

EXHIBIT 1018

Computer Graphics

PRINCIPLES AND PRACTICE

Foley ♦ van Dam ♦ Feiner ♦ Hughes

SECOND EDITION in C



THE SYSTEMS PROGRAMMING SERIES

LG-HISENSE00050051

Ex. 1018 - TCL Indus Holdings Co., Ltd.

TCL Indus Holdings Co., Ltd. v. LGE – IPR2023-00461

Page 2 of 36

THE
SYSTEMS
PROGRAMMING
SERIES

Computer Graphics: Principles and Practice

SECOND EDITION in C

Foley ♦ van Dam ♦ Feiner ♦ Hughes

LG-HISENSE00050052

Ex. 1018 - TCL Indus Holdings Co., Ltd.

TCL Indus Holdings Co., Ltd. v. LGE – IPR2023-00461

Page 3 of 36

SECOND EDITION IN C

Computer Graphics

PRINCIPLES AND PRACTICE

James D. Foley
Georgia Institute of Technology

Andries van Dam
Brown University

Steven K. Feiner
Columbia University

John F. Hughes
Brown University



ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts ♦ Menlo Park, California ♦ New York
Don Mills, Ontario ♦ Wokingham, England ♦ Amsterdam ♦ Bonn
Sydney ♦ Singapore ♦ Tokyo ♦ Madrid ♦ San Juan ♦ Milan ♦ Paris

LG-HISENSE00050053

Ex. 1018 - TCL Indus Holdings Co., Ltd.
TCL Indus Holdings Co., Ltd. v. LGE – IPR2023-00461

Page 4 of 36

Sponsoring Editor: Peter S. Gordon
Production Supervisor: Bette J. Aaronson
Production Supervisor for the C edition: Juliet Silveri
Copy Editor: Lyn Dupré
Text Designer: Herb Caswell
Technical Art Consultant: Joseph K. Vetere
Illustrators: C&C Associates
Cover Designer: Marshall Henrichs
Manufacturing Manager: Roy Logan

This book is in the **Addison-Wesley Systems Programming Series**
Consulting editors: IBM Editorial Board

Library of Congress Cataloging-in-Publication Data

Computer graphics: principles and practice / James D. Foley . . . [et al.]. — 2nd ed. in C.

p. cm.

Includes bibliographical references and index.

ISBN 0-201-84840-6

1. Computer graphics. I. Foley, James D., 1942--

T385.C5735 1996

006.6'6—dc20

95-13631

CIP

Reprinted with corrections, July 1997.

Cover: "Dutch Interior," after Vermeer, by J. Wallace, M. Cohen, and D. Greenberg, Cornell University
(Copyright © 1987 Cornell University, Program of Computer Graphics.)

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The programs and applications presented in this book have been included for their instructional value. They are not guaranteed for any particular purpose. The publisher and the author do not offer any warranties or representations, nor do they accept any liabilities with respect to the programs or applications.

Reprinted with corrections November 1992, November 1993, and July 1995.

Copyright © 1996, 1990 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

Text printed on recycled and acid-free paper.

ISBN 0201848406

10 1112131415 DOC 02 01 00 99

10th Printing July 1999

LG-HISENSE00050054

postprocessing phase can eliminate these edges, as discussed in Chapter 19. A polygon-clipping algorithm based on the parametric-line representation for clipping to upright rectangular clip regions is discussed, along with the Weiler algorithm for clipping polygons to polygons, in Section 19.1.

3.15 GENERATING CHARACTERS

3.15.1 Defining and Clipping Characters

There are two basic techniques for defining characters. The most general but most computationally expensive way is to define each character as a curved or polygonal outline and to scan convert it as needed. We first discuss the other, simpler way, in which each character in a given font is specified as a small rectangular bitmap. Generating a character then entails simply using a `copyPixel` to copy the character's image from an offscreen canvas, called a *font cache*, into the frame buffer at the desired position.

The font cache may actually be in the frame buffer, as follows. In most graphics systems in which the display is refreshed from a private frame buffer, that memory is larger than is strictly required for storing the displayed image. For example, the pixels for a rectangular screen may be stored in a square memory, leaving a rectangular strip of "invisible" screen memory. Alternatively, there may be enough memory for two screens, one of which is being refreshed and one of which is being drawn in, to double-buffer the image. The font cache for the currently displayed font(s) is frequently stored in such invisible screen memory because the display controller's `copyPixel` works fastest within local image memory. A related use for such invisible memory is for saving screen areas temporarily obscured by popped-up images, such as windows, menus, and forms.

The bitmaps for the font cache are usually created by scanning in enlarged pictures of characters from typesetting fonts in various sizes; a typeface designer can then use a paint program to touch up individual pixels in each character's bitmap as necessary. Alternatively, the type designer may use a paint program to create, from scratch, fonts that are especially designed for screens and low-resolution printers. Since small bitmaps do not scale well, more than one bitmap must be defined for a given character in a given font just to provide various standard sizes. Furthermore, each type face requires its own set of bitmaps. Therefore, a distinct font cache is needed for each font loaded by the application.

Bitmap characters are clipped automatically by SRGP as part of its implementation of `copyPixel`. Each character is clipped to the destination rectangle on a pixel-by-pixel basis, a technique that lets us clip a character at any row or column of its bitmap. For systems with slow `copyPixel` operations, a much faster, although cruder, method is to clip the character to even the entire string on an all-or-nothing basis by doing a trivial accept of the character or string extent. Only if the extent is trivially accepted is the `copyPixel` applied to the character or string. For systems with a fast `copyPixel`, it is still useful to do trivial accept/reject testing of the string extent as a precursor to clipping individual characters during the `copyPixel` operation.

SRGP's simple bitmap font-cache technique stores the characters side by side in a canvas that is quite wide but is only as tall as the tallest character; Fig. 3.50 shows a portion

```

typedef point vertex;           /* point holds double x, y */
typedef vertex edge[2];
typedef vertex vertexArray[MAX]; /* MAX is a declared constant */

static void Output (vertex, int *, vertexArray);
static boolean Inside (vertex, edge);
static vertex Intersect (vertex, vertex, edge);

void SutherlandHodgmanPolygonClip (
    vertexArray inVertexArray, /* Input vertex array */
    vertexArray outVertexArray, /* Output vertex array */
    int inLength, /* No. of entries in inVertexArray */
    int *outLength, /* No. of entries in outVertexArray */
    edge clipBoundary) /* Edge of clip polygon */
{
    vertex s, p; /* Start, end pt. of current polygon edge */
    int i; /* Intersection pt. with a clip boundary */
    int j; /* Vertex loop counter */

    *outLength = 0; /* Start with the last vertex in inVertexArray */
    s = inVertexArray[inLength - 1];
    for (j = 0; j < inLength; j++) {
        p = inVertexArray[j]; /* Now s and p correspond to the vertices in Fig. 3.48 */
        if (Inside (p, clipBoundary)) { /* Cases 1 and 4 */
            if (Inside (s, clipBoundary)) /* Case 1 */
                Output (p, outLength, outVertexArray);
            else /* Case 4 */
                i = Intersect (s, p, clipBoundary);
                Output (i, outLength, outVertexArray);
                Output (p, outLength, outVertexArray);
        } else /* Cases 2 and 3 */
            if (Inside (s, clipBoundary)) { /* Case 2 */
                i = Intersect (s, p, clipBoundary);
                Output (i, outLength, outVertexArray);
            } /* No action for case 3 */
            s = p; /* Advance to next pair of vertices */
        } /* for */
    } /* SutherlandHodgmanPolygonClip */

    /* Adds newVertex to outVertexArray and then updates outLength */
    static void Output (vertex newVertex, int *outLength, vertexArray outVertexArray)
    {
        ...
    }

    /* Checks whether the vertex lies inside the clip edge or not */
    static boolean Inside (vertex testVertex, edge clipBoundary)

```

Fig. 3.49 (Cont.)

```

/* Clips polygon edge (first, second) against clipBoundary, outputs the new point */
static vertex Intersect (vertex first, vertex second, edge clipBoundary)

```

Fig. 3.49 Sutherland–Hodgman polygon-clipping algorithm.

of the cache, along with discrete instances of the same characters at low resolution. Each loaded font is described by a struct (declared in Fig. 3.51) containing a reference to the canvas that stores the characters' images, along with information on the height of the characters and the amount of space to be placed between adjacent characters in a text string. Some packages store the space between characters as part of a character's width, to allow variable intercharacter spacing.)

As described in Section 2.1.5, descender height and total height are constants for a given font—the former is the number of rows of pixels at the bottom of the font cache used only by descenders, and the latter is simply the height of the font-cache canvas. The width of a character, on the other hand, is not considered a constant; thus, a character can occupy the space that suits it, rather than being forced into a fixed-width character box. SRGP puts a fixed amount of space between characters when it draws a text string, the amount being specified as part of each font's descriptor. A word-processing application can display lines of text by using SRGP to display individual words of text, and can right-justify lines by using variable spacing between words and after punctuation to fill out lines so that their rightmost characters are aligned at the right margin. This involves using the text-extent inquiry facilities to determine where the right edge of each word is, in order to calculate the start of the next word. Needless to say, SRGP's text-handling facilities are really too crude for sophisticated word processing, let alone for typesetting programs, since such applications require far finer control over the spacing of individual letters to deal with such effects as sub- and superscripting, kerning, and printing text that is not horizontally aligned.

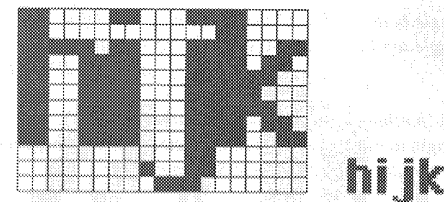


Fig. 3.50 Portion of an example of a font cache.

```
typedef struct {
    int leftX, width;           /* Horizontal location, width of image in font cache */
} charLocation;

typedef struct {
    canvasID cache;
    int descenderHeight, totalHeight; /* Height is a constant; width varies */
    int interCharacterSpacing;        /* Measured in pixels */
    charLocation locationTable[128];   /* Explained in the text */
} fontCacheDescriptor;
```

Fig. 3.51 Type declarations for the font cache.

3.15.2 Implementation of a Text Output Primitive

In the code of Fig. 3.52, we show how SRGP text is implemented internally: Each character in the given string is placed individually, and the space between characters is dictated by the appropriate field in the font descriptor. Note that complexities such as dealing with mixed fonts in a string must be handled by the application program.

```
void SRGP.characterText (
    point origin,           /* Where to place the character in the current canvas */
    char *stringToPrint,
    fontCacheDescriptor fontInfo)
{
    int i;

    /* Origin specified by the application is for baseline and does not include descender. */
    origin.y -= fontInfo.descenderHeight;

    for (i = 0; i < strlen (stringToPrint); i++) {
        rectangle fontCacheRectangle;
        char charToPrint = stringToPrint[i];
        /* Find the rectangular region within the cache wherein the character lies */
        charLocation *fip = &fontInfo.locationTable[charToPrint];

        fontCacheRectangle.bottomLeft = SRGP.defPoint (fip->leftX, 0);
        fontCacheRectangle.topRight = SRGP.defPoint (fip->leftX + fip->width - 1,
                                                    fontInfo.totalHeight - 1);

        SRGP.copyPixel (fontInfo.cache, fontCacheRectangle, origin);
        /* Update the origin to move past the new character plus intercharacter spacing */
        origin.x += fip->width + interCharacterSpacing;
    }
} /* SRGP.characterText */
```

Fig. 3.52 Implementation of character placement for SRGP's text primitive.

We mentioned that the bitmap technique requires a distinct font cache for each combination of font, size, and face for each different resolution of display or output device supported. A single font in eight different point sizes and four faces (normal, bold, italic, bold italic) thus requires 32 font caches! Figure 3.53(a) shows a common way of allowing a single font cache to support multiple face variations: The italic face is approximated by splitting the font's image into regions with horizontal "cuts," then offsetting the regions while performing a sequence of calls to SRGP_copyPixel.

This crude approximation does not make pleasing characters; for example, the dot over the "i" is noncircular. More of a problem for the online user, the method distorts the intercharacter spacing and makes picking much more difficult. A similar trick to achieve boldface is to copy the image twice in or mode with a slight horizontal offset (Fig. 3.53b). These techniques are not particularly satisfactory, in that they can produce illegible characters, especially when combined with subscripting.

A better way to solve the storage problem is to store characters in an abstract, device-independent form using polygonal or curved outlines of their shapes defined with floating-point parameters, and then to transform them appropriately. Polynomial functions called *splines* (see Chapter 11) provide smooth curves with continuous first and higher derivatives and are commonly used to encode text outlines. Although each character definition takes up more space than its representation in a font cache, multiple sizes may be derived from a single stored representation by suitable scaling; also, italics may be quickly approximated by shearing the outline. Another major advantage of storing characters in a completely device-independent form is that the outlines may be arbitrarily translated, rotated, scaled, and clipped (or used as clipping regions themselves).

The storage economy of splined characters is not quite so great as this description suggests. For instance, not all point sizes for a character may be obtained by scaling a single abstract shape, because the shape for an aesthetically pleasing font is typically a function of point size; therefore, each shape suffices for only a limited range of point sizes. Moreover, even conversion of splined text requires far more processing than the simple copyPixel implementation, because the device-independent form must be converted to pixel coordinates on the basis of the current size, face, and transformation attributes. Thus, the font-cache technique is still the most common for personal computers and even is used for many workstations. A strategy that offers the best of both methods is to store the fonts in outline form but to convert the ones being used in a given application to their bitmap equivalents—for example, to build a font cache on the fly. We discuss processing of splined text in more detail in Section 19.4.

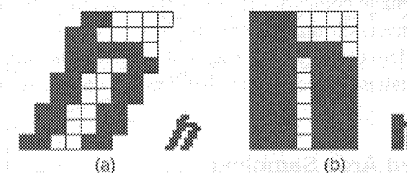


Fig. 3.53 Tricks for creating different faces for a font: (a) Italic. (b) Bold.

- 3.31 Explain why the Sutherland-Hodgman polygon-clipping algorithm works for only convex clipping regions.
- 3.32 Devise a strategy for subdividing a pixel and counting the number of subpixels covered (at least to a significant degree) by a line, as part of a line-drawing algorithm using unweighted area sampling.
- 3.33 Create tables with various decreasing functions of the distance between pixel center and line center. Use them in the antialiased line algorithm of Fig. 3.62. Contrast the results produced with those produced by a box-filtered line.
- 3.34 Generalize the antialiasing techniques for lines to polygons. How might you handle nonpolygonal boundaries of curved primitives and characters?

4

Graphics Hardware

In this chapter, we describe how the important hardware elements of a computer graphics display system work. Section 4.1 covers hardcopy technologies: printers, pen plotters, electrostatic plotters, laser printers, ink-jet plotters, thermal-transfer plotters, and film recorders. The basic technological concepts behind each type of device are described briefly, and a concluding section compares the various devices. Section 4.2, on display technologies, discusses monochrome and color shadow-mask CRTs, the direct-view storage tube (DVST), liquid-crystal displays (LCDs), plasma panels, electroluminescent displays, and several more specialized technologies. Again, a concluding section discusses the pros and cons of the various display technologies.

Raster display systems, which can use any of the display technologies discussed here, are discussed in Section 4.3. A simple, straightforward raster system is first introduced, and is then enhanced with respect to graphics functionality and integration of raster- and general-purpose processors into the system address space. Section 4.4 describes the role of the look-up table and video controller in image display, color control, animation, and image mixing. The almost-obsolete vector (also called *random*, *calligraphic*, *stroke*) display system is discussed briefly in Section 4.5, followed in Section 4.6 by user interaction devices such as tablets, mice, touch panels, and so on. Again, operational concepts rather than technological details are stressed. Section 4.7 briefly treats image input devices, such as film scanners, by means of which an existing image can be input to a computer.

Figure 4.1 shows the relation of these hardware devices to one another. The key element is the integrated CPU and display processor known as a *graphics workstation*, typically consisting of a CPU capable of executing at least several million instructions per second (MIPS), a large disk, and a display with resolution of at least 1000 by 800. The

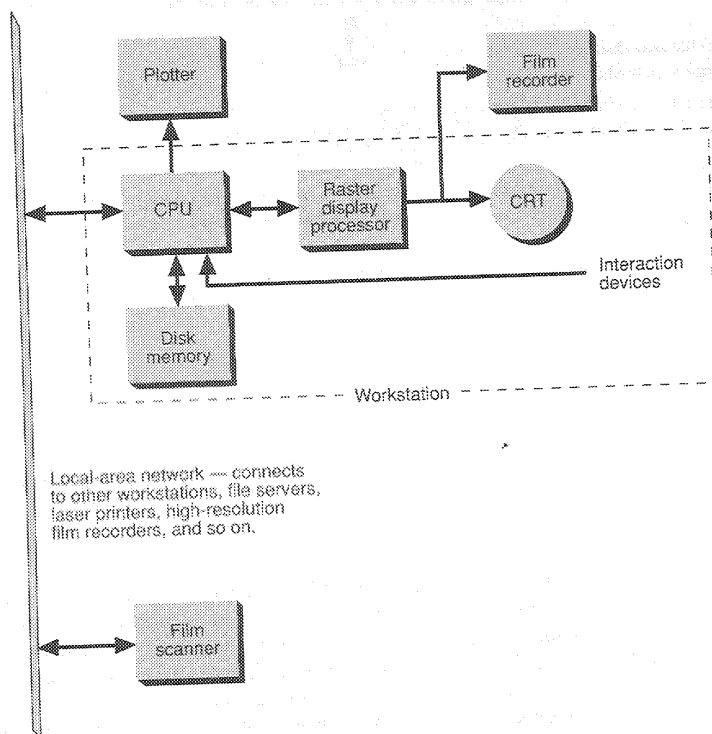


Fig. 4.1 Components of a typical interactive graphics system.

local-area network connects multiple workstations for file sharing, electronic mail, and access to shared peripherals such as high-quality film plotters, large disks, gateways to other networks, and higher-performance computers.

4.1 HARDCOPY TECHNOLOGIES

In this section, we discuss various hardcopy technologies, then summarize their characteristics. Several important terms must be defined first.

The image quality achievable with display devices depends on both the addressability and the dot size of the device. *Dot size* (also called *spot size*) is the diameter of a single dot on the device's output. *Addressability* is the number of individual (not necessarily distinguishable) dots per inch that can be created; it may differ in the horizontal and vertical directions. Addressability in x is just the reciprocal of the distance between the centers of dots at addresses (x, y) and $(x + 1, y)$; addressability in y is defined similarly. *Interdot distance* is the reciprocal of addressability.

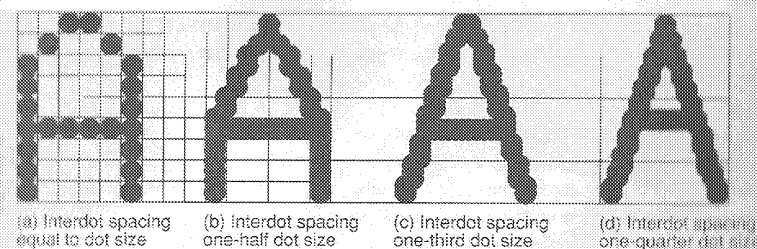


Fig. 4.2 The effects of various ratios of the dot size to the interdot distance.

It is usually desirable that dot size be somewhat greater than the interdot distance, so that smooth shapes can be created. Figure 4.2 illustrates this reasoning. Tradeoffs arise here, however: dot size several times the interdot distance allows very smooth shapes to be printed, whereas a smaller dot size allows finer detail.

Resolution, which is related to dot size and can be no greater than addressability, is the number of distinguishable lines per inch that a device can create. Resolution is defined as the closest spacing at which adjacent black and white lines can be distinguished by observers (this again implies that horizontal and vertical resolution may differ). If 40 black lines interleaved with 40 white lines can be distinguished across one inch, the resolution is 80 lines per inch (also referred to as 40 line-pairs per inch).

Resolution also depends on the cross-sectional intensity distribution of a spot. A spot with sharply delineated edges yields higher resolution than does one with edges that trail off, as shown in Fig. 4.3.

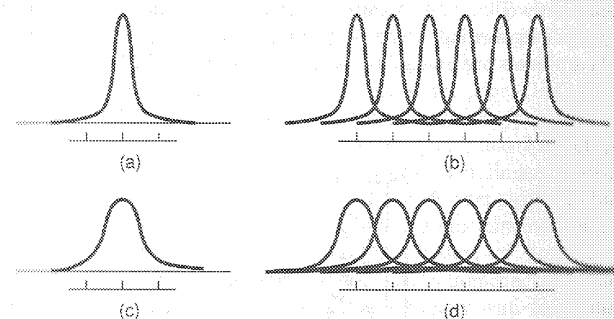


Fig. 4.3 The effect of cross-sectional spot intensity on resolution. (a) A spot with well-defined edges. (b) Several such overlapping spots. (c) A wider spot, with less height, since the energy is spread out over a larger area; its edges are not well defined, as are those in (a). (d) Several of these spots overlapping. The distinction between the peaks in (b) is much clearer in (d). The actual image intensity is the sum of each spot's intensity.

Entry number	Colors loaded in table at each step in animation										
	1	2	3	4	5	6	7	8	9	10	11
0	white	white	white	white	white	white	white	white	white	white	white
1	black	black	black	black	black	black	black	black	black	black	black
2	red	red	red	red	red	red	red	red	red	red	red
3											
4											
5											
6											
7											

Fig. 4.26 Look-up table to bounce a red ball on a black surface against a white background. Unspecified entries are all white.

4.4.2 Bitmap Transformations and Windowing

With some video controllers, the pixmap is decoupled from the view surface; that is, the direct, fixed correspondence between positions in the frame buffer and positions on the display is removed. An *image transformation* then defines the correspondence. The image transformation transforms from the frame buffer to the view surface. The transformation can, in general, include a translation, scaling, rotation, and an additional clipping operation.

Figure 4.27 shows the type of transformation found on some raster displays. Part of the frame buffer, defined by a rectangular clip region, is enlarged to fill the entire view surface. The ratio between the size of the window and the view surface must be an integer (3, in the figure). Pixels in the frame buffer outside the window are not used, and none of the pixel values are modified; the transformations are applied by the video controller at the refresh rate.

The image transformation can be changed many times per second so as to give the real-time-dynamics effect of scrolling over or zooming into an image. Also, a sequence of arbitrary images can be shown rapidly by loading each image into a different area of the frame buffer. The image transformation is changed periodically to display a full-screen view first of one area, then of the next, and so on.

The scaling needed to enlarge an image is trivially accomplished by repeating pixel values from within the window as the image is displayed. For a scale factor of 2, each pixel

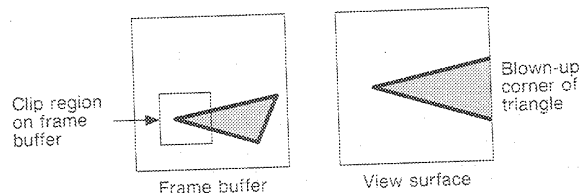


Fig. 4.27 Portion of the frame buffer enlarged on the view surface.

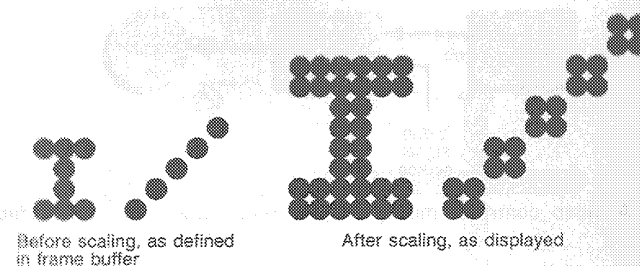


Fig. 4.28 Effect of scaling up a pixmap by a factor of 2.

value is used four times, twice on each of two successive scan lines. Figure 4.28 shows the effect of scaling up a letter and adjoining line by a factor of 2. Unless the image's storage has higher resolution than does its display, scaling up does not reveal more detail; the image is enlarged but has a more jagged appearance. Thus, this animation effect sacrifices spatial resolution but maintains a full range of colors, whereas the double-buffering described in the previous section maintains spatial resolution but decreases the number of colors available in any one image.

In a more general application of image transformations, the scaled image covers only the part of the view surface defined by a viewport, as in Fig. 4.29. Now we must define to the system what is to appear on the view surface outside of the viewport. One possibility is to display some constant color or intensity; another, shown in the figure, is to display the frame buffer itself. The hardware implementation for the latter option is simple. Registers containing the viewport boundary coordinates are compared to the X, Y registers defining the raster scan's current position. If the beam is in the viewport, pixels are fetched from within the window area of the frame buffer and are replicated as required. Otherwise, pixels are fetched from the position in the frame buffer with the same (x, y) coordinates as the beam.

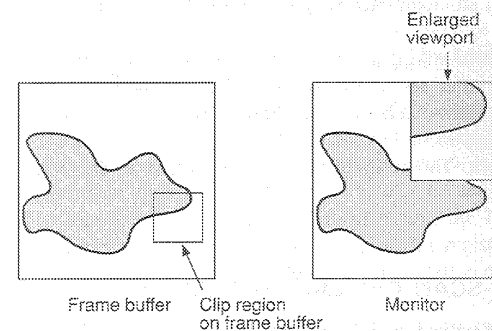


Fig. 4.29 A portion of the frame buffer, specified by the clip region on the frame buffer, scaled up by a factor of 2 and overlaid on the unscaled frame buffer.

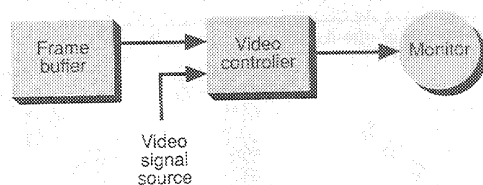


Fig. 4.30 A video controller mixing images from frame buffer and video-signal source.

There are VLSI chips that implement many of these image display functions. These various transformations are special cases of general window-manager operations that might be desired. Window-manager chips are available for use in the video controller [SHIR86]. Each window has a separate pixmap in the system memory, and a data structure in the memory defines window sizes and origins. As a scan line is displayed, the chip knows which window is visible and hence knows from which pixmap to fetch pixels. These and other more advanced hardware issues are discussed in Chapter 18.

4.4.3 Video Mixing

Another useful video-controller function is video mixing. Two images, one defined in the frame buffer and the other defined by a video signal coming from a television camera, recorder, or other source, can be merged to form a composite image. Examples of this merging are seen regularly on television news, sports, and weather shows. Figure 4.30 shows the generic system organization.

There are two types of mixing. In one, a graphics image is set into a video image. The chart or graph displayed over the shoulder of a newscaster is typical of this style. The mixing is accomplished with hardware that treats a designated pixel value in the frame buffer as a flag to indicate that the video signal should be shown instead of the signal from the frame buffer. Normally, the designated pixel value corresponds to the background color of the frame-buffer image, although interesting effects can be achieved by using some other pixel value instead.

The second type of mixing places the video image on top of the frame-buffer image, as when a weather reporter stands in front of a full-screen weather map. The reporter is actually standing in front of a backdrop, whose color (typically blue) is used to control the mixing: Whenever the incoming video is blue, the frame buffer is shown; otherwise, the video image is shown. This technique works well as long as the reporter is not wearing a blue tie or shirt!

4.5 RANDOM-SCAN DISPLAY PROCESSOR

Figure 4.31 shows a typical random (vector) display system. It is generically similar to the display-processor-based raster system architecture discussed in Section 4.3.2. There is of course no pixmap for refreshing the display, and the display processor has no local memory

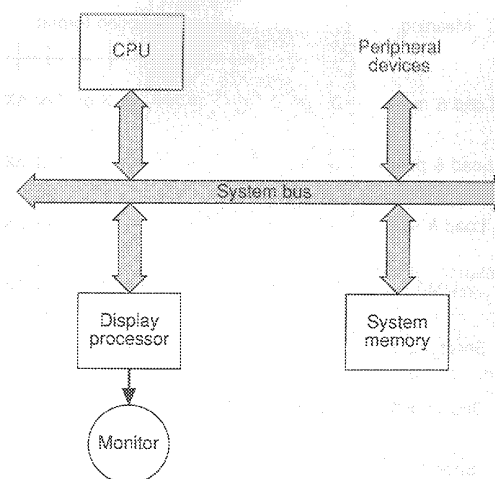


Fig. 4.31 Architecture of a random display system.

for scan-conversion algorithms, since that functionality is typically implemented using programmed logic arrays or microcode.

The random-scan graphics-display processor is often called a *display processing unit* (DPU), or a *graphics controller*. The DPU has an instruction set and instruction address register, and goes through the classic instruction fetch, decode, execute cycle found in any computer. Because there is no pixmap, the display processor must execute its program 30 to 60 times per second in order to provide a flicker-free display. The program executed by the DPU is in main memory, which is shared by the general CPU and the DPU.

The application program and graphics subroutine package also reside in main memory and execute on the general CPU. The graphics package creates a display program of DPU instructions and tells the DPU where to start the program. The DPU then asynchronously executes the display program until it is told to stop by the graphics package. A JUMP instruction at the end of the display program transfers control back to its start, so that the display continues to be refreshed without CPU intervention.

Figure 4.32 shows a set of instructions and mnemonics for a simple random-scan DPU. The processor has X and Y registers and an instruction counter. The instructions are defined for a 16-bit word length. The R/A (relative/absolute) modifier on the LD instructions indicates whether the following coordinate is to be treated as an 11-bit relative coordinate or a 10-bit absolute coordinate. In the former case, the 11 bits are added to either the X or the Y register; in the latter, the 10 bits replace the register contents. (Eleven bits are needed for relative moves for a 2's-complement representation of values in the range -1024 to +1023.)

The same R/A interpretation is used for the JUMP instruction, except the instruction counter is modified, thereby effecting a change in the flow of control. The SM, SP, and SL instructions afford compact representation of contour lines, scatter plots, and the like.

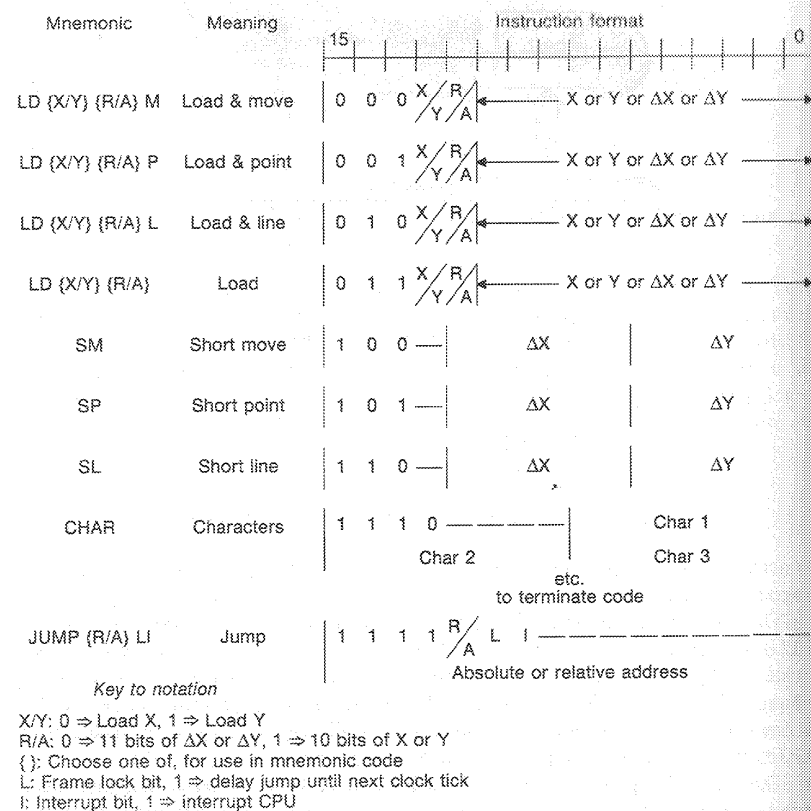


Fig. 4.32 Instruction set for the random-scan display system.

Figure 4.33 is a simple DPU program, written in assembler style, that uses many of these instructions. Notice how the square and diamond are drawn: The first move instruction is absolute, but the rest are relative, so as to facilitate dragging the objects around the screen. If relative instructions were not used, then dragging would require modifying the coordinates of all the instructions used to display the object. The final instruction jumps back to the start of the DPU program. Because the frame lock bit is set, as indicated by the L modifier, execution of the jump is delayed until the next tick of a 30-Hz clock in order to allow the DPU to refresh at 30 Hz but to prevent more frequent refreshing of small programs, which could burn the phosphor.

Notice that, with this instruction set, only one load command (mnemonic LD) is needed to draw or move either horizontally or vertically, because the other coordinate is held constant; oblique movements, however, require two load commands. The two loads can be in either *x-then-y* or *y-then-x* order, with the second of the two always specifying a

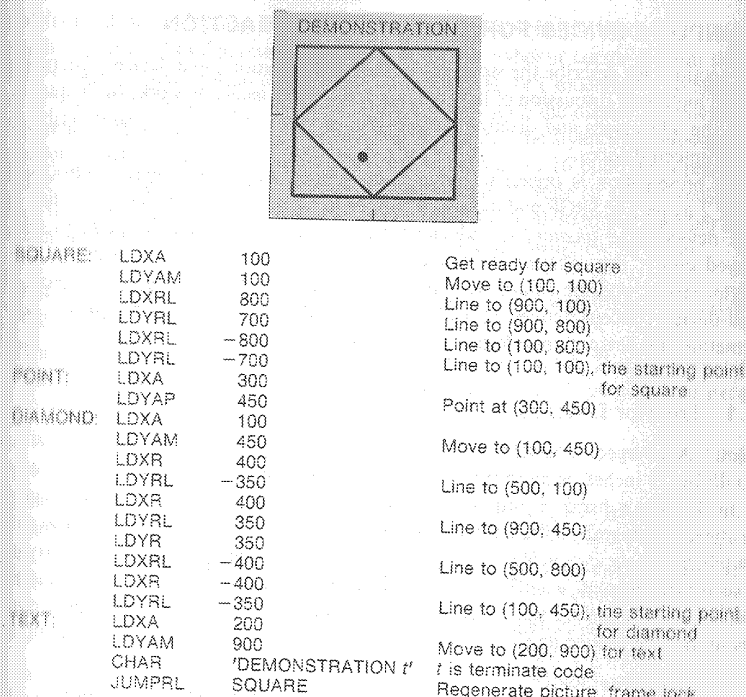


Fig. 4.33 Program for the random-scan display processor.

move, draw-point, or draw-line operation. For the character command (mnemonic CHAR), a character string follows, ended by a nondisplayable termination code.

There are two main differences between these DPU instructions and the instruction set for a general-purpose computer. With the exception of the JUMP command, all the instructions here are rather special-purpose. A register can be loaded and added to, but the result cannot be stored; the register controls only the position of the CRT beam. The second difference, again excluding JUMP, is that all data are immediate; that is, they are part of the instruction. LDXA 100 means "load the data value 100 into the X register," not "load the contents of address 100 into the X register," as in a computer instruction. This restriction is removed in some of the more advanced DPUs described in Chapter 18.

There are only a few differences between the instruction sets for typical random and raster display processors. The random processor lacks area-filling, bit-manipulation, and look-up table commands. But because of the instruction counter, the random processor can have transfer of control commands. Random displays can work at higher resolutions than can raster displays and can draw smooth lines lacking the jagged edges found on raster displays. The fastest random displays can draw about 100,000 short vectors in a refresh cycle, allowing real-time animation of extremely complex shapes.

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

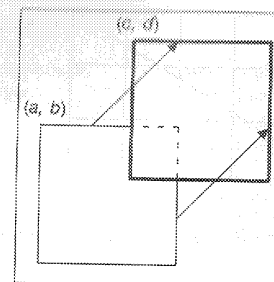


Fig. 17.3 A simple translation of an image.

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 $2/3$ 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

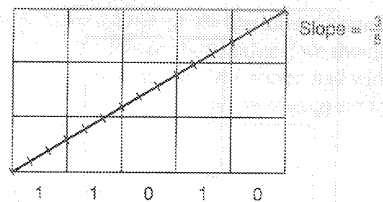


Fig. 17.4 The Rothstein code for a line of slope $\frac{3}{5}$.

with a 1; otherwise, is marked with a 0. Each column contains three tick marks, the one 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 3 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

²A cyclic permutation of a binary sequence is the repeated application of logical shift operations to the sequence.

```

/* Expand the width of an image by p/q. */
void WeimanExpansion(
    const grayscalePixmap source, /* Source image, size n x 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 x k, target image is to be n x ceil(k * p/q). */
    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

⁴The 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.

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.

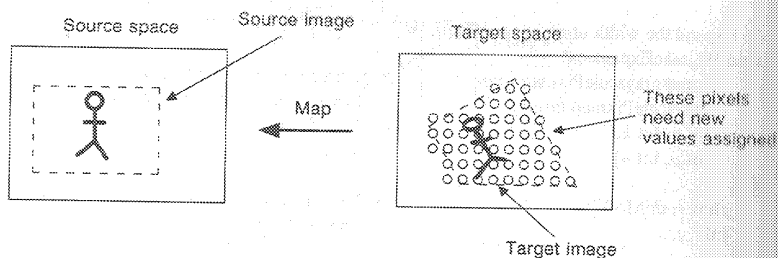


Fig. 17.6 The relationship of the source space, source image, target space, and target image in the Feibush-Levoy-Cook algorithm.

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 map⁵ 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.

⁵A *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.

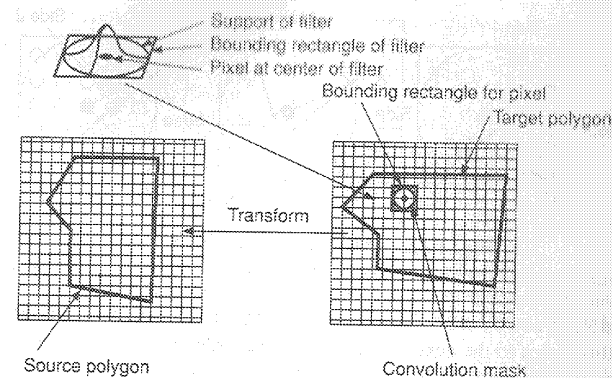


Fig. 17.7 Terms used in the Feibush-Levoy-Cook algorithm.

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

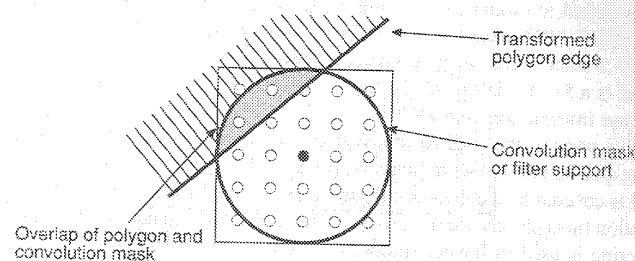


Fig. 17.8 Filtering for a pixel at the edge of the polygon.

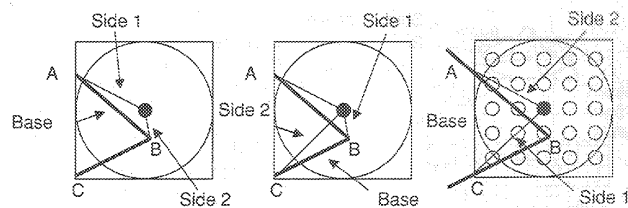


Fig. 17.9 The steps in filtering an edge.

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 $\frac{1}{4}$ 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

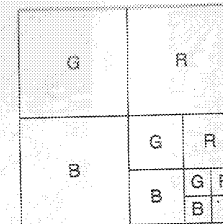


Fig. 17.10 A MIP map. The red, green, and blue channels of the original image fill three quarters of the MIP map. Each is filtered by a factor of 4, and the three resulting images fill up three quarters of the remaining quarter. The process is continued until the MIP map is filled.

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,

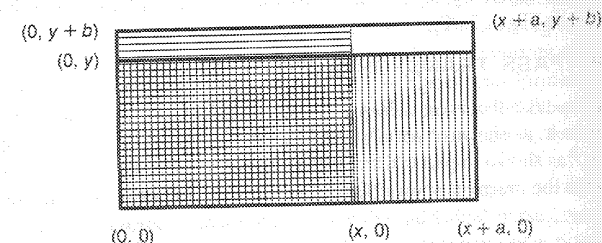


Fig. 17.11 The area of the small white rectangle in the image is computed by subtracting the horizontally and vertically shaded areas from the area of the large rectangle, and then adding back in the area of the crosshatched rectangle.

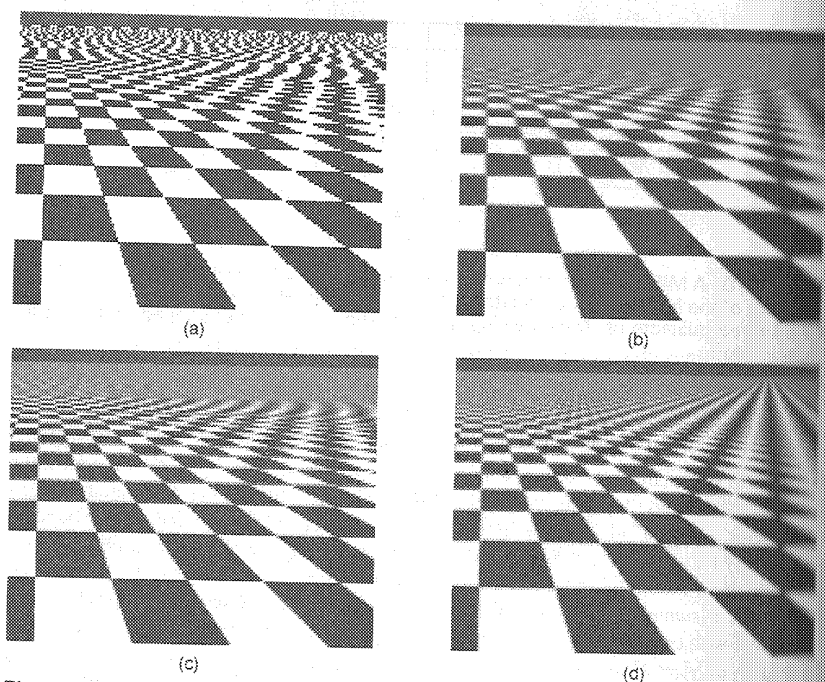


Fig. 17.12 (a) Point sampling of the source image. (b) MIP map filtering. (c) Summed area table filtering. (d) Elliptical weighted average using MIP maps and a Gaussian filter. (Courtesy of P. Heckbert.)

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-Cock 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

⁶The 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.

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

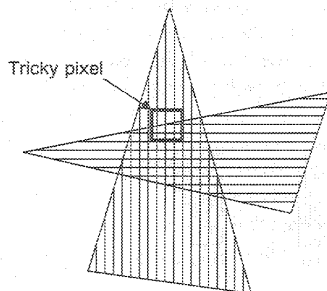


Fig. 17.15 Compositing operations near an edge: How do we color the pixel?

LG-HISENSE00050068

Ex. 1018 - TCL Indus Holdings Co., Ltd.

L Indus Holdings Co., Ltd. v. LGE – IPR2023-00461

Page 19 of 36

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 to 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

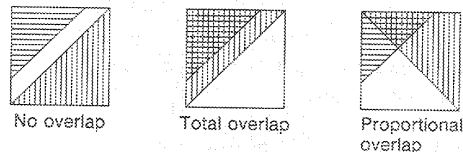


Fig. 17.16 The ways in which polygons can overlap within a pixel. In image composition, the first two cases are considered exceptional; the third is treated as the rule.

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

$$0.6 \left(\frac{1}{3}\right)(1,0,0) + 0.4 \left(\frac{1}{2}\right)(0, 0, 1) = (0.2, 0, 0.2)$$

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 RGB α 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).

⁹In 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.

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 $F_A \alpha_A + F_B \alpha_B$.

A few unary operations can be performed on images. For example, the *darken* operation is defined by

$$\text{darken}(A, \rho) := (\rho R_A, \rho G_A, \rho B_A, \alpha_A) \quad 0 \leq \rho \leq 1,$$

which effectively darkens a pixel while maintaining the same coverage. In contrast, the *fade* operator acts by

$$\text{fade}(A, \delta) := (\delta R_A, \delta G_A, \delta B_A, \delta \alpha_A) \quad 0 \leq \delta \leq 1$$

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

$$\text{opaque}(A, \omega) := (R_A, G_A, B_A, \omega \alpha_A).$$

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

$$\text{fade}(A, t) \text{ plus } \text{fade}(B, 1 - t),$$

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.

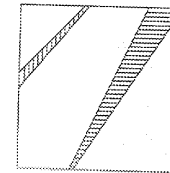


Fig. 17.18 A case where the overlap assumptions fail in compositing.

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.

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).

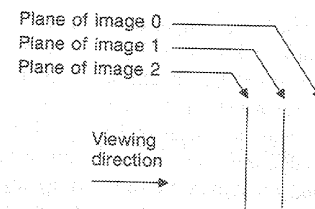


Fig. 17.21 Relation of three images to the viewing direction.

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 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

$$\text{newImage} * t + \text{oldImage} * (1 - t),$$

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),

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

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 used 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

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

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 quadrees.

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 in 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

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 principle¹⁰ 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 $\{w_1, \dots, w_r\}$ 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

$$w \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}; \quad (17.1)$$

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} , $i = 1, \dots, n$; $j = 1, \dots, k$. 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 $w_1, \dots, w_r$ 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 :

$$A = \bigcup_{i=1}^r w_i(A).$$

¹⁰The 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.

```

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  $0 \leq r \leq 1$  */
        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  $V_{i,j}$  */
                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



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_i changes the brightness of the portion of the image coming from w_i . 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.

```

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): 8888
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 in 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. If we 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

19.3.4.) Compositing such an image using an w 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 (quadrees, 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

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

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

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

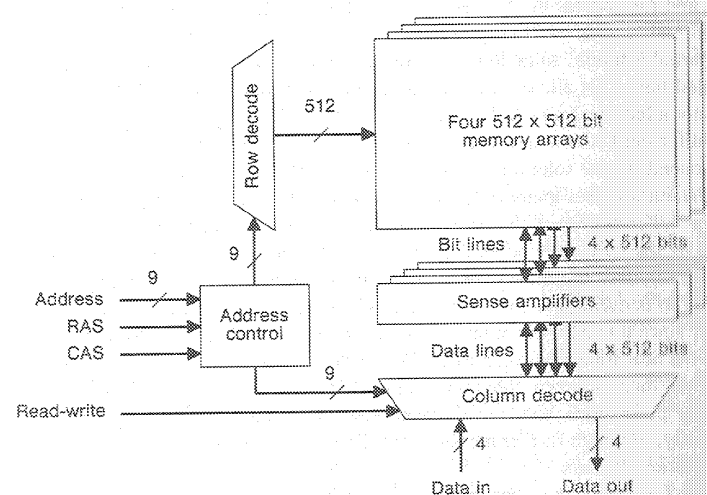


Fig. 18.1 A 1-Mbit ($256K \times 4$) DRAM chip. An entire row is written to or read from each of the four memory arrays at the same time. The column decoder allows a particular element (column) of the selected row to be accessed.

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

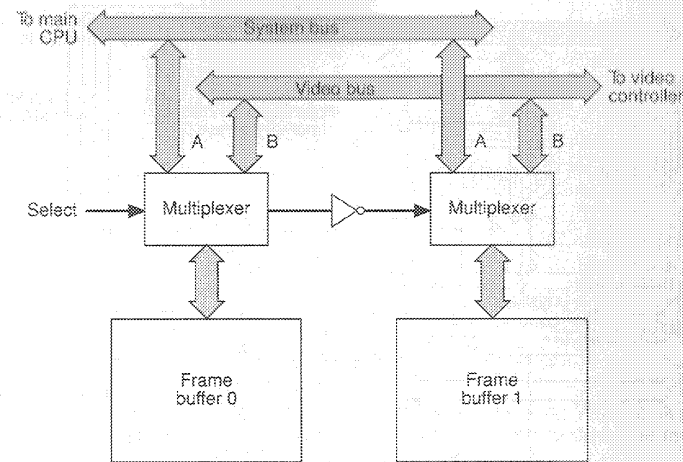


Fig. 18.2 Double-buffered frame buffer.

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.

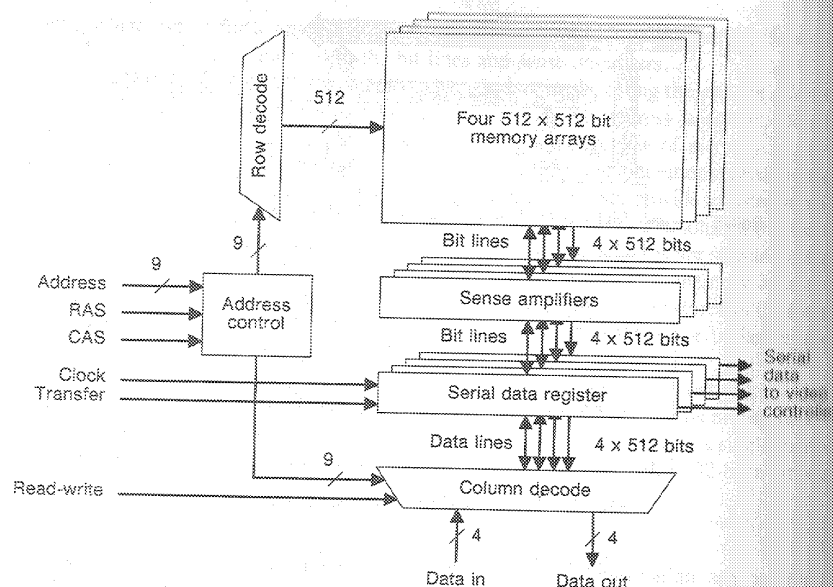


Fig. 18.3 Diagram of a 1-Mbit ($256K \times 4$) VRAM chip. The serial data register provides a second (serial) port to the memory array.

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,

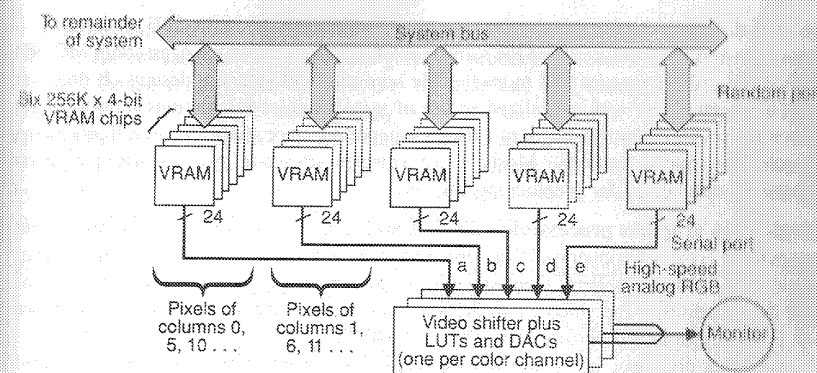


Fig. 18.4 Block diagram of a 1280-by-1024-pixel display using VRAMs. A parallel-in/serial-out shift register provides the high pixel rates required to drive the monitor.

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.

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 Processor¹

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.

¹Material for this example was contributed by Jerry R. Van Aken of Texas Instruments, Inc.

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 blit 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.

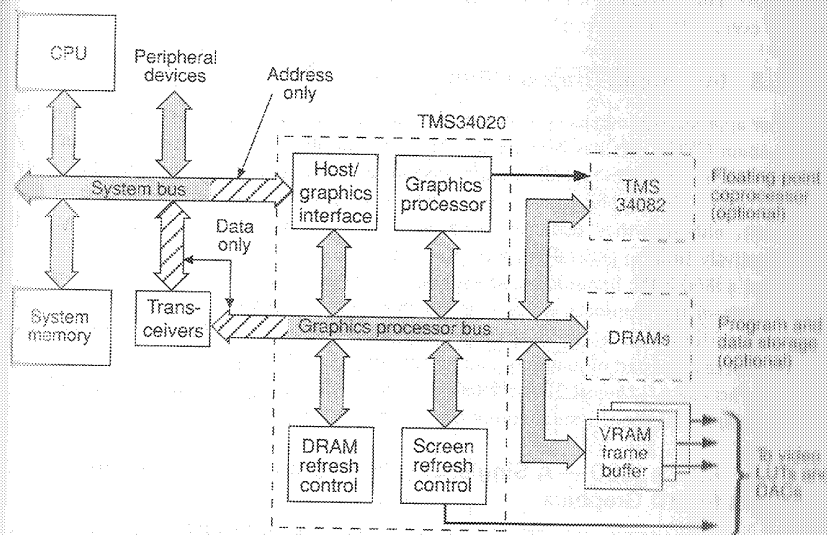


Fig. 18.5 Block diagram of the TMS34020 graphics-processor chip used as a peripheral display processor.

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 VLSI 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

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 [INTES9].

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

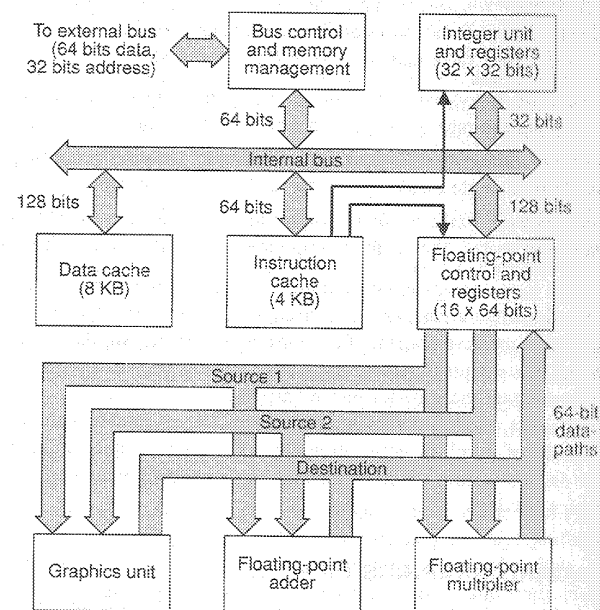


Fig. 18.6 Block diagram of Intel i860 microprocessor datapaths (adapted from [GRIM89]).

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.

LG-HISENSE00050083

Ex. 1018 - TCL Indus Holdings Co., Ltd.

TCL Indus Holdings Co., Ltd. v. LGE – IPR2023-00461

Page 34 of 36

example, AT&T's Pixel Machine Model PXM 924 ray traces simple scenes at interactive speeds—a feat unmatched by any SIMD system.

Since interleaved-memory architectures distribute the frame buffer over multiple processors, some provision must be made for scanning out pixels in the uninterrupted stream required by the video controller and display monitor. If the distributed frame buffer is constructed of VRAMs, this can be done in the manner shown in Fig. 18.18. Note that this is very similar to the technique described in Section 18.1.5 for implementing high-speed video scanout in a conventional frame-buffer memory.

Further subdivision. Suppose we wish to build a system with even higher performance. Since increasing the number of partitions in an interleaved-memory system increases the system's memory bandwidth and processing power, we might consider increasing the number of frame-buffer partitions—say, from 16 to 64, or even more. Unfortunately, the additional processors and datapaths required for each partition make such systems increasingly expensive.

An even more serious difficulty is supporting a larger number of partitions with the same number of frame-buffer memory chips. Each partition requires a minimum number of chips. For example, a partition with a 32-bit datapath between the PE and memory requires 8 4-bit wide chips, or 32 1-bit wide chips. Suppose we wish to build a 16-partition 1024-by-1024-pixel frame buffer with 128 bits per pixel. Using $256K \times 4$ VRAM chips, each partition requires $8 \times 256K \times 4$ VRAM chips, so $16 \cdot 8 = 128$ chips are needed to support all 16 memory partitions. This is the exact number required to store the pixel data.

Suppose, however, that we increase the number of partitions from 16 to 64 (an 8×8 footprint). Although we still need only 128 memory chips to store the pixel data, we need $64 \cdot 8 = 512$ memory chips to support the PE-memory bandwidth. The extra 384 memory chips are needed only to provide communication bandwidth—not for memory. This is an extra expense that continues to grow as we subdivide the frame buffer further.

Increasing the density of memory parts from 1 Mbit to 4 Mbit exacerbates this problem even further. For example, if $1Mbit \times 4$ VRAM memory chips are used in the example mentioned above, 512 chips are still needed, even though each one contains sixteen times the memory actually required. Current systems such as the Silicon Graphics' POWER IRIS GTX (described in the next section), which uses 20 frame-buffer partitions, are already at the bandwidth limit. A way to ameliorate this problem would be for memory manufacturers to provide more data pins on high-density memory parts. Some 4-Mbit DRAM chips have eight data pins, rather than four, which helps somewhat, but only reduces the bandwidth problem by a factor of 2.

18.8.2 Silicon Graphics' POWER IRIS 4D/240GTX—An Interleaved Frame-Buffer Memory Architecture^{2,3}

Silicon Graphics' POWER IRIS 4D/240GTX [AKEL88; AKEL89] uses many of the techniques described in this chapter. Like a number of its competitors, including the Ardent

²Material for this example is adapted from [AKEL88] and [AKEL89].

³In 1990, Silicon Graphics announced POWERVISION (similar to the GTX) that renders 1 million Gouraud-shaded triangles/sec and with 268 bits/pixel for antialiasing and texturing.

Titan [BORD89], the Megatek Sigma 70 [MEGA89], and the Stellar GS2000 (Section 18.11.3), the SGI POWER IRIS is a high-end graphics workstation, designed to combine general-purpose processing and high-speed 3D graphics for engineering and scientific applications.

The POWER IRIS has a powerful general-purpose CPU composed of four tightly coupled multiprocessors sharing a single memory bus. Its graphics subsystem can render over 100,000 full-colored, Gouraud-shaded, z-buffered quadrilaterals per second [AKEL89]. The POWER IRIS continues Silicon Graphics' tradition of immediate mode display traversal, aggressive use of custom VLSI, hardware front-end pipeline, and interleaved-memory frame-buffer architecture. The POWER IRIS's architecture, diagrammed in Fig. 18.19, is composed of five major subsystems:

1. *CPU subsystem*—runs the application and traverses the display model
2. *Geometry subsystem*—transforms and clips graphical data to screen coordinates
3. *Scan-conversion subsystem*—breaks points, lines, and polygons into pixels
4. *Raster subsystem*—computes visibility and writes pixel data to frame buffer
5. *Display subsystem*—displays contents of frame buffer on color monitor.

CPU subsystem. The *CPU subsystem* runs the application and traverses the database. It is composed of four tightly coupled, symmetric, shared-memory multiprocessors. Hardware provides high-speed synchronization between processors, so parallelism can be achieved within a single process (although special programming constructs are required).

Geometry subsystem. The *geometry subsystem* transforms, clips, and lights primitives. It is composed of five floating-point processors arranged in a pipeline. Each of these processors, called a *geometry engine* (GE), contains an input FIFO, a controller, and a floating-point unit capable of 20 MFLOPS. Unlike Silicon Graphics' earlier Geometry Engine (see Section 18.5.4), the POWER IRIS' GEs are based on a commercial floating-point chip, the Weitek 3332.

The first GE transforms vertices and vertex normals. The second GE performs lighting calculations (supporting up to eight point light sources). The third GE performs trivial accept/reject clipping tests. The fourth GE performs exact clipping on primitives that cross clipping boundaries, and also does perspective division for all primitives. The fifth GE clips color components to maximum representable values, calculates depth-cued colors where necessary, and converts all coordinates to screen-space integers.

Scan-conversion subsystem. The *scan-conversion subsystem* rasterizes primitives using the pipeline approach described in Section 18.7.1, except that its spans are vertical columns of pixels, rather than the horizontal rows we have assumed so far (the only effect on the rasterization algorithm is that x and y coordinates are interchanged).

The single *polygon processor* sorts the vertices of each polygon from left to right in screen space. The sorted vertices are then used to decompose the polygon into vertically aligned trapezoids. The upper pair of vertices and the bottom pair of vertices of each trapezoid are used to calculate slopes for use by the edge processors.

The *edge processor* uses vertex and slope information to compute x , y , and z coordinates and color values for each pixel that lies on the top or bottom edges of each

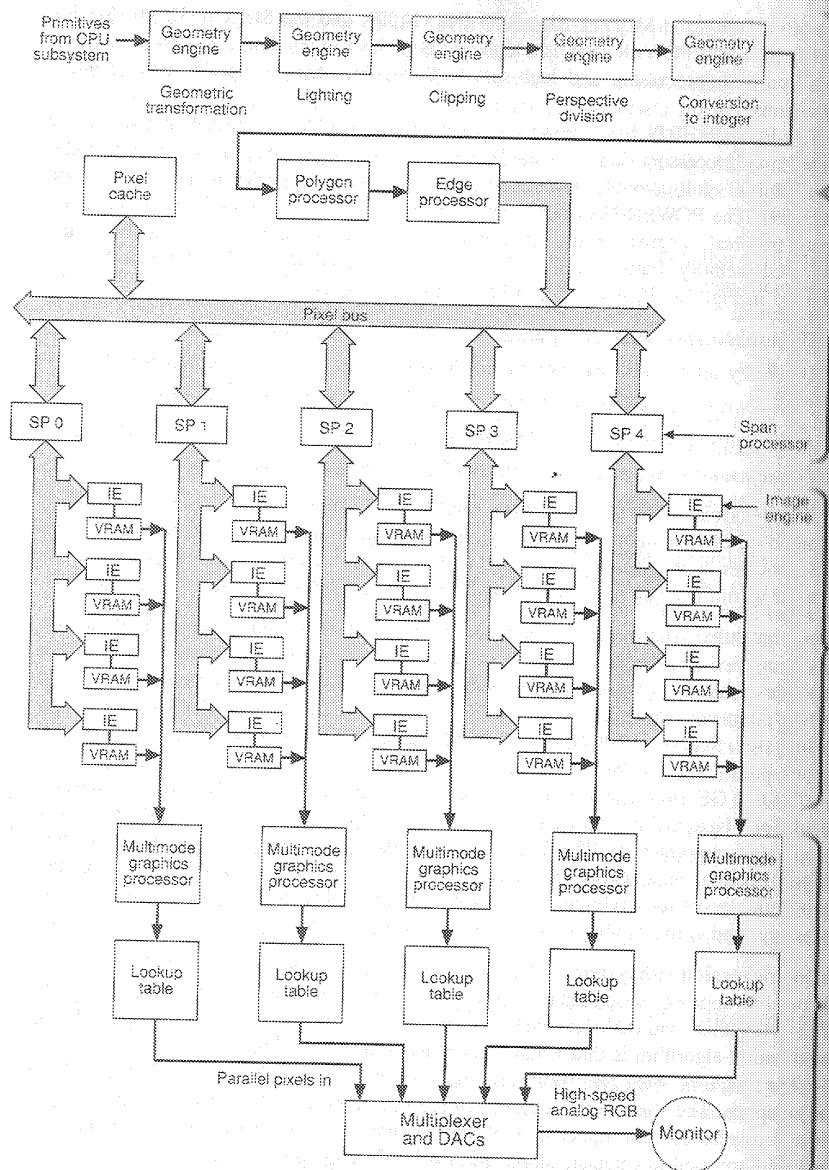


Fig. 18.19 Silicon Graphics' GTX system architecture (based on [AKEL88]).

trapezoid. In a given vertical column, a pair of pixels determines the top and bottom endpoints of a vertical span. These pixel pairs, together with slope information, are passed on to the span processors.

Each of the five parallel *span processors* is responsible for one-fifth of the columns of the display screen. For example, span processor 0 manages scan lines 0, 5, 10, and so on. Span processors calculate z , R , G , B , and α (for transparency and antialiasing) values for each pixel in the span. Because spans generated from a single polygon are adjacent, the processing load over span processors is approximately uniform.

The *pixel cache* buffers blocks of pixels during pixel copy operations so that the full bandwidth of the pixel bus can be used.

Raster subsystem. The *raster subsystem* takes pixel data generated by the span processors and selectively updates the image and z bitplanes of the frame buffer, using the results of a z comparison and α blending. The raster subsystem is composed of 20 *image engines*, each responsible for one-twentieth of the screen's pixels, arranged in a 4×5 -pixel interleaved fashion. 96 bits are associated with each pixel on the screen: two 32-bit image buffers (r , G , B , and α), a 24-bit z -buffer, four overlay/underlay bitplanes, and four window bitplanes.

The overlay/underlay bitplanes support applications that use pop-up menus or windowing backgrounds. The window bitplanes define the display mode (single- or double-buffered, etc.) and window-masking information.

The 20 image engines work on pixels in parallel. Each can blend values based on the pixel's α value, allowing transparent or semitransparent objects to be displayed, and allowing supersampling antialiasing.

Display subsystem. The display subsystem contains five *multimode graphics processors* (MGPs), each assigned one-fifth of the columns in the display. The MGPs concurrently read image data from the frame buffer (together with window-display-mode bits), process them using the appropriate display mode (RGB or pseudocolor), and send them on to digital-to-analog converters for display.

The GTX's architecture is a good example of many of the techniques we have discussed so far. It provides high performance for polygon rendering at a reasonable cost in hardware. Because the GTX's rendering pipeline is highly specialized for graphics tasks, however, the system has difficulty with the advanced rendering techniques we shall discuss in Section 18.11, and its resources cannot be applied easily to nongraphics tasks. Section 18.11.3 discusses the architecture of Stellar's GS2000, which has complementary advantages and disadvantages.

18.8.3 Logic-Enhanced Memory

Since commercial memories may not support enough frame-buffer partitions, one might consider building custom memories with a large number of concurrently accessible partitions on a single chip. Since each (intrachip) partition must have its own connection to its associated (external) processor, extra pins must be added to each memory package to support these additional I/O requirements. Alternatively, multiple processors could be built