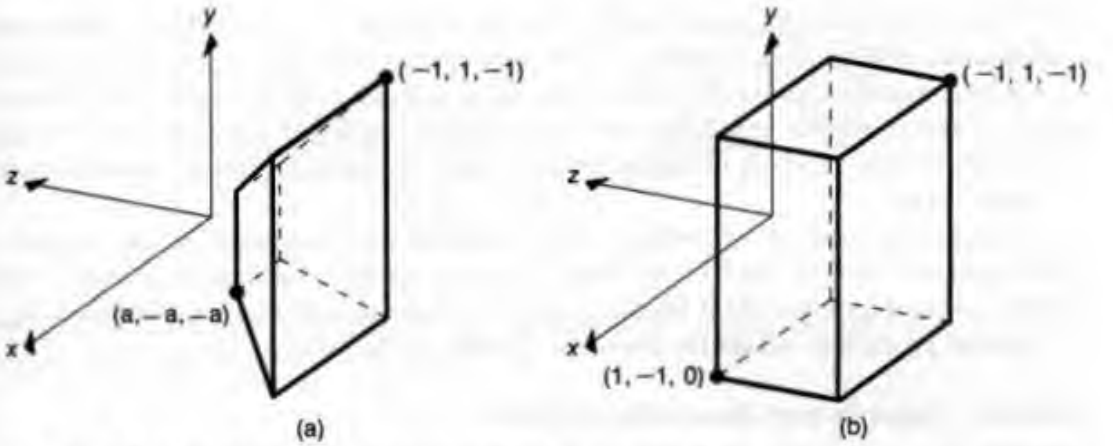


Fig. 15.11 The normalized perspective view volume (a) before and (b) after perspective transformation.

transforms the normalized perspective view volume into the rectangular parallelepiped bounded by

$$-1 \le x \le 1, \qquad -1 \le y \le 1, \qquad -1 \le z \le 0. \tag{15.2}$$

Clipping can be done against the normalized truncated-pyramid view volume before M is applied, but then the clipped results must be multiplied by M. A more attractive alternative is to incorporate M into the perspective normalizing transformation N_{per} from Chapter 6, so that just a single matrix multiplication is needed, and then to clip in homogeneous coordinates prior to the division. If we call the results of that multiplication (X, Y, Z, W), then, for $W > 0$, the clipping limits become

$$-W \le X \le W, \qquad -W \le Y \le W, \qquad -W \le Z \le 0. \tag{15.3}$$

These limits are derived from Eq. (15.2) by replacing x, y, and z by X/W, Y/W, and Z/W, respectively, to reflect the fact that x, y, and z in Eq. (15.2) result from division by W. After clipping, we divide by W to obtain (x_p, y_p, z_p).(See Section 6.5.4. for what to do when

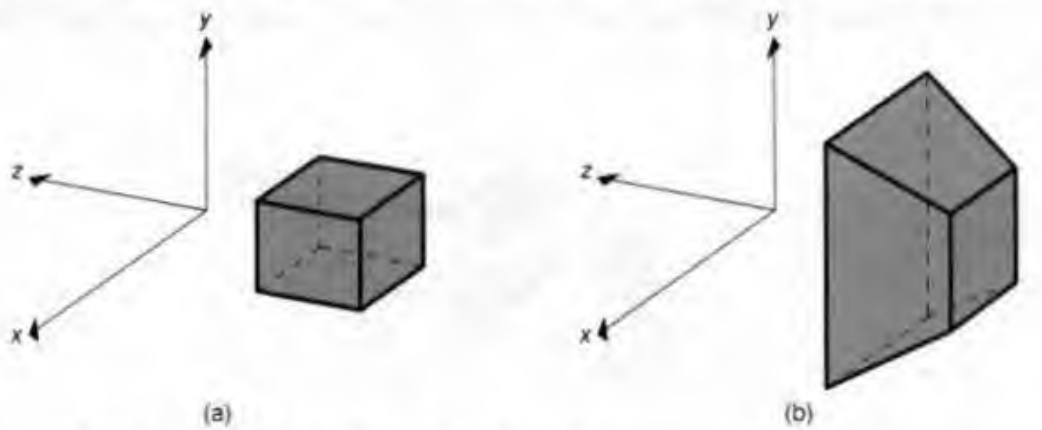


Fig. 15.12 A cube (a) before and (b) after perspective transformation.

Samsung/Google, EX1027, p. 670

$W < 0$.) Note that M assumes the view volume is in the negative z half-space. For notational convenience, however, our examples will often use decreasing positive z values, rather than decreasing negative z values, to indicate increasing distance from the viewer. In contrast, many graphics systems transform their right-handed world into a left-handed viewing coordinate system, in which increasing positive z values correspond to increasing distance from the viewer.

We can now proceed with visible-surface determination unfettered by the complications suggested by Fig. 15.10. Of course, when a parallel projection is specified, the perspective transformation M is unnecessary, because the normalizing transformation N_{par} for parallel projections makes the projectors parallel to the z axis.

15.2.3 Extents and Bounding Volumes

Screen extents, introduced in Chapter 3 as a way to avoid unnecessary clipping, are also commonly used to avoid unnecessary comparisons between objects or their projections. Figure 15.13 shows two objects (3D polygons, in this case), their projections, and the upright rectangular screen extents surrounding the projections. The objects are assumed to have been transformed by the perspective transformation matrix M of Section 15.2.2. Therefore, for polygons, orthographic projection onto the (x, y) plane is done trivially by ignoring each vertex's z coordinate. In Fig. 15.13, the extents do not overlap, so the projections do not need to be tested for overlap with one another. If the extents overlap, one of two cases occurs, as shown in Fig. 15.14: either the projections also overlap, as in part (a), or they do not, as in part (b). In both cases, more comparisons must be performed to determine whether the projections overlap. In part (b), the comparisons will establish that the two projections really do not intersect; in a sense, the overlap of the extents was a false alarm. Extent testing thus provides a service similar to that of trivial reject testing in clipping.

Rectangular-extent testing is also known as *bounding-box* testing. Extents can be used as in Chapter 7 to surround the objects themselves rather than their projections: in this case, the extents become solids and are also known as *bounding volumes*. Alternatively, extents can be used to bound a single dimension, in order to determine, say, whether or not two objects overlap in z. Figure 15.15 shows the use of extents in such a case; here, an extent is the infinite volume bounded by the minimum and maximum z values for each object. There

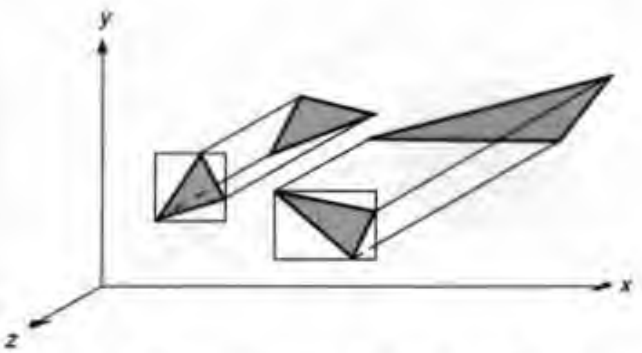


Fig. 15.13 Two objects, their projections onto the (x, y) plane, and the extents surrounding the projections.

Samsung/Google, EX1027, p. 671

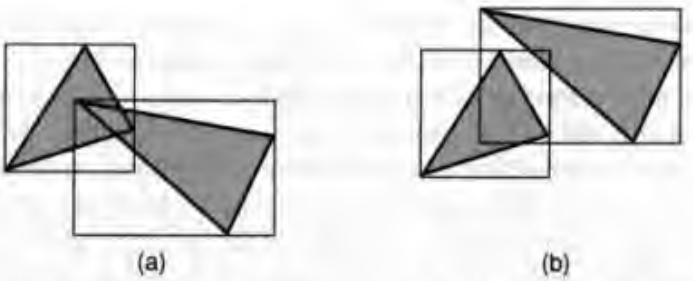


Fig. 15.14 Extents bounding object projections. (a) Extents and projections overlap. (b) Extents overlap, but projections do not.

is no overlap in z if

$$z_{max2} < z_{min1} \quad or \quad z_{max1} < z_{min2}. \tag{15.4}$$

Comparing against minimum and maximum bounds in one or more dimensions is also known as *minmax* testing. When comparing minmax extents, the most complicated part of the job is finding the extent itself. For polygons (or for other objects that are wholly contained within the convex hull of a set of defining points), an extent may be computed by iterating through the list of point coordinates and recording the largest and smallest values for each coordinate.

Extents and bounding volumes are used not only to compare two objects or their projections with each other, but also to determine whether or not a projector intersects an object. This involves computing the intersection of a point with a 2D projection or a vector with a 3D object, as described in Section 15.10.

Although we have discussed only minmax extents so far, other bounding volumes are possible. What is the best bounding volume to use? Not surprisingly, the answer depends on both the expense of performing tests on the bounding volume itself and on how well the volume protects the enclosed object from tests that do not yield an intersection. Weghorst, Hooper, and Greenberg [WEGH84] treat bounding-volume selection as a matter of minimizing the total cost function T of the intersection test for an object. This may be expressed as

$$T = bB + oO, \tag{15.5}$$

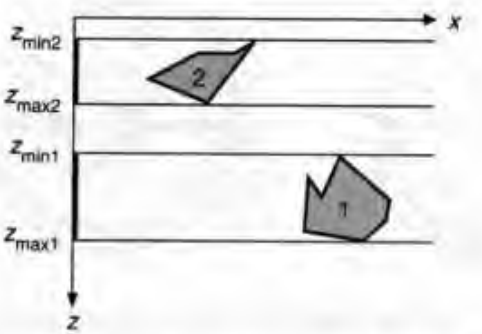


Fig. 15.15 Using 1D extents to determine whether objects overlap.

Samsung/Google, EX1027, p. 672

where b is the number of times the bounding volume is tested for intersection, B is the cost of performing an intersection test on the bounding volume, o is the number of times the object is tested for intersection (the number of times the bounding volume is actually intersected), and O is the cost of performing an intersection test on the object.

Since the object intersection test is performed only when the bounding volume is actually intersected, $o \leq b$. Although O and b are constant for a particular object and set of tests to be performed, B and o vary as a function of the bounding volume's shape and size. A "tighter" bounding volume, which minimizes o, is typically associated with a greater B. A bounding volume's effectiveness may also depend on an object's orientation or the kind of objects with which that object will be intersected. Compare the two bounding volumes for the wagon wheel shown in Fig. 15.16. If the object is to be intersected with projectors perpendicular to the (x, y) plane, then the tighter bounding volume is the sphere. If

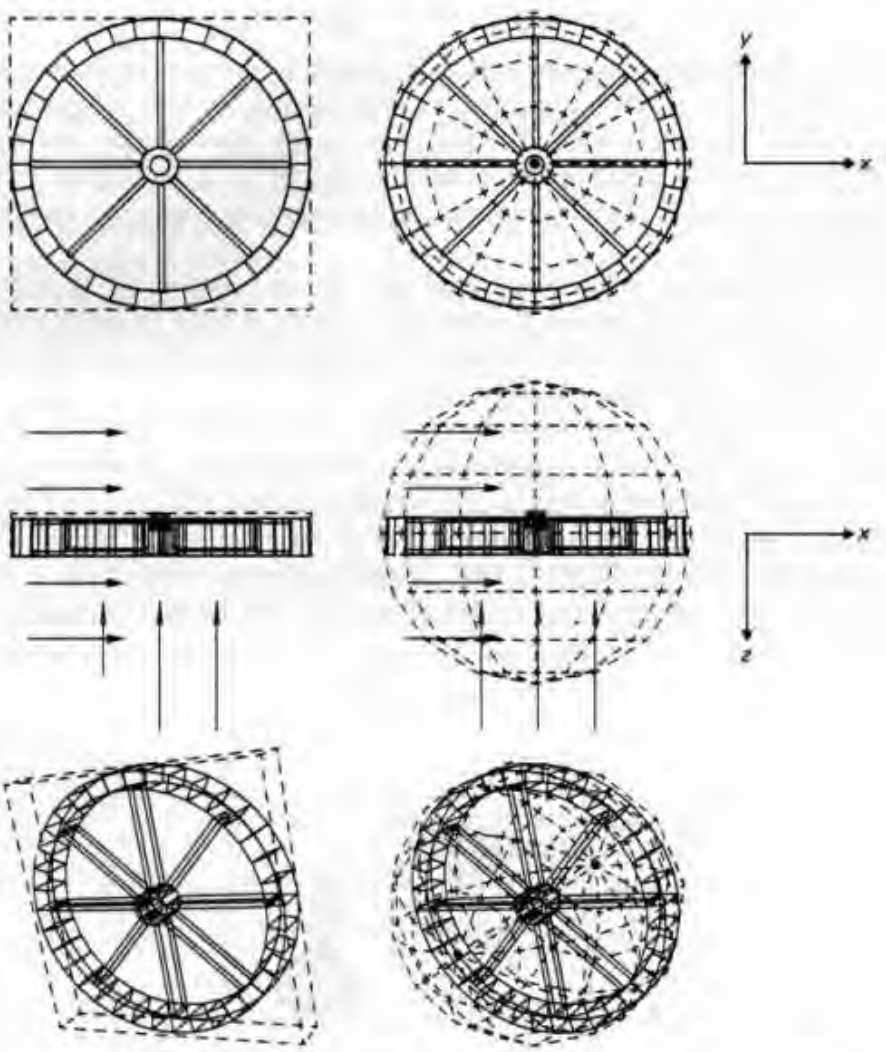


Fig. 15.16 Bounding volume selection. (Courtesy of Hank Weghorst, Gary Hooper, Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1984.)

Samsung/Google, EX1027, p. 673

projectors are perpendicular to the (x, z) or (y, z) planes, then the rectangular extent is the tighter bounding volume. Therefore, multiple bounding volumes may be associated with an object and an appropriate one selected depending on the circumstances.

15.2.4 Back-Face Culling

If an object is approximated by a solid polyhedron, then its polygonal faces completely enclose its volume. Assume that all the polygons have been defined such that their surface normals point out of their polyhedron. If none of the polyhedron's interior is exposed by the front clipping plane, then those polygons whose surface normals point away from the observer lie on a part of the polyhedron whose visibility is completely blocked by other closer polygons, as shown in Fig. 15.17. Such invisible *back-facing* polygons can be eliminated from further processing, a technique known as *back-face culling*. By analogy, those polygons that are not back-facing are often called *front-facing*.

In eye coordinates, a back-facing polygon may be identified by the nonnegative dot product that its surface normal forms with the vector from the center of projection to any point on the polygon. (Strictly speaking, the dot product is positive for a back-facing polygon; a zero dot product indicates a polygon being viewed on edge.) Assuming that the perspective transformation has been performed or that an orthographic projection onto the (x, y) plane is desired, then the direction of projection is $(0, 0, -1)$. In this case, the dot-product test reduces to selecting a polygon as back-facing only if its surface normal has a negative z coordinate. If the environment consists of a single convex polyhedron, back-face culling is the only visible-surface calculation that needs to be performed. Otherwise, there may be front-facing polygons, such as C and E in Fig. 15.17, that are partially or totally obscured.

If the polyhedra have missing or clipped front faces, or if the polygons are not part of polyhedra at all, then back-facing polygons may still be given special treatment. If culling is not desired, the simplest approach is to treat a back-facing polygon as though it were front-facing, flipping its normal in the opposite direction. In PHIGS+, the user can specify a completely separate set of properties for each side of a surface.

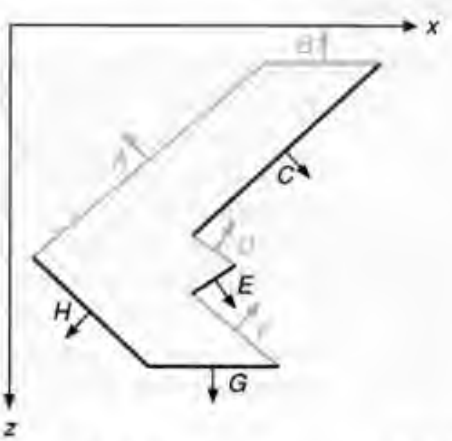


Fig. 15.17 Back-face culling. Back-facing polygons (A,B,D,F) shown in gray are eliminated, whereas front-facing polygons (C,E,G,H) are retained.

Samsung/Google, EX1027, p. 674

Extrapolating from Section 7.12.2's parity-check algorithm for determining whether a point is contained in a polygon, note that a projector passing through a polyhedron intersects the same number of back-facing polygons as of front-facing ones. Thus, a point in a polyhedron's projection lies in the projections of as many back-facing polygons as front-facing ones. Back-face culling therefore halves the number of polygons to be considered for each pixel in an image-precision visible-surface algorithm. On average, approximately one-half of a polyhedron's polygons are back-facing. Thus, back-face culling also typically halves the number of polygons to be considered by the remainder of an object-precision visible-surface algorithm. (Note, however, that this is true only on average. For example, a pyramid's base may be that object's only back- or front-facing polygon.)

As described so far, back-face culling is an object-precision technique that requires time linear in the number of polygons. Sublinear performance can be obtained by preprocessing the objects to be displayed. For example, consider a cube centered about the origin of its own object coordinate system, with its faces perpendicular to the coordinate system's axes. From any viewpoint outside the cube, at most three of its faces are visible. Furthermore, each octant of the cube's coordinate system is associated with a specific set of three potentially visible faces. Therefore, the position of the viewpoint relative to the cube's coordinate system can be used to select one of the eight sets of three potentially visible faces. For objects with a relatively small number of faces, a table may be made up in advance to allow visible-surface determination without processing all the object's faces for each change of viewpoint.

A table of visible faces indexed by viewpoint equivalence class may be quite large, however, for an object with many faces. Tanimoto [TANI77] suggests as an alternative a graph-theoretic approach that takes advantage of frame coherence. A graph is constructed with a node for each face of a convex polyhedron, and a graph edge connecting each pair of nodes whose faces share a polygon edge. The list of edges separating visible faces from invisible ones is then computed for an initial viewpoint. This list contains all edges on the object's silhouette. Tanimoto shows that, as the viewpoint changes between frames, only the visibilities of faces lying between the old and new silhouettes need to be recomputed.

15.2.5 Spatial Partitioning

Spatial partitioning (also known as *spatial subdivision*) allows us to break down a large problem into a number of smaller ones. The basic approach is to assign objects or their projections to spatially coherent groups as a preprocessing step. For example, we can divide the projection plane with a coarse, regular 2D rectangular grid and determine in which grid spaces each object's projection lies. Projections need to be compared for overlap with only those other projections that fall within their grid boxes. This technique is used by [ENCA72; MAHN73; FRAN80; HEDG82]. Spatial partitioning can be used to impose a regular 3D grid on the objects in the environment. The process of determining which objects intersect with a projector can then be sped up by first determining which partitions the projector intersects, and then testing only the objects lying within those partitions (Section 15.10).

If the objects being depicted are unequally distributed in space, it may be more efficient to use *adaptive partitioning*, in which the size of each partition varies. One approach to adaptive partitioning is to subdivide space recursively until some termination criterion is

Samsung/Google, EX1027, p. 675

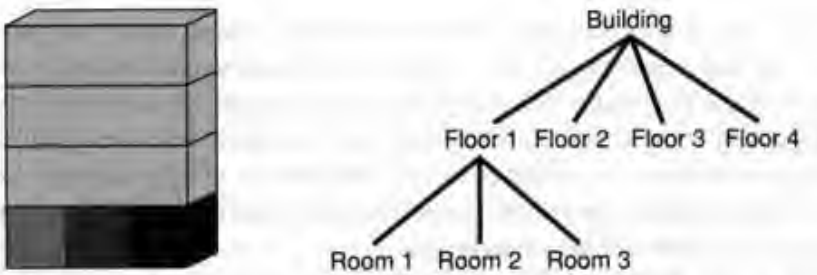


Fig. 15.18 Hierarchy can be used to restrict the number of object comparisons needed. Only if a projector intersects the building and floor 1 does it need to be tested for intersection with rooms 1 through 3.

fulfilled for each partition. For example, subdivision may stop when there are fewer than some maximum number of objects in a partition [TAMM82]. The quadtree, octree, and BSP-tree data structures of Section 12.6 are particularly attractive for this purpose.

15.2.6 Hierarchy

As we saw in Chapter 7, hierarchies can be useful for relating the structure and motion of different objects. A nested hierarchical model, in which each child is considered part of its parent, can also be used to restrict the number of object comparisons needed by a visible-surface algorithm [CLAR76; RUBI80; WEGH84]. An object on one level of the hierarchy can serve as an extent for its children if they are entirely contained within it, as shown in Fig. 15.18. In this case, if two objects in the hierarchy fail to intersect, the lower-level objects of one do not need to be tested for intersection with those of the other. Similarly, only if a projector is found to penetrate an object in the hierarchy must it be tested against the object's children. This use of hierarchy is an important instance of object coherence. A way to automate the construction of hierarchies is discussed in Section 15.10.2.

15.3 ALGORITHMS FOR VISIBLE-LINE DETERMINATION

Now that we have discussed a number of general techniques, we introduce some visible-line and visible-surface algorithms to see how these techniques are used. We begin with visible-line algorithms. The algorithms presented here all operate in object precision and produce as output a list of visible line segments suitable for vector display. The visible-surface algorithms discussed later can also be used for visible-line determination by rendering each surface as a background-colored interior surrounded by a border of the desired line color; most visible-surface algorithms produce an image-precision array of pixels, however, rather than an object-precision list of edges.

15.3.1 Roberts's Algorithm

The earliest visible-line algorithm was developed by Roberts [ROBE63]. It requires that each edge be part of the face of a convex polyhedron. First, back-face culling is used to remove all edges shared by a pair of a polyhedron's back-facing polygons. Next, each remaining edge is compared with each polyhedron that might obscure it. Many polyhedra

Samsung/Google, EX1027, p. 676

can be trivially eliminated from the comparison through extent testing: the extents of their projections may fail to overlap in x or y, or one object's extent may be farther back in z than is the other. Those polyhedra that are tested are compared in sequence with the edge. Because the polyhedra are convex, there is at most one contiguous group of points on any line that is blocked from the observer by any polyhedron. Thus, each polyhedron either obscures the edge totally or causes one or two pieces to remain. Any remaining pieces of the edge are compared with the next polyhedron.

Roberts's visibility test is performed with a parametric version of the projector from the eye to a point on the edge being tested. He uses a linear-programming approach to solve for those values of the line equation that cause the projector to pass through a polyhedron, resulting in the invisibility of the endpoint. The projector passes through a polyhedron if it contains some point that is inside all the polyhedron's front faces. Rogers [ROGE85] provides a detailed explanation of Roberts's algorithm and discusses ways in which that algorithm can be further improved.

15.3.2 Appel's Algorithm

Several more general visible-line algorithms [APPE67; GALI69; LOUT70] require only that lines be the edges of polygons, not polyhedra. These algorithms also consider only lines that bound front-facing polygons, and take advantage of edge-coherence in a fashion typified by Appel's algorithm. Appel [APPE67] defines the *quantitative invisibility* of a point on a line as the number of front-facing polygons that obscure that point. When a line passes behind a front-facing polygon, its quantitative invisibility is incremented by 1; when it passes out from behind that polygon, its quantitative invisibility is decremented by 1. A line is visible only when its quantitative invisibility is 0. Line AB in Fig. 15.19 is annotated with the quantitative invisibility of each of its segments. If interpenetrating polygons are not allowed, a line's quantitative invisibility changes only when it passes behind what Appel

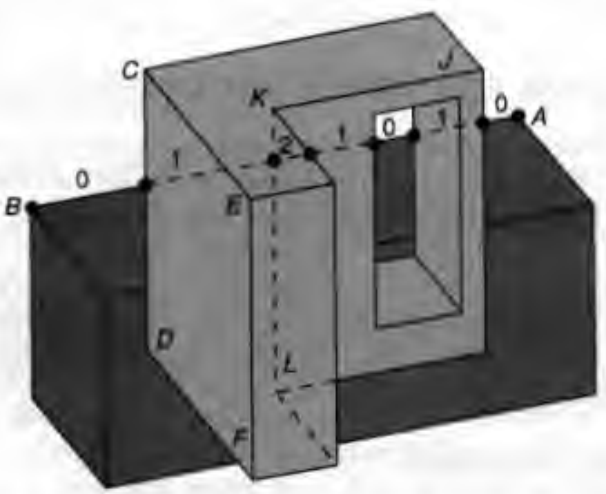


Fig. 15.19 Quantitative invisibility of a line. Dashed lines are hidden. Intersections of AB's projection with projections of contour lines are shown as large dots (•), and each segment of AB is marked with its quantitative invisibility.

Samsung/Google, EX1027, p. 677

calls a *contour line*. A contour line is either an edge shared by a front-facing and a back-facing polygon, or the unshared edge of a front-facing polygon that is not part of a closed polyhedron. An edge shared by two front-facing polygons causes no change in visibility and therefore is not a contour line. In Fig. 15.19, edges *AB*, *CD*, *DF*, and *KL* are contour lines, whereas edges *CE*, *EF*, and *JK* are not.

A contour line passes in front of the edge under consideration if it pierces the triangle formed by the eyepoint and the edge's two endpoints. Whether it does so can be determined by a point-in-polygon containment test, such as that discussed in Section 7.12.2. The projection of such a contour line on the edge can be found by clipping the edge against the plane determined by the eyepoint and the contour line. Appel's algorithm requires that all polygon edges be drawn in a consistent direction about the polygon, so that the sign of the change in quantitative invisibility is determined by the sign of the cross-product of the edge with the contour line.

The algorithm first computes the quantitative invisibility of a "seed" vertex of an object by determining the number of front-facing polygons that hide it. This can be done by a brute-force computation of all front-facing polygons whose intersection with the projector to the seed vertex is closer than is the seed vertex itself. The algorithm then takes advantage of edge coherence by propagating this value along the edges emanating from the point, incrementing or decrementing the value at each point at which an edge passes behind a contour line. Only sections of edges whose quantitative invisibility is zero are drawn. When each line's other endpoint is reached, the quantitative invisibility associated with that endpoint becomes the initial quantitative invisibility of all lines emanating in turn from it.

At vertices through which a contour line passes, there is a complication that requires us to make a correction when propagating the quantitative invisibility. One or more lines emanating from the vertex may be hidden by one or more front-facing polygons sharing the vertex. For example, in Fig. 15.19, edge *JK* has a quantitative invisibility of 0, while edge *KL* has a quantitative invisibility of 1 because it is hidden by the object's top face. This change in quantitative invisibility at a vertex can be taken into account by testing the edge against the front-facing polygons that share the vertex.

For an algorithm such as Appel's to handle intersecting polygons, it is necessary to compute the intersections of edges with front-facing polygons and to use each such intersection to increment or decrement the quantitative invisibility. Since visible-line algorithms typically compare whole edges with other edges or objects, they can benefit greatly from spatial-partitioning approaches. Each edge then needs to be compared with only the other edges or objects in the grid boxes containing its projection.

15.3.3 Haloed Lines

Any visible-line algorithm can be easily adapted to show hidden lines as dotted, as dashed, of lower intensity, or with some other rendering style supported by the display device. The program then outputs the hidden line segments in the line style selected, instead of suppressing them. In contrast, Appel, Rohlf, and Stein [APPE79] describe an algorithm for rendering haloed lines, as shown in Fig. 15.20. Each line is surrounded on both sides by a halo that obscures those parts of lines passing behind it. This algorithm, unlike those

Samsung/Google, EX1027, p. 678

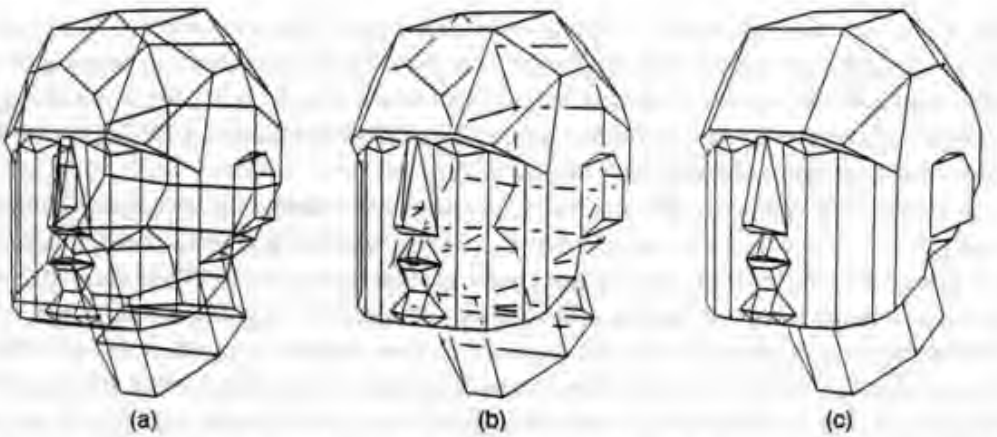


Fig. 15.20 Three heads rendered (a) without hidden lines eliminated, (b) with hidden lines haloed, and (c) with hidden lines eliminated. (Courtesy of Arthur Appel, IBM T.J. Watson Research Center.)

discussed previously, does not require each line to be part of an opaque polygonal face. Lines that pass behind others are obscured only around their intersection on the view plane. The algorithm intersects each line with those passing in front of it, keeps track of those sections that are obscured by halos, and draws the visible sections of each line after the intersections have been calculated. If the halos are wider than the spacing between lines, then an effect similar to conventional hidden-line elimination is achieved, except that a line's halo extends outside a polygon of which it may be an edge.

In the rest of this chapter, we discuss the rich variety of algorithms developed for visible-surface determination. We concentrate here on computing which parts of an object's surfaces are visible, leaving the determination of surface color to Chapter 16. In describing each algorithm, we emphasize its application to polygons, but point out when it can be generalized to handle other objects.

15.4 THE *z*-BUFFER ALGORITHM

The *z-buffer* or *depth-buffer* image-precision algorithm, developed by Catmull [CATM74b], is one of the simplest visible-surface algorithms to implement in either software or hardware. It requires that we have available not only a frame buffer F in which color values are stored, but also a *z-buffer* Z, with the same number of entries, in which a z-value is stored for each pixel. The z-buffer is initialized to zero, representing the z-value at the back clipping plane, and the frame buffer is initialized to the background color. The largest value that can be stored in the z-buffer represents the z of the front clipping plane. Polygons are scan-converted into the frame buffer in arbitrary order. During the scan-conversion process, if the polygon point being scan-converted at (x, y) is no farther from the viewer than is the point whose color and depth are currently in the buffers, then the new point's color and depth replace the old values. The pseudocode for the z-buffer algorithm is shown in Fig. 15.21. The WritePixel and ReadPixel procedures introduced in Chapter 3 are supplemented here by WriteZ and ReadZ procedures that write and read the z-buffer.

Samsung/Google, EX1027, p. 679

```
void zBuffer(void)
{
    int x, y;

    for (y = 0; y < YMAX; y++) {        /* Clear frame buffer and z-buffer */
        for (x = 0; x < XMAX; x++) {
            WritePixel (x, y, BACKGROUND_VALUE);
            WriteZ (x, y, 0);
        }
    }

    for (each polygon) {                /* Draw polygons */
        for (each pixel in polygon's projection) {
            double pz = polygon's z-value at pixel coords (x, y);
            if (pz >= ReadZ (x, y)) {   /* New point is not farther */
                WriteZ (x, y, pz);
                WritePixel (x, y, polygon's color at pixel coords (x, y));
            }
        }
    }
} /* zBuffer */
```

Fig. 15.21 Pseudocode for the z-buffer algorithm.

No presorting is necessary and no object–object comparisons are required. The entire process is no more than a search over each set of pairs $\{Z_i(x, y), F_i(x, y)\}$ for fixed x and y, to find the largest Z_i. The z-buffer and the frame buffer record the information associated with the largest z encountered thus far for each (x, y). Thus, polygons appear on the screen in the order in which they are processed. Each polygon may be scan-converted one scan line at a time into the buffers, as described in Section 3.6. Figure 15.22 shows the addition of two polygons to an image. Each pixel's shade is shown by its color; its z is shown as a number.

Remembering our discussion of depth coherence, we can simplify the calculation of z for each point on a scan line by exploiting the fact that a polygon is planar. Normally, to calculate z, we would solve the plane equation $Ax + By + Cz + D = 0$ for the variable z:

$$z = \frac{-D - Ax - By}{C}. \tag{15.6}$$

Now, if at (x, y) Eq. (15.6) evaluates to z_1, then at $(x + \Delta x, y)$ the value of z is

$$z_1 - \frac{A}{C}(\Delta x). \tag{15.7}$$

Only one subtraction is needed to calculate $z(x + 1, y)$ given $z(x, y)$, since the quotient A/C is constant and $\Delta x = 1$. A similar incremental calculation can be performed to determine the first value of z on the next scan line, decrementing by B/C for each Δy. Alternatively, if

Samsung/Google, EX1027, p. 680

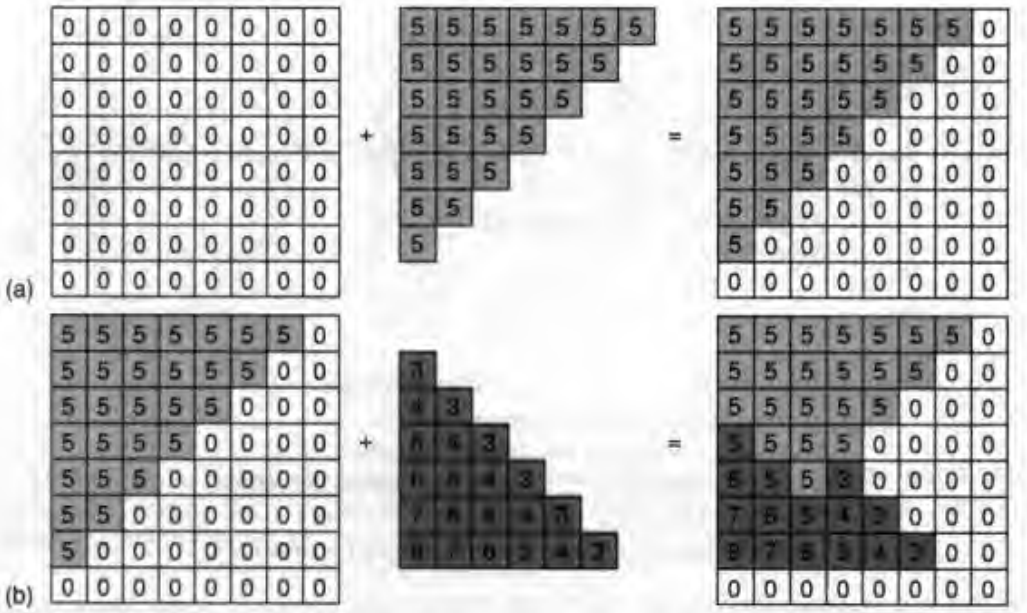


Fig. 15.22 The z-buffer. A pixel's shade is shown by its color, its z value is shown as a number. (a) Adding a polygon of constant z to the empty z-buffer. (b) Adding another polygon that intersects the first.

the surface has not been determined or if the polygon is not planar (see Section 11.1.3), $z(x, y)$ can be determined by interpolating the z coordinates of the polygon's vertices along pairs of edges, and then across each scan line, as shown in Fig. 15.23. Incremental calculations can be used here as well. Note that the color at a pixel does not need to be computed if the conditional determining the pixel's visibility is not satisfied. Therefore, if the shading computation is time consuming, additional efficiency can be gained by performing a rough front-to-back depth sort of the objects to display the closest objects first.

The z-buffer algorithm does not require that objects be polygons. Indeed, one of its most powerful attractions is that it can be used to render any object if a shade and a z-value

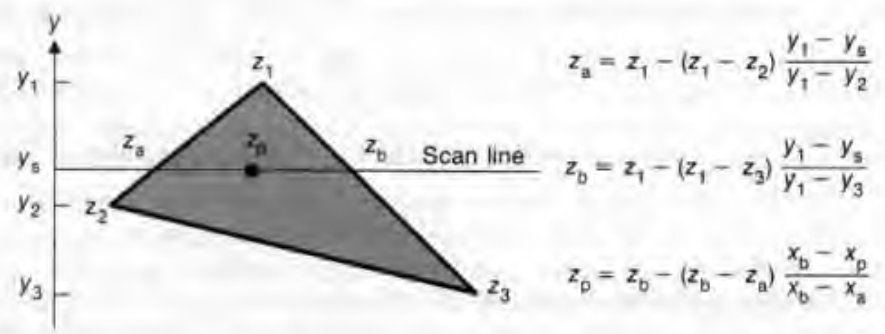


Fig. 15.23 Interpolation of z values along polygon edges and scan lines. z_a is interpolated between z_1 and z_2; z_b between z_1 and z_3; z_p between z_a and z_b.

Samsung/Google, EX1027, p. 681

can be determined for each point in its projection; no explicit intersection algorithms need to be written.

The z-buffer algorithm performs radix sorts in x and y, requiring no comparisons, and its z sort takes only one comparison per pixel for each polygon containing that pixel. The time taken by the visible-surface calculations tends to be independent of the number of polygons in the objects because, on the average, the number of pixels covered by each polygon decreases as the number of polygons in the view volume increases. Therefore, the average size of each set of pairs being searched tends to remain fixed. Of course, it is also necessary to take into account the scan-conversion overhead imposed by the additional polygons.

Although the z-buffer algorithm requires a large amount of space for the z-buffer, it is easy to implement. If memory is at a premium, the image can be scan-converted in strips, so that only enough z-buffer for the strip being processed is required, at the expense of performing multiple passes through the objects. Because of the z-buffer's simplicity and the lack of additional data structures, decreasing memory costs have inspired a number of hardware and firmware implementations of the z-buffer, examples of which are discussed in Chapter 18. Because the z-buffer algorithm operates in image precision, however, it is subject to aliasing. The A-buffer algorithm [CARP84], described in Section 15.7, addresses this problem by using a discrete approximation to unweighted area sampling.

The z-buffer is often implemented with 16- through 32-bit integer values in hardware, but software (and some hardware) implementations may use floating-point values. Although a 16-bit z-buffer offers an adequate range for many CAD/CAM applications, 16 bits do not have enough precision to represent environments in which objects defined with millimeter detail are positioned a kilometer apart. To make matters worse, if a perspective projection is used, the compression of distant z values resulting from the perspective divide has a serious effect on the depth ordering and intersections of distant objects. Two points that would transform to different integer z values if close to the view plane may transform to the same z value if they are farther back (see Exercise 15.13 and [HUGH89]).

The z-buffer's finite precision is responsible for another aliasing problem. Scan-conversion algorithms typically render two different sets of pixels when drawing the common part of two collinear edges that start at different endpoints. Some of those pixels shared by the rendered edges may also be assigned slightly different z values because of numerical inaccuracies in performing the z interpolation. This effect is most noticeable at the shared edges of a polyhedron's faces. Some of the visible pixels along an edge may be part of one polygon, while the rest come from the polygon's neighbor. The problem can be fixed by inserting extra vertices to ensure that vertices occur at the same points along the common part of two collinear edges.

Even after the image has been rendered, the z-buffer can still be used to advantage. Since it is the only data structure used by the visible-surface algorithm proper, it can be saved along with the image and used later to merge in other objects whose z can be computed. The algorithm can also be coded so as to leave the z-buffer contents unmodified when rendering selected objects. If the z-buffer is masked off this way, then a single object can be written into a separate set of overlay planes with hidden surfaces properly removed (if the object is a single-valued function of x and y) and then erased without affecting the contents of the z-buffer. Thus, a simple object, such as a ruled grid, can be moved about the

Samsung/Google, EX1027, p. 682

image in x, y, and z, to serve as a "3D cursor" that obscures and is obscured by the objects in the environment. Cutaway views can be created by making the z-buffer and frame-buffer writes contingent on whether the z value is behind a cutting plane. If the objects being displayed have a single z value for each (x, y), then the z-buffer contents can also be used to compute area and volume. Exercise 15.25 explains how to use the z-buffer for picking.

Rossignac and Requicha [ROSS86] discuss how to adapt the z-buffer algorithm to handle objects defined by CSG. Each pixel in a surface's projection is written only if it is both closer in z and on a CSG object constructed from the surface. Instead of storing only the point with closest z at each pixel, Atherton suggests saving a list of all points, ordered by z and accompanied by each surface's identity, to form an *object buffer* [ATHE81]. A postprocessing stage determines how the image is displayed. A variety of effects, such as transparency, clipping, and Boolean set operations, can be achieved by processing each pixel's list, without any need to re–scan convert the objects.

15.5 LIST-PRIORITY ALGORITHMS

List-priority algorithms determine a visibility ordering for objects ensuring that a correct picture results if the objects are rendered in that order. For example, if no object overlaps another in z, then we need only to sort the objects by increasing z, and to render them in that order. Farther objects are obscured by closer ones as pixels from the closer polygons overwrite those of the more distant ones. If objects overlap in z, we may still be able to determine a correct order, as in Fig. 15.24(a). If objects cyclically overlap each other, as Fig. 15.24(b) and (c), or penetrate each other, then there is no correct order. In these cases, it will be necessary to split one or more objects to make a linear order possible.

List-priority algorithms are hybrids that combine both object-precision and image-precision operations. Depth comparisons and object splitting are done with object precision. Only scan conversion, which relies on the ability of the graphics device to overwrite the pixels of previously drawn objects, is done with image precision. Because the list of sorted objects is created with object precision, however, it can be redisplayed correctly at any resolution. As we shall see, list-priority algorithms differ in how they determine the sorted order, as well as in which objects get split, and when the splitting occurs. The sort need not be on z, some objects may be split that neither cyclically overlap nor penetrate others, and the splitting may even be done independent of the viewer's position.

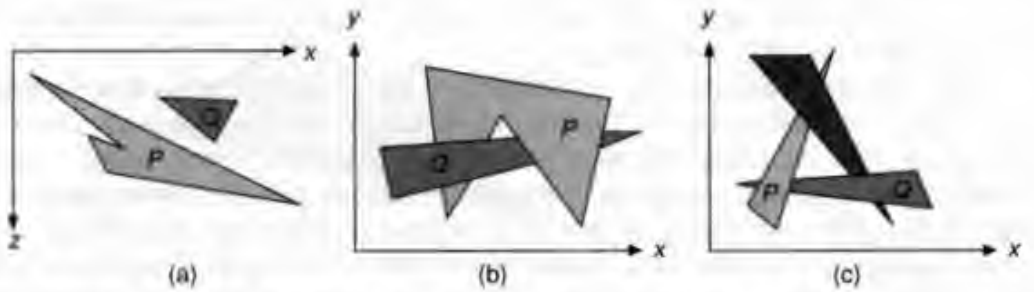


Fig. 15.24 Some cases in which z extents of polygons overlap.

Samsung/Google, EX1027, p. 683

15.5.1 The Depth-Sort Algorithm

The basic idea of the *depth-sort algorithm*, developed by Newell, Newell, and Sancha [NEWE72], is to paint the polygons into the frame buffer in order of decreasing distance from the viewpoint. Three conceptual steps are performed:

1. Sort all polygons according to the smallest (farthest) z coordinate of each
2. Resolve any ambiguities this may cause when the polygons' z extents overlap, splitting polygons if necessary
3. Scan convert each polygon in ascending order of smallest z coordinate (i.e., back to front).

Consider the use of explicit priority, such as that associated with views in SPHIGS. The explicit priority takes the place of the minimum z value, and there can be no depth ambiguities because each priority is thought of as corresponding to a different plane of constant z. This simplified version of the depth-sort algorithm is often known as the *painter's algorithm*, in reference to how a painter might paint closer objects over more distant ones. Environments whose objects each exist in a plane of constant z, such as those of VLSI layout, cartography, and window management, are said to be $2\frac{1}{2}$D and can be correctly handled with the painter's algorithm. The painter's algorithm may be applied to a scene in which each polygon is not embedded in a plane of constant z by sorting the polygons by their minimum z coordinate or by the z coordinate of their centroid, ignoring step 2. Although scenes can be constructed for which this approach works, it does not in general produce a correct ordering.

Figure 15.24 shows some of the types of ambiguities that must be resolved as part of step 2. How is this done? Let the polygon currently at the far end of the sorted list of polygons be called P. Before this polygon is scan-converted into the frame buffer, it must be tested against each polygon Q whose z extent overlaps the z extent of P, to prove that P cannot obscure Q and that P can therefore be written before Q. Up to five tests are performed, in order of increasing complexity. As soon as one succeeds, P has been shown not to obscure Q and the next polygon Q overlapping P in z is tested. If all such polygons pass, then P is scan-converted and the next polygon on the list becomes the new P. The five tests are

1. Do the polygons' x extents not overlap?
2. Do the polygons' y extents not overlap?
3. Is P entirely on the opposite side of Q's plane from the viewpoint? (This is not the case in Fig. 15.24(a), but is true for Fig. 15.25.)
4. Is Q entirely on the same side of P's plane as the viewpoint? (This is not the case in Fig. 15.24(a), but is true for Fig. 15.26.)
5. Do the projections of the polygons onto the (x, y) plane not overlap? (This can be determined by comparing the edges of one polygon to the edges of the other.)

Exercise 15.6 suggests a way to implement tests 3 and 4.

If all five tests fail, we assume for the moment that P actually obscures Q, and therefore test whether Q can be scan-converted before P. Tests 1, 2, and 5 do not need to be repeated,

Samsung/Google, EX1027, p. 684

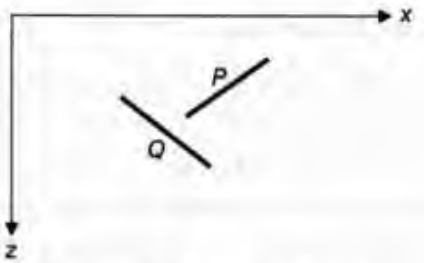


Fig. 15.25 Test 3 is true.

but new versions of tests 3 and 4 are used, with the polygons reversed:

3′. Is Q entirely on the opposite side of P's plane from the viewpoint?

4′. Is P entirely on the same side of Q's plane as the viewpoint?

In the case of Fig. 15.24(a), test 3′ succeeds. Therefore, we move Q to the end of the list and it becomes the new P. In the case of Fig 15.24(b), however, the tests are still inconclusive; in fact, there is no order in which P and Q can be scan-converted correctly. Instead, either P or Q must be split by the plane of the other (see Section 3.14 on polygon clipping, treating the clip edge as a clip plane). The original unsplit polygon is discarded, its pieces are inserted in the list in proper z order, and the algorithm proceeds as before.

Figure 15.24(c) shows a more subtle case. It is possible for P, Q, and R to be oriented such that each polygon can always be moved to the end of the list to place it in the correct order relative to one, but not both, of the other polygons. This would result in an infinite loop. To avoid looping, we must modify our approach by marking each polygon that is moved to the end of the list. Then, whenever the first five tests fail and the current polygon Q is marked, we do not try tests 3′ and 4′. Instead, we split either P or Q (as if tests 3′ and 4′ had both failed) and reinsert the pieces.

Can two polygons fail all the tests even when they are already ordered correctly? Consider P and Q in Fig. 15.27(a). Only the z coordinate of each vertex is shown. With P and Q in their current position, both the simple painter's algorithm and the full depth-sort algorithm scan convert P first. Now, rotate Q clockwise in its plane until it begins to obscure P, but do not allow P and Q themselves to intersect, as shown in Fig. 15.27(b). (You can do this nicely using your hands as P and Q, with your palms facing you.) P and Q

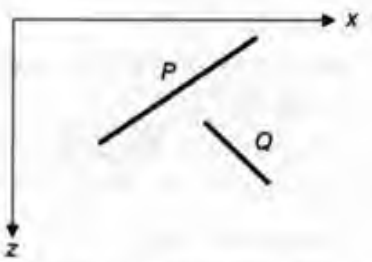


Fig. 15.26 Test 3 is false, but test 4 is true.

Samsung/Google, EX1027, p. 685

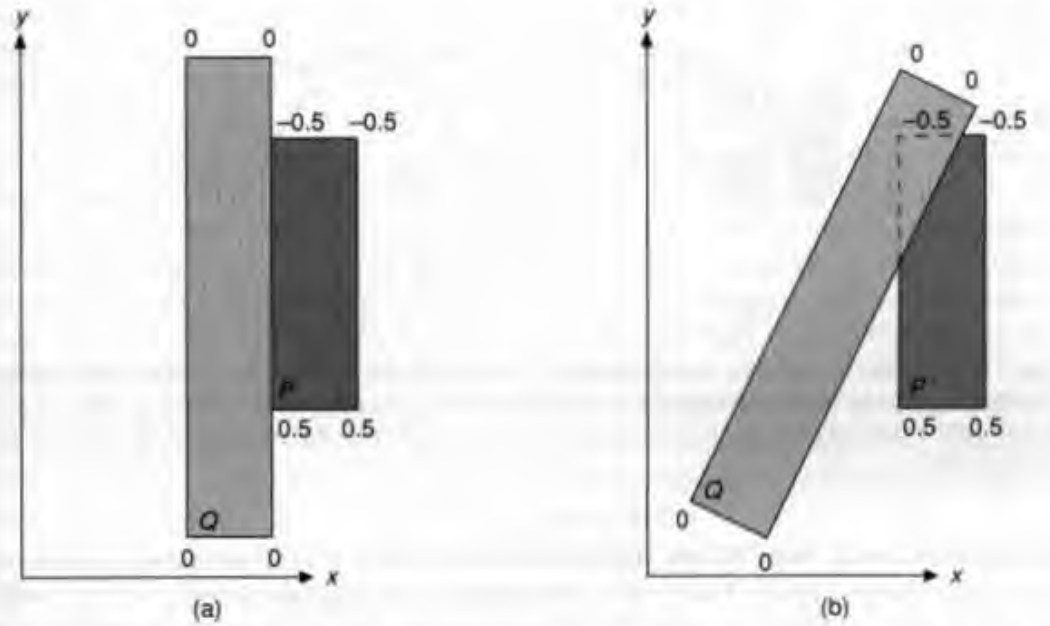


Fig. 15.27 Correctly ordered polygons may be split by the depth-sort algorithm. Polygon vertices are labeled with their z values. (a) Polygons P and Q are scan-converted without splitting. (b) Polygons P and Q fail all five tests even though they are correctly ordered.

have overlapping z extents, so they must be compared. Note that tests 1 and 2 (x and y extent) fail, tests 3 and 4 fail because neither is wholly in one half-space of the other, and test 5 fails because the projections overlap. Since tests 3′ and 4′ also fail, a polygon will be split, even though P can be scan-converted before Q. Although the simple painter's algorithm would correctly draw P first because P has the smallest minimum z coordinate, try the example again with $z = -0.5$ at P's bottom and $z = 0.5$ at P's top.

15.5.2 Binary Space-Partitioning Trees

The binary space-partitioning (BSP) tree algorithm, developed by Fuchs, Kedem, and Naylor [FUCH80; FUCH83], is an extremely efficient method for calculating the visibility relationships among a static group of 3D polygons as seen from an arbitrary viewpoint. It trades off an initial time- and space-intensive preprocessing step against a linear display algorithm that is executed whenever a new viewing specification is desired. Thus, the algorithm is well suited for applications in which the viewpoint changes, but the objects do not.

The BSP tree algorithm is based on the work of Schumacker [SCHU69], who noted that environments can be viewed as being composed of *clusters* (collections of faces), as shown in Fig. 15.28(a). If a plane can be found that wholly separates one set of clusters from another, then clusters that are on the same side of the plane as the eyepoint can obscure, but cannot be obscured by, clusters on the other side. Each of these sets of clusters can be recursively subdivided if suitable separating planes can be found. As shown in Fig. 15.28(b), this partitioning of the environment can be represented by a binary tree rooted at

Samsung/Google, EX1027, p. 686

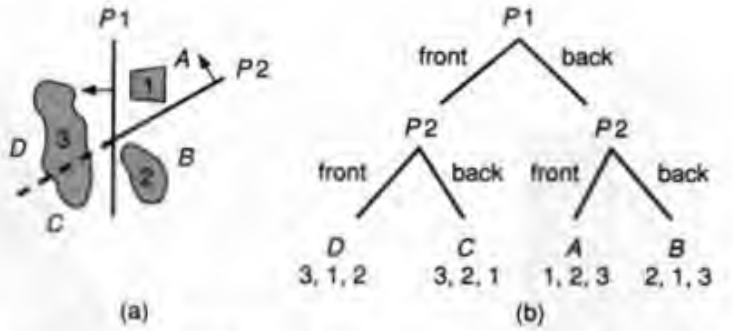


Fig. 15.28 Cluster priority. (a) Clusters 1 through 3 are divided by partitioning planes *P*1 and *P*2, determining regions *A* through *D* in which the eyepoint may be located. Each region has a unique cluster priority. (b) The binary-tree representation of (a). (Based on [SUTH74a].)

the first partitioning plane chosen. The tree's internal nodes are the partitioning planes; its leaves are regions in space. Each region is associated with a unique order in which clusters can obscure one another if the viewpoint is located in that region. Determining the region in which the eyepoint lies involves descending the tree from the root and choosing the left or right child of an internal node by comparing the viewpoint with the plane at that node.

Schumacker selects the faces in a cluster so that a priority ordering can be assigned to each face independent of the viewpoint, as shown in Fig. 15.29. After back-face culling has been performed relative to the viewpoint, a face with a lower priority number obscures a face with a higher number wherever the faces' projections intersect. For any pixel, the correct face to display is the highest-priority (lowest-numbered) face in the highest-priority cluster whose projection covers the pixel. Schumacker used special hardware to determine the frontmost face at each pixel. Alternatively, clusters can be displayed in order of increasing cluster priority (based on the viewpoint), with each cluster's faces displayed in order of their increasing face priority. Rather than take this two-part approach to computing an order in which faces should be scan-converted, the BSP tree algorithm uses a generalization of Schumacker's approach to calculating cluster priority. It is based on the observation that a polygon will be scan-converted correctly (i.e., will not incorrectly overlap or be incorrectly

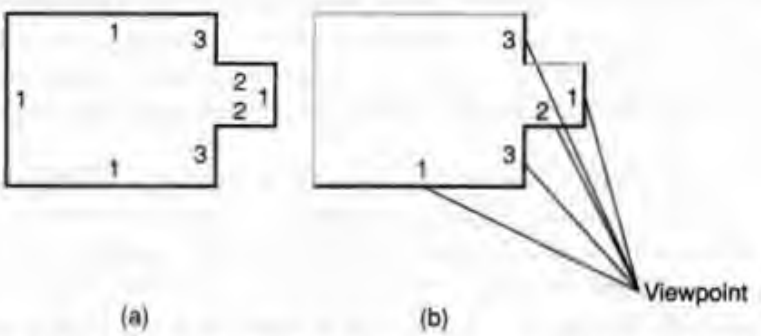


Fig. 15.29 Face priority. (a) Faces in a cluster and their priorities. A lower number indicates a higher priority. (b) Priorities of visible faces. (Based on [SUTH74a].)

Samsung/Google, EX1027, p. 687

overlapped by other polygons) if all polygons on the other side of it from the viewer are scan-converted first, then it, and then all polygons on the same side of it as the viewer. We need to ensure that this is so for each polygon.

The algorithm makes it easy to determine a correct order for scan conversion by building a binary tree of polygons, the *BSP tree*. The BSP tree's root is a polygon selected from those to be displayed; the algorithm works correctly no matter which is picked. The root polygon is used to partition the environment into two half-spaces. One half-space contains all remaining polygons in front of the root polygon, relative to its surface normal; the other contains all polygons behind the root polygon. Any polygon lying on both sides of the root polygon's plane is split by the plane and its front and back pieces are assigned to the appropriate half-space. One polygon each from the root polygon's front and back half-space become its front and back children, and each child is recursively used to divide the remaining polygons in its half-space in the same fashion. The algorithm terminates when each node contains only a single polygon. Pseudocode for the tree-building phase is shown in Fig. 15.30; Fig. 15.31 shows a tree being built.

Remarkably, the BSP tree may be traversed in a modified in-order tree walk to yield a correct priority-ordered polygon list for an arbitrary viewpoint. Consider the root polygon. It divides the remaining polygons into two sets, each of which lies entirely on one side of the root's plane. Thus, the algorithm needs to only guarantee that the sets are displayed in the correct relative order to ensure both that one set's polygons do not interfere with the other's and that the root polygon is displayed properly and in the correct order relative to the others. If the viewer is in the root polygon's front half-space, then the algorithm must first display all polygons in the root's rear half-space (those that could be obscured by the root), then the root, and finally all polygons in its front half-space (those that could obscure the root). Alternatively, if the viewer is in the root polygon's rear half-space, then the algorithm must first display all polygons in the root's front half-space, then the root, and finally all polygons in its rear half-space. If the polygon is seen on edge, either display order suffices. Back-face culling may be accomplished by not displaying a polygon if the eye is in its rear half-space. Each of the root's children is recursively processed by this algorithm. Pseudocode for displaying a BSP tree is shown in Fig. 15.32; Fig. 15.33 shows how the tree of Fig.15.31 (c) is traversed for two different projections.

Each polygon's plane equation can be transformed as it is considered, and the polygon's vertices can be transformed by the displayPolygon routine. The BSP tree can also assist in 3D clipping. Any polygon whose plane does not intersect the view volume has one subtree lying entirely outside of the view volume that does not need to be considered further.

Which polygon is selected to serve as the root of each subtree can have a significant effect on the algorithm's performance. Ideally, the polygon selected should cause the fewest splits among all its descendants. A heuristic that is easier to satisfy is to select the polygon that splits the fewest of its children. Experience shows that testing just a few (five or six) polygons and picking the best provides a good approximation to the best case [FUCH83].

Like the depth-sort algorithm, the BSP tree algorithm performs intersection and sorting entirely at object precision, and relies on the image-precision overwrite capabilities of a raster device. Unlike depth sort, it performs all polygon splitting during a pre-processing step that must be repeated only when the environment changes. Note that more

Samsung/Google, EX1027, p. 688

```
typedef struct {
    polygon root;
    BSP_tree *backChild, *frontChild;
} BSP_tree;

BSP_tree *BSP_makeTree (polygon *polyList)
{
    polygon root;
    polygon *backList, *frontList;
    polygon p, backPart, frontPart;      /* We assume each polygon is convex. */

    if (polyList == NULL)
        return NULL;
    else {
        root = BSP_selectAndRemovePoly (&polyList);
        backList = NULL;
        frontList = NULL;
        for (each remaining polygon p in polyList) {
            if (polygon p in front of root)
                BSP_addToList (p, &frontList);
            else if (polygon p in back of root)
                BSP_addToList (p, &backList);
            else {                       /* Polygon p must be split. */
                BSP_splitPoly (p, root, &frontPart, &backPart);
                BSP_addToList (frontPart, &frontList);
                BSP_addToList (backPart, &backList);
            }
        }
        return BSP_combineTree (BSP_makeTree (frontList),
                                root,
                                BSP_makeTree (backList));
    }
}  /* BSP_makeTree */
```

Fig. 15.30 Pseudocode for building a BSP tree.

polygon splitting may occur than in the depth-sort algorithm.

List-priority algorithms allow the use of hardware polygon scan converters that are typically much faster than are those that check the z at each pixel. The depth-sort and BSP tree algorithms display polygons in a back-to-front order, possibly obscuring more distant ones later. Thus, like the z-buffer algorithm, shading calculations may be computed more than once for each pixel. Alternatively, polygons can instead be displayed in a front-to-back order, and each pixel in a polygon can be written only if it has not yet been.

If a list-priority algorithm is used for hidden-line removal, special attention must be paid to the new edges introduced by the subdivision process. If these edges are

Samsung/Google, EX1027, p. 689

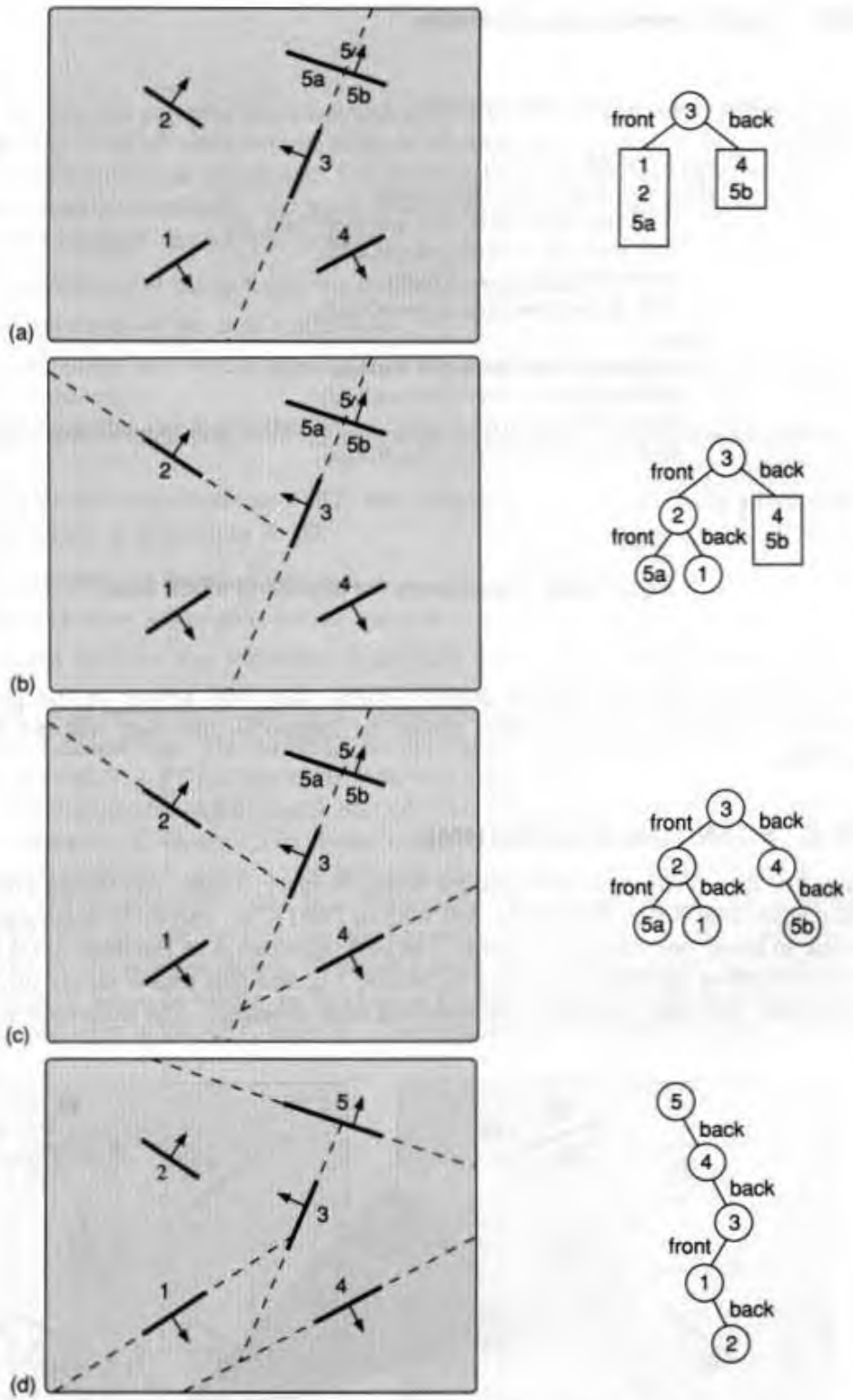


Fig. 15.31 BSP trees. (a) Top view of scene with BSP tree before recursion with polygon 3 as root. (b) After building left subtree. (c) Complete tree. (d) Alternate tree with polygon 5 as root. (Based on [FUCH83].)

Samsung/Google, EX1027, p. 690

```
void BSP_displayTree (BSP_tree *tree)
{
    if (tree != NULL) {
        if (viewer is in front of tree->root) {
            /* Display back child, root, and front child. */
            BSP_displayTree (tree->backChild);
            displayPolygon (tree->root);
            BSP_displayTree (tree->frontChild);
        } else {
            /* Display front child, root, and back child. */
            BSP_displayTree (tree->frontChild);
            displayPolygon (tree->root);      /* Only if back-face culling not desired */
            BSP_displayTree (tree->backChild);
        }
    }
} /* BSP_displayTree */
```

Fig. 15.32 Pseudocode for displaying a BSP tree.

scan-converted like the original polygon edges, they will appear in the picture as unwelcome artifacts, and they thus should be flagged so that they will not be scan-converted.

15.6 SCAN-LINE ALGORITHMS

Scan-line algorithms, first developed by Wylie, Romney, Evans, and Erdahl [WYLI67], Bouknight [BOUK70a; BOUK70b], and Watkins [WATK70], operate at image precision to create an image one scan line at a time. The basic approach is an extension of the polygon scan-conversion algorithm described in Section 3.6, and thus uses a variety of forms of coherence, including scan-line coherence and edge coherence. The difference is that we

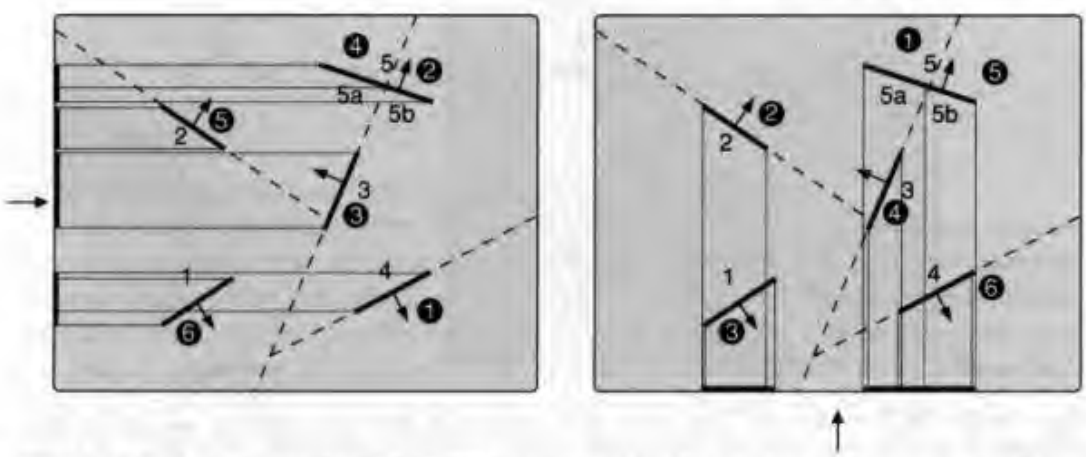


Fig. 15.33 Two traversals of the BSP tree corresponding to two different projections. Projectors are shown as thin lines. White numbers indicate drawing order.

Samsung/Google, EX1027, p. 691

deal not with just one polygon, but rather with a set of polygons. The first step is to create an *edge table* (ET) for all nonhorizontal edges of all polygons projected on the view plane. As before, horizontal edges are ignored. Entries in the ET are sorted into buckets based on each edge's smaller y coordinate, and within buckets are ordered by increasing x coordinate of their lower endpoint. Each entry contains

1. The x coordinate of the end with the smaller y coordinate
2. The y coordinate of the edge's other end
3. The x increment, Δx, used in stepping from one scan line to the next (Δx is the inverse slope of the edge)
4. The polygon identification number, indicating the polygon to which the edge belongs.

Also required is a *polygon table* (PT) that contains at least the following information for each polygon, in addition to its ID:

1. The coefficients of the plane equation
2. Shading or color information for the polygon
3. An in–out Boolean flag, initialized to *false* and used during scan-line processing.

Figure 15.34 shows the projection of two triangles onto the (x, y) plane; hidden edges are shown as dashed lines. The sorted ET for this figure contains entries for AB, AC, FD, FE, CB, and DE. The PT has entries for ABC and DEF.

The *active-edge table* (AET) used in Section 3.6 is needed here also. It is always kept in order of increasing x. Figure 15.35 shows ET, PT, and AET entries. By the time the algorithm has progressed upward to the scan line $y = \alpha$, the AET contains AB and AC, in that order. The edges are processed from left to right. To process AB, we first invert the in–out flag of polygon ABC. In this case, the flag becomes *true*; thus, the scan is now "in" the polygon, so the polygon must be considered. Now, because the scan is "in" only one polygon (ABC), it must be visible, so the shading for ABC is applied to the *span* from edge AB to the next edge in the AET, edge AC. This is an instance of span coherence. At this

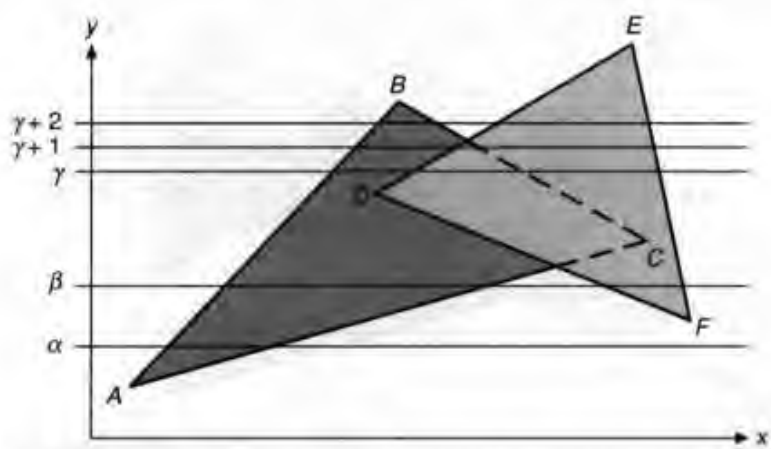


Fig. 15.34 Two polygons being processed by a scan-line algorithm.

Samsung/Google, EX1027, p. 692

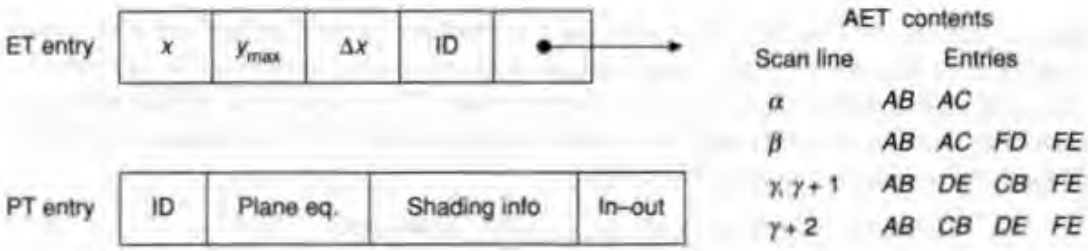


Fig. 15.35 ET, PT, AET for the scan-line algorithm.

edge the flag for *ABC* is inverted to false, so that the scan is no longer "in" any polygons. Furthermore, because *AC* is the last edge in the AET, the scan-line processing is completed. The AET is updated from the ET and is again ordered on x because some of its edges may have crossed, and the next scan line is processed.

When the scan line $y = \beta$ is encountered, the ordered AET is *AB*, *AC*, *FD*, and *FE*. Processing proceeds much as before. There are two polygons on the scan line, but the scan is "in" only one polygon at a time.

For scan line $y = \gamma$, things are more interesting. Entering *ABC* causes its flag to become *true*. *ABC*'s shade is used for the span up to the next edge, *DE*. At this point, the flag for *DEF* also becomes *true*, so the scan is "in" two polygons. (It is useful to keep an explicit list of polygons whose in–out flag is *true*, and also to keep a count of how many polygons are on the list.) We must now decide whether *ABC* or *DEF* is closer to the viewer, which we determine by evaluating the plane equations of both polygons for z at $y = \gamma$ and with x equal to the intersection of $y = \gamma$ with edge *DE*. This value of x is in the AET entry for *DE*. In our example, *DEF* has a larger z and thus is visible. Therefore, assuming nonpenetrating polygons, the shading for *DEF* is used for the span to edge *CB*, at which point *ABC*'s flag becomes *false* and the scan is again "in" only one polygon *DEF* whose shade continues to be used up to edge *FE*. Figure 15.36 shows the relationship of the two polygons and the $y = \gamma$ plane; the two thick lines are the intersections of the polygons with the plane.

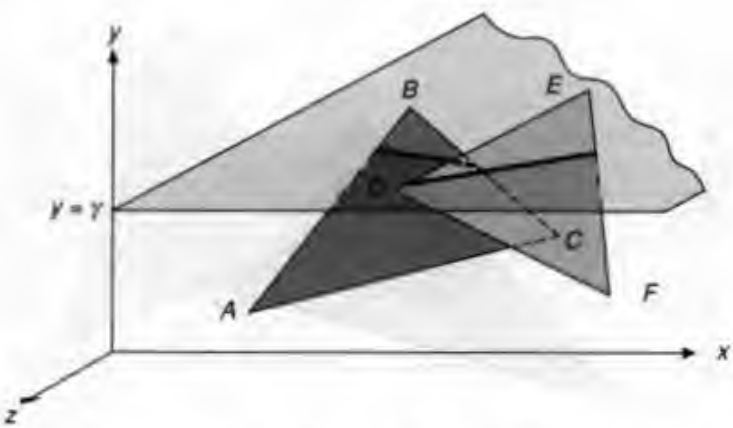


Fig. 15.36 Intersections of polygons *ABC* and *DEF* with the plane $y = \gamma$.

Samsung/Google, EX1027, p. 693

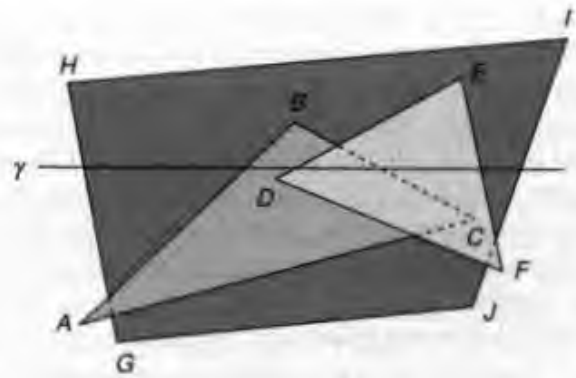


Fig. 15.37 Three nonpenetrating polygons. Depth calculations do not need to be made when scan line γ leaves the obscured polygon *ABC*, since nonpenetrating polygons maintain their relative *z* order.

Suppose there is a large polygon *GHIJ* behind both *ABC* and *DEF*, as in Fig. 15.37. Then, when the $y = \gamma$ scan line comes to edge *CB*, the scan is still "in" polygons *DEF* and *GHIJ*, so depth calculations are performed again. These calculations can be avoided, however, if we assume that none of the polygons penetrate another. This assumption means that, when the scan leaves *ABC*, the depth relationship between *DEF* and *GHIJ* cannot change, and *DEF* continues to be in front. Therefore, depth computations are unnecessary when the scan leaves an obscured polygon, and are required only when it leaves an obscuring polygon.

To use this algorithm properly for penetrating polygons, as shown in Fig. 15.38, we break up *KLM* into *KLL'M'* and *L'MM'*, introducing the *false edge M'L'*. Alternatively, the algorithm can be modified to find the point of penetration on a scan line as the scan line is processed.

Another modification to this algorithm uses *depth coherence*. Assuming that polygons do not penetrate each other, Romney noted that, if the same edges are in the AET on one scan line as are on the immediately preceding scan line, and if they are in the same order, then no changes in depth relationships have occurred on any part of the scan line and no new depth computations are needed [ROMN68]. The record of visible spans on the previous scan line then defines the spans on the current scan line. Such is the case for scan lines $y = \gamma$

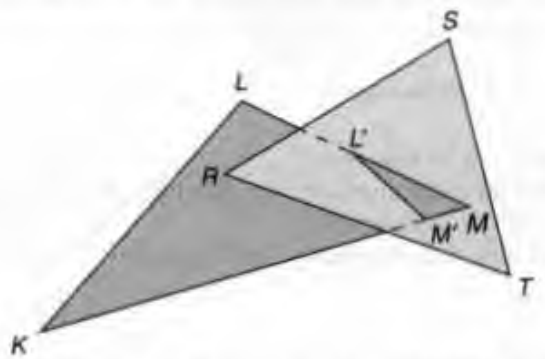


Fig. 15.38 Polygon *KLM* pierces polygon *RST* at the line *L'M'*.

Samsung/Google, EX1027, p. 694

and $y = \gamma + 1$ in Fig.15.34, for both of which the spans from *AB* to *DE* and from *DE* to *FE* are visible. The depth coherence in this figure is lost, however, as we go from $y = \gamma + 1$ to $y = \gamma + 2$, because edges *DE* and *CB* change order in the AET (a situation that the algorithm must accommodate). The visible spans therefore change and, in this case, become *AB* to *CB* and *DE* to *FE*. Hamlin and Gear [HAML77] show how depth coherence can sometimes be maintained even when edges do change order in the AET.

We have not yet discussed how to treat the background. The simplest way is to initialize the frame buffer to the background color, so the algorithm needs to process only scan lines that intersect edges. Another way is to include in the scene definition a large enough polygon that is farther back than any others are, is parallel to the projection plane, and has the desired shading. A final alternative is to modify the algorithm to place the background color explicitly into the frame buffer whenever the scan is not "in" any polygon.

Although the algorithms presented so far deal with polygons, the scan-line approach has been used extensively for more general surfaces, as described in Section 15.9. To accomplish this, the ET and AET are replaced by a *surface table* and *active-surface table*, sorted by the surfaces' (x, y) extents. When a surface is moved from the surface table to the active-surface table, additional processing may be performed. For example, the surface may be decomposed into a set of approximating polygons, which would then be discarded when the scan leaves the surface's y extent; this eliminates the need to maintain all surface data throughout the rendering process. Pseudocode for this general scan-line algorithm is shown in Fig. 15.39. Atherton [ATHE83] discusses a scan-line algorithm that renders polygonal objects combined using the regularized Boolean set operations of constructive solid geometry.

A scan-line approach that is appealing in its simplicity uses a z-buffer to resolve the visible-surface problem [MYER75]. A single-scan-line frame buffer and z-buffer are cleared for each new scan line and are used to accumulate the spans. Because only one scan line of storage is needed for the buffers, extremely high-resolution images are readily accommodated.

```
add surfaces to surface table;
initialize active-surface table;

for (each scan line) {

    update active-surface table;

    for (each pixel on scan line) {
        determine surfaces in active-surface table that project to pixel;
        find closest such surface;
        determine closest surface's shade at pixel;
    }
}
```

Fig. 15.39 Pseudocode for a general scan-line algorithm.

Samsung/Google, EX1027, p. 695

Crocker [CROC84] uses a scan-line z-buffer to exploit what he calls *invisibility coherence*, the tendency for surfaces that are invisible on one scan line to be invisible on the next. When the active-surface table is made up for a given scan line, a separate *invisible-surface table* is also built. A surface is added to the invisible-surface table if its maximum z value for the current scan line is less than the z values in the previous line's z buffer at the surface's minimum and maximum x values. For example, given the cube and contained triangle shown in Fig 15.40(a), the triangle and the contents of the previous scan line's z-buffer projected onto the (x, z) plane are shown in Fig. 15.40(b). The triangle's z_{max} is less than the previous scan line's z-buffer values at the triangle's x_{min} and x_{max}, so the triangle is added to the invisible-surface table. Placing a surface in the invisible-surface table eliminates it from much of the visible-surface processing. Some surfaces in the invisible-surface table may not belong there. To remedy this, as each pixel on the current scan line is processed, surfaces are removed from the invisible-surface table and are added to the active-surface table if their maximum z value is greater than the z value of what is currently determined to be the visible surface at the pixel. For example, even though the triangle shown in Fig. 15.40(c) was placed in the invisible-surface table, it is actually visible because the cube has been clipped, and it will be removed and added to the active-surface table.

Sechrest and Greenberg [SECH82] have developed an object-precision algorithm for nonintersecting polygons that is somewhat in the spirit of a scan-line algorithm. Their algorithm relies on the fact that the visibility of edges can change only at vertices and edge crossings. It sorts vertices and edge crossings by y, effectively dividing up the scene into horizontal bands inside of which the visibility relationships are constant (see Fig. 15.41). Object-precision coordinates of the edge segments visible in each strip are output as the strip is processed and are supplemented with information from which the contours of visible polygons can be reconstructed for scan conversion. Initially, only vertices that are local minima are inspected in sorted order. An AET is kept, and is modified whenever a vertex is encountered in the scan. Edge crossings are determined on the fly by testing only the active edges for intersections.

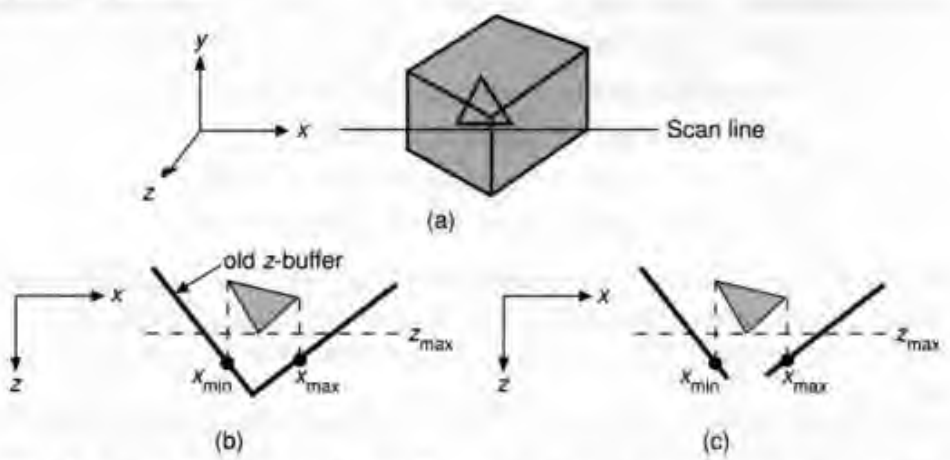


Fig. 15.40 Invisibility coherence. (a) Triangle in a box. (b) Triangle is correctly placed in invisible table. (c) Triangle is incorrectly placed in invisible-surface table. (Based on [CROC84].)

Samsung/Google, EX1027, p. 696

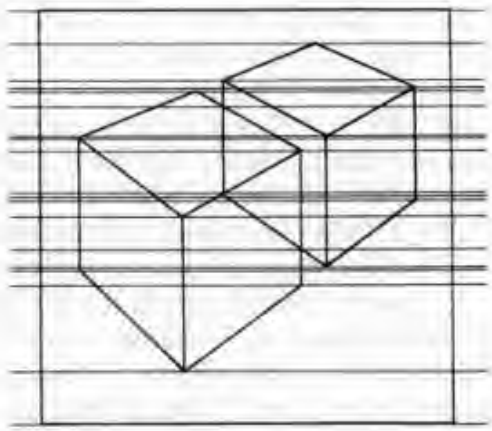


Fig. 15.41 The Sechrest and Greenberg object-precision algorithm divides the picture plane into horizontal strips at vertices and edge crossings. (Courtesy of Stuart Sechrest and Donald P. Greenberg, Program of Computer Graphics, Cornell Univeristy, 1982.)

15.7 AREA-SUBDIVISION ALGORITHMS

Area-subdivision algorithms all follow the divide-and-conquer strategy of spatial partitioning in the projection plane. An area of the projected image is examined. If it is easy to decide which polygons are visible in the area, they are displayed. Otherwise, the area is subdivided into smaller areas to which the decision logic is applied recursively. As the areas become smaller, fewer polygons overlap each area, and ultimately a decision becomes possible. This approach exploits *area coherence*, since sufficiently small areas of an image will be contained in at most a single visible polygon.

15.7.1 Warnock's Algorithm

The area-subdivision algorithm developed by Warnock [WARN69] subdivides each area into four equal squares. At each stage in the recursive-subdivision process, the projection of each polygon has one of four relationships to the area of interest (see Fig. 15.42):

1. *Surrounding polygons* completely contain the (shaded) area of interest (Fig. 15.42a)
2. *Intersecting polygons* intersect the area (Fig. 15.42b)
3. *Contained polygons* are completely inside the area (Fig. 15.42c)
4. *Disjoint polygons* are completely outside the area (Fig. 15.42d).

Disjoint polygons clearly have no influence on the area of interest. The part of an intersecting polygon that is outside the area is also irrelevant, whereas the part of an intersecting polygon that is interior to the area is the same as a contained polygon and can be treated as such.

In four cases, a decision about an area can be made easily, so the area does not need to be divided further to be conquered:

1. All the polygons are disjoint from the area. The background color can be displayed in the area.

Samsung/Google, EX1027, p. 697

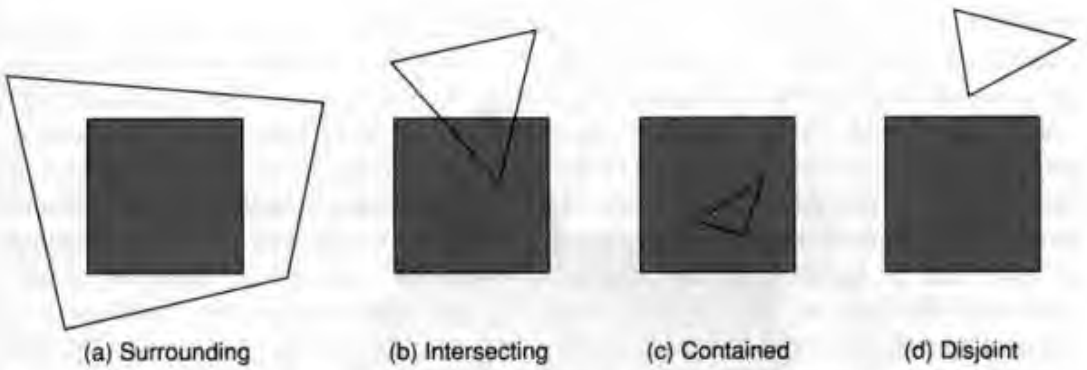


Fig. 15.42 Four relations of polygon projections to an area element: (a) surrounding, (b) intersecting, (c) contained, and (d) disjoint.

2. There is only one intersecting or only one contained polygon. The area is first filled with the background color, and then the part of the polygon contained in the area is scan-converted.
3. There is a single surrounding polygon, but no intersecting or contained polygons. The area is filled with the color of the surrounding polygon.
4. More than one polygon is intersecting, contained in, or surrounding the area, but one is a surrounding polygon that is in front of all the other polygons. Determining whether a surrounding polygon is in front is done by computing the z coordinates of the planes of all surrounding, intersecting, and contained polygons at the four corners of the area; if there is a surrounding polygon whose four corner z coordinates are larger (closer to the viewpoint) than are those of any of the other polygons, then the entire area can be filled with the color of this surrounding polygon.

Cases 1, 2, and 3 are simple to understand. Case 4 is further illustrated in Fig. 15.43.

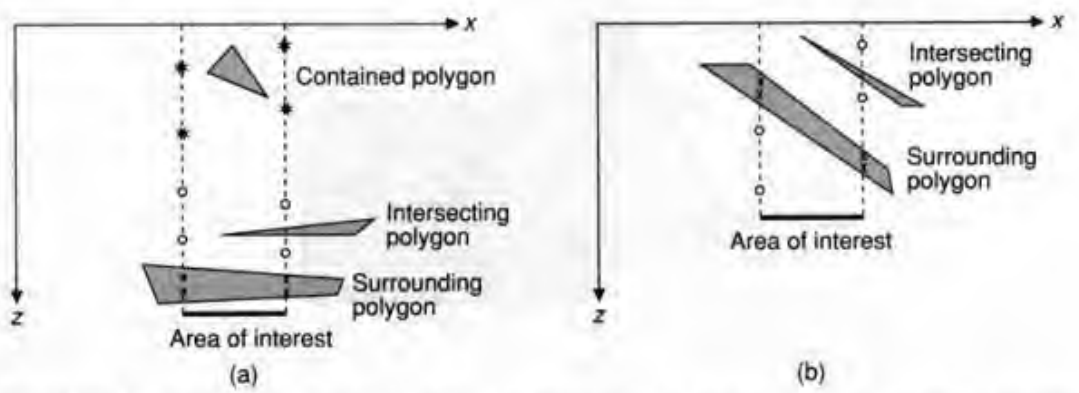


Fig. 15.43 Two examples of case 4 in recursive subdivision. (a) Surrounding polygon is closest at all corners of area of interest. (b) Intersecting polygon plane is closest at left side of area of interest. × marks the intersection of surrounding polygon plane; ○ marks the intersection of intersecting polygon plane; * marks the intersection of contained polygon plane.

Samsung/Google, EX1027, p. 698

In part (a), the four intersections of the surrounding polygon are all closer to the viewpoint (which is at infinity on the $+z$ axis) than are any of the other intersections. Consequently, the entire area is filled with the surrounding polygon's color. In part (b), no decision can be made, even though the surrounding polygon seems to be in front of the intersecting polygon, because on the left the plane of the intersecting polygon is in front of the plane of the surrounding polygon. Note that the depth-sort algorithm accepts this case without further subdivision if the intersecting polygon is wholly on the side of the surrounding polygon that is farther from the viewpoint. Warnock's algorithm, however, always subdivides the area to simplify the problem. After subdivision, only contained and intersecting polygons need to be reexamined: Surrounding and disjoint polygons of the original area are surrounding and disjoint polygons of each subdivided area.

Up to this point, the algorithm has operated at object precision, with the exception of the actual scan conversion of the background and clipped polygons in the four cases. These image-precision scan-conversion operations, however, can be replaced by object-precision operations that output a precise representation of the visible surfaces: either a square of the area's size (cases 1, 3, and 4) or a single polygon clipped to the area, along with its Boolean complement relative to the area, representing the visible part of the background (case 2). What about the cases that are not one of these four? One approach is to stop subdividing when the resolution of the display device is reached. Thus, on a 1024 by 1024 raster display, at most 10 levels of subdivision are needed. If, after this maximum number of subdivisions, none of cases 1 to 4 have occurred, then the depth of all relevant polygons is computed at the center of this pixel-sized, indivisible area. The polygon with the closest z coordinate defines the shading of the area. Alternatively, for antialiasing, several further levels of subdivision can be used to determine a pixel's color by weighting the color of each

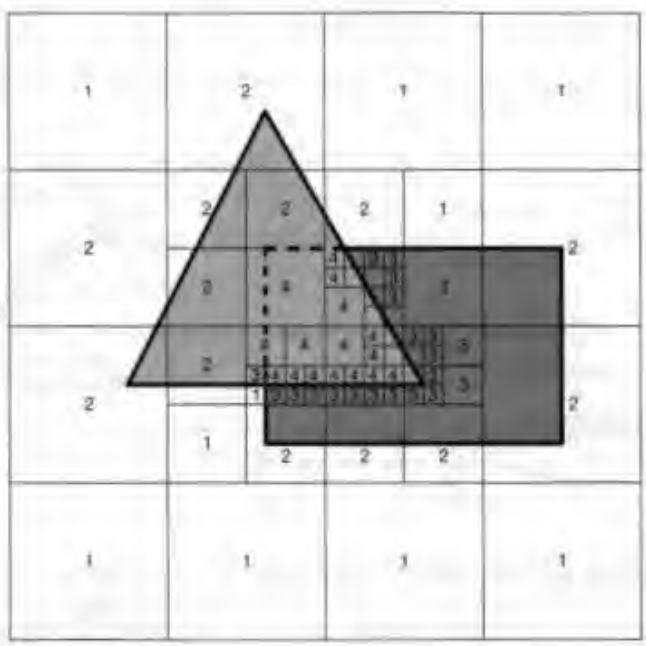


Fig. 15.44 Area subdivision into squares.

Samsung/Google, EX1027, p. 699

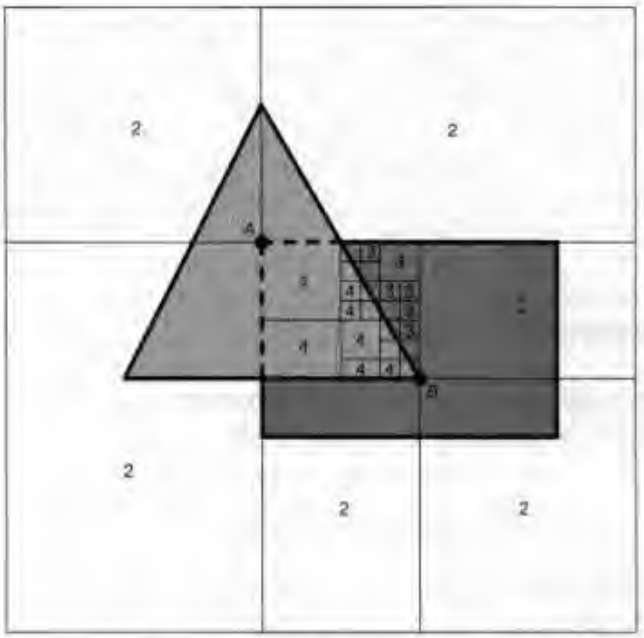


Fig. 15.45 Area subdivision about circled polygon vertices. The first subdivision is at vertex *A*; the second is at vertex *B*.

of its subpixel-sized areas by its size. It is these image-precision operations, performed when an area is not one of the simple cases, that makes this an image-precision approach.

Figure 15.44 shows a simple scene and the subdivisions necessary for that scene's display. The number in each subdivided area corresponds to one of the four cases; in unnumbered areas, none of the four cases are true. Compare this approach to the 2D spatial partitioning performed by quadtrees (Section 12.6.3). An alternative to equal-area subdivision, shown in Fig. 15.45, is to divide about the vertex of a polygon (if there is a vertex in the area) in an attempt to avoid unnecessary subdivisions. Here subdivision is limited to a depth of five for purposes of illustration.

15.7.2 The Weiler–Atherton Algorithm

Warnock's algorithm was forced to use image-precision operations to terminate because it could clip a polygon only against a rectangular area. Another strategy, developed later by Weiler and Atherton [WEIL77], subdivides the screen area along polygon boundaries rather than along rectangle boundaries. This approach requires a powerful clipping algorithm, such as that described in Section 19.1.4, that can clip one concave polygon with holes against another. The first step, not mandatory but useful to improve efficiency, is to sort polygons on some value of z, such as on their nearest z coordinate. The polygon closest to the viewer by this criterion is then used to clip all the polygons, including the clip polygon, into two lists containing the pieces inside and outside the clip polygon. All polygons on the inside list that are behind the clip polygon are then deleted, since they are invisible. If any polygon on the inside list is closer than is the clip polygon, the initial sort did not give a correct priority order (in fact, such a priority order does not exist in the case of cyclic

Samsung/Google, EX1027, p. 700

overlap discussed in Section 15.5). Each such offending polygon is processed recursively to clip the pieces on the inside list against it. When this recursive subdivision is over, the inside list is displayed. The algorithm then continues to process the polygons on the outside list. Clipping is always performed with a copy of one of the original polygons as the clip polygon, never a fragment, since clipping to the original polygon is assumed to be less expensive than clipping to one or more fragmented pieces of the original polygon. Thus,

```
void WA_visibleSurface (void)
{
    polygon *polyList = list of copies of all polygons;
    sort polyList by decreasing value of maximum z;
    clear stack;

    /* Process each remaining polygonal region. */
    while (polyList != NULL)
        WA_subdivide (first polygon on polyList, &polyList);
}   /* WA_visibleSurface */

void WA_subdivide (polygon clipPolygon, polygon **polyList)
{
    polygon *inList;        /* Fragments inside clipPolygon */
    polygon *outList;       /* Fragments outside clipPolygon */

    inList = NULL;
    outList = NULL;

    for (each polygon in *polyList)
        clip polygon to ancestor of clipPolygon, placing inside pieces on
            inList, outside pieces on outList;

    remove polygons behind clipPolygon from inList;

    /* Process incorrectly ordered fragments recursively. */
    for (each polygon in inList that is not on stack and not a part of clipPolygon) {
        push clipPolygon onto stack;
        WA_subdivide (polygon, &inList);
        pop stack;
    }

    /* Display remaining polygons inside clipPolygon. */
    for (each polygon in inList)
        display polygon;

    *polyList = outList;    /* Subtract inList from *polyList. */
}   /* WA_subdivide */
```

Fig. 15.46 Pseudocode for the Weiler–Atherton visible surface algorithm.

Samsung/Google, EX1027, p. 701

when a polygon is clipped, each piece must point back to the original input polygon from which it is derived. As a further efficiency consideration, the clipping algorithm can treat any polygon derived from the clip polygon as a special case and place it on the inside list with no additional testing.

The algorithm uses a stack to handle those cases of cyclic overlap in which one polygon is both in front of and behind another, as in Fig. 15.24(b). The stack contains a list of polygons that are currently in use as clipping polygons, but whose use has been interrupted because of recursive subdivision. If a polygon is found to be in front of the current clip polygon, it is searched for in the stack. If it is on the stack, then no more recursion is necessary since all polygon pieces inside and behind that polygon have already been removed. Pseudocode for the algorithm is shown in Fig. 15.46.

For the simple example of Fig. 15.47, triangle A is used as the first clip polygon because its nearest z coordinate is the largest. A is placed on its own inside list; next, rectangle B is subdivided into two polygons: $B_{in}A$, which is added to the inside list, and $B_{out}A$, which is placed on the outside list. $B_{in}A$ is then removed from the inside list, since it is behind A. Now, since no member of the inside list is closer than A, A is output. $B_{out}A$ is processed next, and is trivially output since it is the only polygon remaining.

Figure 15.48 shows a more complex case in which the original sorted order (or any other order) is incorrect. Part (a) depicts four polygons whose vertices are each marked with their z value. Rectangle A is considered to be closest to the viewer because its maximum z coordinate is greatest. Therefore, in the first call to WA_subdivide, A is used to clip all the polygons, as shown in part (b). The inside list is A, $B_{in}A$, $C_{in}A$, and $D_{in}A$; the outside list is $B_{out}A$, $C_{out}A$, and $D_{out}A$. $B_{in}A$ and $D_{in}A$ are discarded because they are farther back than A is, leaving only A and $C_{in}A$ on the inside list. Since $C_{in}A$ is found to be on the near side of A's plane, however, it is apparent that the polygons were ordered incorrectly. Therefore, recursive subdivision is accomplished by calling WA_subdivide to clip the current inside list against C, the ancestor of the offending polygon, as shown in part (c). The new inside list for this level of recursion is $A_{in}C$ and $C_{in}A$; the new outside list is $A_{out}C$. $A_{in}C$ is removed from the inside list because it is behind C. Only $C_{in}A$ is left on the inside list; since it is a part of the clip polygon, it is displayed. Before returning from the recursive call to WA_subdivide, *polyList* is set to the new outside list containing only $A_{out}C$. Since *polyList* is the caller's *inList*, $A_{out}C$ is displayed next, as shown in part (d), in which displayed

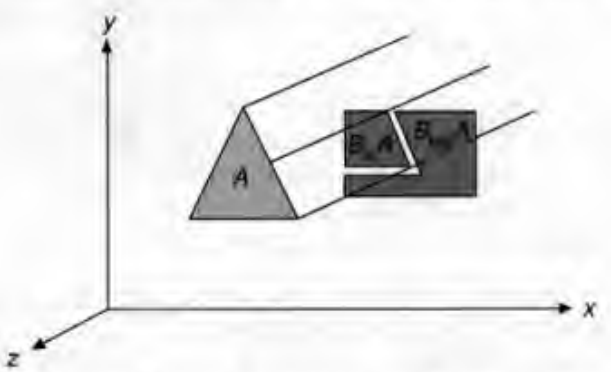


Fig. 15.47 Subdivision of a simple scene using the Weiler–Atherton algorithm.

Samsung/Google, EX1027, p. 702

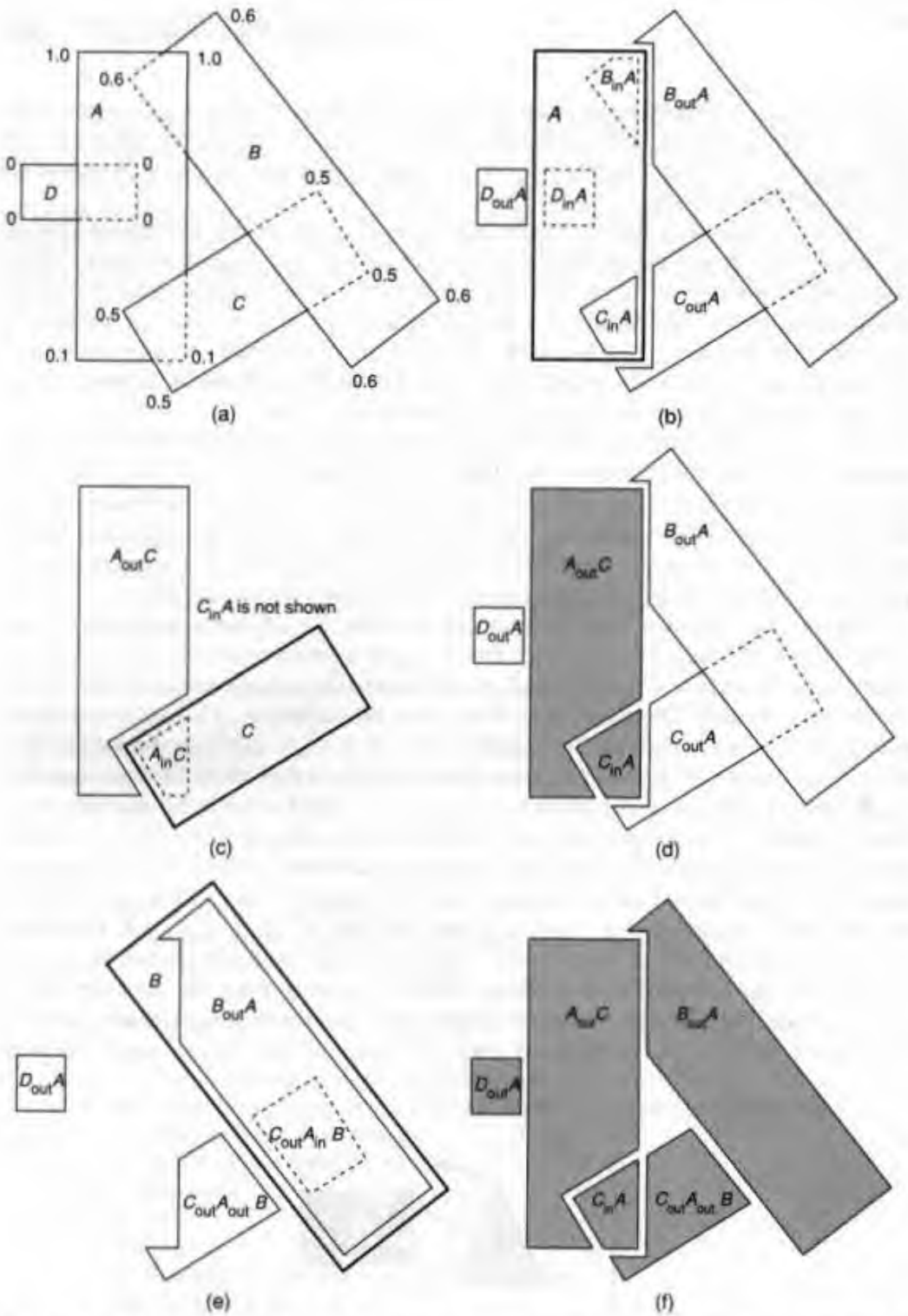


Fig. 15.48 Using the Weiler–Atherton algorithm with recursion. Clip polygon is shown with heavy outline. Displayed polygons are shaded. Numbers are vertex z values. (a) Original scene. (b) Polygons clipped to *A*. (c) *A*'s inside list clipped to *C* during recursive subdivision. (d) Visible fragments inside *A* displayed. (e) Polygons clipped to *B*. (f) All visible fragments displayed at completion.

Samsung/Google, EX1027, p. 703

fragments are shaded. The initial invocation of WA_subdivide then sets *polyList* to its outside list ($B_{out}A$, $C_{out}A$, and $D_{out}A$) and returns.

Next, WA_subdivide is used to process $B_{out}A$ with the new *polyList* containing only $B_{out}A$, $C_{out}A$ and $D_{out}A$. $B_{out}A$'s ancestor B is used to clip these polygons, producing an inside list of $B_{out}A$ and $C_{out}A_{in}B$ and an outside list of $C_{out}A_{out}B$ and $D_{out}A$, as shown in part (e). $C_{out}A_{in}B$ is discarded because it is behind B, leaving only $B_{out}A$ on the inside list, which is then displayed. The polygon list is then set to the new outside list before WA_subdivide returns. Next, WA_subdivide is called once each to process and display $C_{out}A_{out}B$ and $D_{out}A$. The complete set of displayed fragments is shown in part (f).

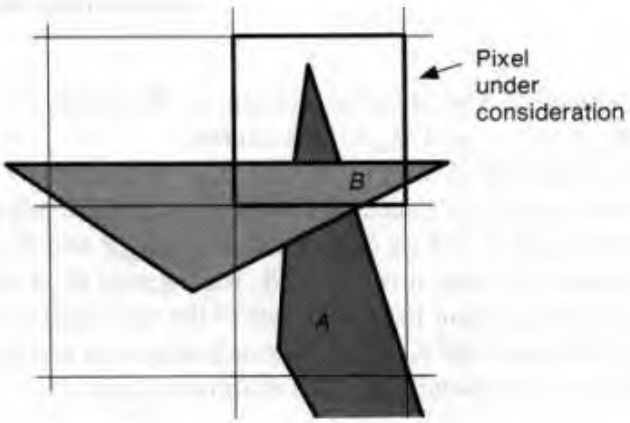
15.7.3 Subpixel Area-Subdivision Algorithms

As is true of any object-precision algorithm, the Weiler–Atherton algorithm potentially requires comparing every polygon with every other. Spatial subdivision, discussed in Section 15.2.5, can reduce the number of comparisons by breaking up the screen into areas (or the environment into volumes) whose objects are processed separately [WEIL77]. Even so, the polygons produced must ultimately be rendered, raising the issue of antialiasing. If spatial subdivision is performed at the subpixel level, however, it can also be used to accomplish antialiasing.

Catmull's object-precision antialiasing algorithm. Catmull [CATM78b] has developed an accurate but expensive scan-line algorithm that does antialiasing by performing object-precision unweighted area sampling at each pixel, using an algorithm similar to the Weiler–Atherton algorithm. In essence, the idea is to perform a full visible-surface algorithm at every pixel, comparing only the polygon fragments that project to each pixel. Catmull first uses the Sutherland–Hodgman algorithm of Section 3.14.1 to clip each polygon intersecting the scan line to the pixels on the line it overlaps, as shown in Fig. 15.49(a). This determines the polygon fragments that project to each pixel, spatially partitioning them by the pixel grid. Then an algorithm similar to the Weiler–Atherton algorithm, but designed for simpler polygon geometry, is executed at each pixel to determine the amount of the pixel covered by the visible part of each fragment, as shown in Fig. 15.49(b). This allows a weighted sum of the visible parts' colors to be computed, and to be used to shade the pixel. Thus, each pixel's shade is determined by box filtering the polygon fragments that project to it.

The A-buffer. Using a full object-precision visible-surface algorithm at each pixel is expensive! Carpenter's A-buffer algorithm [CARP84] addresses this problem by approximating Catmull's per-pixel object-precision area sampling with per-pixel image-precision operations performed on a subpixel grid. It thus provides a discrete approximation to area sampling with a box filter. Polygons are first processed in scan-line order by clipping them to each square pixel they cover. This creates a list of clipped polygon fragments for each pixel. Each fragment is associated with a 4 by 8 bit mask of those parts of the pixel it covers, as shown in Fig. 15.49(c). The bit mask for a fragment is computed by **xor**ing together masks representing each of the fragment's edges. An edge mask has 1s on the edge and to the right of the edge in those rows through which the edge passes, as shown in Fig. 15.49(d). When all polygons intersecting a pixel have been processed, the area-weighted

Samsung/Google, EX1027, p. 704

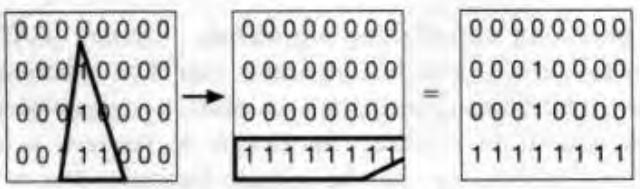


(a)



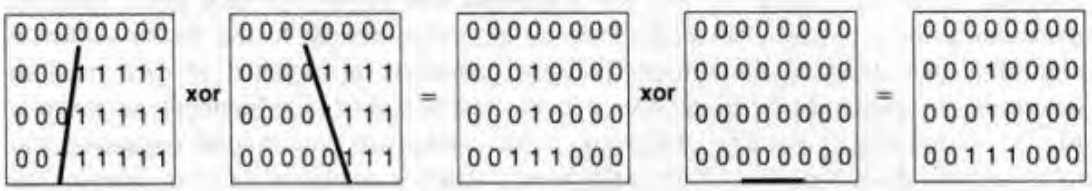
List of subpixel fragments

(b)

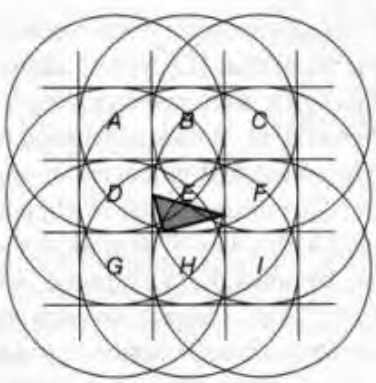


List of subpixel fragments

(c)



(d)



(e)

Samsung/Google, EX1027, p. 705

average of the colors of the pixel's visible surfaces is obtained by selecting fragments in depth-sorted order and using their bit masks to clip those of farther fragments. The bit masks can be manipulated efficiently with Boolean operations. For example, two fragment bit masks can be **and**ed together to determine the overlap between them. The A-buffer algorithm saves only a small amount of additional information with each fragment. For example, it includes the fragment's z extent, but no information about which part of the fragment is associated with these z values. Thus, the algorithm must make assumptions about the subpixel geometry in cases in which fragment bit masks overlap in z. This causes inaccuracies, especially where multiple surfaces intersect in a pixel.

Using precomputed convolution tables for better filtering. Both subpixel area-subdivision algorithms described so far use unweighted area sampling; thus, they restrict a fragment's influence to the single pixel to which it is clipped. If we would like to use filters with wider support, however, we must take into account that each fragment lies within the support of a number of filters positioned over nearby pixels, as shown in Fig. 15.49(e). Abram, Westover, and Whitted [ABRA85] describe how to incorporate better filters in these algorithms by classifying each visible fragment into one of a number of classes, based on the fragment's geometry. For each pixel within whose filter the fragment lies (pixels A through I in Fig 15.49e), the fragment's class and its position relative to the pixel are used to index into a look-up table. The look-up table contains the precomputed convolution of the desired filter kernel with prototype fragments at a set of different positions. The selected entry is multiplied by the fragment's intensity value and is added to an accumulator at that pixel. Those fragments that do not fit into one of the classes are approximated either as sums and differences of simpler fragments or by using bit masks.

15.8 ALGORITHMS FOR OCTREES

Algorithms for displaying octree-encoded objects (see Section 12.6.3) take advantage of the octree's regular structure of nonintersecting cubes. Since the octree is spatially presorted, list-priority algorithms have been developed that yield a correct display order for parallel projections [DOCT81; MEAG82a; GARG86]. In a back-to-front enumeration, nodes are listed in an order in which any node is guaranteed not to obscure any node listed after it. For an orthographic projection, a correct back-to-front enumeration can be determined from the VPN alone. One approach is to display the farthest octant first, then those three neighbors that share a face with the farthest octant in any order, then those three neighbors of the closest octant in any order, then the closest octant. In Fig. 15.50, one such enumeration for a VPN from 0 to V is 0, 1, 2, 4, 3, 5, 6, 7. No node in this enumeration can obscure any node enumerated after it. As each octant is displayed, its descendants are displayed recursively in this order. Furthermore, because each leaf node is a cube, at most three of its faces are visible, the identities of which may also be determined from the VPN.

Fig. 15.49 Subpixel area-subdivision algorithms. (a) Sample pixel contents. (b) Catmull algorithm subpixel geometry. (c) A-buffer algorithm subpixel geometry. (d) A-buffer algorithm subpixel mask for a fragment is computed by **xor**ing together masks for its edges. (e) Abram, Westover, and Whitted algorithm adds polygon's contribution to all pixels it affects. (Part e is based on [ABRA85].)

Samsung/Google, EX1027, p. 706

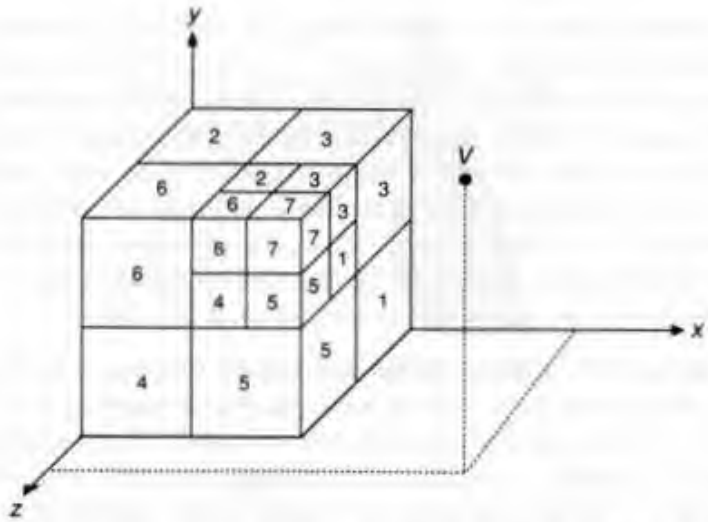


Fig. 15.50 Octree enumeration for back-to-front display. (Node 0 is at the lower-left back corner.) For a VPN from the origin to *V*, nodes may be recursively displayed using several different ordering systems.

Table 15.1 shows eight different back-to-front orders as determined by the signs of the three coordinates of the VPN and the visible octant faces associated with each. (Note that only the first and last octants in each order are fixed.) A positive or negative VPN x coordinate means the right (R) or left (L) face is visible, respectively. Similarly, the y coordinate determines the visibility of the up (U) and down (D) faces, and the z coordinate controls the front (F) and back (B) faces. If any VPN coordinate is zero, then neither of the faces associated with it is visible. Only the VPN's nonzero coordinates are significant in determining an order. Since all octree nodes are identically oriented, the visible faces and their relative polygonal projections for all nodes need to be determined only once. Arbitrary parallel projections can be accommodated by considering the DOP instead of the VPN.

Another approach to back-to-front enumeration for orthographic projections iterates through the octants in slices perpendicular to one of the axes, and in either rows or columns

TABLE 15.1 BACK-TO-FRONT ENUMERATION AND VISIBLE FACES

VPN				
z	y	x	Back-to-front order	Visible faces*
−	−	−	7,6,5,3,4,2,1,0	B,D,L
−	−	+	6,7,4,2,5,3,0,1	B,D,R
−	+	−	5,4,7,1,6,0,3,2	B,U,L
−	+	+	4,5,6,0,7,1,2,3	B,U,R
+	−	−	3,2,1,7,0,6,5,4	F,D,L
+	−	+	2,3,0,6,1,7,4,5	F,D,R
+	+	−	1,0,3,5,2,4,7,6	F,U,L
+	+	+	0,1,2,4,3,5,6,7	F,U,R

*R = right, L = left; U = up; D = down; F = front; B = back.

Samsung/Google, EX1027, p. 707

within each slice. The sign of each component of the VPN determines the direction of iteration along the corresponding octree axis. A positive component indicates increasing order along its axis, whereas a negative component indicates decreasing order. The order in which the axes are used does not matter. For example, in Fig. 15.50, one such enumeration for a VPN with all positive coordinates is 0, 4, 2, 6, 1, 5, 3, 7, varying first z, then y, and then x. This approach is easily generalized to operate on voxel arrays [FRIE85].

It is not necessary to display all an object's voxels, since those that are surrounded entirely by others will ultimately be invisible; more efficient scan conversion can be accomplished by rendering only the voxels on the octree's border [GARG86]. The set of border voxels can be determined using the algorithm presented in Section 12.6.3. Further improvement may be obtained by noting that, even when only border voxels are displayed, some faces may be drawn and then overwritten. Gargantini [GARG86] uses the information obtained during border extraction to identify for each voxel those faces that abut another voxel. These faces need not be drawn, since they will always be obscured. Rather than scan convert each voxel as a small cube, it is also possible to approximate each voxel with a single upright rectangle (a pixel in the limiting case).

Meagher [MEAG82b] describes a front-to-back algorithm that uses the reverse of the back-to-front order described previously. It represents the image being rendered as a quadtree that is initially empty. Each full or partially full octree node is considered in front-to-back order and is compared with the quadtree nodes that its projection intersects. Those octree nodes whose projections intersect only full quadtree nodes are invisible; they and their descendants are not considered further. If a partially full octree node's projection intersects one or more partially full quadtree nodes, then the octree node's children are compared with the children of these quadtree nodes. If a full octree node's projection intersects partially full quadtree nodes, then only these partially full quadtree nodes are further subdivided to determine the previously empty nodes that are covered by the projection. Any empty quadtree node enclosed by a full octree node's projection is shaded with the octree node's value.

As shown in Fig. 15.51, Meagher bounds each octree-node projection with an upright rectangular extent. Any extent needs to be compared with only four of the lowest-level quadtree nodes whose edge size is at least as great as the extent's largest dimension. These

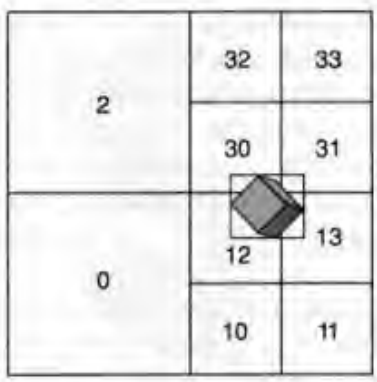


Fig. 15.51 Front-to-back octree scan conversion. Each octree node's projection and its rectangular extent are compared with four quadtree nodes, here 12, 13, 30, and 31.

Samsung/Google, EX1027, p. 708

are the quadtree node containing the extent's lower-left corner and the three adjacent quadtree nodes in the N, E, and NE directions. In Fig. 15.51, for example, these are quadtree nodes 12, 13, 30, and 31. If the rectangular extent intersects a rectangular quadtree node, then whether the octree node's projection (a convex polygon) also intersects the quadtree node can be determined efficiently [MEAG82b]. In contrast to the list-priority back-to-front algorithms, this front-to-back algorithm operates at image precision because it relies on an image-precision, quadtree representation of the projections on the image plane.

15.9 ALGORITHMS FOR CURVED SURFACES

All the algorithms presented thus far, with the exception of the z-buffer, have been described only for objects defined by polygonal faces. Objects such as the curved surfaces of Chapter 11 must first be approximated by many small facets before polygonal versions of any of the algorithms can be used. Although this approximation can be done, it is often preferable to scan convert curved surfaces directly, eliminating polygonal artifacts and avoiding the extra storage required by polygonal approximation.

Quadric surfaces, discussed in Section 11.4, are a popular choice in computer graphics. Visible-surface algorithms for quadrics have been developed by Weiss [WEIS66], Woon [WOON71], Mahl [MAHL72], Levin [LEVI76], and Sarraga [SARR83]. They all find the intersections of two quadrics, yielding a fourth-order equation in x, y, and z whose roots must be found numerically. Levin reduces this to a second-order problem by parameterizing the intersection curves. Spheres, a special case of quadrics, are easier to work with, and are of particular interest because molecules are often displayed as collections of colored spheres (see Color Plate II.19). A number of molecular display algorithms have been developed [KNOW77; STAU78; MAX79; PORT79; FRAN81; MAX84]. Section 15.10 discusses how to render spheres using ray tracing.

Even more flexibility can be achieved with the parametric spline surfaces introduced in Chapter 11, because they are more general and allow tangent continuity at patch boundaries. Catmull [CATM74b; CATM75] developed the first display algorithm for bicubics. In the spirit of Warnock's algorithm, a patch is recursively subdivided in s and t into four patches until its projection covers no more than one pixel. A z-buffer algorithm determines whether the patch is visible at this pixel. If it is, a shade is calculated for it and is placed in the frame buffer. The pseudocode for this algorithm is shown in Fig. 15.52. Since checking the size of the curved patch itself is time consuming, a quadrilateral defined by the patch's corner vertices may be used instead. Extra efficiency may be gained by comparing each patch (or its extent) with the clip window. If it is wholly inside the window, then no patch generated from it needs to be checked for clipping. If it is wholly outside the window, then it may be discarded. Finally, if it may be partially visible, then each patch generated from it must be checked.

Since then, Blinn and Whitted [LANE80b] have each developed scan-line algorithms for bicubics that track the visible edges of the surface from one scan line to the next. Edges may be defined by actual patch boundaries or by *silhouette edges*, as shown in Fig. 15.53. At a silhouette edge, the z component of the surface normal in the 3D screen coordinate system is zero as it passes between positive and negative values.

Samsung/Google, EX1027, p. 709

```
for (each patch) {
    push patch onto stack;

    while (stack not empty) {
        pop patch from stack;

        if (patch covers ≤ 1 pixel) {
            if (patch's pixel closer in z)
                determine shade and draw
        } else {
            subdivide patch into 4 subpatches;
            push subpatches onto stack;
        }
    }
}
```

Fig. 15.52 Pseudocode for the Catmull recursive-subdivision algorithm.

Blinn deals directly with the parametric representation of the patch. For the scan line $y = \alpha$, he finds all s and t values that satisfy the equation

$$y(s, t) - \alpha = 0. \tag{15.8}$$

These values of s and t are then used to evaluate $x(s, t)$ and $z(s, t)$. Unfortunately, Eq. (15.8) does not have a closed-form solution and its roots are therefore found numerically using Newton–Raphson iteration (see Appendix). Since the root-finding algorithm requires an initial value, coherence can be exploited by beginning with the previous scan line's solution for the current scan line. There are also special cases in which the roots cannot be found, causing the algorithm to fail. Similarly, Whitted uses numerical methods plus some approximations to the curve in the (x, z) plane defined by the intersection of the $y = \alpha$ plane

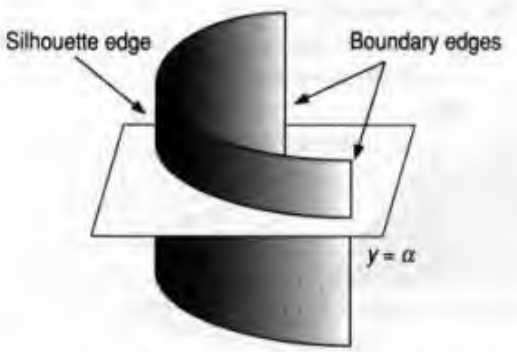


Fig. 15.53 The visible edges of a patch are defined by its boundary edges and silhouette edges.

Samsung/Google, EX1027, p. 710

with the bicubic surface patch. Whitted's algorithm fails to handle certain silhouette edges properly, however; an algorithm that does a more robust job of silhouette-edge detection is described in [SCHW82].

One highly successful approach is based on the adaptive subdivision of each bicubic patch until each subdivided patch is within some given tolerance of being flat. This tolerance depends on the resolution of the display device and on the orientation of the area being subdivided with respect to the projection plane, so unnecessary subdivisions are eliminated. The patch needs to be subdivided in only one direction if it is already flat enough in the other. Once subdivided sufficiently, a patch can be treated like a quadrilateral. The small polygonal areas defined by the four corners of each patch are processed by a scan-line algorithm, allowing polygonal and bicubic surfaces to be readily intermixed.

Algorithms that use this basic idea have been developed by Lane and Carpenter [LANE80b], and by Clark [CLAR79]. They differ in the choice of basis functions used to derive the subdivision difference equations for the surface patches and in the test for flatness. The Lane–Carpenter algorithm does the subdivision only as required when the scan line being processed begins to intersect a patch, rather than in a preprocessing step as does Clark's algorithm. The Lane–Carpenter patch subdivision algorithm is described in Section 11.3.5. Pseudocode for the Lane–Carpenter algorithm is shown in Fig. 15.54.

```
add patches to patch table;
initialize active-patch table;

for (each scan line) {

    update active-patch table;

    for (each patch in active-patch table) {
        if (patch can be approximately by planar quadrilateral)
            add patch to polygon table;
        else {
            split patch into subpatches;
            for (each new subpatch) {
                if (subpatch intersects scan line)
                    add to active-patch table;
                else
                    add to patch table;
            }
        }
    }

    process polygon table for current scan line;
}
```

Fig. 15.54 Pseudocode for the Lane–Carpenter algorithm.

Samsung/Google, EX1027, p. 711

Since a patch's control points define its convex hull, the patch is added to the active-patch table for processing at the scan line whose y value is that of the minimum y value of its control points. This saves large amounts of memory. The test for flatness must determine whether the patch is sufficiently planar and whether the boundary curves are sufficiently linear. Unfortunately, subdivision can introduce cracks in the patch if the same patch generates one patch that is determined to be flat and an adjacent patch that must be subdivided further. What should be a common shared edge between the patches may, instead, be a single line for the first patch and a piecewise linear approximation to a curve for the subpatches derived from the second patch. This can be avoided by changing the tolerance in the flatness test such that patches are subdivided more finely than necessary. An alternative solution uses Clark's method of subdividing an edge as though it were a straight line, once it has been determined to be sufficiently flat.

15.10 VISIBLE-SURFACE RAY TRACING

Ray tracing, also known as *ray casting*, determines the visibility of surfaces by tracing imaginary rays of light from the viewer's eye to the objects in the scene.[2] This is exactly the prototypical image-precision algorithm discussed at the beginning of this chapter. A center of projection (the viewer's eye) and a window on an arbitrary view plane are selected. The window may be thought of as being divided into a regular grid, whose elements correspond to pixels at the desired resolution. Then, for each pixel in the window, an *eye ray* is fired from the center of projection through the pixel's center into the scene, as shown in Fig. 15.55. The pixel's color is set to that of the object at the closest point of intersection. The pseudocode for this simple ray tracer is shown in Fig. 15.56.

Ray tracing was first developed by Appel [APPE68] and by Goldstein and Nagel [MAGI68; GOLD71]. Appel used a sparse grid of rays used to determine shading, including whether a point was in shadow. Goldstein and Nagel originally used their algorithm to simulate the trajectories of ballistic projectiles and nuclear particles; only later

[2]Although *ray casting* and *ray tracing* are often used synonymously, sometimes *ray casting* is used to refer to only this section's visible-surface algorithm, and *ray tracing* is reserved for the recursive algorithm of Section 16.12.

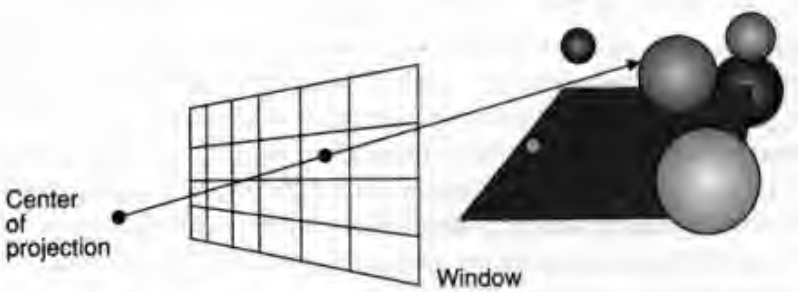


Fig. 15.55 A ray is fired from the center of projection through each pixel to which the window maps, to determine the closest object intersected.

Samsung/Google, EX1027, p. 712

```
select center of projection and window on viewplane;
for (each scan line in image) {
    for (each pixel in scan line) {
        determine ray from center of projection through pixel;
        for (each object in scene) {
            if (object is intersected and is closest considered thus far)
                record intersection and object name;
        }
        set pixel's color to that at closest object intersection;
    }
}
```

Fig. 15.56 Pseudocode for a simple ray tracer.

did they apply it to graphics. Appel was the first to ray trace shadows, whereas Goldstein and Nagel pioneered the use of ray tracing to evaluate Boolean set operations. Whitted [WHIT80] and Kay [KAY79a] extended ray tracing to handle specular reflection and refraction. We discuss shadows, reflection, and refraction—the effects for which ray tracing is best known—in Section 16.12, where we describe a full recursive ray-tracing algorithm that integrates both visible-surface determination and shading. Here, we treat ray tracing only as a visible-surface algorithm.

15.10.1 Computing Intersections

At the heart of any ray tracer is the task of determining the intersection of a ray with an object. To do this task, we use the same parametric representation of a vector introduced in Chapter 3. Each point (x, y, z) along the ray from (x_0, y_0, z_0) to (x_1, y_1, z_1) is defined by some value t such that

$$x = x_0 + t\,(x_1 - x_0), \qquad y = y_0 + t\,(y_1 - y_0), \qquad z = z_0 + t\,(z_1 - z_0). \quad (15.9)$$

For convenience, we define Δx, Δy, and Δz such that

$$\Delta x = x_1 - x_0, \qquad \Delta y = y_1 - y_0, \qquad \Delta z = z_1 - z_0. \quad (15.10)$$

Thus,

$$x = x_0 + t\,\Delta x, \qquad y = y_0 + t\,\Delta y, \qquad z = z_0 + t\,\Delta z. \quad (15.11)$$

If (x_0, y_0, z_0) is the center of projection and (x_1, y_1, z_1) is the center of a pixel on the window, then t ranges from 0 to 1 between these points. Negative values of t represent points behind the center of projection, whereas values of t greater than 1 correspond to points on the side of the window farther from the center of projection. We need to find a representation for each kind of object that enables us to determine t at the object's intersection with the ray. One of the easiest objects for which to do this is the sphere, which accounts for the plethora of spheres observed in typical ray-traced images! The sphere with center (a, b, c) and radius r may be represented by the equation

$$(x - a)^2 + (y - b)^2 + (z - c)^2 = r^2. \quad (15.12)$$

Samsung/Google, EX1027, p. 713

The intersection is found by expanding Eq. (15.12), and substituting the values of x, y, and z from Eq. (15.11) to yield

$$x^2 - 2ax + a^2 + y^2 - 2by + b^2 + z^2 - 2cz + c^2 = r^2, \tag{15.13}$$

$$(x_0 + t\Delta x)^2 - 2a(x_0 + t\Delta x) + a^2 + (y_0 + t\Delta y)^2 - 2b(y_0 + t\Delta y) + b^2 \tag{15.14}$$
$$+ (z_0 + t\Delta z)^2 - 2c(z_0 + t\Delta z) + c^2 = r^2,$$

$$x_0^2 + 2x_0\Delta xt + \Delta x^2t^2 - 2ax_0 - 2a\Delta xt + a^2 \tag{15.15}$$
$$+ y_0^2 + 2y_0\,\Delta yt + \Delta y^2t^2 - 2by_0 - 2b\Delta yt + b^2$$
$$+ z_0^2 + 2z_0\,\Delta zt + \Delta z^2t^2 - 2cz_0 - 2c\Delta zt + c^2 = r^2.$$

Collecting terms gives

$$(\Delta x^2 + \Delta y^2 + \Delta z^2)t^2 + 2t[\Delta x(x_0 - a) + \Delta y(y_0 - b) + \Delta z(z_0 - c)] \tag{15.16}$$
$$+ (x_0^2 - 2ax_0 + a^2 + y_0^2 - 2by_0 + b^2 + z_0^2 - 2cz_0 + c^2) - r^2 = 0,$$

$$(\Delta x^2 + \Delta y^2 + \Delta z^2)t^2 + 2t[\Delta x\,(x_0 - a) + \Delta y(y_0 - b) + \Delta z(z_0 - c)] \tag{15.17}$$
$$+ (x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - r^2 = 0.$$

Equation (15.17) is a quadratic in t, with coefficients expressed entirely in constants derived from the sphere and ray equations, so it can be solved using the quadratic formula. If there are no real roots, then the ray and sphere do not intersect; if there is one real root, then the ray grazes the sphere. Otherwise, the two roots are the points of intersection with the sphere; the one that yields the smallest positive t is the closest. It is also useful to normalize the ray so that the distance from (x_0, y_0, z_0) to (x_1, y_1, z_1) is 1. This gives a value of t that measures distance in WC units, and simplifies the intersection calculation, since the coefficient of t^2 in Eq. (15.17) becomes 1. We can obtain the intersection of a ray with the general quadric surfaces introduced in Chapter 11 in a similar fashion.

As we shall see in Chapter 16, we must determine the surface normal at the point of intersection in order to shade the surface. This is particularly easy in the case of the sphere, since the (unnormalized) normal is the vector from the center to the point of intersection: The sphere with center (a, b, c) has a surface normal $((x - a)/r, (y - b)/r, (z - c)/r)$ at the point of intersection (x, y, z).

Finding the intersection of a ray with a polygon is somewhat more difficult. We can determine where a ray intersects a polygon by first determining whether the ray intersects the polygon's plane and then whether the point of intersection lies within the polygon. Since the equation of a plane is

$$Ax + By + Cz + D = 0, \tag{15.18}$$

substitution from Eq. (15.11) yields

$$A(x_0 + t\Delta x) + B(y_0 + t\Delta y) + C(z_0 + t\Delta z) + D = 0, \tag{15.19}$$

$$t(A\Delta x + B\Delta y + C\Delta z) + (Ax_0 + By_0 + Cz_0 + D) = 0, \tag{15.20}$$

$$t = -\frac{(Ax_0 + By_0 + Cz_0 + D)}{(A\Delta x + B\Delta y + C\Delta z)}. \tag{15.21}$$

Samsung/Google, EX1027, p. 714

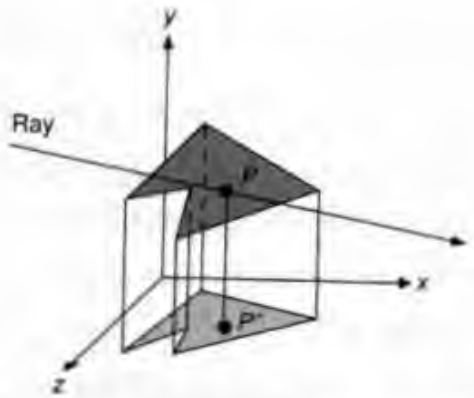


Fig. 15.57 Determining whether a ray intersects a polygon. The polygon and the ray's point of intersection p with the polygon's plane are projected onto one of the three planes defining the coordinate system. Projected point p' is tested for containment within the projected polygon.

If the denominator of Eq. (15.21) is 0, then the ray and plane are parallel and do not intersect. An easy way to determine whether the point of intersection lies within the polygon is to project the polygon and point orthographically onto one of the three planes defining the coordinate system, as shown in Fig. 15.57. To obtain the most accurate results, we should select the axis along which to project that yields the largest projection. This corresponds to the coordinate whose coefficient in the polygon's plane equation has the largest absolute value. The orthographic projection is accomplished by dropping this coordinate from the polygon's vertices and from the point. The polygon-containment test for the point can then be performed entirely in 2D, using the point-in-polygon algorithm of Section 7.12.2.

Like the z-buffer algorithm, ray tracing has the attraction that the only intersection operation performed is that of a projector with an object. There is no need to determine the intersection of two objects in the scene directly. The z-buffer algorithm approximates an object as a set of z values along the projectors that intersect the object. Ray tracing approximates objects as the set of intersections along each projector that intersects the scene. We can extend a z-buffer algorithm to handle a new kind of object by writing a scan-conversion and z-calculation routine for it. Similarly, we can extend a visible-surface ray tracer to handle a new kind of object by writing a ray-intersection routine for it. In both cases, we must also write a routine to calculate surface normals for shading. Intersection and surface-normal algorithms have been developed for algebraic surfaces [HANR83], for parametric surfaces [KAJI82; SEDE84; TOTH85; JOY86], and for many of the objects discussed in Chapter 20. Surveys of these algorithms are provided in [HAIN89; HANR89].

15.10.2 Efficiency Considerations for Visible-Surface Ray Tracing

At each pixel, the z-buffer algorithm computes information only for those objects that project to that pixel, taking advantage of coherence. In contrast, the simple but expensive version of the visible-surface ray tracing algorithm that we have discussed, intersects each of the rays from the eye with each of the objects in the scene. A 1024 by 1024 image of 100

Samsung/Google, EX1027, p. 715

objects would therefore require 100M intersection calculations. It is not surprising that Whitted found that 75 to over 95 percent of his system's time was spent in the intersection routine for typical scenes [WHIT80]. Consequently, the approaches to improving the efficiency of visible-surface ray tracing we discuss here attempt to speed up individual intersection calculations, or to avoid them entirely. As we shall see in Section 16.12, recursive ray tracers trace additional rays from the points of intersection to determine a pixel's shade. Therefore, several of the techniques developed in Section 15.2, such as the perspective transformation and back-face culling, are not in general useful, since all rays do not emanate from the same center of projection. In Section 16.12, we shall augment the techniques mentioned here with ones designed specifically to handle these recursive rays.

Optimizing intersection calculations. Many of the terms in the equations for object–ray intersection contain expressions that are constant either throughout an image or for a particular ray. These can be computed in advance, as can, for example, the orthographic projection of a polygon onto a plane. With care and mathematical insight, fast intersection methods can be developed; even the simple intersection formula for a sphere given in Section 15.10.1 can be improved [HAIN89]. If rays are transformed to lie along the z axis, then the same transformation can be applied to each candidate object, so that any intersection occurs at $x = y = 0$. This simplifies the intersection calculation and allows the closest object to be determined by a z sort. The intersection point can then be transformed back for use in shading calculations via the inverse transformation.

Bounding volumes provide a particularly attractive way to decrease the amount of time spent on intersection calculations. An object that is relatively expensive to test for intersection may be enclosed in a bounding volume whose intersection test is less expensive, such as a sphere [WHIT80], ellipsoid [BOUV85], or rectangular solid [RUBI80; TOTH85]. The object does not need to be tested if the ray fails to intersect with its bounding volume.

Kay and Kajiya [KAY86] suggest the use of a bounding volume that is a convex polyhedron formed by the intersection of a set of infinite *slabs*, each of which is defined by a pair of parallel planes that bound the object. Figure 15.58(a) shows in 2D an object bounded by four slabs (defined by pairs of parallel lines), and by their intersection. Thus, each slab is represented by Eq. (15.18), where A, B, and C are constant, and D is either D_{min} or D_{max}. If the same set of parameterized slabs is used to bound all objects, each bound can be described compactly by the D_{min} and D_{max} of each of its slabs. A ray is intersected with a bound by considering one slab at a time. The intersection of a ray with a slab can be computed using Eq. (15.21) for each of the slab's planes, producing near and far values of t. Using the same set of parameterized slabs for all bounds, however, allows us to simplify Eq.(15.21), yielding

$$t = (S + D)T, \tag{15.22}$$

where $S = Ax_0 + By_0 + Cz_0$ and $T = -1/(A\Delta x + B\Delta y + C\Delta z)$. Both S and T can be calculated once for a given ray and parameterized slab.

Since each bound is an intersection of slabs, the intersection of the ray with an entire bound is just the intersection of the ray's intersections with each of the bound's slabs. This can be computed by taking the maximum of the near values of t and the minimum of the far values of t. In order to detect null intersections quickly, the maximum near and minimum

Samsung/Google, EX1027, p. 716

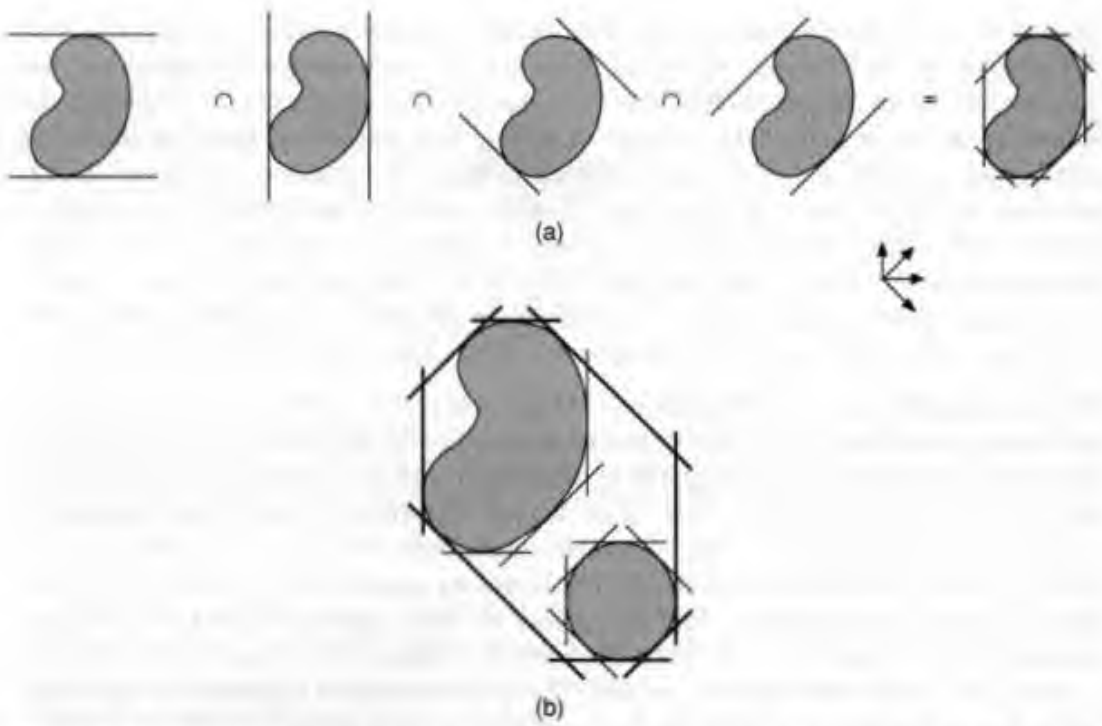


Fig. 15.58 Bounds formed intersection of slabs. (a) Object bounded by a fixed set of parameterized slabs. (b) The bounding volume of two bounding volumes.

far values of t for a bound can be updated as each of its slabs is processed, and the processing of the bound terminated if the former ever exceeds the latter.

Avoiding intersection calculations. Ideally, each ray should be tested for intersection only with objects that it actually intersects. Furthermore, in many cases we would like each ray to be tested against only that object whose intersection with the ray is closest to the ray's origin. There is a variety of techniques that attempt to approximate this goal by preprocessing the environment to partition rays and objects into equivalence classes to help limit the number of intersections that need to be performed. These techniques include two complementary approaches introduced in Section 15.2: hierarchies and spatial partitioning.

Hierarchies. Although bounding volumes do not by themselves determine the order or frequency of intersection tests, bounding volumes may be organized in nested hierarchies with objects at the leaves and internal nodes that bound their children [RUBI80; WEGH84; KAY86]. For example, a bounding volume for a set of Kay–Kajiya bounding volumes can be computed by taking for each pair of planes the minimum D_{min} and the maximum D_{max} of the values for each child volume, as shown in Fig. 15.58(b).

A child volume is guaranteed not to intersect with a ray if its parent does not. Thus, if intersection tests begin with the root, many branches of the hierarchy (and hence many

Samsung/Google, EX1027, p. 717

objects) may be trivially rejected. A simple method to traverse the hierarchy is

```
void HIER_traverse (ray r, node n)
{
    if (r intersects n's bounding volume)
        if (n is a leaf)
            intersect r with n's object;
        else
            for (each child c of n)
                HIER_traverse (r, c);
} /* HIER_traverse */
```

Efficient hierarchy traversal. HIER_traverse explores a hierarchy depth first. In contrast, Kay and Kajiya [KAY86] have developed an efficient method for traversing hierarchies of bounding volumes that takes into account the goal of finding the closest intersection. Note that the intersection of a ray with a Kay–Kajiya bound yields two values of t, the lower of which is a good estimate of the distance to the object. Therefore, the best order in which to select objects for intersection tests is that of increasing estimated distance from the ray's origin. To find the closest object intersected by a ray, we maintain the list of nodes to be tested in a priority queue, implemented as a heap. Initially, the heap is empty. If the root's bound is intersected by the ray, then the root is inserted in the heap. As long as the heap is not empty and its top node's estimated distance is closer than the closest object tested so far, nodes are extracted from the heap. If the node is a leaf, then its object's ray intersection is calculated. Otherwise, it is a bound, in which case each of its children's bounds is tested and is inserted in the heap if it is intersected, keyed by the estimated distance computed in the bound-intersection calculation. The selection process terminates when the heap is empty or when an object has been intersected that is closer than the estimated distance of any node remaining in the heap. Pseudocode for the algorithm is shown in Fig. 15.59.

Automated hierarchy generation. One problem with hierarchies of bounding volumes, such as those used by the Kay–Kajiya algorithm, is that generating good hierarchies is difficult. Hierarchies created during the modeling process tend to be fairly shallow, with structure designed for controlling objects rather than for minimizing the intersections of objects with rays. In addition, modeler hierarchies are typically insensitive to the actual position of objects. For example, the fingers on two robot hands remain in widely separated parts of the hierarchy, even when the hands are touching. Goldsmith and Salmon [GOLD87] have developed a method for generating good hierarchies for ray tracing automatically. Their method relies on a way of determining the quality of a hierarchy by estimating the cost of intersecting a ray with it.

Consider how we might estimate the cost of an individual bounding volume. Assume that each bounding volume has the same cost for computing whether a ray intersects it. Therefore, the cost is directly proportional to the number of times a bounding volume will be hit. The probability that a bounding volume is hit by an eye ray is the percentage of rays from the eye that will hit it. This is proportional to the bounding volume's area projected on the view plane. On the average, for convex bounding volumes, this value is roughly

Samsung/Google, EX1027, p. 718

```
void KayKajiya (void)
{
    object *p = NULL;          /* Pointer to nearest object hit */
    double t = ∞;              /* Distance to nearest object hit */

    precompute ray intersection;
    if (ray hits root's bound) {
        insert root into heap;
    }
    while (heap is not empty and distance to top node < t) {
        node *c = top node removed from heap;

        if (c is a leaf) {
            intersect ray with c's object;
            if (ray hits it and distance < t) {
                t = distance;
                p = object;
            }
        } else {        /* c is a bound */
            for (each child of c) {
                intersect ray with child's bound;
                if (ray hits child's bound)
                    insert child into heap;
            }
        }
    }
}  /* KayKajiya */
```

Fig. 15.59 Pseudocode for using Kay–Kajiya bounds to find the closest object intersected by a ray.

proportional to the bounding volume's surface area. Since each bounding volume is contained within the root's bounding volume, the conditional probability that a ray will intersect the ith bounding volume if it intersects the root can be approximated by A_i / A_r, where A_i is the surface area of the ith bounding volume and A_r is the surface area of the root.

If a ray intersects a bounding volume, we assume that we must perform an intersection calculation for each of the bounding volume's k children. Thus, the bounding volume's total estimated cost in number of intersections is kA_i / A_r. The root's estimated cost is just its number of children (since $A_i / A_r = 1$), and the cost of a leaf node is zero (since $k = 0$). To compute the estimated cost of a hierarchy, we sum the estimated costs of each of its bounding volumes. Consider, for example, the hierarchy shown in Fig. 15.60 in which each node is marked with its surface area. Assuming the root A is hit at the cost of one intersection, the root's estimated cost is 4 (its number of children). Two of its children (C

Samsung/Google, EX1027, p. 719

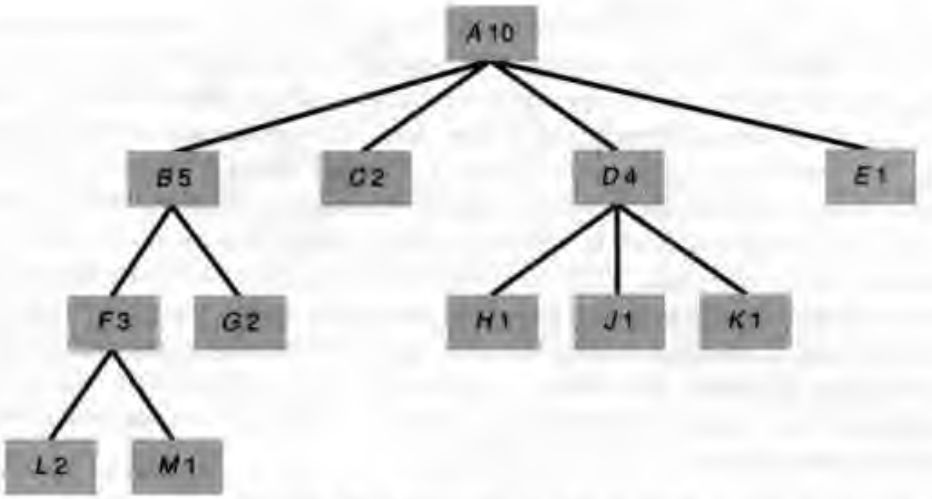


Fig. 15.60 Estimating the cost of a hierarchy. Letter is node name; number is node surface area.

and E) are leaves, and therefore have zero cost. B has two children and a surface area of 5. Thus, its estimated cost is $2(5/10) = 1.0$. D has three children and a surface area of 4, so its estimated cost is $3(4/10) = 1.2$. The only other nonleaf node is F, which has two children and a surface area of 3, giving an estimated cost of $2(3/10) = .6$. The total estimated cost is 1 (to hit the root) + 4 + 1.0 + 1.2 + .6 = 7.8 expected intersections.

Since we are interested in only relative values, there is no need to divide by the root's surface area. Furthermore, we do not need the actual surface area of the bounding volume—we need only a value proportional to it. For example, rather than use $2lw + 2lh + 2wh$ for a rectangular prism, we can factor out the 2 and rearrange terms to yield $(w + h)l + wh$.

Goldsmith and Salmon create the hierarchy incrementally, adding one new node at a time. The order in which nodes are added affects the algorithm. The modeler's order can be used, but for many scenes better results can be obtained by randomizing the order by shuffling nodes. Each node may be added by making it a child of an existing node or by replacing an existing node with a new bounding volume node that contains both the original node and the new node. In each case, instead of evaluating the cost of the new tree from scratch, the incremental cost of adding the node can be determined. If the node is being added as a child of an existing node, it may increase the parent's surface area, and it also increases the parent's number of children by 1. Thus, the difference in estimated cost of the parent is $k\,(A_{new} - A_{old}) + A_{new}$, where A_{new} and A_{old} are the parent's new and old surface areas, and k is the original number of children. If the node is added by creating a new parent with both the original and new nodes as children, the incremental cost of the newly created parent is $2A_{new}$. In both cases, the incremental cost to the new child's grandparent and older ancestors must also be computed as $k(A_{new} - A_{old})$, where k, A_{new}, and A_{old} are the values for the ancestor node. This approach assumes that the position at which the node is placed has no effect on the size of the root bounding volume.

A brute-force approach would be to evaluate the increased cost of adding the new node at every possible position in the tree and to then pick the position that incurred the least

Samsung/Google, EX1027, p. 720

increase in cost. Instead, Goldsmith and Salmon use a heuristic search that begins at the root by evaluating the cost of adding the node to it as a child. They then prune the tree by selecting the subtree that would experience the smallest increase in its bounding volume's surface area if the new node were added as a child. The search then continues with this subtree, the cost of adding the node to it as a child is evaluated, and a subtree of it is selected to follow based on the minimum surface area increase criterion. When a leaf node is reached, the cost is evaluated of creating a new bounding volume node containing the original leaf and the new node. When the search terminates, the node is inserted at the point with the smallest evaluated increase. Since determining the insertion point for a single node requires an $O(\log n)$ search, the entire hierarchy can be built in $O(n \log n)$ time. The search and evaluation processes are based on heuristics, and consequently the generated hierarchies are not optimal. Nevertheless, these techniques can create hierarchies that provide significant savings in intersection costs.

Spatial partitioning. Bounding-volume hierarchies organize objects bottom-up; in contrast, spatial partitioning subdivides space top-down. The bounding box of the scene is calculated first. In one approach, the bounding box is then divided into a regular grid of equal-sized extents, as shown in Fig. 15.61. Each partition is associated with a list of objects it contains either wholly or in part. The lists are filled by assigning each object to the one or more partitions that contain it. Now, as shown in 2D in Fig. 15.62, a ray needs to be intersected with only those objects that are contained within the partitions through which it passes. In addition, the partitions can be examined in the order in which the ray passes through them; thus, as soon as a partition is found in which there is an intersection, no more partitions need to be inspected. Note that we must consider all the remaining objects in the partition, to determine the one whose intersection is closest. Since the partitions follow a regular grid, each successive partition lying along a ray may be calculated using a 3D version of the line-drawing algorithm discussed in Section 3.2.2, modified to list every partition through which the ray passes [FUJI85; AMAN87].

If a ray intersects an object in a partition, it is also necessary to check whether the intersection itself lies in the partition; it is possible that the intersection that was found may be further along the ray in another partition and that another object may have a closer intersection. For example, in Fig. 15.63, object B is intersected in partition 3 although it is encountered in partition 2. We must continue traversing the partitions until an intersection is found in the partition currently being traversed, in this case with A in partition 3. To avoid

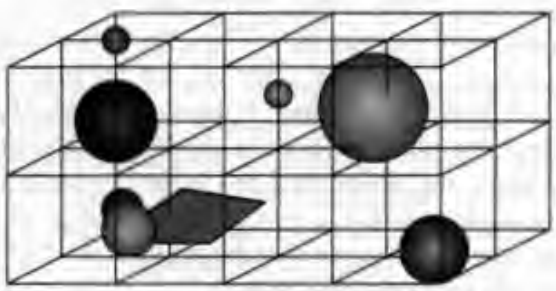


Fig. 15.61 The scene is partitioned into a regular grid of equal-sized volumes.

Samsung/Google, EX1027, p. 721

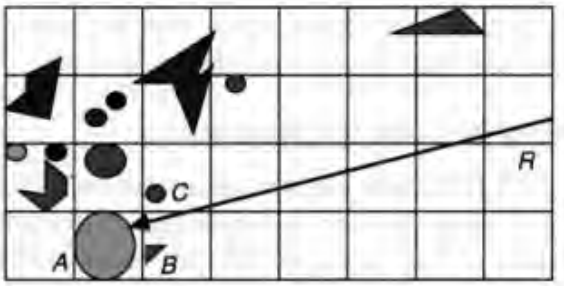


Fig. 15.62 Spatial partitioning. Ray *R* needs to be intersected with only objects *A*, *B*, and *C*, since the other partitions through which it passes are empty.

recalculating the intersection of a ray with an object that is found in multiple partitions, the point of intersection and the ray's ID can be cached with the object when the object is first encountered.

Dippé and Swensen [DIPP84] discuss an adaptive subdivision algorithm that produces unequal-sized partitions. An alternative adaptive spatial-subdivision method divides the scene using an octree [GLAS84]. In this case, the octree neighbor-finding algorithm sketched in Section 12.6.3 may be used to determine the successive partitions lying along a ray [SAME89b]. Octrees, and other hierarchical spatial partitionings, can be thought of as a special case of hierarchy in which a node's children are guaranteed not to intersect each other. Because these approaches allow adaptive subdivision, the decision to subdivide a partition further can be sensitive to the number of objects in the subdivision or the cost of intersecting the objects. This is advantageous in heterogeneous, unevenly distributed environments.

Spatial partitioning and hierarchy can be used together to combine their advantages. Snyder and Barr [SNYD87] describe an approach that uses hand-assembled hierarchies whose internal nodes are either lists or regular 3D grids. This allows the person designing an environment to choose lists for small numbers of sparsely arranged objects and grids for

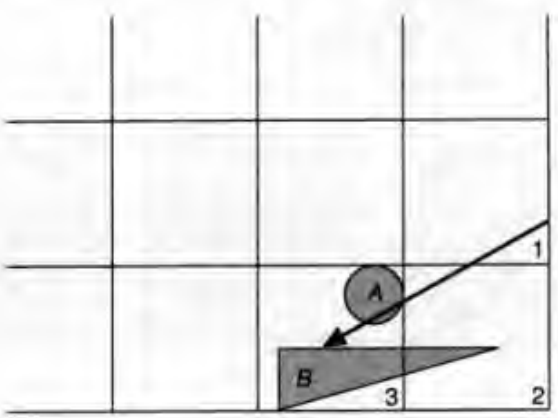


Fig. 15.63 An object may be intersected in a different voxel than the current one.

Samsung/Google, EX1027, p. 722

large numbers of regularly distributed objects. Color Plate III.1 shows a scene with 2×10^9 primitives that was ray-traced with Snyder's and Barr's system.

15.10.3 Computing Boolean Set Operations

Goldstein and Nagel [GOLD71] were the first researchers to ray trace combinations of simple objects produced using Boolean set operations. Determining the 3D union, difference, or intersection of two solids is difficult when it must be done by direct comparison of one solid with another using the methods in Chapter 12. In contrast, ray tracing allows the 3D problem to be reduced to a set of simple 1D calculations. The intersections of each ray and primitive object yield a set of t values, each of which specifies a point at which the ray enters or exits the object. Each t value thus defines the beginning of a span in which the ray is either in or out of the object. (Of course, care must be taken if the ray grazes the object, intersecting it only once.) Boolean set operations are calculated one ray at a time by determining the 1D union, difference, or intersection of spans from the two objects along the same ray. Figure 15.64 shows the spans defined by a ray passing through two objects, and the combinations of the spans that result when the set operations are

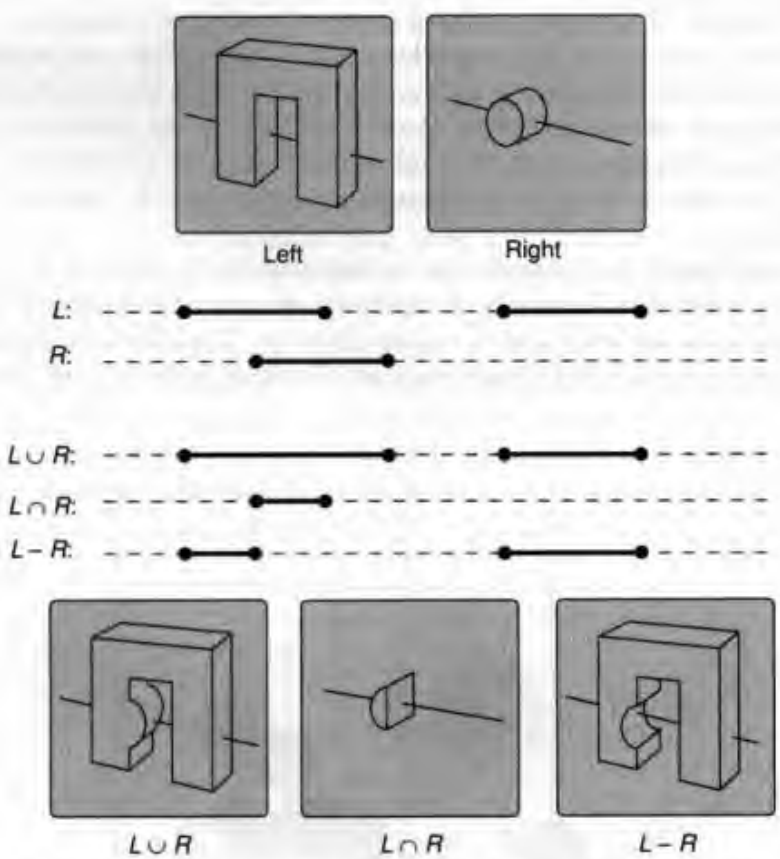


Fig. 15.64 Combining ray–object intersection span lists. (Adapted from [ROTH82] with permission.)

Samsung/Google, EX1027, p. 723

```
span *CSG_intersect (Ray *ray, CSG_node *node)
{
    span *leftIntersect, *rightIntersect;      /* lists of spans */

    if (node is composite) {
        leftIntersect = CSG_intersect (ray, node->leftChild);
        if (leftIntersect == NULL && node->op != UNION)
            return NULL;
        else {
            rightIntersect = CSG_intersect (ray, node->rightChild);
            return CSG_combine (node->op, leftIntersect, rightIntersect);
        }
    } else      /* node is primitive */
        return intersections of object with ray;
}  /* CSG_intersect */
```

Fig. 15.65 Pseudocode for evaluating the intersection of a ray with a CSG hierarchy.

performed. The CSG hierarchy is traversed for each ray by evaluating the left and right intersection lists at each node, as shown in the pseudocode of Fig. 15.65. Color Plate III.2 is a ray-traced bowl defined by a CSG hierarchy.

Roth points out that, if there is no intersection with the left side of the tree, then there is no reason to intersect with the right side of the tree if the operation is a difference or intersection [ROTH82]. Only if the operation is a union can the result be nonempty. In fact, if we need to determine only whether or not the compound object is intersected (rather than the actual set of intersections), then the right-hand side need not be evaluated if the left-hand side intersects and the operation is a union.

The CSG_combine function takes two lists of intersection records, each ordered by increasing t, and combines them according to the operation being performed. The lists are merged by removing the intersection record that has the next largest value of t. Whether the ray is "in" the left list or the right list is noted by setting a flag associated with the list from which the record is removed. Whether the span starting at that intersection point is in the combined object is determined by table lookup based on the operator and the two "in" flags, using Table 15.2. A record is then placed on the combined list only if it begins or

TABLE 15.2 POINT CLASSIFICATION FOR OBJECTS COMBINED BY BOOLEAN SET OPERATIONS

Left	Right	$\cup$	$\cap$	$-$
in	in	in	in	out
in	out	in	out	in
out	in	in	out	out
out	out	out	out	out

Samsung/Google, EX1027, p. 724

ends a span of the combined object, not if it is internal to one of the combined object's spans. If a ray can begin inside an object, the flags must be initialized correctly.

15.10.4 Antialiased Ray Tracing

The simple ray tracer described so far uses point sampling on a regular grid, and thus produces aliased images. Whitted [WHIT80] developed an adaptive method for firing more rays into those parts of the image that would otherwise produce the most severe aliasing. These additional samples are used to compute a better value for the pixel. His *adaptive supersampling* associates rays with the corners, rather than with the centers, of each pixel, as shown in Fig. 15.66(a) and (b). Thus, at first, only an extra row and an extra column of rays are needed for the image. After rays have been fired through all four corners of a pixel, the shades they determine are averaged; the average is then used for the pixel if the shades differ from it by only a small amount. If they differ by too much, then the pixel is subdivided further by firing rays through the midpoints of its sides and through its center, forming four subpixels (Fig. 15.66c). The rays at the four corners of each subpixel are then compared using the same criterion. Subdivision proceeds recursively until a predefined maximum subdivision depth is reached, as in the Warnock algorithm, or until the ray shades are determined to be sufficiently similar. The pixel's shade is the area-weighted average of its subpixels' shades. Adaptive supersampling thus provides an improved approximation to unweighted area sampling, without the overhead of a uniformly higher sampling rate.

Consider, for example, Fig. 15.66(a), which shows the rays fired through the corners of two adjacent pixels, with a maximum subdivision depth of two. If no further subdivision is needed for the pixel bordered by rays A, B, D, and E in part (b), then, representing a ray's shade by its name, the pixel's shade is $(A + B + D + E)/4$. The adjacent pixel requires further subdivision, so rays G, H, I, J, and K are traced, defining the vertices of four subpixels in part (c). Each subpixel is recursively inspected. In this case, only the lower-right subpixel is subdivided again by tracing rays L, M, N, O, and P, as shown in part (d). At this point, the maximum subdivision depth is reached. This pixel's shade is

$$\frac{1}{4}\left[\frac{B+G+H+I}{4}+\frac{1}{4}\left[\frac{G+L+M+N}{4}+\frac{L+C+N+O}{4}+\frac{M+N+I+P}{4}+\frac{N+O+P+J}{4}\right]+\frac{H+I+E+K}{4}+\frac{I+J+K+F}{4}\right].$$

Aliasing problems can also arise when the rays through a pixel miss a small object. This produces visible effects if the objects are arranged in a regular pattern and some are not visible, or if a series of pictures of a moving object show that object popping in and out of view as it is alternately hit and missed by the nearest ray. Whitted avoids these effects by surrounding each object with a spherical bounding volume that is sufficiently large always to be intersected by at least one ray from the eye. Since the rays converge at the eye, the size of the bounding volume is a function of the distance from the eye. If a ray intersects the

Samsung/Google, EX1027, p. 725

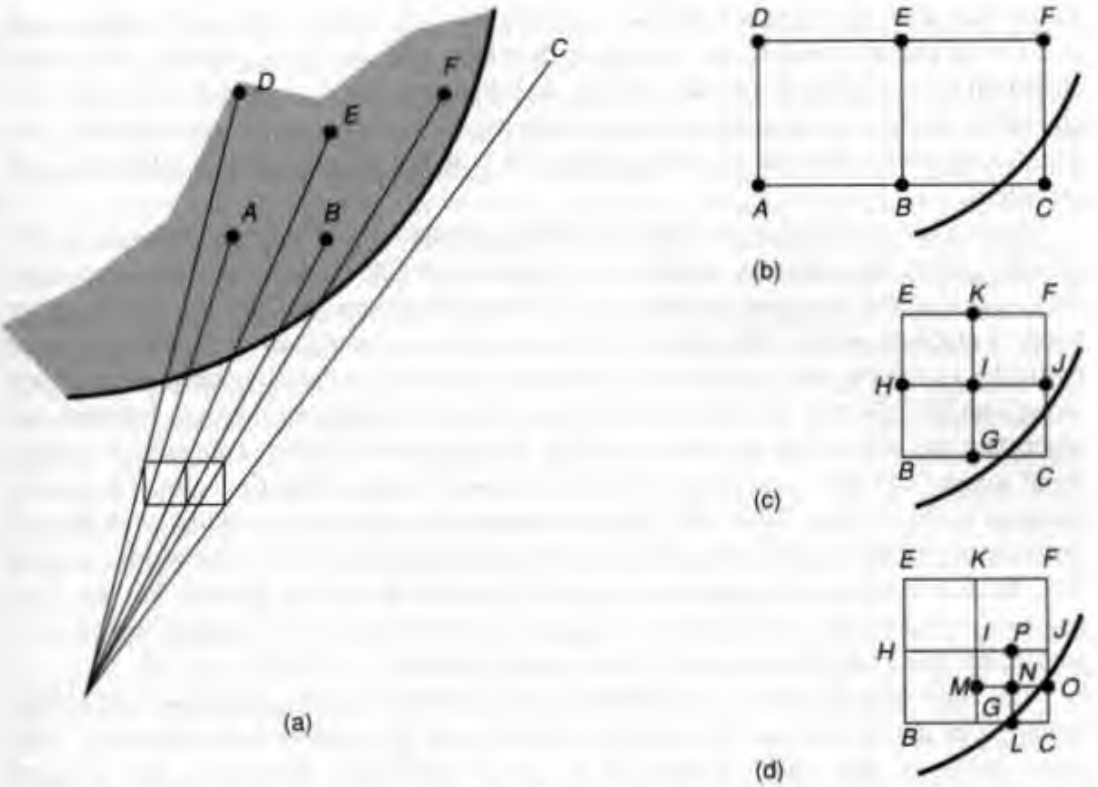


Fig. 15.66 Adaptive supersampling. (a) Two pixels and the rays fired through their corners. (b) The left pixel is not subdivided. (c) The right pixel is subdivided. (d) The lower-right subpixel is subdivided.

bounding volume but does not intersect the object, then all pixels sharing that ray are further subdivided until the object is intersected. Several more recent approaches to antialiased ray tracing are discussed in Section 16.12.

15.11 SUMMARY

Sutherland, Sproull, and Schumacker [SUTH74a] stress that the heart of visible-surface determination is sorting. Indeed, we have seen many instances of sorting and searching in the algorithms, and efficient sorting is vital to efficient visible-surface determination. Equally important is avoiding any more sorting than is absolutely necessary, a goal typically achieved by exploiting coherence. For example, the scan-line algorithms use scan-line coherence to eliminate the need for a complete sort on x for each scan line. Hubschman and Zucker use frame coherence to avoid unnecessary comparisons in animation sequences [HUBS82].

Algorithms can be classified by the order in which they sort. The depth-sort algorithm sorts on z and then on x and y (by use of extents in tests 1 and 2); it is thus called a zxy algorithm. Scan-line algorithms sort on y (with a bucket sort), then sort on x (initially with

Samsung/Google, EX1027, p. 726

an insertion sort, then with a bubble sort as each scan line is processed), and finally search in z for the polygon nearest the viewpoint; therefore, they are yxz algorithms. Warnock's algorithm does a parallel sort on x and y, and then searches in z, and hence is an $(xy)z$ algorithm (sorting on a combination of dimensions is indicated by parentheses). The z-buffer algorithm does no explicit sorting and searches only in z; it is called an (xyz) algorithm.

Sancha has argued that the order of sorting is unimportant: There is no intrinsic benefit in sorting along any particular axis first as opposed to another because, at least in principle, the *average* object is equally complex in all three dimensions [SUTH74a]. On the other hand, a graphics scene, like a Hollywood set, may be constructed to look best from a particular viewpoint, and this may entail building in greater complexity along one axis than along another. Even if we assume roughly symmetric object complexity, however, all algorithms are still not equally efficient: They differ in how effectively coherence is used to avoid sorting and other computation and in the use of space–time tradeoffs. The results reported in [SUTH74a, Table VII], which compare the estimated performance of four of the basic algorithms we have presented, are summarized in Table 15.3. The authors suggest that, because these are only estimates, small differences should be ignored, but that "we feel free to make order of magnitude comparisons between the various algorithms to learn something about the effectiveness of the various methods" [SUTH74a, p. 52].

The depth-sort algorithm is efficient for small numbers of polygons because the simple overlap tests almost always suffice to decide whether a polygon can be scan-converted. With more polygons, the more complex tests are needed more frequently and polygon subdivision is more likely to be required. The z-buffer algorithm has constant performance because, as the number of polygons in a scene increases, the number of pixels covered by a single polygon decreases. On the other hand, its memory needs are high. The individual tests and calculations involved in the Warnock area-subdivision algorithm are relatively complex, so it is generally slower than are the other methods.

In addition to these informal estimates, there has been some work on formalizing the visible-surface problem and analyzing its computational complexity [GILO78; FOUR88; FIUM89]. For example, Fiume [FIUM89] proves that object-precision visible-surface algorithms have a lower bound that is worse than that of sorting: Even the simple task of

TABLE 15.3 RELATIVE ESTIMATED PERFORMANCE OF FOUR ALGORITHMS FOR VISIBLE-SURFACE DETERMINATION

	Number of polygonal faces in scene		
Algorithm	100	2500	60,000
Depth sort	1*	10	507
z-buffer	54	54	54
Scan line	5	21	100
Warnock area subdivision	11	64	307

*Entries are normalized such that this case is unity.

Samsung/Google, EX1027, p. 727

computing the visible surfaces of a set of n convex polygons can result in the creation of $\Omega(n^2)$ polygons, requiring $\Omega(n^2)$ time to output, as shown in Fig. 15.67.

In general, comparing visible-surface algorithms is difficult because not all algorithms compute the same information with the same accuracy. For example, we have discussed algorithms that restrict the kinds of objects, relationships among objects, and even the kinds of projections that are allowed. As we shall see in the following chapter, the choice of a visible-surface algorithm is also influenced by the kind of shading desired. If an expensive shading procedure is being used, it is better to choose a visible-surface algorithm that shades only parts of objects that are visible, such as a scan-line algorithm. Depth sort would be a particularly bad choice in this case, since it draws all objects in their entirety. When interactive performance is important, hardware z-buffer approaches are popular. The BSP-tree algorithm, on the other hand, can generate new views of a static environment quickly, but requires additional processing whenever the environment changes. Scan-line algorithms allow extremely high resolution because data structures need to represent fully elaborated versions only of primitives that affect the line being processed. As with any algorithm, the time spent implementing the algorithm and the ease with which it can be modified (e.g., to accommodate new primitives) is also a major factor.

One important consideration in implementing a visible-surface algorithm is the kind of hardware support available. If a parallel machine is available, we must recognize that, at each place where an algorithm takes advantage of coherence, it depends on the results of previous calculations. Exploiting parallelism may entail ignoring some otherwise useful form of coherence. Ray tracing has been a particularly popular candidate for parallel implementation because, in its simplest form, each pixel is computed independently. As we shall see in Chapter 18, there are many architectures that have been designed to execute specific visible-surface algorithms. For example, plummeting memory costs have made hardware z-buffer systems ubiquitous.

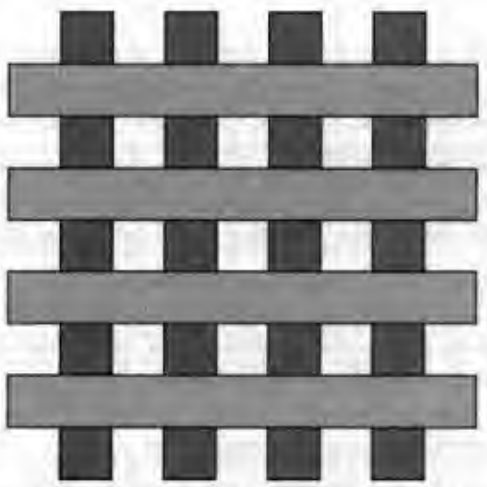


Fig. 15.67 $n/2$ rectangles laid across $n/2$ more distant rectangles can yield $n/2$ visible whole rectangles + $(n/2)$ $(n/2 + 1)$ visible fragments. This is $\Omega(n^2)$ visible polygons. (After [FIUM89].)

Samsung/Google, EX1027, p. 728

EXERCISES

15.1 Prove that the transformation M in Section 15.2.2 preserves (a) straight lines, (b) planes, and (c) depth relationships.

15.2 Given a plane $Ax + By + Cz + D = 0$, apply M from Section 15.2.2 and find the new coefficients of the plane equation.

15.3 How can a scan-line algorithm be extended to deal with polygons with shared edges? Should a shared edge be represented once, as a shared edge, or twice, once for each polygon it borders, with no record kept that it is a shared edge? When the depth of two polygons is evaluated at their common shared edge, the depths will, of course, be equal. Which polygon should be declared visible, given that the scan is entering both?

15.4 Warnock's algorithm generates a quadtree. Show the quadtree corresponding to Fig. 15.44. Label all nodes to indicate how the triangle (T) and the rectangle (R) relate to the node, as (a) surrounding, (b) intersecting, (c) contained, and (d) disjoint.

15.5 For each of the visible-surface algorithms discussed, explain how piercing polygons would be handled. Are they a special case that must be treated explicitly, or are they accommodated by the basic algorithm?

15.6 Consider tests 3 and 4 of the depth-sort algorithm. How might they be implemented efficiently? Consider examining the sign of the equation of the plane of polygon P for each vertex of polygon Q, and vice versa. How do you know to which side of the plane a positive value of the equation corresponds?

15.7 How can the algorithms discussed be adapted to work with polygons containing holes?

15.8 Describe how the visible-line algorithms for functions of two variables, described in Section 15.1, can be modified to work as visible-surface algorithms using the approach taken in the painter's algorithm.

15.9 Why does the Roberts visible-line algorithm not eliminate *all* lines that are edges of a back-facing polygon?

15.10 One of the advantages of the z-buffer algorithm is that primitives may be presented to it in any order. Does this mean that two images created by sending primitives in different orders will have identical values in their z-buffers and in their frame buffers? Explain your answer.

15.11 Consider merging two images of identical size, represented by their frame-buffer and z-buffer contents. If you know the z_{min} and z_{max} of each image and the values of z to which they originally corresponded, can you merge the images properly? Is any additional information needed?

15.12 Section 15.4 mentions the z-compression problems caused by rendering a perspective projection using an integer z-buffer. Choose a perspective viewing specification and a small number of object points. Show how, in the perspective transformation, two points near the center of projection are mapped to different z values, whereas two points separated from each other by the same distance, but farther from the center of projection, are mapped to a single z value.

15.13 a. Suppose view volume V has a front clipping plane at distance F and a back clipping plane at distance B, and that view volume V' has clipping planes at F' and B'. After transformation of each view volume to the canonical-perspective view volume, the back clipping plane of V will be at $z = -1$, and the front clipping plane at $z = A$. For V', the front clipping plane will be at $z = A'$. Show that, if $B / F = B' / F'$, then $A = A'$.

Samsung/Google, EX1027, p. 729

b. Part (a) shows that, in considering the effect of perspective, we need to consider only the ratio of back-plane to front-plane distance. We can therefore simply study the canonical view volume with various values of the front-plane distance. Suppose, then, that we have a canonical-perspective view volume, with front clipping plane $z = A$ and back clipping plane $z = -1$, and we transform it, through the perspective transformation, to the parallel view volume between $z = 0$ and $z = -1$. Write down the formula for the transformed z coordinate in terms of the original z coordinate. (Your answer will depend on A, of course.) Suppose that the transformed z values in the parallel view volume are multiplied by 2^n, and then are rounded to integers (i.e., they are mapped to an integer z-buffer). Find two values of z that are as far apart as possible, but that map, under this transformation, to the same integer. (Your answer will depend on n and A.)

c. Suppose you want to make an image in which the backplane-to-frontplane ratio is R, and objects that are more than distance Q apart (in z) must map to different values in the z-buffer. Using your work in part (b), write a formula for the number of bits of z-buffer needed.

15.14 Show that the back-to-front display order determined by traversing a BSP tree is not necessarily the same as the back-to-front order determined by the depth-sort algorithm, even when no polygons are split. (Hint: Only two polygons are needed.)

15.15 How might you modify the BSP-tree algorithm to accept objects other than polygons?

15.16 How might you modify the BSP-tree algorithm to allow limited motion?

15.17 Suppose that you are designing a ray tracer that supports CSG. How would you handle a polygon that is not part of a polyhedron?

15.18 Some graphics systems implement hardware transformations and homogeneous-coordinate clipping in X, Y, and Z using the same mathematics, so that clipping limits are

$$-W \le X \le W, \qquad -W \le Y \le W, \qquad -W \le Z \le W,$$

instead of

$$-W \le X \le W, \qquad -W \le Y \le W, \qquad -W \le Z \le 0.$$

How would you change the viewing matrix calculation to take this into account?

15.19 When ray tracing is performed, it is typically necessary to compute only whether or not a ray intersects an extent, not what the actual points of intersection are. Complete the ray–sphere intersection equation (Eq. 15.17) using the quadratic formula, and show how it can be simplified to determine only whether or not the ray and sphere intersect.

15.20 Ray tracing can also be used to determine the mass properties of objects through numerical integration. The full set of intersections of a ray with an object gives the total portion of the ray that is inside the object. Show how you can estimate an object's volume by firing a regular array of parallel rays through that object.

15.21 Derive the intersection of a ray with a quadric surface. Modify the method used to derive the intersection of a ray with a sphere in Eqs. (15.13) through (15.16) to handle the definition of a quadric given in Section 11.4.

15.22 In Eq. (15.5), O, the cost of performing an object intersection test, may be partially underwritten by B, the cost of performing a bounding-volume intersection test, if the results of the

Samsung/Google, EX1027, p. 730

bounding-volume intersection test can be reused to simplify the object intersection test. Describe an object and bounding volume for which this is possible.

15.23 Implement one of the polygon visible surface algorithms in this chapter, such as a z-buffer algorithm, scan-line algorithm, or BSP tree algorithm.

15.24 Implement a simple ray tracer for spheres and polygons, including adaptive supersampling. (Choose one of the illumination models from Section 16.1.) Improve your program's performance through the use of spatial partitioning or hierarchies of bounding volumes.

15.25 If you have implemented the z-buffer algorithm, then add hit detection to it by extending the pick-window approach described in Section 7.12.2 to take visible-surface determination into account. You will need a SetPickMode procedure that is passed a mode flag, indicating whether objects are to be drawn (drawing mode) or instead tested for hits (pick mode). A SetPick Window procedure will let the user set a rectangular pick window. The z-buffer must already have been filled (by drawing all objects) for pick mode to work. When in pick mode, neither the frame-buffer nor the z-buffer is updated, but the z-value of each of the primitive's pixels that falls inside the pick window is compared with the corresponding value in the z-buffer. If the new value would have caused the object to be drawn in drawing mode, then a flag is set. The flag can be inquired by calling InquirePick, which then resets the flag. If InquirePick is called after each primitive's routine is called in pick mode, picking can be done on a per-primitive basis. Show how you can use InquirePick to determine which object is actually visible at a pixel.

Samsung/Google, EX1027, p. 731

16 Illumination and Shading

In this chapter, we discuss how to shade surfaces based on the position, orientation, and characteristics of the surfaces and the light sources illuminating them. We develop a number of different *illumination models* that express the factors determining a surface's color at a given point. Illumination models are also frequently called *lighting models* or *shading models*. Here, however, we reserve the term *shading model* for the broader framework in which an illumination model fits. The shading model determines when the illumination model is applied and what arguments it will receive. For example, some shading models invoke an illumination model for every pixel in the image, whereas others invoke an illumination model for only some pixels, and shade the remaining pixels by interpolation.

When we compared the accuracy with which the visible-surface calculations of the previous chapter are performed, we distinguished between algorithms that use the actual object geometry and those that use polyhedral approximations, between object-precision and image-precision algorithms, and between image-precision algorithms that take one point sample per pixel and those that use better filters. In all cases, however, the single criterion for determining the direct visibility of an object at a pixel is whether something lies between the object and the observer along the projector through the pixel. In contrast, the interaction between lights and surfaces is a good deal more complex. Graphics researchers have often approximated the underlying rules of optics and thermal radiation, either to simplify computation or because more accurate models were not known in the graphics community. Consequently, many of the illumination and shading models traditionally used in computer graphics include a multitude of kludges, "hacks," and simplifications that have no firm grounding in theory, but that work well in practice. The

Samsung/Google, EX1027, p. 732

first part of this chapter covers these simple models, which are still in common use because they can produce attractive and useful results with minimal computation.

We begin, in Section 16.1, with a discussion of simple illumination models that take into account an individual point on a surface and the light sources directly illuminating it. We first develop illumination models for monochromatic surfaces and lights, and then show how the computations can be generalized to handle the color systems discussed in Chapter 13. Section 16.2 describes the most common shading models that are used with these illumination models. In Section 16.3, we expand these models to simulate textured surfaces.

Modeling refraction, reflection, and shadows requires additional computation that is very similar to, and often is integrated with, hidden-surface elimination. Indeed, these effects occur because some of the "hidden surfaces" are not really hidden at all—they are seen through, reflected from, or cast shadows on the surface being shaded! Sections 16.4 through 16.6 discuss how to model these effects. We next introduce, in Section 16.7, illumination models that more accurately characterize how an individual surface interacts with the light sources directly illuminating it. This is followed by coverage of additional ways to generate more realistic images, in Section 16.8 through 16.10.

Sections 16.11 through 16.13 describe *global illumination models* that attempt to take into account the interchange of light between all surfaces: recursive ray tracing and radiosity methods. Recursive ray tracing extends the visible-surface ray-tracing algorithm introduced in the previous chapter to interleave the determination of visibility, illumination, and shading at each pixel. Radiosity methods model the energy equilibrium in a system of surfaces; they determine the illumination of a set of sample points in the environment in a view-independent fashion before visible-surface determination is performed from the desired viewpoint. More detailed treatments of many of the illumination and shading models covered here may be found in [GLAS89; HALL89].

Finally, in Section 16.14, we look at several different graphics pipelines that integrate the rasterization techniques discussed in this and the previous chapters. We examine some ways to implement these capabilities to produce systems that are both efficient and extensible.

16.1 ILLUMINATION MODELS

16.1.1 Ambient Light

Perhaps the simplest illumination model possible is that used implicitly in this book's earliest chapters: Each object is displayed using an intensity intrinsic to it. We can think of this model, which has no external light source, as describing a rather unrealistic world of nonreflective, self-luminous objects. Each object appears as a monochromatic silhouette, unless its individual parts, such as the polygons of a polyhedron, are given different shades when the object is created. Color Plate II.28 demonstrates this effect.

An illumination model can be expressed by an *illumination equation* in variables associated with the point on the object being shaded. The illumination equation that expresses this simple model is

$$I = k_i, \tag{16.1}$$

Samsung/Google, EX1027, p. 733

where I is the resulting intensity and the coefficient k_i is the object's intrinsic intensity. Since this illumination equation contains no terms that depend on the position of the point being shaded, we can evaluate it once for each object. The process of evaluating the illumination equation at one or more points on an object is often referred to as *lighting* the object.

Now imagine, instead of self-luminosity, that there is a diffuse, nondirectional source of light, the product of multiple reflections of light from the many surfaces present in the environment. This is known as *ambient* light. If we assume that ambient light impinges equally on all surfaces from all directions, then our illumination equation becomes

$$I = I_a k_a. \tag{16.2}$$

I_a is the intensity of the ambient light, assumed to be constant for all objects. The amount of ambient light reflected from an object's surface is determined by k_a, the *ambient-reflection coefficient*, which ranges from 0 to 1. The ambient-reflection coefficient is a *material property*. Along with the other material properties that we will discuss, it may be thought of as characterizing the material from which the surface is made. Like some of the other properties, the ambient-reflection coefficient is an empirical convenience and does not correspond directly to any physical property of real materials. Furthermore, ambient light by itself is not of much interest. As we see later, it is used to account for all the complex ways in which light can reach an object that are not otherwise addressed by the illumination equation. Color Plate II.28 also demonstrates illumination by ambient light.

16.1.2 Diffuse Reflection

Although objects illuminated by ambient light are more or less brightly lit in direct proportion to the ambient intensity, they are still uniformly illuminated across their surfaces. Now consider illuminating an object by a *point light source*, whose rays emanate uniformly in all directions from a single point. The object's brightness varies from one part to another, depending on the direction of and distance to the light source.

Lambertian reflection. Dull, matte surfaces, such as chalk, exhibit *diffuse reflection*, also known as *Lambertian reflection*. These surfaces appear equally bright from all viewing angles because they reflect light with equal intensity in all directions. For a given surface, the brightness depends only on the angle θ between the direction $\overline{L}$ to the light source and the surface normal $\overline{N}$ of Fig. 16.1. Let us examine why this occurs. There are two factors at work here. First, Fig. 16.2 shows that a beam that intercepts a surface covers an area whose size is inversely proportional to the cosine of the angle θ that the beam makes with $\overline{N}$. If the

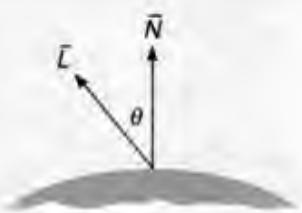


Fig. 16.1 Diffuse reflection.

Samsung/Google, EX1027, p. 734

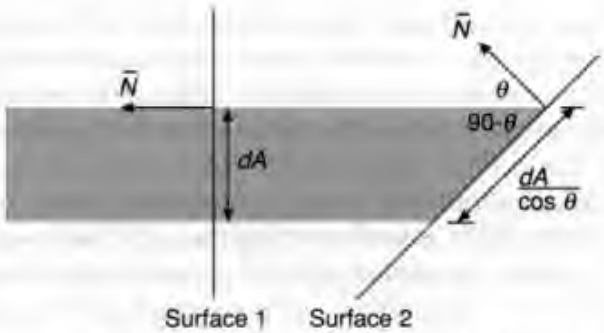


Fig. 16.2 Beam (shown in 2D cross-section) of infinitesimal cross-sectional area dA at angle of incidence θ intercepts area of dA / cos θ.

beam has an infinitesimally small cross-sectional differential area dA, then the beam intercepts an area dA / cos θ on the surface. Thus, for an incident light beam, the amount of light energy that falls on dA is proportional to cos θ. This is true for any surface, independent of its material.

Second, we must consider the amount of light seen by the viewer. Lambertian surfaces have the property, often known as Lambert's law, that the amount of light reflected from a unit differential area dA toward the viewer is directly proportional to the cosine of the angle between the direction to the viewer and $\bar{N}$. Since the amount of surface area seen is inversely proportional to the cosine of this angle, these two factors cancel out. For example, as the viewing angle increases, the viewer sees more surface area, but the amount of light reflected at that angle per unit area of surface is proportionally less. Thus, for Lambertian surfaces, the amount of light seen by the viewer is independent of the viewer's direction and is proportional only to cos θ, the angle of incidence of the light.

The diffuse illumination equation is

$$I = I_p k_d \cos \theta. \tag{16.3}$$

I_p is the point light source's intensity; the material's *diffuse-reflection coefficient* k_d is a constant between 0 and 1 and varies from one material to another. The angle θ must be between 0° and 90° if the light source is to have any direct effect on the point being shaded. This means that we are treating the surface as *self-occluding*, so that light cast from behind a point on the surface does not illuminate it. Rather than include a max(cos θ, 0) term explicitly here and in the following equations, we assume that θ lies within the legal range. When we want to light self-occluding surfaces, we can use abs(cos θ) to invert their surface normals. This causes both sides of the surface to be treated alike, as if the surface were lit by two opposing lights.

Assuming that the vectors $\bar{N}$ and $\bar{L}$ have been normalized (see Appendix), we can rewrite Eq. (16.3) by using the dot product:

$$I = I_p k_d (\bar{N} \cdot \bar{L}). \tag{16.4}$$

The surface normal $\bar{N}$ can be calculated using the methods discussed in Chapter 11. If polygon normals are precomputed and transformed with the same matrix used for the polygon vertices, it is important that nonrigid modeling transformations, such as shears or

Samsung/Google, EX1027, p. 735



Fig. 16.3 Spheres shaded using a diffuse-reflection model (Eq. 16.4). For all spheres, $I_p = 1.0$. From left to right, $k_d = 0.4, 0.55, 0.7, 0.85, 1.0$. (By David Kurlander, Columbia University.)

differential scaling, not be performed; these transformations do not preserve angles and may cause some normals to be no longer perpendicular to their polygons. The proper method to transform normals when objects undergo arbitrary transformations is described in Section 5.6. In any case, the illumination equation must be evaluated in the WC system (or in any coordinate system isometric to it), since both the normalizing and perspective transformations will modify θ.

If a point light source is sufficiently distant from the objects being shaded, it makes essentially the same angle with all surfaces sharing the same surface normal. In this case, the light is called a *directional light source*, and $\overline{L}$ is a constant for the light source.

Figure 16.3 shows a series of pictures of a sphere illuminated by a single point source. The shading model calculated the intensity at each pixel at which the sphere was visible using the illumination model of Eq. (16.4). Objects illuminated in this way look harsh, as when a flashlight illuminates an object in an otherwise dark room. Therefore, an ambient term is commonly added to yield a more realistic illumination equation:

$$I = I_a k_a + I_p k_d (\overline{N} \cdot \overline{L}). \tag{16.5}$$

Equation (16.5) was used to produce Fig. 16.4.

Light-source attenuation. If the projections of two parallel surfaces of identical material, lit from the eye, overlap in an image, Eq. (16.5) will not distinguish where one surface leaves off and the other begins, no matter how different are their distances from the light source. To do this, we introduce a light-source attenuation factor, f_{att}, yielding

$$I = I_a k_a + f_{att} I_p k_d (\overline{N} \cdot \overline{L}). \tag{16.6}$$



Fig. 16.4 Spheres shaded using ambient and diffuse reflection (Eq. 16.5). For all spheres, $I_a = I_p = 1.0$, $k_d = 0.4$. From left to right, $k_a = 0.0, 0.15, 0.30, 0.45, 0.60$. (By David Kurlander, Columbia University.)

Samsung/Google, EX1027, p. 736

An obvious choice for f_{att} takes into account the fact that the energy from a point light source that reaches a given part of a surface falls off as the inverse square of d_L, the distance the light travels from the point source to the surface. In this case,

$$f_{att} = \frac{1}{d_L^2}. \tag{16.7}$$

In practice, however, this often does not work well. If the light is far away, $1 / d_L^2$ does not vary much; if it is very close, it varies widely, giving considerably different shades to surfaces with the same angle θ between $\overline{N}$ and $\overline{L}$. Although this behavior is correct for a point light source, the objects we see in real life typically are not illuminated by point sources and are not shaded using the simplified illumination models of computer graphics. To complicate matters, early graphics researchers often used a single point light source positioned right at the viewpoint. They expected f_{att} to approximate some of the effects of atmospheric attenuation between the viewer and the object (see Section 16.1.3), as well as the energy density falloff from the light to the object. A useful compromise, which allows a richer range of effects than simple square-law attenuation, is

$$f_{att} = \min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right). \tag{16.8}$$

Here c_1, c_2, and c_3 are user-defined constants associated with the light source. The constant c_1 keeps the denominator from becoming too small when the light is close, and the expression is clamped to a maximum of 1 to ensure that it always attenuates. Figure 16.5 uses this illumination model with different constants to show a range of effects.

Colored lights and surfaces. So far, we have described monochromatic lights and surfaces. Colored lights and surfaces are commonly treated by writing separate equations for each component of the color model. We represent an object's *diffuse color* by one value of O_d for each component. For example, the triple (O_{dR}, O_{dG}, O_{dB}) defines an object's diffuse red, green, and blue components in the RGB color system. In this case, the illuminating light's three primary components, I_{pR}, I_{pG}, and I_{pB}, are reflected in proportion to $k_d O_{dR}$, $k_d O_{dG}$, and $k_d O_{dB}$, respectively. Therefore, for the red component,

$$I_R = I_{aR} k_a O_{dR} + f_{att} I_{pR} k_d O_{dR} (\overline{N} \cdot \overline{L}). \tag{16.9}$$

Similar equations are used for I_G and I_B, the green and blue components. The use of a single coefficient to scale an expression in each of the equations allows the user to control the amount of ambient or diffuse reflection, without altering the proportions of its components. An alternative formulation that is more compact, but less convenient to control, uses a separate coefficient for each component; for example, substituting k_{aR} for $k_a O_{dR}$ and k_{dR} for $k_d O_{dR}$.

A simplifying assumption is made here that a three-component color model can completely model the interaction of light with objects. This assumption is wrong, as we discuss in Section 16.9, but it is easy to implement and often yields acceptable pictures. In theory, the illumination equation should be evaluated continuously over the spectral range being modeled; in practice, it is evaluated for some number of discrete spectral samples.

Samsung/Google, EX1027, p. 737

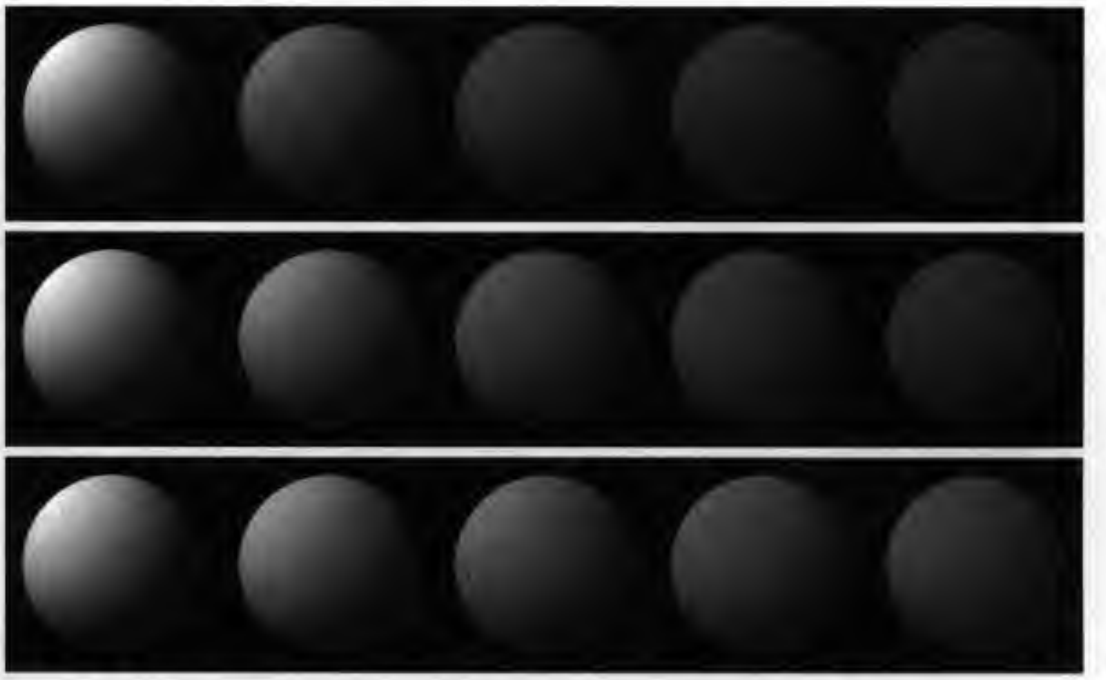


Fig. 16.5 Spheres shaded using ambient and diffuse reflection with a light-source-attenuation term (Eqs. 16.6 and 16.8). For all spheres, $I_a = I_p = 1.0$, $k_a = 0.1$, $k_d = 0.9$. From left to right, sphere's distance from light source is 1.0, 1.375, 1.75, 2.125, 2.5. Top row: $c_1 = c_2 = 0.0$, $c_3 = 1.0$ $(1/d_L^2)$. Middle row: $c_1 = c_2 = 0.25$, $c_3 = 0.5$. Bottom row: $c_1 = 0.0$, $c_2 = 1.0$, $c_3 = 0.0$ $(1/d_L)$. (By David Kurlander, Columbia University.)

Rather than restrict ourselves to a particular color model, we explicitly indicate those terms in an illumination equation that are wavelength-dependent by subscripting them with a λ. Thus, Eq. (16.9) becomes

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} k_d O_{d\lambda} (\overline{N} \cdot \overline{L}). \tag{16.10}$$

16.1.3 Atmospheric Attenuation

To simulate the atmospheric attenuation from the object to the viewer, many systems provide *depth cueing*. In this technique, which originated with vector-graphics hardware, more distant objects are rendered with lower intensity than are closer ones. The PHIGS+ standard recommends a depth-cueing approach that also makes it possible to approximate the shift in colors caused by the intervening atmosphere. Front and back depth-cue reference planes are defined in NPC; each of these planes is associated with a scale factor, s_f and s_b, respectively, that ranges between 0 and 1. The scale factors determine the blending of the original intensity with that of a depth-cue color, $I_{dc\lambda}$. The goal is to modify a previously computed I_λ to yield the depth-cued value I'_λ that is displayed. Given z_o, the object's z coordinate, a scale factor s_o is derived that will be used to interpolate between I_λ and $I_{dc\lambda}$, to determine

$$I'_\lambda = s_o I_\lambda + (1 - s_o) I_{dc\lambda}. \tag{16.11}$$

Samsung/Google, EX1027, p. 738

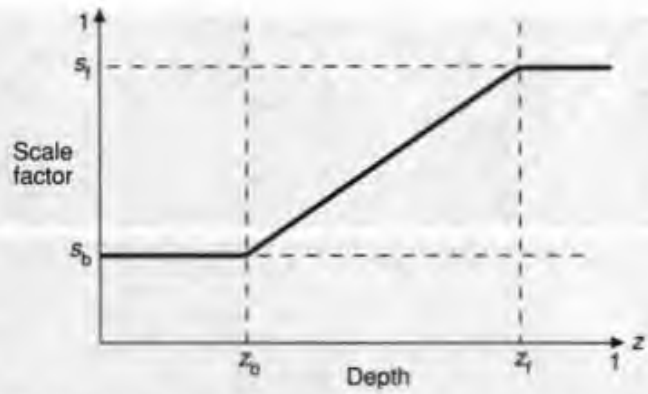


Fig. 16.6 Computing the scale factor for atmospheric attenuation.

If z_o is in front of the front depth-cue reference plane's z coordinate z_f, then $s_o = s_f$. If z_o is behind the back depth-cue reference plane's z coordinate z_b, then $s_o = s_b$. Finally, if z_o is between the two planes, then

$$s_o = s_b + \frac{(z_o - z_b)(s_f - s_b)}{z_f - z_b}. \tag{16.12}$$

The relationship between s_o and z_o is shown in Fig. 16.6. Figure 16.7 shows spheres shaded with depth cueing. To avoid complicating the equations, we ignore depth cueing as we develop the illumination model further. More realistic ways to model atmospheric effects are discussed in Section 20.8.2.

16.1.4 Specular Reflection

Specular reflection can be observed on any shiny surface. Illuminate an apple with a bright white light: The highlight is caused by specular reflection, whereas the light reflected from the rest of the apple is the result of diffuse reflection. Also note that, at the highlight, the apple appears to be not red, but white, the color of the incident light. Objects such as waxed apples or shiny plastics have a transparent surface; plastics, for example, are typically composed of pigment particles embedded in a transparent material. Light specularly reflected from the colorless surface has much the same color as that of the light source.



Fig. 16.7 Spheres shaded using depth cueing (Eqs. 16.5, 16.11, and 16.12). Distance from light is constant. For all spheres, $I_a = I_p = 1.0$, $k_a = 0.1$, $k_d = 0.9$, $z_f = 1.0$, $z_b = 0.0$, $s_f = 1.0$, $s_b = 0.1$, radius = 0.09. From left to right, z at front of sphere is 1.0, 0.77, 0.55, 0.32, 0.09. (By David Kurlander, Columbia University.)

Samsung/Google, EX1027, p. 739

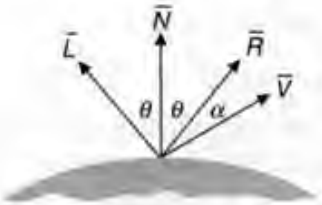


Fig. 16.8 Specular reflection.

Now move your head and notice how the highlight also moves. It does so because shiny surfaces reflect light unequally in different directions; on a perfectly shiny surface, such as a perfect mirror, light is reflected *only* in the direction of reflection $\bar{R}$, which is $\bar{L}$ mirrored about $\bar{N}$. Thus the viewer can see specularly reflected light from a mirror only when the angle α in Fig. 16.8 is zero; α is the angle between $\bar{R}$ and the direction to the viewpoint $\bar{V}$.

The Phong illumination model. Phong Bui-Tuong [BUIT75] developed a popular illumination model for nonperfect reflectors, such as the apple. It assumes that maximum specular reflectance occurs when α is zero and falls off sharply as α increases. This rapid falloff is approximated by $\cos^n \alpha$, where n is the material's *specular-reflection exponent*. Values of n typically vary from 1 to several hundred, depending on the surface material being simulated. A value of 1 provides a broad, gentle falloff, whereas higher values simulate a sharp, focused highlight (Fig. 16.9). For a perfect reflector, n would be infinite. As before, we treat a negative value of $\cos \alpha$ as zero. Phong's illumination model is based on earlier work by researchers such as Warnock [WARN69], who used a $\cos^n \theta$ term to model specular reflection with the light at the viewpoint. Phong, however, was the first to account for viewers and lights at arbitrary positions.

The amount of incident light specularly reflected depends on the angle of incidence θ. If $W(\theta)$ is the fraction of specularly reflected light, then Phong's model is

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} [k_d O_{d\lambda} \cos \theta + W(\theta) \cos^n \alpha]. \tag{16.13}$$

If the direction of reflection $\bar{R}$, and the viewpoint direction $\bar{V}$ are normalized, then $\cos \alpha = \bar{R} \cdot \bar{V}$. In addition, $W(\theta)$ is typically set to a constant k_s, the material's *specular-reflection coefficient*, which ranges between 0 and 1. The value of k_s is selected experimentally to

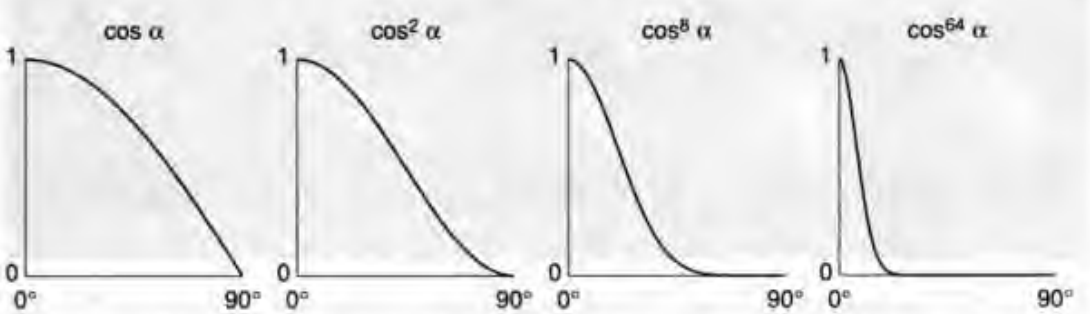


Fig. 16.9 Different values of $\cos^n \alpha$ used in the Phong illumination model.

Samsung/Google, EX1027, p. 740

produce aesthetically pleasing results. Then, Eq. (16.13) can be rewritten as

$$I_\lambda = I_{a\lambda}k_a O_{d\lambda} + f_{att}I_{p\lambda}[k_d O_{d\lambda}(\overline{N} \cdot \overline{L}) + k_s (\overline{R} \cdot \overline{V})^n], \tag{16.14}$$

Note that the color of the specular component in Phong's illumination model is *not* dependent on any material property; thus, this model does a good job of modeling specular reflections from plastic surfaces. As we discuss in Section 16.7, specular reflection is affected by the properties of the surface itself, and, in general, may have a different color than diffuse reflection when the surface is a composite of several materials. We can accommodate this effect to a first approximation by modifying Eq. (16.14) to yield

$$I_\lambda = I_{a\lambda}k_a O_{d\lambda} + f_{att}I_{p\lambda}[k_d O_{d\lambda}(\overline{N} \cdot \overline{L}) + k_s O_{s\lambda} (\overline{R} \cdot \overline{V})^n], \tag{16.15}$$

where $O_{s\lambda}$ is the object's *specular color*. Figure 16.10 shows a sphere illuminated using Eq. (16.14) with different values of k_s and n.

Calculating the reflection vector. Calculating $\overline{R}$ requires mirroring $\overline{L}$ about $\overline{N}$. As shown in Fig. 16.11, this can be accomplished with some simple geometry. Since $\overline{N}$ and $\overline{L}$ are normalized, the projection of $\overline{L}$ onto $\overline{N}$ is $\overline{N} \cos\theta$. Note that $\overline{R} = \overline{N} \cos\theta + \overline{S}$, where $|\overline{S}|$ is sin θ. But, by vector subtraction and congruent triangles, $\overline{S}$ is just $\overline{N} \cos\theta - \overline{L}$. Therefore, $\overline{R} = 2\,\overline{N} \cos\theta - \overline{L}$. Substituting $\overline{N} \cdot \overline{L}$ for cos θ and $\overline{R} \cdot \overline{V}$ for cos α yields

$$\overline{R} = 2\overline{N}\,(\overline{N} \cdot \overline{L}) - \overline{L}, \tag{16.16}$$

$$\overline{R} \cdot \overline{V} = (2\overline{N}\,(\overline{N} \cdot \overline{L}) - \overline{L}) \cdot \overline{V}. \tag{16.17}$$



Fig. 16.10 Spheres shaded using Phong's illumination model (Eq. 16.14) and different values of k_s and n. For all spheres, $I_a = I_p = 1.0$, $k_a = 0.1$, $k_d = 0.45$. From left to right, n = 3.0, 5.0, 10.0, 27.0, 200.0. From top to bottom, k_s = 0.1, 0.25, 0.5. (By David Kurlander, Columbia University.)

Samsung/Google, EX1027, p. 741

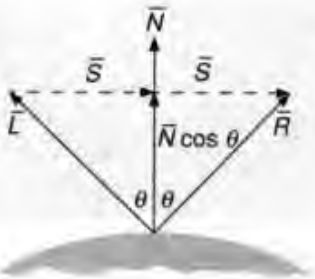


Fig. 16.11 Calculating the reflection vector.

If the light source is at infinity, $\overline{N} \cdot \overline{L}$ is constant for a given polygon, whereas $\overline{R} \cdot \overline{V}$ varies across the polygon. For curved surfaces or for a light source not at infinity, both $\overline{N} \cdot \overline{L}$ and $\overline{R} \cdot \overline{V}$ vary across the surface.

The halfway vector. An alternative formulation of Phong's illumination model uses the *halfway vector* $\overline{H}$, so called because its direction is halfway between the directions of the light source and the viewer, as shown in Fig. 16.12. $\overline{H}$ is also known as the direction of maximum highlights. If the surface were oriented so that its normal were in the same direction as $\overline{H}$, the viewer would see the brightest specular highlight, since $\overline{R}$ and $\overline{V}$ would also point in the same direction. The new specular-reflection term can be expressed as $(\overline{N} \cdot \overline{H})^n$, where $\overline{H} = (\overline{L} + \overline{V}) / |\overline{L} + \overline{V}|$. When the light source and the viewer are both at infinity, then the use of $\overline{N} \cdot \overline{H}$ offers a computational advantage, since $\overline{H}$ is constant. Note that β, the angle between $\overline{N}$ and $\overline{H}$, is not equal to α, the angle between $\overline{R}$ and $\overline{V}$, so the same specular exponent n produces different results in the two formulations (see Exercise 16.1). Although using a cos^n term allows the generation of recognizably glossy surfaces, you should remember that it is based on empirical observation, not on a theoretical model of the specular-reflection process.

16.1.5 Improving the Point-Light-Source Model

Real light sources do not radiate equally in all directions. Warn [WARN83] has developed easily implemented lighting controls that can be added to any illumination equation to model some of the directionality of the lights used by photographers. In Phong's model, a point light source has only an intensity and a position. In Warn's model, a light L is modeled by a point on a hypothetical specular reflecting surface, as shown in Fig. 16.13. This surface is illuminated by a point light source L' in the direction $\overline{L}'$. Assume that $\overline{L}'$ is normal to the hypothetical reflecting surface. Then, we can use the Phong illumination

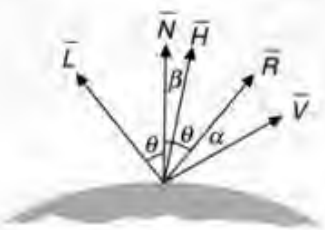


Fig. 16.12 $\overline{H}$, the halfway vector, is halfway between the direction of the light source and the viewer.

Samsung/Google, EX1027, p. 742

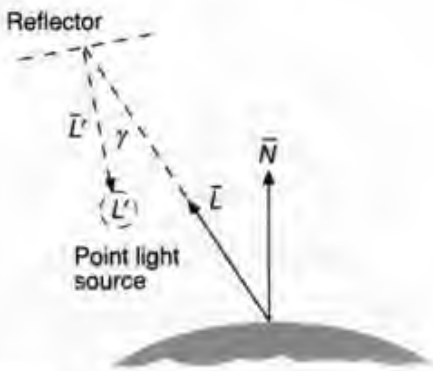


Fig. 16.13 Warn's lighting model. A light is modeled as the specular reflection from a single point illuminated by a point light source.

equation to determine the intensity of L at a point on the object in terms of the angle γ between $\bar{L}$ and $\bar{L}'$. If we further assume that the reflector reflects only specular light and has a specular coefficient of 1, then the light's intensity at a point on the object is

$$I_{L'\lambda} \cos^p \gamma, \tag{16.18}$$

where $I_{L'\lambda}$ is the intensity of the hypothetical point light source, p is the reflector's specular exponent, and γ is the angle between $-\bar{L}$ and the hypothetical surface's normal, $\bar{L}'$, which is the direction to L'. Equation (16.18) models a symmetric directed light source whose axis of symmetry is $\bar{L}'$, the direction in which the light may be thought of as pointing. Using dot products, we can write Eq.(16.18) as

$$I_{L'\lambda} (-\bar{L} \cdot \bar{L}')^p. \tag{16.19}$$

Once again, we treat a negative dot product as zero. Equation (16.19) can thus be substituted for the light-source intensity $I_{p\lambda}$ in the formulation of Eq. (16.15) or any other illumination equation. Contrast the intensity distribution of the uniformly radiating point source with the $\cos^p$ distribution of the Warn light source in Fig. 16.14. Each distribution is plotted in cross-section, showing intensity as a function of angular direction around the light's axis in polar coordinates. $\bar{L}'$ is shown as an arrow. These plots are called *goniometric diagrams*. The larger the value of p, the more the light is concentrated along $\bar{L}'$. Thus, a

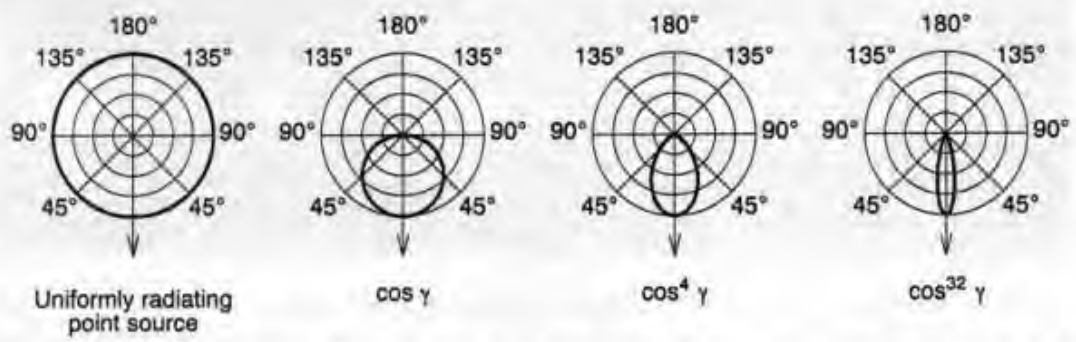


Fig. 16.14 Intensity distributions for uniformly radiating point source and Warn light source with different values of p.

Samsung/Google, EX1027, p. 743

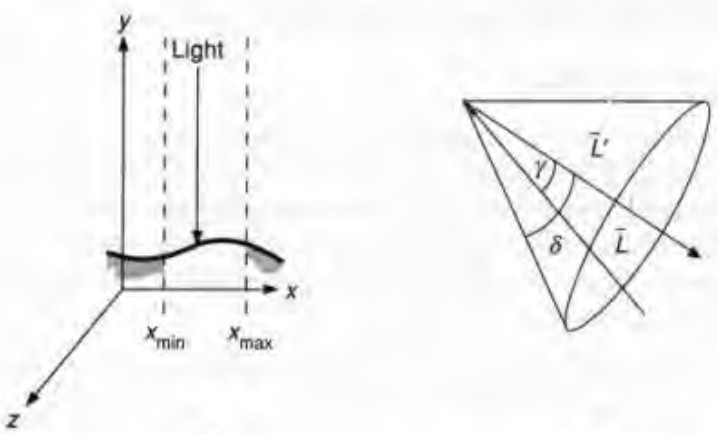


(a) (b) (c) (d) (e)

Fig. 16.15 Cube and plane illuminated using Warn lighting controls. (a) Uniformly radiating point source (or $p = 0$). (b) $p = 4$. (c) $p = 32$. (d) Flaps. (e) Cone with $\delta = 18°$. (By David Kurlander, Columbia University.)

large value of p can simulate a highly directional spotlight, whereas a small value of p can simulate a more diffuse floodlight. If p is 0, then the light acts like a uniformly radiating point source. Figure 16.15(a–c) shows the effects of different values of p. Verbeck [VERB84] and Nishita et al. [NISH85b] have modeled point light sources with more complex irregular intensity and spectral distributions. In general, however, once we determine a point light source's intensity as seen from a particular direction, this value can be used in any illumination equation.

To restrict a light's effects to a limited area of the scene, Warn implemented *flaps* and *cones*. Flaps, modeled loosely after the "barn doors" found on professional photographic lights, confine the effects of the light to a designated range in x, y, and z world coordinates. Each light has six flaps, corresponding to user-specified minimum and maximum values in each coordinate. Each flap also has a flag indicating whether it is on or off. When a point's shade is determined, the illumination model is evaluated for a light only if the point's coordinates are within the range specified by the minimum and maximum coordinates of those flaps that are on. For example, if $\bar{L}'$ is parallel to the y axis, then the x and z flaps can sharply restrict the light's effects, much like the photographic light's barn doors. Figure 16.16(a) shows the use of the x flaps in this situation. The y flaps can also be used here to



(a) (b)

Fig. 16.16 The Warn intensity distribution may be restricted with (a) flaps and (b) cones.

Samsung/Google, EX1027, p. 744

restrict the light in a way that has no physical counterpart, allowing only objects within a specified range of distances from the light to be illuminated. In Fig. 16.15(d) the cube is aligned with the coordinate system, so two pairs of flaps can produce the effects shown.

Warn makes it possible to create a sharply delineated spotlight through the use of a cone whose apex is at the light source and whose axis lies along $\overline{L}'$. As shown in Fig. 16.16(b), a cone with a generating angle of δ may be used to restrict the light source's effects by evaluating the illumination model only when $\gamma < \delta$ (or when $\cos\gamma > \cos\delta$, since $\cos\gamma$ has already been calculated). The PHIGS+ illumination model includes the Warn $\cos^p\gamma$ term and cone angle δ. Figure 16.15(e) demonstrates the use of a cone to restrict the light of Fig. 16.15(c). Color Plate II.17 shows a car rendered with Warn's lighting controls.

16.1.6 Multiple Light Sources

If there are m light sources, then the terms for each light source are summed:

$$I_\lambda = I_{a\lambda}k_a O_{d\lambda} + \sum_{1 \le i \le m} f_{att_i} I_{p\lambda_i} [k_d O_{d\lambda}(\overline{N} \cdot \overline{L}_i) + k_s O_{s\lambda} (\overline{R}_i \cdot \overline{V})^n]. \qquad (16.20)$$

The summation harbors a new possibility for error in that I_λ can now exceed the maximum displayable pixel value. (Although this can also happen for a single light, we can easily avoid it by an appropriate choice of f_{att} and the material coefficients.) Several approaches can be used to avoid overflow. The simplest is to clamp each I_λ individually to its maximum value. Another approach considers all of a pixel's I_λ values together. If at least one is too big, each is divided by the largest to maintain the hue and saturation at the expense of the value. If all the pixel values can be computed before display, image-processing transformations can be applied to the entire picture to bring the values within the desired range. Hall [HALL89] discusses the tradeoffs of these and other techniques.

16.2 SHADING MODELS FOR POLYGONS

It should be clear that we can shade any surface by calculating the surface normal at each visible point and applying the desired illumination model at that point. Unfortunately, this brute-force shading model is expensive. In this section, we describe more efficient shading models for surfaces defined by polygons and polygon meshes.

16.2.1 Constant Shading

The simplest shading model for a polygon is *constant shading*, also known as *faceted shading* or *flat shading*. This approach applies an illumination model once to determine a single intensity value that is then used to shade an entire polygon. In essence, we are sampling the value of the illumination equation once for each polygon, and holding the value across the polygon to reconstruct the polygon's shade. This approach is valid if several assumptions are true:

1. The light source is at infinity, so $\overline{N} \cdot \overline{L}$ is constant across the polygon face
2. The viewer is at infinity, so $\overline{N} \cdot \overline{V}$ is constant across the polygon face
3. The polygon represents the actual surface being modeled, and is not an approximation to a curved surface.

Samsung/Google, EX1027, p. 745

If a visible-surface algorithm is used that outputs a list of polygons, such as one of the list-priority algorithms, constant shading can take advantage of the ubiquitous single-color 2D polygon primitive.

If either of the first two assumptions is wrong, then, if we are to use constant shading, we need some method to determine a single value for each of $\overline{L}$ and $\overline{V}$. For example, values may be calculated for the center of the polygon, or for the polygon's first vertex. Of course, constant shading does not produce the variations in shade across the polygon that should occur in this situation.

16.2.2 Interpolated Shading

As an alternative to evaluating the illumination equation at each point on the polygon, Wylie, Romney, Evans, and Erdahl [WYLI67] pioneered the use of *interpolated shading*, in which shading information is linearly interpolated across a triangle from values determined for its vertices. Gouraud [GOUR71] generalized this technique to arbitrary polygons. This is particularly easy for a scan-line algorithm that already interpolates the z value across a span from interpolated z values computed for the span's endpoints. For increased efficiency, a difference equation may be used, like that developed in Section 15.4 to determine the z value at each pixel. Although z interpolation is physically correct (assuming that the polygon is planar), note that interpolated shading is not, since it only approximates evaluating the illumination model at each point on the polygon.

Our final assumption, that the polygon accurately represents the surface being modeled, is most often the one that is incorrect, which has a much more substantial effect on the resulting image than does the failure of the other two assumptions. Many objects are curved, rather than polyhedral, yet representing them as a polygon mesh allows the use of efficient polygon visible-surface algorithms. We discuss next how to render a polygon mesh so that it looks as much as possible like a curved surface.

16.2.3 Polygon Mesh Shading

Suppose that we wish to approximate a curved surface by a polygonal mesh. If each polygonal facet in the mesh is shaded individually, it is easily distinguished from neighbors whose orientation is different, producing a "faceted" appearance, as shown in Color Plate II.29. This is true if the polygons are rendered using constant shading, interpolated shading, or even per-pixel illumination calculations, because two adjacent polygons of different orientation have different intensities along their borders. The simple solution of using a finer mesh turns out to be surprisingly ineffective, because the perceived difference in shading between adjacent facets is accentuated by the Mach band effect (discovered by Mach in 1865 and described in detail in [RATL72]), which exaggerates the intensity change at any edge where there is a discontinuity in magnitude or slope of intensity. At the border between two facets, the dark facet looks darker and the light facet looks lighter. Figure 16.17 shows, for two separate cases, the actual and perceived changes in intensity along a surface.

Mach banding is caused by *lateral inhibition* of the receptors in the eye. The more light a receptor receives, the more that receptor inhibits the response of the receptors adjacent to

Samsung/Google, EX1027, p. 746

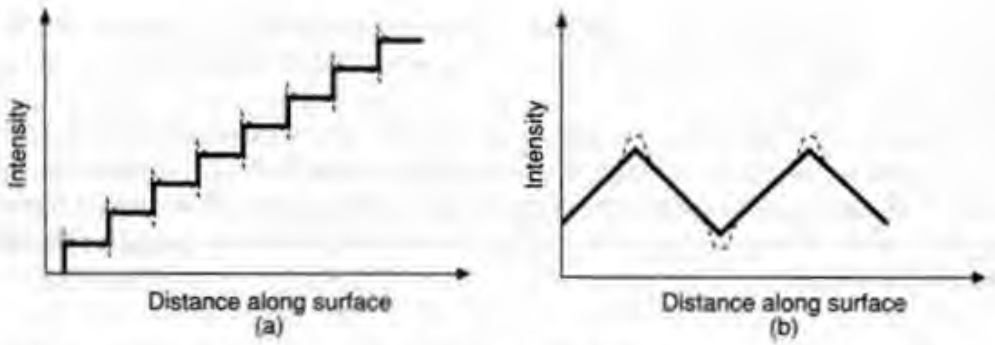


Fig. 16.17 Actual and perceived intensities in the Mach band effect. Dashed lines are perceived intensity; solid lines are actual intensity.

it. The response of a receptor to light is inhibited by its adjacent receptors in inverse relation to the distance to the adjacent receptor. Receptors directly on the brighter side of an intensity change have a stronger response than do those on the brighter side that are farther from the edge, because they receive less inhibition from their neighbors on the darker side. Similarly, receptors immediately to the darker side of an intensity change have a weaker response than do those farther into the darker area, because they receive more inhibition from their neighbors on the brighter side. The Mach band effect is quite evident in Color Plate II.29, especially between adjacent polygons that are close in color.

The polygon-shading models we have described determine the shade of each polygon individually. Two basic shading models for polygon meshes take advantage of the information provided by adjacent polygons to simulate a smooth surface. In order of increasing complexity (and realistic effect), they are known as Gouraud shading and Phong shading, after the researchers who developed them. Current 3D graphics workstations typically support one or both of these approaches through a combination of hardware and firmware.

16.2.4 Gouraud Shading

Gouraud shading [GOUR71], also called *intensity interpolation shading* or *color interpolation shading*, eliminates intensity discontinuities. Color Plate II.30 uses Gouraud shading. Although most of the Mach banding of Color Plate II.29 is no longer visible in Color Plate II.30, the bright ridges on objects such as the torus and cone are Mach bands caused by a rapid, although not discontinuous, change in the slope of the intensity curve; Gouraud shading does not completely eliminate such intensity changes.

Gouraud shading extends the concept of interpolated shading applied to individual polygons by interpolating polygon vertex illumination values that take into account the surface being approximated. The Gouraud shading process requires that the normal be known for each vertex of the polygonal mesh. Gouraud was able to compute these *vertex normals* directly from an analytical description of the surface. Alternatively, if the vertex normals are not stored with the mesh and cannot be determined directly from the actual surface, then, Gouraud suggested, we can approximate them by averaging the surface

Samsung/Google, EX1027, p. 747

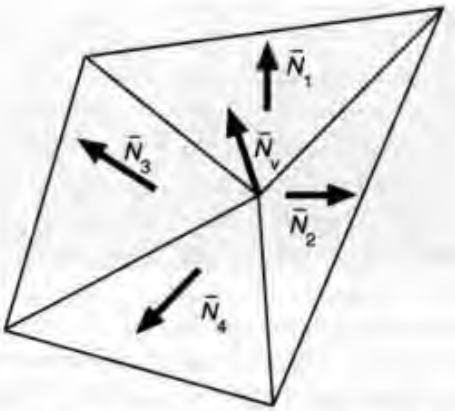


Fig. 16.18 Normalized polygon surface normals may be averaged to obtain vertex normals. Averaged normal $\bar{N}_v$ is $\sum_{1 \le i \le n} \bar{N}_i / | \sum_{1 \le i \le n} \bar{N}_i |$.

normals of all polygonal facets sharing each vertex (Fig. 16.18). If an edge is meant to be visible (as at the joint between a plane's wing and body), then we find two vertex normals, one for each side of the edge, by averaging the normals of polygons on each side of the edge separately. Normals were not averaged across the teapot's patch cracks in Color Plate II.30. (See caption to Color Plate II.21.)

The next step in Gouraud shading is to find *vertex intensities* by using the vertex normals with any desired illumination model. Finally, each polygon is shaded by linear interpolation of vertex intensities along each edge and then between edges along each scan line (Fig. 16.19) in the same way that we described interpolating z values in Section 15.4. The term *Gouraud shading* is often generalized to refer to intensity interpolation shading of even a single polygon in isolation, or to the interpolation of arbitrary colors associated with polygon vertices.

The interpolation along edges can easily be integrated with the scan-line visible-surface algorithm of Section 15.6. With each edge, we store for each color component the starting intensity and the change of intensity for each unit change in y. A visible span on a scan line is filled in by interpolating the intensity values of the two edges bounding the span. As in all linear-interpolation algorithms, a difference equation may be used for increased efficiency.

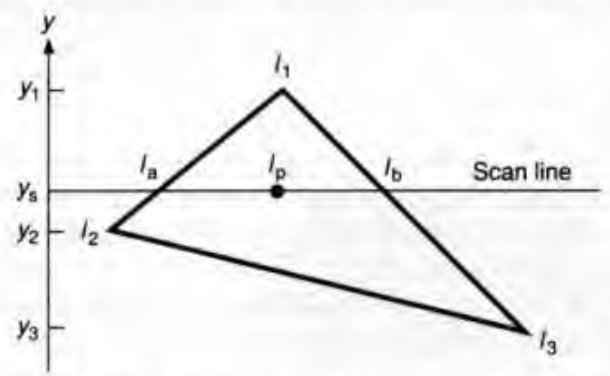


Fig. 16.19 Intensity interpolation along polygon edges and scan lines.

Samsung/Google, EX1027, p. 748

16.2.5 Phong Shading

Phong shading [BUIT75], also known as *normal-vector interpolation shading*, interpolates the surface normal vector $\overline{N}$, rather than the intensity. Interpolation occurs across a polygon span on a scan line, between starting and ending normals for the span. These normals are themselves interpolated along polygon edges from vertex normals that are computed, if necessary, just as in Gouraud shading. The interpolation along edges can again be done by means of incremental calculations, with all three components of the normal vector being incremented from scan line to scan line. At each pixel along a scan line, the interpolated normal is normalized, and is backmapped into the WC system or one isometric to it, and a new intensity calculation is performed using any illumination model. Figure 16.20 shows two edge normals and the normals interpolated from them, before and after normalization.

Color Plates II.31 and II.32 were generated using Gouraud shading and Phong shading respectively, and an illumination equation with a specular-reflectance term. Phong shading yields substantial improvements over Gouraud shading when such illumination models are used, because highlights are reproduced more faithfully, as shown in Fig. 16.21. Consider what happens if n in the Phong $\cos^n \alpha$ illumination term is large and one vertex has a very small α, but each of its adjacent vertices has a large α. The intensity associated with the vertex that has a small α will be appropriate for a highlight, whereas the other vertices will have nonhighlight intensities. If Gouraud shading is used, then the intensity across the polygon is linearly interpolated between the highlight intensity and the lower intensities of the adjacent vertices, spreading the highlight over the polygon (Fig. 16.21a). Contrast this with the sharp drop from the highlight intensity that is computed if linearly interpolated normals are used to compute the $\cos^n \alpha$ term at each pixel (Fig. 16.21b). Furthermore, if a highlight fails to fall at a vertex, then Gouraud shading may miss it entirely (Fig. 16.21c), since no interior point can be brighter than the brightest vertex from which it is interpolated. In contrast, Phong shading allows highlights to be located in a polygon's interior (Fig. 16.21d). Compare the highlights on the ball in Color Plates II.31 and II.32.

Even with an illumination model that does not take into account specular reflectance, the results of normal-vector interpolation are in general superior to intensity interpolation, because an approximation to the normal is used at each point. This reduces Mach-band problems in most cases, but greatly increases the cost of shading in a straightforward implementation, since the interpolated normal must be normalized every time it is used in an illumination model. Duff [DUFF79] has developed a combination of difference

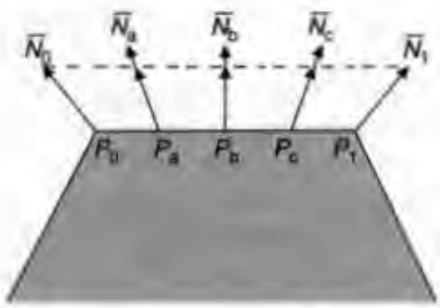


Fig. 16.20 Normal vector interpolation. (After [BUIT75].)

Samsung/Google, EX1027, p. 749



(a) (b) (c) (d)

Fig. 16.21 A specular-reflection illumination model used with Gouraud shading and Phong shading. Highlight falls at left vertex: (a) Gouraud shading, (b) Phong shading. Highlight falls in polygon interior: (c) Gouraud shading, (d) Phong shading. (By David Kurlander, Columbia University.)

equations and table lookup to speed up the calculation. Bishop and Weimer [BISH86] provide an excellent approximation of Phong shading by using a Taylor series expansion that offers even greater increases in shading speed.

Another shading model, intermediate in complexity between Gouraud and Phong shading, involves the linear interpolation of the dot products used in the illumination models. As in Phong shading, the illumination model is evaluated at each pixel, but the interpolated dot products are used to avoid the expense of computing and normalizing any of the direction vectors. This model can produce more satisfactory effects than Gouraud shading when used with specular-reflection illumination models, since the specular term is calculated separately and has power-law, rather than linear, falloff. As in Gouraud shading, however, highlights are missed if they do not fall at a vertex, since no intensity value computed for a set of interpolated dot products can exceed those computed for the set of dot products at either end of the span.

16.2.6 Problems with Interpolated Shading

There are many problems common to all these interpolated-shading models, several of which we list here.

Polygonal silhouette. No matter how good an approximation an interpolated shading model offers to the actual shading of a curved surface, the silhouette edge of the mesh is still clearly polygonal. We can improve this situation by breaking the surface into a greater number of smaller polygons, but at a corresponding increase in expense.

Perspective distortion. Anomalies are introduced because interpolation is performed after perspective transformation in the 3D screen-coordinate system, rather than in the WC system. For example, linear interpolation causes the shading information in Fig. 16.19 to be incremented by a constant amount from one scan line to another along each edge. Consider what happens when vertex 1 is more distant than vertex 2. Perspective foreshortening means that the difference from one scan line to another in the untransformed z value along an edge increases in the direction of the farther coordinate. Thus, if $y_s = (y_1 + y_2) / 2$, then $I_s = (I_1 + I_2) / 2$, but z_s will not equal $(z_1 + z_2) / 2$. This problem can also be

Samsung/Google, EX1027, p. 750

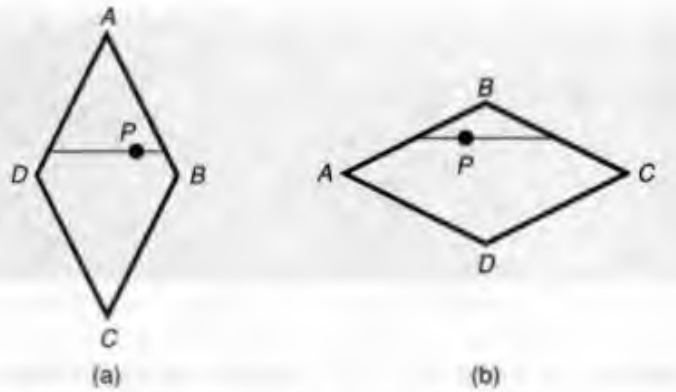


Fig. 16.22 Interpolated values derived for point *P* on the same polygon at different orientations differ from (a) to (b). *P* interpolates *A, B, D* in (a) and *A, B, C* in (b).

reduced by using a larger number of smaller polygons. Decreasing the size of the polygons increases the number of points at which the information to be interpolated is sampled, and therefore increases the accuracy of the shading.

Orientation dependence. The results of interpolated-shading models are not independent of the projected polygon's orientation. Since values are interpolated between vertices and across horizontal scan lines, the results may differ when the polygon is rotated (see Fig. 16.22). This effect is particularly obvious when the orientation changes slowly between successive frames of an animation. A similar problem can also occur in visible-surface determination when the z value at each point is interpolated from the z values assigned to each vertex. Both problems can be solved by decomposing polygons into triangles (see Exercise 16.2). Alternatively, Duff [DUFF79] suggests rotation-independent, but expensive, interpolation methods that solve this problem without the need for decomposition.

Problems at shared vertices. Shading discontinuities can occur when two adjacent polygons fail to share a vertex that lies along their common edge. Consider the three polygons of Fig. 16.23, in which vertex *C* is shared by the two polygons on the right, but not by the large polygon on the left. The shading information determined directly at *C* for the polygons at the right will typically not be the same as the information interpolated at that point from the values at *A* and *B* for the polygon at the left. As a result, there will be a discontinuity in the shading. The discontinuity can be eliminated by inserting in the

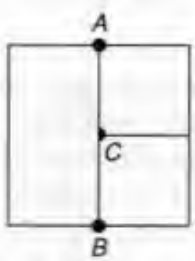


Fig. 16.23 Vertex *C* is shared by the two polygons on the right, but not by the larger rectangular polygon on the left.

Samsung/Google, EX1027, p. 751

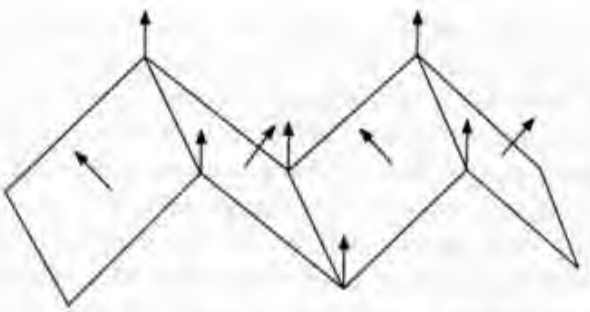


Fig. 16.24 Problems with computing vertex normals. Vertex normals are all parallel.

polygon on the left an extra vertex that shares C's shading information. We can preprocess a static polygonal database in order to eliminate this problem; alternatively, if polygons will be split on the fly (e.g., using the BSP-tree visible-surface algorithm), then extra bookkeeping can be done to introduce a new vertex in an edge that shares an edge that is split.

Unrepresentative vertex normals. Computed vertex normals may not adequately represent the surface's geometry. For example, if we compute vertex normals by averaging the normals of the surfaces sharing a vertex, all of the vertex normals of Fig. 16.24 will be parallel to one another, resulting in little or no variation in shade if the light source is distant. Subdividing the polygons further before vertex normal computation will solve this problem.

Although these problems have prompted much work on rendering algorithms that handle curved surfaces directly, polygons are sufficiently faster (and easier) to process that they still form the core of most rendering systems.

16.3 SURFACE DETAIL

Applying any of the shading models we have described so far to planar or bicubic surfaces produces smooth, uniform surfaces—in marked contrast to most of the surfaces we see and feel. We discuss next a variety of methods developed to simulate this missing surface detail.

16.3.1 Surface-Detail Polygons

The simplest approach adds gross detail through the use of *surface-detail polygons* to show features (such as doors, windows, and lettering) on a base polygon (such as the side of a building). Each surface-detail polygon is coplanar with its base polygon, and is flagged so that it does not need to be compared with other polygons during visible-surface determination. When the base polygon is shaded, its surface-detail polygons and their material properties take precedence for those parts of the base polygon that they cover.

16.3.2 Texture Mapping

As detail becomes finer and more intricate, explicit modeling with polygons or other geometric primitives becomes less practical. An alternative is to map an image, either

Samsung/Google, EX1027, p. 752

digitized or synthesized, onto a surface, a technique pioneered by Catmull [CATM74b] and refined by Blinn and Newell [BLIN76]. This approach is known as *texture mapping* or *pattern mapping*; the image is called a *texture map*, and its individual elements are often called *texels*. The rectangular texture map resides in its own (u, v) texture coordinate space. Alternatively, the texture may be defined by a procedure. Color Plate II.35 shows several examples of texture mapping, using the textures shown in Fig. 16.25. At each rendered pixel, selected texels are used either to substitute for or to scale one or more of the surface's material properties, such as its diffuse color components. One pixel is often covered by a number of texels. To avoid aliasing problems, we must consider all relevant texels.

As shown in Fig. 16.26, texture mapping can be accomplished in two steps. A simple approach starts by mapping the four corners of the pixel onto the surface. For a bicubic patch, this mapping naturally defines a set of points in the surface's (s, t) coordinate space. Next, the pixel's corner points in the surface's (s, t) coordinate space are mapped into the texture's (u, v) coordinate space. The four (u, v) points in the texture map define a quadrilateral that approximates the more complex shape into which the pixel may actually map due to surface curvature. We compute a value for the pixel by summing all texels that lie within the quadrilateral, weighting each by the fraction of the texel that lies within the quadrilateral. If a transformed point in (u, v) space falls outside of the texture map, the texture map may be thought of as being replicated, like the patterns of Section 2.1.3.

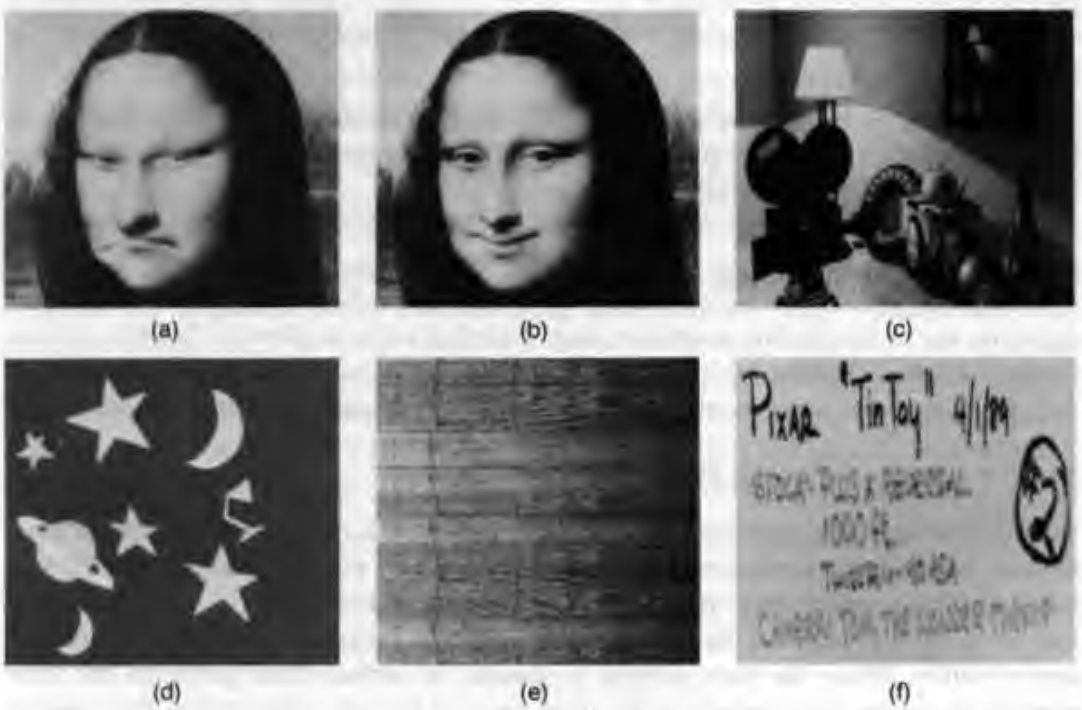


Fig. 16.25 Textures used to create Color Plate II.35. (a) Frowning Mona. (b) Smiling Mona. (c) Painting. (d) Wizard's cap. (e) Floor. (f) Film label. (Copyright © 1990, Pixar. Images rendered by Thomas Williams and H. B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

Samsung/Google, EX1027, p. 753

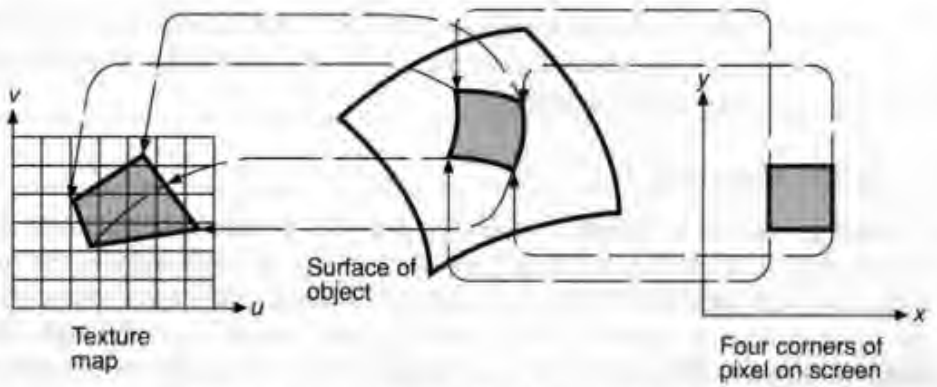


Fig. 16.26 Texture mapping from pixel to the surface to the texture map.

Rather than always use the identity mapping between (s, t) and (u, v), we can define a correspondence between the four corners of the 0-to-1 (s, t) rectangle and a quadrilateral in (u, v). When the surface is a polygon, it is common to assign texture map coordinates directly to its vertices. Since, as we have seen, linearly interpolating values across arbitrary polygons is orientation-dependent, polygons may be decomposed into triangles first. Even after triangulation, however, linear interpolation will cause distortion in the case of perspective projection. This distortion will be more noticeable than that caused when interpolating other shading information, since texture features will not be correctly foreshortened. We can obtain an approximate solution to this problem by decomposing polygons into smaller ones, or an exact solution, at greater cost, by performing the perspective division while interpolating.

The approach just described assumes square pixel geometry and simple box filtering. It also fails to take into account pixels that map to only part of a surface. Feibush, Levoy, and Cook [FEIB80] address these problems for texture-mapping polygons. Think of the square pixel in Fig. 16.26 as the bounding rectangle of the support of an arbitrary filter centered at a pixel. Pixels and texels can then be treated as points. In effect, all texels that lie within the mapped intersection of the transformed bounding rectangle and polygon are selected, and these texels' coordinates are transformed into the coordinate system of the bounding rectangle. Each texel's transformed coordinates are used to index into a filter table to determine the texel's weighting, and the weighted average of the texel intensities is computed. This weighted average must in turn be weighted by the percentage contribution that the polygon makes to the pixel's intensity. The process is repeated for each polygon whose projection intersects the pixel, and the values are summed. Section 17.4.2 discusses this algorithm in more detail.

The Feibush, Levoy, and Cook algorithm can be quite inefficient. Consider mapping a checkerboard pattern onto an infinite ground plane. An extremely large number of texels may have to be weighted and summed just to texture a single distant ground-plane pixel. One solution to this problem is to prefilter the texture and to store the results in a way that is space-efficient and that allows quick determination of the weighted average of texels mapping to a pixel. Algorithms by Williams [WILL83], Crow [CROW84], Glassner [GLAS86], and Heckbert [HECK86a] that take this approach are discussed in Section

Samsung/Google, EX1027, p. 754

17.4.3. Catmull and Smith's efficient technique [CATM80] for mapping an entire texture map directly to a surface is discussed in Exercise 17.10. Heckbert [HECK86] provides a thorough survey of texture-mapping methods.

16.3.3 Bump Mapping

Texture mapping affects a surface's shading, but the surface continues to appear geometrically smooth. If the texture map is a photograph of a rough surface, the surface being shaded will not look quite right, because the direction to the light source used to create the texture map is typically different from the direction to the light source illuminating the surface. Blinn [BLIN78b] developed a way to provide the appearance of modified surface geometry that avoids explicit geometrical modeling. His approach involves perturbing the surface normal before it is used in the illumination model, just as slight roughness in a surface would perturb the surface normal. This method is known as *bump mapping*, and is based on texture mapping.

A *bump map* is an array of displacements, each of which can be used to simulate displacing a point on a surface a little above or below that point's actual position. Let us represent a point on a surface by a vector $\overline{P}$, where $\overline{P} = [x(s, t), y(s, t), z(s, t)]$. We call the partial derivatives of the surface at $\overline{P}$ with respect to the surface's s and t parameterization axes, $\overline{P}_s$ and $\overline{P}_t$. Since each is tangent to the surface, their cross-product forms the (unnormalized) surface normal at $\overline{P}$. Thus,

$$\overline{N} = \overline{P}_s \times \overline{P}_t. \tag{16.21}$$

We can displace point $\overline{P}$ by adding to it the normalized normal scaled by a selected bump-map value B. The new point is

$$\overline{P}' = \overline{P} + \frac{B\overline{N}}{|\overline{N}|}. \tag{16.22}$$

Blinn shows that a good approximation to the new (unnormalized) normal $\overline{N}'$ is

$$\overline{N}' = \overline{N} + \frac{B_u(\overline{N} \times \overline{P}_t) - B_v(\overline{N} \times \overline{P}_s)}{|\overline{N}|}, \tag{16.23}$$

where B_u and B_v are the partial derivatives of the selected bump-map entry B with respect to the bump-map parameterization axes, u and v. $\overline{N}'$ is then normalized and substituted for the surface normal in the illumination equation. Note that only the partial derivatives of the bump map are used in Eq. (16.23), not its values. Bilinear interpolation can be used to derive bump-map values for specified (u, v) positions, and finite differences can be used to compute B_u and B_v.

The results of bump mapping can be quite convincing. Viewers often fail to notice that an object's texture does not affect its silhouette edges. Color Plates III.3 and III.4 show two examples of bump mapping. Unlike texture mapping, aliasing cannot be dealt with by filtering values from the bump map, since these values do not correspond linearly to intensities; filtering the bump map just smooths out the bumps. Instead, subpixel intensities may be computed and filtered for each pixel, or some prefiltering may be performed on the bump map to improve gross aliasing.

Samsung/Google, EX1027, p. 755

16.3.4 Other Approaches

Although 2D mapping can be effective in many situations, it often fails to produce convincing results. Textures frequently betray their 2D origins when mapped onto curved surfaces, and problems are encountered at texture "seams." For example, when a wood-grain texture is mapped onto the surface of a curved object, the object will look as if it were painted with the texture. Peachey [PEAC85] and Perlin [PERL85] have investigated the use of solid textures for proper rendering of objects "carved" of wood or marble, as exemplified by Color Plate IV.21. In this approach, described in Section 20.8.3, the texture is a 3D function of its position in the object.

Other surface properties can be mapped as well. For example, Gardner [GARD84] has used transparency mapping to make impressionistic trees and clouds from otherwise simple shapes, as described in Section 20.8.2. Color Plate IV.24 shows the application of a complex functional transparency texture to objects formed from groups of quadric surfaces. Cook has implemented *displacement mapping*, in which the actual surface is displaced, instead of only the surface normals [COOK84a]; this process, which must be carried out before visible-surface determination, was used to modify the surfaces of the cone and torus in Color Plate II.36. Using fractals to create richly detailed geometry from an initial simple geometric description is discussed in Section 20.3.

So far, we have made the tacit assumption that the process of shading a point on an object is unaffected by the rest of that object or by any other object. But an object might in fact be shadowed by another object between it and a light source; might transmit light, allowing another object to be seen through it; or might reflect other objects, allowing another object to be seen because of it. In the following sections, we describe how to model these effects.

16.4 SHADOWS

Visible-surface algorithms determine which surfaces can be seen from the viewpoint; shadow algorithms determine which surfaces can be "seen" from the light source. Thus, visible-surface algorithms and shadow algorithms are essentially the same. The surfaces that are visible from the light source are not in shadow; those that are not visible from the light source are in shadow. When there are multiple light sources, a surface must be classified relative to each of them.

Here, we consider shadow algorithms for point light sources; extended light sources are discussed in Sections 16.8, 16.12, and 16.13. Visibility from a point light source is, like visibility from the viewpoint, all or nothing. When a point on a surface cannot be seen from a light source, then the illumination calculation must be adjusted to take it into account. The addition of shadows to the illumination equation yields

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \le i \le m} S_i f_{att_i} I_{p\lambda_i} [k_d O_{d\lambda} (\overline{N} \cdot \overline{L}_i) + k_s O_{s\lambda} (\overline{R}_i \cdot \overline{V})^n], \qquad (16.24)$$

where

$$S_i = \begin{cases} 0, & \text{if light } i \text{ is blocked at this point;} \\ 1, & \text{if light } i \text{ is not blocked at this point.} \end{cases}$$

Samsung/Google, EX1027, p. 756

Note that areas in the shadow of all point light sources are still illuminated by the ambient light.

Although computing shadows requires computing visibility from the light source, as we have pointed out, it is also possible to generate "fake" shadows without performing any visibility tests. These can be created efficiently by transforming each object into its polygonal projection from a point light source onto a designated ground plane, without clipping the transformed polygon to the surface that it shadows or checking for whether it is blocked by intervening surfaces [BLIN88]. These shadows are then treated as surface-detail polygons. For the general case, in which these fake shadows are not adequate, various approaches to shadow generation are possible. We could perform all shadow processing first, interleave it with visible-surface processing in a variety of ways, or even do it after visible-surface processing has been performed. Here we examine algorithms that follow each of these approaches, building on the classification of shadow algorithms presented in [CROW77a]. To simplify the explanations, we shall assume that all objects are polygons unless otherwise specified.

16.4.1 Scan-Line Generation of Shadows

One of the oldest methods for generating shadows is to augment a scan-line algorithm to interleave shadow and visible-surface processing [APPEL68; BOUK70b]. Using the light source as a center of projection, the edges of polygons that might potentially cast shadows are projected onto the polygons intersecting the current scan line. When the scan crosses one of these shadow edges, the colors of the image pixels are modified accordingly.

A brute-force implementation of this algorithm must compute all $n(n-1)$ projections of every polygon on every other polygon. Bouknight and Kelley [BOUK70b] instead use a clever preprocessing step in which all polygons are projected onto a sphere surrounding the light source, with the light source as center of projection. Pairs of projections whose extents do not overlap can be eliminated, and a number of other special cases can be identified to limit the number of polygon pairs that need be considered by the rest of the algorithm. The authors then compute the projection from the light source of each polygon onto the plane of each of those polygons that they have determined it could shadow, as shown in Fig. 16.27. Each of these shadowing polygon projections has associated information about the polygons casting and potentially receiving the shadow. While the scan-line algorithm's regular scan keeps track of which regular polygon edges are being crossed, a separate, parallel shadow scan keeps track of which shadowing polygon projection edges are crossed, and thus which shadowing polygon projections the shadow scan is currently "in." When the shade for a span is computed, it is in shadow if the shadow scan is "in" one of the shadow projections cast on the polygon's plane. Thus span *bc* in Fig. 16.27(a) is in shadow, while spans *ab* and *cd* are not. Note that the algorithm does not need to clip the shadowing polygon projections analytically to the polygons being shadowed.

16.4.2 A Two-Pass Object-Precision Shadow Algorithm

Atherton, Weiler, and Greenberg have developed an algorithm that performs shadow determination before visible-surface determination [ATHE78]. They process the object description by using the same algorithm twice, once for the viewpoint, and once for the

Samsung/Google, EX1027, p. 757

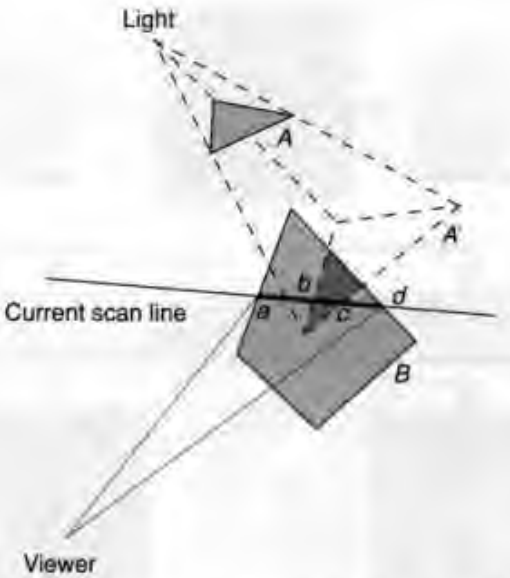


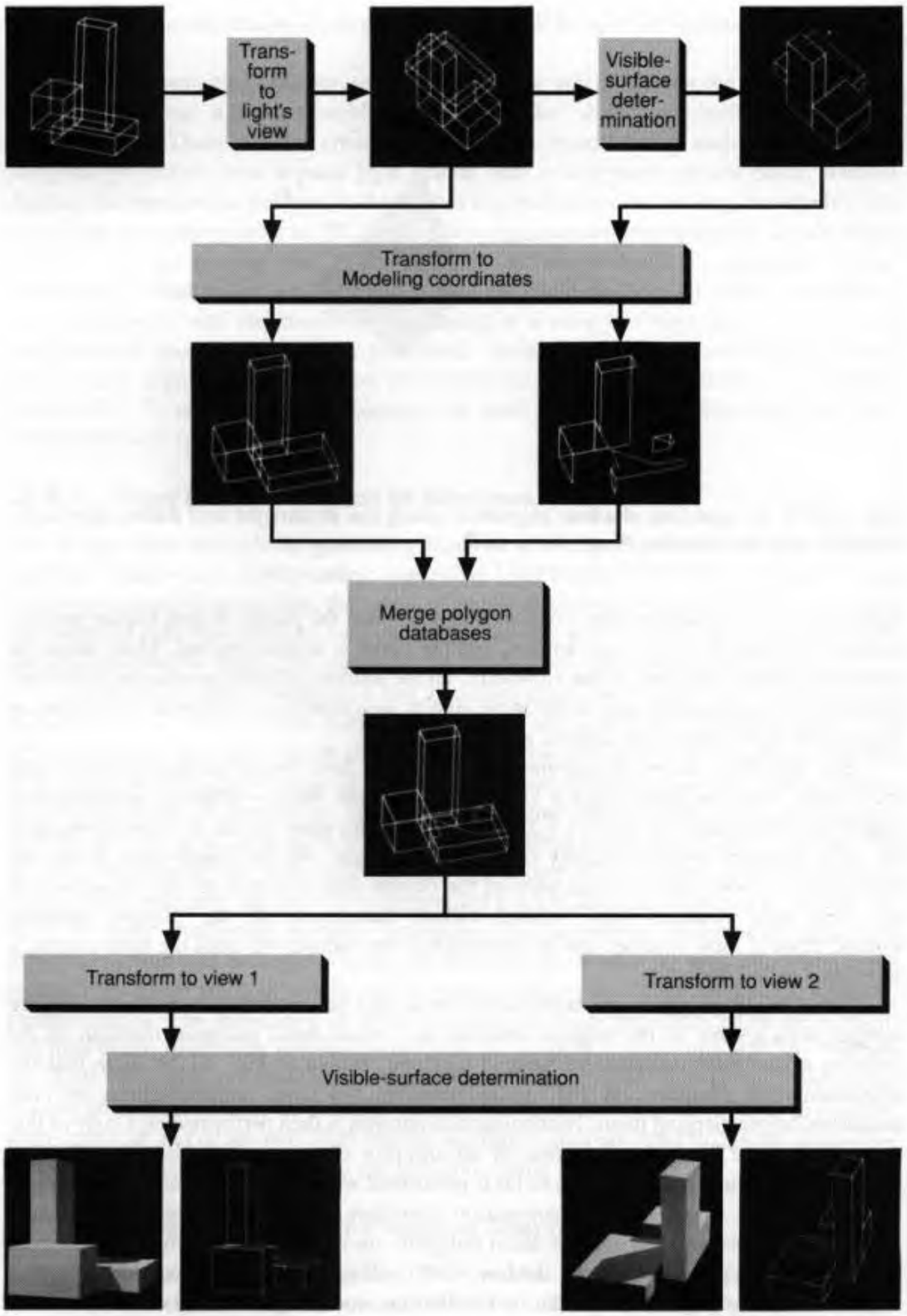
Fig. 16.27 A scan-line shadow algorithm using the Bouknight and Kelley approach. Polygon *A* casts shadow *A'* on plane of *B*.

light source. The results are then combined to determine the pieces of each visible part of a polygon that are lit by the light source, and the scene is scan-converted. Thus, since the shadows are not dependent on the viewpoint, all the shadow calculations may be performed just once for a series of images of the same objects seen from many different viewpoints, as long as the light source and objects are fixed.

The algorithm, shown in overview in Fig. 16.28, first determines those surfaces that are visible from the light source's viewpoint, using the Weiler–Atherton visible-surface algorithm discussed in Section 15.7.2. The output of this pass is a list of lit polygons, each of which is tagged with the identity of its parent polygon. All the objects must fit into the light source's view volume, since parts of the objects that do not fit are not recognized as lit. If a light source's view volume cannot encompass all the objects, multiple nonoverlapping view volumes can be constructed that radiate out from the light source, a technique called *sectoring*.

Next, the lit polygons are transformed back into the modeling coordinates and are merged with a copy of the original database as surface-detail polygons (Section 16.3), creating a viewpoint-independent merged database, shown in Fig. 16.29. Note that the implementation illustrated in Fig. 16.28 performs the same transformations on both databases before merging them. Hidden-surface removal is then performed on a copy of this merged database from the viewpoint of an arbitrary observer, again using the Weiler–Atherton algorithm. All processing so far is performed with object precision and results in a list of polygons. A polygon scan-conversion algorithm is then used to render the image. Visible surfaces covered by surface-detail polygons are rendered as lit, whereas uncovered visible surfaces are rendered in shadow. Color Plate III.5 was generated using this approach. Multiple light sources can be handled by processing the merged database from the viewpoint of each new light source, merging the results of each pass.

Samsung/Google, EX1027, p. 758



Samsung/Google, EX1027, p. 759

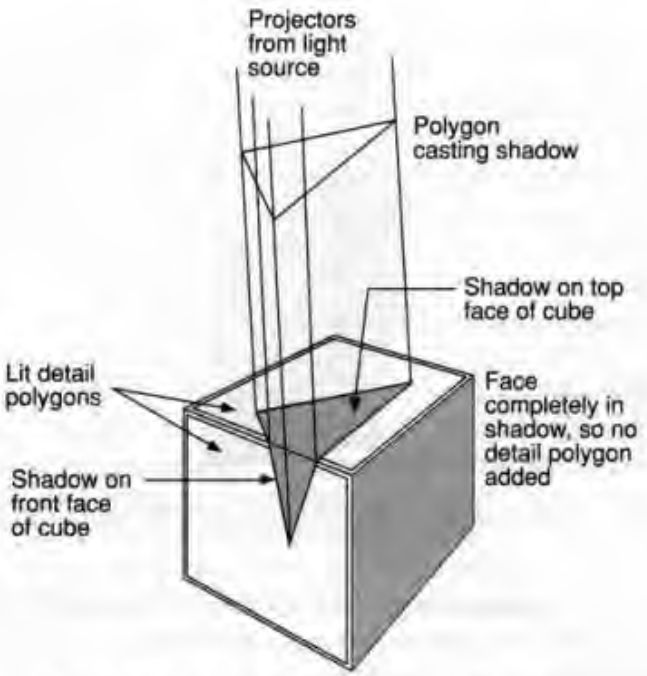


Fig. 16.29 Lit surface-detail polygons.

16.4.3 Shadow Volumes

Crow [CROW77a] describes how to generate shadows by creating for each object a *shadow volume* that the object blocks from the light source. A shadow volume is defined by the light source and an object and is bounded by a set of invisible *shadow polygons*. As shown in Fig. 16.30, there is one quadrilateral shadow polygon for each silhouette edge of the object relative to the light source. Three sides of a shadow polygon are defined by a silhouette edge of the object and the two lines emanating from the light source and passing through that edge's endpoints. Each shadow polygon has a normal that points out of the shadow volume. Shadow volumes are generated only for polygons facing the light. In the implementation described by Bergeron [BERG86a], the shadow volume—and hence each of its shadow polygons—is capped on one end by the original object polygon and on the other end by a scaled copy of the object polygon whose normal has been inverted. This scaled copy is located at a distance from the light beyond which its attenuated energy density is assumed to be negligible. We can think of this distance as the light's *sphere of influence*. Any point outside of the sphere of influence is effectively in shadow and does not require any additional shadow processing. In fact, there is no need to generate a shadow volume for any object wholly outside the sphere of influence. We can generalize this approach to apply to nonuniformly radiating sources by considering a *region of influence*, for example by culling

Fig. 16.28 Shadow creation and display in the Atherton, Weiler, and Greenberg algorithm. (Images by Peter Atherton, Kevin Weiler, Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1978.)

Samsung/Google, EX1027, p. 760

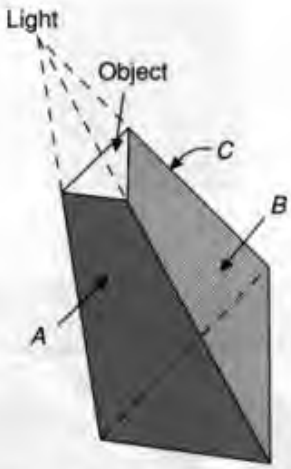


Fig. 16.30 A shadow volume is defined by a light source and an object.

objects outside of a light's flaps and cone. The shadow volume may also be further clipped to the view volume if the view volume is known in advance. The cap polygons are also treated as shadow polygons by the algorithm.

Shadow polygons are not rendered themselves, but are used to determine whether the other objects are in shadow. Relative to the observer, a front-facing shadow polygon (polygon *A* or *B* in Fig. 16.30) causes those objects behind it to be shadowed; a back-facing shadow polygon (polygon *C*) cancels the effect of a front-facing one. Consider a vector from the viewpoint *V* to a point on an object. The point is in shadow if the vector intersects more front-facing than back-facing shadow polygons. Thus, points *A* and *C* in Fig. 16.31(a) are in shadow. This is the only case in which a point is shadowed when *V* is not shadowed; therefore, point *B* is lit. If *V* is in shadow, there is one additional case in which a point is shadowed: when all the back-facing shadow polygons for the object polygons shadowing the eye have not yet been encountered. Thus, points *A*, *B*, and *C* in Fig. 16.31(b) are in shadow,

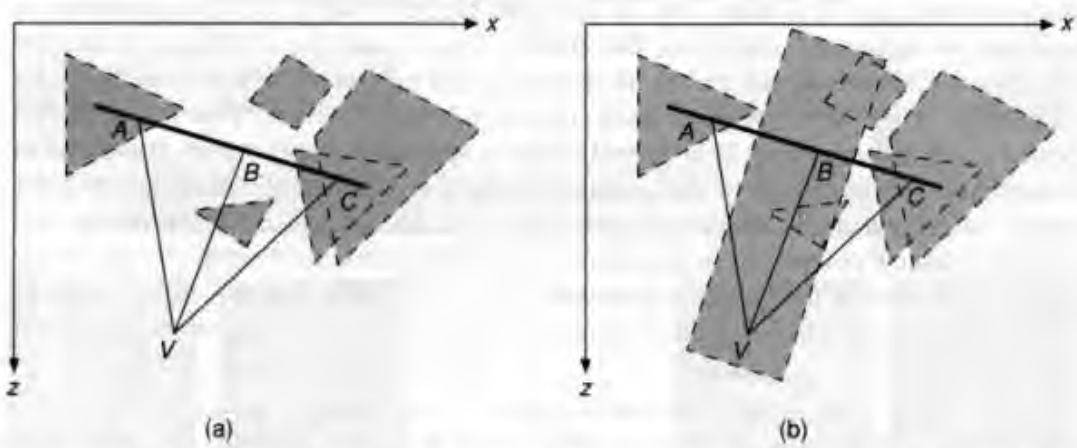


Fig. 16.31 Determining whether a point is in shadow for a viewer at *V*. Dashed lines define shadow volumes (shaded in gray). (a) *V* is not in shadow. Points *A* and *C* are shadowed; point *B* is lit. (b) *V* is in shadow. Points *A*, *B*, and *C* are shadowed.

Samsung/Google, EX1027, p. 761

even though the vector from V to B intersects the same number of front-facing and back-facing shadow polygons as it does in part (a).

We can compute whether a point is in shadow by assigning to each front-facing (relative to the viewer) shadow polygon a value of $+1$ and to each back-facing shadow polygon a value of -1. A counter is initially set to the number of shadow volumes that contain the eye and is incremented by the values associated with all shadow polygons between the eye and the point on the object. The point is in shadow if the counter is positive at the point. The number of shadow volumes containing the eye is computed only once for each viewpoint, by taking the negative of the sum of the values of all shadow polygons intercepted by an arbitrary projector from the eye to infinity.

Although is is possible to compute a shadow volume for each polygon, we can take advantage of object coherence by computing a single shadow volume for each connected polyhedron. This can be accomplished by generating shadow polygons from only those edges that are silhouette edges relative to the light source; these are the contour edges relative to the light source (as defined in Section 15.3.2).

Multiple light sources can be handled by building a separate set of shadow volumes for each light source, marking the volume's shadow polygons with their light source identifier, and keeping a separate counter for each light source. Brotman and Badler [BROT84] have implemented a z-buffer version of the shadow-volume algorithm, and Bergeron [BERG86a] discusses a scan-line implementation that efficiently handles arbitrary polyhedral objects containing nonplanar polygons.

Chin and Feiner [CHIN89] describe an object-precision algorithm that builds a single shadow volume for a polygonal environment, using the BSP-tree solid modeling representation discussed in Section 12.6.4. Polygons are processed in front-to-back order relative to the light source. Each polygon facing the light source is filtered down the tree, dividing the polygon into lit and shadowed fragments. Only lit fragments cast shadows, so the semi-infinite pyramid defined by the light source and each lit fragment is added to the volume. Because of the front-to-back order, every polygon is guaranteed not to lie between the light source and the polygons processed previously. Therefore, since no polygon needs to be compared with the plane of a previously processed polygon, the polygons themselves do not need to be added to the shadow volume. As with the Atherton–Weiler–Greenberg algorithm, the lit fragments may be added to the environment as surface-detail polygons or the lit and shadowed fragments may be displayed together instead. Multiple light sources are accommodated by filtering the polygon fragments of one shadow-volume BSP tree down the shadow-volume BSP tree of the next light source. Each fragment is tagged to indicate the light sources that illuminate it, allowing the resulting fragmented environment to be displayed with any polygon visible-surface algorithm. Because of the shadow volume representation, lights may be positioned anywhere relative to the objects; thus, sectoring is not necessary. Several optimizations and a parallel version of the algorithm are discussed in [CHIN90]. Color Plate III.6(a) is rendered with the algorithm; Color Plate III.6(b) shows the fragments created in filtering the polygons down the shadow-volume BSP tree.

16.4.4 A Two-Pass z-Buffer Shadow Algorithm

Williams [WILL78] developed a shadow-generation method based on two passes through a z-buffer algorithm, one for the viewer and one for the light source. His algorithm, unlike the two-pass algorithm of Section 16.4.2, determines whether a surface is shadowed by using

Samsung/Google, EX1027, p. 762

image-precision calculations. Figure 16.32(a) shows an overview of an environment lit by a light at L; a shadowless image from viewpoint V is shown in Fig. 16.32(d). The algorithm begins by calculating and storing just the z-buffer for the image from the viewpoint of the light (Fig. 16.32b). In Fig. 16.32(b), increasing intensities represent increasing distance. Next, the z-buffer (Fig. 16.32c) and the image (Fig. 16.32e) are calculated from the viewpoint of the observer using a z-buffer algorithm with the following modification. Whenever a pixel is determined to be visible, its object-precision coordinates in the observer's view (x_o, y_o, z_o) are transformed into coordinates in the light source's view (x'_o, y'_o, z'_o). The transformed coordinates x'_o and y'_o are used to select the value z_L in the light source's z-buffer to be compared with the transformed value z'_o. If z_L is closer to the light than is z'_o, then there is something blocking the light from the point, and the pixel is shaded as being in shadow; otherwise the point is visible from the light and it is shaded as lit. In analogy to texture mapping, we can think of the light's z-buffer as a *shadow map*. Multiple light sources can be accommodated by use of a separate shadow map for each light source.

Like the regular z-buffer visible-surface algorithm, this algorithm requires that each rendered pixel be shaded. Here, this means that shadow calculations must be performed for the pixel, even if it is ultimately painted over by closer objects. Williams has suggested a variation on his algorithm that exploits the ease with which the z-buffer algorithm can interleave visible-surface determination with illumination and shading, and eliminates shadow calculations for obscured objects. Rather than computing just the shadow map first, the modified algorithm also computes the regular shaded image from the observer's point of view (Fig. 16.32d), along with its z-buffer (all these computations can use conventional z-buffer–based hardware). Shadows are then added using a postprocess that is linear in the

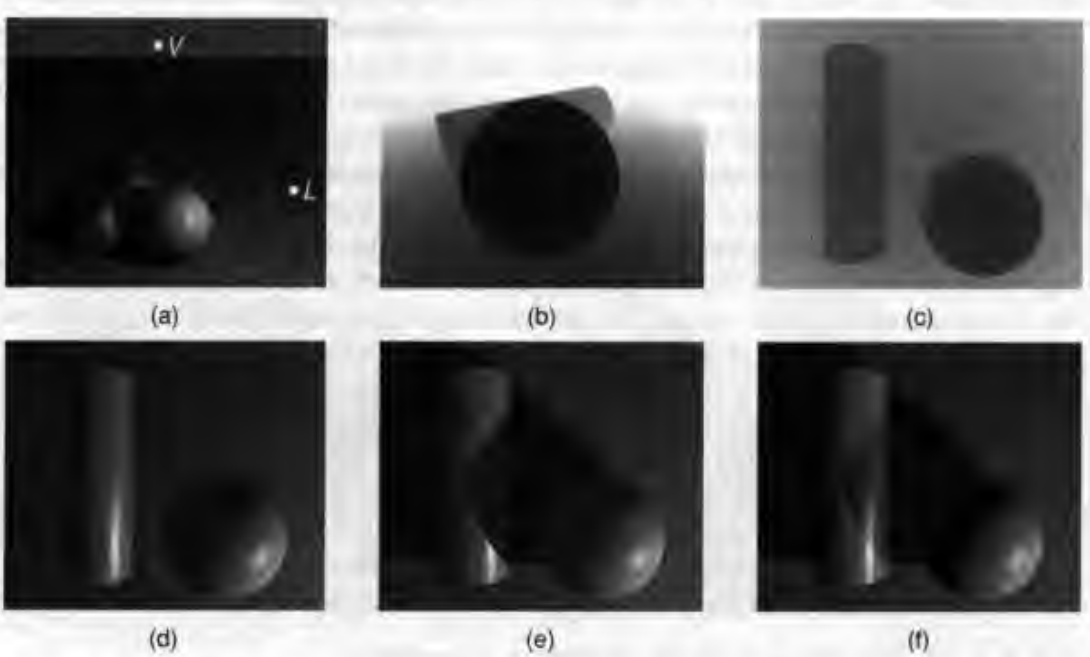


Fig. 16.32 z-buffer shadow-generation method. (a) Overview. (b) Light's z-buffer. (c) Observer's z-buffer. (d) Observer's image. (e) Observer's image with shadows. (f) Observer's image with post-processed shadows. (By David Kurlander, Columbia University.)

Samsung/Google, EX1027, p. 763



Fig. 16.33 Shadow map used to create Color Plate II.36. (Copyright © 1990, Pixar. Shadow map rendered by Thomas Williams and H. B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

number of pixels in the image to produce Fig. 16.32(f). The same transformation and comparison operation as before are performed for each pixel in the observer's image. If z_L is closer to the light than is z'_0, then the pixel's previously computed shade in the observer's image is darkened. Although this approach is significantly more efficient than is the original algorithm, it results in artifacts; most noticeably, shadowed objects will have (darkened) specular highlights, even though there should be no specular highlights on an object that is shielded from the light source. In addition, the z_0 to be transformed is at object precision in the first version, but at the typically lower z-buffer precision here. (See Exercise 16.15.)

Unlike the other shadow algorithms discussed so far, Williams's algorithm makes it especially easy to generate shadows for any object that can be scan-converted, including curved surfaces. Because all operations are performed in image precision, however, allowance must be made for the limited numerical precision. For example, the transformation from z_0 to z'_0 should also move z'_0 a little closer to the light source, to avoid having a visible point cast a shadow on itself. Like the z-buffer visible-surface algorithm from which it is constructed, this shadow algorithm is prone to aliasing. Williams describes how filtering and dithering can reduce the effects of aliasing. Reeves, Salesin, and Cook [REEV87] demonstrate improvements using *percentage closer filtering*. Each z'_0 is compared with values in a region of the shadow map, and the percentage of closer values determines the amount of shadowing. This improved algorithm was used to render Color Plates D, F, and II.36. Figure 16.33 shows the shadow map used to create Color Plate II.36.

16.4.5 Global Illumination Shadow Algorithms

Ray-tracing and radiosity algorithms have been used to generate some of the most impressive pictures of shadows in complex environments. Simple ray tracing has been used to model shadows from point light sources, whereas more advanced versions allow extended light sources. Both are discussed in Section 16.12. Radiosity methods, discussed in Section 16.13, model light sources as light-emitting surfaces that may have the same geometry as any other surface; thus, they implicitly support extended light sources.

Samsung/Google, EX1027, p. 764

16.5 TRANSPARENCY

Much as surfaces can have specular and diffuse reflection, those that transmit light can be transparent or translucent. We can usually see clearly through *transparent* materials, such as glass, although in general the rays are refracted (bent). Diffuse transmission occurs through *translucent* materials, such as frosted glass. Rays passing through translucent materials are jumbled by surface or internal irregularities, and thus objects seen through translucent materials are blurred.

16.5.1 Nonrefractive Transparency

The simplest approach to modeling transparency ignores refraction, so light rays are not bent as they pass through the surface. Thus, whatever is visible on the line of sight through a transparent surface is also geometrically located on that line of sight. Although refractionless transparency is not realistic, it can often be a more useful effect than refraction. For example, it can provide a distortionless view through a surface, as depicted in Color Plate III.7. As we have noted before, total photographic realism is not always the objective in making pictures.

Two different methods have been commonly used to approximate the way in which the colors of two objects are combined when one object is seen through the other. We shall refer to these as *interpolated* and *filtered* transparency.

Interpolated transparency. Consider what happens when transparent polygon 1 is between the viewer and opaque polygon 2, as shown in Fig. 16.34. *Interpolated transparency* determines the shade of a pixel in the intersection of two polygons' projections by linearly interpolating the individual shades calculated for the two polygons:

$$I_\lambda = (1 - k_{t1})I_{\lambda 1} + k_{t1}I_{\lambda 2}. \tag{16.25}$$

The *transmission coefficient* k_{t1} measures the *transparency* of polygon 1, and ranges between 0 and 1. When k_{t1} is 0, the polygon is opaque and transmits no light; when k_{t1} is 1, the polygon is perfectly transparent and contributes nothing to the intensity I_λ; The value $1 - k_{t1}$ is called the polygon's *opacity*. Interpolated transparency may be thought of as modeling a polygon that consists of a fine mesh of opaque material through which other objects may be seen; k_{t1} is the fraction of the mesh's surface that can be seen through. A totally transparent

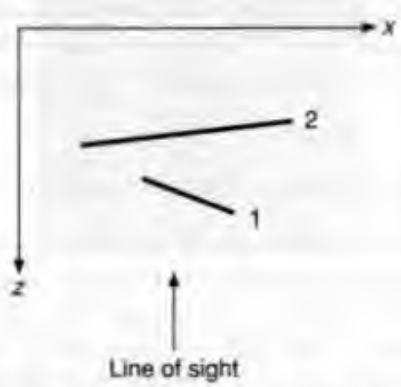


Fig. 16.34 Cross-section of two polygons.

Samsung/Google, EX1027, p. 765

polygon that is processed this way will not have any specular reflection. For a more realistic effect, we can interpolate only the ambient and diffuse components of polygon 1 with the full shade of polygon 2, and then add in polygon 1's specular component [KAY79b].

Another approach, often called *screen-door transparency*, literally implements a mesh by rendering only some of the pixels associated with a transparent object's projection. The low-order bits of a pixel's (x, y) address are used to index into a transparency bit mask. If the indexed bit is 1, then the pixel is written; otherwise, it is suppressed, and the next closest polygon at that pixel is visible. The fewer 1 bits in the mask, the more transparent the object's appearance. This approach relies on having our eyes perform spatial integration to produce interpolated transparency. Note, however, that an object fully obscures any other object drawn with the same transparency mask and that other undesirable interactions between masks are difficult to avoid.

Filtered transparency. *Filtered transparency* treats a polygon as a transparent filter that selectively passes different wavelengths; it can be modeled by

$$I_\lambda = I_{\lambda 1} + k_{t1} O_{t\lambda} I_{\lambda 2}, \tag{16.26}$$

where $O_{t\lambda}$ is polygon 1's *transparency color*. A colored filter may be modeled by choosing a different value of $O_{t\lambda}$ for each λ (but see Section 16.9). In either interpolated or filtered transparency, if additional transparent polygons are in front of these polygons, then the calculation is invoked recursively for polygons in back-to-front order, each time using the previously computed I_λ as $I_{\lambda 2}$.

Implementing transparency. Several visible-surface algorithms can be readily adapted to incorporate transparency, including scan-line and list-priority algorithms. In list-priority algorithms, the color of a pixel about to be covered by a transparent polygon is read back and used in the illumination model while the polygon is being scan-converted.

Most z-buffer–based systems support screen-door transparency because it allows transparent objects to be intermingled with opaque objects and to be drawn in any order. Adding transparency effects that use Eqs. (16.25) or (16.26) to the z-buffer algorithm is more difficult, because polygons are rendered in the order in which they are encountered. Imagine rendering several overlapping transparent polygons, followed by an opaque one. We would like to slip the opaque polygon behind the appropriate transparent ones. Unfortunately, the z-buffer does not store the information needed to determine which transparent polygons are in front of the opaque polygon, or even the polygons' relative order. One simple, although incorrect, approach is to render transparent polygons last, combining their colors with those already in the frame buffer, but not modifying the z-buffer; when two transparent polygons overlap, however, their relative depth is not taken into account.

Mammen [MAMM89] describes how to render transparent objects properly in back-to-front order in a z-buffer-based system through the use of multiple rendering passes and additional memory. First, all the opaque objects are rendered using a conventional z-buffer. Then, transparent objects are processed into a separate set of buffers that contain, for each pixel, a transparency value and a flag bit, in addition to the pixel's color and z value. Each flag bit is initialized to off and each z value is set to the closest possible value. If a transparent object's z value at a pixel is closer than the value in the opaque z-buffer, but is

Samsung/Google, EX1027, p. 766

more distant than that in the transparent z-buffer, then the color, z value, and transparency are saved in the transparent buffers, and the flag bit is set. After all objects have been processed, the transparent object buffers contain information for the most distant transparent object at each pixel whose flag bit is set. Information for flagged pixels is then blended with that in the original frame buffer and z-buffer. A flagged pixel's transparency z-value replaces that in the opaque z-buffer and the flag bit is reset. This process is repeated to render successively closer objects at each pixel. Color Plate III.7 was made using this algorithm.

Kay and Greenberg [KAY79b] have implemented a useful approximation to the increased attenuation that occurs near the silhouette edges of thin curved surfaces, where light passes through more material. They define k_t in terms of a nonlinear function of the z component of the surface normal after perspective transformation,

$$k_t = k_{t\min} + (k_{t\max} - k_{t\min})(1 - (1 - z_N)^m), \tag{16.27}$$

where $k_{t\min}$ and $k_{t\max}$ are the object's minimum and maximum transparencies, z_N is the z component of the normalized surface normal at the point for which k_t is being computed, and m is a power factor (typically 2 or 3). A higher m models a thinner surface. This new value of k_t may be used as k_{ti} in either Eq. (16.25) or (16.26).

16.5.2 Refractive Transparency

Refractive transparency is significantly more difficult to model than is nonrefractive transparency, because the geometrical and optical lines of sight are different. If refraction is considered in Fig. 16.35, object A is visible through the transparent object along the line of sight shown; if refraction is ignored, object B is visible. The relationship between the angle of incidence θ_i and the angle of refraction θ_t is given by Snell's law

$$\frac{\sin \theta_i}{\sin \theta_t} = \frac{\eta_{t\lambda}}{\eta_{i\lambda}}, \tag{16.28}$$

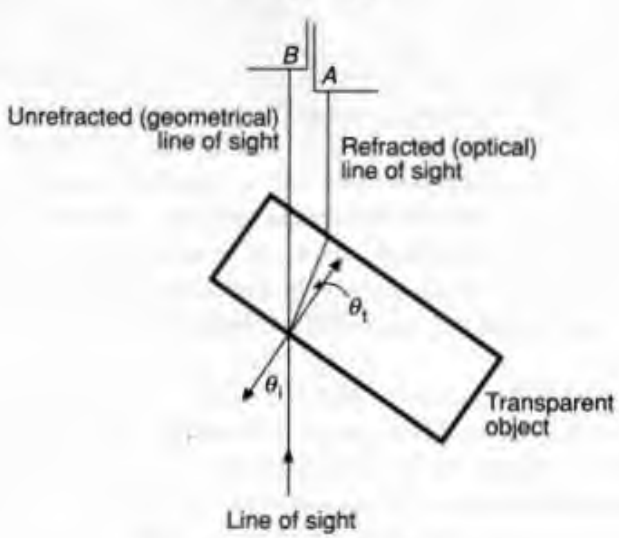


Fig. 16.35 Refraction.

Samsung/Google, EX1027, p. 767

where $\eta_{i\lambda}$ and $\eta_{t\lambda}$ are the *indices of refraction* of the materials through which the light passes. A material's index of refraction is the ratio of the speed of light in a vacuum to the speed of light in the material. It varies with the wavelength of the light and even with temperature. A vacuum has an index of refraction of 1.0, as does the atmosphere to close approximation; all materials have higher values. The index of refraction's wavelength-dependence is evident in many instances of refraction as *dispersion*—the familiar, but difficult to model, phenomenon of refracted light being spread into its spectrum [THOM86; MUSG89].

Calculating the refraction vector. The unit vector in the direction of refraction, $\overline{T}$, can be calculated as

$$\overline{T} = \sin \theta_t \, \overline{M} - \cos \theta_t \, \overline{N}, \tag{16.29}$$

where $\overline{M}$ is a unit vector perpendicular to $\overline{N}$ in the plane of the incident ray $\overline{I}$ and $\overline{N}$ [HECK84] (Fig. 16.36). Recalling the use of $\overline{S}$ in calculating the reflection vector $\overline{R}$ in Section 16.1.4, we see that $\overline{M} = (\overline{N} \cos \theta_i - \overline{I}) / \sin \theta_i$. By substitution,

$$\overline{T} = \frac{\sin \theta_t}{\sin \theta_i}(\overline{N} \cos \theta_i - \overline{I}) - \cos \theta_t \, \overline{N}. \tag{16.30}$$

If we let $\eta_{r\lambda} = \eta_{i\lambda} / \eta_{t\lambda} = \sin \theta_t / \sin \theta_i$, then after rearranging terms

$$\overline{T} = (\eta_{r\lambda} \cos \theta_i - \cos \theta_t) \, \overline{N} - \eta_{r\lambda} \, \overline{I}. \tag{16.31}$$

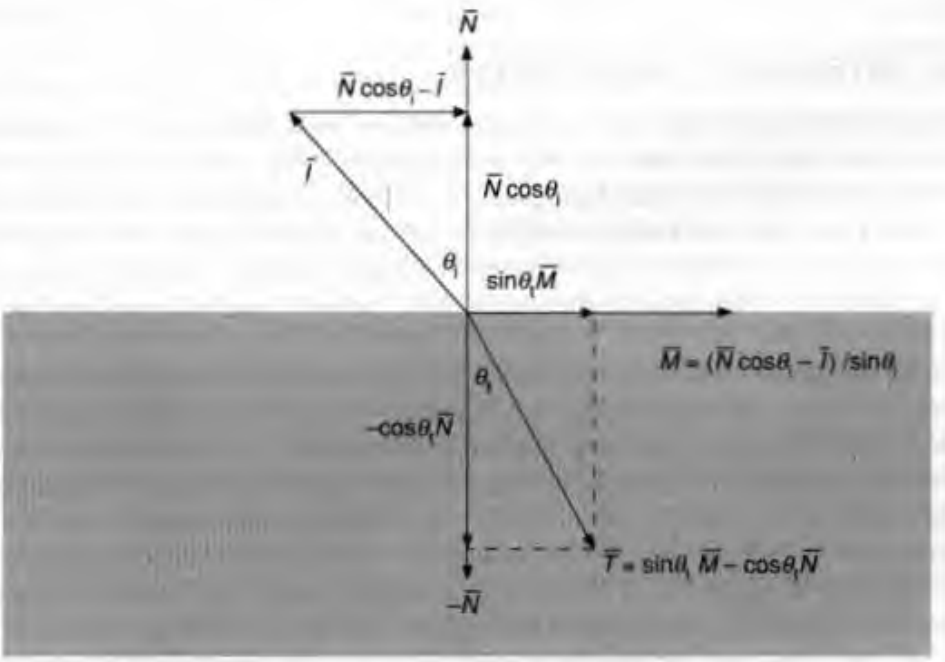


Fig. 16.36 Calculating the refraction vector.

Samsung/Google, EX1027, p. 768

Note that $\cos\theta_i$ is $\bar{N} \cdot \bar{I}$, and $\cos\theta_t$ can be computed as

$$\cos\theta_t = \sqrt{1 - \sin^2\theta_t} = \sqrt{1 - \eta_{r\lambda}^2 \sin^2\theta_i} = \sqrt{1 - \eta_{r\lambda}^2(1 - (\bar{N} \cdot \bar{I})^2)}. \quad (16.32)$$

Thus,

$$\bar{T} = \left(\eta_{r\lambda}(\bar{N} \cdot \bar{I}) - \sqrt{1 - \eta_{r\lambda}^2(1 - (\bar{N} \cdot \bar{I})^2)}\right)\bar{N} - \eta_{r\lambda}\bar{I}. \quad (16.33)$$

Total internal reflection. When light passes from one medium into another whose index of refraction is lower, the angle θ_t of the transmitted ray is greater than the angle θ_i. If θ_i becomes sufficiently large, then θ_t exceeds 90° and the ray is reflected from the interface between the media, rather than being transmitted. This phenomenon is known as *total internal reflection*, and the smallest θ_i at which it occurs is called the *critical angle*. You can observe total internal reflection easily by looking through the front of a filled fish tank and trying to see your hand through a side wall. When the viewing angle is greater than the critical angle, the only visible parts of your hand are those pressed firmly against the tank, with no intervening layer of air (which has a lower index of refraction than glass or water). The critical angle is the value of θ_i at which $\sin\theta_t$ is 1. If $\sin\theta_t$ is set to 1 in Eq. (16.28), we can see that the critical angle is $\sin^{-1}(\eta_{t\lambda} / \eta_{i\lambda})$. Total internal reflection occurs when the square root in Eq. (16.33) is imaginary.

Section 16.12 discusses the use of Snell's law in modeling refractive transparency with ray tracing; translucency is treated in Sections 16.12.4 and 16.13. An approximation of refraction can also be incorporated into renderers that proceed in back-to-front order [KAY79b].

16.6 INTEROBJECT REFLECTIONS

Interobject reflections occur when a surface reflects other surfaces in its environment. These effects range from more or less sharp specular reflections that change with the viewer's position (like specular highlights), to diffuse reflections that are insensitive to the viewer's position. Ray tracing (Section 16.12) and radiosity methods (Section 16.13) have produced some of the most visually impressive pictures exhibiting specular and diffuse interobject reflections; earlier techniques, however, can also produce attractive results.

Blinn and Newell [BLIN76] developed *reflection mapping* (also known as *environment mapping*) to model specular interobject reflection. A center of projection is chosen from which to map the environment to be reflected onto the surface of a sphere surrounding the objects to be rendered. The mapped environment can then be treated as a 2D texture map. At each point on an object to be displayed, the reflection map is indexed by the polar coordinates of the vector obtained by reflecting $\bar{V}$ about $\bar{N}$. The reflection map's x and y axes represent longitude (from 0° to 360°) and latitude (from −90° to 90°), respectively, as shown in Fig. 16.37(a). Hall [HALL86] suggests a variant in which the y axis is sin (latitude), so that equal areas on the sphere map to equal areas on the reflection map (Fig.

Samsung/Google, EX1027, p. 769

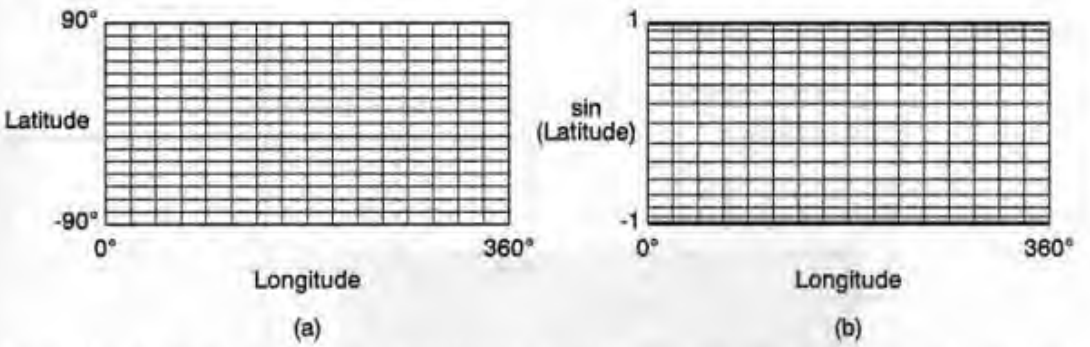


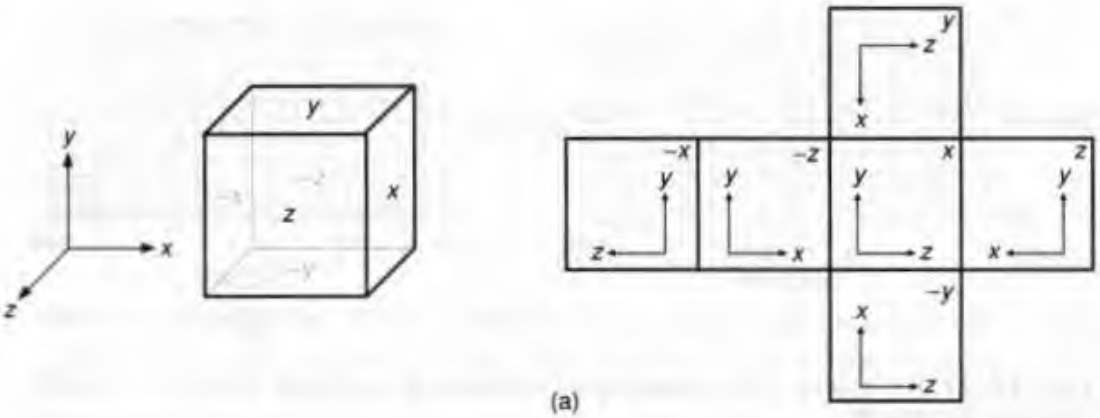
Fig. 16.37 Reflection map coordinate systems. (a) Latitude–longitude. (b) Sin (latitude)–longitude.

16.37b). Alternatively, six projections onto the sides of a surrounding cube may be used. The cube is aligned with the WC axes, so that the largest coordinate of the normalized reflection vector indicates the appropriate side to index. Figure 16.38(a) shows the correspondence between the unfolded sides and the cube; Fig. 16.38(b) is the reflection map used for the teapot in Color Plate II.37. As in texture mapping, antialiasing is accomplished by filtering some number of reflection-map values surrounding the indexed value to determine the reflected light at the given point on the object. In Fig. 16.38(b) an angle slightly wider than 90° was used to provide a margin about each side's borders that helps avoid the need to consider more than one side at a time when filtering.

Although reflection mapping can be used to produce a number of useful effects [GREE86], it provides only an approximation to the correct reflection information. By taking into account just the surface's reflection direction and not its position in the sphere, it models an infinitely large environment sphere. This problem can be partly remedied by using the surface's position to help determine the part of the reflection map to index, modeling a sphere of finite size. In either case, however, the farther a surface is from the center of projection used to create the map, the more distorted a view of the world it shows, since the reflection map takes into account visibility relationships at only a single point. A useful compromise is to create multiple reflection maps, each centered about a key object, and to index into the one closest to an object whose surface is being mapped. Simple but effective reflection effects can be obtained with even a 1D reflection map. For example, the y component of the reflection of $\overline{V}$ may be used to index into an array of intensities representing the range of colors from ground through horizon to sky.

Planar surfaces present difficulties for reflection mapping, because the reflection angle changes so slowly. If reflections from a planar surface are to be viewed from only a single viewpoint, however, another technique can be used. The viewpoint is reflected about the surface's plane and an inverted image of the scene is rendered from the reflected viewpoint, as shown in cross-section in Fig. 16.38(c). This image can then be merged with the original image wherever the surface is visible. Figure 16.38(d) was created with this technique and was used to render the reflections on the floor in Color Plate II.37.

Samsung/Google, EX1027, p. 770



(a)



(b)

16.7 PHYSICALLY BASED ILLUMINATION MODELS

The illumination models discussed in the previous sections are largely the result of a common-sense, practical approach to graphics. Although the equations used approximate some of the ways light interacts with objects, they do not have a physical basis. In this section, we discuss physically based illumination models, relying in part on the work of Cook and Torrance [COOK82].

Thus far, we have used the word *intensity* without defining it, referring informally to the intensity of a light source, of a point on a surface, or of a pixel. It is time now to formalize our terms by introducing the radiometric terminology used in the study of thermal radiation, which is the basis for our understanding of how light interacts with objects [NICO77; SPARR78; SIEG81; IES87]. We begin with *flux*, which is the rate at which light energy is emitted and is measured in watts (W). To refer to the amount of flux emitted in or received from a given direction, we need the concept of a *solid angle*, which is the angle at the apex of a cone. Solid angle is measured in terms of the area on a sphere intercepted by a cone whose apex is at the sphere's center. A *steradian* (sr) is the solid angle of such a cone

Samsung/Google, EX1027, p. 771



(c)



(d)

Fig. 16.38 Reflection maps. (a) Cube reflection map layout. (b) Reflection map for teapot in Color Plate II.37. (c) Geometry of planar reflection. (d) Reflected image merged with floor in Color Plate II.37. (Copyright © 1990, Pixar. Images in (b) and (d) rendered by Thomas Williams and H. B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

that intercepts an area equal to the square of the sphere's radius r. If a point is on a surface, we are concerned with the hemisphere above it. Since the area of a sphere is $4\pi r^2$, there are $4\pi r^2 / 2r^2 = 2\pi$ sr in a hemisphere. Imagine projecting an object's shape onto a hemisphere centered about a point on the surface that serves as the center of projection. The solid angle ω subtended by the object is the area on the hemisphere occupied by the projection, divided by the square of the hemisphere's radius (the division eliminates dependence on the size of the hemisphere). Thus, for convenience, we often speak of solid angle in terms of the area projected on a unit sphere or hemisphere, as shown in Fig. 16.39.

Radiant intensity is the flux radiated into a unit solid angle in a particular direction and is measured in W / sr. When we used the word *intensity* in reference to a point source, we were referring to its radiant intensity.

Radiance is the radiant intensity per unit foreshortened surface area, and is measured in W / (sr · m²). *Foreshortened surface area,* also known as *projected surface area,* refers to the projection of the surface onto the plane perpendicular to the direction of radiation. The foreshortened surface area is found by multiplying the surface area by $\cos \theta_r$, where θ_r is the

Samsung/Google, EX1027, p. 772

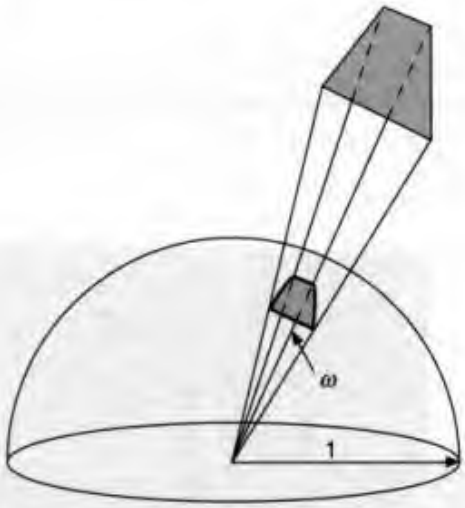


Fig. 16.39 The solid angle subtended by an object from a point on a surface is the area covered by the object's projection onto a unit hemisphere above the point.

angle of the radiated light relative to the surface normal. A small solid angle $d\omega$ may be approximated as the object's foreshortened surface area divided by the square of the distance from the object to the point at which the solid angle is being computed. When we used the word *intensity* in reference to a surface, we were referring to its radiance. Finally, *irradiance*, also known as *flux density*, is the incident flux per (unforeshortened) unit surface area and is measured in W / m^2.

In graphics, we are interested in the relationship between the light incident on a surface and the light reflected from and transmitted through that surface. Consider Fig. 16.40. The irradiance of the incident light is

$$E_i = I_i(\overline{N} \cdot \overline{L})\, d\omega_i, \tag{16.34}$$

where I_i is the incident light's radiance, and $\overline{N} \cdot \overline{L}$ is cos θ_i. Since irradiance is expressed per

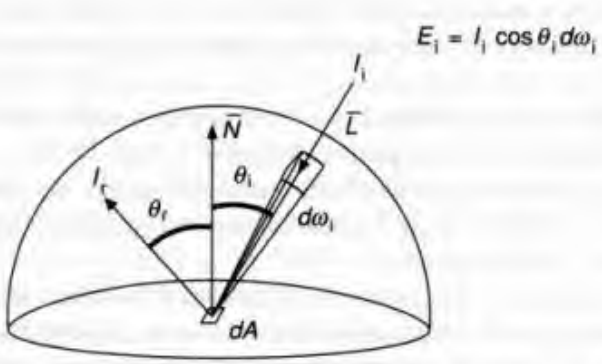


Fig. 16.40 Reflected radiance and incident irradiance.

Samsung/Google, EX1027, p. 773

unit area, whereas radiance is expressed per unit foreshortened area, multiplying by $\overline{N} \cdot \overline{L}$ converts it to the equivalent per unit unforeshortened area.

It is not enough to consider just I_i (the incident radiance) when determining I_r (the reflected radiance); E_i (the incident irradiance) must instead be taken into account. For example, an incident beam that has the same radiant intensity (W / sr) as another beam but a greater solid angle has proportionally greater E_i and causes the surface to appear proportionally brighter. The ratio of the reflected radiance (intensity) in one direction to the incident irradiance (flux density) responsible for it from another direction is known as the *bidirectional reflectivity*, ρ, which is a function of the directions of incidence and reflection,

$$\rho = \frac{I_r}{E_i}. \tag{16.35}$$

Thus, substituting for E_i from Eq. (16.34), we get

$$I_r = \rho I_i(\overline{N} \cdot \overline{L})d\omega_i. \tag{16.36}$$

The irradiance incident on a pixel in the image (the *image irradiance*) is proportional to the radiance emitted by the scene (the *scene radiance*) that is focused on the pixel [HORN79]. The factor of proportionality is a function of the imaging system being used.

As we have seen, it is conventional in computer graphics to consider bidirectional reflectivity as composed of diffuse and specular components. Therefore,

$$\rho = k_d\rho_d + k_s\rho_s, \tag{16.37}$$

where ρ_d and ρ_s are respectively the diffuse and specular bidirectional reflectivities, and k_d and k_s are respectively the diffuse and specular reflection coefficients introduced earlier in this chapter; $k_d + k_s = 1$. It is important to note that Eq. (16.37) is a useful approximation that is not applicable to all surfaces. For example, the lunar surface has a ρ that peaks in the direction of incidence [SIEG81]. During a full moon, when the sun, earth, and moon are nearly in line, this accounts for why the moon appears as a disk of roughly uniform intensity. If the moon had a Lambertian surface, it would, instead, reflect more light at its center than at its sides.

In addition to the effects of direct light-source illumination, we need to take into account illumination by light reflected from other surfaces. The lighting models discussed so far have assumed that this ambient light is equally incident from all directions, is independent of viewer position, and is not blocked by any nearby objects. Later in this chapter we shall discuss how to lift these restrictions. For now we retain them, modeling the ambient term as $\rho_a I_a$, where ρ_a is the nondirectional ambient reflectivity, and I_a is the incident ambient intensity.

The resulting illumination equation for n light sources is

$$I_r = \rho_a I_a + \sum_{1 \le j \le n} I_{ij}(\overline{N} \cdot \overline{L}_j)\, d\omega_{ij}(k_d\rho_d + k_s\rho_s). \tag{16.38}$$

To reinterpret the illumination model of Eq. (16.15) in terms of Eq. (16.38), we expressed diffuse reflectivity as the object's diffuse color, and specular reflectivity using the product of the object's specular color and a $\cos^n \alpha$ term. We have already noted some of the

Samsung/Google, EX1027, p. 774

inadequacies of that specular reflectivity formulation. Now we shall examine how to replace it.

16.7.1 Improving the Surface Model

The Torrance–Sparrow model [TORR66; TORR67], developed by applied physicists, is a physically based model of a reflecting surface. Blinn was the first to adapt the Torrance–Sparrow model to computer graphics, giving the mathematical details and comparing it to the Phong model in [BLIN77a]; Cook and Torrance [COOK82] were the first to approximate the spectral composition of reflected light in an implementation of the model.

In the Torrance–Sparrow model, the surface is assumed to be an isotropic collection of planar microscopic facets, each a perfectly smooth reflector. The geometry and distribution of these *microfacets* and the direction of the light (assumed to emanate from an infinitely distant source, so that all rays are parallel) determine the intensity and direction of specular reflection as a function of I_p (the point light source intensity), $\overline{N}$, $\overline{L}$, and $\overline{V}$. Experimental measurements show a very good correspondence between the actual reflection and the reflection predicted by this model [TORR67].

For the specular component of the bidirectional reflectivity, Cook and Torrance use

$$\rho_s = \frac{F_\lambda}{\pi} \frac{DG}{(\overline{N} \cdot \overline{V})(\overline{N} \cdot \overline{L})}, \tag{16.39}$$

where D is a distribution function of the microfacet orientations, G is the *geometrical attenuation factor*, which represents the masking and shadowing effects of the microfacets on each other, and F_λ is the Fresnel term computed by Fresnel's equation (described later), which, for specular reflection, relates incident light to reflected light for the smooth surface of each microfacet. The π in the denominator is intended to account for surface roughness (but see [JOY88, pp. 227–230] for an overview of how the equation is derived). The $\overline{N} \cdot \overline{V}$ term makes the equation proportional to the surface area (and hence to the number of microfacets) that the viewer sees in a unit piece of foreshortened surface area, whereas the $\overline{N} \cdot \overline{L}$ term makes the equation proportional to the surface area that the light sees in a unit piece of foreshortened surface area.

16.7.2 The Microfacet Distribution Function

Since the microfacets are considered to be perfect specular reflectors, the model considers only those microfacets whose normals lie along the halfway vector $\overline{H}$, introduced in Section 16.1.4. Only a fraction D of the microfacets have this orientation. Torrance and Sparrow assumed a Gaussian distribution function for D in their original work. Blinn used the Trowbridge and Reitz distribution [TROW75], and Cook and Torrance used the Beckmann distribution function [BECK63]. Cook and Torrance point out the Beckmann distribution has a good theoretical basis and has no arbitrary constants, unlike the distributions used by Torrance and Sparrow, and by Blinn. The Beckmann distribution function for rough surfaces is

$$D = \frac{1}{4m^2 \cos^4\beta} e^{-[(\tan\beta)/m]^2}, \tag{16.40}$$

Samsung/Google, EX1027, p. 775

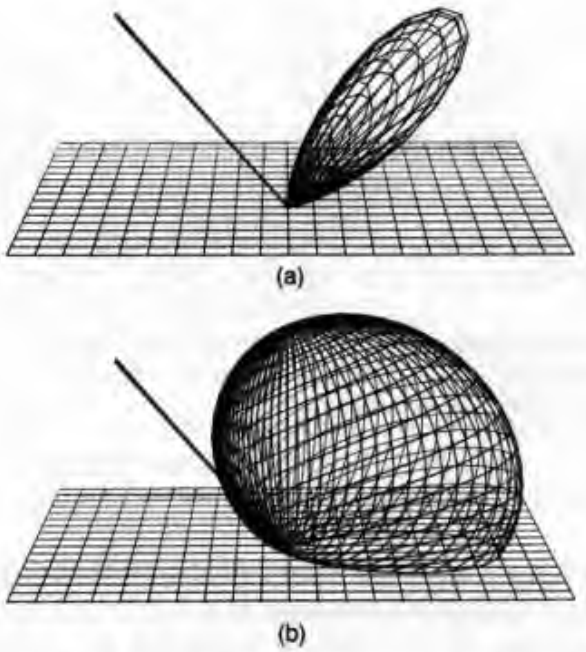


Fig. 16.41 Beckmann microfacet distribution function for (a) $m = 0.2$ and (b) $m = 0.6$. (From [COOK82] with permission. By Robert Cook, Program of Computer Graphics, Cornell University.)

where β is the angle between $\overline{N}$ and $\overline{H}$, and m is the root-mean-square slope of the microfacets.[1] When m is small, the microfacet slopes vary only slightly from the surface normal and, as we would expect, the reflection is highly directional (Fig. 16.41a). When m is large, the microfacet slopes are steep and the resulting rough surface spreads out the light it reflects (Fig. 16.41b). To model surfaces that have multiple scales of roughness, Cook and Torrance use a weighted sum of distribution functions,

$$D = \sum_{1 \le j \le n} w_j D(m_j), \tag{16.41}$$

where the sum of the weights w_j is 1.

16.7.3 The Geometrical Attenuation Factor

The model takes into account that some microfacets may shadow others. Torrance and Sparrow and Blinn discuss the calculation of G, considering three different situations. Figure 16.42(a) shows a microfacet whose incident light is totally reflected. Figure 16.42(b) shows a microfacet that is fully exposed to the rays, but has some reflected rays that are shielded by other microfacets. These shielded rays ultimately contribute to the diffuse reflection. Figure 16.42(c) shows a microfacet that is partially shielded from the light. The geometric attenuation factor G ranges from 0 (total shadowing) to 1 (no shadowing).

[1] Hall [HALL89] mentions that [COOK82] is missing the 4 in the denominator.

Samsung/Google, EX1027, p. 776

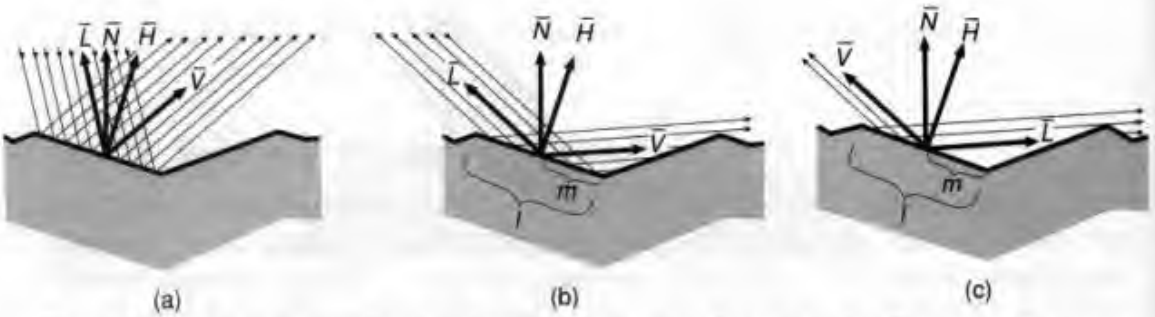


Fig. 16.42 Light rays reflecting from surface microfacets in the Torrance-Sparrow model. (a) No interference. (b) Partial interception of reflected light. (c) Partial interception of incident light. (After [BLIN77a].)

To simplify the analysis, the microfacets are assumed to form V-shaped grooves that are symmetric about the average surface normal $\overline{N}$. In Fig. 16.42(a), all the incident light is reflected to the viewer; thus, G_a is 1. In both other cases, the ratio of intercepted light is given by m / l, where l is the total area of the facet and m is the area whose reflected light is blocked (Fig. 16.42b) or that is itself blocked (Fig 16.42c) from the light. Therefore, $G_b = G_c = 1 - m / l$. Blinn gives a trigonometric derivation of the proportion of reflected light in Fig. 16.42(b) as

$$G_b = \frac{2(\overline{N} \cdot \overline{H})(\overline{N} \cdot \overline{V})}{(\overline{V} \cdot \overline{H})}. \tag{16.42}$$

The ratio for the case in Fig. 16.42(c) follows by noticing that it is the same as the case in part (b), except that $\overline{L}$ and $\overline{V}$ trade places:

$$G_c = \frac{2(\overline{N} \cdot \overline{H})(\overline{N} \cdot \overline{L})}{(\overline{V} \cdot \overline{H})}. \tag{16.43}$$

The denominator does not need to change because $(\overline{V} \cdot \overline{H}) = (\overline{L} \cdot \overline{H})$ by definition of $\overline{H}$ as the halfway vector between $\overline{V}$ and $\overline{L}$.

G is the minimum of the three values:

$$G = \min\left\{1, \frac{2(\overline{N} \cdot \overline{H})(\overline{N} \cdot \overline{V})}{(\overline{V} \cdot \overline{H})}, \frac{2(\overline{N} \cdot \overline{H})(\overline{N} \cdot \overline{L})}{(\overline{V} \cdot \overline{H})}\right\}. \tag{16.44}$$

16.7.4 The Fresnel Term

The Fresnel equation for unpolarized light specifies the ratio of reflected light from a dielectric (nonconducting) surface as

$$F_\lambda = \frac{1}{2}\left(\frac{\tan^2(\theta_i - \theta_t)}{\tan^2(\theta_i + \theta_t)} + \frac{\sin^2(\theta_i - \theta_t)}{\sin^2(\theta_i + \theta_t)}\right) = \frac{1}{2}\frac{\sin^2(\theta_i - \theta_t)}{\sin^2(\theta_i + \theta_t)}\left(1 + \frac{\cos^2(\theta_i + \theta_t)}{\cos^2(\theta_i - \theta_t)}\right), \tag{16.45}$$

Samsung/Google, EX1027, p. 777

where θ_i is the angle of incidence relative to $\overline{H}$ (i.e., $\cos^{-1}(\overline{L} \cdot \overline{H})$), and, as before, θ_t is the angle of refraction; $\sin\theta_t = (\eta_{i\lambda} / \eta_{t\lambda})\sin\theta_i$, where $\eta_{i\lambda}$ and $\eta_{t\lambda}$ are the indices of refraction of the two media. The Fresnel equation can also be expressed as

$$F_\lambda = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left(1 + \frac{[c(g+c)-1]^2}{[c(g-c)+1]^2}\right), \tag{16.46}$$

where $c = \cos\theta_i = \overline{L} \cdot \overline{H}$, $g^2 = \eta_\lambda^2 + c^2 - 1$, and $\eta_\lambda = \eta_{i\lambda} / \eta_{t\lambda}$.

In a conducting medium, which attenuates light, it is necessary to refer to $\overline{\eta}_\lambda$, the material's *complex index of refraction*, defined as

$$\overline{\eta}_\lambda = \eta_\lambda - i\kappa_\lambda, \tag{16.47}$$

where κ_λ is the material's *coefficient of extinction*, which measures the amount that the material attenuates intensity per unit path length. To simplify the reflection computations for conductors, κ_λ can be assumed to be zero and a single equivalent real value for η_λ can be determined.

Blinn created Figs. 16.43 and 16.44, comparing the effects of the Phong illumination model and the Torrance–Sparrow model. He made the simplifying assumptions that the specular term depends on only the color of the incident light, and that the viewer and light source are both at infinity. Figures 16.43 and 16.44 show the reflected illumination from a surface for angles of incidence of 30° and 70°, respectively. In each figure, the vertical arrow represents the surface normal, the incoming arrow represents the direction of light rays, and the outgoing arrow represents the direction of reflection for a perfect reflector. The rounded part of each figure is the diffuse reflection, whereas the bump is the specular reflection. For the 30° case in Fig. 16.43, the models produce nearly similar results, but for the 70° case in Fig. 16.44, the Torrance–Sparrow model has much higher specular reflectance and the peak occurs at an angle greater than the angle of incidence. This so-called *off-specular peak* is observed in actual environments. Figure 16.45, also by Blinn, shows a marked difference in the visual effect of the two models as the light source moves away from the viewpoint to the side and then to the rear of a metallic sphere.



(a) Phong model (b) Torrance-Sparrow model

Fig. 16.43 Comparison of Phong and Torrance–Sparrow illumination models for light at a 30° angle of incidence. (By J. Blinn [BLIN77a], courtesy of the University of Utah.)

Samsung/Google, EX1027, p. 778



(a) Phong model (b) Torrance-Sparrow model

Fig. 16.44 Comparison of Phong and Torrance–Sparrow illumination models for light at a 70° angle of incidence. (By J. Blinn [BLIN77a], courtesy of the University of Utah.)

The specular-reflection color shift. Like Blinn, Cook and Torrance use the Torrance–Sparrow surface model to determine the specular term. Unlike Blinn, however, they make the color of the specular reflection a function of the interaction between the material and the incident light, depending both on the light's wavelength and on its angle of incidence. This is correct because the Fresnel equation, Eq. (16.45), is responsible for a shift in the specular reflection color based on the angle the incident light makes with the microfacet normal $\overline{H}$, as shown in Fig. 16.46.

When the incident light is in the same direction as $\overline{H}$, then $\theta_i = 0$, so $c = 1$ and $g = \eta_\lambda$. Substituting these values in Eq. (16.46) yields the Fresnel term for $\theta_i = 0$,

$$F_{\lambda 0} = \left(\frac{\eta_\lambda - 1}{\eta_\lambda + 1}\right)^2. \tag{16.48}$$



(a) Phong model (b) Torrance-Sparrow model

Fig. 16.45 Comparison of Phong and Torrance–Sparrow illumination models for a metallic sphere illuminated by a light source from different directions. Differences are most apparent for back-lit cases (bottom rows). (By J. Blinn [BLIN77a], courtesy of the University of Utah.)

Samsung/Google, EX1027, p. 779



Fig. 16.46 Fresnel term for a copper mirror as a function of wavelength and angle of incidence. (By Robert Cook, Program of Computer Graphics, Cornell University.)

When the incident light grazes the surface of the microfacet, then $\theta_i = \pi / 2$, so $c = 0$. Substituting in Eq. (16.46) yields the Fresnel term for $\theta_i = \pi / 2$,

$$F_{\lambda\pi/2} = 1. \tag{16.49}$$

Thus, if light is normal to the surface, then $F_{\lambda 0}$, and hence the specular reflectance ρ_s, are functions of the surface's index of refraction, which in turn varies with the wavelength. When the light grazes the surface (and when the viewer is looking 180° opposite from the light, since only microfacets with normals of $\overline{H}$ are considered), $F_{\lambda\pi/2}$, and hence the specular reflection, are both 1. Specular reflectance depends on η_λ for all θ_i except $\pi / 2$. For metals, essentially all reflection occurs at the surface and is specular. Only at extreme glancing angles is the specular reflection not influenced by the object's material. Notice how this differs from the Phong specular-reflection term, which was always unaffected by the object color. Cook and Torrance point out that the monochromatic Phong specular term is a good model of plastic, which is colored by pigment particles embedded in a transparent substrate. The presence of these particles causes the diffuse reflection, resulting from light bouncing around in the substrate, to be a function of the pigment and light colors. Specular reflection occurs from the transparent surface, however, and therefore is unaffected by the pigment color. This is why objects rendered with a Phong illumination model look plastic.

If the indices of refraction at different wavelengths are known, they may be used directly in the Fresnel equation. More typically, however, they are not known. Reflectance, however, has been measured for many materials with $\theta_i = 0$ at a variety of wavelengths, as recorded in sources such as [TOUL70; TOUL72a; TOUL72b]. In this case, each η_λ may be determined from Eq. (16.48), as

$$\eta_\lambda = \frac{1 + \sqrt{F_{\lambda 0}}}{1 - \sqrt{F_{\lambda 0}}}. \tag{16.50}$$

A derived value of η_λ may then be used in the Fresnel equation to determine F_λ for an arbitrary θ_i. Rather than perform this calculation for each value of λ, Cook and Torrance simplify the computationally expensive color shift calculation by calculating $F_{\text{avg}\theta_i}$ for η_{avg},

Samsung/Google, EX1027, p. 780

the average index of refraction for average normal reflectance. They use the value computed for $F_{\text{avg}\theta_i}$ to interpolate between the color of the material at $\theta_i = 90°$ and the color of the material at $\theta_i = 0°$ for each component of the color model. Because $F_{\lambda\pi/2}$ is always 1, the color of the material at $\theta_i = 90°$ is the color of the light source. Thus, when the light grazes the surface at a 90° angle of incidence, the color of the reflected light is that of the incident light. Using the RGB system, we call the red component of the material at $\theta_i = 0°$, Red_0, and the red component of the light, $\text{Red}_{\pi/2}$. Red_0 is calculated by integrating the product of F_0, the spectrum of the incident light, and the color-matching curves of Fig. 13.22, and applying the inverse of matrix M of Eq. (13.24) to the result. $\text{Red}_{\pi/2}$ is obtained by applying the inverse of M to Eq. (13.18). The approximation computes the color of the material at θ_i as

$$\text{Red}_{\theta_i} = \text{Red}_0 + (\text{Red}_{\pi/2} - \text{Red}_0)\frac{\max(0, F_{\text{avg}\theta_i} - F_{\text{avg}0})}{F_{\text{avg}\pi/2} - F_{\text{avg}0}}. \tag{16.51}$$

Red_{θ_i} is then used in place of F_λ in Eq. (16.39). Because the approximation takes the light's spectrum into account, Eq. (16.38) must be modified to multiply the specular term by a wavelength-independent intensity scale factor, instead of by the light's spectrum. Hall [HALL89] suggests an alternative approximation that interpolates a value for $F_{\lambda\theta_i}$, given $F_{\lambda 0}$. Since $F_{\lambda\pi/2}$ is always 1 (as is $F_{\text{avg}\pi/2}$ in Eq. 16.51),

$$F_{\lambda\theta_i} = F_{\lambda 0} + (1 - F_{\lambda 0})\frac{\max(0, F_{\text{avg}\theta_i} - F_{\text{avg}0})}{1 - F_{\text{avg}0}}. \tag{16.52}$$

Color Plate III.8 shows two copper vases rendered with the Cook–Torrance model, both of which use the bidirectional reflectance of copper for the diffuse term. The first models the specular term using the reflectance of a vinyl mirror and represents results similar to those obtained with the original Phong illumination model of Eq. (16.14). The second models the specular term with the reflectance of a copper mirror. Note how accounting for the dependence of the specular highlight color on both angle of incidence and surface material produces a more convincing image of a metallic surface.

In general, the ambient, diffuse, and specular components are the color of the material for both dielectrics and conductors. Composite objects, such as plastics, typically have diffuse and specular components that are different colors. Metals typically show little diffuse reflection and have a specular component color that ranges between that of the metal and that of the light source as θ_i approaches 90°. This observation suggests a rough approximation to the Cook–Torrance model that uses Eq. (16.15) with $O_{s\lambda}$ chosen by interpolating from a look-up table based on θ_i.

Further work. Kajiya [KAJI85] has generalized the Cook–Torrance illumination model to derive *anisotropic illumination models* whose reflective properties are not symmetric about the surface normal. These more accurately model the way that light is reflected by hair or burnished metal—surfaces whose microfeatures are preferentially oriented. To do this, Kajiya extends bump mapping to perturb not just the surface normal, but also the

Samsung/Google, EX1027, p. 781

tangent and a binormal formed by the cross-product of the tangent and normal. Together these form a coordinate system that determines the orientation of the surface relative to the anisotropic illumination model. The surface shown in Fig. 16.47 is mapped to an anisotropic texture that represents a cross-weave of threads. Cabral, Max and Springmeyer [CABR87] have developed a method for determining G and ρ for a surface finish specified by an arbitrary bump map by computing the shadowing and masking effects of the bumps.

The Fresnel equation used by Blinn and by Cook and Torrance is correct only for unpolarized light. The polarization state of light changes, however, when light is reflected from a surface, and a surface's ρ is a function of the polarization state of the light incident on it. Wolff and Kurlander [WOLF90] have extended the Cook–Torrance model to take this into account and have generated images that evidence two effects that are most visible after two or more interobject reflections. First, dielectrics have an angle of incidence, known as the *Brewster angle*, at which incident light is completely polarized when reflected, or is not reflected at all if it is inappropriately polarized. If interobject specular reflection of initially unpolarized light occurs between two dielectric surfaces such that the angle of incidence of each reflection is equal to the Brewster angle, and the plane defined by $\overline{N}$ and $\overline{L}$ on one surface is perpendicular to that of the other, then no light at all will be specularly reflected from the second object; we can produce noticeable, but less dramatic, attenuation if the angles and orientations are varied. Second, colored conductors (metals such as copper or gold) tend to polarize light at different wavelengths differently. Therefore, when a colored conductor is reflected from a dielectric surface, the reflection will have a color slightly different from that when polarization is not taken into account.

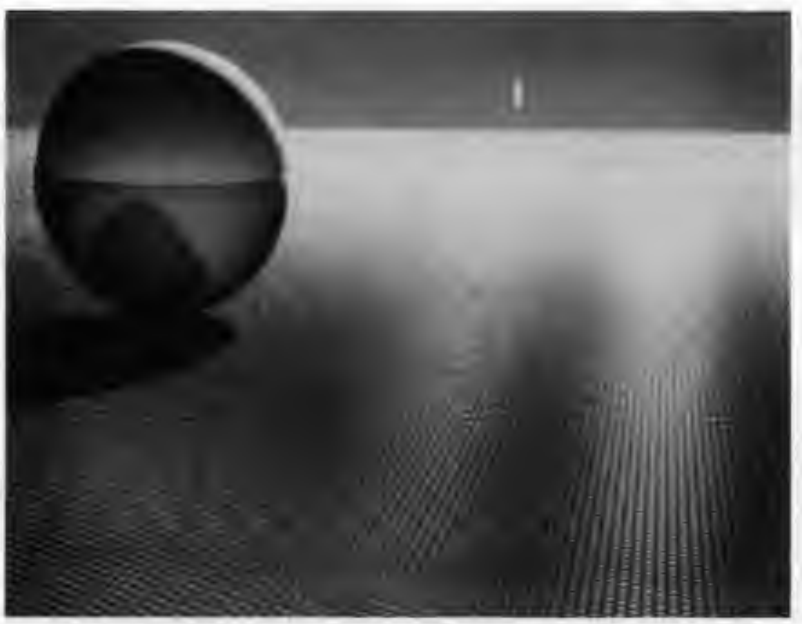


Fig. 16.47 Anisotropic texture. (By J. Kajiya [KAJI85], California Institute of Technology.)

Samsung/Google, EX1027, p. 782

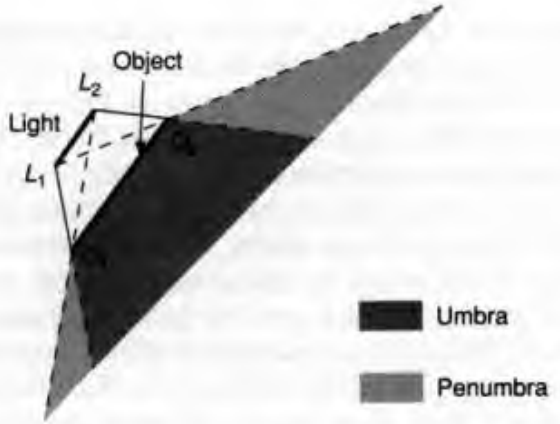


Fig. 16.48 Umbra and penumbra.

16.8 EXTENDED LIGHT SOURCES

The light sources discussed thus far have been point lights. In contrast, *extended* or *distributed* light sources actually have area and consequently cast "soft" shadows containing areas only partially blocked from the source, as shown in Fig. 16.48. The part of a light source's shadow that is totally blocked from the light source is the shadow's *umbra*;

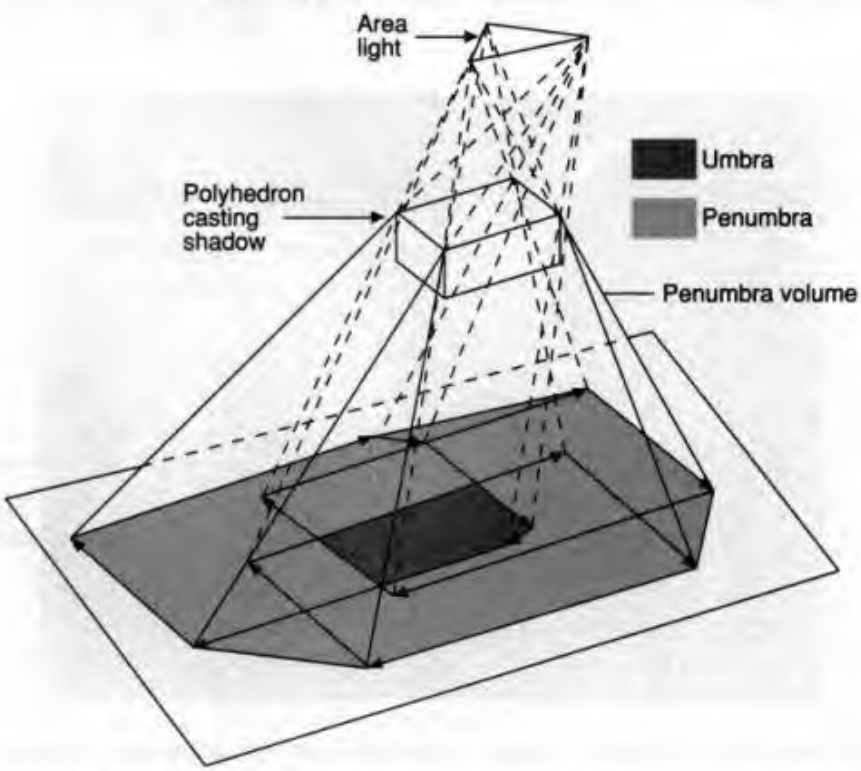


Fig. 16.49 Determining the penumbra and umbra volumes. (After [NISH85a].)

Samsung/Google, EX1027, p. 783

that part of the shadow that is only partially shielded from the source is the shadow's *penumbra*. All of a point light source's shadow is umbra. An obvious approach to modeling an extended light source is to approximate it with closely clustered point light sources [VERB84; BROT84]. The process is computationally expensive, however, and the results are less than satisfactory unless there are very many point sources and no specular reflection.

Nishita and Nakamae [NISH83; NISH85a; NISH85b] have developed an extension of shadow volumes for modeling linear, convex polygonal, and convex polyhedral light sources with Lambertian intensity distributions. Their method employs an object-precision algorithm for determining shadow volumes for convex polyhedral objects. As shown in Fig. 16.49, the shadow volumes defined by each vertex of the source and the polyhedron are determined. The penumbra volume is then the smallest convex polyhedron containing these shadow volumes (their convex hull), and the umbra volume is the intersection of the shadow volumes. The volumes are then intersected with the faces of the objects to determine the areas in each penumbra or umbra volume. Any point lying within an umbra volume defined by a light source and any other polyhedron is not affected by the light source. Determining the color of a point in a penumbra involves computing those parts of the light source visible from the point on the object. A BSP-tree–based approach is discussed in [CHIN90].

Excellent simulations of extended light sources have been achieved by using variations on ray-tracing (Section 16.12) and radiosity methods (Section 16.13).

16.9 SPECTRAL SAMPLING

As light is reflected from a surface, the spectral energy distribution P_λ of the incident light is modified by the spectral reflectance function of the surface, ρ_λ. This curve specifies the percentage of light of each wavelength λ that is reflected. Therefore, the spectral energy distribution of the reflected light is $P_\lambda\rho_\lambda$. Similarly, light passing through a material is modified by the spectral transmittance function of the material, τ_λ. Once a final spectral energy distribution for light falling on the eye at a particular point (i.e., for a particular pixel) is determined, then Eq. (13.18) can be used to find the corresponding CIE XYZ specification for the light, and this result can be converted to RGB using the inverse of M of Eq. (13.24).

It is tempting to assume that ρ_λ and τ_λ can be replaced by equivalent tristimulus RGB or XYZ color specifications. If this could be done, then the product $P_\lambda\rho_\lambda$ could be replaced by a sum of the tristimulus values for each of P_λ and ρ_λ, of the form

$$R\rho_R + G\rho_G + B\rho_B. \tag{16.53}$$

This is clearly incorrect. Consider the P_λ and ρ_λ shown in Fig. 16.50. P_λ is uniform except for a narrow interval around 600 nanometers, whereas ρ_λ reflects light only at the same narrow interval. The product $P_\lambda\rho_\lambda$ is thus zero everywhere, and the surface appears black. On the other hand, the XYZ components of P_λ will be nonzero (this can be seen by applying the inverse of M to the results of Eq. (13.18). Similarly, substituting ρ_λ for P_λ in Eq. (13.18) and applying the inverse of M gives nonzero values for ρ_R, ρ_G, ρ_B, so the product (Eq. 16.53) is also nonzero! This is the wrong answer, and demonstrates that ρ_λ and τ_λ cannot in general be replaced by an equivalent tristimulus color specification. As Hall

Samsung/Google, EX1027, p. 784

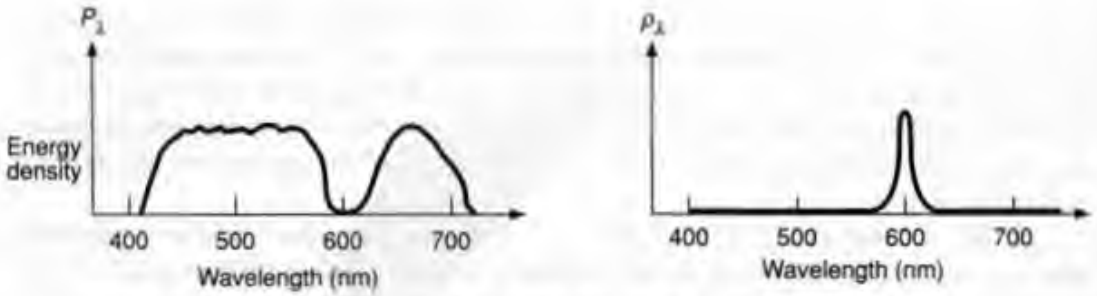


Fig. 16.50 P_λ and ρ_λ whose product is 0 everywhere.

[HALL83] points out, this approach fails because it is actually point sampling in color space and consequently is prone to all the problems that arise from undersampling.

A more accurate representation of a light source's spectral distribution or a surface's reflectivity or transmissivity can be obtained by representing either one as a curve that interpolates a sufficiently large set of samples spaced across the visible spectrum. Color Plate III.9 shows images, generated with different color models, of two overlapping filters lit by a D6500 light source. The filters do not pass a common band of wavelengths, so no transmitted light should be visible where they overlap. Color Plate III.9(a), the control picture, was generated using spectra maintained at 1-nm intervals from 360 to 830 nanometers. Color Plates III.9(b) and III.9(c) were generated with three samples for the primaries of the CIE and RGB color spaces, respectively. Color Plate III.9(d) approximates the spectra of Color Plate III.9(a) with nine spectral values. Note how much more closely Color Plate III.9(d) matches the control picture than do the others.

16.10 IMPROVING THE CAMERA MODEL

Thus far, we have modeled the image produced by a pinhole camera: each object, regardless of its position in the environment, is projected sharply and without distortion in the image. Real cameras (and eyes) have lenses that introduce a variety of distortion and focusing effects.

Depth of field. Objects appear to be more or less in focus depending on their distance from the lens, an effect known as *depth of field*. A lens has a focal length F corresponding to the distance from the lens at which a perfectly focused image of an object converges. If a point is out of focus, its image converges on a plane that is closer or farther than F. An out-of-focus point projects on a plane at F as a circle known as the *circle of confusion*.

Potmesil and Chakravarty [POTM82] have developed a postprocessing technique for simulating some of the effects of depth of field and other properties of real lenses, demonstrated in Color Plate II.38. Their system first produces images using a conventional pinhole-lens renderer that generates not only the intensity at each point, but also that point's z value. Each sampled point is then turned into a circle of confusion with a size and intensity distribution determined by its z value and the lens and aperture being used. The intensity of each pixel in the output image is calculated as a weighted average of the intensities in the circles of confusion that overlap the pixel. Since the image is initially computed from a single point at the center of projection, the results of this technique are

Samsung/Google, EX1027, p. 785

still only an approximation. A real lens's focusing effect causes light rays that would not pass through the pinhole to strike the lens and to converge to form the image. These rays see a slightly different view of the scene, including, for example, parts of surfaces that are not visible to the rays passing through the pinhole. This information is lost in the images created with the Potmesil and Chakravarty model.

Motion blur. *Motion blur* is the streaked or blurred appearance that moving objects have because a camera's shutter is open for a finite amount of time. To achieve this effect, we need to solve the visible-surface problem over time, as well as over space, to determine which objects are visible at a given pixel and when they are visible. Korein and Badler [KORE83] describe two contrasting approaches: an analytic algorithm that uses continuous functions to model the changes that objects undergo over time, and a simple image-precision approach that relies on temporal supersampling. In the temporal-supersampling method, a separate image is rendered for each point in time to be sampled. The motion-blurred image is created by taking a weighted sum of the images, in essence convolving them with a temporal filter. For example, if each of n images is weighted by $1/n$, this corresponds to temporal box filtering. The more closely the samples are spaced, the better the results. Temporal supersampling, like spatial supersampling, suffers from aliasing: Unless samples are spaced sufficiently close together in time, the final image will appear to be a set of discrete multiple exposures. Potmesil and Chakravarty [POTM83] have extended their depth-of-field work to model the ways in which actual camera shutters move. As we shall see in Section 16.12, the stochastic sampling techniques used in distributed ray tracing offer a uniform framework for integrating lens effects, motion blur, and spatial antialiasing demonstrated in Color Plates II.39 and III.16.

16.11 GLOBAL ILLUMINATION ALGORITHMS

An illumination model computes the color at a point in terms of light directly emitted by light sources and of light that reaches the point after reflection from and transmission through its own and other surfaces. This indirectly reflected and transmitted light is often called *global illumination*. In contrast, *local illumination* is light that comes directly from the light sources to the point being shaded. Thus far, we have modeled global illumination by an ambient illumination term that was held constant for all points on all objects. It did not depend on the positions of the object or the viewer, or on the presence or absence of nearby objects that could block the ambient light. In addition, we have seen some limited global illumination effects made possible by shadows, transparency, and reflection maps.

Much of the light in real-world environments does not come from direct light sources. Two different classes of algorithms have been used to generate pictures that emphasize the contributions of global illumination. Section 16.12 discusses extensions to the visible-surface ray-tracing algorithm that interleave visible-surface determination and shading to depict shadows, reflection, and refraction. Thus, global specular reflection and transmission supplement the local specular, diffuse, and ambient illumination computed for a surface. In contrast, the radiosity methods discussed in Section 16.13 completely separate shading and visible-surface determination. They model all an environment's interactions with light sources first in a view-independent stage, and then compute one or more images

Samsung/Google, EX1027, p. 786

for the desired viewpoints using conventional visible-surface and interpolative shading algorithms.

The distinction between view-dependent algorithms, such as ray tracing, and view-independent ones, such as radiosity, is an important one. *View-dependent* algorithms discretize the view plane to determine points at which to evaluate the illumination equation, given the viewer's direction. In contrast, *view-independent* algorithms discretize the environment, and process it in order to provide enough information to evaluate the illumination equation at any point and from any viewing direction. View-dependent algorithms are well-suited for handling specular phenomena that are highly dependent on the viewer's position, but may perform extra work when modeling diffuse phenomena that change little over large areas of an image, or between images made from different viewpoints. On the other hand, view-independent algorithms model diffuse phenomena efficiently, but require overwhelming amounts of storage to capture enough information about specular phenomena.

Ultimately, all these approaches attempt to solve what Kajiya [KAJI86] has referred to as the *rendering equation*, which expresses the light being transferred from one point to another in terms of the intensity of the light emitted from the first point to the second and the intensity of light emitted from all other points that reaches the first and is reflected from the first to the second. The light transferred from each of these other points to the first is, in turn, expressed recursively by the rendering equation. Kajiya presents the rendering equation as

$$I(x, x') = g(x, x')\left[\varepsilon(x, x') + \int_S \rho(x, x', x'')\, I(x', x'')dx''\right], \qquad (16.54)$$

where x, x', and x'' are points in the environment. $I(x, x')$ is related to the intensity passing from x' to x. $g(x, x')$ is a geometry term that is 0 when x and x' are occluded from each other, and $1 / r^2$ when they are visible to each other, where r is the distance between them. $\varepsilon(x, x')$ is related to the intensity of light that is emitted from x' to x. The initial evaluation of $g(x, x')\varepsilon(x, x')$ for x at the viewpoint accomplishes visible-surface determination in the sphere about x. The integral is over all points on all surfaces S. $\rho(x, x', x'')$ is related to the intensity of the light reflected (including both specular and diffuse reflection) from x'' to x from the surface at x'. Thus, the rendering equation states that the light from x' that reaches x consists of light emitted by x' itself and light scattered by x' to x from all other surfaces, which themselves emit light and recursively scatter light from other surfaces.

As we shall see, how successful an approach is at solving the rendering equation depends in large part on how it handles the remaining terms and the recursion, on what combinations of diffuse and specular reflectivity it supports, and on how well the visibility relationships between surfaces are modeled.

16.12 RECURSIVE RAY TRACING

In this section, we extend the basic ray-tracing algorithm of Section 15.10 to handle shadows, reflection, and refraction. This simple algorithm determined the color of a pixel at the closest intersection of an eye ray with an object, by using any of the illumination models described previously. To calculate shadows, we fire an additional ray from the point of intersection to each of the light sources. This is shown for a single light source in Fig.

Samsung/Google, EX1027, p. 787

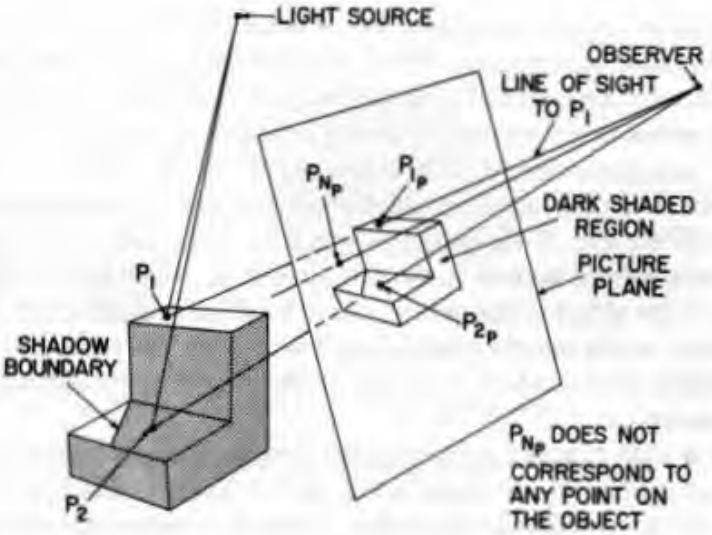


Fig. 16.51 Determining whether a point on an object is in shadow. (Courtesy of Arthur Appel, IBM T.J. Watson Research Center.)

16.51, which is reproduced from a paper by Appel [APPE68]—the first paper published on ray tracing for computer graphics. If one of these *shadow rays* intersects any object along the way, then the object is in shadow at that point and the shading algorithm ignores the contribution of the shadow ray's light source. Figure 16.52 shows two pictures that Appel rendered with this algorithm, using a pen plotter. He simulated a halftone pattern by placing a different size "+" at each pixel in the grid, depending on the pixel's intensity. To compensate for the grid's coarseness, he drew edges of visible surfaces and of shadows using a visible-line algorithm.

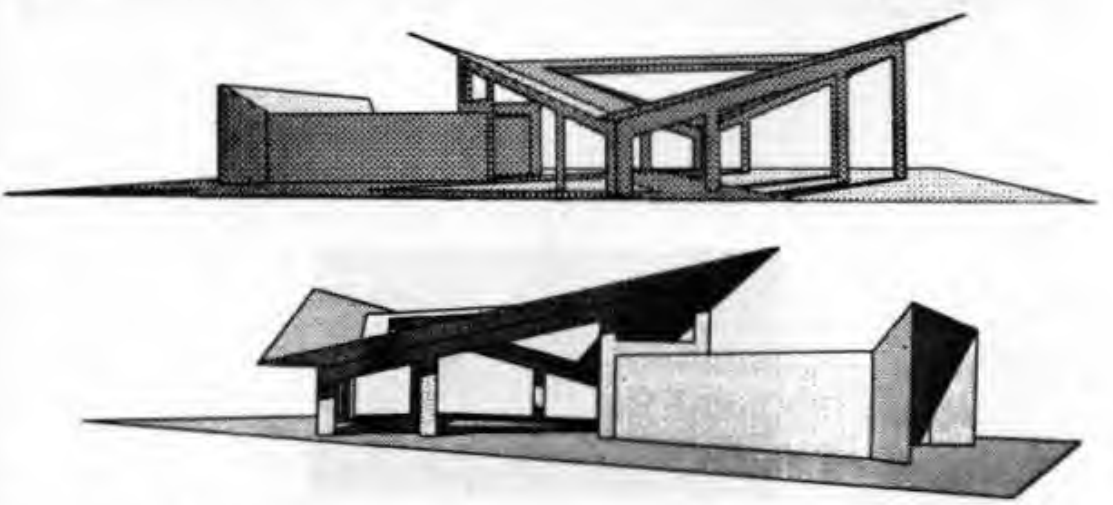


Fig. 16.52 Early pictures rendered with ray tracing. (Courtesy of Arthur Appel, IBM T.J. Watson Research Center.)

Samsung/Google, EX1027, p. 788

The illumination model developed by Whitted [WHIT80] and Kay [KAY79a] fundamentally extended ray tracing to include specular reflection and refractive transparency. Color Plate III.10 is an early picture generated with these effects. In addition to shadow rays, Whitted's recursive ray-tracing algorithm conditionally spawns *reflection rays* and *refraction rays* from the point of intersection, as shown in Fig. 16.53. The shadow, reflection, and refraction rays are often called *secondary rays*, to distinguish them from the *primary rays* from the eye. If the object is specularly reflective, then a reflection ray is reflected about the surface normal in the direction of $\overline{R}$, which may be computed as in Section 16.1.4. If the object is transparent, and if total internal reflection does not occur, then a refraction ray is sent into the object along $\overline{T}$ at an angle determined by Snell's law, as described in Section 16.5.2. (Note that your incident ray may be oppositely oriented to those in these sections.)

Each of these reflection and refraction rays may, in turn, recursively spawn shadow, reflection, and refraction rays, as shown in Fig. 16.54. The rays thus form a *ray tree*, such as that of Fig. 16.55. In Whitted's algorithm, a branch is terminated if the reflected and refracted rays fail to intersect an object, if some user-specified maximum depth is reached or if the system runs out of storage. The tree is evaluated bottom-up, and each node's intensity is computed as a function of its children's intensities. Color Plate III.11(a) and (b) were made with a recursive ray-tracing algorithm.

We can represent Whitted's illumination equation as

$$I_\lambda = I_{a\lambda}k_aO_{d\lambda} + \sum_{1 \le i \le m} S_i f_{att_i} I_{p\lambda_i}[k_dO_{d\lambda}(\overline{N} \cdot \overline{L}_i) + k_s(\overline{N} \cdot \overline{H}_i)^n] + k_sI_{r\lambda} + k_tI_{t\lambda}, \quad (16.55)$$

where $I_{r\lambda}$ is the intensity of the reflected ray, k_t is the *transmission coefficient* ranging between 0 and 1, and $I_{t\lambda}$ is the intensity of the refracted transmitted ray. Values for $I_{r\lambda}$ and $I_{t\lambda}$ are determined by recursively evaluating Eq. (16.55) at the closest surface that the reflected and transmitted rays intersect. To approximate attenuation with distance, Whitted multiplied the I_λ calculated for each ray by the inverse of the distance traveled by the ray. Rather than treating S_i as a delta function, as in Eq. (16.24), he also made it a continuous function

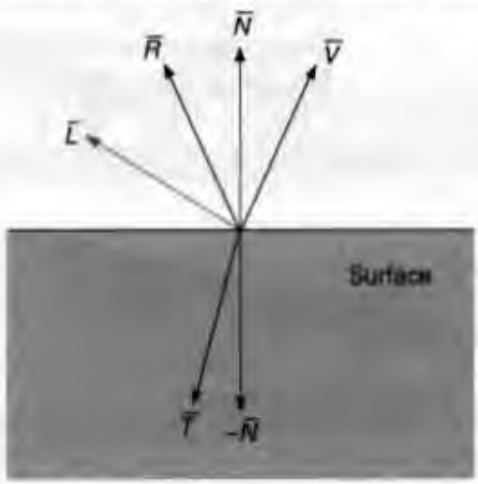


Fig. 16.53 Reflection, refraction, and shadow rays are spawned from a point of intersection.

Samsung/Google, EX1027, p. 789

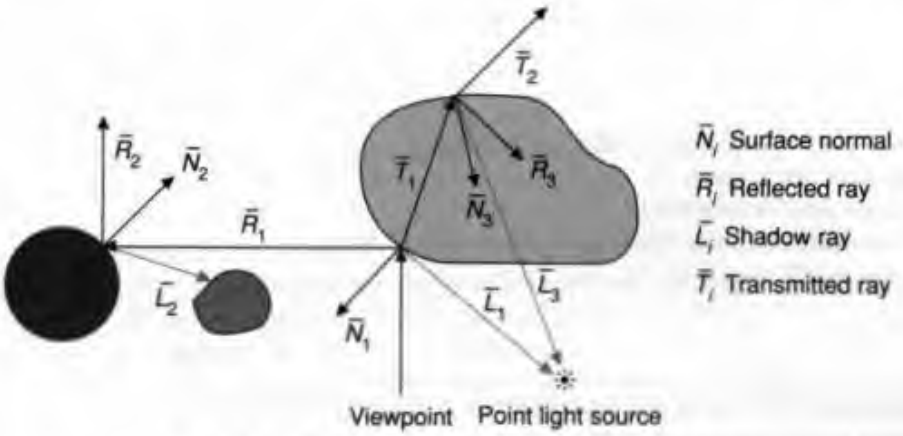


Fig. 16.54 Rays recursively spawn other rays.

of the k_t of the objects intersected by the shadow ray, so that a transparent object obscures less light than an opaque one at those points it shadows.

Figure 16.56 shows pseudocode for a simple recursive ray tracer. RT_trace determines the closest intersection the ray makes with an object and calls RT_shade to determine the shade at that point. First, RT_shade determines the intersection's ambient color. Next, a shadow ray is spawned to each light on the side of the surface being shaded to determine its contribution to the color. An opaque object blocks the light totally, whereas a transparent one scales the light's contribution. If we are not too deep in the ray tree, then recursive calls are made to RT_trace to handle reflection rays for reflective objects and refraction rays for transparent objects. Since the indices of refraction of two media are needed to determine the direction of the refraction ray, the index of refraction of the material in which a ray is traveling can be included with each ray. RT_trace retains the ray tree only long enough to determine the current pixel's color. If the ray trees for an entire image can be preserved, then surface properties can be altered and a new image recomputed relatively quickly, at the

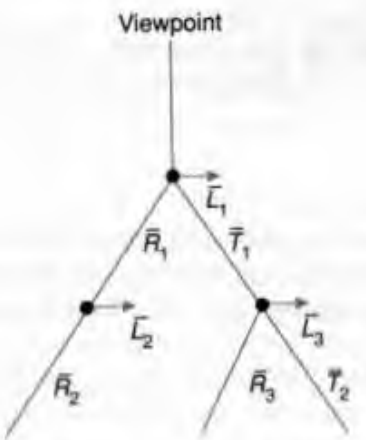


Fig. 16.55 The ray tree for Fig. 16.54.

Samsung/Google, EX1027, p. 790

```
select center of projection and window on view plane;
for (each scan line in image) {
    for (each pixel in scan line) {
        determine ray from center of projection through pixel;
        pixel = RT_trace (ray, 1);
    }
}

/* Intersect ray with objects and compute shade at closest intersection. */
/* Depth is current depth in ray tree. */

RT_color RT_trace (RT_ray ray, int depth)
{
    determine closest intersection of ray with an object;

    if (object hit) {
        compute normal at intersection;
        return RT_shade (closest object hit, ray, intersection, normal, depth);
    } else
        return BACKGROUND_VALUE;
}  /* RT_trace */

/* Compute shade at point on object, tracing rays for shadows, reflection and refraction. */
RT_color RT_shade (
    RT_object object,          /* Object intersected */
    RT_ray ray,                /* Incident ray */
    RT_point point,            /* Point of intersection to shade */
    RT_normal normal,          /* Normal at point */
    int depth)                 /* Depth in ray tree */
{
    RT_color color;            /* Color of ray */
    RT_ray rRay, tRay, sRay;   /* Reflected, refracted, and shadow rays */
    RT_color rColor, tColor;   /* Reflected and refracted ray colors */

    color = ambient term;
    for (each light) {
        sRay = ray to light from point;
        if (dot product of normal and direction to light is positive) {
            compute how much light is blocked by opaque and transparent surfaces,
                and use to scale diffuse and specular terms before adding them to color;
        }
    }
```

Fig. 16.56 *(Cont'd.)*

Samsung/Google, EX1027, p. 791

```
if (depth < maxDepth) {        /* Return if depth is too deep. */
    if (object is reflective) {
        rRay = ray in reflection direction from point;
        rColor = RT_trace (rRay, depth + 1);
        scale rColor by specular coefficient and add to color;
    }

    if (object is transparent) {
        tRay = ray in refraction direction from point;
        if (total internal reflection does not occur) {
            tColor = RT_trace (tRay, depth + 1);
            scale tColor by transmission coefficient and add to color;
        }
    }
}
return color;                  /* Return color of ray */
} /* RT_shade */
```

Fig. 16.56 Pseudocode for simple recursive ray tracing without antialiasing.

cost of only reevaluating the trees. Sequin and Smyrl [SEQU89] present techniques that minimize the time and space needed to process and store ray trees.

Figure 16.54 shows a basic problem with how ray tracing models refraction: The shadow ray $\overline{L}_3$ is not refracted on its path to the light. In fact, if we were to simply refract $\overline{L}_3$ from its current direction at the point where it exits the large object, it would not end at the light source. In addition, when the paths of rays that are refracted are determined, a single index of refraction is used for each ray. Later, we discuss some ways to address these failings.

Ray tracing is particularly prone to problems caused by limited numerical precision. These show up when we compute the objects that intersect with the secondary rays. After the x, y, and z coordinates of the intersection point on an object visible to an eye ray have been computed, they are then used to define the starting point of the secondary ray for which we must determine the parameter t (Section 15.10.1). If the object that was just intersected is intersected with the new ray, it will often have a small, nonzero t, because of numerical-precision limitations. If not dealt with, this false intersection can result in visual problems. For example, if the ray were a shadow ray, then the object would be considered as blocking light from itself, resulting in splotchy pieces of incorrectly "self-shadowed" surface. A simple way to solve this problem for shadow rays is to treat as a special case the object from which a secondary ray is spawned, so that intersection tests are not performed on it. Of course, this does not work if objects are supported that really could obscure themselves or if transmitted rays have to pass through the object and be reflected from the inside of the same object. A more general solution is to compute abs(t) for an intersection, to compare it with a small tolerance value, and to ignore it if it is below the tolerance.

Samsung/Google, EX1027, p. 792

The paper Whitted presented at *SIGGRAPH '79* [WHIT80], and the movies he made using the algorithm described there, started a renaissance of interest in ray tracing. Recursive ray tracing makes possible a host of impressive effects—such as shadows, specular reflection, and refractive transparency—that were difficult or impossible to obtain previously. In addition, a simple ray tracer is quite easy to implement. Consequently, much effort has been directed toward improving both the algorithm's efficiency and its image quality. We provide a brief overview of these issues here, and discuss several parallel hardware implementations that take advantage of the algorithm's intrinsic parallelism in Section 18.11.2. For more detail, see [GLAS89].

16.12.1 Efficiency Considerations for Recursive Ray Tracing

Section 15.10.2 discussed how to use extents, hierarchies, and spatial partitioning to limit the number of ray–object intersections to be calculated. These general efficiency techniques are even more important here than in visible-surface ray tracing for several reasons. First, a quick glance at Fig. 16.55 reveals that the number of rays that must be processed can grow exponentially with the depth to which rays are traced. Since each ray may spawn a reflection ray and a refraction ray, in the worst case, the ray tree will be a complete binary tree with $2^n - 1$ rays, where the tree depth is n. In addition, each reflection or refraction ray that intersects with an object spawns one shadow ray for each of the m light sources. Thus, there are potentially $m(2^n - 1)$ shadow rays for each ray tree. To make matters worse, since rays can come from any direction, traditional efficiency ploys, such as clipping objects to the view volume and culling back-facing surfaces relative to the eye, cannot be used in recursive ray tracing. Objects that would otherwise be invisible, including back faces, may be reflected from or refracted through visible surfaces.

Item buffers. One way of speeding up ray tracing is simply not to use it at all when determining those objects directly visible to the eye. Weghorst, Hooper, and Greenberg [WEGH84] describe how to create an *item buffer* by applying a less costly visible-surface algorithm, such as the z-buffer algorithm, to the scene, using the same viewing specification. Instead of determining the shade at each pixel, however, they record in the item buffer pixel the identity of the closest object. Then, only this object needs to be processed by the ray tracer to determine the eye ray's exact intersection for this pixel, so that further rays may be spawned.

Reflection maps. Tracing rays can be avoided in other situations, too. Hall [HALL86] shows how to combine ray tracing with the reflection maps discussed in Section 16.6. The basic idea is to do less work for the secondary rays than for primary rays. Those objects that are not directly visible in an image are divided into two groups on the basis of an estimation of their indirect visibility. Ray tracing is used to determine the global lighting contributions of the more visible ones, whereas indexing into a suitably prepared reflection map handles the others. One way to estimate the extent to which an object is indirectly visible is to measure the solid angle subtended by the directly visible objects as seen from the centroid of the indirectly visible object. If the solid angle is greater than some threshold, then the object will be included in the environment to be traced by reflection rays (this environment

Samsung/Google, EX1027, p. 793

includes the directly visible objects); otherwise, the object will be represented only in the reflection map. When ray tracing is performed, if a reflection ray does not intersect one of the objects in the reflection-ray environment, then the ray is used to index into the reflection map. Hall also points out that reflected and refracted images are often extremely distorted. Therefore, good results may be achieved by intersecting the reflection and refraction rays with object definitions less detailed than those used for the eye rays.

Adaptive tree-depth control. Although ray tracing is often used to depict highly specular objects, most of an image's area is usually not filled with such objects. Consequently, a high recursion level often results in unnecessary processing for large parts of the picture. Hall [HALL83] introduced the use of *adaptive tree-depth control*, in which a ray is not cast if its contribution to the pixel's intensity is estimated to be below some preset threshold. This is accomplished by approximating a ray's maximum contribution by calculating its intensity with the assumption that the ray's child rays have intensities of 1. This allows the ray's contribution to its parent to be estimated. As the ray tree is built, the maximum contribution of a ray is multiplied by those of its ancestors to derive the ray's maximum contribution to the pixel. For example, suppose that $\overline{R}_1$ and $\overline{R}_2$ in Fig. 16.55 are spawned at surfaces with k_s values of .1 and .05, respectively. At the first intersection, we estimate the maximum contribution to the pixel of $\overline{R}_1$ to be .1. At the second intersection, we estimate the maximum contribution to the pixel of $\overline{R}_2$ to be $.05 \times .1 = .005$. If this is below our threshold, we may decide not to cast $\overline{R}_2$. Although adaptive tree-depth control has been shown to work well for many images, it is easy to design cases in which it will fail. Although one uncast ray may have an imperceptible effect on a pixel's shade, a pixel may receive a significant amount of light from a large number of individually insignificant rays.

Light buffers. We noted that m shadow rays are spawned for each reflection or refraction ray that hits an object. Shadow rays are special, however, in that each is fired toward one of a relatively small set of objects. Haines and Greenberg [HAIN86] have introduced the notion of a *light buffer* to increase the speed with which shadow rays are processed. A light buffer is a cube centered about a light source and aligned with the world-coordinate axes, as shown in Fig. 16.57(a). Each side is tiled with a regular grid of squares, and each square is associated with a depth-sorted list of surfaces that can be seen through it from the light. The

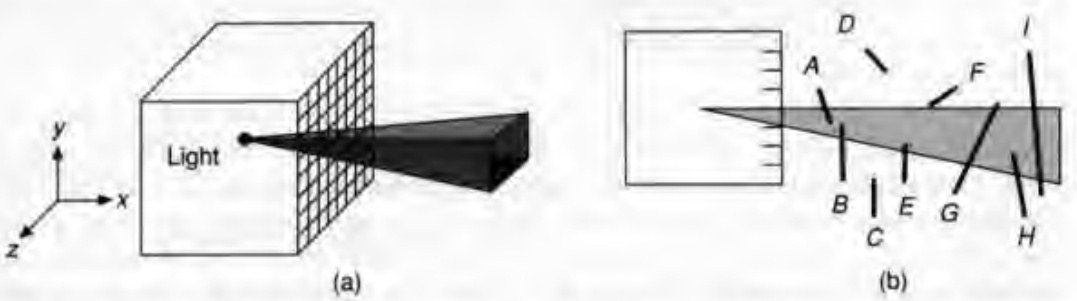


Fig. 16.57 Light buffer centered about a light source. (a) Volume defined by one square. (b) Cross-section shows volume of square in 2D with intersected surfaces. Square's depth-sorted list is initially *A*, *B*, *E*, *G*, *H*, and *I*.

Samsung/Google, EX1027, p. 794

lists are filled by scan converting the objects in the scene onto each face of the light buffer with the center of projection at the light. The scan-conversion algorithm places an object on the list of every square covered by that object's projection, no matter how small the overlap. Figure 16.57(b) shows a square's set of intersecting surfaces. A shadow ray is processed by determining the light-buffer square through which it passes. The ray needs to be tested only for intersection against that square's ordered list of surfaces. Thus, a light buffer implements a kind of 3D spatial partitioning of the 3D view of its light.

A number of efficiency speedups are possible. For example, any object whose projection onto the light buffer totally covers a square (e.g., G in Fig. 16.57b) can be associated with a special "full-occlusion" record in the square's list; any object more distant from the light than a fully occluding object is purged from the list. In this case, H and I are purged. Whenever the ray from an intersection point is tested against a square's list, the test can be terminated immediately if there is a full-occlusion record closer to the light than the intersection point. In addition, back-face culling may be used to avoid adding any faces to a list that are both part of a closed opaque solid and back facing relative to the light. Color Plate III.12 was made using the light-buffer technique.

Haines and Greenberg also mention an interesting use of object coherence to determine shadows that can be applied even without using light buffers. A pointer is associated with each light and is initialized to null. When an object is found to be shadowed from a light by some opaque surface, the light's pointer is set to the shadowing surface. The next time a shadow ray is cast for the light, it is first intersected with the object pointed at by the light's pointer. If the ray hits it, then intersection testing is finished; otherwise, the pointer is set to null and testing continues.

Ray classification. The spatial-partitioning approaches discussed in Section 15.10.2 make it possible to determine which objects lie in a given region of 3D space. Arvo and Kirk [ARVO87] have extended this concept to partition rays by the objects that they intersect, a technique called *ray classification*. A ray may be specified by its position in 5D *ray space*, determined by the 3D position of its origin in space and its 2D direction in spherical coordinates. A point in ray space defines a single ray, whereas a finite subset of ray space defines a family of rays or *beam*. Ray classification adaptively partitions ray space into subsets, each of which is associated with a list of objects that it contains (i.e., that one of its rays could intersect). To determine the candidate objects that may be intersected by any ray, we need only to retrieve the list of objects associated with the subset of ray space in which the ray resides.

Instead of using spherical coordinates to specify direction, Arvo and Kirk use the (u, v) coordinate system on each of the six sides of an axis-aligned cube centered about a ray's origin. They subdivide ray space with six copies of a 5D bintree (Section 12.6.3); one copy is needed for each direction along an axis, since a (u, v) pair specifies a position on the cube's side, but not the side itself. As shown in Fig. 16.58(a), in 2D a set of ray origins (a rectangle in (x, y)), combined with a set of ray directions (an interval in u on one of four sides), define a partially bounded polygon that defines the beam. In 3D, shown in Fig. 16.58(b), a set of ray origins (a rectangular parallepiped in (x, y, z)), combined with a set of ray directions (a rectangle in (u, v)), defines a partially bounded polyhedral volume that defines the beam. Objects (or their extents) may be intersected with this volume (e.g., using

Samsung/Google, EX1027, p. 795

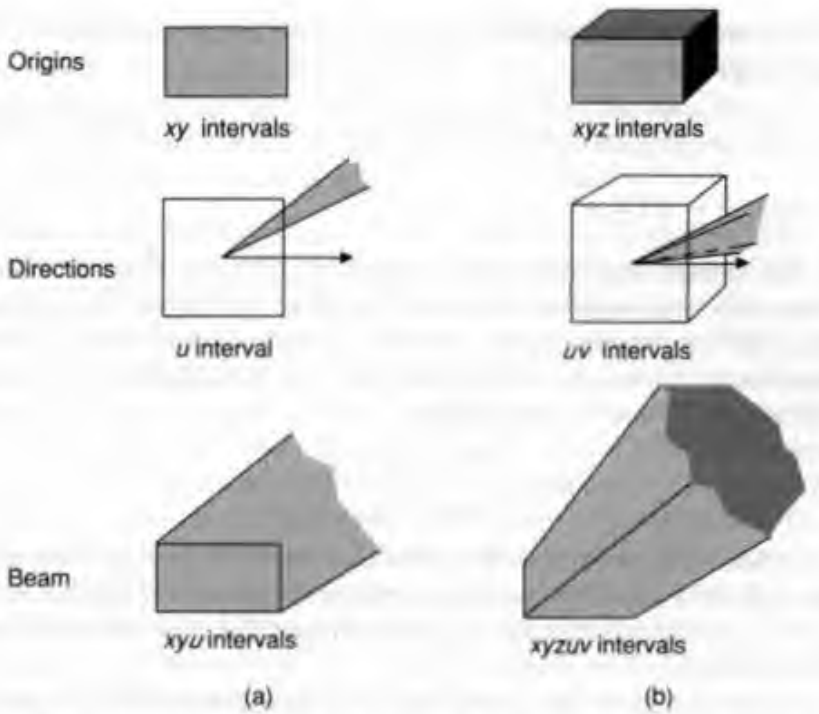


Fig. 16.58 Ray classification: (a) 2-space beams; (b) 3-space beams. (After [ARVO87].)

the BSP-tree-based approach of Section 12.6.4) to determine whether they are contained in it. Initially, each unpartitioned bintree represents positions in all of the environment's 3-space extent, and directions that pass through any part of the bintree's designated side. Each bintree is initially associated with all the objects in the environment.

Arvo and Kirk use *lazy evaluation* to build the trees at the same time that they trace rays. When a ray is traced, its direction selects the root of the bintree whose (u, v) coordinate system it will intersect. If more than some threshold number of objects exist at the node, the bintree is subdivided equally along each of its five axes. With each subdivision, only the child node in which the ray resides is processed; it inherits from its parent those objects that lie within the child. When subdivision stops, only those objects associated with the ray's partition are returned for intersection. Partitioning data structures as the rays are traced minimizes the amount of processing done for objects that are not intersected.

16.12.2 A Better Illumination Model

It is possible to make a number of extensions to Eq. (16.55). Hall [HALL83] has developed a model in which the specular light expressions are scaled by a wavelength-dependent Fresnel reflectivity term. An additional term for transmitted local light is added to take into account the contribution of transmitted light directly emitted by the light sources, and is also scaled by the Fresnel transmissivity term. The global reflected and refracted rays

Samsung/Google, EX1027, p. 796

further take into account the transmittance of the medium through which they travel. Hall's model may be expressed as

$$I_\lambda = I_{a\lambda}k_a O_{d\lambda} + \sum_{1 \le i \le m} S_i f_{atti}\, I_{p\lambda i}\, [k_d O_{d\lambda}(\overline{N} \cdot \overline{L}_i) + k_s(\overline{N} \cdot \overline{H}_i)^n F_\lambda + k_s(\overline{N} \cdot \overline{H'}_i)^n T_\lambda]$$

$$+ k_s F_\lambda I_{r\lambda} A_r^{d_r} + k_s T_\lambda I_{t\lambda} A_t^{d_t}, \tag{16.56}$$

where d_r is the distance traveled by the reflected ray, d_t is the distance traveled by the transmitted ray, A_r is the transmissivity per unit length of material for the reflected ray, and A_t is the transmissivity per unit length of material for the transmitted ray. $\overline{H'}$ is the normal for those microfacets that are aligned such that they will refract light directly from the light source to the viewer. It may be computed as

$$\overline{H'} = \frac{\overline{V} - (\eta_{avg\theta_t} / \eta_{avg\theta_i})\overline{L}}{(\eta_{avg\theta_t} / \eta_{avg\theta_i}) - 1}, \tag{16.57}$$

where $\eta_{avg\theta_t}$ is the average coefficient of refraction of the object through which the ray to the light passes. T_λ is the Fresnel transmissivity term for the material. Note that k_t is not used here; instead, k_s is scaled by either F_λ or T_λ. Since all light that is not reflected is transmitted (and possibly) absorbed, $F_\lambda + T_\lambda = 1$.

Color Plate III.13 shows the same scene rendered with Whitted's model and with Hall's model. The color of the glass sphere in part (b) clearly shows the filtering effects resulting from the use of the transmissivity terms.

16.12.3 Area-Sampling Variations

One of conventional ray tracing's biggest drawbacks is that this technique point samples on a regular grid. Whitted [WHIT80] suggested that unweighted area sampling could be accomplished by replacing each linear eye ray with a pyramid defined by the eye and the four corners of a pixel. These pyramids would be intersected with the objects in the environment, and sections of them would be recursively refracted and reflected by the objects that they intersect. A pure implementation of this proposal would be exceedingly complex, however, since it would have to calculate exact intersections with occluding objects, and to determine how the resulting pyramid fragments are modified as they are recursively reflected from and refracted by curved surfaces. Nevertheless, it has inspired several interesting algorithms that accomplish antialiasing, and at the same time decrease rendering time by taking advantage of coherence.

Cone tracing. *Cone tracing*, developed by Amanatides [AMAN84], generalizes the linear rays into cones. One cone is fired from the eye through each pixel, with an angle wide enough to encompass the pixel. The cone is intersected with objects in its path by calculating approximate fractional blockage values for a small set of those objects closest to the cone's origin. Refraction and reflection cones are determined from the optical laws of spherical mirrors and lenses as a function of the surface curvature of the object intersected and the area of intersection. The effects of scattering on reflection and refraction are simulated by further broadening the angles of the new reflection and refraction cones. The

Samsung/Google, EX1027, p. 797

soft-edged shadows of extended light sources are reproduced by modeling the sources as spheres. A shadow cone is generated whose base is the cross-section of the light source. The light source's intensity is then scaled by the fraction of the cone that is unblocked by intervening objects. Color Plate III.14 was rendered using cone tracing; the three spheres have successively rougher surfaces, and all cast soft shadows.

Beam tracing. *Beam tracing*, introduced by Heckbert and Hanrahan [HECK84], is an object-precision algorithm for polygonal environments that traces pyramidal beams, rather than linear rays. Instead of tracing beams through each pixel, as Whitted suggested, Heckbert and Hanrahan take advantage of coherence by beginning with a single beam defined by the viewing pyramid. The viewing pyramid's beam is intersected with each polygon in the environment, in front-to-back sorted order, relative to the pyramid's apex. If a polygon is intersected, and therefore visible, it must be subtracted from the pyramid using an algorithm such as that described in Section 19.1.4. For each visible polygon fragment, two polyhedral pyramids are spawned, one each for reflection and refraction. The algorithm proceeds recursively, with termination criteria similar to those used in ray tracing. The environment is transformed into each new beam's coordinate system by means of an appropriate transformation. Although beam tracing models reflection correctly, refraction is not a linear transformation since it bends straight lines, and the refracted beam is thus only approximated. Beam tracing produces an object-precision beam tree of polygons that may be recursively rendered using a polygon scan-conversion algorithm. Each polygon is rendered using a local illumination equation, and then its reflected and refracted child polygons are rendered on top and are averaged with it, taking into account the parent's specular and transmissive properties. Beam tracing takes advantage of coherence to provide impressive speedups over conventional ray tracing at the expense of a more complex algorithm, limited object geometry, and incorrectly modeled refraction. Color Plate III.15 was rendered using this algorithm.

Beam tracing can accommodate shadows by using a variant of the Atherton–Weiler–Greenberg shadow algorithm (Section 16.4.2). Beams are traced from the point of view of each light source to determine all surfaces directly visible from the light sources, and the resulting polygons are added to the data structure as lit detail polygons that affect only the shading. This produces shadows similar to those obtained with conventional ray tracing.

Pencil tracing. Shinya, Takahashi, and Naito [SHIN87] have implemented an approach called *pencil tracing* that solves some of the problems of cone tracing and beam tracing. A *pencil* is a bundle of rays consisting of a central *axial* ray, surrounded by a set of nearby *paraxial* rays. Each paraxial ray is represented by a 4D vector that represents its relationship to the axial ray. Two dimensions express the paraxial ray's intersection with a plane perpendicular to the axial ray; the other two dimensions express the paraxial ray's direction. In many cases, only an axial ray and solid angle suffice to represent a pencil. If pencils of sufficiently small solid angle are used, then reflection and refraction can be approximated well by a linear transformation expressed as a 4×4 matrix. Shinya, Takahashi, and Naito have developed error-estimation techniques for determining an appropriate solid angle for a pencil. Conventional rays must be traced where a pencil would intersect the edge of an object, however, since the paraxial transformations are not valid in these cases.

Samsung/Google, EX1027, p. 798

16.12.4 Distributed Ray Tracing

The approaches we have just discussed avoid the aliasing problems of regular point sampling by casting solid beams rather than infinitesimal rays. In contrast, *distributed ray tracing*, developed by Cook, Porter, and Carpenter [COOK84b], is based on a stochastic approach to supersampling that trades off the objectionable artifacts of aliasing for the less offensive artifacts of noise [COOK86]. As we shall see, the ability to perform antialiased spatial sampling can also be exploited to sample a variety of other aspects of the scene and its objects to produce effects such as motion blur, depth of field, extended light sources, and specular reflection from rough surfaces. The word *distributed* in this technique's name refers to the fact that rays are stochastically distributed to sample the quantities that produce these effects. The basic concepts have also been applied to other algorithms besides ray tracing [COOK87].

Stochastic sampling. As explained in Section 14.10, aliasing results when a signal is sampled with regularly spaced samples below the Nyquist rate. This is true even if we supersample and filter to compute the value of a pixel. If the samples are not regularly spaced, however, the sharply defined frequency spectrum of the aliases is replaced by noise, an artifact that viewers find much less objectionable than the individually recognizable frequency components of regular aliasing, such as staircasing.

It is not enough, however, merely to replace ray tracing's regular grid of eye rays with an equal number of rays passing through random points on the image plane, since purely random samples cluster together in some areas and leave others unsampled. Cook [COOK86] suggests the desirability of a minimum-distance Poisson distribution in which no pair of random samples is closer than some minimum distance. Calculating such a distribution is expensive, however, and even if one is created in advance, along with filters to determine each sample's contributions to neighboring pixels, a very large look-up table will be required to store the information. Instead, a satisfactory approximation to the minimum-distance Poisson distribution is obtained by displacing by a small random distance the position of each element of a regularly spaced sample grid. This technique is called *jittering*. In sampling the 2D image plane, each sample in a regular grid is jittered by two uncorrelated random quantities, one each for x and y, both generated with a sufficiently small variance that the samples do not overlap (Fig. 16.59). Figure 16.60 shows a minimum-distance Poisson distribution and a jittered regular distribution. In fact, if the



Fig. 16.59 Jittered sampling. Each sample in a regular 2D grid is jittered by two small uncorrelated random quantities. × = original position; ● = position after jitter.

Samsung/Google, EX1027, p. 799

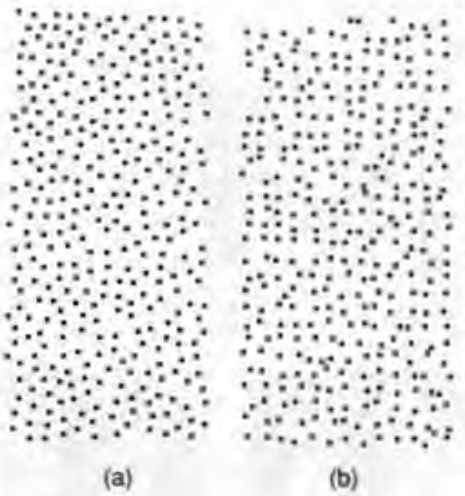


Fig. 16.60 (a) A minimum distance Poisson distribution. (b) A jittered regular distribution. (Courtesy of Mark A. Z. Dippé and Earl Wold, University of California, Berkeley.)

amount of jitter is small compared to the filter width, then the filter can be precomputed, taking into account the positions of the unjittered samples. Cook, Porter, and Carpenter found that a 4×4 subpixel grid is adequate for most situations. Poisson and jittered sampling are analyzed in [DIPP85], and strategies for performing adaptive stochastic sampling, including the statistical analysis of samples to determine whether to place new ones, are discussed in [DIPP85; LEE85b; KAJI86; MITC87].

Figure 16.61 compares the use of regularly spaced samples with and without added jitter to sample frequencies at rates above and below the Nyquist rate. In Fig. 16.61(a), sampling above the Nyquist rate, the shape of the signal is captured well, but with some added noise. In Fig. 16.61(b), the sampled amplitude is totally random if there is an integral number of cycles in the sampled range. If there is a fractional number of cycles in the range, then some parts of the waveform have a better chance of being sampled than do others, and thus a combination of aliasing and noise will result. The higher the frequency, the greater the proportion of noise to aliasing. Figure 16.62 demonstrates how a comb of regularly spaced triangles, each $(n + 1)/n$ pixels wide, produces an aliased image when it is sampled by regularly spaced sample points, and produces a noisy image when the sample points are jittered.

Sampling other dimensions. As long as the extra rays needed for spatial antialiasing have been cast, this same basic technique of stochastic sampling can also be used to distribute the rays to sample other aspects of the environment. Motion blur is produced by distributing rays in time. Depth of field is modeled by distributing the rays over the area of the camera lens. The blurred specular reflections and translucent refraction of rough surfaces are simulated by distributing the rays according to the specular reflection and transmission functions. Soft shadows are obtained by distributing the shadow rays over the solid angle subtended by an extended light source as seen from the point being shaded. In all cases, distributed ray tracing uses stochastic sampling to perturb the same rays that would be cast to accomplish spatial antialiasing alone.

Samsung/Google, EX1027, p. 800

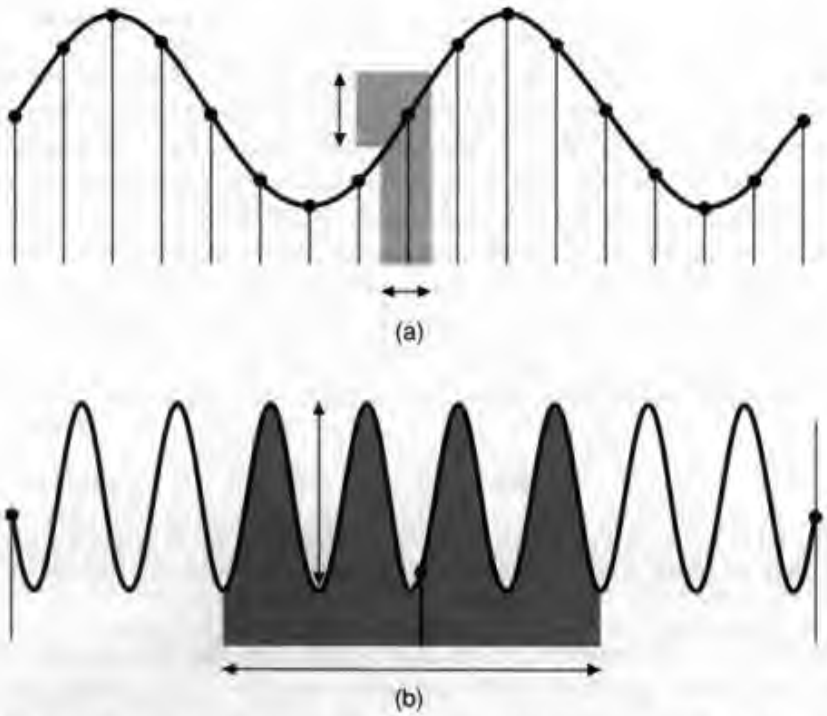


Fig. 16.61 Use of regularly spaced samples with added jitter to sample frequencies (a) above and (b) below the Nyquist rate. The nominal position of each sample is shown as a dot (•). Horizontal arrows indicate the range of jittered sample positions. Vertical arrows indicate how much sampled values can vary. (After [COOK86].)

Sampling in a nonspatial dimension is accomplished by associating each of the pixel's subsampled rays with a range of the value being sampled. Jittering is then used to determine the exact sample point. The ranges may be allocated by partitioning the entire interval being sampled into the same number of subintervals as there are subpixels and randomly allocating subintervals to subpixels. Thus, subpixel *ij* of each pixel is always associated with the same range for a particular dimension. It is important to ensure that the method of allocating the ranges for each dimension does not correlate the values of any two dimensions. For example, if temporally earlier samples tended to sample the left side of an extended light source, and later samples tended to sample the right side, then obscuring the right side of the light early in the temporal interval being depicted might have no effect on the shadow cast. In the case of temporal sampling, each object being intersected must first be moved to its position at the point in time associated with the sampling ray. Cook [COOK86] suggests computing a bounding box for the object's entire path of motion, so that the bounding-box test can be performed without the expense of moving the object.

A weighted distribution in some dimension can be simulated by applying unequal weights to evenly distributed samples. Figure 16.63(a) shows such a distribution. A more attractive alternative, however, is to use *importance sampling*, in which proportionately more sample points are located at positions of higher weight. This is accomplished by dividing the weighting filter into regions of equal area and assigning the same number of equally weighted sample points to each, as shown in Fig. 16.63(b). The amount of jitter

Samsung/Google, EX1027, p. 801

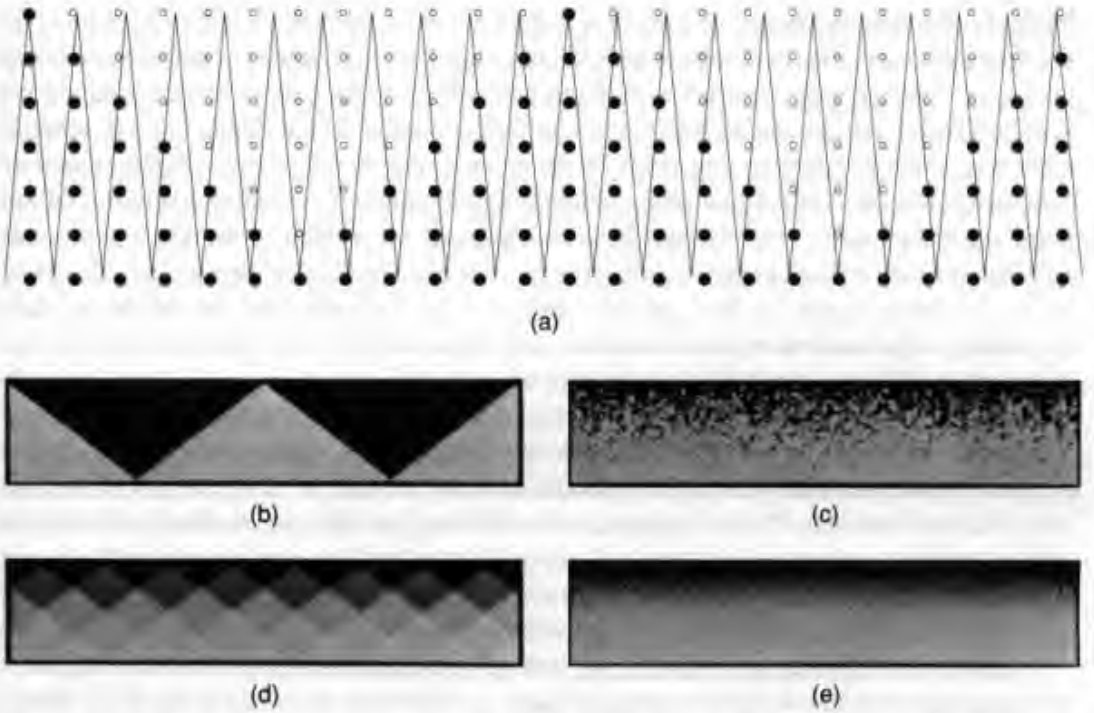


Fig. 16.62 Aliasing vs. noise. (a) A comb with regularly spaced triangles, each $(n + 1)/n$ pixels wide, sampled with one sample per pixel. o = samples that fall outside comb; • = samples that fall inside comb. (b) A comb with 200 triangles, each 1.01 pixels wide and 50 pixels high. 1 sample/pixel, regular grid. (c) 1 sample/pixel, jittered $\pm\frac{1}{2}$ pixel. (d) 16 samples/pixel, regular grid. (e) 16 samples/pixel, jittered $\pm\frac{1}{8}$ pixel. (Images (b)–(e) by Robert Cook, Lucasfilm Ltd.)

associated with a region also varies in proportion to its size. Color Plate III.16 was created using distributed ray tracing. It shows five billiard balls with motion blur and penumbrae cast by extended light sources. Note the blurred shadows and reflections of the four moving billiard balls.

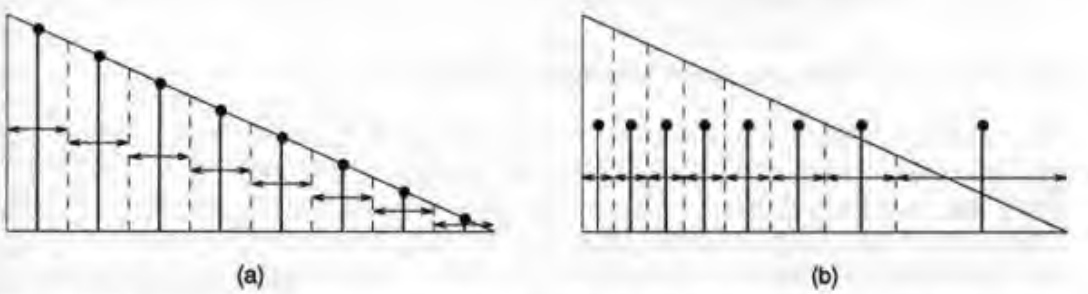


Fig. 16.63 Importance sampling is accomplished by partitioning the weighting function into regions of equal area. The horizontal axis is the dimension sampled; the vertical axis is the weighting. Dots show nominal position of samples; arrows show jitter range. (a) Evenly distributed, unequally weighted samples. (b) Importance sampling: unevenly distributed, equally weighted samples.

Samsung/Google, EX1027, p. 802

Path tracing and the integral equation method. Kajiya [KAJI86] has implemented an efficient variation on distributed ray tracing called *path tracing*. Rather than each ray being grown into a binary tree, exactly one reflection or refraction ray is fired at each intersection to form a linear path, along with one ray to each light source. The decision to shoot either a reflection or a refraction ray is guided by the desired distribution of the different kinds of rays for each pixel. Kajiya has also extended this algorithm to develop a solution to the rendering equation (Eq. 16.54), called the *integral equation method*, that takes into account all ways in which light can reach a point. He uses variance-reduction methods to calculate a random variable based on the specular, diffuse, and transmission coefficients at each intersection. The random variable is used to determine whether the single ray cast from the intersection will be a specular reflection, diffuse reflection, or refraction ray, and the ray's direction is then chosen by sampling. In addition, a shadow ray is cast to a point on a light source, also chosen using variance-reduction methods. Because diffuse rays are traced, this approach models diffuse interobject reflections, an effect that we shall discuss in Section 16.13. Color Plate III.17 was rendered with the integral equation method. All objects shown are gray, except for the floor and green glass balls. The gray objects reflect green light focused by the balls and reddish light diffusely reflected from the floor, phenomena not modeled by conventional ray tracing (or by path tracing).

Kajiya's two approaches evaluate the diffuse reflection separately at each pixel, even though diffuse reflection tends to change relatively little from one pixel to the next. Ward, Rubinstein, and Clear [WARD88] have supplemented a ray tracer with a recursive diffuse reflection stage in which rays are used to trace some number of diffuse bounces from a surface to others illuminating it. Rather than computing the diffuse reflection for each pixel separately, they instead cache all of the values computed. When a pixel is processed, they use an estimate of the illuminance gradient at each cached diffuse reflection value "nearby" the pixel's intersection point to estimate the error associated with using that value. If the error is considered acceptable, then a weighted average of these cached values is used to compute a new value for the pixel; otherwise, a new diffuse calculation is made by tracing rays that sample the hemisphere, and its value is cached. The cached values can then be reused in computing other views of the same scene. Color Plate III.18 shows a series of images rendered using this technique, with different numbers of diffuse bounces. Color Plate III.19 contrasts a photograph of a conference room with a rendered image.

16.12.5 Ray Tracing from the Light Sources

One serious problem with ray tracing is caused by tracing all rays from the eye. Shadow rays are cast only to direct sources of light that are treated separately by the algorithm. Therefore, the effects of indirect reflected and refracted light sources, such as mirrors and lenses, are not reproduced properly: Light rays bouncing off a mirror do not cast shadows, and the shadows of transparent objects do not evidence refraction, since shadow rays are cast in a straight line toward the light source.

It might seem that we would need only to run a conventional ray tracer "backward" from the light sources to the eye to achieve these effects. This concept has been called *backward ray tracing*, to indicate that it runs in the reverse direction from regular ray tracing, but it is also known as *forward ray tracing* to stress that it follows the actual path from the lights to the eye. We call it *ray tracing from the light sources* to avoid confusion!

Samsung/Google, EX1027, p. 803

Done naively, ray tracing from the light sources results in new problems, since an insufficient number of rays ever would strike the image plane, let alone pass through the focusing lens or pinhole. Instead, ray tracing from the light sources can be used to supplement the lighting information obtained by regular ray tracing. Heckbert and Hanrahan [HECK84] suggest an elaboration of their proposed beam-tracing shadow method (Section 16.12.3) to accomplish this. If a light's beam tree is traced recursively, successive levels of the tree below the first level represent indirectly illuminated polygon fragments. Adding these to the database as surface-detail polygons allows indirect specular illumination to be modeled.

Arvo [ARVO86] has implemented a ray tracer that uses a preprocessing step in which rays from each light source are sent into the environment. Each ray is assigned an initial quota of energy, some of which is deposited at each intersection it makes with a diffusely reflecting object. He compensates for the relative sparseness of ray intersections by mapping each surface to a regular rectangular grid of counters that accumulate the deposited energy. Each ray's contribution is bilinearly partitioned among the four counters that bound the grid box in which the ray hits. A conventional ray-tracing pass is then made, in which the first pass's interpolated contributions at each intersection are used, along with the intensities of the visible light sources, to compute the diffuse reflection. Unfortunately, if a light ray strikes an object on the invisible side of a silhouette edge as seen from the eye, the ray can affect the shading on the visible side. Note that both these approaches to ray tracing from the light sources use purely specular reflectivity geometry to propagate rays in both directions.

16.13 RADIOSITY METHODS

Although ray tracing does an excellent job of modeling specular reflection and dispersionless refractive transparency, it still makes use of a directionless ambient-lighting term to account for all other global lighting contributions. Approaches based on thermal-engineering models for the emission and reflection of radiation eliminate the need for the ambient-lighting term by providing a more accurate treatment of interobject reflections. First introduced by Goral, Torrance, Greenberg, and Battaile [GORA84] and by Nishita and Nakamae [NISH85a], these algorithms assume the conservation of light energy in a closed environment. All energy emitted or reflected by every surface is accounted for by its reflection from or absorption by other surfaces. The rate at which energy leaves a surface, called its *radiosity*, is the sum of the rates at which the surface emits energy and reflects or transmits it from that surface or other surfaces. Consequently, approaches that compute the radiosities of the surfaces in an environment have been named *radiosity methods*. Unlike conventional rendering algorithms, radiosity methods first determine all the light interactions in an environment in a view-independent way. Then, one or more views are rendered, with only the overhead of visible-surface determination and interpolative shading.

16.13.1 The Radiosity Equation

In the shading algorithms considered previously, light sources have always been treated separately from the surfaces they illuminate. In contrast, radiosity methods allow any surface to emit light; thus, all light sources are modeled inherently as having area. Imagine

Samsung/Google, EX1027, p. 804

breaking up the environment into a finite number n of discrete patches, each of which is assumed to be of finite size, emitting and reflecting light uniformly over its entire area. If we consider each patch to be an opaque Lambertian diffuse emitter and reflector, then, for surface i,

$$B_i = E_i + \rho_i \sum_{1 \le j \le n} B_j F_{j-i} \frac{A_j}{A_i}. \tag{16.58}$$

B_i and B_j are the radiosities of patches i and j, measured in energy/unit time/unit area (i.e., W / m^2). E_i is the rate at which light is emitted from patch i and has the same units as radiosity. ρ_i is patch i's reflectivity and is dimensionless. F_{j-i} is the dimensionless *form factor* or *configuration factor*, which specifies the fraction of energy leaving the entirety of patch j that arrives at the entirety of patch i, taking into account the shape and relative orientation of both patches and the presence of any obstructing patches. A_i and A_j are the areas of patches i and j.

Equation (16.58) states that the energy leaving a unit area of surface is the sum of the light emitted plus the light reflected. The reflected light is computed by scaling the sum of the incident light by the reflectivity. The incident light is in turn the sum of the light leaving the entirety of each patch in the environment scaled by the fraction of that light reaching a unit area of the receiving patch. $B_j F_{j-i}$ is the amount of light leaving a unit area of A_j that reaches all of A_i. Therefore, it is necessary to multiply by the area ratio A_j / A_i to determine the light leaving all of A_j that reaches a unit area of A_i.

Conveniently, a simple reciprocity relationship holds between form factors in diffuse environments,

$$A_i F_{i-j} = A_j F_{j-i}. \tag{16.59}$$

Thus, Eq. (16.58) can be simplified, yielding

$$B_i = E_i + \rho_i \sum_{1 \le j \le n} B_j F_{i-j}. \tag{16.60}$$

Rearranging terms,

$$B_i - \rho_i \sum_{1 \le j \le n} B_j F_{i-j} = E_i. \tag{16.61}$$

Therefore, the interaction of light among the patches in the environment can be stated as a set of simultaneous equations:

$$\begin{bmatrix} 1 - \rho_1 F_{1-1} & -\rho_1 F_{1-2} & \cdots & -\rho_1 F_{1-n} \\ -\rho_2 F_{2-1} & 1 - \rho_2 F_{2-2} & \cdots & -\rho_2 F_{2-n} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ -\rho_n F_{n-1} & -\rho_n F_{n-2} & \cdots & 1 - \rho_n F_{n-n} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \cdot \\ \cdot \\ \cdot \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \cdot \\ \cdot \\ \cdot \\ E_n \end{bmatrix}. \tag{16.62}$$

Note that a patch's contribution to its own reflected energy must be taken into account (e.g., it may be concave); so, in the general case, each term along the diagonal is not merely 1. Equation (16.62) must be solved for each band of wavelengths considered in the lighting model, since ρ_i and E_i are wavelength-dependent. The form factors, however, are

Samsung/Google, EX1027, p. 805

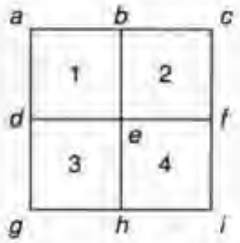


Fig. 16.64 Computing vertex radiosities from patch radiosities.

independent of wavelength and are solely a function of geometry, and thus do not need to be recomputed if the lighting or surface reflectivity changes.

Equation (16.62) may be solved using Gauss–Seidel iteration [PRES88], yielding a radiosity for each patch. The patches can then be rendered from any desired viewpoint with a conventional visible-surface algorithm; the set of radiosities computed for the wavelength bands of each patch are that patch's intensities. Instead of using faceted shading, we can compute vertex radiosities from the patch radiosities to allow intensity interpolation shading.

Cohen and Greenberg [COHE85] suggest the following approach for determining vertex radiosities. If a vertex is interior to a surface, it is assigned the average of the radiosities of the patches that share it. If it is on the edge, then the nearest interior vertex v is found. The radiosity of the edge vertex when averaged with B_v should be the average of the radiosities of the patches that share the edge vertex. Consider the patches in Fig. 16.64. The radiosity for interior vertex e is $B_e = (B_1 + B_2 + B_3 + B_4) / 4$. The radiosity for edge vertex b is computed by finding its nearest interior vertex, e, and noting that b is shared by patches 1 and 2. Thus, to determine B_b, we use the preceding definition; $(B_b + B_e) / 2 = (B_1 + B_2) / 2$. Solving for B_b, we get $B_b = B_1 + B_2 - B_e$. The interior vertex closest to a is also e, and a is part of patch 1 alone. Thus, since $(B_a + B_e) / 2 = B_1$, we get $B_a = 2B_1 - B_e$. Radiosities for the other vertices are computed similarly.

The first radiosity method was implemented by Goral et al. [GORA84], who used contour integrals to compute exact form factors for convex environments with no occluded surfaces, as shown in Color Plate III.20. Note the correct "color-bleeding" effects due to diffuse reflection between adjacent surfaces, visible in both the model and the rendered image: diffuse surfaces are tinged with the colors of other diffuse surfaces that they reflect. For radiosity methods to become practical, however, ways to compute form factors between occluded surfaces had first to be developed.

16.13.2 Computing Form Factors

Cohen and Greenberg [COHE85] adapted an image-precision visible-surface algorithm to approximate form factors for occluded surfaces efficiently. Consider the two patches shown in Fig. 16.65. The form factor from differential area dA_i to differential area dA_j is

$$dF_{di-dj} = \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij}\, dA_j. \tag{16.63}$$

Samsung/Google, EX1027, p. 806

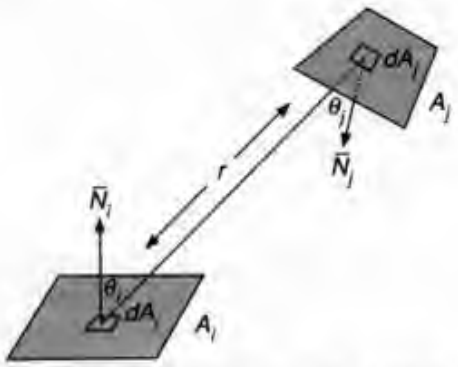


Fig. 16.65 Computing the form factor between a patch and a differential area.

For the ray between differential areas dA_i and dA_j in Fig. 16.65, θ_i is the angle that the ray makes with A_i's normal, θ_j is the angle that it makes with A_j's normal, and r is the ray's length. H_{ij} is either 1 or 0, depending on whether or not dA_j is visible from dA_i. To determine F_{di-j}, the form factor from differential area dA_i to finite area A_j, we need to integrate over the area of patch j. Thus,

$$F_{di-j} = \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij}\, dA_j. \tag{16.64}$$

Finally, the form factor from A_i to A_j is the area average of Eq. (16.64) over patch i:

$$F_{i-j} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij}\, dA_j\, dA_i. \tag{16.65}$$

If we assume that the center point on a patch typifies the patch's other points, then F_{i-j} can be approximated by F_{di-j} computed for dA_i at patch i's center.

Nusselt has shown [SIEG81] that computing F_{di-j} is equivalent to projecting those parts of A_j that are visible from dA_i onto a unit hemisphere centered about dA_i, projecting this projected area orthographically down onto the hemisphere's unit circle base, and dividing by the area of the circle (Fig. 16.66). Projecting onto the unit hemisphere accounts for $\cos\theta_j / r^2$ in Eq. (16.64), projecting down onto the base corresponds to a multiplication by $\cos\theta_i$, and dividing by the area of the unit circle accounts for the π in the denominator.

Rather than analytically projecting each A_j onto a hemisphere, Cohen and Greenberg developed an efficient image-precision algorithm that projects onto the upper half of a cube centered about dA_i, with the cube's top parallel to the surface (Fig. 16.67). Each face of this *hemicube* is divided into a number of equal-sized square cells. (Resolutions used in pictures included in this book range from 50 by 50 to several hundred on a face.) All the other patches are clipped to the view-volume frusta defined by the center of the cube and each of its upper five faces, and then each of the clipped patches is projected onto the appropriate face of the hemicube. An item-buffer algorithm (Section 16.12.1) is used that records the

Samsung/Google, EX1027, p. 807

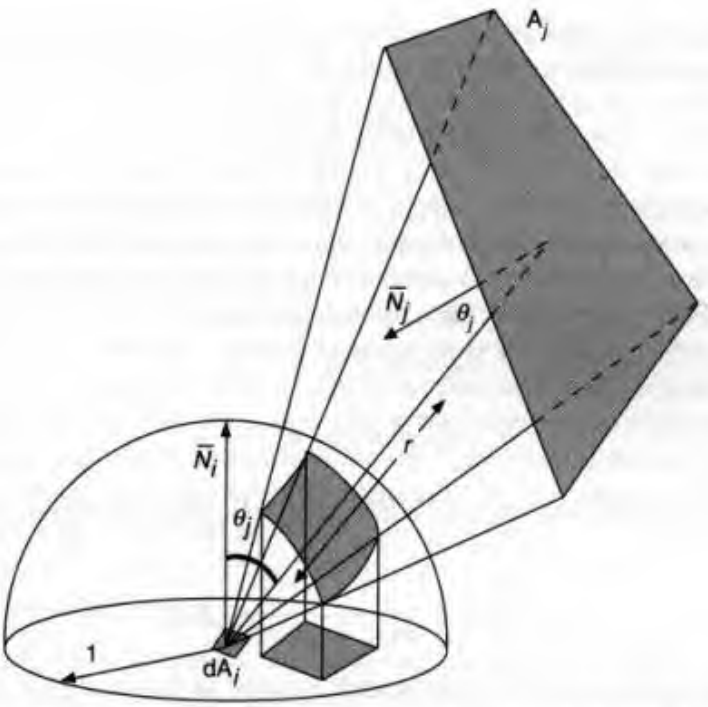


Fig. 16.66 Determining the form factor between a differential area and a patch using Nusselt's method. The ratio of the area projected onto the hemisphere's base to the area of the entire base is the form factor. (After [SIEG81].)

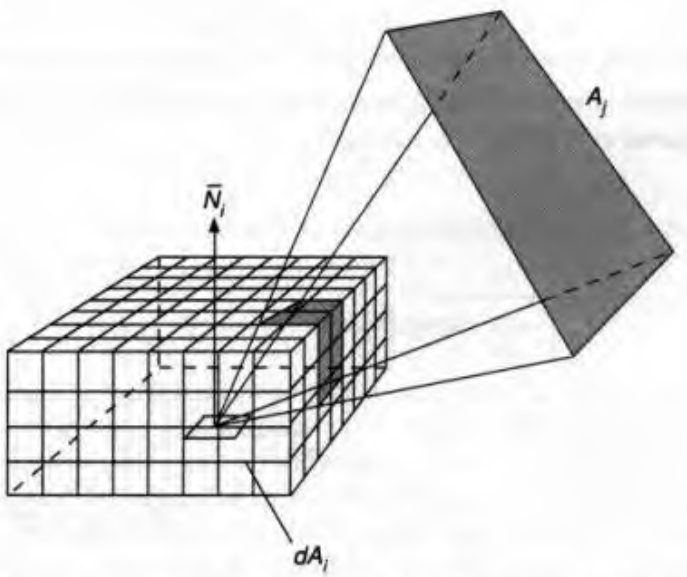


Fig. 16.67 The hemicube is the upper half of a cube centered about the patch. (After [COHE85].)

Samsung/Google, EX1027, p. 808

identity of the closest intersecting patch at each cell. Each hemicube cell p is associated with a precomputed delta form factor value,

$$\Delta F_p = \frac{\cos\theta_i \cos\theta_p}{\pi r^2}\Delta A, \tag{16.66}$$

where θ_p is the angle between the cell p's surface normal and the vector between dA_i and p, r is this vector's length, and ΔA is the area of a cell, as shown in Fig. 16.68. Assume that the hemicube has its own (x, y, z) coordinate system, with the origin at the center of the bottom face. For the top face in Fig. 16.68(a), we have

$$r = \sqrt{x_p^2 + y_p^2 + 1}, \tag{16.67}$$

$$\cos\theta_i = \cos\theta_p = \frac{1}{r},$$

where x_p and y_p are the coordinates of a hemicube cell. Thus, for the top face, Eq. (16.66) simplifies to

$$\Delta F_p = \frac{1}{\pi(x_p^2 + y_p^2 + 1)^2}\Delta A. \tag{16.68}$$

For a side face perpendicular to the hemicube's x axis, as shown in Fig. 16.68(b), we have

$$r = \sqrt{y_p^2 + z_p^2 + 1}, \tag{16.69}$$

$$\cos\theta_i = \frac{z_p}{r}, \qquad \cos\theta_p = \frac{1}{r}.$$

Here, Eq. (16.66) simplifies to

$$\Delta F_p = \frac{z_p}{\pi(y_p^2 + z_p^2 + 1)^2}\Delta A. \tag{16.70}$$

Because of symmetry, the values of ΔF_p need to be computed for only one-eighth of the top face and one-quarter of a single side half face.

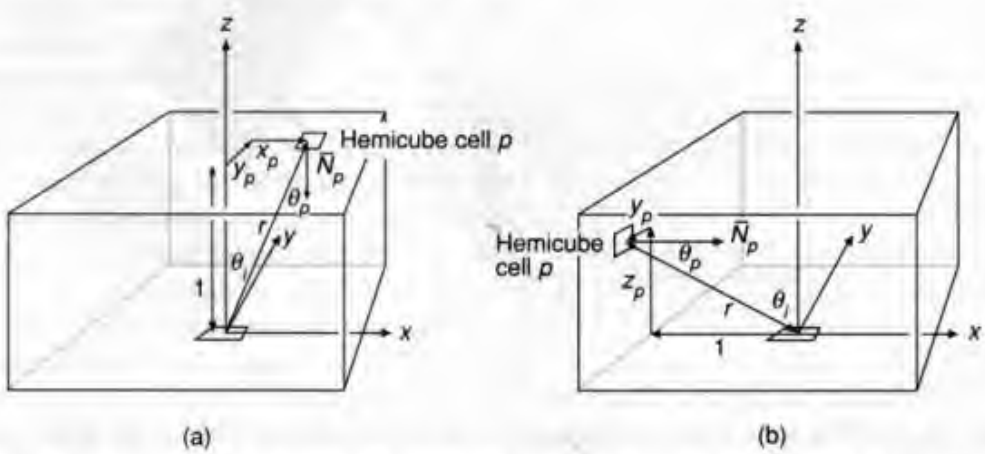


Fig. 16.68 Delta form factors. (a) The top face. (b) A side face. (After [COHE85].)

Samsung/Google, EX1027, p. 809

We can approximate F_{dA_i-j} for any patch j by summing the values of ΔF_p associated with each cell p in A_j's hemicube projections. (Note that the values of ΔF_p for all the hemicube's cells sum to 1.) Assuming that the distance between the patches is large relative to the size of the patches, these values for F_{dA_i-j} may be used as the values of F_{i-j} in Eq. (16.62) to compute the patch radiosities. Color Plate III.21(a–b) was made with the hemicube algorithm. Because much of the computation performed using the hemicube involves computing item buffers, it can take advantage of existing z-buffer hardware. On the other hand, because it uses image-precision operations, the hemicube is prone to aliasing.

Nishita and Nakamae [NISH85a] have adopted a different approach to computing form factors in occluded environments by incorporating their shadow algorithm for area light sources (Section 16.8) into a radiosity algorithm that was used to make Color Plate III.22(a–b). Color Plate III.22(c) [NISH86] adds to this a model of sky light, approximated by a large hemispherical source of diffuse light. The hemisphere is divided into bands that are transversely uniform and longitudinally nonuniform. As with other luminous surfaces, the effects of occluding objects are modeled.

16.13.3 Substructuring

The finer the patch parametrization, the better the results, at the expense of increased computation time for n^2 form factors. To prevent this square-law increase in the number of form factors, Cohen, Greenberg, Immel, and Brock [COHE86] adaptively subdivide patches into subpatches at places in the patch mesh at which a high radiosity gradient is found. The subpatches created by this *substructuring* process are not treated like full-fledged patches. Whenever a patch i is subdivided into subpatches, the form factors F_{s-j} from each subpatch s to each patch j are computed using the hemicube technique, but form factors from the patches to the subpatches are not computed. After a patch has been broken into subpatches, however, the previously calculated values of each form factor from the patch to other patches are replaced by the more accurate area-weighted average of the form factors from its m subpatches:

$$F_{i-j} = \frac{1}{A_i} \sum_{1 \le s \le m} F_{s-j} A_s. \tag{16.71}$$

After patch radiosities are calculated as described before, the radiosity of each subpatch s of patch i can be computed as

$$B_s = E_i + \rho_i \sum_{1 \le j \le n} B_j F_{s-j}. \tag{16.72}$$

The algorithm iterates, adaptively subdividing subpatches at places of high radiosity gradient, until the differences reach an acceptable level. The final subpatch radiosities are then used to determine the vertex radiosities. Color Plate III.21(b), made using a nonadaptive version of the algorithm in which subdivision is specified by the user, shows that the same image takes substantially less time to compute when patches are divided into one level of subpatches, than when an equivalent number of patches are used. The adaptive version of the algorithm is initialized with a "first guess" subdivision specified by the user. Color Plate III.21(c) was created by adaptively subdividing the subpatches of Color Plate III.21(b). Note the improved shadow resolution about the table's legs.

Samsung/Google, EX1027, p. 810

Substructuring allows subpatch radiosities to be determined without changing the size of the matrix to be solved in Eq. (16.62). Note that a subpatch's contribution to other patches is still approximated coarsely by its patch's radiosity, but this is a second-order effect in diffuse environments. In a similar fashion, texture mapping can be implemented by computing a single average reflectivity value for a texture-mapped patch that is used for the radiosity computations [COHE86]. When each pixel in the texture-mapped surface is finally rendered, its shade is scaled by the ratio of the texture-map reflectivity value computed for the pixel and the average reflectivity used for the patch.

16.13.4 Progressive Refinement

Given the high costs of executing the radiosity algorithm described thus far, it makes sense to ask whether it is possible to approximate the algorithm's results incrementally. Can we produce a useful, although perhaps inaccurate, image early on, which can be successively refined to greater accuracy as more time is allocated? The radiosity approach described in the previous sections will not let us do this for two reasons. First, an entire Gauss–Seidel iteration must take place before an estimate of the patch radiosities becomes available. Second, form factors are calculated between all patches at the start and must be stored throughout the computation, requiring $O(n^2)$ time and space. Cohen, Chen, Wallace, and Greenberg [COHE88] have developed a progressive-refinement radiosity algorithm that addresses both of these issues.

Consider the approach described thus far. Evaluating the ith row of Eq. (16.62) provides an estimate of patch i's radiosity, B_i, expressed in Eq. (16.60), based on the estimates of the other patch radiosities. Each term of the summation in Eq. (16.60) represents patch j's effect on the radiosity of patch i:

$$B_i \text{ due to } B_j = \rho_i B_j F_{i-j}, \qquad \text{for all } j. \tag{16.73}$$

Thus, this approach *gathers* the light from the rest of the environment. In contrast, the progressive-refinement approach *shoots* the radiosity from a patch into the environment. A straightforward way to do this is to modify Eq. (16.73) to yield

$$B_j \text{ due to } B_i = \rho_j B_i F_{j-i}, \qquad \text{for all } j. \tag{16.74}$$

Given an estimate of B_i, the contribution of patch i to the rest of the environment can be determined by evaluating Eq. (16.74) for each patch j. Unfortunately, this will require knowing F_{j-i} for each j, each value of which is determined with a separate hemicube. This imposes the same overwhelmingly large space–time overhead as does the original approach. By using the reciprocity relationship of Eq. (16.59), however, we can rewrite Eq. (16.74) as

$$B_j \text{ due to } B_i = \rho_j B_i F_{i-j} \frac{A_i}{A_j}, \qquad \text{for all } j. \tag{16.75}$$

Evaluating this equation for each j requires only the form factors calculated using a single hemicube centered about patch i. If the form factors from patch i can be computed quickly (e.g., by using z-buffer hardware), then they can be discarded as soon as the radiosities shot

Samsung/Google, EX1027, p. 811

from patch i have been computed. Thus, only a single hemicube and its form factors need to be computed and stored at a time.

As soon as a patch's radiosity has been shot, another patch is selected. A patch may be selected to shoot again after new light has been shot to it from other patches. Therefore, it is not patch i's total estimated radiosity that is shot, but rather ΔB_i, the amount of radiosity that patch i has received since the last time that it shot. The algorithm iterates until the desired tolerance is reached. Rather than choose patches in random order, it makes sense to select the patch that will make the most difference. This is the patch that has the most energy left to radiate. Since radiosity is measured per unit area, a patch i is picked for which $\Delta B_i A_i$ is the greatest. Initially, $B_i = \Delta B_i = E_i$ for all patches, which is nonzero only for light sources. The pseudocode for a single iteration is shown in Fig. 16.69.

Each execution of the pseudocode in Fig. 16.69 will cause another patch to shoot its unshot radiosity into the environment. Thus, the only surfaces that are illuminated after the first execution are those that are light sources and those that are illuminated directly by the first patch whose radiosity is shot. If a new picture is rendered at the end of each execution, the first picture will be relatively dark, and those following will get progressively brighter. To make the earlier pictures more useful, we can add an ambient term to the radiosities. With each additional pass through the loop, the ambient term will be decreased, until it disappears.

One way to estimate the ambient term uses a weighted sum of the unshot patch radiosities. First, an average diffuse reflectivity for the environment, ρ_{avg} is computed as a weighted sum of the patch diffuse reflectivities,

$$\rho_{avg} = \sum_{1 \le i \le n} \rho_i A_i \Big/ \sum_{1 \le i \le n} A_i. \tag{16.76}$$

This equation is used to compute an overall reflection factor R, intended to take into account the different reflected paths through which energy can travel from one patch to another,

$$R = 1 + \rho_{avg} + \rho_{avg}^2 + \rho_{avg}^3 + \ldots = \frac{1}{1 - \rho_{avg}}. \tag{16.77}$$

```
select patch i;

calculate F_{i-j} for each patch j;

for (each patch j) {
    ΔRadiosity = ρ_j ΔB_i F_{i-j} A_i / A_j;
    ΔB_j += ΔRadiosity;
    B_j += ΔRadiosity;
}

ΔB_i = 0;
```

Fig. 16.69 Pseudocode for shooting radiosity from a patch.

Samsung/Google, EX1027, p. 812

```
for (each patch i) {
    ΔB_i = E_i;
    for (each subpatch s in i)
        B_s = E_i;
}

AreaSum = Σ_{1≤i≤n} A_i;

Ambient = R Σ_{1≤i≤n} (ΔB_i A_i)/AreaSum;

while (not converged) {
    select patch i with greatest ΔB_i A_i;
    determine F_{i-s} for all subpatches s in all patches;

    /* ΔEnergy is initialized to the total energy shot. */
    ΔEnergy = ΔB_i A_i;

    /* Shoot radiosity from patch i. */
    for (each patch j seen by i) {
        OldΔB = ΔB_j;
        for (each subpatch s in j seen by i) {
            ΔRadiosity = ρ_j ΔB_i F_{i-s} A_i / A_s;
            B_s += ΔRadiosity;
            ΔB_j += ΔRadiosity A_s / A_j;
        }
        /* Decrement ΔEnergy by total energy gained by patch j. */
        ΔEnergy -= (ΔB_j - OldΔB) A_j;
    }

    determine vertex radiosities from subpatch radiosities, using
        B_s + ρ_j Ambient as radiosity of subpatch s of patch j;
    if (radiosity gradient between adjacent vertices is too high)
        subdivide offending subpatches and reshoot from patch i to them;
    ΔB_i = 0;

    perform view-dependent visible-surface determination and shading;

    /* Use ΔEnergy (energy absorbed by patches hit) to determine new value of Ambient. */
    Ambient -= R ΔEnergy / AreaSum;
} /* while */
```

Fig. 16.70 Pseudocode for progressive-refinement radiosity method with ambient light and substructuring.

Samsung/Google, EX1027, p. 813

Each patch's unshot radiosity is weighted by the ratio of the patch's area to the environment's area, providing an approximation to the form factor from an arbitrary differential area to that patch. Thus, the estimate of the ambient term accounting for unshot radiosity is

$$Ambient = R \sum_{1 \le i \le n} (\Delta B_i A_i) / \sum_{1 \le i \le n} A_i. \tag{16.78}$$

This ambient term is used to augment the patch's radiosity for display purposes only, yielding

$$B'_i = B_i + \rho_i Ambient. \tag{16.79}$$

Figure 16.70 shows the pseudocode for the entire algorithm. Substructuring is provided by shooting radiosity from patches to subpatches to determine subpatch radiosities. Thus, hemicubes are created for patches, but not for subpatches. Adaptive subdivision is accomplished by subdividing a patch further when the radiosity gradient between adjacent subpatch vertices is found to be too high. Color Plate III.23, which is rendered using an ambient term, depicts stages in the creation of an image after 1, 2, 24, and 100 iterations.

16.13.5 Computing More Accurate Form Factors

Although the use of fast z-buffer hardware makes the hemicube an efficient algorithm, the technique has a number of failings [BAUM89; WALL89]:

- Recall that the identity of only one patch is stored per hemicube pixel. Therefore, a grid of patches may alias when projected onto a side of the hemicube, just as they would when processed with a z-buffer algorithm. This can show up as a regular pattern of patches that are not represented in the hemicube. Furthermore, a patch that is small when projected on the hemicube may be large when projected on the image plane.
- Use of the hemicube assumes that the center point of a patch is representative of the patch's visibility to other patches. If this assumption is shown to be untrue, the surface can be broken up into subpatches, but there is only a single subdivision granularity for the patch; the same patch cannot be subdivided to different levels for different patches that it views.
- Patches must be far from each other for the hemicube approach to be correct. This is a serious problem if two patches are adjacent; since all calculations are done from the center of the hemicube, the form factor will be underestimated, because the calculations do not take into account the proximity of the adjacent parts of the patches.

A progressive radiosity approach developed by Wallace, Elmquist, and Haines [WALL89] uses ray tracing to evaluate form factors, instead of the hemicube. When a source patch is to shoot its radiosity, rays are fired from each vertex in the scene to the source to compute the form factor from the source to the vertex. This is accomplished by decomposing the source patch into a number of small finite subareas, each of which is the target of a ray shot from a vertex. If the ray is not occluded, then the target is visible, and the form factor between the differential vertex and the finite area target is computed, using an analytic expression, based on some simplifying geometric assumptions. If desired, the

Samsung/Google, EX1027, p. 814

ray intersection calculations can be performed with a resolution-independent true curved-surface database, so that the time for an individual ray test does not depend on the number of polygons. The form factor between the vertex and the entire source is computed as an area-weighted average of the form factors between each subarea and the vertex, and the result is used to compute the contribution of the source to the vertex. This approach has a number of advantages. Radiosities are computed at the vertices themselves, where they are ultimately needed for shading. Vertex normals can be used, allowing polygonal meshes that approximate curved surfaces. Nonphysical point light sources can be handled by tracing a single ray to the light source and using its illumination equation to determine the irradiance at each vertex. The number of areas into which a source is decomposed and whether rays are actually fired (i.e., if shadow testing is to be performed) can all be determined individually for each vertex. Color Plates III.24 and III.25 were created using this algorithm.

A contrasting approach to solving inaccuracies caused by the hemicube is taken by Baum, Rushmeier, and Winget [BAUM89]. They recognize that the hemicube form factors often are accurate; therefore, they have developed error-analysis tests to choose, for each patch, when to use the hemicube, when to subdivide the patch further, and when to use a more expensive, but more accurate, analytic technique for computing form factors.

16.13.6 Specular Reflection

The radiosity methods described so far treat only diffuse reflection. Therefore, all of a patch's radiosity may be treated uniformly when it is dispersed to other patches: The radiosity leaving a patch in any direction is influenced by the patch's total radiosity, not by the directions from which its incoming energy was acquired. Immel, Cohen, and Greenberg [IMME86] extended the radiosity method to model specular reflection. Rather than compute a single radiosity value for each patch, they partition the hemisphere over the patch into a finite set of solid angles, each of which establishes a direction for incoming or outgoing energy. Given the patch's bidirectional reflectivity (Section 16.7), they compute the outgoing radiosity in each direction in terms of its emittance in that direction and the incident light from each of the set of directions, weighting each direction's contribution accordingly. Finally, they render a picture from intensities that are determined at each vertex by using the direction from the vertex to the eye to interpolate among the closest directional radiosities. Although the results shown in Color Plate III.26 are promising, the approach has a tremendous overhead in both time and space, which will only increase if highly specular surfaces are modeled. One solution is to combine a radiosity method with ray tracing.

16.13.7 Combining Radiosity and Ray Tracing

Consider the tradeoffs between radiosity methods and ray tracing. Radiosity methods are well suited to diffuse reflection because a diffuse surface's bidirectional reflectivity is constant in all outgoing directions. Thus, all radiosities computed are view-independent. On the other hand, the pure radiosity method for specular surfaces described previously is not practical, because specular reflection from a surface is highly dependent on the angle with which an observer (or another surface) views the surface. Therefore, much extra

Samsung/Google, EX1027, p. 815

information must be computed, because no information about the desired view is provided. In addition, this directional information is discretized and must be interpolated to accommodate a specific view. Not only does the interpolation make it impossible to model sharp reflections, but also the sampling performed by the discretization can result in aliasing.

In contrast, ray tracing calculates specular reflections well, since the eyepoint is known in advance. Although conventional ray tracing does not model global diffuse phenomena, some of the approaches discussed in Section 16.12.4 do. Correctly solving for the diffuse reflection from a piece of surface requires that all the surfaces with which a surface exchanges energy be taken into account; in short, it requires a radiosity method.

It makes sense to combine ray tracing and radiosity to take advantage of ray tracing's ability to model specular phenomena and of the radiosity method's ability to model diffuse interactions. Unfortunately, simply summing the pixel values computed by a diffuse radiosity method and a specular ray tracer will not suffice. For example, the diffuse radiosity method will fail to take into account the extra illumination falling on a diffuse surface from a specular surface. It is necessary to account for transfer from diffuse to diffuse, diffuse to specular, specular to diffuse, and specular to specular reflection.

Wallace, Cohen, and Greenberg [WALL87] describe a two-pass approach that combines a view-independent radiosity method, executed in the first pass, with a view-dependent ray-tracing approach, executed in the second pass. As mentioned previously, the first pass must take into account specular, as well as diffuse, reflection. If only perfect, mirrorlike specular reflection is allowed, this can be supported by reflecting each patch about the plane of a specular surface [RUSH86]. Each specular patch is thus treated as a window onto a "mirror world." The form factor from a patch to one of these mirror reflections accounts for the specular reflection from the patch that is doing the mirroring.

In the second view-dependent pass, a *reflection frustum* is erected at each point on a surface that corresponds to a pixel in the image. As shown in Fig. 16.71, the reflection frustum consists of a little z-buffer, positioned perpendicular to the reflection direction, and covering the small incoming solid angle that is most significant for the surface's

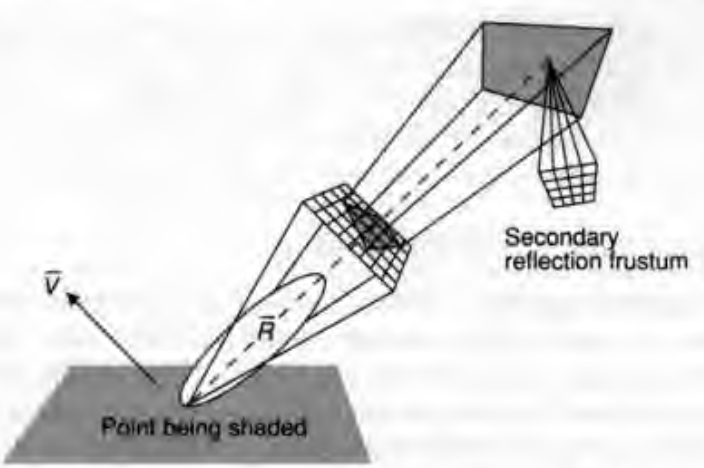


Fig. 16.71 The reflection frustum. (After [WALL87].)

Samsung/Google, EX1027, p. 816

bidirectional reflectivity. The patches are z-buffered onto the frustum, using Gouraud shading to interpolate the patches' first-pass diffuse intensities across their projections. A ray is traced recursively through each pixel on the frustum that sees a specular surface, spawning a new reflection frustum at each intersection. The values computed for each frustum pixel are then weighted to model the surface's ρ_s. The second pass thus uses specular transport to combine radiosities determined during the first pass. Transparency can be accommodated by erecting a transmission frustrum in the direction of refraction. The image on the cover of this book (Color Plate I.9) was created using this algorithm.

The mirror-world approach used in the first pass handles only perfect specular reflection and results in a proliferation of form factors. Shao, Peng, and Liang [SHAO88] have implemented a two-pass approach that allows Phong-like bidirectional reflectance functions in the first pass, without the need to duplicate patches.

Sillion and Puech [SILL89] extend the two-pass technique to calculate *extended form factors* in the first pass that model any number of specular reflections or refractions. Rather than proliferating mirror-reflection form factors, they instead use recursive ray tracing to compute the form factors, as well as in the view-dependent second pass. Color Plate III.27 demonstrates why the diffuse first pass must take specular reflection into account. Color Plate III.27(a) shows the results of a conventional diffuse radiosity approach. (The part of the table near the mirror is lit by light diffusely reflected from the inside of the tall lamp.) The conventional diffuse first pass was augmented with a pure ray-tracing second pass to produce Color Plate III.27(b), which includes a specular reflection from the mirror. In contrast, Color Plate III.27(c) shows the results of Sillion and Puech's two-pass approach. It shares the same ray-tracing second pass as Color Plate III.27(b), but uses extended form factors in the first pass. Each surface acts as a diffuse illuminator in the first pass, but the use of the extended form factors means that the diffusely emitted energy takes specular interreflection into account. Note the light specularly reflected from the mirror onto the table and the back of the vase during the first pass. Color Plate III.28 is a more complex example that includes a reflecting sphere.

16.14 THE RENDERING PIPELINE

Now that we have seen a variety of different ways to perform visible-surface determination, illumination, and shading, we shall review how these processes fit into the standard graphics pipeline introduced in Chapter 7 and depicted in Fig. 7.26. For simplicity, we assume polygonal environments, unless otherwise specified. Chapter 18 provides a more detailed discussion of how some of these pipelines may be implemented in hardware.

16.14.1 Local Illumination Pipelines

z-buffer and Gouraud shading. Perhaps the most straightforward modification to the pipeline occurs in a system that uses the z-buffer visible-surface algorithm to render Gouraud-shaded polygons, as shown in Fig. 16.72. The z-buffer algorithm has the advantage that primitives may be presented to it in any order. Therefore, as before, primitives are obtained by traversing the database, and are transformed by the modeling transformation into the WC system.

Primitives may have associated surface normals that were specified when the model was built. Since the lighting step will require the use of surface normals, it is important to

Samsung/Google, EX1027, p. 817

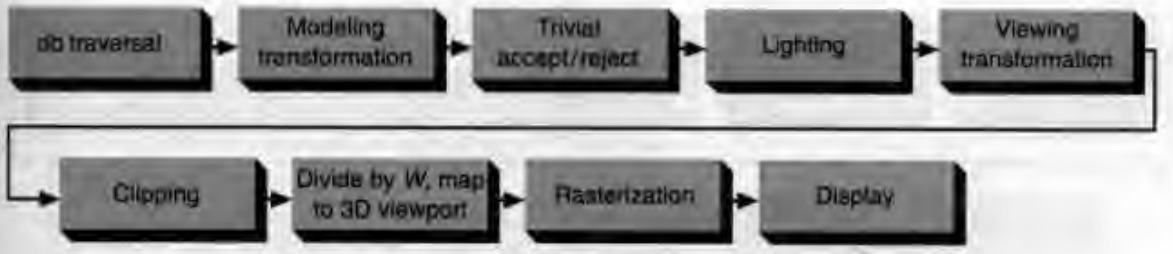


Fig. 16.72 Rendering pipeline for *z*-buffer and Gouraud shading.

remember that normals must be transformed correctly, using the methods discussed in the Appendix. Furthermore, we cannot just ignore stored normals and attempt to recompute new ones later using the correctly transformed vertices. The normals defined with the objects may represent the true surface geometry, or may specify user-defined surface blending effects, rather than just being the averages of the normals of shared faces in the polygonal mesh approximation.

Our next step is to cull primitives that fall entirely outside of the window and to perform back-face culling. This trivial-reject phase is typically performed now because we want to eliminate unneeded processing in the lighting step that follows. Now, because we are using Gouraud shading, the illumination equation is evaluated at each vertex. This operation must be performed in the WC system (or in any coordinate system isometric to it), before the viewing transformation (which may include skew and perspective transformations), to preserve the correct angle and distance from each light to the surface. If vertex normals were not provided with the object, they may be computed immediately before lighting the vertices. Culling and lighting are often performed in a lighting coordinate system that is a rigid body transformation of WC (e.g., VRC when the view orientation matrix is created with the standard PHIGS utilities).

Next objects are transformed to NPC by the viewing transformation, and clipped to the view volume. Division by W is performed, and objects are mapped to the viewport. If an object is partially clipped, correct intensity values must be calculated for vertices created during clipping. At this point, the clipped primitive is submitted to the z-buffer algorithm, which performs rasterization, interleaving scan conversion with the interpolation needed to compute the z value and color-intensity values for each pixel. If a pixel is determined to be visible, its color-intensity values may be further modified by depth cueing (Eq. 16.11), not shown here.

Although this pipeline may seem straightforward, there are many new issues that must be dealt with to provide an efficient and correct implementation. For example, consider the problems raised by handling curved surfaces, such as B-spline patches, which must be tessellated. Tessellation should occur after transformation into a coordinate system in which screen size can be determined. This enables tessellation size to be determined adaptively, and limits the amount of data that is transformed. On the other hand, tessellated primitives must be lit in a coordinate system isometric to world coordinates. Abi-Ezzi [ABIE89] addresses these issues, proposing a more efficient, yet more complex, formulation of the pipeline that incorporates feedback loops. This new pipeline uses a lighting coordinate system that is an isometric (i.e., rigid or Euclidean) transformation of WC, yet is computationally close to DC to allow tessellation decisions to be made efficiently.

Samsung/Google, EX1027, p. 818

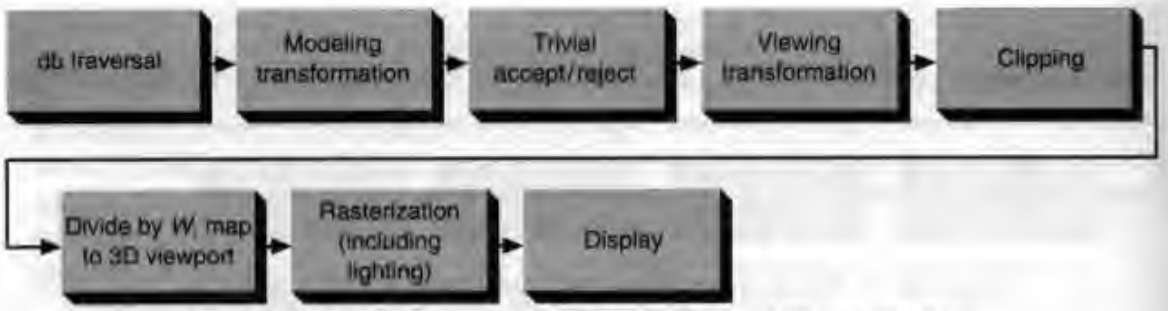


Fig. 16.73 Rendering pipeline for z-buffer and Phong shading.

z-buffer and Phong shading. This simple pipeline must be modified if we wish to accommodate Phong shading, as shown in Fig. 16.73. Because Phong shading interpolates surface normals, rather than intensities, the vertices cannot be lit early in the pipeline. Instead, each object must be clipped (with properly interpolated normals created for each newly created vertex), transformed by the viewing transformation, and passed to the z-buffer algorithm. Finally, lighting is performed with the interpolated surface normals that are derived during scan conversion. Thus, each point and its normal must be backmapped into a coordinate system that is isometric to WC to evaluate the illumination equation.

List-priority algorithm and Phong shading. When a list-priority algorithm is used, primitives obtained from traversal and processed by the modeling transformation are inserted in a separate database, such as a BSP tree, as part of preliminary visible-surface determination. Figure 16.74 presents the pipeline for the BSP tree algorithm, whose preliminary visible-surface determination is view-independent. As we noted in Chapter 7, the application program and the graphics package may each keep separate databases. Here, we see that rendering can require yet another database. Since, in this case, polygons are split, correct shading information must be determined for the newly created vertices. The rendering database can now be traversed to return primitives in a correct, back-to-front order. The overhead of building this database can, of course, be applied toward the creation of multiple pictures. Therefore, we have shown it as a separate pipeline whose output is a new database. Primitives extracted from the rendering database are clipped and normalized, and are presented to the remaining stages of the pipeline. These stages are structured much like those used for the z-buffer pipeline, except that the only visible-surface process they need to perform is to guarantee that each polygon will correctly overwrite any

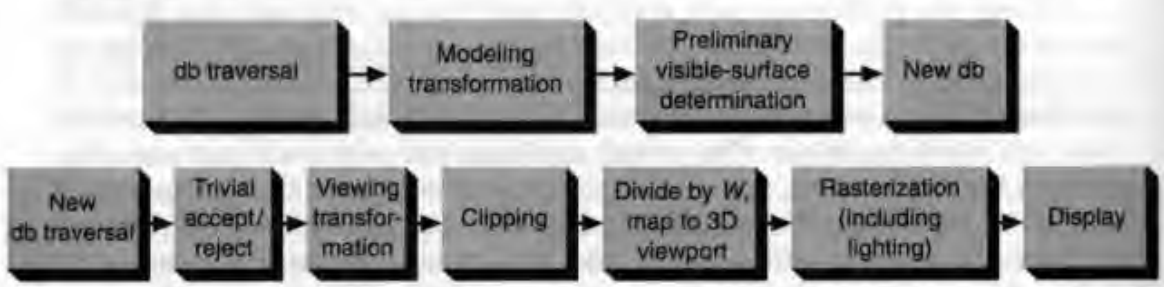


Fig. 16.74 Rendering pipeline for list-priority algorithm and Phong shading.

Samsung/Google, EX1027, p. 819

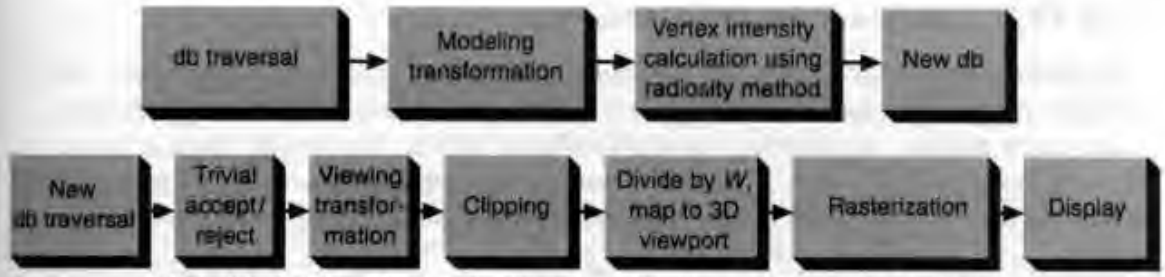


Fig. 16.75 Rendering pipeline for radiosity and Gouraud shading.

previously scan-converted polygon that it intersects. Even this simple overwrite capability is not needed if we instead use an object-precision algorithm that generates a list of fully visible primitives, such as the Weiler–Atherton algorithm.

16.14.2 Global Illumination Pipelines

Thus far, we have ignored global illumination. As we have noted before, incorporating global illumination effects requires information about the geometric relationships between the object being rendered and the other objects in the world. One approach, of which we have seen many examples, is to calculate needed information from a specific viewpoint in advance of scan conversion and to store it in tables (e.g., reflection maps and shadow maps). This eliminates the need to access the full db representation of other objects while processing the current object. In the case of shadows, which depend only on the position of the light source, and not on that of the viewer, preprocessing the environment to add surface-detail polygon shadows is another way to allow the use of an otherwise conventional pipeline.

Radiosity. The diffuse radiosity algorithms offer an interesting example of how to take advantage of the conventional pipeline to achieve global-illumination effects. These algorithms process objects and assign to them a set of view-independent vertex intensities. These objects may then be presented to a modified version of the pipeline for z-buffer and Gouraud shading, depicted in Fig. 16.75, that eliminates the lighting stage.

Ray tracing. Finally, we consider ray tracing, whose pipeline, shown in Fig. 16.76, is the simplest because those objects that are visible at each pixel and their illumination are determined entirely in WC. Once objects have been obtained from the database and transformed by the modeling transformation, they are loaded into the ray tracer's WC database, which is typically implemented using the techniques of Sections 15.10.2 and 16.12.1, to support efficient ray intersection calculations.

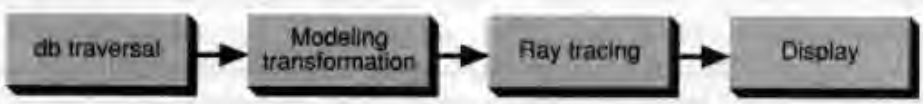


Fig. 16.76 Rendering pipeline for ray tracing.

Samsung/Google, EX1027, p. 820

16.14.3 Designing Flexible Renderers

As we have seen, a wide variety of illumination and shading models has been created. The choice of which to use may be based on concerns as diverse as increasing efficiency, increasing realism, or obtaining visually interesting effects. Simply put, there is no one model that pleases all users. Therefore, several design approaches have been suggested to increase the ease with which illumination and shading algorithms may be implemented and used.

Modularization. A straightforward approach is to modularize the illumination and shading model in a part of the rendering system that is often known as its *shader*. Whitted and Weimer [WHIT82] showed that, by establishing a standard mechanism for passing parameters to shaders, different shaders can be used in the same system; the decision about which shader to call can even be made at run time based on some attribute of the object. Their system performs scan conversion using a scan-line algorithm, and accumulates results as a linked list of spans for each line. Each span contains information about a set of values associated with its endpoints, including their x and z values, and additional information such as normal components, and intensities. The shader interpolates specified values across each span. (Since it typically uses the interpolated z values to perform visible-surface determination with a scan-line z-buffer, the term *shader* is being used quite loosely.)

The Doré graphics system [ARDE89] is designed to offer the programmer additional flexibility. It provides a standard way of expressing a scene database in terms of a set of objects that have methods for performing operations such as rendering, picking, or computing a bounding volume. The display list and its traversal functions form a common core that is intended to make it easy to interface to different rendering systems. A programmer can use the standard set of objects, methods, and attributes, or can design her own using the framework.

Special languages. In contrast to providing extensibility at the level of the programming language in which the system is built, it is possible to design special languages that are better suited to specific graphics tasks. Cook [COOK84a] has designed a special-purpose language in which a shader is built as a tree expression called a *shade tree*. A shade tree is a tree of nodes, each of which takes parameters from its children and produces parameters for its parent. The parameters are the terms of the illumination equation, such as the specular coefficient, or the surface normal. Some nodes, such as *diffuse*, *specular*, or *square root*, are built into the language with which shade trees are specified. Others can be defined by the user and dynamically loaded when needed. All nodes can access information about the lights. Figure 16.77 shows a shade tree for a description of copper. A shade tree thus describes a particular shading process and is associated with one or more objects through use of a separate modeling language. Different objects may have different shade trees, so that an image can be rendered in which a multiplicity of different special-purpose models are mixed. Similarly, in Cook's system, lights and their parameters are defined by light trees, and atmospheric effects, such as haze, are defined by atmosphere trees.

Perlin [PERL85] has developed the notion of a *pixel-stream editor* that takes as input and produces as output arrays of pixels. A pixel is not rigidly defined and may include

Samsung/Google, EX1027, p. 821

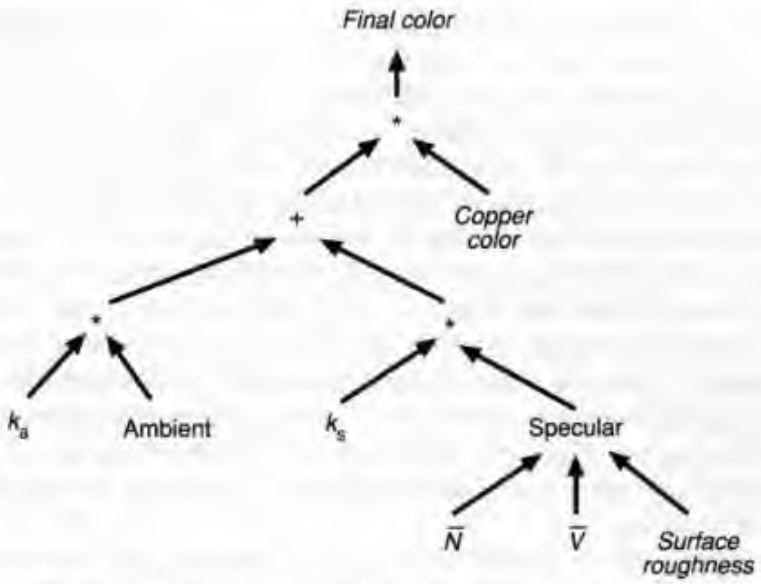


Fig. 16.77 Shade tree for copper. (After [COOK84].)

arbitrary data for a point in the image, such as the material identifier or normal vector at that point. An output pixel need not have the same structure as an input pixel. The pixel-stream editor executes a program written by the user in a high-level language oriented toward pixel manipulation. Thus, the user is encouraged to think of creating images in a series of passes, with intermediate results represented by arrays of pixels that may differ in the kind of information they encode.

The flexibility of shade trees and pixel-stream editors may be combined by designing a rendering system that allows its users to write their own shaders in a special programming language and to associate them with selected objects. This approach is taken in the RenderMan Interface [PIXA88; UPST89], a scene description specification that provides such a shading language. RenderMan defines a set of key places in the rendering process at which user-defined or system-defined shaders can be called. For example, the most common kind of shader, called a *surface shader*, returns the light reflected in a specified direction given a point on the surface, its orientation, and a set of light sources. A user-provided surface shader could implement an illumination equation totally different from those discussed so far. Other shaders include atmosphere shaders that modify the color of light passing between two points, and—in another example of how the word *shader* can be stretched—projection shaders that allow user-defined projections implementing other than parallel or linear perspective projections.

An example. Cook, Carpenter, and Catmull's Reyes Rendering Architecture [COOK87], which was used to produce Color Plates II.24–37, D, and F, provides an interesting example of how to structure a renderer. Reyes chops all objects up into *micropolygons*: small, constant-shaded quadrilaterals that are approximately ½ pixel on a side. This approach, known as *dicing*, occurs along boundaries that are natural for the

Samsung/Google, EX1027, p. 822

object. A patch, for example, is diced parallel to its (s,t) coordinate system. Dicing is performed prior to perspective transformation, based on an estimate of the size of the resulting micropolygons after projection. Much like Catmull's patch subdivision algorithm of Section 15.9, which subdivides patches until they are pixel-sized, Reyes subdivides objects until they are sufficiently small. Each kind of object is associated with a procedure that determines whether it should be subdivided further into other primitives or diced. An object is subdivided further if no method has been provided for dicing it directly, if it is determined that it would give rise to too many micropolygons, or if it is estimated that its micropolygons would differ too much in their final projected size. This recursive subdivision must ultimately result in objects that can be diced. To avoid the need to clip objects analytically to the view volume, when Reyes subdivides objects, only those parts that are at least partially within the view volume are kept. Perspective problems that would result from projecting an object that is too close to or behind the eye are avoided by subdividing further any object that spans both the hither plane and another plane that lies slightly in front of the eye.

Dicing an object results in a quadrilateral mesh of micropolygons that is shaded in WC. Because the micropolygons are sufficiently small, each is given a single shade, avoiding all the interpolated shading problems discussed in Section 16.2.6. Since a patch is diced parallel to its (s,t) coordinate system, some of the texture-mapping approaches discussed in Chapter 17 are particularly efficient to use. Dicing and shading can both take advantage of incremental algorithms. Reyes relies on the mapping techniques discussed in this chapter for its global lighting effects.

Visible surface determination is done with a subpixel z-buffer whose subpixel centers are jittered to accomplish stochastic sampling. The closest micropolygon covering a subpixel center is visible at that subpixel. To avoid the need to store micropolygon meshes and subpixel z and intensity values for the entire image, Reyes uses spatial partitioning. The image is divided into rectangular partitions into which each object is sorted by the upper left hand corner of its extent. The partitions are then processed left to right, top to bottom. As objects are subdivided or diced, the resulting subobjects or micropolygons are placed in the partitions that they intersect. Thus, only enough z-buffer memory is needed for a single partition, and other storage needed for a partition can be freed after it is processed.

16.14.4 Progressive Refinement

One interesting modification to the pipelines that we have discussed takes advantage of the fact that the image is viewed for a finite time. Instead of attempting to render a final version of a picture all at once, we can first render the picture coarsely, and then *progressively refine* it, to improve it. For example, a first image might have no antialiasing, simpler object models, and simpler shading. As the user views an image, idle cycles may be spent improving its quality [FORR85]. If there is some metric by which to determine what to do next, then refinement can occur adaptively. Bergman, Fuchs, Grant, and Spach [BERG86b] have developed such a system that uses a variety of heuristics to determine how it should spend its time. For example, a polygon is Gouraud-shaded, rather than constant-shaded, only if the range of its vertex intensities exceeds a threshold. Ray-tracing [PAIN89] and radiosity [COHE88] algorithms are both amenable to progressive refinement.

Samsung/Google, EX1027, p. 823

16.15 SUMMARY

In this chapter, we encountered many different illumination models, some inspired primarily by the need for efficiency, others that attempt to account for the physics of how surfaces actually interact with light. We saw how interpolation could be used in shading models, both to minimize the number of points at which the illumination equation is evaluated, and to allow curved surfaces to be approximated by polygonal meshes. We contrasted local illumination approaches that consider in isolation each surface point and the lights illuminating each point directly, with global approaches that support refraction and reflection of other objects in the environment. In each case, we noted that there are some methods that use the full geometric description of the environment in computing global effects, and others that use simpler descriptions, such as reflection maps.

As we have stressed throughout this chapter, the wide range of illumination and shading algorithms gives rise to a corresponding diversity in the images that can be produced of the same scene with the same viewing specification. The decision about which algorithms should be used depends on many factors, including the purposes for which an image is to be rendered. Although photorealism is often sacrificed in return for efficiency, advances in algorithms and hardware will soon make real-time implementations of physically correct, global illumination models a reality. When efficiency is no longer an issue, however, we may still choose to render some images without texture, shadows, reflections, or refraction, because in some cases this will remain the best way to communicate the desired information to the viewer.

EXERCISES

16.1 (a) Describe the difference in appearance you would expect between a Phong illumination model that used $(\overline{N} \cdot \overline{H})^n$ and one that used $(\overline{R} \cdot \overline{V})^n$. (b) Show that $\alpha = 2\beta$ when all vectors of Fig. 16.12 are coplanar. (c) Show that this relationship is *not* true in general.

16.2 Prove that the results of interpolating vertex information across a polygon's edges and scan lines are independent of orientation in the case of triangles.

16.3 Suppose there are polygons A, B, and C intersecting the same projector in order of increasing distance from the viewer. Show that, in general, if polygons A and B are transparent, the color computed for a pixel in the intersection of their projections will depend on whether Eq. (16.25) is evaluated with polygons A and B treated as polygons 1 and 2 or as polygons 2 and 1.

16.4 Consider the use of texture mapping to modify or replace different material properties. List the effects you can produce by mapping properties singly or in combination. How would you apply antialiasing to them?

16.5 Although using a reflection map may appear to require precomputing the lighting for the environment, a reflection map containing object identities and surface normals could be used instead. What are the disadvantages of using this kind of map?

16.6 Explain how to simulate reflections from surfaces of different roughness using a reflection map.

16.7 What other lighting effects can you think of that would generalize Warn's flaps and cones?

16.8 Suppose that the array of patches shown in Fig. 16.64 is continued for another two rows, adding patches 5 and 6, and that the radiosity values for the patches are $B_1 = B_2 = 2$, $B_3 = B_4 = 4$, $B_5 = B_6 = 6$. Show that B_A and B_e are 5 and 3, respectively. Then show that B_b is 1. Is this a reasonable value?

Samsung/Google, EX1027, p. 824

Notice that it extends the linear trend from h to e. What happens as you add more rows of patches in a similar pattern? Suppose that you added a mirror image of the patches about the line ac and computed the radiosity values. Then B_b would be 2. Does this seem contradictory? Explain your answer.

16.9 Implement a simple recursive ray tracer based on the material in Sections 15.10 and 16.12.

16.10 Make your ray tracer from Exercise 16.9 more efficient by using some of the techniques discussed in Section 16.12.1.

16.11 Extend your ray tracer from Exercise 16.10 to do distributed ray tracing.

16.12 Implement a progressive-refinement radiosity algorithm, based on the pseudocode of Fig. 16.70. Use the hemicube method of computing form factors. Begin by computing only patch to patch exchange (ignoring substructuring). Leave out the ambient computation to make coding and visual debugging easier. Check your hemicube code by verifying that the delta form factors sum to (approximately) 1.

To display your images, you will need to implement a polygon visible-surface algorithm (perhaps the one used by your hemicube) or have access to an existing graphics system. Using constant-shaded polygons will improve interactivity if shaded graphics hardware is not available (and will make programming and debugging easier).

16.13 Explain why lighting must be done before clipping in the pipeline of Fig. 16.72.

16.14 Implement a testbed for experimenting with local illumination models. Store an image that contains for each pixel its visible surface's index into a table of material properties, the surface normal, the distance from the viewer, and the distance from and normalized vector to one or more light sources. Allow the user to modify the illumination equation, the intensity and color of the lights, and the surface properties. Each time a change is made, render the surface. Use Eq. (16.20) with light-source attenuation (Eq. 16.8) and depth-cueing (Eq. 16.11).

16.15 Add a shadow algorithm to a visible-surface algorithm that you have already implemented. For example, if you have built a z-buffer system, you might want to add the two-pass z-buffer shadow algorithm discussed in Section 16.4.4. (The postprocessing variant may be particularly easy to add if you have access to a graphics system that uses a hardware z-buffer. Explain how extra storage at each pixel, as described in Exercise 16.14, could be used to design a shadow postprocess that produced correct shading and proper highlights.)

16.16 Add interobject reflections to a visible-surface algorithm. Use reflection mapping for curved surfaces and the mirror approach for planar surfaces, as described in Section 16.6.

Samsung/Google, EX1027, p. 825

17
Image Manipulation and Storage

In this chapter, we explore methods for manipulating and storing images efficiently. We begin by considering the kinds of operations we would like to perform on images. Bear in mind that the images we are manipulating may be used either as images in their own right, or in the manufacture of some subsequent image, as in the environment mapping described in Chapter 16.

Several sorts of operations on images immediately come to mind. One is combining two images by overlaying or blending them, known as *compositing*. One application of compositing is in animation, when we wish to show a character moving around in front of a complicated background that remains unchanged. Rather than rerendering the background for each frame, we can instead render the background once and then generate many frames of the character moving about on a black background. We can then composite these individual frames as overlays to the background frame, thus producing images of a character moving about on the background. In compositing operations like this, antialiasing becomes extremely important to ensure that the outline of the character is not jagged against the background. It is also necessary to distinguish the background of an image from the content; in our example, the black background against which the character is drawn is the *background*, and the character itself is the *content*.

Often, the images to be composited are of different sizes, so we may wish to translate, scale, or rotate them before the composition. We may even wish to distort an image, so that it appears in perspective or appears to have be drawn on a rubber sheet and then stretched. Although we could make these changes by rerendering the image with an appropriate geometric transformation, this is often so difficult or time consuming as to be impractical. Indeed, it can be impossible if, say, the image has been obtained from an optical scan of a

Samsung/Google, EX1027, p. 826

photograph, or if the original program or parameters used to create it have been lost.

We might also wish to apply various filters to an image so as to produce false colors, to blur the image, or to accentuate color or intensity discontinuities. This sort of filtering is applied to satellite photographs and to computed-tomography (CT) data, where the intensity of a point in the image reflects the density of material in the body. For example, very slight changes in intensity may indicate the boundaries between normal and cancerous cells, and we may wish to highlight these boundaries.

Images tend to be very large collections of data. A 1024 by 1024 image in which the color of each pixel is represented by a n-bit number takes $n/8$ MB of memory (in an 8-bit-per-byte machine). As described in Chapter 4, many graphics systems dedicate a great deal of memory to image storage (the frame buffer). If the image memory is accessible by other programs, then it may be used for output by one program, and then for input by another, or even for output by two different programs. This happens, for example, when we use a pixel-painting program to adjust individual pixels of a rendered image. This use of image memory (and the rigid structure of the memory, which constitutes a database format for diverse programs) has been called "frame-buffer synergy" by Blinn [BLIN85].

When an image is being stored in secondary memory, it is often convenient to compress the stored data (but not the image). Several schemes have been developed. The look-up tables (LUTs) described in Chapter 4, for example, significantly reduce the storage needed for an image, provided the image contains substantial color repetition. Of course, storing LUTs is typically done only when the frame buffers used for displaying the image support LUTs. We discuss several more sophisticated methods in Section 17.7.

Here we begin by reexamining our notion of an image. Then we describe some elementary operations on images: filtering and geometric transformations. We then discuss techniques for storing additional data with each pixel of an image, and using these data in compositing. Following this, we discuss various image storage formats; finally, we describe a few special effects that can be performed at the image level rather than in modeling or rendering.

17.1 WHAT IS AN IMAGE?

Images, as described in Chapter 14, are (at the most basic level) arrays of *values*, where a value is a collection of numbers describing the attributes of a pixel in the image (in bitmaps, e.g., the values are single binary digits). Often these numbers are fixed-point representations of a range of real numbers; for example, the integers 0 through 255 often are used to represent the numbers from 0.0 to 1.0. Often, too, these numbers represent the intensity at a point in the image (*gray scale*) or the intensity of one color component at that point. The dimensions of the array are called the *width* and *height* of the image, and the number of bits associated with each pixel in the array is called the *depth*.

We often consider an image as more than a mere array of values, however. An image is usually intended to represent an *abstract* image, which is a function of a continuous variable; each position in the abstract image has some value.[1] The images we work with

[1]What we are really talking about is a function whose domain is a rectangle in the Euclidean plane, rather than a discrete lattice of points in the plane.

Samsung/Google, EX1027, p. 827

(sometimes called *digital images* or *discrete images*) are functions of a discrete variable; for each $[i, j]$ pair, a value is associated with the pixel labeled $[i, j]$. As described in Chapters 3 and 14, choosing the best discrete image to represent an abstract image is difficult. In this chapter, we sometimes discuss reconstructing the abstract image in order to take new samples from it. Of course, we do not actually perform this reconstruction, since to do so we would need to generate values at infinitely many points. But we can reconstruct any individual value in the abstract image—in particular, we can reconstruct the finitely many values we want to sample.

If we create a discrete image from an abstract image by sampling (see Chapter 14), then reconstruct an abstract image from the digital image, the reconstructed abstract image and the original abstract image may or may not be the same. If the original abstract image had no high-frequency components, then the reconstructed image would be the same as the original, and the reconstruction would be said to be *faithful*. On the other hand, if the original image had components whose frequencies were too high, then the sampled image could not represent it accurately, and the reconstructed image would differ from the original.

One other aspect of images is important. Although filtering theory tells us a great deal about selecting a discrete image to represent an abstract image most accurately, much of the theory assumes that the abstract-image values at each point are real numbers and that the discrete-image values at each pixel will also be real numbers. In the case of bitmaps, however, nothing could be further from the truth: The values are binary. In more complex pixmaps, the values may be small binary numbers (e.g., 4 bits per pixel), or may range over so large a collection of numbers as to be effectively continuous. This *value discretization* leads to significant questions in image manipulation, such as how best to compress a bitmap. If 4 pixels—2 white and 2 black—are to be compressed into 1, should the compressed pixel be black or white? We discuss the consequences of value discretization when they are known and significant, but note that there is much that we do not yet understand.

17.2 FILTERING

Suppose we have an image produced without any antialiasing—for example, a drawing of a graph that was read into memory with an optical scanner that sampled the drawing at an array of points. How can we improve its appearance? The image certainly has jaggies that we would like to remove. But *every* image we can create is correct for *some* source image (where by *correct* we mean that it accurately represents a sample of the source image after low-pass filtering). If applying some mechanism alters and improves one image, applying the same mechanism to another image may damage that image. Thus, the mechanism we are about to describe should be used only on images that need smoothing. If jagged steps are present in a image that has been generated properly, then they are meant to be there, and postfiltering will only blur the image. (After all, what *should* an image of a staircase look like?)

Suppose we do want to smooth out an image to hide some jaggies. What can we do? An obvious start is to replace each pixel with the average of itself and its neighbors. This process, applied to the discrete image rather than to the abstract image, is called

Samsung/Google, EX1027, p. 828

postfiltering. With postfiltering, pixels near the stair steps in the jaggies are blended so as to hide the steps; see Fig. 17.1, in which the filtering has been exaggerated. As we saw in Chapter 14, this constitutes filtering with a box filter; other, more sophisticated filters may yield better results. Before we examine other filters, let us consider the drawbacks of even this simple filtering method.

Suppose that we point sample a photograph of a picket fence. The pickets and the gaps between them are of the same width, the pickets are white, and the background is black. The pickets are spaced in the photograph so that the width of nine pickets and nine gaps covers a width of 10 pixels in the image. What will the sampled image look like? If the photograph is positioned so that the first pixel is exactly at the left-hand edge of the first picket, then the first pixel will be white, the next 5 pixels will be black, but the sixth through tenth pixels will be at pickets and hence will be white. The next 5 pixels will be black, and so on (see Fig. 17.2).

Now, what does our postfiltering do in this situation? It smoothes out the boundary between the sixth and seventh pixels, and leaves a large block of black followed by a large block of white. It cannot possibly fix all the problems implicit in the image. Clearly, postfiltering is not a good solution to the aliasing problem. In addition, since postfiltering will also blur any other edges in the image (even those that *should* be there), the resulting image will be unpleasantly fuzzy.

This problem can be partly remedied at the cost of shrinking the image: We can convert a $2n$ by $2n$ image into an n by n image by imagining that the source image is overlaid with a grid, each square of the grid enclosing 4 pixels of the source image. We can then average the 4 pixels in the square to create 1 pixel in the target image for each grid square. This amounts to postfiltering the image, then selecting alternate pixels on alternate scan lines. Note that less filtering computation is involved; we need to apply the filter to compute values for only those pixels included in the final image. That is, for only those pixels to appear in the output image, we compute a weighted average of pixels around the corresponding point in the source image. Of course, the resulting image is one-fourth the size of the original.

We can see how this works by recalling the analysis in Chapter 14. If the source image is produced by sampling at a frequency of 2ω, then any component of the original signal whose frequency is between 0 and ω will be accurately represented. For any frequency above ω, say $\omega + \phi$, the sampled signal will contain an alias at frequency $\omega - \phi$. Box filtering the sampled signal with a filter of width 2 substantially (but not completely) filters out the components of this signal with frequencies greater than $\omega/2$ (because the Fourier transform of the box filter is a sinc function, which tapers off rapidly as the frequency increases). Resampling at alternate pixels yields an effective sampling rate of ω; with this



Fig. 17.1 The stair steps are smoothed by box filtering.

Samsung/Google, EX1027, p. 829

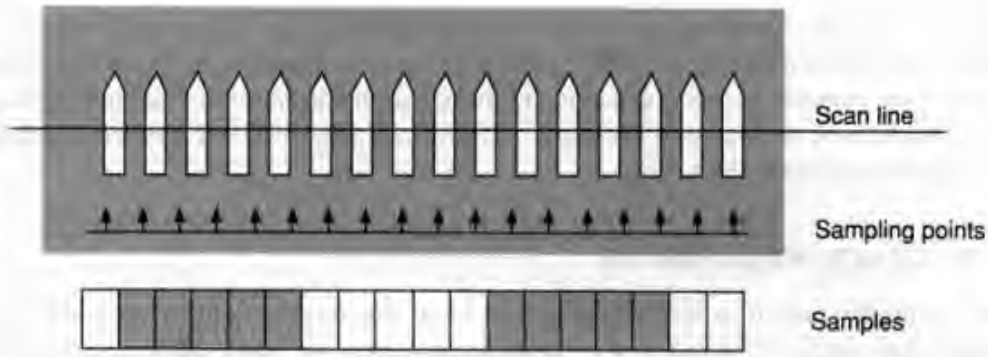


Fig. 17.2 Aliasing in a sampled image.

sampling rate, all frequencies up to $\omega/2$ can be accurately represented. But after applying the filter, these are exactly the frequencies that remain. If the original signal had components $\omega + \phi$ for which ϕ was large (i.e., greater than $\omega/2$), then the aliases of these components occur at frequencies below $\omega/2$, and hence persist in the final image. But for small values of ϕ, the aliases are filtered out, and so supersampling and postfiltering really do help reduce aliases.

Remember that the signals that represent such primitives as lines, rectangles, and any other geometric objects with clearly defined edges have components of arbitrarily high frequencies, so there is no hope of representing these correctly by any such method. At best, we can hope postfiltering will improve a bad image at the cost of fuzziness.

Other filters, such as the sinc filter, the Catmull–Rom filter, and the triangle filter, can produce better postfiltering results than can a pure box filter. The analysis of those filters given in Chapter 14 applies here as well. As a convenient rule of thumb, Whitted has suggested that postfiltering a high-resolution image produces obvious fuzziness, but that a 2048 by 2048 image can usually be postfiltered and sampled down to a 512 by 512 image with good results [WHIT85].

Now consider a temporal analogue of this problem: The spokes on a wagon wheel pass by a pixel on the screen very fast in an animation of a rolling wagon (this is the temporal analog of an object being striped with rapidly changing color; i.e., to closely spaced stripes). The frequency with which the spokes pass a point may be far greater than 30 times per second, the speed of typical video recording. Temporal aliasing is the inevitable result: The spokes appear to stand still or to turn backward. We are used to seeing this effect in movies, of course. In a movie, however, we actually see a *blurred* wagon wheel moving backward, because, in each exposure of a movie frame, the shutter is open for a brief (but not infinitesimal) time period (about one half of the time allocated to the frame; the remaining half is dedicated to moving the film forward). The shutter effectively applies a box filter to the scene in the time dimension. The result is some blurring, but aliases are still present. The blurring is due to the box filtering, and the aliases are due to the narrowness of the filter. All the box filters taken together cover only about half of the time sequence of the movie—the remainder is lost while the shutter is closed. The implication for computer graphics is clear: To get movie-quality frames for animation, we need to do (at the very

Samsung/Google, EX1027, p. 830

least) box filtering—*prefiltering*—over time. Postfiltering removes some ill effects, but many remain. Notice that, to get really accurate images, we should actually do sinc filtering over the time domain. If movie cameras did this (or even had wider box filters), the wagon wheels would look the way they do in life—they would appear to roll forward and then to blend into a continuous blur.

17.3 IMAGE PROCESSING

Now we briefly turn to a different problem: How can we highlight or suppress certain features in an image? This question is really in the domain of *image processing* rather than computer graphics, but a few basic ideas are worth discussing. By scanning an image for rapid changes in value at adjacent points, we can do *edge detection* and *enhancement*. At places where the values of adjacent points differ sufficiently, we can push the values even further apart. If an image is *noisy*—that is, if random displacements have been added to its pixel values—then it can be *smoothed* by the filtering techniques discussed in the previous section. If the noise is sufficiently random, then filtering, which computes averages of adjacent pixels, should average out the noise, or at least filter its high-frequency components.

Another image-processing technique is *thresholding*, in which the points of an image at or near a particular value are highlighted. In a gray-scale image, this highlighting can be done by converting all pixels below some value to black, and all pixels above that value to white, producing a threshold edge between the black and white regions. The marching-cubes algorithm discussed in Chapter 20 gives a different mechanism for thresholding (in 3D): It explicitly constructs the boundary between the two regions as a surface (or a curve, in the 2D case). The components of this boundary can then be rendered into a new image with appropriate antialiasing to give a smoother indication of the threshold. For further information on this, see [GONZ87; SIG85].

17.4 GEOMETRIC TRANSFORMATIONS OF IMAGES

Suppose we wish to transform an image geometrically. Such transformations include translation, rotation, scaling, and other, nonlinear, operations. How can we do this?

Translating an image makes sense only if the image is thought of as a subimage of some larger image. Suppose we wish to move an n by k array of pixels (the *source*), whose upper-left corner is at (a, b), to a new position, with the upper-left corner at position (c, d) (the *target*). This transformation should be easy; we simply copy pixels from the source position to the target position, and (if we want) replace all source pixels that are not target pixels with the background color (see Fig. 17.3). Provided that care is taken to ensure that the copying is done in an order that prevents overwriting source pixels, when the source and destination overlap, and provided that the four numbers a, b, c, and d are all integers, this approach works fine.

But what if the starting and ending positions are not integers? Then we wish to reconstruct the abstract image for the source image, to translate it, and to sample this translated version. To do this explicitly is not feasible—we certainly do not wish to reconstruct the abstract image at *every* possible location, and then to select just a few of

Samsung/Google, EX1027, p. 831

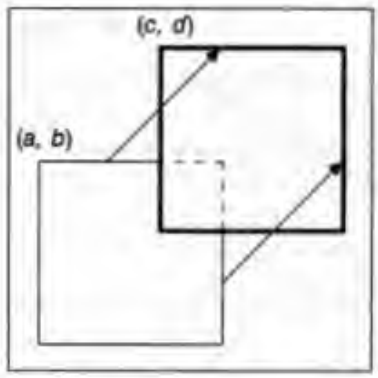


Fig. 17.3 A simple translation of an image.

these (infinitely many) points. Indeed, the same objection holds for scaling and rotation. Algorithms have been developed, however, that perform these operations in ways that are computationally correct (in various senses). Weiman has developed algorithms for performing scaling and shearing of images by rational amounts [WEIM80]. Rotations can be performed by a clever combination of these algorithms (see Exercise 17.1). Finding a similar algorithm for translating by arbitrary rational amounts is posed as Exercise 17.2.

17.4.1 Basic Geometric Transformations

Weiman posits that a gray-scale pixmap represents an abstract image in the following fashion: The abstract image is divided into squares (one square per pixel), and the average intensity of this abstract image in the square is the value assigned to the pixel. He thus assumes that he can perform a faithful reconstruction by drawing a picture consisting of gray squares whose tone is determined by the pixel values. Stretching a pixmap by a factor of p/q takes q columns of the original and stretches them to cover p columns of the target image. Performing area sampling on the result then generates the target image. Filtering theory tells us that this assumption about the nature of a sampled image and the consequent stretching algorithm are wrong in every sense: An abstract image should never be sampled while it has frequency components above the Nyquist frequency, and hence a proper reconstruction of an abstract image from a sampled one never has high-frequency components. An image in which adjacent squares have different (constant) values is a perfect example of an image with lots of high-frequency components, so this is certainly a bad reconstruction. And finally, filtering theory says that when converting such an image to a pixmap, we should use sinc filtering rather than box filtering. Nevertheless, if Weiman's hypotheses are allowed, his algorithm for performing these linear transformations is quite clever. It is also the basis for a very good bitmap-scaling algorithm (see Exercise 17.3).

Suppose we wish to scale an image by a factor p/q (where $p > q$, and p and q are integers with no common factors). The first step is to generate a Rothstein code [ROTH76] for the number p/q. This code is a binary sequence that describes a line whose slope is q/p (any scan-converted line can be used to generate a similar code). Figure 17.4 shows a line of slope $\frac{4}{11}$ with 15 tick marks on it. As the line passes from left to right through the figure, it crosses the horizontal grid lines. If a column contains such a grid-line crossing, it is marked

Samsung/Google, EX1027, p. 832

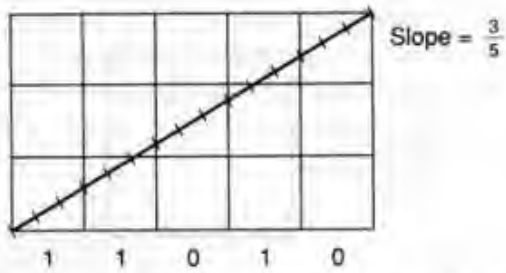


Fig. 17.4 The Rothstein code for a line of slope $\frac{3}{5}$.

with a 1; otherwise, is marked with a 0. Each column contains three tick marks; the bit associated with the column is 0 unless one of the three tick marks is at a multiple of 5, since multiples of 5 are where horizontal-line crossings occur. Thus, the interval between mark 9 and mark 12 is assigned a 1, since mark 10 lies within it. (A tick at the left side of a column is considered to be in the column, whereas ticks on the right are not.)

The Rothstein code may be viewed as a mechanism for distributing q 1s evenly among p binary digits. We can therefore use it to tell us how to distribute each of the q columns of the source image among p columns of the target image. Unfortunately, using the 1s in the Rothstein code as indicators of where to copy the source data leaves some of the target columns blank. The Rothstein code can be cyclically permuted, however, to give different mappings of source to destination.[2] Taking the average of these gives the result.

The pseudocode for this procedure is shown in Fig. 17.5.

To scale by a number smaller than 1, we simply reverse the process. The Rothstein code for q/p tells us which of the source columns should appear in the target (a 1 in the Rothstein code tells us to select that column). Again, we average over all cyclic permutations of the code.

The Rothstein code can also be used to generate a description of shearing. A 1 in the Rothstein code indicates that the corresponding column of the source pixmap should be shifted up 1 pixel. Over the course of q columns, there are p shifts, resulting in a vertical shear of amount q/p. Once again, we should cyclically permute and average the results.

17.4.2 Geometric Transformations with Filtering

Feibush, Levoy, and Cook give a somewhat more sophisticated mechanism for transforming images [FEIB80]. (Their algorithm is developed for use in an algorithm for mapping textures onto surfaces—see Chapter 16.) Before discussing the details, we note the algorithm's good and bad points. The algorithm has the advantage that it computes reasonable values for boundary pixels: If an image is rotated so that some target pixel is only partly covered by a source pixel, the algorithm recognizes that pixel as a special case and processes it accordingly. It computes values by applying a weighted filter to the source image to determine pixel values for the target image, which helps to reduce aliasing in the resulting image. But since this filter is more than 1 pixel wide, if the algorithm is used to

[2] A cyclic permutation of a binary sequence is the repeated application of logical shift operations to the sequence.

Samsung/Google, EX1027, p. 833

```
/* Expand the width of an image by p/q. */
void WeimanExpansion(
        const grayscalePixmap source,      /* Source image, size n × k */
        grayscalePixmap target,            /* Target image, width at least k * p/q */
        int n, int k,                      /* Size of source image */
        int p, int q)                      /* Scale factor is p/q */
{
    char roth[MAX];                        /* The array must hold at least p items. */
    int i, j, s;                           /* Loop indices */

    /* Source image is n × k, target image is to be n × ceil (k * p/q).[3] */
    int targetWidth = ceil (k * p / (double) q);

    Rothstein (roth, p, q);                /* Store the Rothstein code for p/q in array roth. */
    SetToBlank (target, n, targetWidth);   /* Clear the target array. */
    for (i = 0; i < p; i++) {              /* For several passes through the algorithm... */
        int sourceCol = 0;
        Permute (roth);                    /* Apply cyclic permutation to Rothstein code. */
        /* For each column of the target */
        for (j = 0; j < targetWidth; j++) {
            if (roth[j] == 1) {            /* If code says to copy source column */
                for (s = 0; s < n; s++) {  /* Copy all the pixels. */
                    target[s][j] += source[s][sourceCol];
                }
                sourceCol++;               /* Go to next column. */
            }
        }
    }

    /* Divide by q to compensate for adding each source column to target q times. */
    for (i = 0; i < n; i++)
        for (j = 0; j < targetWidth; j++)
            target[i][j] = rint (target[i][j] / (double) q);
}  /* WeimanExpansion */
```

Fig. 17.5 The Weiman algorithm for expanding an image.

perform the identity transformation on an image, it blurs the values. In Section 17.5.3, we discuss this and other drawbacks of transformation algorithms.[4]

At this point, it is worthwhile to separate two aspects of image transformation. The first is computing which point in the source image is mapped to the center of the pixel in the

[3] The ceiling of a number is the smallest integer greater than or equal to the number; ceiling(1.6) = 2, ceiling(1.1) = 2, and ceiling(6.0) = 6.

[4] Feibush, Levoy and Cook note that any filter can be used, but describe the algorithm in terms of a filter of diameter 2. The algorithm generally performs better with this filter than it does with a unit-area box filter.

Samsung/Google, EX1027, p. 834

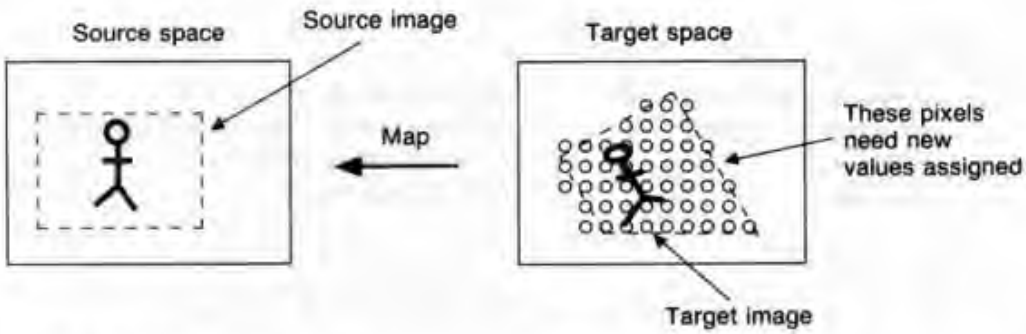


Fig. 17.6 The relationship of the source space, source image, target space, and target image in the Feibush–Levoy–Cook algorithm.

target image. The second is computing the value for the pixel in the target image. The first task is merely algebraic, in that it involves computing values (and inverse values) of a transformation. This may be done efficiently by various incremental methods. The second task also has numerous solutions, all of which involve choosing some filtering function to apply to the original image. The method described in the next few pages assumes a filtering function that is circularly symmetric and has a modest size (i.e., is nonzero only on a small part of the plane).

The algorithm starts with a source image (thought of as lying in one copy of the Euclidean plane, called the *source space*), a projective map[5] from another copy of the Euclidean plane (the *target space*) to the source space, and a polygonal region in the target space. The target image is the collection of pixels in the target space that are near the polygonal region, and it is these pixels whose values need to be assigned (see Fig. 17.6). Note that the projective map here goes from target to source, the reverse of the usual naming convention for mathematical functions.

To start, we choose a symmetric filter function that is nonzero only for (x, y) very close to $(0, 0)$ (perhaps within a 2-pixel distance). The *support* of this filter function is the set of points on which it is nonzero. We take a copy of the bounding rectangle for the support of the filter and translate it to each pixel in the target space. Whenever this rectangle intersects the target polygon, the pixel is considered to be in the target image. This translated rectangle is called the *bounding rectangle for the target pixel*, and the translated support of the filter function is called the pixel's *convolution mask* (see Fig. 17.7).

The vertices of the target polygon are transformed to source space just once, for repeated use. The resulting polygon is called the *source polygon*. The bounding rectangle of each target pixel is transformed to the source space, where it becomes a quadrilateral. A bounding rectangle for this quadrilateral is computed, then is clipped by the source polygon (because clipping a rectangle by the source polygon is much easier than clipping a general quadrilateral). The pixels in the source space that lie in this clipped quadrilateral are transformed to the target space; only those that fall within the target pixel's bounding rectangle are retained.

---

[5]A *projective map* is a map represented by a 3×3 matrix operating on the plane using homogeneous coordinates, in the manner described in Chapter 5.

Samsung/Google, EX1027, p. 835

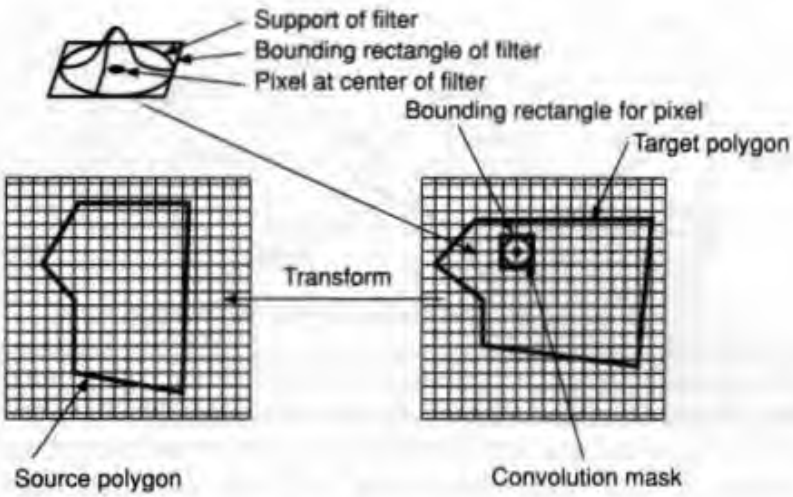


Fig. 17.7 Terms used in the Feibush–Levoy–Cook algorithm.

These transformed pixels are then averaged together by the weights given by the filter to yield a value for the target pixel. This target pixel value is correct only if the entire pixel is within the transformed image boundaries. If the image has been rotated, for example, then the transformed edges of the image may cut across pixels (more precisely, across their convolution masks).

Thus pixels are not entirely determined by the value just computed; that value only contributes to the pixel's value, in proportion to the coverage of the pixels. The contribution can be determined analytically. Figure 17.8 shows the transformed edge of the source image passing through a pixel's bounding rectangle, and within that rectangle passing through the pixel's convolution mask. To find the contribution of the computed value to the pixel's final value, we do the following:

1. Clip the image polygon against the bounding rectangle for the pixel (see Fig. 17.9). The points of intersection with the edges of the bounding rectangle were already computed in determining whether the pixel was in the target image.
2. For each vertex of the clipped polygon (in Fig. 17.9, a single triangle with vertices

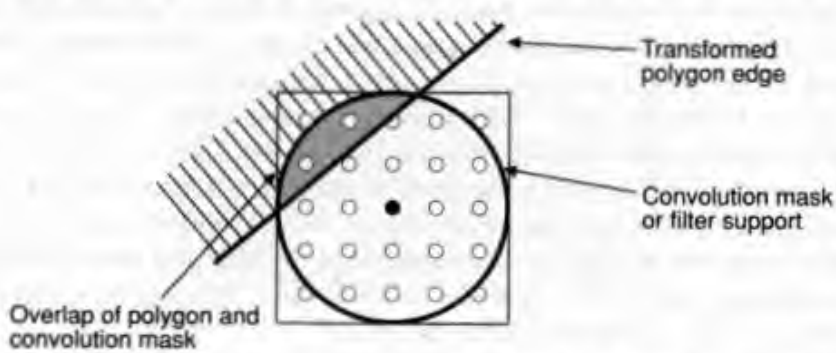


Fig. 17.8 Filtering for a pixel at the edge of the polygon.

Samsung/Google, EX1027, p. 836

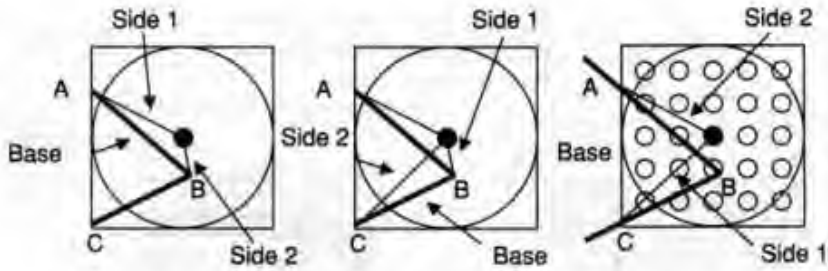


Fig. 17.9 The steps in filtering an edge.

labeled *A*, *B*, and *C*, in clockwise order), construct a triangle with sides *BASE* (the edge from this vertex to the next vertex in clockwise order around the polygon), *SIDE1* (the edge from this vertex to the center of the pixel) and *SIDE2* (the edge from the next vertex to the center of the pixel).

3. Consider the filter function as being plotted in a third dimension above the convolution mask. The weight a region contributes to the total for a pixel is proportional to the volume above that region and under the graph of the filter. So we now compute the volumes above each triangle. As Fig. 17.9 shows, some of these volumes must be added and some subtracted to create the correct total contribution for the region. The rule is that the volume is added if the cross-product of *SIDE1* and *SIDE2* points into the page; otherwise, it is subtracted (see Exercise 17.4).

Computing the volumes in step 3 is easier than it might appear, in that they can be precomputed and then extracted from a look-up table during the actual filtering process. Exercise 17.5 shows how to do this precomputation.

17.4.3 Other Pattern Mapping Techniques

The Feibush-Levoy-Cook algorithm provides excellent results for pattern mapping onto polygons, but requires computing a filtered value at each point, so that for each pixel in the image a filtering computation is performed. In a perspective picture of a plane (receding to a vanishing point) a single pixel in the final image may correspond to thousands of pixels in the source pattern, and thus require an immense filtering computation. Several techniques have been developed to produce more rapid (if sometimes slightly less accurate) filtering.

Williams [WILL83] takes the source image and creates a MIP (*multum in parvo*—many things in a small place) map, which occupies $\frac{4}{3}$ of the memory of the original. If the original image is a 512 by 512 pixel, 24-bit true color image, using 8 bits each for the red, green, and blue information, the MIP map is a 1024 by 1024 by 8 bit image. The red, green, and blue parts of the original image each occupy one quarter of the MIP map, and the remaining quarter is filled with filtered versions of these, as shown in Fig. 17.10. When a target pixel is covered by a collection of source pixels, the MIP map pixels corresponding to this collection most closely are used to give a filtered value. Linear interpolation between levels of filtering is used to further smooth the values.

Crow [CROW84] devised a scheme by which box filtering of an image over any aligned rectangle can be done rapidly. For quick pattern mapping, this suffices in many cases—a rectangular box corresponding closely to the shape of the transformed target pixel is used to

Samsung/Google, EX1027, p. 837

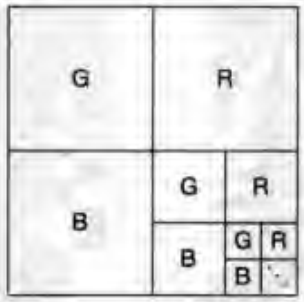


Fig. 17.10 A MIP map. The red, green, and blue channels of the original image fill three quarters of the MIP map. Each is filtered by a factor of 4, and the three resulting images fill up three quarters of the remaining quarter. The process is continued until the MIP map is filled.

compute a filtered pattern value for the pixel. The scheme is based on the algebraic identity $(x + a)(y + b) - (x + a)\,y - x\,(y + b) + xy = ab$. Interpreted geometrically, this says that the area of the small white rectangle in Fig. 17.11 can be computed by taking the area of the large rectangle and subtracting the areas of both the vertically and the horizontally shaded rectangles, and then adding back in the crosshatched rectangle (which has been subtracted twice). By taking the source image and creating a new image, whose value at pixel (x, y) is the sum of all the values in the source image in the rectangle with corners $(0, 0)$ and (x, y), we create a *summed area table*, S. We can now compute the sum of the pixels in the rectangle with corners at (x, y) and $(x + a, y + b)$, for example, by taking $S[x + a, y + b] - S[x + a, y] - S[x, y + b] + S[x, y]$.

Glassner [GLAS86] observes that if the transformed pixel is not approximately an aligned rectangle, then summed area tables may blur the result excessively. He therefore develops a system in which the excess area in the aligned bounding box for the pixel is systematically trimmed, in order to provide a more accurate estimate of the filtered source image at the point. This requires detecting the geometry of the inverse-mapped target pixel relative to its bounding box.

Heckbert [HECK86a] proposes a system using both the Feibush-Levoy-Cook method and MIP maps. He maps the target pixel's filter support (which is supposed to be circular,

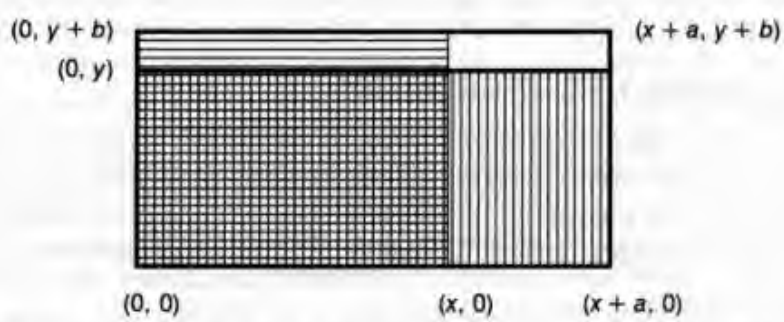


Fig. 17.11 The area of the small white rectangle in the image is computed by subtracting the horizontally and vertically shaded areas from the area of the large rectangle, and then adding back in the area of the crosshatched rectangle.

Samsung/Google, EX1027, p. 838

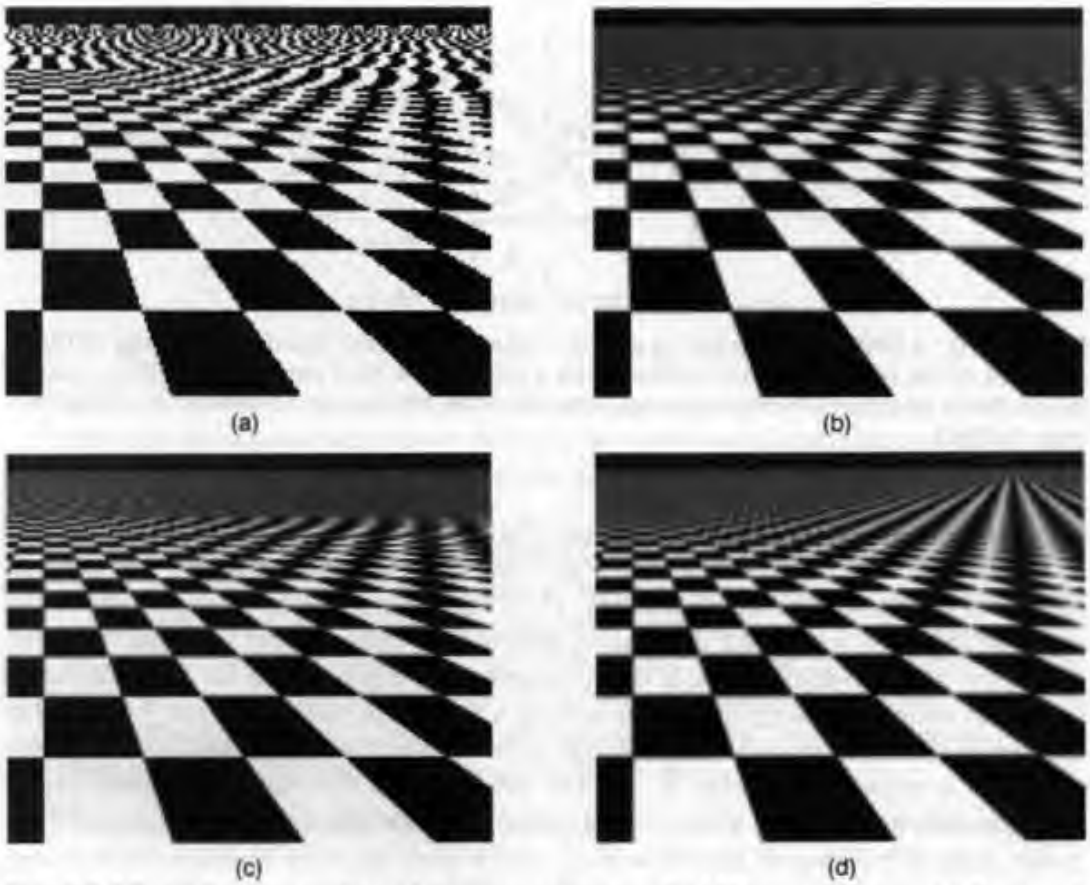


Fig. 17.12 (a) Point sampling of the source image. (b) MIP map filtering. (c) Summed area table filtering. (d) Elliptical weighted average using MIP maps and a Gaussian filter. (Courtesy of P. Heckbert.)

and is defined by a quadratic function) to an elliptical region in the source image (defined by a different quadratic). Depending on the size of this region in the source image, an appropriate level in a MIP map for the source image is selected, and the pixels within it are collected in a weighted sum over the elliptical region. This weighted sum is the value assigned to the target pixel. This combines the accuracy of the Feibush-Levoy-Cook technique with the efficiency of the MIP map system. A comparison of pattern-mapping results is shown in Fig. 17.12.

17.5 MULTIPASS TRANSFORMATIONS

Suppose that we take the image shown in Fig. 17.13(a) and apply a vertical shearing transformation to it, as shown in part (b), and then we follow this with a horizontal shearing transformation,[6] as shown in part (c). Provided we choose the right transformations, the net effect is to rotate the image as shown [CATM80]. Such a two-pass technique may be much

[6]The two shearing transformations are actually shear-and-scale transformations. The first takes a column of pixels and translates and compresses it in the vertical direction. The second does the same for a row of pixels.

Samsung/Google, EX1027, p. 839



(a)



(b)



(c)

Fig. 17.13 A rotation may be expressed as a composition of a column-preserving and a row-preserving transformation. (a) The original image. (b) A column-preserving transformation has been applied to the image. (c) A row-preserving transformation has been applied to the second image. (Courtesy of George Wolberg, Columbia University.)

faster to compute than a direct application of the rotation transformation, since it operates on one vertical or horizontal line of pixels at a time, and the computations within each such line can be performed incrementally. Also, in many cases, the filtering necessary to avoid aliasing artifacts can be performed line by line as well. More important still is that a wide class of transformations can be implemented as multipass transformations [CATM80, SMIT87]. This multipass technique has been implemented in the Ampex digital optics (ADO) machine [BENN84], which is widely used in video production. A survey of this and other image warping techniques is given in [WOLB90].

Implementing two-pass (or multipass) transformations can be divided into two subtasks: finding the correct transformations for the individual passes, which is a purely algebraic problem, and applying the correct filtering to generate new pixels, which is an antialiasing problem. Since the second part will depend on the solution to the first, we begin by solving the first problem in the case of a rotation.

Samsung/Google, EX1027, p. 840

17.5.1 The Algebra of Multipass Transforms

To simplify the discussion, we will use three different sets of coordinates. The original image will be written in (x, y) coordinates, the vertically sheared image in (u, v) coordinates, and the final image in (r, s) coordinates. The first shearing transformation will be called A, the second B, and their composition, which is the rotation, will be called T. Thus,

$$\begin{pmatrix} u \\ v \end{pmatrix} = A \begin{pmatrix} x \\ y \end{pmatrix},$$

and

$$\begin{pmatrix} r \\ s \end{pmatrix} = B\begin{pmatrix} u \\ v \end{pmatrix} = B\left(A\begin{pmatrix} x \\ y \end{pmatrix}\right) = T\begin{pmatrix} x \\ y \end{pmatrix}.$$

From Chapter 5, we know the formula for T:

$$\begin{pmatrix} r \\ s \end{pmatrix} = T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x\cos\phi - y\sin\phi \\ x\sin\phi + y\cos\phi \end{pmatrix}.$$

From this, we will determine the formulae for A and B.

The transformation A is supposed to be *column preserving*; that is, it must send each column of the original image into the corresponding column of the transformed image. Thus, if the pixel (x, y) is sent to (u, v) by A, then $u = x$. In other words, A must be written in the form

$$\begin{pmatrix} u \\ v \end{pmatrix} = A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ f(x, y) \end{pmatrix}$$

for some function f. In the same way, B is supposed to be *row preserving*, so B must be written in the form

$$\begin{pmatrix} r \\ s \end{pmatrix} = B\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} g(u, v) \\ v \end{pmatrix}$$

for some function g. To determine the formulae for A and B, we need to find the functions f and g.

Writing out the composite, we have

$$\begin{pmatrix} r \\ s \end{pmatrix} = B\begin{pmatrix} u \\ v \end{pmatrix} = B\left(A\begin{pmatrix} x \\ y \end{pmatrix}\right) = B\begin{pmatrix} x \\ f(x, y) \end{pmatrix} = \begin{pmatrix} g(x, f(x, y)) \\ f(x, y) \end{pmatrix}.$$

From this equation, we see that s and $f(x, y)$ are equal. Thus, the formula for s in terms of x and y gives the formula for $f(x, y)$: $f(x, y) = x\sin\phi + y\cos\phi$. Determining the formula for $g(u, v)$ is more complex. We know that, in terms of x and y, we can write $g(u, v) = x\cos\phi - y\sin\phi$. To write this in terms of u and v, we must solve for x and y in terms of u and v and substitute. Solving for x is easy, since we observed previously that $u = x$. Solving for y is slightly more difficult: $v = f(x, y) = x\sin\phi + y\cos\phi$, so $y = (v - x\sin\phi) / \cos\phi =$

Samsung/Google, EX1027, p. 841

$(v - u \sin \phi) / \cos \phi$. Substituting this result into the formula for $g(u, v)$ in terms of x and y, we get

$$g(u, v) = u \cos \phi - \frac{v - u \sin \phi}{\cos \phi} \sin \phi = u \sec \phi - v \tan \phi.$$

In summary, if we define

$$A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ x \sin \phi + y \cos \phi \end{pmatrix},$$

and

$$B\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} u \sec \phi - v \tan \phi \\ v \end{pmatrix},$$

then computing the composite gives

$$B\left(A\begin{pmatrix} x \\ y \end{pmatrix}\right) = \begin{pmatrix} x \cos \phi - y \sin \phi \\ x \sin \phi + y \cos \phi \end{pmatrix},$$

as desired.

To do this for a general transformation T, we must do exactly the same work. If

$$T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} t_1(x, y) \\ t_2(x, y) \end{pmatrix},$$

then we define $u = x$ and $v = f(x, y) = t_2(x, y)$. To define $g(u, v)$, we need to solve for y in terms of u and v, using these definitions—that is, to find a function h such that $(u, v) = (x, t_2(x, y))$ is equivalent to $(x, y) = (u, h(u, v))$. When we have found h, the formula for $g(u, v)$ is just $g(u, v) = t_2(u, h(u, v))$.

The difficult part of the process is finding h. In fact, in our example, $h(u, v) = (v - u \sin \phi) / \cos \phi$, which is undefined if $\cos \phi = 0$—that is, if $\phi = 90°$ or $270°$—so that finding h may be impossible. Fortunately, rotating by 90° is very easy (just map (x, y) to $(-y, x)$), so that this is not a problem. In fact, we shall see that, to rotate nearly 90°, it is better to rotate the full 90° and then to rotate a small amount back; thus, to rotate 87°, we would rotate 90° and then −3°. Algebraically, there is no difference between the two maps; at the pixel level, however, where filtering is involved, the difference is significant.

A rotation can also be broken into three transformations so as to avoid this *bottleneck* problem [PAET86; TANA86; WOLB90]. The decomposition for a rotation by ϕ is

$$\begin{bmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{bmatrix} = \begin{bmatrix} 1 & -\tan \phi/2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \sin \phi & 1 \end{bmatrix} \begin{bmatrix} 1 & -\tan \phi/2 \\ 0 & 1 \end{bmatrix}.$$

Note that each transformation involves a computation with one multiplication and one addition. Also, when $\phi > 90°$, we can do the rotation by first rotating by 180° and then by $180° - \phi$, so that the argument of the tangent function is never greater than 45°.[7]

---

[7]The tangent function is well behaved for angles near 0°, but has singularities at ±90°. Evaluating it for angles near 0° is therefore preferable.

Samsung/Google, EX1027, p. 842

To show that the multipass technique is not limited to rotations, let us factor a different map, which distorts a square into a trapezoid. (Such maps arise in the perspective transformations described in Chapter 6.) As an example, we take

$$T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x/(y+1) \\ y/(y+1) \end{pmatrix}.$$

Just as before, we wish to find functions

$$A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ f(x, y) \end{pmatrix} \quad \text{and} \quad B\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} g(u, v) \\ v \end{pmatrix}$$

such that $B\left(A\begin{pmatrix} x \\ y \end{pmatrix}\right) = T\begin{pmatrix} x \\ y \end{pmatrix}$. In this case, $v = f(x, y) = t_2(x, y) = y/(y+1)$. We need to find $g(u, v)$ so that $g(u, v) = x/(y+1)$. Solving the equation of f for y, we get $y = -v/(v-1)$. Thus (recalling that $u = x$), we can write $g(u, v) = u/(-v/(v-1)+1) = u/(-1/(v-1)) = u(1-v)$. Our two passes become

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} x \\ y/(y+1) \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} r \\ s \end{pmatrix} = \begin{pmatrix} u(1-v) \\ v \end{pmatrix}.$$

You should check that the composition of these transformations is really the original transformation T.

The technique has been generalized to handle other maps by Smith and colleagues [SMIT87]. Translation, rotation, scaling, and shearing all work easily. In addition, Smith considers functions of the form

$$T(x, y) = S(m(x)\, h_1(y),\ m(x)\, h_2(y)),$$

where S is a standard computer graphics transform—that is, a transformation of the plane by translation, scaling, rotation, and perspective transformations—and $m(x)$, $h_1(y)$ and $h_2(y)$ are arbitrary. He also considers maps T whose component functions $t_1(x, y)$ and $t_2(x, y)$ are bicubic functions of x and y, under the special hypothesis that T is injective (i.e., no two (x, y) points map to the same (r, s) point).

17.5.2 Generating Transformed Images with Filtering

When we transform an image by a row-preserving (or column-preserving) transformation, the source pixels are likely not to map exactly to the target pixels. For example, the pixels in a row might all be translated by $3\frac{1}{2}$ pixels to the right. In this case, we must compute values for the target pixels by taking combinations of the source pixels. What we are doing, in effect, is considering the values of the source pixels as samples of a function on a real line (the row); the values at the target pixels will be different samples of this same function. Hence, the process is called *resampling*.

The theoretically ideal resampling process is to take, for a given target pixel, a weighted average of the source pixels whose transformed positions are near it. The weights associated with each source pixel should be $\mathrm{sinc}(kd)$, where d is the distance from the transformed source pixel to the target pixel and k is some constant. Unfortunately, this requires that every source pixel in a row contribute to every target pixel. As usual, we can instead work

Samsung/Google, EX1027, p. 843



Fig. 17.14 The pixels that contribute to an output pixel in a two-pass rotation form a small area in (x, y) space. (a) A single pixel in (r, s) space; (b) The horizontal span of pixels in (u, v) space that contribute to the value of that pixel; (c) The pixels in (x, y) space that contribute to the values of the pixels in (u, v) space.

with various approximations to the sinc filter. The simplest is a box filter: Each source pixel is assumed to represent an interval in its row, and the endpoints of this interval are transformed. The contribution to the target pixel is the overlap of this transformed interval with the target pixel's interval multiplied by the value of the source pixel.

Using this method, each target pixel has a value that is a weighted average of a short span of source pixels (the length of the span depends on the exact transformation). In a two-pass rotation, a pixel in (r, s)-space has a value that is a weighted average of a horizontal span of (u, v)-pixels. Figure 17.14(a) shows a pixel in (r, s) space, and (b) shows the span of pixels in (u, v) space that contribute to the value of that pixel. Each of these (u, v) pixels, however, has a value that is an average of a vertical span of pixels in (x, y) space. Figure 17.14(c) shows these pixels in (x, y) space. Notice that the vertical spans in (x, y) space form a rhombus rather than a square, since the transformation from (x, y) to (u, v) is a shearing transformation.

We know from Chapter 14 that the pixels contributing to an output pixel really ought to form a circular shape[8] (i.e., the filter should be radially symmetric). If the rhombus is too different from a square, the filtering will begin to degenerate and will produce bad results. The result of such a separation of the filtering process into two component filters is discussed further in [MITC88].

In addition, we want to avoid the bottlenecking problem described previously, where many pixels in the source for a transformation contribute to each output pixel. In the case of a shear-and-scale operation, this occurs when the scale factor is small.

Thus, in doing two-pass rotations (or any other multipass transformation) we want to avoid extreme shearing or bottlenecking in our transformations. This is why rotating by 90° and then by $-3°$ is superior to rotating by 87°. In general, when we are constructing a two-pass transform, we can transform by rows and then columns, or by columns and then rows, or can rotate 90° before doing either of these. According to Smith, one of these approaches appears always to resolve the bottlenecking problem, at least for standard operations such as translation, rotation, scaling, and shearing [SMIT87]. Nonetheless, there are more general transformations where this technique may not succeed: If one portion of an image is rotated 90° while some other stays fixed (imagine bending a long thin

[8]This is particular to the case of a rotation. For a general transformation, the source pixels contributing to an output pixel should consist of those pixels that are transformed into a small disk about the target pixel; these pixels may or may not constitute a disk in the source image.

Samsung/Google, EX1027, p. 844

rectangle into a quarter-circle shape), then there is certain to be bottlenecking at some point, no matter what the order in which the transformations are applied.

Wolberg and Boult [WOLB89] have developed a technique in which two simultaneous versions of a multipass transform are done at once. The original image is first transformed by a row-preserving map and a column-preserving map, and then also is transformed by a 90° rotation and a different pair of maps. For both transformation sequences, the method records the amount of bottlenecking present at each pixel.

Thus, each output pixel can be computed in two different ways. Wolberg and Boult select, for each pixel, the route that has less bottlenecking, so that some portions of the image may be row–column transformed, whereas others are column-row transformed. Since the two sets of transformations can be performed simultaneously in parallel processors, this technique is ideally suited to implementation in hardware.

17.5.3 Evaluating Transformation Methods

There are several criteria for judging image-transformation algorithms. Filtering theory tells us that an image can be reconstructed from its samples, provided the original image had no high-frequency components. Indeed, from *any* set of samples, one can reconstruct *some* image with no high-frequency components. If the original image had no high frequencies, then we get the original back; if it did contain high frequencies, then the sampled image contained aliases, and the reconstructed image is likely to differ from the original (see Exercise 17.6).

So how should we judge a transformation algorithm? Ideally, a transformation algorithm would have the following properties:

- Translation by a zero vector should be the identity
- A sequence of translations should have the same effect as a single, composite translation
- Scaling up by a factor of $\lambda > 1$ and then scaling down by $1/\lambda$ should be the identity transformation
- Rotating by any sequence of angles totaling 360° should be the identity transformation.

Many workable algorithms clearly fail to satisfy any of these criteria. Weiman's algorithm fails on all but the fourth criterion. Feibush, Levoy, and Cook's algorithm fails on the first if a filter more than 1 pixel wide is used. Even Catmull and Smith's two-pass algorithm fails on all four criteria.

None of this is surprising. To resample an image, we ought to reconstruct it faithfully from its samples by convolving with a sinc filter. Thus, each new pixel ought to be a weighted average of *all* the pixels in the original image. Since all the methods are sensible enough to use filters of finite extent (for the sake of computational speed), all of them end up blurring the images.

There are many image-transformation methods not covered here. They each have advantages and disadvantages, mostly in the form of time–space tradeoffs. These are described in detail in an excellent survey by Heckbert [HECK86b].

Samsung/Google, EX1027, p. 845

17.6 IMAGE COMPOSITING

In this section, we discuss compositing of images—that is, combining images to create new images. Porter and Duff [PORT84] suggest that compositing is a good way to produce images in general, since it is fairly easy to do, whereas rendering the individual portions of the image may be difficult. With compositing, if one portion of the image needs alteration, the whole image does not need to be regenerated. Even more important, if some portions of an image are not rendered but have been optically scanned into memory instead, compositing may be the only way to incorporate them in the image.

We describe compositing using the α channel in Section 17.6.1, compositing using frame-buffer hardware in Section 17.6.2, the artificial generation of α values in Section 17.6.3, and an interface for image assembly in Section 17.6.4.

17.6.1 α-Channel Compositing

What sort of operations can be done in compositing? The value of each pixel in the composited image is computed from the component images in some fashion. In an *overlay*, the pixels of the foreground image must be given *transparency values* as well as whatever other values they may have (typically RGB or other color information). A pixel's value in the composited image is taken from the background image unless the foreground image has a nontransparent value at that point, in which case the value is taken from the foreground image. In a *blending* of two images, the resulting pixel value is a linear combination of the values of the two component pixels. In this section, we describe the Porter–Duff mechanism for compositing images using such combinations and transparency values [PORT84].

Suppose we have two images, one of a red polygon and one of a blue polygon, each on a transparent background. If we overlay the two images with the red polygon in front, then, at interior points of the front polygon, only the color red is visible. At points outside the front polygon but inside the back polygon, only blue is visible. But what about a pixel lying on the edge of the front polygon but inside the back polygon (see Fig. 17.15)? Here, the front polygon covers only part of the area of the pixel. If we color it red only, aliasing artifacts will result. On the other hand, if we know that the front polygon covers 70 percent

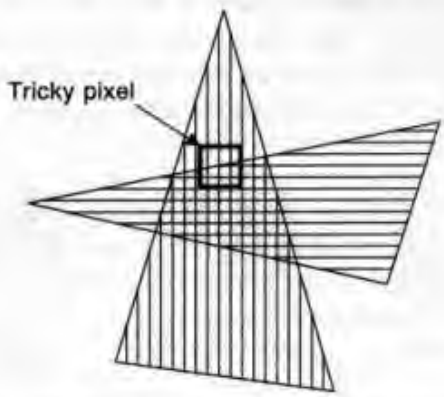


Fig. 17.15 Compositing operations near an edge: How do we color the pixel?

Samsung/Google, EX1027, p. 846

of the pixel, we can make the composited pixel 70 percent red and 30 percent blue and get a much more attractive result.

Suppose that as an image is produced, coverage information is recorded: The color associated with each pixel in the image is given an α *value* representing the coverage of the pixel. For an image that is to become the foreground element of a composited image, many of the pixels are registered as having coverage zero (they are transparent); the remainder, which constitute the important content of the foreground image, have larger coverage values (usually one).

To do compositing in a reasonable fashion, we need this α information at each pixel of the images being composited. We therefore assume that, along with the RGB values of an image, we also have an α value encoding the coverage of each pixel. This collection of α values is often called the α *channel* (see Section 17.7). Some types of renderers generate this coverage information easily, but it may be more difficult to generate for images that have been scanned into memory. We discuss this problem briefly in Section 17.6.3.

How do α values combine? Suppose we have a red polygon covering one-third of the area of a pixel, and a blue polygon that, taken separately, covers one-half of the area of the pixel. How much of the first polygon is covered by the second? As Fig. 17.16 shows, the first polygon can be completely covered, partly covered, or not covered at all. But suppose we know nothing more than the coverage information given. What is a *reasonable* guess for the amount of the first polygon covered by the second? Let us suppose that the area covered by the first is randomly distributed through the pixel area, and that the same is true of the second. Then any tiny spot's chance of being in the red polygon is $\frac{1}{3}$, and its chance of being in the blue polygon is $\frac{1}{2}$, so its chance of being in both is $\frac{1}{6}$. Notice that this value is exactly one-half of $\frac{1}{3}$, so that exactly one-half of the first polygon is covered by the second. This will be our general assumption: The area covered is distributed randomly across the pixel, so that the fraction of the first polygon covered by the second (within a particular pixel) is the same as the fraction of the whole pixel covered by the second polygon. The consequence of this assumption in practice is that compositing images with very fine detail that is parallel in the two images can have bad results. Porter and Duff report, however, that they have had no noticeable problems of this sort [PORT84].

Now, how do we compute the color of the pixel resulting from a 60—40 blend of these 2 pixels? Since the pixel is one-third covered by a color that, if it totally covered the pixel, would generate light of (1,0,0), the color's contribution to the light of the pixel is $(\frac{1}{3})(1,0,0)$. We want to take 60 percent of this and combine it with 40 percent of the other. We thus combine the RGB triple for the red pixel, (1, 0, 0), and the RGB triple for the

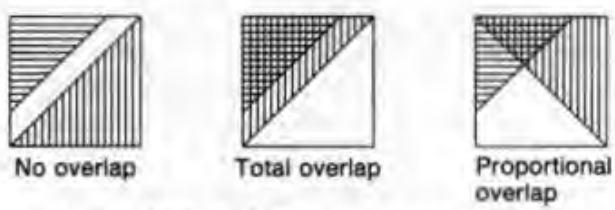


Fig. 17.16 The ways in which polygons can overlap within a pixel. In image composition, the first two cases are considered exceptional; the third is treated as the rule.

Samsung/Google, EX1027, p. 847

TABLE 17.1 AREAS AND POSSIBLE COLORS FOR REGIONS OF OVERLAP IN COMPOSITING

Region	Area	Possible colors
neither	$(1 - \alpha_A)(1 - \alpha_B)$	0
A alone	$\alpha_A(1 - \alpha_B)$	0, A
B alone	$\alpha_B(1 - \alpha_A)$	0, B
both	$\alpha_A\alpha_B$	0, A, B

blue pixel, (0, 0, 1), as follows: We say that

$$0.6\ (\tfrac{1}{3})(1,0,0) + 0.4\ (\tfrac{1}{2})(0, 0, 1) = (0.2, 0, 0.2)$$

is the resulting color.

Note that, whenever we combine 2 pixels, we use the product of the α value and the color of each pixel. This suggests that, when we store an image (within a compositing program), we should store not (R, G, B, α), but rather (αR, αG, αB, α) for each pixel, thus saving ourselves the trouble of performing the multiplications each time we use an image. Henceforth, when we refer to an RGBα value for a pixel, we mean exactly this. Thus, it should always be true that the R, G and B components of a pixel are no greater than that pixel's α component.[9] (In rare cases, we may want to consider pixels for which this condition is violated. Such pixels are effectively luminescent.)

Suppose now that two images, A and B, are to be combined, and suppose that we are looking at pixel P. If the α value of P in image A is α_A and the α value of P in image B is α_B, we can ask what fraction of the resulting pixel is covered by A only, what part by B only, what part by both, and what part by neither.

We have already assumed that the amount covered by both is $\alpha_A\alpha_B$. This means that $\alpha_A - \alpha_A\alpha_B$ is covered just by A, and $\alpha_B - \alpha_A\alpha_B$ is covered just by B. The amount left is 1 minus the sum of these three, which reduces algebraically to $(1 - \alpha_A)(1 - \alpha_B)$. In the composited image, the area that was covered by A alone might end up with color A or no color, and similarly for color B. The area that was covered neither by A nor B should end up with no color, and the area covered by both might end up with no color, color A, or color B. Table 17.1 lays out these possibilities.

How many possible ways of coloring this pixel are there? With three choices for the both-colors region, two for each of the single-color regions, and one for the blank region, we have 12 possibilities. Figure 17.17 lists these possibilities.

Of these operations, the ones that make the most intuitive sense (and are most often used) are A **over** B, A **in** B and A **held out by** B, which denote respectively the result of hiding B behind A, showing only the part of A that lies within B (useful if B is a picture of a hole), and showing only the part of A outside of B (if B represents the frame of a window, this is the part that shows through the pane of the window).

[9] In a fixed-point scheme for representing the colors, pixels whose α value is small have a more discrete realm of colors than do those whose α value is 1. Porter and Duff say that this lack of resolution in the color spectrum for pixels with small α values has been of no consequence to them.

Samsung/Google, EX1027, p. 848

operation	quadruple	diagram	F_A	F_B
clear	(0, 0, 0, 0)		0	0
A	(0, *A*, 0, *A*)		1	0
B	(0, 0, *B*, *B*)		0	1
A **over** *B*	(0, *A*, *B*, *A*)		1	$1-\alpha_A$
B **over** *A*	(0, *A*, *B*, *B*)		$1-\alpha_B$	1
A **in** *B*	(0, 0, 0, *A*)		α_B	0
B **in** *A*	(0, 0, 0, *B*)		0	α_A
A **held out by** *B*	(0, *A*, 0, 0)		$1-\alpha_B$	0
B **held out by** *A*	(0, 0, *B*, 0)		0	$1-\alpha_A$
A **atop** *B*	(0, 0, *B*, *A*)		α_B	$1-\alpha_A$
B **atop** *A*	(0, *A*, 0, *B*)		$1-\alpha_B$	α_A
A **xor** *B*	(0, *A*, *B*, 0)		$1-\alpha_B$	$1-\alpha_A$

Fig. 17.17 The possibilities for compositing operations. The quadruple indicates the colors for the "neither," "A," "B," and "both" regions. Adapted from [PORT84]. (Courtesy of Thomas Porter and Tom Duff.)

In each case we can compute, as before, how much of each color should survive in the result. For example, in *A* **over** *B*, all the color from image *A* survives, while only a fraction $(1 - \alpha_A)$ of the color from *B* survives. The total color in the result is $c_A + (1 - \alpha_A)c_B$, where c_A denotes the color from image *A* (with α_A already multiplied in).

In general, if F_A denotes the fraction of the pixel from image *A* that still shows, and similarly for F_B, then the resulting color will be $F_A c_A + F_B c_B$. We must also compute the resulting α value: If fraction F_A of *A* is showing, and the original contribution of *A* is α_A, then the new contribution of *A* to the coverage is $F_A \alpha_A$. The same goes for *B*; hence, the total coverage for the new pixel is $F_A \alpha_A + F_B \alpha_B$.

Samsung/Google, EX1027, p. 849

A few unary operations can be performed on images. For example, the **darken** operation is defined by

$$\textbf{darken}(A, \rho) := (\rho R_A, \rho G_A, \rho B_A, \alpha_A) \qquad 0 \le \rho \le 1,$$

which effectively darkens a pixel while maintaining the same coverage. In contrast, the **fade** operator acts by

$$\textbf{fade}(A, \delta) := (\delta R_A, \delta G_A, \delta B_A, \delta \alpha_A) \qquad 0 \le \delta \le 1$$

which causes a pixel to become more transparent while maintaining its color (the color components are multiplied by δ because of the requirement that the colors must always be premultiplied by the α value).

A natural third operator (essentially a composite of these two) is the **opaque** operator, which acts on the α channel alone and is defined by

$$\textbf{opaque}(A, \omega) := (R_A, G_A, B_A, \omega \alpha_A).$$

As ω varies between 0 and 1, the coverage of the background by the pixel changes. Of course, if ω is made small, one of the color components may end up larger than the α component. For example, if ω is zero, then we get a pixel with color but a zero α component. Such a pixel cannot obscure anything, but can contribute light to a composited pixel; hence, it is called *luminous*. The possibility that color components can be larger than the α component requires that we clip the colors to the interval [0, 1] when reconstructing the true color (to reconstruct the color, we compute $(R/\alpha, G/\alpha, B/\alpha)$, and then clamp all three numbers to a maximum of 1).

One last binary operator, **plus,** is useful. In this operation, the color components and α components are added. Thus, to fade smoothly from image A to image B, we use

$$\textbf{fade}(A, t) \textbf{ plus } \textbf{fade}(B, 1 - t),$$

and let t vary from 1 to 0. Judicious combinations of these operations let us combine pictures in a great variety of ways.

We should examine the consequences of the assumptions made at the start: If a pixel is covered a fraction F_1 by one polygon and a fraction F_2 by another, then the first polygon (within the pixel) is covered a fraction F_2 by the second. If the two polygons happen to overlap in a reasonable way, this works fine, as in Figure 17.17. But if they happen to cross the pixel as parallel stripes, as in Fig. 17.18, then the assumption is invalid. When geometric entities are being composited, this problem is not particularly likely to occur.



Fig. 17.18 A case where the overlap assumptions fail in compositing.

Samsung/Google, EX1027, p. 850

However, when repeated instances of an image are composited atop one another, it may happen frequently.

To illustrate the use of compositing, we describe the composition of the frame from the Genesis effect described in Chapter 20, and shown in Color Plate IV.14. In this case, there are four images composited: *FFire*, the particle systems in front of the planet; *BFire*, the particle systems in back of the planet; *Planet*, the planet itself; and *Stars*, the background star field. The composite expression for the image is [PORT84, p. 259]:

(*FFire* **plus** (*BFire* **held out by** *Planet*)) **over darken** (*Planet*, 0.8) **over** *Stars*.

The planet is used to mask out the parts of the particle systems behind it, and the results are added to the front particle systems. These are composited over a slightly darkened planet (so that the particles obscure the underlying planet), and the result is placed over the background star field.

17.6.2 Alternate Compositing Methods

Let us consider two other mechanisms for compositing. The first of these is to composite images in compressed form (e.g., bitmaps that are stored with 8 pixels per byte), by doing very simple operations (*A* **over** *B*, *A* **and** *B*, *A* **xor** *B*) on the compressed forms. These operations are simple to implement: For any two possible bytes, we use a bitwise operation in hardware to combine them. In a more sophisticated version, the algorithm can be extended to handle this sort of compositing on run-length–encoded bitmaps.

The other compositing technique uses the frame-buffer hardware to implement compositing. Consider a simple example. We usually think of a 3-bit-per-pixel bitmap as containing a single image with 3 bits per pixel. However, we can also think of it as containing two images, one with 2 bits per pixel and the other with 1 bit per pixel, or as three separate images, each with 1 bit per pixel. In any case, the look-up table is used to select or combine the separate images to form a single composite image displayed on the view surface. For instance, to display only image 2, the image defined by the high-order bit of each pixel value, we load the table as shown in Fig. 17.19. To display image 0, the image

Entry number (decimal)	Entry number (binary)	Contents of look-up table (decimal)
0	0 0 0	0
1	0 0 1	0
2	0 1 0	0
3	0 1 1	0
4	1 0 0	7
5	1 0 1	7
6	1 1 0	7
7	1 1 1	7
	Image 2 Image 1 Image 0	0 = black 7 = white

Fig. 17.19 Look-up table to display an image defined by the high-order bit of each pixel.

Samsung/Google, EX1027, p. 851

Entry number (decimal)	Entry number (binary)	Contents of look-up table (decimal)
0	0 0 0	0
1	0 0 1	2
2	0 1 0	2
3	0 1 1	4
4	1 0 0	2
5	1 0 1	4
6	1 1 0	4
7	1 1 1	6
	Image 2 Image 1 Image 0	

Fig. 17.20 Look-up table to display a sum of three 1-bit images.

defined by the low-order bit of each pixel, we load the table with a 7 in those locations for which the low-order bit is 1, and with a 0 for the other locations: 0, 7, 0, 7, 0, 7, 0, 7. If the displayed image is to be the sum of the three images and if each image that is "on" at a pixel is to contribute two units of intensity, then we load the table as shown in Fig. 17.20. If 1 of the 3 pixel bits is on, a 2 is placed in the table; if 2 of 3 are on, a 4; and if 3 of 3 are on, a 6.

As another example, think of each image as being defined on parallel planes, as in Fig. 17.21. The plane of image 2 is closest to the viewer; the plane of image 0 is farthest away. Thus, image 2 obscures both images 0 and 1, whereas image 1 obscures only image 0. This priority can be reflected in the look-up table, as shown in Fig. 17.22. In this case, image 2 is displayed at intensity 7, image 1 at intensity 5, and image 0 at intensity 3, so that images "closer" to the viewer appear brighter than those farther away. Where no image is defined, intensity 0 is displayed.

Yet another possibility is to use the look-up table to store a weighted sum of the intensities of two images, creating a double-exposure effect. If the weight applied to one image is decreased over time as the weight applied to the other is increased, we achieve the fade-out, fade-in effect called a *lap-dissolve*. When colored images are used, the colors displayed during the fade sequence depend on the color space in which the weighted sum is calculated (see Chapter 13).

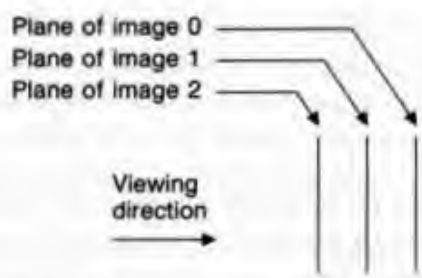


Fig. 17.21 Relation of three images to the viewing direction.

Samsung/Google, EX1027, p. 852

Entry number (decimal)	Entry number (binary)	Contents of look-up table (decimal)
0	0 0 0	0 no image present
1	0 0 1	3 image 0 visible
2	0 1 0	5 image 1 visible
3	0 1 1	5 image 1 visible
4	1 0 0	7 image 2 visible
5	1 0 1	7 image 2 visible
6	1 1 0	7 image 2 visible
7	1 1 1	7 image 2 visible
	Image 2 Image 1 Image 0	

Fig. 17.22 Look-up table to assign priorities to three 1-bit images.

Deciding how to load the look-up table to achieve a particular result can be tedious, especially if many images are used and the table has many entries. The Image Composition Language (ICL) [FOLE87c] allows the programmer to declare images (made up of one or more bit planes) as variables. The image to be displayed is described by a composition expression consisting of variables combined by arithmetic, relational, and conditional operations. The lap-dissolve is specified in ICL with the expression

$$newImage * t + oldImage * (1 - t),$$

in which the variables *oldImage* and *newImage* are images in the bit plane and t is a scalar variable varying from 0 to 1. The following composition expression adds red (the triplets in braces are RGB color specifications) to those values of an image c in the range [0.6, 0.8] and green to those values in the range [0.3, 0.5]:

```
if (c < 0.8) and (c > 0.6) then c + {1,0,0}
    else if (c < 0.5) and (c > 0.3) then c + {0,1,0}
    else c
endif endif
```

This short composition expression replaces a code segment that has a considerably larger number of lines.

17.6.3 Generating α Values with Fill Mechanisms

In the preceding section, we described compositing images that come equipped with an α channel. What if an image is produced by scanning of a photograph or is provided by some other source lacking this information? Can it still be composited? If we can generate an α channel for the image, we can use that channel for compositing. Even if we merely assign an α value of zero to black pixels and an α value of 1 to all others, we can use the preceding algorithms, although ragged-edge problems will generally arise.

Recall from Chapters 3 and 4 that there are various ways to *fill* regions of an image with new values. If we use a fill algorithm to alter not the color of a pixel but its α value, we can assign α values to various regions in the image. If the colors in the image represent foreground and background (e.g., a picture of a person standing in front of a white wall),

Samsung/Google, EX1027, p. 853

we can choose to fill the background region with its original color but assign it a reduced α value. Fishkin and Barsky give an algorithm for recognizing regions that consist of pixels that are either entirely or partially made up of some color [FISH84]. For the details of this algorithm, see Section 19.5.3.

Of course, applying a seed-fill algorithm (see Section 19.5.2) is bound to fail if the background is not a single connected piece. If we attempt to correct this difficulty by applying the Fishkin–Barsky criterion for similarity to the background to every pixel in the image, items in the foreground whose colors are close to the background color are treated incorrectly. (Imagine the person in our example, standing in front of a white wall. Is the white writing on his green T-shirt part of the background?) If we try to seed fill each separate piece of background, the task may be hopeless. (Imagine our person again and suppose that the background shows through his curly hair. There may be thousands of background regions.) Nonetheless, a soft-seedfill of the background to determine new α values makes a good preliminary step, and for simple images can improve substantially on the approach of assigning α values of 0.0 or 1.0 to every pixel.

17.6.4 An Interface for Image Assembly

How are the tools used for compositing and applying geometric transformations of images used in practice? An image like Color Plate IV.19 is described by a complex collection of operations. The compositing operations such as A **over** B can be described by a tree structure, where the leaf nodes are images and the internal nodes are operators with operands as child nodes. But before the images can be composited, they must be placed correctly. This requirement suggests an *image-assembly tree* structure, in which each internal node is either an image transformation or a compositing operation, and each leaf node is an image.

Such an image-assembly structure can be implemented in a convenient user interface, in which the user adds nodes or moves pieces of the tree with a mouse, and places a marker at some node to view the image described by that node and its children. This view can be structural, merely showing the relative positions and sizes of the child images within the parent. On a workstation with limited color capabilities, the view can be a dithered version of the true-color image; on a sophisticated workstation, it can be a full-sized version of the image in full color. The structural view can be extremely useful, since the user can edit the geometric transformations in this view by mouse dragging, eliminating the need to type exact coordinates for geometric transformations; of course, the ability to enter precise coordinates is also essential.

An image assembler of this sort has been developed by Kauffman [KAUF88a]. Far more sophisticated image assemblers form the core of the video processors that generate many of the special effects seen on television.

17.7 MECHANISMS FOR IMAGE STORAGE

When we store an image, we are storing a 2D array of *values*, where each value represents the data associated with a pixel in the image. For a bitmap, this value is a binary digit. For a color image, the value may be a collection of three numbers representing the intensities of the red, green, and blue components of the color at that pixel, or three numbers that are

Samsung/Google, EX1027, p. 854

indices into tables of red, green, and blue intensities, or a single number that is an index into a table of color triples, or an index into any of a number of other data structures that can represent a color, including CIE or XYZ color systems, or even a collection of four or five spectral samples for each color.

In addition, each pixel may have other information associated with it, such as the z-buffer value of the pixel, a triple of numbers indicating the normal to the surface drawn at that pixel, or the α-channel information. Thus, we may consider an image as consisting of a collection of *channels*, each of which gives some single piece of information about the pixels in the image. Thus, we speak of the *red*, *green*, and *blue channels* of an image.

Although this idea might seem contrary to good programming practice, in which we learn to collect information associated with a single object into a single data structure, it often helps to separate the channels for convenience in storage. However, some methods of image compression do treat the image as a 2D array, such as the quadtree and fractal encoding schemes described later, so separation into channels is inappropriate.

Before discussing algorithms for storing images as channels or arrays, we describe two important methods for storing pictures: use of the *metafile* and use of *application-dependent data*. Neither of these is, strictly speaking, an image format, but each is a mechanism for conveying the information that is represented in an image.

If an image is produced by a sequence of calls to some collection of routines, a metafile stores this sequence of calls rather than the image that was generated. This sequence of calls may be far more compact than the image itself (an image of the Japanese flag can be produced by one call to a rectangle-drawing routine and one call to a circle-drawing routine, but could take several MB to store as RGB triples). If the routines are sufficiently simple or are implemented in hardware, redisplaying a metafile image may be faster than redisplaying a pixel image. The term *metafile* is also used to refer to a device-independent description of a standardized data structure, such as the PHIGS data structure described in Chapter 7. To store an image in such a metafile, we traverse the current data structure and record the data structure in some device-independent fashion for redisplay later. This description may be not a sequence of function calls, but instead a textual transcription of some hierarchical structure.

The second storage scheme entails application-dependent data. If an application displays a particular class of images, it may be convenient to record the data from which these images were created, or even differences between the data and some standard set of data. If the images are all head-on views of human faces described as polygons, it may be simpler to store just a list of those polygons whose positions are different from their position in some standard facial image (and their new positions, of course). A more extreme version of this sort of condensation of information has been use in the Talking Heads project at the MIT Media Lab, in which only the positions of eyeballs, lips, and other high-level features are stored [BOLT84]. At this point, image description becomes more of a scene description, and properly belongs to the domain of modeling, rather than to that of image storage.

17.7.1 Storing Image Data

Now let us consider how to store the sort of image that consists of several channels of data. If our displays expect to be given information about an image in the form of RGB triples, it

Samsung/Google, EX1027, p. 855

may be most convenient to store the image as RGB triples. But if space is at a premium, as is often the case, then it may be worth trying to compress the channels in some way. Approaches to compression must be weighed against the cost of decompression: The more sophisticated the compression technique, the more likely decompression is to be expensive. Although all of these techniques apply equally well to any channel of information, our discussion will be couched in terms of color channels, since these are the ones most often present in images (z-buffer, normal vector, and other information being optional).

If an image has few colors and each color occurs many times (as in an image of a newspaper, in which there may be only black, dark gray, light gray, and white), it may be worthwhile to make a table of colors that occur (here, the table would have only four entries), and then to make a *single* channel that is an index into this color table. In our newspaper example, this single channel would need only 2 bits of information per pixel, rather than perhaps 8 bits per color per pixel; the resulting image is compressed by a factor of 12. In pictures with more colors, the savings are less substantial; in the extreme case where each pixel in the image is a different color, the look-up table is as large as the image would have been if stored as RGB triples, and the indices into the look-up table take even more space. Roughly speaking, indexing into a look-up table begins to be worthwhile if the number of colors is less than one-half the number of pixels. (Of course, if the hardware for displaying the image works by using look-up tables as well, it may be easier and faster to store the image in this fashion than as RGB triples. Typically, such hardware provides a modest space for the look-up table, about 8 to 12 bits per pixel.)

This single-channel approach still requires at least one piece of information per pixel. If the image has a great deal of repetition, it may be possible to compress it further by *run-length encoding* a channel. Run-length encoding consists of giving a count and a value, where the count indicates the number of times the value is to be repeated. The design of the Utah Raster Toolkit [PETE86] includes a number of improvements on this basic idea. For instance, the count, n, is an 8-bit *signed* integer (with values -128 through 127): a negative count indicates that n pixels' worth of unencoded data follow; a nonnegative count indicates that the next piece of information is the value to be used for $n + 1$ pixels. Further improvements might include reserving certain negative values for special meanings: -128 might indicate that the next few bytes of information give a scan line and position to which to jump (in order to skip the recording of large areas of background color), and -127 might be reserved to indicate a jump to the start of a specified scan line. Such a naive run-length–encoding scheme at worst adds 1 byte for every 126 values (a -126 indicating that 126 pixels worth of unencoded data follow), a cost of about 0.25 percent for an image with 8 bits per pixel for each of red, green, and blue. In the best case, an image in which all pixels have the same value, the compression would be by a factor of about 100: 128 pixels' worth of values compress to 1 pixel's worth (24 bits, in this example), but the count byte adds another 8 bits.

There are other clever formats for compressing channels. For example, we could store the value of each pixel from a bitmap in an integer (as a 0 or 1), but most of the bits in the integer would be wasted. Instead, we might store one pixel value in each bit (this is the origin of the term *bitmap*). If the image being represented contains regions filled with patterns whose width is a factor of 8, then we can perform a similar run-length encoding, in which the first byte gives a count, n, and the next byte gives a pattern to be repeated for the next $8n$ pixels of the bitmap. This method is less likely to generate savings than is the

Samsung/Google, EX1027, p. 856

ordinary run-length encoding for color images, since a block of 8 values must be repeated for any compression to take place.

Run-length encoding and other standard information-theory approaches such as Huffman encoding treat the image in channels, which can be imagined as linear arrays of values (although multiple channels can be considered a single large channel, so that we can run-length encode sets of RGB triples as well). Other methods treat the image as a 2D array of values, and hence can exploit any inter-row coherence. One of these techniques is based on the use of quadtrees.

The fundamental idea of the quadtree-based image description is that a region of an image may be fairly constant, and hence all pixels in the region can be treated as having the same value. Determining these near-constant regions is the core of the algorithm. This algorithm can be used either on a single component of the image, such as the 2D array of red values, or on the aggregate value associated with each pixel; for simplicity, we shall describe the algorithm for a single numerical component. The algorithm requires a mechanism for determining the mean value of the image in a region, and the extent of the deviations from the mean within a region.

The image is first considered as a whole. If the deviation from the mean in the image is sufficiently small (less than or equal to some nonnegative tolerance), then the image is reported as having a value equal to the mean, repeated over the entire image. (If the tolerance is set to zero, then the image really must be constant for this to occur.) If the deviation from the mean is not smaller than the tolerance, the mean of the image is recorded, the image is divided into quadrants, and the same algorithm is applied to each quadrant. The algorithm terminates because repeated subdivision of quadrants eventually breaks them into single-pixel regions, if necessary; for a single-pixel region, the deviation from the mean must be zero, and hence is less than or equal to any tolerance value.

We can improve the algorithm by recording not the mean of the image, but rather the means of the four quadrants whenever the image is subdivided, and the mean of the image when it is not. The advantage is that, when the image is redisplayed, if the quadtree is parsed breadth-first, the display may be constantly updated to show more and more refined images. The first image is four colored rectangles. Then, each rectangle is subdivided and its color is refined, and so on. In a system designed for scanning through a large number of images, this approach may be extremely convenient: after just a few bytes of information have been transmitted, a general sense of the image may begin to appear, and the user may choose to reject the image and to move on to the next one. This rapid detection of the sense of an image is especially useful if the images are transmitted over a low-bandwidth communications channel. Quadtree compression of images has been exploited by Knowlton, Sloan, and Tanimoto [KNOW80, SLOA79], and the algorithm has been further refined by Hill [HILL83]. Exercise 17.7 discusses other mechanisms for building a quadtree describing an image, some of which may be more efficient than the one described here.

17.7.2 Iterated Function Systems for Image Compression

A second image-compression algorithm is based on the notion of iterated function systems (IFSs). In this case, the compression factor can be extremely high, but the cost of

Samsung/Google, EX1027, p. 857

compressing the image tends to be large as well. The algorithm requires, for each image, that the user interactively solve a geometric problem, described later [BARN88a]. Also, like all nondestructive compression schemes, the pigeonhole principle[10] says that, if some images are compressed, others must be expanded by some modest amount (since there can be no one-to-one mapping of all n by k arrays to all p by q arrays where pq is less than nk). The advantage of the IFS technique is that images with substantial geometric regularity are the ones that are compressed, whereas those that look like noise are more likely to be expanded.

An *IFS code* is a finite collection of affine maps $\{w_1, \ldots, w_r\}$ of the plane to itself, together with a probability p_i associated with w_i. The maps must be *contractive*; that is, the distance between points must be reduced by the maps, on the average (the precise requirement is described in [BARN88a]). Recall that an affine map of the plane is given by a formula of the form

$$w\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}; \tag{17.1}$$

so it is entirely determined by six numbers, a, b, c, d, e, and f. Notice that these affine maps are just combinations of rotations, translations and scalings in the plane. The condition that they be contractive says that the scaling factors must be less than 1.

The next several pages give a rough description of how to produce a gray-scale image from an IFS code; the method is easily generalized to producing three gray-scale images from three IFS codes, that can then be used as the RGB components of a color image. ([BARN88a] uses a somewhat different scheme for encoding color.) This production of an image from an IFS code is essentially this decompression part of the IFS algorithm; we discuss it first.

Consider a rectangle, V, in the plane defining our image, and imagine V as divided into a rectangular grid whose subrectangles are V_{ij}, $i = 1, \ldots, n; j = 1, \ldots, k$. Choose a point (x_0, y_0) that remains fixed under one of the maps (say w_1, without loss of generality). We now proceed to apply the maps $w_1, \ldots, w_r$ in fairly random order (determined by the probabilities p_i), and watch where the point (x_0, y_0) is sent. We use the number of times it lands in V_{ij} for each i and j to determine the eventual brightness of pixel $[i, j]$ in the image. The pseudocode for this process is shown in Fig. 17.23.

Before the last step of this algorithm, each image$[i, j]$ entry indicates how often the starting point, in the course of being moved randomly by the affine maps, falls into the $[i, j]$th square of the image. For this number accurately to represent the *probability* of falling into the square over an infinite sequence of steps, the number of iterations of the algorithm must be very large: K should be a large multiple of the number of pixels in the image.

The effect of this algorithm is essentially to create a picture of the *attractor* of the IFS. The attractor is a set, A, with the property that, if all the affine maps are applied to A, and the results are combined, the result is A:

$$A = \bigcup_{i=1}^{r} w_i(A).$$

[10]The pigeonhole principle is the observation that, if more than m objects are placed in m boxes, then some box must contain more than one object.

Samsung/Google, EX1027, p. 858

```
void IFS (double image[MAX][MAX])
/* Given a collection of affine maps w_i, with associated probabilities p_i, */
/* which are global variables, generate a gray-scale image. */
{
    int x, y;                          /* A location in the plane */
    int i, j;                          /* Loop counters */
    int m;

    Initialize x and y to be a fixed point of w_0;
    Initialize image[i][j] to 0 for all i and j;
    for (i = 0; i < K; i++) {
        double r = Random (0, 1);      /* A random number 0 <= r <= 1 */
        double total = p[0];           /* Probability tally */
        int k = 0;
        while (total < r) {
            k++;
            total += p[k];
        }
        apply (k, x, y);               /* Apply w_k to the point (x, y) */
        for (each i, j pair)
            if (LiesIn (x, y, i, j))   /* TRUE if (x, y) is in V_ij */
                image[i][j]++;
    }

    m = maximum of all image[i][j] entries;
    for (each (i, j) pair)
        image[i][j] /= m;
}  /* IFS */
```

Fig. 17.23 The iterated function system rendering algorithm.

The set A consists of the places to which (x_0, y_0) is sent in the course of iterating the maps. Some places are visited more often than others are, and the likelihood of a region being visited defines a probability measure on the set. The measure associated with a small region Q is p if a point, in the course of infinitely many iterations of the maps, spends a fraction p of its time in the region Q. It is this probability measure that we are using to associate values to pixels.

Since K must be so large, the time spent in reconstructing an image from an IFS code is substantial (although the process is highly parallelizable). What about creating an IFS code from an image? To do so, we must find a collection of affine maps of the plane to itself with the property that, after the affine maps have been applied to the original image, the union of the results "looks like" the original image. Figure 17.24 shows how to make a leaf by

Samsung/Google, EX1027, p. 859



Fig. 17.24 A leaf made as a collage. (© Michael Barnsley, *Fractals Everywhere*, Academic Press.)

creating four (slightly overlapping) smaller versions of the leaf and making a collage from them.

The collage theorem [BARN88a] guarantees that any IFS that uses these affine maps has an attractor that looks like the original image. Choosing the probability associated with w_j changes the brightness of the portion of the image coming from w_j. Still, to compress an image into an IFS code, a user must find a way to recreate the original image as a union of repeated subimages, each of which is an affine transform of the original. This is the previously mentioned geometric problem to be solved by the user. Barnsley has announced that this process can be automated [BARN88b]; until a mechanism for doing so is made public, the technique is hardly usable for compressing large numbers of images. It *is* usable for modeling interesting objects, however (see Chapter 20). Color Plate IV.1 shows an entire forest modeled with an IFS.

17.7.3 Image Attributes

When we store an image in the conventional manner as a collection of channels, we certainly must store information about each pixel—namely, the value of each channel at each pixel. Other information may be associated with the image as a whole, such as width and height, and any image-description format must include this kind of information as well. It is insufficient to allocate a few bytes at the start of the image description for width and height; experience has shown that other image attributes will arise. Typical examples are the space required for look-up tables, the depth of the image (the number of bitplanes it occupies), the number of channels that follow, and the name of the creator of the image. For accurate color reproduction of images on other devices, we may also want to record reference spectra for the pure red, green, and blue colors used to make the image, and some indication of what gamma correction, if any, has been applied.

The need to store such properties has prompted the creation of flexible formats such as RIFF [SELF79] and BRIM (derived from RIFF) [MEIE83], which are general attribute-value database systems. In BRIM, for example, an image always has a width, height, and creator, and also a "history" field, which describes the creation of the image and modifications to it. Programs using BRIM can add their own signature and timestamp to the history field so that this information is automatically kept up to date. Figure 17.25 shows a text listing of a typical BRIM header for an image.

Samsung/Google, EX1027, p. 860

```
TYPE (string, 5): BRIM
FORMAT (string, 11): FORMAT_SEQ
TITLE (string, 31): Molecular Modeling Intro Frame
NAME (string, 27): Charles Winston  447 C.I.T.
DATE (string, 25): Sun Oct  9 12:42:16 1988
HISTORY (string, 82): Sun Oct  9 12:42:16 1988 crw RAY/n/
00033: Sun Oct  9 13:13:18 1988 crw brim_convert
DESC (string, 21): A ray-traced picture
IMAGE_WIDTH (int, 1): 640
IMAGE_HEIGHT (int, 1): 512
BRIM_VERSION (int, 1): 1
CHANNEL_DESC (string, 21): RED GREEN BLUE ALPHA
CHANNEL_WIDTH (short, 4): 8 8 8 8
ENCODING (string, 4): RLE
```

Fig. 17.25 A BRIM header for an image.

Many image-handling packages have been developed. One of the most widely used is the Utah Raster Toolkit [PETE86], which was written in fairly portable C so that the same tools can be used on a number of different architectures. Particularly troublesome issues for designing such toolkits are the numbers of bytes per word and the ordering of bytes within a word.

17.8 SPECIAL EFFECTS WITH IMAGES

The image-processing techniques described in Section 17.3 can be applied to an image to generate interesting special effects. If an image is processed with a high-pass filter, only the small details of the image remain, while all slowly varying aspects are deleted. By processing an image with a derivative filter, we can arrange to highlight all points where sharp transitions occur. Filters that reduce all intensity values below a certain level to 0 and increase all other values to 1 can be used to generate high-contrast images, and so on.

A large number of video techniques can be applied to blur images, to fade them, to slide them off a screen in real time, and so on. Many of these effects are created by using the electronic hardware for generating video signals, and modifying the signals as they are being shown. These techniques all lie in the domain of electrical engineering rather than in that of computer graphics, although the combination of effects from both disciplines can be fruitful.

We conclude this chapter by describing a digital technique for simulating neon tubing. If we paint the shape of a neon tube onto a black background, using a constant-width (antialiased) brush of constant color, our image does not look particularly exciting. But suppose we filter the image with an averaging filter—each pixel becomes the average of its immediate neighborhood. If we do this several times, the edges of the band we have drawn become blurred. If we now filter the image with an intensity-mapping filter that brightens those pixels whose intensities are above some threshold and dims those pixels whose intensities are lower than that threshold, the result looks quite a lot like neon tube. (There is another way to produce the same effect; see the discussion of antialiased brushes in Section

Samsung/Google, EX1027, p. 861

19.3.4.) Compositing such an image using an ω value greater than 1 and an α value less than 1 can cause the "neon" to illuminate whatever it is placed over, while the tube remains partially transparent.

17.9 SUMMARY

We have discussed several techniques for storing images, including some that are organized by programming considerations (multiple channels, headers), and some that are motivated by compactness or ease of transmission (quadtrees, IFS encodings). There are many other image storage formats, including a number of commercial formats competing for the privilege of being the "standard." Given the differences of opinion about the amount of color information that should be stored (should it be just RGB or should it consist of multiple spectral samples?), and about what information should be present in an image (should z-buffer values or α values be stored?), we expect no universal image format to evolve for some time.

We have also discussed geometric transformations on images, including multipass algorithms with filtering and the necessity of performing the filtering during such transformations. The number of interesting effects possible with image transformations is quite surprising. Due to the filtering used in such transformations, however, repeated transformations can blur an image. Transforming bitmaps involves other difficulties, since no gray scale is available to soften the aliasing artifacts that arise in the transformations. The simplicity of the data, however, makes it possible to develop very fast algorithms.

We also have discussed image compositing, which has become an extremely popular tool for generating complex images. When the α channel is used, composited images show no seams at the points where the component images overlap, unless the component images have some high geometric correlation. If rendering speed increases to the point where regenerating images is inexpensive, compositing may cease to be as important a tool as it now is. But if progress in computer graphics continues as it has, with each new generation of hardware allowing more complex rendering techniques, we should expect the image quality to increase, but the time-per-image to remain approximately constant. Therefore, we should expect compositing to be in use for some time.

EXERCISES

17.1 Show that the product of the matrices

$$\begin{bmatrix} 1 & \tan(t) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\sin(t)\cos(t) & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \cos(t) \end{bmatrix} \begin{bmatrix} 1/\cos(t) & 0 \\ 0 & 1 \end{bmatrix}$$

is exactly

$$\begin{bmatrix} \cos(t) & \sin(t) \\ -\sin(t) & \cos(t) \end{bmatrix}$$

Use this result to show how to create a rotation map from shears and scales. Use this technique to describe a pixmap-rotation algorithm derived from the Weiman algorithm (see Section 17.4.1).

17.2 How would you create a Weiman-style translation algorithm? Suppose a pixmap has alternating

Samsung/Google, EX1027, p. 862

columns of black and white pixels. What is the result of translating this pixmap by $\frac{1}{2}$ pixel? What is the result of applying Weiman's scaling algorithm to stretch this image by a factor of 2? What do you think of these results?

17.3 When scaling a bitmap, you cannot perform averaging, as you can in the Weiman algorithm. What is a good selection rule for the value of the target pixel? Is majority rule best? What if you want to preserve features of the image, so that scaling an image with a black line in the middle of a white page should result in a black line still being present? Is the cyclic permutation of the Rothstein code still necessary?

17.4 Show that in the Feibush, Levoy and Cook filtering method (see Section 17.4.2), the correct sign is assigned to volumes associated with triangles in step 3 of the edge-filtering process. (Hint: Evidently *some* sign must be given to each triangle, and this sign is a continuous function of the shape of the triangle—two triangles that look alike will have the same sign. The sign changes only when you modify a triangle by passing it through a degenerate triangle—one where the vertices are collinear. Thus, to do this exercise, you need only to show that the sign is correct for two triangles, one of each orientation.)

17.5 This problem fills in the details of the edge-filtering mechanism in the Feibush, Levoy and Cook image-transformation algorithm (see Section 17.4.2.) Given a triangle within a rectangular region, with one vertex at the center, C, of the rectangle, and a function (drawn as height) on the rectangular region, the volume over the triangle may be computed in the following way:

1. Call the triangle ABC. Draw a perpendicular from C to the base of the triangle, AB, intersecting AB at the point D. Express the triangle as either the sum or the difference of the two triangles ACD and BCD.

2. Find the volume over the triangles ACD and BCD, and use these values to compute the volume over ABC.

3. Observe that, if the filter function is circularly symmetric, then the volume over ACD computed in step (2) is a function of only the length of the base, AD, and the height of the triangle, CD (the same is true for BCD, of course).

    a. Draw a picture of the situation described in step 1. Do this for two cases: angle ACB is less than 90° and angle ACB is greater than 90°.

    b. Find a condition on A, B, and D that determines whether ABD is the sum of the difference of ACD and BCD.

    c. Suggest a method for computing the volume above an arbitrary right triangle as described in step 3. Since the given function may not be integrable in elementary terms, consider Monte Carlo methods.

    d. Describe how you would arrange a table of widths and heights to store all the volumes computed in step 3 in a look-up table.

17.6 Consider the 1 by 3 image that is described as follows. The pixel centers in this image are at the points $(-2,0)$, $(0, 0)$, and $(2, 0)$, and the values at these points are -1, 0, and 1. The Nyquist frequency for the image is 1, and the image can be considered as a sample of a unique image that is a linear combination of sine and cosine functions with frequency 1 or less. All the cosine terms are zero (you can see that they are by noting that the function is odd, in the sense that $f(-x) = -f(x)$ for all x).

    a. Compute the coefficients, a_k, of $\sin(kx)$ (for $0 \leq k < 1$) so that samples of $\Sigma a_k \sin(kx)$ give this image. Hint: $a_k \neq 0$ for only one value of k.

    b. The image might appear to be simply a gray-scale ramp (if we imagine -1 as black and 1 as

Samsung/Google, EX1027, p. 863

white). What would happen if we sampled the signal computed in part a at the points (−1, 0), and (1, 0). Would they interpolate the gray-scale ramp as expected?

This exercise shows the difficulty we encounter when we use exact reconstruction without filtering.

17.7 Assume that you are given an 2^k by 2^k gray-scale image with intensity values $0 \ldots 2^n - 1$. You can generate a 2^{k-1} by 2^{k-1} image by condensing 2 by 2 regions of the image into single bits. In Section 17.7.1, we proposed doing the condensation by computing the mean of the region. There are other possibilities, however. Analyze the selection method for condensation, where *selection* means choosing one particular corner of the region as the representative value.

a. Assume that you want to transmit a 2^k by 2^k gray-scale image that has been completely condensed using selection. How many pixel values do you need to send? (Assume that the receiving hardware knows how to decode the incoming data, and can draw filled rectangles on a display.)

b. On what class of pictures will the selection method give bad artifacts until the image is nearly completely displayed? Would some other condensation rule work better for these images?

c. What other condensation rules would be preferable for sending black-and-white documents (such as typical pages from this book, which may contain figures)? Explain your choices.

17.8 (Note: To do this programming problem, you must have access to hardware that supports very rapid *bitBlt* operations on bitmaps.) Write a program that uses a 1-bit-deep α-buffer to composite bitmaps. Add such features as painting in a bitmap with an α-buffer (which determines where the paint "sticks" or shows). This problem is discussed extensively in [SALE85].

17.9 Suppose you were given a corrupted run-length encoded image, from which several bytes were missing. Could you reconstruct most of it? Why or why not? Suggest a run-length encoding enhancement that would make partial recovery of a corrupted image easier.

17.10 We saw in Section 17.5 how various transformations can be implemented in multiple passes.

a. If a pattern is mapped linearly onto a polygon and then projected onto the viewing plane, show that the composite map from the pattern map (x, y) coordinates to the image (u, v) coordinates has the form

$$(x, y) \rightarrow (u, v),$$

where

$$u = (Ax + By + C)/(Dx + Ey + F)$$
$$v = (Px + Qy + R)/(Sx + Ty + U)$$

b. Show how to factor this map into two passes as we did for the map

$$u = x/(y + 1),\ v = y/(y + 1).$$

This idea of using two-pass transforms for texture mapping was introduced by Catmull and Smith in [CATM80]. The names of variables for the coordinates here are chosen to agree with the convention used in Section 17.6 and not with the names of the coordinates used in describing texture mapping elsewhere.

Samsung/Google, EX1027, p. 864

18
Advanced Raster Graphics Architecture

Steven Molnar
and Henry Fuchs

In this chapter, we discuss in more detail the issues of raster graphics systems architecture introduced in Chapter 4. We examine the major computations performed by raster systems, and the techniques that can be used to accelerate them. Although the range of graphics architectures and algorithms is wide, we concentrate here on architectures for displaying 3D polygonal models, since this is the current focus in high-end systems and is the foundation for most systems that support more complex primitives or rendering methods.

Graphics systems architecture is a specialized branch of computer architecture. It is driven, therefore, by the same advances in semiconductor technology that have driven general-purpose computer architecture over the last several decades. Many of the same speed-up techniques can be used, including pipelining, parallelism, and tradeoffs between memory and computation. The graphics application, however, imposes special demands and makes available new opportunities. For example, since image display generally involves a large number of repetitive calculations, it can more easily exploit massive parallelism than can general-purpose computations. In high-performance graphics systems, the number of computations usually exceeds the capabilities of a single CPU, so parallel systems have become the rule in recent years. The organization of these parallel systems is a major focus of graphics architecture and of this chapter.

We begin by reviewing the simple raster-display architecture described in Chapter 4. We then describe a succession of techniques to add performance to the system, discussing the bottlenecks that arise at each performance level and techniques that can be used to overcome them. We shall see that three major performance bottlenecks consistently resist attempts to increase rendering speed: the number of floating-point operations to perform

Samsung/Google, EX1027, p. 865

geometry calculations, the number of integer operations to compute pixel values, and the number of frame-buffer memory accesses to store the image and to determine visible surfaces. These demands have a pervasive influence on graphics architecture and give rise to the diversity of multiprocessor graphics architectures seen today. At the end of the chapter, we briefly discuss several unusual architectures, such as those for ray tracing, for true 3D displays, and for commercial flight simulators.

18.1 SIMPLE RASTER-DISPLAY SYSTEM

As described in Chapter 4, a simple raster-display system contains a CPU, system bus, main memory, frame buffer, video controller, and CRT display (see Fig. 4.18). In such a system, the CPU performs all the modeling, transformation, and display computations, and writes the final image to the frame buffer. The video controller reads pixel data from the frame buffer in raster-scan order, converts digital pixel values to analog, and drives the display.

It is important to remember that, if such a system has sufficient frame-buffer memory, has a suitable CRT display, and is given enough time, it can generate and display scenes of virtually unlimited complexity and realism. None of the architectures or architectural techniques discussed here enhance this fundamental capability (except for a few exotic 3D displays treated in Section 18.11). Rather, most work in graphics architecture concerns the quest for increased rendering speed.

In Chapter 4, we discussed two problems that limit the performance of this simple system: the large number of frame-buffer memory cycles needed for video scanout and the burden that image generation places on the main CPU. We now consider each of these problems in greater detail.

18.1.1 The Frame-Buffer Memory-Access Problem

In Chapter 4, we calculated the time between successive memory accesses when a low-resolution monochrome display is being refreshed. For a system with 16-bit words, the access rate is substantial—one memory access every 864 nanoseconds. Systems with higher-resolution color monitors require much higher memory speeds. For example, refreshing a 1280 by 1024 screen with 32-bit (one-word) pixels at 60 Hz requires that memory accesses occur every $1/(1280 \cdot 1024 \cdot 60) = 12.7$ nanoseconds. Even this is only the average memory access rate, not the peak rate, since pixels are not scanned out during horizontal and vertical retrace times [WHIT84]. A simple *dynamic random-access memory* (DRAM) system, on the other hand, has a cycle time of approximately 200 nanoseconds, a factor of 16 slower than the speed required. Clearly, something must be done to increase the bandwidth to frame-buffer memory.

The following sections discuss solutions that have been used by various system designers. Some of these provide only modest performance increases, but are sufficient for low-resolution systems. Others provide greatly increased memory bandwidth, but incur significant system complexity. We begin by reviewing briefly the fundamentals of DRAM

Samsung/Google, EX1027, p. 866

memories, since the characteristics of DRAMs strongly influence the set of solutions available.

18.1.2 Dynamic Memories

DRAMs are the memories of choice for most computer memory systems. *Static random-access memories* (SRAMs), which retain stored data indefinitely, can be made to run faster than DRAMs, which must be accessed every few milliseconds to retain data, but DRAMs are much denser and cheaper per bit. (DRAMs also require more complicated timing for reading and writing data, but these problems are easily solved with supporting circuitry.)

Figure 18.1 is a block diagram of a typical 1-Mbit DRAM chip. As in most DRAMs, single-bit storage elements are arranged in one or more square arrays (in this case, four arrays, each with dimension 512 by 512). Vertical *bit lines* transfer data to and from the storage arrays, one bit line for each column of each array. During read and write operations, one memory cell in each column is connected to its corresponding bit line. A *sense amplifier* attached to each bit line amplifies and restores the tiny signals placed on the bit line during read operations.

In a DRAM chip, read and write operations each require two steps. The first step is to select a row. This is done by asserting the *row address strobe* (RAS) while the desired row address is on the address inputs. The row decoder produces a 512-bit vector, whose bits are

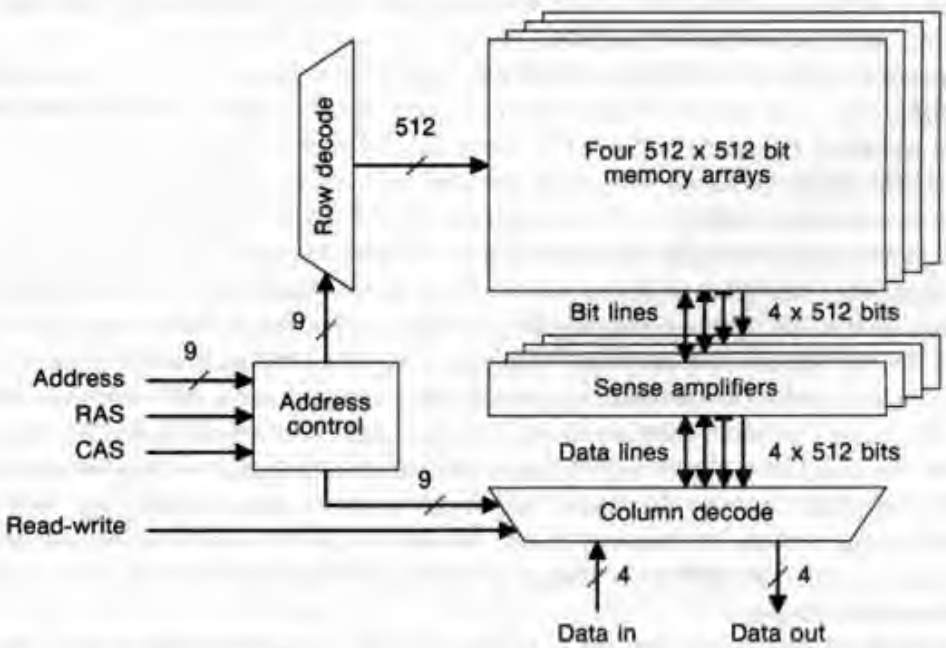


Fig. 18.1 A 1-Mbit (256K × 4) DRAM chip. An entire row is written to or read from each of the four memory arrays at the same time. The column decoder allows a particular element (column) of the selected row to be accessed.

Samsung/Google, EX1027, p. 867

0 everywhere except for a single 1 at the selected row. This bit vector determines which row's storage cells are connected to the bit lines and sense amplifiers.

The second step is to select a column, which is done by asserting the *column address strobe* (CAS) and read–write signal while the desired column address is on the address inputs. The column address selects a single bit from the active row of memory in each array. The selected bits are either buffered for output (during read operations) or set to the value on the data inputs (during write operations). Some DRAMs provide a faster access method called *page mode* for successive reads or writes to memory locations on the same row. In page mode, a row is selected just once, and successive columns are selected using the column address bits and CAS signal. In page mode, consecutive memory accesses require just one address cycle, instead of two. Thus, if adjacent pixels are stored in the same memory row, page mode can nearly double the bandwidth available for updating and displaying from the frame buffer.

Word widths greater than the number of data pins on a single memory chip can be accommodated by connecting multiple memory chips in parallel. With this arrangement, an entire word of data can be read from or written to the memory system in a single memory cycle. For example, eight 4-bit-wide DRAMs can be used to build a 32-bit memory system.

18.1.3 Increasing Frame-Buffer Memory Bandwidth

As we have seen, obtaining sufficient frame-buffer memory bandwidth both for video scanout and for updates by the CPU is a daunting problem. Different solutions have been proposed and used in various systems.

A partial solution, suitable for low-resolution displays, is to place the frame buffer on an isolated bus. This allows video scanout to occur simultaneously with CPU operations that do not affect the frame buffer. Only when the CPU needs to read from or write to the frame buffer must it contend with video scanout for memory cycles.

A second partial solution is to take advantage of DRAMs' page-mode feature. Since video scanout accesses memory locations in order, DRAM page misses are infrequent, so the frame buffer can run at almost twice its normal speed. Note, however, that frame-buffer accesses by the CPU may be less regular and may not be able to take advantage of page mode. We can decrease the number of page misses by the CPU by providing it with a data cache. As in conventional computing, a cache decreases the traffic between the CPU and memory, since a memory location accessed during one cycle is very likely to be accessed again in the near future. Furthermore, since cache entries frequently occupy several words, transferring data between the cache and main memory can exploit page mode. By combining page mode and a data cache, we can reduce the memory-access problem substantially, although further measures, as discussed in Section 18.1.5, are necessary for high-resolution displays.

A third, independent approach is to duplicate the frame-buffer memory, creating a *double-buffered* system in which the image in one buffer is displayed while the image in the other buffer is computed. Figure 18.2 shows one possible implementation, in which multiplexers connect each frame buffer to the system bus and video controller. Double-buffering allows the CPU to have uninterrupted access to one of the buffers while the video

Samsung/Google, EX1027, p. 868

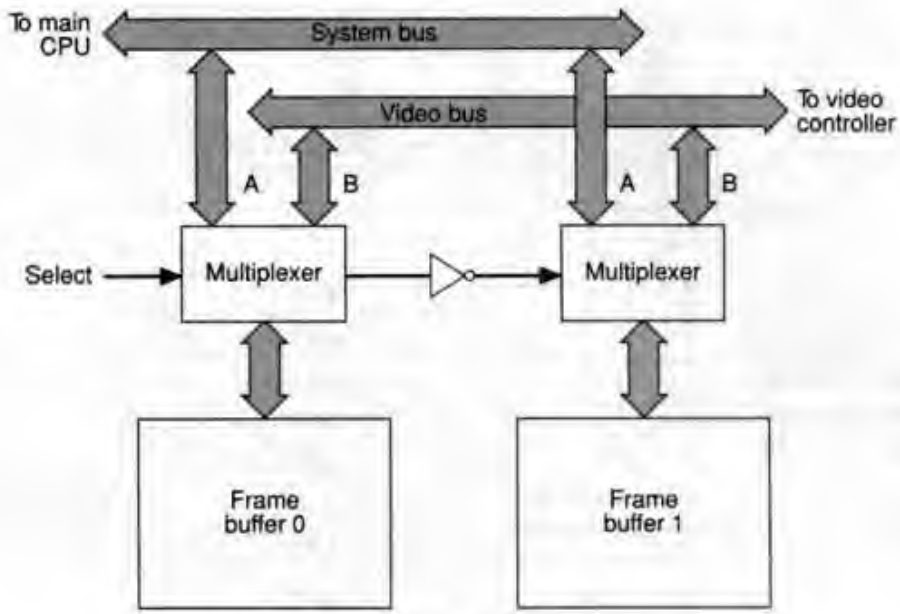


Fig. 18.2 Double-buffered frame buffer.

controller has uninterrupted access to the other. Double-buffering in this manner is expensive, however, since twice as much memory is needed as for a single-buffered display. Also, the multiplexers that provide dual access to the DRAMs require numerous chips, which increase the size of the system. (Actual dual-ported memories, with two fully independent read–write ports, could be used, but their low density and high cost make them impractical in almost all applications.)

18.1.4 The Video RAM

In 1983, Texas Instruments introduced a new type of DRAM, the *video random-access memory* (VRAM), designed specifically to allow video scanout to be independent of other frame-buffer operations [PINK83]. A VRAM chip, as shown in Fig. 18.3, is similar to a conventional DRAM chip, but contains a parallel-in/serial-out data register connected to a second data port. The serial register is as wide as the memory array and can be parallel loaded by asserting the transfer signal while a row of memory is being read. The serial register has its own data clock, enabling it to transfer data out of the chip at high speeds. The serial register and port effectively provide a second, serial port to the memory array. If this port is used for video scanout, scanout can occur in parallel with normal reads from and writes to the chip, virtually eliminating the video-scanout problem.

Since the shift register occupies only a small fraction of a VRAM's chip area and very few pins are needed to control it, VRAMs ideally should be only slightly more expensive than DRAMs. Unfortunately, the economics of scale tend to raise the price of VRAMs relative to DRAMs, since fewer VRAMs are produced (in 1989, VRAMs were roughly twice as expensive as DRAMs of the same density). In spite of this price differential, VRAMs are an excellent choice for many frame-buffer memories.

Samsung/Google, EX1027, p. 869

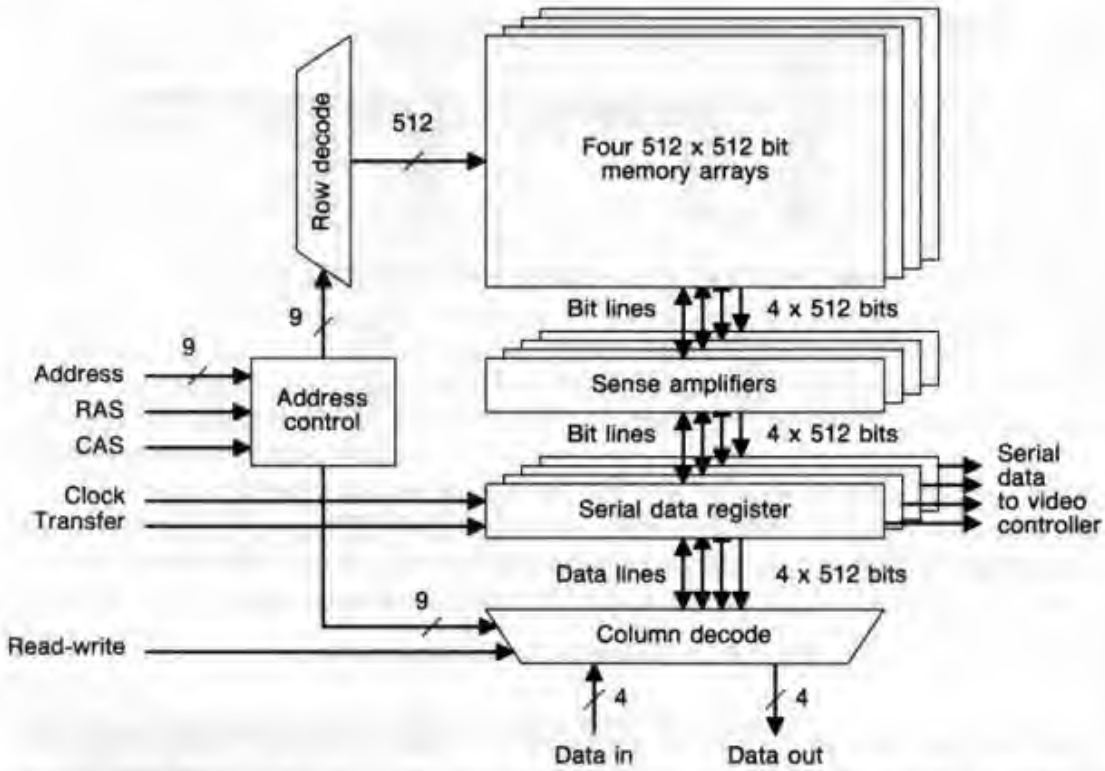


Fig. 18.3 Diagram of a 1-Mbit (256K × 4) VRAM chip. The serial data register provides a second (serial) port to the memory array.

18.1.5 A Frame Buffer for a High-Resolution Display

Figure 18.4 shows a typical frame-buffer design using VRAMs. The CPU has essentially unrestricted access to the frame-buffer memory. Only once every n cycles (where n is the dimension of the memory array) is a memory cycle required to load a new row into the VRAMs' serial registers. With screen resolutions up to 512 by 512 pixels, VRAMs can handle video scanout unassisted. With higher-resolution displays, the pixel rate exceeds the VRAMs' serial-port data rate of 30 MHz or less. A standard solution is to connect multiple banks of VRAMs to a high-speed off-chip shift register, as shown in the figure. Multiple pixels, one from each VRAM bank, can be loaded into the shift register in parallel and then shifted out serially at video rates. (Notice that this 1280- by 1024-pixel system, with five times the resolution of a 512 by 512 system, uses five-way multiplexing.) This shift register frequently is incorporated into a single chip containing look-up tables and digital-to-analog converters, such as Brooktree's Bt458 RAMDAC [BROO88]. In systems with 24 bits per pixel, such as the one shown in Fig. 18.4, three RAMDAC chips are typically used—one for each 8-bit color channel.

VRAMs provide an elegant solution to the frame-buffer memory-access problem. The bottleneck that remains in such systems is CPU processing speed. To generate a raster image (in 2D or 3D), the processor must calculate every pixel of every primitive. Since a typical primitive covers many pixels, and images typically contain thousands of primitives,

Samsung/Google, EX1027, p. 870

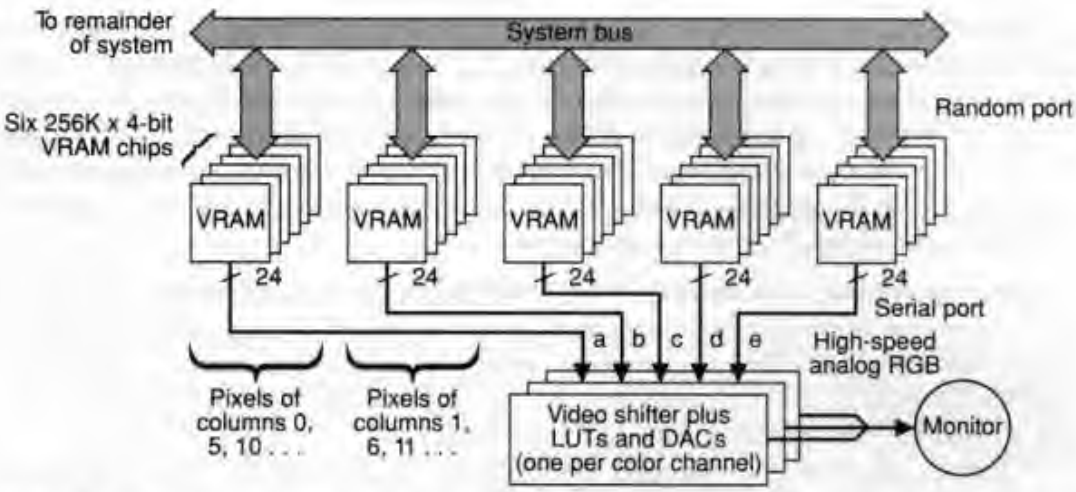


Fig. 18.4 Block diagram of a 1280- by 1024-pixel display using VRAMs. A parallel-in/serial-out shift register provides the high pixel rates required to drive the monitor.

generating a high-resolution image can require many millions of CPU operations. Because of this, even the fastest CPUs have difficulty rendering images quickly enough for interactive use.

18.2 DISPLAY-PROCESSOR SYSTEMS

To build a system with higher performance than the single-CPU system described in the previous section, we must either remove some of the graphics burden from the CPU or increase the CPU's ability to perform these tasks. Here we consider strategies that do not substantially increase the size or cost of the system (we consider larger, multiprocessing solutions in Sections 18.6 through 18.10). These strategies generally fall into three categories: (1) coprocessors that share the system bus with the main CPU, (2) display processors with their own buses and memory systems, and (3) integrated processors containing internal hardware support for graphics operations.

Coprocessors were discussed extensively in Chapter 4 (single-address-space (SAS) architectures). These have been found useful in 2D systems, but have not achieved wide use in 3D systems, largely because 3D systems require a larger set of operations than 2D systems and because bus contention in a SAS architecture severely limits system performance. The latter two approaches have been used successfully for 3D graphics, and we now discuss them further.

18.2.1 Peripheral Display Processors

Figure 4.22 shows a typical peripheral-display-processor system organization. The display processor generally has its own memory system (including the frame buffer) and a fast communication link to the main CPU. The main CPU sends display commands to the display processor, which executes them using resident software or firmware routines, or using specialized hardware.

Samsung/Google, EX1027, p. 871

The display processor may be the same processor type as the main CPU, or it may be a special-purpose processor. Using the same type of processor, as in the Masscomp MC-500 [MASS85], allows simpler and more flexible hardware and software designs. It does not exploit the regularity and specialized nature of most graphics calculations, however, and therefore provides at most a factor of 2 increase in performance, since the overall processing capacity is simply doubled. Most current display-processor systems use peripheral processors specialized for graphics operations.

Single-chip graphics processors. Single-chip graphics processors are a successful and inexpensive type of peripheral-display processor that developed as functional extensions of single-chip video controllers. Consequently, many current graphics-processor chips combine the functions of the video controller and display processor. (Section 18.2.2 describes one popular graphics processor chip, the TMS34020.)

Graphics-processor chips targeted especially for 3D graphics have been introduced only recently. A pioneering chip of this type is Intel's i860, which can be used either as a stand-alone processor with support for graphics or as a 3D graphics processor. Unlike many 2D graphics-processor chips, the i860 does not provide hardware support for screen refresh or video timing. The i860 will be discussed in depth in Section 18.2.4.

Programming the display processor. A major point of difference among display processor systems is where they store the model or database from which the image is generated: in the CPU's memory, or in the display processor's memory. Storing the database in the display processor's memory frees the main CPU from database traversal. Consequently, only a low-bandwidth channel is needed between the CPU and the display processor. However, such a system also inherits all the disadvantages of a structure database described in Chapter 7, particularly the inflexibility of a canonical, "hard-wired" database model.

Whether or not the display processor manages the database, it is an extra processor to program. Frequently, this program takes the form of a standard graphics library, such as PHIGS+, which satisfies the needs of many applications. Applications invariably arise, however, that require features not supported by the library. In such cases, users are forced either to program the display processor themselves (which may not even be possible), or to devise cumbersome workarounds using existing display-processor features.

18.2.2 Texas Instruments' TMS34020—A Single-Chip Peripheral Display Processor[1]

Texas Instruments' TMS34020 [TEXA89] is an example of a single-chip graphics processor designed to accelerate 2D displays on PCs and workstations. It can be paired with a TMS34082 floating-point coprocessor, which rapidly performs 3D geometric transformations and clipping needed for 3D graphics. Unlike graphics-processor chips that perform only graphics-specific operations, the 34020 is a fully programmable 32-bit processor that can be used as a stand-alone processor if desired.

[1]Material for this example was contributed by Jerry R. Van Aken of Texas Instruments, Inc.

Samsung/Google, EX1027, p. 872

Figure 18.5 shows a typical system configuration using the 34020 (and optionally the 34082). As indicated in the figure, the graphics processor's local memory may contain VRAMs for the frame buffer and DRAMs for code and data storage. Alternatively, the graphics processor's entire memory may be built from VRAMs. The 34020 contains on-chip timers and registers to control video scanout and DRAM refreshing.

The 34020 supports data types for pixels and pixmaps, allowing pixels to be accessed by their (x, y) coordinates as well as by their addresses in memory. The following instructions implement common 2D graphics operations:

PIXBLT: The PIXBLT (pixel-block transfer) instruction implements a general bitBlt operation. It copies pixels of 1 to 8 bits, automatically aligning, masking, and clipping rows of pixels fetched from the source array before writing them to the destination array. It uses many of the optimizations discussed in Chapter 19, as well as processing several pixels in parallel, so the transfer rate approaches the system's full memory bandwidth—up to 18 million 8-bit pixels per second. A special version of PIXBLT supports text by efficiently expanding packed 1-bit bitmaps into 8-bit pixmaps.

FILL: The FILL instruction fills a rectangular array of pixels with a solid color. Large areas are filled at rates approaching the memory-bus speed. Texas Instruments' 1-Mbit VRAM supports a special block-write mode that allows up to four memory locations to be written at once from the same on-chip color register. Using this block-write mode, the 34020 fills areas at up to 160 million 8-bit pixels per second.

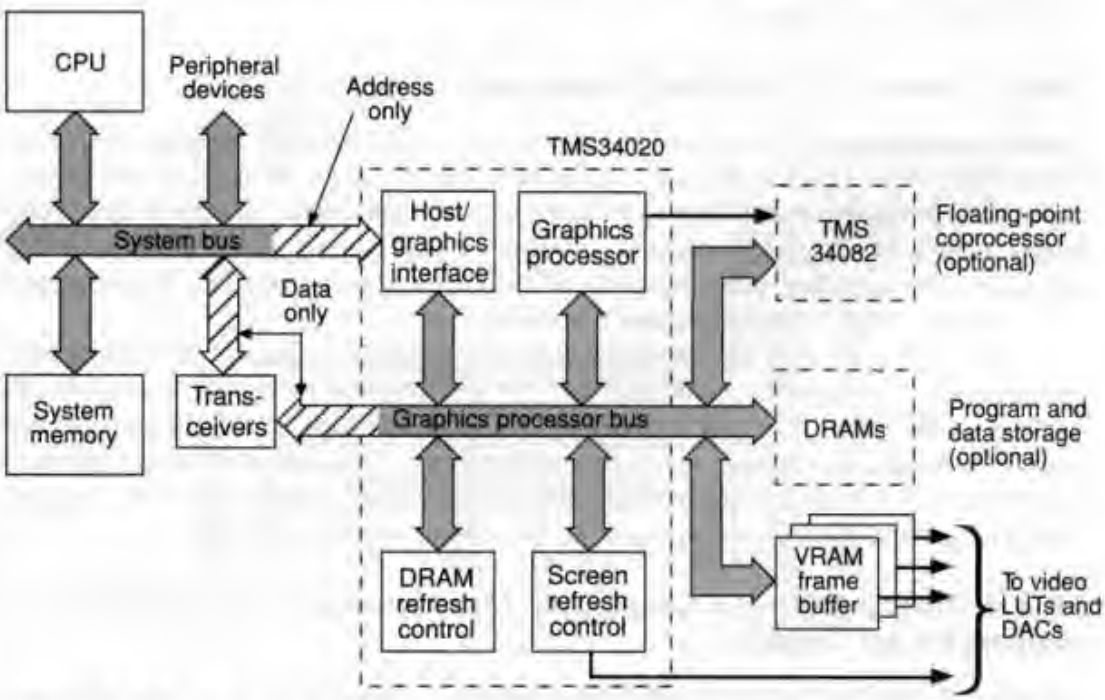


Fig. 18.5 Block diagram of the TMS34020 graphics-processor chip used as a peripheral display processor.

Samsung/Google, EX1027, p. 873

FLINE: The FLINE instruction draws 1-pixel-thick straight lines using a midpoint scan-conversion algorithm (see Section 3.2.2). The FLINE instruction draws up to 5 million pixels per second.

DRAV: The DRAV (draw-and-advance) instruction draws the pixel indicated by a pair of (x, y) coordinates, then increments the x and y coordinates in parallel. It is used in the inner loops of incremental algorithms for drawing circles and ellipses; for example, the inner loop of a midpoint circle routine (see Section 3.3.2) draws 2 million pixels per second.

Software for the 34020 is typically divided into time-critical graphics subroutines, written in assembly language, and other software, written in a high-level language. Software running on the 34020 communicates with the application program running on the main CPU.

The 34020 contains a 512-byte on-chip instruction cache and 30 general-purpose registers—enough memory to store the instructions and data required within the inner loop of many graphics subroutines. Once the instructions and data for the loop have been read from external memory, the memory bus is available exclusively for moving pixels to and from the frame buffer.

The TMS34082 floating-point coprocessor chip interfaces directly to the 34020, providing floating-point instructions and registers that increase the system's capabilities for 3D and other floating-point-intensive applications. The 34082 monitors the 34020's memory bus and receives floating-point instructions and data transmitted over the bus by the 34020. The 34082 can transform a 3D homogeneous point (including the homogeneous divide) every 2.9 microseconds.

18.2.3 Integrated Graphics Processors

Another approach to increasing system performance is to add support for graphics directly to the main CPU. This has become feasible only recently, with the development of very dense VLSI technologies that allow entire systems to be implemented on a single chip. Fully programmable 2D graphics processors, such as TI's 34020, are early examples of this approach, although they dedicate much of their chip area to graphics functions and comparatively little to general-purpose processing.

Intel's i860 is the first microprocessor chip to integrate direct support for 3D graphics. Proponents of this approach argue that the chip area required to support reasonable 3D graphics performance is relatively small and the payoff is high, since 3D graphics has become an integral part of many computing environments. This approach may well produce high-performance, low-cost 3D systems as increasing VLSI densities allow more sophisticated systems and more system components to be built onto a single chip.

18.2.4 Intel's i860—A Single-Chip Microprocessor with Integrated Support for 3D Graphics

Advances in VSLI technology have made it possible to build chips with extremely large numbers of transistors—more than 1 million in 1989. This allows high-performance CPU chips to include other features, such as caches, I/O controllers, and support for specialized

Samsung/Google, EX1027, p. 874

instructions. Intel's i860 microprocessor, also known as the 80860, [GRIM89] is an example of such a chip with features supporting 3D graphics. Figure 18.6 is a block diagram of the major data paths in the i860.

According to Intel, the i860 achieves the following benchmarks: 33 million VAX-equivalent instructions per second (on compiled C code), 13 double-precision MFLOPs (using the Linpack benchmark), and 500,000 homogeneous vector transformations [INTE89].

The i860's graphics instructions operate in parallel on as many pixels as can be packed into a 64-bit data word. For applications using 8-bit pixels, this means eight operations can occur simultaneously. For 3D shaded graphics, which generally requires 32-bit pixels, two operations can occur simultaneously. Parallel graphics instructions include:

- Calculating multiple linear interpolations in parallel
- Calculating multiple z-buffer comparisons in parallel
- Conditionally updating multiple pixels in parallel.

For systems using 8-bit pixels (where eight operations can occur in parallel), the i860 can scan convert and shade 50,000 Gouraud-shaded 100-pixel triangles per second.

The i860 may be used either as a peripheral display processor (with an 80486

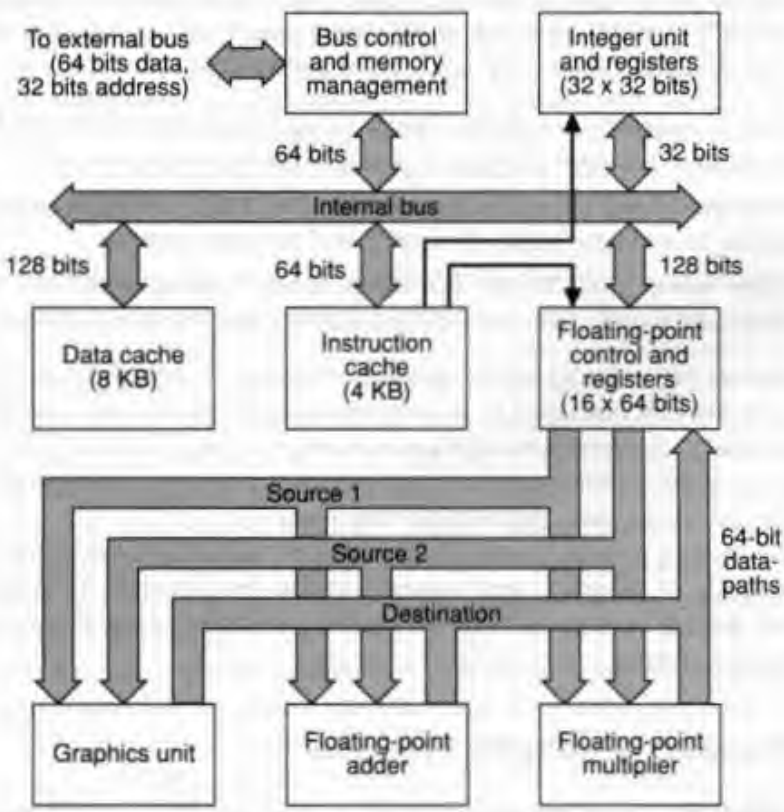


Fig. 18.6 Block diagram of Intel i860 microprocessor datapaths (adapted from [GRIM89]).

Samsung/Google, EX1027, p. 875

microprocessor as the main CPU, for example) or as a stand-alone processor. By combining front-end and back-end capabilities in a single processor, the i860 allows a powerful 3D graphics system to be built with very few parts—essentially the processor, memory, and the video system.

As VLSI densities continue to improve, we can expect to see even higher performance and more complex processors designed for graphics. Features commonly found in 2D graphics-processor chips, such as scanout and video-timing circuitry, may be included in such processors as well.

18.2.5 Three Performance Barriers

Systems can be built with a large range of performances using the techniques described up to this point. Low-end systems using a general-purpose microprocessor for the application processor and a 2D graphics processor are well suited for displays on PCs. Higher-performance systems using a powerful application processor and a high-speed 3D graphics processor, containing display-list memory and transformation hardware, are suitable for engineering and scientific workstations.

But how far can such designs be pushed? Let us imagine a display system made with the fastest available CPU and display processor, and a frame buffer containing the fastest available VRAMs. Such a system will achieve impressive—but still limited—performance. In practice, such *single-stream* architectures have been unable to achieve the speeds necessary to display large 3D databases at interactive rates. Architectures that display more primitives at faster update rates must overcome three barriers:

- *Floating-point geometry processing*: Accelerating the transformation and clipping of primitives beyond the rates possible in a single floating-point processor
- *Integer pixel processing*: Accelerating scan conversion and pixel processing beyond the rates possible in a single display processor and memory system
- *Frame-buffer memory bandwidth*: Providing higher bandwidth into the frame buffer (faster reads and writes) than can be supported by a conventional memory system.

These barriers have analogs in conventional computer design: the familiar processing and memory-bandwidth bottlenecks. In conventional computers, the presence of these bottlenecks has led to the development of concurrent processing. Similarly, the major focus of research in graphics architecture over the last decade has been investigating ways to use concurrency to overcome these performance barriers.

Before we discuss parallel graphics architectures, let us establish some groundwork about the demands of graphics applications and the fundamentals of multiprocessing. Later, we shall discuss various methods for building parallel systems that overcome these three performance barriers.

18.3 STANDARD GRAPHICS PIPELINE

Here we review from a hardware perspective the standard graphics pipeline discussed in Chapter 16. As has been mentioned, the rendering pipeline is a logical model for the computations needed in a raster-display system, but is not necessarily a physical model, since the stages of the pipeline can be implemented in either software or hardware.

Samsung/Google, EX1027, p. 876

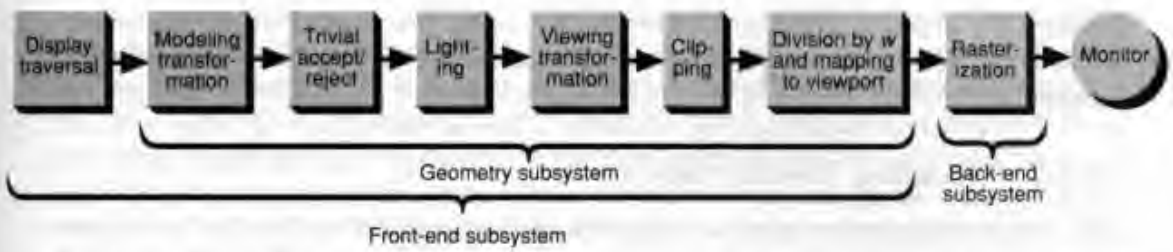


Fig. 18.7 Standard graphics pipeline for Gouraud- or Phong-shaded polygons.

Figure 18.7 shows a version of the rendering pipeline that is typical for systems using conventional primitives (lines and polygons) and conventional shading techniques (constant, Gouraud, or Phong). We shall discuss each stage of the pipeline in turn, paying particular attention to algorithms that have been used successfully in hardware systems and to those that may be useful in the future. At the end of this section, we estimate the number of computations needed at each stage of the graphics pipeline to process a sample database containing 10,000 polygons.

18.3.1 Display Traversal

The first stage of the pipeline is traversal of the display model or database. This is necessary because the image may change by an arbitrary amount between successive frames. All the primitives in the database must be fed into the remainder of the display pipeline, along with context information, such as colors and current-transformation matrices. Chapter 7 described the two types of traversal: immediate mode and retained mode. Both methods have advantages and disadvantages, and the choice between them depends on the characteristics of the application and of the particular hardware architecture used.

Immediate mode offers flexibility, since the display model does not need to conform to any particular display-list structure and the application has the luxury of recreating the model differently for every frame. The main CPU must perform immediate-mode traversal, however, expending cycles it could use in other ways. Retained mode, on the other hand, can be handled by a display processor if the structure database is stored in its local memory. Retained-mode structure traversal can be accelerated by optimizing the database storage and access routines or by using a dedicated hardware traverser. Furthermore, since the main CPU only edits the database each frame, rather than rebuilding it from scratch, a low-bandwidth channel between the main CPU and the display processor is sufficient. Of course, relatively few changes can be made to the structure database between frames, or system performance will suffer.

The choice between traversal modes is a controversial matter for system designers [AKEL89]. Many argue that retained mode offers efficiency and high performance. Others believe that immediate mode supports a wider range of applications and does not necessarily lead to reduced performance if the system has a sufficiently powerful CPU.

Unfortunately, it is difficult to estimate the processing requirements for display traversal, since they depend on the traversal method used and on the characteristics of the particular display model. At the very least, a read operation and a write operation must be performed for each word of data to be displayed. The processing requirements may be much

Samsung/Google, EX1027, p. 877

greater if the structure hierarchy is deep or if it contains many modeling transformations (because of the overhead in pointer chasing, state saving, concatenating transformation matrices, etc.).

18.3.2 Modeling Transformation

In this stage of the pipeline, graphics primitives are transformed from the object-coordinate system to the world-coordinate system. This is done by transforming the vertices of each polygon with a single transformation matrix that is the concatenation of the individual modeling transformation matrices. In addition, one or more surface-normal vectors may need to be transformed, depending on the shading method to be applied.

Constant shading requires world-space surface-normal vectors for each polygon. We compute these by multiplying object-space surface normals by the transpose of the inverse modeling transformation matrix. Gouraud and Phong shading require world-space normals for each vertex, rather than for each polygon, so each vertex-normal vector must be multiplied by the transpose inverse transformation matrix.

Let us compute the number of floating-point calculations required to transform a single vertex if Gouraud shading is to be applied. Multiplying a homogeneous point by a 4×4 matrix requires 16 multiplications and 12 additions. Multiplying each vertex normal by the inverse transformation matrix requires 9 multiplications and 6 additions (only the upper-left 3×3 portion of the matrix is needed—see Exercise 18.1). Therefore, transforming a single vertex with surface normal requires $16 + 9 = 25$ multiplications and $12 + 6 = 18$ additions.

18.3.3 Trivial Accept/Reject Classification

In the trivial accept/reject classification stage, primitives (now in world coordinates) are tested to see whether they lie wholly inside or outside the view volume. By identifying primitives that lie outside the view volume early in the rendering pipeline, processing in later stages is minimized. We will clip primitives that cannot be trivially accepted or rejected in the clipping stage.

To trivially accept or reject a primitive, we must test each transformed vertex against the six bounding planes of the view volume. In general, the bounding planes will not be aligned with the coordinate axes. Each test of a vertex against a bounding plane requires 4 multiplications and 3 additions (the dot product of a homogeneous point with a 3D plane equation). A total of $6 \cdot 4 = 24$ multiplications and $6 \cdot 3 = 18$ additions are required per vertex.

18.3.4 Lighting

Depending on the shading algorithm to be applied (constant, Gouraud, or Phong), an illumination model must be evaluated at various locations: once per polygon for constant shading, once per vertex for Gouraud shading, or once per pixel for Phong shading. Ambient, diffuse, and specular illumination models are commonly used in high-performance systems.

Samsung/Google, EX1027, p. 878

In constant shading, a single color is computed for an entire polygon, based on the position of the light source and on the polygon's surface-normal vector and diffuse color (see Eq. 16.9). The first step is to compute the dot product of the surface-normal vector and the light vector (3 multiplications and 2 additions for directional light sources). If an attenuation factor based on the distance to the light source is used, we must calculate it and multiply it by the dot product here. Then, for each of the red, green, and blue color components, we multiply the dot product by the light-source intensity and diffuse-reflection coefficient (2 multiplications), multiply the ambient intensity by the ambient-reflection coefficient (1 multiplication), and add the results (1 addition). If we assume a single directional light source, calculating a single RGB triple requires $3 + 3 \cdot (2 + 1) = 12$ multiplications and $2 + 3 \cdot 1 = 5$ additions. Gouraud-shading a triangle requires three RGB triples—one for each vertex.

Phong shading implies evaluating the illumination model at each pixel, rather than once per polygon or once per polygon vertex. It therefore requires many more computations than does either constant or Gouraud shading. These computations, however, occur during the rasterization stage of the pipeline.

18.3.5 Viewing Transformation

In this stage, primitives in world coordinates are transformed to normalized projection (NPC) coordinates. This transformation can be performed by multiplying vertices in world coordinates by a single 4×4 matrix that combines the perspective transformation (if used) and any skewing or nonuniform scaling transformations needed to convert world coordinates to NPC coordinates. This requires 16 multiplications and 12 additions per vertex. Viewing transformation matrices, however, have certain terms that are always zero. If we take advantage of this, we can reduce the number of computations for this stage by perhaps 25 percent. We will assume that 12 multiplications and 9 additions per vertex are required in the viewing transformation stage.

Note that if a simple lighting model (one that does not require calculating the distance between the light source and primitive vertices) is used, modeling and viewing transformation matrices can be combined into a single matrix. In this case only one transformation stage is required in the display pipeline—a significant savings.

18.3.6 Clipping

In the clipping stage, lit primitives that were not trivially accepted or rejected are clipped to the view volume. As described in Chapter 6, clipping serves two purposes: preventing activity in one screen window from affecting pixels in other windows, and preventing mathematical overflow and underflow from primitives passing behind the eye point or at great distances.

Exact clipping is computationally practical only for simple primitives, such as lines and polygons. These primitives may be clipped using any of the 3D clipping algorithms described in Section 6.5. Complicated primitives, such as spheres and parametrically defined patches, are difficult to clip, since clipping can change the geometric nature of the primitive. Systems designed to display only triangles have a related problem, since a clipped triangle may have more than three vertices.

Samsung/Google, EX1027, p. 879

An alternative to exact clipping is scissoring, described in Section 3.11. Here, primitives that cross a clipping boundary are processed as usual until the rasterization stage, where only pixels inside the viewport window are written to the frame buffer. Scissoring is a source of inefficiency, however, since effort is expended on pixels outside the viewing window. Nevertheless, it is the only practical alternative for clipping many types of complex primitives.

In the pipeline described here, all clipping is performed in homogeneous coordinates. This is really only necessary for z clipping, since the w value is needed to recognize vertices that lie behind the eye. Many systems clip to x and y boundaries after the homogeneous divide for efficiency. This simplifies x and y clipping, but still allows primitives that pass behind the eye to be recognized and clipped before w information is lost.

The number of computations required for clipping depends on how many primitives cross the clipping boundaries, which may change from one frame to the next. A common assumption is that only a small percentage of primitives (10 percent or fewer) need clipping. If this assumption is violated, system performance may decrease dramatically.

18.3.7 Division by w and Mapping to 3D Viewport

Homogeneous points that have had a perspective transformation applied, in general, have w values not equal to 1. To compute true x, y, and z values, we must divide the x, y, and z components of each homogeneous point by w. This requires 3 divisions per vertex. In many systems, vertex x and y coordinates must be mapped from the clipping coordinate system to the coordinate system of the actual 3D viewport. This is a simple scaling and translation operation in x and y that requires 2 multiplications and 2 additions per vertex.

18.3.8 Rasterization

The rasterization stage converts transformed primitives into pixel values, and generally stores them in a frame buffer. As discussed in Section 7.12.1, rasterization consists of three subtasks: *scan conversion*, *visible-surface determination*, and *shading*. Rasterization, in principle, requires calculating each primitive's contribution to each pixel, an $O(nm)$ operation, where n is the number of primitives and m is the number of pixels.

In a software rendering system, rasterization can be performed in either of two orders: primitive by primitive (object order), or pixel by pixel (image order). The pseudocode in Fig. 18.8 describes each of these two approaches, which correspond to the two main families of image-precision visibility algorithms: z-buffer and scan-line algorithms.

Most systems today rasterize in object order, using the z-buffer algorithm to compute

```
for (each primitive P)
    for (each pixel q within P)
        update frame buffer based on color
            and visibility of P at q;
```
(a)

```
for (each pixel q)
    for (each primitive P covering q)
        compute P's contribution to q,
            outputting q when finished;
```
(b)

Fig. 18.8 Pseudocode for (a) object-order and (b) image-order rasterization algorithms.

Samsung/Google, EX1027, p. 880

visibility. The z-buffer algorithm has only recently become practical with the availability of inexpensive DRAM memory. If we assume that visibility is determined using a z-buffer, the number of computations required in the rasterization stage still depends on the scan-conversion and shading methods. Scan-conversion calculations are difficult to categorize, but can be significant in a real system.

Constant shading requires no additional computations in the rasterization stage, since the polygon's uniform color was determined in the lighting stage. Gouraud shading requires red, green, and blue values to be bilinearly interpolated across each polygon. Incremental addition can be used to calculate RGB values for each succeeding pixel. Phong shading requires significantly more computation. The x, y, and z components of the surface normal must be linearly interpolated across polygons based on the normal vectors at each vertex. Since the interpolated vector at each pixel does not, in general, have unit length, it must be normalized. This requires 2 additions, an expensive reciprocal square-root operation, and 3 multiplications for every pixel. Then, the Phong illumination model must be evaluated, requiring a dot product, more multiplications and additions, and, in the general case, the evaluation of an exponential. Bishop and Weimer's approximation to Phong shading, as mentioned in Section 16.2.5, can reduce some of these calculations, albeit at the expense of realism for surfaces with high specularity. Other shading techniques, such as transparency and texturing, may be desired as well. The architectural implications of these techniques are discussed in Section 18.11.

In addition to the shading calculations, updating each pixel requires reading the z-buffer, comparing old and new z values, and, if the primitive is visible, writing new color and z values to the frame buffer.

18.3.9 Performance Requirements of a Sample Application

In Section 18.2, we mentioned the three major performance barriers for raster graphics systems: the number of floating-point operations required in geometry calculations and the number of pixel-oriented computations and frame-buffer accesses required in rasterization. To appreciate the magnitude of these problems, we shall estimate the number of computations needed at each stage of the rendering pipeline to display a sample database at a modest update rate. To make this estimate, we must define a representative database and application.

A sample database. We use triangles for primitives in our sample database, since they are common and their computation requirements are easy to estimate (complex primitives with shared vertices, such as triangle strips and quadrilateral meshes, are becoming increasingly popular, however. Exercise 18.2 explores the savings that can be achieved using these primitives.) We assume a modest database size of 10,000 triangles, a size that current workstations can display at interactive rates [AKEL88; APGA88; BORD89; MEGA89]. In addition, we assume that each triangle covers an average of 100 pixels.

The number of polygons that fall on clipping boundaries can vary widely from one frame to the next. For simplicity, we assume that no primitives need clipping in our sample application. This means we underestimate the amount of calculations required for clipping, but maximize the work required in succeeding stages. Image *depth complexity*, the average number of polygons mapping to a pixel, depends on the database as well as on the view. We

Samsung/Google, EX1027, p. 881

assume arbitrarily that one-half of the pixels of all the triangles are obscured by some other triangle.

We assume that an ambient/diffuse illumination model and Gouraud shading are to be applied to each primitive (this is the lowest common standard for current 3D systems, although Phong illumination combined with Gouraud shading is becoming increasingly popular). We assume a screen size of 1280 by 1024 pixels and an update rate of 10 frames per second, typical of current interactive applications, but far from ideal.

In summary, our sample application has the following characteristics:

- 10,000 triangles (none clipped)
- Each triangle covers an average of 100 pixels, one-half being obscured by other triangles
- Ambient and diffuse illumination models (not Phong)
- Gouraud shading
- 1280 by 1024 display screen, updated at 10 frames per second.

We cannot calculate all the computation and memory-bandwidth requirements in our sample application, since many steps are difficult to categorize. Instead, we concentrate on the three performance barriers: the number of floating-point operations for geometry computations, the number of integer operations for computing pixel values, and the number of frame-buffer accesses for rasterization.

Geometry calculations. For each frame, we must process $10{,}000 \cdot 3 = 30{,}000$ vertices and vertex-normal vectors. In the modeling transformation stage, transforming a vertex (including transforming the normal vector) requires 25 multiplications and 18 additions. The requirements for this stage are thus $30{,}000 \cdot 25 = 750{,}000$ multiplications and $30{,}000 \cdot 18 = 540{,}000$ additions.

Trivial accept/reject classification requires testing each vertex of each primitive against the six bounding planes of the viewing volume, a total of 24 multiplications and 18 additions per vertex. The requirements for this stage are thus $30{,}000 \cdot 24 = 720{,}000$ multiplications and $30{,}000 \cdot 18 = 540{,}000$ additions, regardless of how many primitives are trivially accepted or rejected.

Lighting requires 12 multiplications and 5 additions per vertex, a total of $30{,}000 \cdot 12 = 360{,}000$ multiplications and $30{,}000 \cdot 5 = 150{,}000$ additions.

The viewing transformation requires 8 multiplications and 6 additions per vertex, a total of $30{,}000 \cdot 8 = 240{,}000$ multiplications and $30{,}000 \cdot 6 = 180{,}000$ additions.

The requirements for clipping are variable; the exact number depends on the number of primitives that cannot be trivially accepted or rejected, which in turn depends on the scene and on the viewing angle. We have assumed the simplest case for our database, that all primitives lie completely within the viewing volume. If a large fraction of the primitives needs clipping, the computational requirements could be substantial (perhaps even more than in the geometric transformation stage).

Division by w requires 3 divisions per vertex, a total of $30{,}000 \cdot 3 = 90{,}000$ divisions. Mapping to the 3D viewport requires 2 multiplications and 2 additions per vertex, a total of 60,000 multiplications and 60,000 additions.

Samsung/Google, EX1027, p. 882

Summing the floating-point requirements for all of the geometry stages gives a total of 2,220,000 multiplications/divisions and 1,470,000 additions/subtractions per frame. Since a new frame is calculated every $\frac{1}{10}$ second, a total of 22.2 million multiplications/divisions and 14.7 million additions/subtractions (36.9 million aggregate floating-point operations) as required per second—a very substantial number.

Rasterization calculations and frame-buffer accesses. Let us now estimate the number of pixel calculations and frame-buffer memory accesses required in each frame. We assume that z values and RGB triples each occupy one word (32 bits) of frame-buffer memory (typical in most current high-performance systems). For each pixel that is initially visible (i.e., results in an update to the frame buffer), z, R, G, and B values are calculated (4 additions per pixel if forward differences are used), a z value is read from the frame buffer (1 frame-buffer cycle), the z values are compared (1 subtraction) and new z values and colors are written (2 frame-buffer cycles). For each pixel that is initially not visible, only the z value needs to be calculated (1 addition), and a z value is read from the frame buffer (1 frame-buffer cycle), and the two z values are compared (1 subtraction). Note that initially visible pixels may get covered, but initially invisible pixels can never be exposed.

Since we assume that one-half of the pixels of each triangle are visible in the final scene, a reasonable guess is that three-quarters of the pixels are initially visible and one-quarter of the pixels are initially invisible. Each triangle covers 100 pixels, so $\frac{3}{4} \cdot 100 \cdot 10{,}000 = 750{,}000$ pixels are initially visible and $\frac{1}{4} \cdot 100 \cdot 10{,}000 = 250{,}000$ pixels are initially invisible. To display an entire frame, therefore, a total of $(750{,}000 \cdot 5) + (250{,}000 \cdot 2) = 4.25$ million additions and $(750{,}000 \cdot 3) + (250{,}000 \cdot 1) = 2.5$ million frame-buffer accesses is required. To initialize each frame, both color and z-buffers must be cleared, an additional $1280 \cdot 1024 \cdot 2 = 2.6$ million frame-buffer accesses. The total number of frame-buffer accesses per frame, therefore, is 2.5 million + 2.6 million = 5.1 million. If 10 frames are generated per second, 42.5 million additions and 51 million frame-buffer accesses are required per second.

In 1989, the fastest floating-point processors available computed approximately 20 million floating-point operations per second, the fastest integer processors computed approximately 40 million integer operations per second, and DRAM memory systems had cycle times of approximately 100 nanoseconds. The floating-point and integer requirements of our sample application, therefore, are just at the limit of what can be achieved in a single CPU. The number of frame-buffer accesses, however, is much higher than is possible in a conventional memory system. As we mentioned earlier, this database is only modestly sized for systems available in 1989. In the following sections, we show how multiprocessing can be used to achieve the performance necessary to display databases that are this size and larger.

18.4 INTRODUCTION TO MULTIPROCESSING

Displaying large databases at high frame rates clearly requires dramatic system performance, both in terms of computations and of memory of bandwidth. We have seen that the geometry portion of a graphics system can require more processing power than a single CPU can provide. Likewise, rasterization can require more bandwidth into memory than a single memory system can provide. The only way to attain such performance levels is to

Samsung/Google, EX1027, p. 883

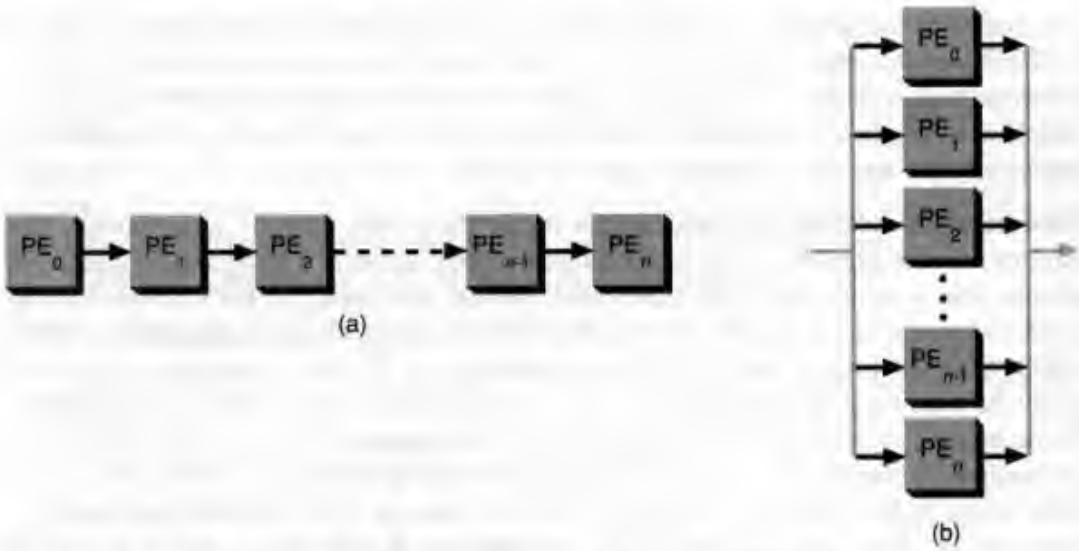


Fig. 18.9 Basic forms of multiprocessing: (a) pipelining, and (b) parallelism.

perform multiple operations concurrently and to perform multiple reads and writes to memory concurrently—we need concurrent processing.

Concurrent processing, or *multiprocessing*, is the basis of virtually all high-performance graphics architectures. Multiprocessing has two basic forms: *pipelining* and *parallelism* (we reserve the term *concurrency* for multiprocessing in general). A *pipeline processor* contains a number of *processing elements* (PEs) arranged such that the output of one becomes the input of the next, in pipeline fashion (Fig. 18.9a). The PEs of a *parallel processor* are arranged side by side and operate simultaneously on different portions of the data (Fig. 18.9b).

18.4.1 Pipelining

To pipeline a computation, we partition it into stages that can be executed sequentially in separate PEs. Obviously, a pipeline can run only as fast as its slowest stage, so the processing load should be distributed evenly over the PEs. If this is not possible, PEs can be sized according to the jobs they must perform.

An important issue in pipeline systems is *throughput* versus *latency*. Throughput is the overall rate at which data are processed; latency is the time required for a single data element to pass from the beginning to the end of the pipeline. Some calculations can be pipelined using a large number of stages to achieve very high throughput. Pipeline latency increases with pipeline length, however, and certain computations can tolerate only a limited amount of latency. For example, real-time graphics systems, such as flight simulators, must respond quickly to changes in flight controls. If more than one or two frames are in the rendering pipeline at once, the system's interactivity may be impaired, regardless of the frame rate.

Samsung/Google, EX1027, p. 884

18.4.2 Parallelism

To parallelize a computation, we partition the data into portions that can be processed independently by different PEs. Frequently, PEs can execute the same program. *Homogeneous* parallel processors contain PEs of the same type; *heterogeneous* parallel processors contain PEs of different types. In any parallel system, the overall computation speed is determined by the time required for the slowest PE to finish its task. It is important, therefore, to balance the processing load among the PEs.

A further distinction is useful for homogeneous parallel processors: whether the processors operate in *lock step* or independently. Processors that operate in lock step generally share a single code store and are called *single-instruction multiple-data* (SIMD) processors. Processors that operate independently must have a separate code store for each PE and are called *multiple-instruction multiple-data* (MIMD) processors.

SIMD processors. Because all the PEs in a SIMD processor share a single code store, SIMD processors are generally less expensive than MIMD processors. However, they do not perform well on algorithms that contain conditional branches or that access data using pointers or indirection. Since the path taken in a conditional branch depends on data specific to a PE, different PEs may follow different branch paths. Because all the PEs in a SIMD processor operate in lock step, they all must follow every possible branch path. To accommodate conditional branches, PEs generally contain an *enable register* to qualify write operations. Only PEs whose enable registers are set write the results of computations. By appropriately setting and clearing the enable register, PEs can execute conditional branches (see Fig. 18.10a).

Algorithms with few conditional branches execute efficiently on SIMD processors. Algorithms with many conditional branches can be extremely inefficient, however, since

```
statement 1;
if not condition then
  enable = FALSE;
statement 2;
toggle enable;
statement 3;
statement 4;
enable = TRUE;
statement 5;
statement 6;
```

Total operations:
10 if condition evaluates TRUE,
10 if condition evaluates FALSE

(a)

```
statement 1;
if condition then
  statement 2;
else
  begin
    statement 3;
    statement 4;
  end;
statement 5;
statement 6;
```

Total operations:
5 if condition evaluates TRUE,
6 if condition evaluates FALSE

(b)

Fig. 18.10 (a) SIMD and (b) MIMD expressions of the same algorithm. In a SIMD program, conditional branches transform into operations on the enable register. When the enable register of a particular PE is FALSE, the PE executes the current instruction, but does not write the result.

Samsung/Google, EX1027, p. 885

most PEs may be disabled at any given time. Data structures containing pointers (such as linked lists or trees) or indexed arrays cause similar problems. Since a pointer or array index may contain a different value at each PE, all possible values must be enumerated to ensure that each PE can make its required memory reference. For large arrays or pointers, this is an absurd waste of processing resources. A few SIMD processors provide separate address wires for each PE in order to avoid this problem, but this adds size and complexity to the system.

MIMD processors. MIMD processors are more expensive than SIMD processors, since each PE must have its own code store and controller. PEs in a MIMD processor often execute the same program. Unlike SIMD PEs, however, they are not constrained to operate in lock step. Because of this freedom, MIMD processors suffer no disadvantage when they encounter conditional branches; each PE makes an independent control-flow decision, skipping instructions that do not need to be executed (see Fig. 18.10b). As a result, MIMD processors achieve higher efficiency on general types of computations. However, since processors may start and end at different times and may process data at different rates, synchronization and load balancing are more difficult, frequently requiring FIFO buffers at the input or output of each PE.

18.4.3 Multiprocessor Graphics Systems

Pipeline and parallel processors are the basic building blocks of virtually all current high-performance graphics systems. Both techniques can be used to accelerate front-end and back-end subsystems of a graphics system, as shown in Fig. 18.11.

In the following sections, we examine each of these strategies. Sections 18.5 and 18.6 discuss pipeline and parallel front-end architectures. Sections 18.8 and 18.9 discuss pipeline and parallel back-end architectures. Section 18.10 discusses back-end architectures that use parallel techniques in combination.

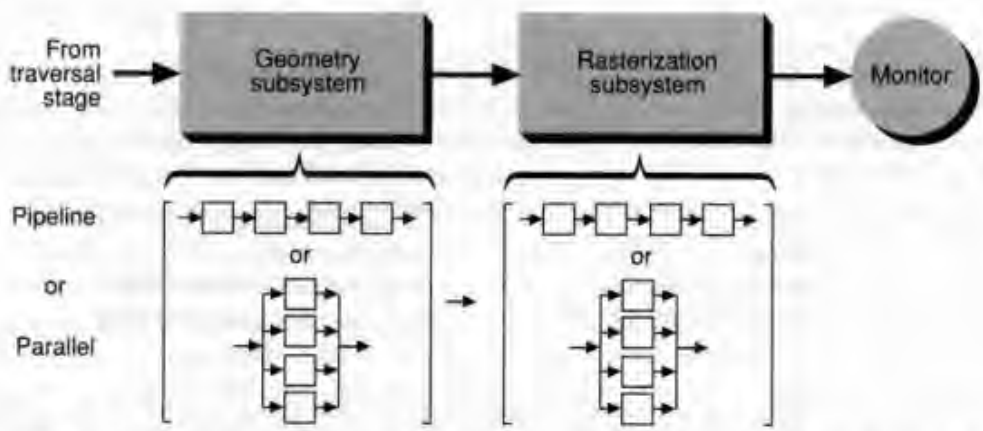


Fig. 18.11 Pipelining and parallelism can be used to accelerate both front-end and back-end portions of a graphics system.

Samsung/Google, EX1027, p. 886

18.5 PIPELINE FRONT-END ARCHITECTURES

Recall from Section 18.3 that the front end of a graphics display system has two major tasks: traversing the display model and transforming primitives into screen space. As we have seen, to achieve the rendering rates required in current applications, we must use concurrency to speed these computations. Both pipelining and parallelism have been used for decades to build front ends of high-performance graphics systems. Since the front end is intrinsically pipelined, its stages can be assigned to separate hardware units. Also, the large numbers of primitives in most graphics databases can be distributed over multiple processors and processed in parallel. In this section, we discuss pipeline front-end systems. We discuss parallel front-end systems in Section 18.6.

In introducing the standard graphics pipeline of Fig. 18.7, we mentioned that it provides a useful conceptual model of the rendering process. Because of its linear nature and fairly even allocation of processing effort, it also maps well onto a physical pipeline of processors. This has been a popular approach to building high-performance graphics systems since the 1960s, as described in [MYER68], a classic paper on the evolution of graphics architectures. Each stage of the pipeline can be implemented in several ways: as an individual general-purpose processor, as a custom hardware unit, or as a pipeline or parallel processor itself. We now discuss implementations for each stage in the front-end pipeline.

18.5.1 Application Program and Display Traversal

Some processor must execute the application program that drives the entire graphics system. In addition to feeding the graphics pipeline, this processor generally handles input devices, file I/O, and all interaction with the user. In systems using immediate-mode traversal, the display model is generally stored in the CPU's main memory. The CPU must therefore traverse the model as well as run the application. In systems using retained mode, the model is generally (but not always) stored in the display processor's memory, with the display processor performing traversal. Because such systems use two processors for these tasks, they are potentially faster, although they are less flexible and have other limitations, as discussed in Section 7.2.2.

Where very high performance is desired, a single processor may not be powerful enough to traverse the entire database with sufficient speed. The only remedy is to partition the database and to traverse it in parallel. This relatively new technique is discussed in Section 18.6.1.

18.5.2 Geometric Transformation

The geometric transformation stages (modeling transformation and viewing transformation) are highly compute-intensive. Fortunately, vector and matrix multiplications are simple calculations that require no branching or looping, and can readily be implemented in hardware.

The most common implementation of these stages is a single processor or functional unit that sequentially transforms a series of vertices. A pioneering processor of this type was the Matrix Multiplier [SUTH68], which could multiply a four-element vector by a

Samsung/Google, EX1027, p. 887

homogeneous transformation matrix in 20 microseconds. Other special-purpose geometry processors have been developed since then, most notably Clark's Geometry Engine, which can perform clipping as well (see Section 18.5.5). Recent geometry processors have exploited the power and programmability of commercial floating-point chips.

If pipelining does not provide enough performance, transformation computations can be parallelized in several ways:

- Individual components of a vertex may be calculated in parallel. Four parallel processors, each containing the current transformation matrix, can evaluate the expressions for x, y, z, and w in parallel.
- Multiple vertices can be transformed in parallel. If primitives are all of a uniform type—say, triangles—the three vertices of each triangle can be transformed simultaneously.
- Entire primitives can be transformed in parallel. If n transformation engines are available, each processor can transform every nth primitive. This technique has many of the advantages and disadvantages of parallel front-end systems, which we will discuss in Section 18.6.

18.5.3 Trivial Accept/Reject Classification

Trivial accept and reject tests are straightforward to implement, since they require at worst a dot product and at best a single floating-point comparison (or subtract) to determine on which side of each clipping plane each vertex lies. Because these tests require little computation, they are generally performed by the processor that transforms primitives.

18.5.4 Lighting

Like geometric transformation, lighting calculations are straightforward and are floating-point–intensive. A specialized hardware processor can calculate vertex colors based on a polygon's color and the light vector. More frequently, lighting calculations are performed using a programmable floating-point processor. In lower-performance systems, lighting calculations can be done in the same processor that transforms vertices. Note that if Phong shading is used, lighting calculations are deferred until the rasterization stage.

18.5.5 Clipping

Polygon clipping was once considered cumbersome, since the number of vertices can change during the clipping process and concave polygons can fragment into multiple polygons during clipping. Sutherland and Hodgman [SUTH74] showed that arbitrary convex or concave polygons can be clipped to a convex view volume by passing the polygon's vertices through a single processing unit multiple times. Each pass through the unit clips the polygon to a different plane. In 1980, Clark proposed unwrapping this processing loop into a simple pipeline of identical processors, each of which could be implemented in a single VLSI chip, which he named the *Geometry Engine* [CLAR82]. The Geometry Engine was general enough that it could transform primitives and perform perspective division as well.

Samsung/Google, EX1027, p. 888

Clipping using a Geometry Engine (or similar processor) can be performed either by a single processor that clips each polygon by as many planes as necessary, or by a pipeline of clipping processors, one for each clipping plane. The technique chosen affects the worst-case performance of the graphics system: Systems with only one clipping processor may bog down during frames in which large numbers of primitives need to be clipped, whereas systems with a clipping processor for each clipping plane can run at full speed. However, most of the clipping processors are idle for most databases and views in the latter approach.

General-purpose floating-point units recently have begun to replace custom VLSI transformation and clipping processors. For example, Silicon Graphics, which for many years employed custom front-end processors, in 1989 used the Weitek 3332 floating-point chip for transformations and clipping in their POWER IRIS system (described in detail in Section 18.8.2). The delicate balance between performance and cost now favors commodity processors. This balance may change again in the future if new graphics-specific functionality is needed and cannot be incorporated economically into general-purpose processors.

18.5.6 Division by *w* and Mapping to 3D Viewpoint

Like geometric transformation and lighting, the calculations in this stage are straightforward but require substantial floating-point resources. A floating-point divide is time consuming even for most floating-point processors (many processors use an iterative method to do division). Again, these stages can be implemented in custom functional units or in a commercial floating-point processor. In very high-performance systems, these calculations can be performed in separate, pipelined processors.

18.5.7 Limitations of Front-End Pipelines

Even though pipelining is the predominant technique for building high-performance front-end systems, it has several limitations that are worth considering. First, a different algorithm is needed for each stage of the front-end pipeline. Thus, either a variety of hard-wired functional units must be designed or, if programmable processors are used, different programs must be written and loaded into each processor. In either case, processor or functional-unit capabilities must be carefully matched to their tasks, or bottlenecks will occur.

Second, since the rendering algorithm is committed to hardware (or at least to firmware, since few systems allow users to reprogram pipeline processors), it is difficult to add new features. Even if users have programming support for the pipeline processors, the distribution of hardware resources in the system may not adequately support new features such as complex primitives or collision detection between primitives.

A final shortcoming of pipelined front ends is that the approach breaks down when display traversal can no longer be performed by a single processor, and this inevitably occurs at some performance level. For example, if we assume that traversal is performed by a 20-MHz processor and memory system, that the description of each triangle in the database requires 40 words of data (for vertex coordinates, normal vectors, colors, etc.), and that each word sent to the pipeline requires two memory/processor cycles (one to read it

Samsung/Google, EX1027, p. 889

from memory, another to load it into the pipeline), then a maximum of 20,000,000 / (2 · 40) = 250,000 triangles per second can be displayed by the system, no matter how powerful the processors in the pipeline are. Current systems are rapidly approaching such limits.

What else can be done, then, to achieve higher performance? The alternative to pipelining front-end calculations is to parallelize them. The following section describes this second way to build high-performance front-end systems.

18.6 PARALLEL FRONT-END ARCHITECTURES

Since graphics databases are regular, typically consisting of a large number of primitives that receive nearly identical processing, an alternate way to add concurrency is to partition the data into separate streams and to process them independently. For most stages of the front-end subsystem, such partitioning is readily done; for example, the geometric-transformation stages can use any of the parallel techniques described in Section 18.5.2. However, stages in which data streams diverge (display traversal) or converge (between the front end and back end) are problematic, since they must handle the full data bandwidth.

18.6.1 Display Traversal

Almost all application programs assume a single, contiguous display model or database. In a parallel front-end system, the simplest technique is to traverse the database in a single processor (serial traversal) and then to distribute primitives to the parallel processors. Unfortunately, this serial traversal can become the bottleneck in a parallel front-end system. Several techniques can be used to accelerate serial traversal:

- Traversal routines can be optimized or written in assembly code
- The database can be stored in faster memory (i.e., SRAM instead of DRAM)
- A faster traversal processor (or one optimized for the particular structure format) can be used.

If these optimizations are not enough, the only alternative is to traverse the database in parallel. The database either can be stored in a single memory system that allows parallel access by multiple processors (a shared-memory model), or can be distributed over multiple processors, each with its own memory system (a distributed-memory model).

The advantage of the shared-memory approach is that the database can remain in one place, although traversal must be divided among multiple processors. Presumably, each processor is assigned a certain portion of the database to traverse. Unfortunately, inherited attributes in a hierarchical database model mean that processors must contend for access to the same data. For example, each processor must have access to the current transformation matrix and to other viewing and lighting parameters. Since the data bandwidth to and from a shared-memory system may not be much higher than that of a conventional memory system, the shared-memory approach may not provide enough performance.

In the distributed-memory approach, each processor contains a portion of the database in its local memory. It traverses its portion of the database for each frame and may also perform other front-end computations. Distributing the database presents its own problems, however: Unless the system gives the application programmer the illusion of a contiguous database, it cannot support portable graphics libraries. Also, the load must be balanced

Samsung/Google, EX1027, p. 890

over the traversal processors if system resources are to be utilized fully. Hierarchical databases exacerbate both of these problems, since attributes in one level of a hierarchy affect primitives below them, and structures deep in a hierarchy may be referenced by multiple higher-level structure calls.

The following two sections examine two ways to distribute a hierarchical database over multiple processors: by structure, where each traversal processor is given a complete branch of the structure hierarchy; or by primitive, where each traversal processor is given a fraction of the primitives at each block in the hierarchy.

Distributing by structure. Distributing by structure is outwardly appealing, since state-changing elements in the structure apparently need to be stored only once. This can be an illusion, however, since multiple high-level structures may refer to the same lower-level substructure. For example, a database containing several cars, each described by a separate *car* structure, can be distributed by assigning each *car* structure to a separate processor. However, if each *car* structure refers to a number of *wheel* structures, *wheel* structures must also be replicated at every processor.

Load balancing among processors is also difficult. Since primitives in a structure are likely to be spatially coherent, changing the viewpoint or geometry within a scene may cause entire portions of the structure to be clipped or to reappear. Maintaining even loading among the multiple processors would require reassigning portions of the database dynamically.

Distributing by primitive. Distributing by primitive is costly, since the entire hierarchical structure of the database and any state-changing commands must be replicated at each processor. Structure editing is also expensive, since changes must be broadcast to every processor. Load balancing, however, is automatic. Since objects in a hierarchical database typically contain a large number of simple primitives (e.g., polygons forming a tiled surface), these primitives will be scattered over all the processors, and each processor will have a similar processing load.

Parallel display traversal is a relatively new technique. In 1989, the highest-performance architectures were just approaching the point where serial traversal becomes insufficient, and only a few systems had experimented with parallel traversal [FUCH89]. Neither of the distribution techniques for hierarchical databases that we have described is ideal. Compared to geometry processing, which easily partitions into parallel tasks, display traversal is much more difficult. Nevertheless, parallel traversal is likely to become increasingly important as system performance levels increase.

18.6.2 Recombining Parallel Streams

The transition between the front-end and back-end portions of the rendering pipeline is troublesome as well. In a parallel front-end system, the multiple streams of transformed and clipped primitives must be directed to the processor or processors doing rasterization. This can require sorting primitives based on spatial information if different processors are assigned to different screen regions.

A second difficulty in parallel front-end systems is that the ordering of data may change as those data pass through parallel processors. For example, one processor may transform two small primitives before another processor transforms a single, larger one. This does not

Samsung/Google, EX1027, p. 891

matter for many graphics primitives and rendering techniques. Certain global commands, however, such as commands to update one window instead of another or to switch between double buffers, require that data be synchronized before and after the command. If a large number of commands such as these occurs, some type of hardware support for synchronization may be necessary. A Raster Technologies system [TORB87] incorporates a special FIFO into each PE that stores tag codes for each command and allows commands to be resynchronized after they have been processed in separate PEs.

18.6.3 Pipelining versus Parallelism

We have seen that both pipelining and parallelism can be used to build high-performance front-end subsystems. Although pipelining has been the predominant technique in systems of the last decade, parallelism offers several advantages, including reconfigurability for different algorithms, since a single processor handles all front-end calculations, and more modularity, since PEs in a parallel system can be made homogeneous more easily than in a pipeline system. Because the performance of a pipeline system is limited by the throughput of its slowest stage, pipelines do not scale up as readily as do parallel systems. Parallel systems, on the other hand, require more complicated synchronization and load balancing and cannot use specialized processors as well as can pipelined systems. Both designs are likely to be useful in the future; indeed, the highest-performance systems are likely to combine the two.

18.7 MULTIPROCESSOR RASTERIZATION ARCHITECTURES

Recall that the output of the front-end subsystem is typically a set of primitives in screen coordinates. The rasterization (back-end) subsystem creates the final image by scan converting each of these primitives, determining which primitives are visible at each pixel, and shading the pixel accordingly. Section 18.2.4 identified two basic reasons why simple display-processor/frame-buffer systems are inadequate for high-performance rasterization subsystems:

1. A single display processor does not have enough processing power for all the pixel calculations.
2. Memory bandwidth into the frame buffer is insufficient to handle the pixel traffic—even if the display processor could compute pixels rapidly enough.

Much of the research in graphics architecture over the past decade has concerned ways to overcome these limitations. A great variety of techniques has been proposed, and many have been implemented in commercial and experimental systems. In this section, we consider low-cost, moderate-performance architectures that cast conventional algorithms into hardware. In Sections 18.8 and 18.9, we consider ways to improve performance by adding large amounts of parallelism to speed the calculation of the algorithm's "inner loop." In Section 18.10, we consider hybrid architectures that combine multiple techniques for improved efficiency or even higher performance. Figure 18.12 summarizes the concurrent approaches we shall discuss here.

Samsung/Google, EX1027, p. 892

Rasterization algorithm	*Architectural technique*		
	Serial pipeline	Highly parallel	Hybrid
Object order	Pipelined object order	Image parallel	Virtual buffer/ virtual processor
z-buffer, depth-sort, and BSP-tree algorithms	Polygon/edge/span-processor pipeline	Partitioned image memory Logic-enhanced memory	Parallel virtual buffer
Image order	Pipelined image order	Object parallel	
Scanline algorithms	Scan-line pipeline	Processor per primitive pipeline Tree-structured	Image composition

Fig. 18.12 Taxonomy of concurrent rasterization approaches.

18.7.1 Pipelined Object-Order Architectures

A direct way to add concurrency to rasterization calculations is to cast the various steps of a software algorithm into a hardware pipeline. This technique has been used to build a number of inexpensive, moderately high-performance systems. This approach can be used with either of the two main rasterization approaches: object order (*z*-buffer, depth-sort, and BSP-tree algorithms) and image order (scan-line algorithms). We consider object-order rasterization now and image-order rasterization in Section 18.7.2.

Object-order rasterization methods include the *z*-buffer, depth-sort, and BSP-tree algorithms (the *z*-buffer is by far the most common in 3D systems). The outer loop of these algorithms is an enumeration of primitives in the database, and the inner loop is an enumeration of pixels within each primitive. For polygon rendering, the heart of each of these algorithms is rasterizing a single polygon.

Figure 18.13 shows the most common rasterization algorithm for convex polygons. This algorithm is an extension of the 2D polygon scan-conversion algorithm presented in Section 3.6, using fixed-point arithmetic rather than integer arithmetic. Delta values are used to calculate the expressions for x, z, R, G, and B incrementally from scan line to scan line, and from pixel to pixel. We shall describe each step of the algorithm.

Polygon processing. Computations performed only once per polygon are grouped into this stage. The first step is to determine the initial scan line intersected by the polygon (this is determined by the vertex with the smallest y value). In most cases, the polygon intersects this scan line at a single pixel, with two edges projecting upward, the *left* and *right* edges. Delta values are calculated for x, z, R, G, and B for each edge. These delta values are sometimes called *slopes*.

Edge processing. Computations performed once for each scan line are grouped here. Scan lines within each primitive are processed one by one. The delta values computed previously are used to calculate x, z, R, G, and B values at the intersection points of the left and right edges with the current scan line (P_{left} and P_{right} in the figure). A contiguous sequence of pixels on a scan line, such as those between P_{left} and P_{right}, is called a *span*. Delta

Samsung/Google, EX1027, p. 893

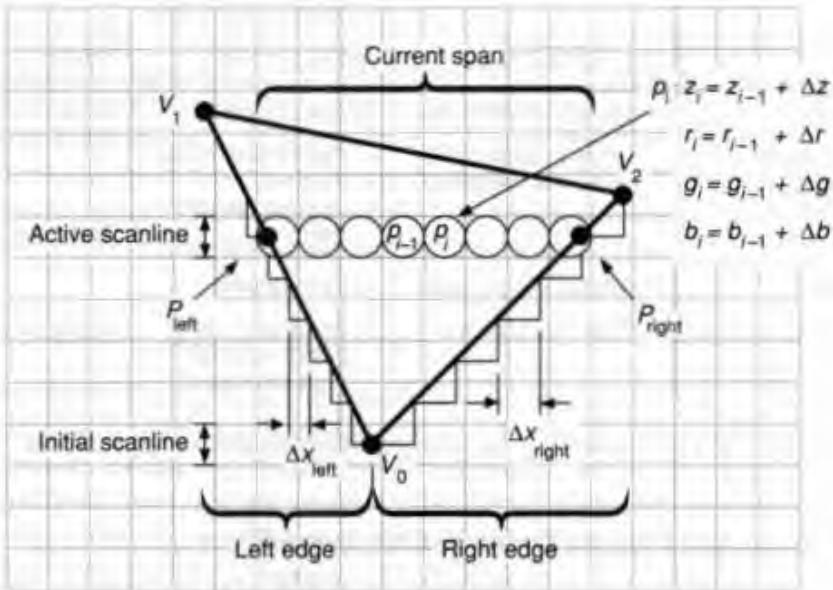


Fig. 18.13 Rasterizing a triangle. Each vertex (V_0, V_1, and V_2), span endpoint (P_{left} and P_{right}), and pixel (p_0, p_1, etc.) has z, R, G, and B components.

values for incrementing z, R, G, and B from pixel to pixel within the span are then calculated from the values at P_{left} and P_{right}.

Span processing. Operations that must be performed for each pixel within each span are performed here. For each pixel within the span, z, R, G, and B values are calculated by adding delta values to the values at the preceding pixel. The z value is compared with the previous z value stored at that location; if it is smaller, the new pixel value replaces the old one.

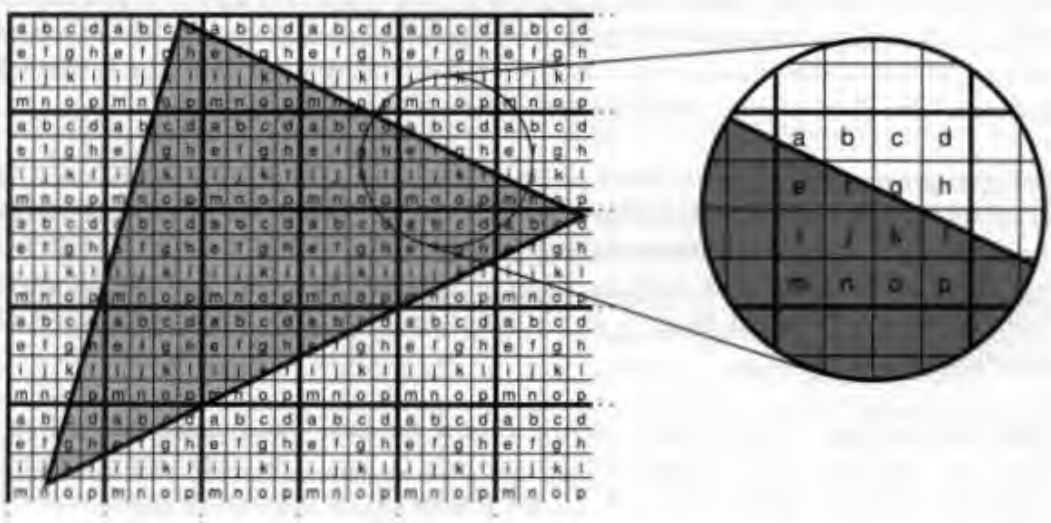


Fig. 18.14 A 4 × 4 interleaved memory organization. Each memory partition ("a" through "p") contains one pixel from each 4 × 4 block of pixels.

Samsung/Google, EX1027, p. 894