A Language for Shading and Lighting Calculations

Pat Hanrahan* and Jim Lawson†

*Princeton University

†Pixar

Abstract

A shading language provides a means to extend the shading and lighting formulae used by a rendering system. This paper discusses the design of a new shading language based on previous work of Cook and Perlin. This language has various types of shaders for light sources and surface reflectances, point and color data types, control flow constructs that support the casting of outgoing and the integration of incident light, a clearly specified interface to the rendering system using global state variables, and a host of useful built-in functions. The design issues and their impact on the implementation are also discussed.

CR Categories: I.3.3 [Computer Graphics] Picture/Image Generation- Display algorithms; I.3.5 [Computer Graphics] Three-Dimensional Graphics and Realism - Color, shading, shadowing and texture.

Additional Keywords and Phrases: Shading language, little language, illumination, lighting, rendering

1. Introduction

The appearance of objects in computer generated imagery, whether they be realistic or artistic looking, depends both on their shape and shading. The shape of an object arises from the geometry of its surfaces and their position with respect to the camera. The shade or color of an object depends on its illumination environment and its optical properties. In this paper the term *shading* refers to the combination of light, shade (as in shadows), texture and color that determine the appearance of an object. Many remarkable pictures can be created with objects having a simple shape and complex shading. A well-designed, modular rendering program provides clean interfaces between the geometric processing, which involves transformation, hidden surface removal, etc., and the optical processing, which involves the propagating and filtering of light. This paper describes a language for programming shading computations, and hence, extending the types of materials and light sources available to a rendering system. In Bentley's terminology it would be called a "little" language[3], since, because it is based on a simple subset of C, it easy to parse and implement, but, because it has many high-level features that customize it for shading and lighting calculations, it is easy to use.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Two major aspects of shading are the specification of surface reflectance and light source distribution functions. The earliest surface reflectance models have terms for ambient, diffuse and specular reflection. More recent research has added anisotropic scattering terms[14, 16, 21] and made explicit wavelength and polarization effects. Although not nearly as well publicized, many improvements have also been made in light source description. The earliest light source models consisted of distant or point light sources. Verbeck and Greenberg[29] introduced a general framework for describing light sources which includes attaching them to geometric primitives and specifying their intensity distribution as a function of direction and wavelength.

Light sources and surface reflectance functions are inherently *local processes*. However, many lighting effects arise because light rays traveling from light to surface are blocked by intervening surfaces or because light arriving at a surface comes indirectly via another surface. Turner Whitted termed these effects *global illumination processes*. Kajiya introduced the general light transport equation, which he aptly termed the *rendering equation*, and showed how all these techniques are tied together[15]. Most recent research in shading and lighting calculations is now being focussed on making these global illumination algorithms efficient. Note that global and local illumination processes are independent aspects of the general illumination process.

In parallel to the development of specific shading models is the development of shading systems. Most current systems implement a single parameterized shading model. There are several problems with this approach. First, there is little agreement on what this shading model should be. Almost every rendering system written has used a slightly different shading model. Second, it seems unlikely that a single parameterized model could ever be sufficient. As mentioned earlier, the development of shading models is an active area of research and new material models are continually being developed. Shading also involves many *tricks*, one major example being texture mapping, and the use of rendering tricks is completely open ended. Furthermore, the surface reflectance models of simple and composite materials are phenomenologically based and not derivable from first principles. Shading models that capture the effects of applying varnish or lacquer to wood, or of adding an additional flap to a stage light, are much better expressed procedurally than as mathematical formulae. Another problem with this single parameterized model approach, is that simple shading formula carry the overhead of the most complicated case. This overhead makes it more difficult for users to control, and more time consuming for the rendering

ATI Ex. 2002
IPR2023-00687
Page 1 of 10

system to compute.

Because of these difficulties with the single shading model approach, several systems have been described that provide greater flexibility and extensibility. Whitted proposed that the rendering system have a collection of built-in shaders accessible via a shader dispatch table[31]. Presumably, the interface to these shaders was well-defined so that experienced hackers with access to the source code could extend the system. Cook developed a model which separated the conceptually independent tasks of light source specification, surface reflectance, and atmospheric effects[6]. The user could control each of these shading processes independently by giving a sequence of expressions which were read in, parsed, and executed at run-time by the rendering system. Perlin's image synthesizer carried this idea further by providing a full language including conditional and looping constructs, function and procedure definitions, and a full set of arithmetic and logical operators[24]. But Perlin abandoned the distinction between the shading processes proposed by Cook, and instead introduced a "pixel stream" model. In the pixel stream model, shading is a postprocess which occurs after visible surface calculations. Unfortunately, this makes his language hard to use within the context of a radiosity or ray-tracing program, where much of the shading calculation is independent of surface visibility.

In this paper, a new language is described which incorporates features of both Cook's and Perlin's systems. The goals in the design of the new language were to:

- Develop an abstract shading model based on ray optics suitable for both global and local illumination models. It should also be abstract in the sense of being independent of a specific algorithm or implementation in either hardware or software.
- Define the interface between the rendering program and the shading modules. All the information that might logically be available to a built-in shading module should be made available to the user of the shading language.
- Provide a high-level language which is easy to use. It should have features – point and color types and operators, integration statements, built-in functions – that allow shading calculations to be expressed naturally and succinctly.

A detailed description of the shading language grammar is available in the RenderMan interface specification[1], and many examples of its use are contained in Upstill[28]. The intent of this paper is to point out the features of the new language beyond those described by Perlin and Cook. The design of the new language also raised many subtle design and implementation issues whose resolution required a combination of graphics and systems perspectives. The alternatives that were considered, and the factors that influenced the choices that were made are discussed. Finally, we discuss some of the more interesting parts of the implementation, particularly those aspects where a combination of techniques drawn from graphics, systems and compiler theory were used to improve performance.

2. Model of the Shading Process

Kajiya has pointed out that the rendering process can be modeled as a integral equation representing the transport of light through the environment[15].

$$i(x, x') = v(x, x')\,[l(x, x') + \int r(x, x', x'')\, i(x', x'')\, dx'']$$

The solution, $i(x, x')$, is the intensity of light at x which comes from x'. The integral computes the amount of light reflected from a surface at x' as a function of the surface bidirectional reflectance function $r(x, x', x'')$ and the incoming light intensity distribution $i(x', x'')$. The term $l(x, x')$ gives the amount of light emitted by light sources at x' in the direction towards x. The sum of these two terms is the amount of light initially traveling from x' to x, but not all that light makes it to x: some of it may be scattered or blocked by an intervening material. The term $v(x, x')$ gives the percentage of light that makes it from x' to x.

The shading language allows procedures, called shaders, to be written that compute the various terms in the above equation. Shaders implement the local processes involved in shading; all the global processes used in solving the rendering equation are controlled by the renderer. The three major types of shaders are:

- *Light Source Shaders.* A light source shader calculates the term $l(x, x')$, the color and intensity of light emitted from a particular point on a light source in a particular direction.
- *Surface Reflectance Shaders.* A surface reflectance shader calculates the integral of the bidirectional reflectance function $r(x, x', x'')$ with the incoming light distribution $i(x', x'')$.
- *Volume or Atmosphere Shaders.* Volume shaders compute the term $v(x, x')$. Only scattering effects need be computed; the effects of light rays intersecting other surfaces are handled by the renderer.

A surface shader shades an infinitesimal surface element given the incoming light distribution and all the local properties of the surface element. A surface shader assumes nothing about how this incoming light distribution was calculated, or whether the incoming light came directly from light sources or indirectly via other surfaces. A surface shader can be bound to any geometric primitive. The rendering program is responsible for evaluating the geometry, and provides enough information to characterize the infinitesimal surface element. Similarly, when a light source shader computes the emitted light, it makes no assumptions about what surfaces that light may fall on, or whether the light will be blocked before it reaches the surface. A light source shader also makes no assumptions about whether it is bound to a geometric primitive to form an area light.

Kajiya has shown how standard rendering techniques can be viewed as approximate solutions of the rendering equation. The simplest approximation assumes that light is scattered only once. A ray is emitted from a light source, reflected by a surface, and modulated on its way towards the eye. This is often referred to as local shading, in contrast to global shading, because no information about other objects is used when shading an object. This shading model is what is used by most real-time graphics hardware and is easily accommodated by the shading language. Whitted's ray tracing algorithm[30] considers these direct transport paths, plus light transported from surfaces intersected by reflected and refracted rays. This can also be accommodated by the shading language by recursively calling light source and surface shaders.

To summarize, the abstraction used by the shading language provides a way of specifying all the local interactions of light, without assuming how the rendering program solves the light transport equation. Thus, shaders written in the shading language can be used by many different types of rendering algorithms.

3. Language Features

The shading language is modeled after C[17], much like many other programming languages developed under UNIX. The specification of the syntax and grammar is available in the specification[1] and examples of its use are described in a recent book[28]. In the following sections the novel features of the shading language are discussed. The emphasis is on the high-level design issues that influenced each feature, and the implementation problems that they posed. The features discussed include the semantics of the color and point data types, the meta types

ATI Ex. 2002
IPR2023-00687
Page 2 of 10

uniform and varying, the classes and subclasses of shaders and how shaders are attached to geometry, the intrinsic state information which is provided by the rendering system, the special control constructs for integrating incoming light in surface shaders and for casting outgoing light in light source shaders, some of the more unusual built-in functions, and support for texture mapping.

3.1. Types

The shading language supports a very small set of fixed data types: floats, strings, colors, and points. Since points and colors are the fundamental objects passed between shaders and the renderer, these are supported at a high level in the shading language. No facilities exists to define new types or data structures.

3.1.1. Colors

The physical basis of color is a spectrum of light. The spectrum describes the amount of light energy as a continuous function of wavelength. Since calculations involving continuous spectra are generally not feasible, spectra are represented with a fixed number of samples. Different methods for sampling spectra are described in Hall[12, 13]. In the shading language, color is an abstract data type which represents a sampled spectra. The number of color samples per color can be set before rendering to control the precision of the color computations. One sample implies a monochrome color space; three samples a triaxial color space; and more samples are available for more precise color computations. There is no support for sampling or resampling spectra; this is assumed to be done by the modeling program driving the renderer or the output program generating the final display.

Within the spectral color space model there are two important operations involved in shading calculations: *additive light combination*, where the result is the spectrum formed by combining multiple sources of light, and *filtering*, where the result is the spectrum produced after light interacts with some material. These operations are mapped into the language by overloading the standard addition and multiplication operators ("+" and "*"), respectively. All color computations in the language are expressed with these operators, so that they are independent of the actual number of samples. A typical color computation might be expressed as

```
Cd * (La + Ld) + Cs * Ls + Ct * Lt
```

where Cd and Cs are the diffuse and specular colors of a material, La, Ld, and Ls, are the amount of ambient, diffuse, and specular light reflected from the surface, and Ct and Lt are the transparency and amount of light transmitted through the material. Note that transparency is treated just like any other color, that is, it has the same number of color components. Many rendering systems make the mistake of modeling transparency with a single number. This is presumably motivated by the use of RGBA color models[26] where α which was originally developed to represent *coverage* is treated as an opacity (equal to one minus the transparency).

We considered having two types of color, one for light spectra and another for material absorption spectra, and to restrict the values of light spectra samples to always be positive, since they represent energy which must be positive, and the values of material spectra to always be between 0 and 1, since they represent percent absorption. However, this was thought to be too restrictive - for example, negative light sources are sometimes used for faking shadows, and reflective colors greater than 1 are used for modeling stimulated emission. For similar reasons, we also allowed other arithmetic operators between colors, although they are very seldomly used. In our experience, it is very convenient to think of color as light and hence, not to clamp it to some maximum value.

Eventually, after all shading computations have been performed, the light "exposes" film using a non-linear remapping from light intensities to output pixel colors. After this process, a pixel color value of 1 is treated as the maximum display intensity.

Since the addition and multiplication of colors is performed on a component by component basis, the shading language makes no assumptions about what physical color each sample actually represents. Also, if only color operators are used to combine colors inside a shader, an arbitrary linear transformation can be applied to the input or output colors without affecting the results. This gives the modeling program driving the renderer complete control over what spectral color space the renderer is computing in. One advantage of this is that color computations can performed in absolute or calibrated color spaces, just by transforming the input or output color space.

There are many other color spaces used in computer graphics for defining colors. We refer to these as *modeling color spaces* to distinguish them from spectral *rendering color spaces*. In general, adding and multiplying colors in these other color spaces has no physical basis, and hence executing shaders with colors in non-spectral color spaces can lead to unpredictable results. The language supports the use of modeling color spaces by providing built-in functions which immediately convert constant colors defined in these color spaces to the current rendering color space.

3.1.2. Points.

The type point is used to represent a three component vector. The arithmetic operators are overloaded so that when they involve points they are treated in the standard vector algebra sense. New operators were added to implement the operations of dot product (".") and cross product ("^"). Using this syntax, a Lambertion shading formula can be expressed on one line.

```
C * max( 0, L.N )
```

where L and N are the light and normal vectors, and C is a color. The main advantage of having built-in vector operations is that the standard shading formulae can be expressed in a succinct and natural way, often just by copying them right from the literature.

Another advantage of expressing shading calculations using vector arithmetic is that they are then expressed in a coordinate-free way, which means that the shader could be evaluated in different coordinate systems. Care must be taken in applying this idea since, in general, transformations between coordinate systems do not preserve metric properties such as angles and distances. Since the physics underlying shading calculations are based on these metric quantities, the results of shading calculations will be different in coordinate systems which do not preserve them. For this reason shading calculations are defined "to appear" as if they took place in the world coordinate system, but it is permissible for the renderer to shade in other coordinate systems that are isometric to the world coordinate system. A common example of this is shading in "camera" or "eye" space instead of world space.

In certain situations, it is necessary for a calculation involving a point to be performed in a specific coordinate system. For example, all surface shaders accessing a solid texture need to access the texture in the solid texture's coordinate system. This is supported by providing a procedure which transforms points between named coordinate systems. The standard named coordinate systems are "raster", "screen", "camera", "world" and "object". In our system it is also possible to mark other coordinate systems, and then to refer to them within shaders.

ATI Ex. 2002
IPR2023-00687
Page 3 of 10

3.2. Uniform and Varying Variables

All variables in the shading language fall into one of two general classes: uniform and varying. uniform variables are those whose values are independent of position, and hence, constant once all the properties have been bound; varying variables are those whose values are allowed to change as a function of position.

Varying variables come about in two ways. First, geometric properties of the surface usually change across the surface. Two examples are surface parameters and the position of a point on a surface. The normal changes on a curved surface such as a bicubic patch, but remains constant if the surface is a planar polygon. Second, variables attached to polygon vertices or to corners of a parametric surface are automatically interpolated across the surface by the rendering program, and hence are varying. The best examples of varying variables attached to polygons are vertex colors in Gouraud shading and vertex normals in Phong shading. The concept of interpolating arbitrary variables during scan conversion was first introduced by Whitted and Weimer[31].

The concept of uniform and varying variables allows the shading language compiler to make significant optimizations. If the shader contains an expression or subexpression involving only constants and uniform variables, then that expression need only be evaluated once, and not every time the shader is executed. This is similar to constant folding at compile time, but differs in that a different uniform subexpression may occur each time a new instance of a shader is created, or each time a shader is bound to a surface. Because shading calculations are so expensive, a folklore has developed over hand coding these types of optimizations. For example, if the viewing transformation is a parallel projection, meaning the eye is at infinity, and a planar polygon is being shaded, the incoming direction, the normal, and hence the direction of the reflection vector are all constant and need only be computed once per polygon. A similar situation occurs with local and distant lights. The advantage of using uniform variables and having the compiler look for uniform expressions is that these optimizations are done automatically.

3.3. Shader Classes and Instances

It is often convenient to think of shaders in an object-oriented way. There are several major subclasses of shaders, corresponding to the set of methods required by the rendering system. The most general class of shading procedures is a shader, and there are subclasses for light sources, surface reflectance functions, and volume scattering. A shader for a specific subclass is created by prefixing its definition by a keyword: `surface` for a surface shader, `light` for a light shader, and `volume` for a volume shader†. Surface shaders describe different types of material such as metal and plastic; and light source shaders different classes of lights such as spotlights and bulbs.

Shader definitions are similar to procedure definitions in that they contain a formal argument list. The arguments to a shader, however, are very different than the arguments to a shading language function. Calling a shader to perform its task is under the control of the rendering system, and all information from the renderer is passed to the shader through external variables and not via its arguments (see Section 3.4). Shaders are never called from other shaders or from other functions in the shading language.

† Actually the following types also exist: `displacement`, `transformation`, and `imager`.

The shader arguments define the shader's *instance* variables and are used to set the properties of a shader when the user adds the shader to the graphics state or attaches it to a geometric primitive. All instance variables have default values that must be specified as part of the definition of the shader. However, the defaults can easily be overridden when creating an instance. This is done by giving a list of name-value pairs; the name refers to which instance variable is being set, and the value to its new value. This method of defaulting makes it easy to use complicated shaders with many parameters. For example, the shader `metal` is declared in the shading language as

```
surface
metal( float Ka=1, Ks=1, roughness=.1 )
```

and is attached to the surface with the following procedure call.

```
RiSurface( "metal", "Ka", 0.5 );
```

This instance of `"metal"` has a different `Ka` than the default instance.

The arguments, or instance variables, of a shader are typically uniform variables because they describe overall properties of the surface or light source. User-defined interpolated variables are created in the shading language by declaring an argument to a shader to be a varying variable. The interpolated value is made available to the shader via that shading language variable. Finally, it is possible to pass point variables as instance variables to a shader. As a convenience, these points are interpreted to be in the current coordinate system, and transformed automatically to the coordinate system in which the shading calculation is being performed. For example, `pointlight` has as part of its declaration.

```
light
pointlight( ...;
    point from = point "shader" (0,0,0);
            ... )
```

The "shader" coordinate system is the one in effect when the shader was instanced. In the above example the light is placed at the origin of this coordinate system. Note how the transformations apply to default points as well as points supplied when instancing.

3.4. Intrinsic State

When a shader is bound to a geometric primitive, it inherits a set of varying variables that describe the geometry of the surface of the geometric primitive. All shaders also inherit the color and opacity of the primitive and the local illumination environment described as a set of light rays. This information is made available to a shader as variables which are bound to the correct values by the rendering system before the shader is evaluated. A shader is very much like a function *closure*, which contains pointers to the appropriate variables based on the scoping rules for that function[27]. The names and types of the major external variables are shown in Figures 1 and 2.

These external variables, along with a few built-in functions, specify exactly what information is passed between the rendering system and the shading system. Because this is the only way these modules communicate, determining these variables was one of the most difficult aspects of the design of the shading language. Two general principles were followed: (i) the material information should be minimal, but extensible, and (ii) the geometric and optical information should be complete. A simpler interface between the shading and geometry is specified in Fleischer and Witkin[10].

ATI Ex. 2002
IPR2023-00687
Page 4 of 10

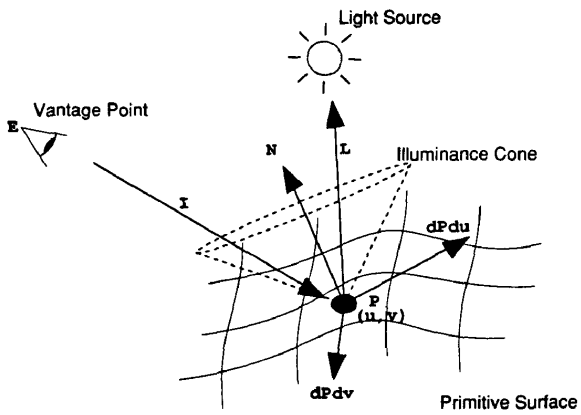


Figure 1. Surface shader state.

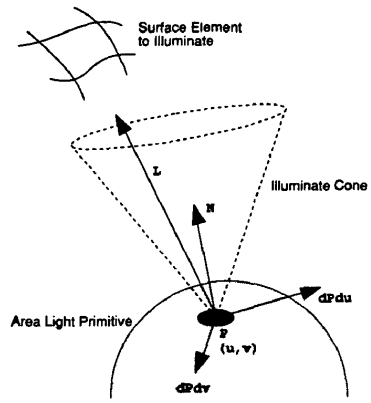


Figure 2. Light source shader state.

Since one of the major goals of the shading language is to extend the types of materials used by the rendering system, it is important to be able to assign arbitrary properties to new materials. The only material properties assumed to always be present, and hence made available as global variables, are color and opacity. All other material properties are explicitly declared as arguments to the shader. Since there is no restriction, in principle, to the number or types of arguments to a shader, the properties of materials can involve any amount of information.

The rendering system may perform shading calculations at many points on the surface of a geometric primitive. It provides enough geometric information to characterize the surface element in the neighborhood of the point being shaded. Most shading formulae involve only the position `P` and normal `N` of the surface. When doing texture mapping it is often necessary to provide the surface parameters. More advanced shading methods, such as bump mapping[5] or tangent bundle mapping[14] require the parametric derivatives of the position vector. From a mathematical point of view, to completely characterize a surface at a point requires knowledge of all its parametric derivatives and cross derivatives at the point. Other intrinsic surface properties, such as Gaussian curvature, can be computed from this information. There are CAD and mathematical applications which require methods to visualize local properties of the surface geometry[9]. Providing all these derivatives of position through global variables would be unwieldy, so functions were provided to take derivatives of position with respect to the surface parameters. This derivative function is discussed in more detail below (see Section 3.7).

It is expensive for the rendering system to compute all this information, and this is wasted computation if it is not being used by the shader. Unnecessary calculation can be prevented by having the shaders provide a bitmask indicating which external variables are referenced within the shader before it is executed. Alternatively, the runtime environment uses a lazy evaluation to compute the values of the variables on demand. It is also useful to provide a bitmask indicating which variables are changed by the shader, since in some cases the renderer may want to retain the original values.

3.5. Light Sources

The most general light source description defines the intensity and color of the emitted light as a function of position and direction[29]. The shading language provides a method for describing an arbitrary light source distribution procedurally. It does this by providing two constructs, `solar` and `illuminate`. The illuminate statement is used to set the color of light coming from a finite point. Its arguments define a cone with its apex at the point from which the light emanates. Only points inside that cone receive light from that light source. The solar statement is used to set the color of light coming from distant or infinite light sources. Its arguments define a cone, centered on the point being illuminated, to which distant light will be cast. Within the block of a illuminate or solar statement, the emitted light direction is available as an independent read-only variable `L`, and the color of the emitted light `Cl` is treated as a dependent variable which should be set to define the color and intensity of light emitted as a function of direction.

During the design process the following canonical types of lights were considered, and they illustrate the types of light sources which can be modeled.

- *Ambient light source.* Ambient light is non-directional and ambient light shaders do not use either an illuminate or a solar statement. Note that ambient light can still vary as a function of position.
- *Point light source.* A point light source casts equal amounts of light in all directions from a given point. This is the simplest example of the use of an illuminate statement.
- *Spot light source.* This is point light source whose intensity is maximum along the direction the light is pointed and falls off in other directions. A spotlight also has a circular flap which limits angle of the beam. This is an example of a procedurally defined point light source.
- *Shadowed light source.* This is a point light source whose intensity is modulated by a texture or shadow map.
- *Distant light source.* An infinite light casts light in only one direction. This is the simplest example of the use of a solar statement.
- *Illumination or environment map.* This is a omnidirectional distant light source whose intensity is given by a texture map.

ATI Ex. 2002
IPR2023-00687
Page 5 of 10

- *Phong light source.* This is a procedurally defined distant light source.

Code for the light shaders for each of these types of sources is contained in the specification[1].

The light distribution from an area source is determined conceptually by evaluating a light shader at different points on the geometric primitive defining the shape of the source. The results are summed, in effect, convolving the light source distribution function with the shape of the source. Since all area light sources are finite, the light source shader attached to a primitive must contain an illuminate statement. Area light source shaders also have access to all the surface properties, just like surface shaders, so it is possible to define lights whose color, intensity and even directional dependence varies across the surface.

3.6. Surface Reflectance

A surface reflectance shader integrates all the incoming light with a procedurally defined bidirectional reflection function to compute the color and intensity of light reflected in a particular direction. This integration is controlled using an `illuminance` statment. The arguments to the illuminance statement define a cone, usually the upper hemisphere centered on the surface element, over which the integration occurs. Inside of the block defined by an illuminance statement, two independent read-only variables are defined: the incoming color and intensity of light (`Cl`) and the light ray direction (`L`). It is up to the programmer to accumulate the results of the reflectance computations. There is no restriction on the number of illuminance statements within a shader, although they cannot be nested. There are built-in functions that compute the ambient, diffuse, and specular reflection functions, because these are so commonly used.

As an example of the use of the illuminance statement, the diffuse component of the anisotropic shading model for fur proposed by Kay and Kajiya[16] would be programmed as

```
color C = 0;
illuminance( P, N, Pi/2 ) {
    L = normalize(L);
    C += Kd * Cd * Cl * length(L ^ T);
}
```

where `T` is the direction tangent to the fur. The length of the cross product of `L` and `T` is proportional to the sin of the angle between them, assuming they are unit vectors. The arguments to the illuminance statement define the upper hemisphere centered at the point being shaded; light coming from other directions is not considered when performing the integral. The variable `C` is used to accumulate the results of the integral.

3.7. Built-In Functions

A lot of attention was paid to selecting the built-in functions in the language. Functions were added so that common shading operations were readily available and could be easily composed. Frequently used functions were also built-in, so that they could be implemented in the most efficient way possible. In a few cases, functions were built-in because their calculations could not be expressed within the shading language; however, this was always considered bad and whenever possible features were added to the language to make it more complete. Expressing all the functions using the language makes it possible for users to modify them to suit their needs.

For maximum flexibility many of the built-in functions are *polymorphic*, that is, they can accept arguments with different types and perform the appropriate operations depending on the type of input. The most straightforward example is `printf` which can print any type in the language. More generally, polymorphic functions may return a type which also depends on the type of inputs or their values. Polymorphism significantly complicates the compiler and the run-time system, so to simplify, the return type of a polymorphic function can only be a function of the types of the inputs. These typing rules are built into the compiler, and so it is not possible for the user to write polymorphic functions. In some cases, the return type depends on the value of an argument and not its type (for example texture access, where the return type depends on the texture map being used which is identified by passing in a texture map name). This case is handled by requiring an explicit type cast before these types of functions.

The remainder of this section will describe several of the more unusual functions and their impact on the implementation.

Taking the derivative of position was discussed in Section 3.4. For generality, another function was added to take the derivative of any varying variable with respect to any other varying variable. These functions turned out to be very useful, but much more difficult to implement than we expected. First, the derivative of a procedurally defined variable is not always continuous. For example, when a value is computed within a conditional statement, neighboring values computed in different branches of the *if* may be quite different. The logical expression controlling the *if* statement acts as a step function between the values computed in the branches of the *if*, so the derivative becomes infinite at the step. Fortunately, these situations tend to be avoided in practice because such shading formula are very prone to aliasing and cause artifacts. Second, to form derivatives of an arbitrary expression requires information about how it varies in a neighborhood of the point where the derivative is taken. This can be done reliably, if the variable is only a function of the properties of the surface, for example, rate of change of color as a function of texture coordinates. However, if the variable is a function of the illumination environment, then the values of its neighbors may be contingent on knowing the illumination environment of its neighbors, which may be difficult for certain types of renderers to provide.

One difficulty with procedural shading models is that they are prone to aliasing. Shading functions will alias if they contain frequencies greater than the rate at which the surface is being shaded. Providing the sampling rate to the shader allows shading functions to *clamp* themselves so that no frequencies greater than the Nyquist frequency are present[22]. The best example of this is the use of band-limitied noise functions for solid texture synthesis[18, 25]. The sampling rate is provided by a function which returns the screen area covered by a single shading calculation, and also by two global variables, `du` and `dv`, which estimate the change in the surface parameters between adjacent samples. If the area is 1, the surface element being shaded occupies approximately one pixel in the final image. A very interesting area for future work is to have the shading language compiler try to automatically band-limit the shading functions so that clamping need not be done explicitly.

3.8. Texture Mapping Functions

The texture mapping functions return filtered texture values based on user-supplied indices. There are four built-in texture functions:

- `texture` returns floats or colors as a function of the surface's texture coordinates.
- `bump` returns a point specifying the normal perturbation as a function of the surface's texture coordinates.
- `environment` returns floats or colors as a function of a direction in space passed as a point.
- `shadow` returns a float specifying the percentage of the surface element in shadow as a function of its position.

There is no limit, in principle, to the number of texture maps allowed per shader or per scene. The texture indices may be the

ATI Ex. 2002
IPR2023-00687
Page 6 of 10

surface parameters or texture coordinates, or they may be computed with the shading language. This flexibility in defining the mapping to texture space allows techniques such as decals[2], reflection maps[11, 20], and two-part texture mapping[4] to be programmed in the shading language.

4. Implementation

A shader passes through various stages in its life cycle. We give an initial definition of the various stages in the shader life cycle, then examine each step in more detail.

- *Compilation.* Compilation occurs prior to rendering. Files containing shading language source statements are processed by a compiler which generates code for a particular run-time interpreter. The compiler is responsible for the inline expansion of user functions and performs several implementation-independent optimizations.
- *Loading.* During the specification of the scene to the renderer, shaders are placed into the current graphics state or instanced. The first time a shader is instanced, the renderer must read in the code and symbol table.
- *Instancing.* The shader's instance variables must be bound to the shader, replacing the default values. The "shader" coordinate system is also defined when the shader is instanced, and any point parameters passed to the shader must be transformed from this coordinate system to the coordinate system being used for calculation.
- *Binding to a Geometric Primitive.* Next the shader is bound to a particular geometric primitive. This occurs whenever a geometric primitive is created, and since different primitives may share a shader this occurs more frequently than instancing. At this point, the "object" coordinate system has been defined and can be bound. The surface color and opacity are inherited from the current graphics state or from the geometric primitive. Since these attributes can come from either source, the uniform or varying nature of late-binding variables is only now determined.
- *Elaboration.* Surface parameters are bound to values in the graphics state, and data that may be located in various caches is organized for efficient execution.
- *Evaluation* The shader is evaluated at different points on the surface.

Each subsequent stage in the life-cycle is usually performed many more times than the previous stages. This life cycle strains the implementation since it presents a general binding problem. For flexibility, it is desirable to bind as many values as late as possible, but for efficiency, it is desirable to optimize later stages in the life cycle. Since these optimization steps are themselves time-consuming, it is desirable for them to occur as soon as possible in the life cycle so that they are not repeated. However, the optimizers do the best job once all the bindings have been established. Table 1 gives time and invocation counts for these various life cycle stages for a typical image (a single frame from the animated short "Luxo Jr.").

Stage	Invocations	Time (seconds)	Percentage
Compile	1	5.6	1.33
Load	12	2.11	.50
Instance	45	3.61	.86
Bind	153	3.96	.94
Elaborate	9004	0.78	.18
Evaluate	11255	403.08	96.16

Table 1: Shader Life Cycle

We now consider these steps in more detail.

4.1. Compilation

The compiler uses the shader class for determining allowed access to the intrinsic state. Only light shaders may set the light color and contain solar and illuminate statements. Only surface shaders may contain illuminance statements and only within these statements is access to the light color and direction allowed. These constraints are intended to limit the environment for each class of shader for performance reasons, and to detect user errors at compile time.

Typically the time spent compiling a shader is many orders of magnitude less than the time spent actually executing a shader. This in turn implies that no compile-time optimizations are prohibitively expensive. Two of the more unusual optimizations performed by the compiler involve expression classification and compiled function inlining.

Shading language expressions can be categorized as falling into one of two classes: uniform or varying. In principle, uniform expressions need only be evaluated once. The compiler has facilities for rearranging uniform expressions in order to execute them separately from the varying expressions in the shader body. There are implementation difficulties with this: the compiler has to be careful about code motion across loop and conditional boundaries, and the late binding of shader parameters restrict the type inferencing that can be done. In these cases, the compiler assumes the expression is varying but provides sufficient information for the run-time system to make the final decision (at a somewhat reduced level of efficiency). It should be noted in passing that the uniform/varying distinction poses a classical time/space tradeoff. If the cost of saving the result of a computation is in some sense cheaper than the computation, it makes sense to save the result. There may be implementations where the storage requirements for these values are far more expensive (in terms of their size or management) than simply performing the computation each time its result is required.

The compiler is also responsible for the inline insertion of previously compiled shading language functions. The compiler reconstructs a parse tree from the compiled function and performs the normal optimizations after inserting the parse tree inline. If the function can be evaluated at compile time (that is, if its arguments are known constants), its result will be substituted for the function call.

In general, we attempt to do as much optimization at compile-time as is possible, and for those cases where we must defer to the run-time system, we attempt to provide it with sufficient information for it to further optimize execution. To this end, the "object" file output by the compiler contains an extensive symbol table. In fact, there is sufficient information in the symbol table to reconstruct the original parse tree. For those shader parameters with default values, the symbol table contains a pointer to the block of code to compute this value. For each symbol referenced by the shader, the symbol contains pointers to the initial and final read and write references in the shader code. This is used to determine if a shader requires access to a external variable, or if the shader changes the variable, to ensure that a local copy is made or that related properties (position and normals for instance) are updated consistently.

4.2. Loading

The run-time system loads shaders in response to *shader directives* which normally occur when a shader is made current. The shader name is mapped into an (operating system dependent) external file name for the initial request to read the file containing the shader. In order to minimize the amount of I/O traffic the system maintains a simple *shader cache*. The (system independent)

ATI Ex. 2002
IPR2023-00687
Page 7 of 10

shader name is used as the key for subsequent searches. The first time a shader is read, it is converted from an external to an internal executable format. This converted shader object is subsequently stored in the shader cache. Future references to this shader will fetch the copy from the cache thus avoiding the overhead of file I/O and the external to internal conversion process.

4.3. Instancing

Associated with a shader directive is a parameter list. It is quite common for modeling systems to generate complex objects composed of alternating shaders containing the same parameters. If each shader directive results in the creation of a separate instance of the referenced shader, we would end up filling memory with many copies of identical shader instances. This problem is solved with the addition of a higher level shader *instance cache*. The shader's parameters are packaged up and inserted in the shader instance cache along with a reference to the basic entry in the lower level shader cache. The shader name and its parameters are used as a key to index the instance cache. Thus, shader directives that refer to the same shader with the same parameters will all share a common instance of that shader. These caches may be flushed at the discretion of the renderer, usually at end of frame.

4.4. Binding to a Geometric Primitive

It is not until we actually bind the shader to a geometric primitive that we can finally determine the uniform/varying nature of its parameters. But the values of these variables are not yet available because they may be computed as part of the rendering process. Consequently, we package up the number and identity of varying parameters along with a reference to the shader instance, and insert this item into a *bound instance cache*.

4.5. Elaboration

Eventually a geometric primitive has been reduced to an object for which the rendering system wishes to invoke a shader. Any varying parameters have been interpolated over the surface of the primitive. At this time, all parameters have been bound to the shader and we are in a position to determine their values. We allocate storage for the shader's local and temporary variables and finally evaluate its parameters. The parameters of the shader are assigned (in order of decreasing precedence) either:

- the value associated with the geometric primitive,
- the value specified in a shader directive, or
- the default value as specified in the shader.

We refer to this as an *elaborated* shader instance and update the bound instance cache entry for the shader to indicate an elaborated instance is available. Subsequent attempts to evaluate the shader will use this elaborated instance.

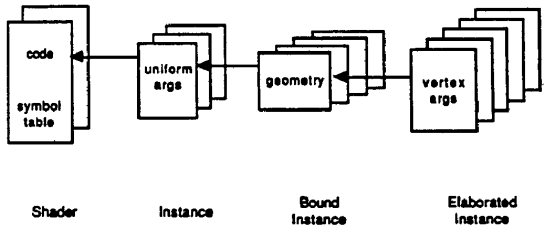


Figure 3. Hierarchy of shader caches.

Figure 3 illustrates the various internal caches and Table 2 demonstrates the effectiveness of the hierarchical cache scheme for the same Luxo Jr. frame described in Table 1.

Cache	Storage	Probes	Misses	Hit Rate
Shader	29.2K	14	12	14%
Instance	1.5K	45	14	68%
Bound Instance	na	na	na	na
Elaborated Instance	21.0K	9004	42	99%

Table 2: Shader Cache Hit Rates

The instance cache size is basically determined by the number of different types of surfaces and lights, the elaborated instance cache by the number of geometric primitives. The relatively low hit rates for the lower level caches are due to the effectiveness of the higher level caches. We have only counted actual probes of each cache. If one counts actual post-probe references (11255) to cached data, the importance of the lower level caches is more apparent. There are no vertex arguments in this example, so the bound instance cache is bypassed.

4.6. Evaluation

The current run-time system implements a virtual SIMD array processing architecture. Geometric primitives are diced into *surface elements* or *grids* containing some number of surface points[8]. One geometric primitive may give rise to many grids. The shading interpreter executes shader operators once per grid, each operator performing its calculations over every point in the grid. This helps reduce the effect of interpreter overhead by amortizing it over the number of points in the grid.

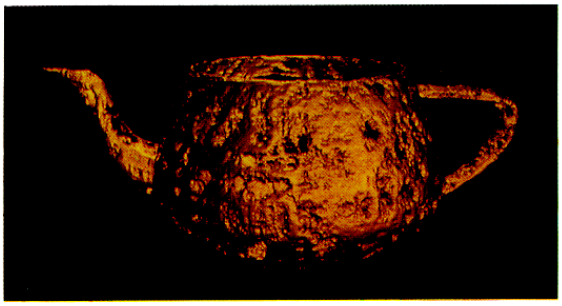
Earlier we mentioned that due to the late binding of shader parameters, certain optimizations must be left to the run-time system. Each operator is passed the address, type, and uniform/varying nature of its arguments. The individual operators may use this information to optimize their computations. For example, if an operator receives uniform arguments and is required to compute a varying result, it may perform its operation once, then replicate the result value. Thus, at the very worst, each uniform expression is computed at most once per grid.

The SIMD nature of the run-time system does complicate the handling of loops and conditionals, especially if normal SISD semantics are to be preserved. Nevertheless, we felt it was important to hide the details of the implementation, and so resisted adding language constructs to deal with the lower level SIMD machine.

5. Discussion

Overall the shading language has met most of the goals set out in the introduction. Figure 4 shows one example of the type of flexibility possible in the language. This surface shader dents the teapot using a fractal-like procedural displacement until the metal breaks. This breakage is modeled by making the surface transparent if the magnitude of the dent exceeds a certain value. Note that the shader shown in Figure 4 is quite short, which attests to the high-level nature of the language. Most shaders that have been written take much less than a screenful of text. The frame from the film *knickknack* shown in Figure 5 was created using procedural shading for everything except the texture mapped text on the pyramid and pool, and the drawing on the surfboard. The shading language encourages the development of compact procedural texture representations. In the future, we expect to see a great deal of research in procedural material representations of appearance. Catalogs of materials will be available much like catalogs of clip art are available today.

ATI Ex. 2002
IPR2023-00687
Page 8 of 10



```
surface
dent( float Ks=.4, Kd=.5, Ka=.1, roughness=.25, dent=.4 )
{
    float turbulence;
    point Nf, V;
    float i, freq;

    /* Transform to solid texture coordinate system */
    V = transform("shader",P);

    /* Sum 6 "octaves" of noise to form turbulence */
    turbulence = 0; freq = 1.0;
    for( i=0; i<6; i+= 1 ) {
          turbulence += 1/freq * abs( 0.5 - noise( 4*freq*V ) );
          freq *= 2;
    }

    /* Sharpen turbulence */
    turbulence *= turbulence * turbulence;
    turbulence *= dent;

    /* Displace surface and compute normal */
    P -= turbulence * normalize(N);
    Nf = faceforward( normalize(calculatenormal(P)), I );
    V = normalize(-I);

    /* Perform shading calculation */
    Oi = 1 - smoothstep( 0.03, 0.05, turbulence );
    Ci = Oi * Cs * (Ka*ambient() + Ks*specular(Nf,V,roughness));
}
```

Figure 4. Corroded teapot.

A crucial question is how efficient is this approach relative to built-in shading models. Compiler efficiency is not really an issue here, as the time spent compiling shaders is very small compared to the time spent evaluating them. In fact, the evaluating/compiling ratio is so large that we are quite willing to spend a considerable amount of time trying to perform optimizations during the compile stage in exchange for reduced rendering time. Also, shading language programs are quite short. This is an excellent opportunity to employ normally costly "aggressive" optimization strategies such as superoptimization which searches for the best possible generated code[19]. The compiler could also convert functions or complicated expressions of a single varying variable to tables and evaluate them using table lookup. For example, the power function used for specular shading is a function of varying quantity `N.H` and a uniform exponent. Once the exponent is available, the compiler could generate a table for this function. The uniform/varying variable distinction allows one to find constant uniform expressions and save the cost of reevaluating them, which can reduce shading time considerably. In the example above, if the normal and highlight direction are fixed, then the power function need only be evaluated once. The point is that many of the techniques currently used to speed up shading algorithms could be implemented by better shading compilers. If such compilers are successful, then compiled shaders could be more efficient than hand-coded shaders.

The language presented does pose some difficult and challenging problems for graphics algorithm developers. The specification of area and procedural light sources was designed for full generality, but as a result is beyond the capability of current lighting and shading algorithms, since they assume the renderer can correctly integrate over directional or positional distributions. However, it seems likely that distributed stochastic sampling[7] would yield good approximations to these integrals, although this is certainly an area that needs more research. Also, as mentioned in the introduction, global illumination models such as radiosity work only with simple diffuse shading formula. Implementing a global illumination algorithm for the procedural shaders describable in this language is also an area for future research. Finally, the flexibility introduced by having programmable texture functions taxes the texturing subsystem. This is an area which needs further research; some preliminary results are reported in Peachey[23].

However, this approach to shading does have its limitations. Perhaps the crucial assumption is that it is a good idea to decouple geometric and optical calculations. Kajiya has proposed a hierarchy of detail which involves smooth transitions between geometry, displacement mapping, bump mapping, and surface reflectance functions – that is, a smooth transition from geometric to optical calculations[14]. In such a system, it might make sense to more tightly couple the two types of calculations.

Finally, the shading language is the interface which programmers use to extend the types of materials and light sources available to a rendering system. A better interface for most people, however, is an interactive editor that allows them to build and modify existing shaders. Previous work along these lines allows the user to control parameters of preprogrammed shaders; the challenge here is to find methods that allow for quick updates so the system is feels responsive. This is a difficult problem because fast methods involve approximations that affect the final appearance, so *what you see is not what you get*. Compiler technology might help an interactive parameter editor by figuring out how to evaluate shading formulae incrementally if only a single parameter is changing. Another interesting area for future research is to create an interactive shading editor that models the optical properties of materials by simulating the processes that are used to create their appearance. Each object description would begin by cutting out the geometry from an underlying solid material (for example, wood or steel), which would then be covered by a surface material (for example, veneer or decals) and a coating (for example, paint and varnish). The result could then be polished and subject to wear and tear. This type of editor would create shading language programs by patching together together program fragments that model the various stages used to produce the final appearance.

6. Acknowledgements

Rob Cook designed and implemented the original shade-tree system. Tony Apodaca provided constant input on many features of the language and in particular helped think through the light source specification. Sam Leffler and Darwyn Peachey implemented the texturing subsystem. Mike Paquette and Malcolm Blanchard helped port the system to various platforms. Mark Leather and Jeff Mock tried successfully to break pieces of the system with enormous shaders; Jeff also was kind enough to contribute the shader for Figure 4. Bill Reeves, Eben Ostby, and John Lasseter suffered through the entire development process, finding and fixing bugs, and tried to make a movie at the same time; they were also kind enough to contribute Figure 5. Finally, we would like to thank Ed Catmull, Tom Porter and Mickey Mantle for their help with this project.

ATI Ex. 2002
IPR2023-00687
Page 9 of 10

Figure 5. Procedural texture in *Knickknack*.

References

1. *The RenderMan Interface*, PIXAR (December 1989).
2. BARR, ALAN H., "Decal Projections," *ACM SIGGRAPH '84 Course Notes 15: Mathematics of Computer Graphics*, (1984).
3. BENTLEY, JON, "Little Languages," pp. 83-100 in *More Programming Pearls*, Addison-Wesley, Reading, Massachusetts (1988).
4. BIER, ERIC A. AND KENNETH R. SLOAN JR., "Two-Part Texture Mapping," *IEEE Computer Graphics and Applications* **6**(9) pp. 40-53 (September 1986).
5. BLINN, JAMES F., "Simulation of Wrinkled Surfaces," *Computer Graphics* **12**(3) pp. 286-292 (August 1978).
6. COOK, ROBERT L., "Shade Trees," *Computer Graphics* **18**(3) pp. 223-231 (July 1984).
7. COOK, ROBERT L., "Stochastic Sampling in Computer Graphics," *ACM Transactions on Graphics* **5**(1) pp. 51-72 (January 1986).
8. COOK, ROBERT L., LOREN CARPENTER, AND EDWIN CATMULL, "The Reyes Image Rendering Architecture," *Computer Graphics* **21**(4) pp. 95-102 (July 1987).
9. DILL, JOHN C., "An Application of Color Graphics to the Display of Surface Curvature," *Computer Graphics* **15**(3) pp. 153-161 (August 1981).
10. FLEISCHER, KURT AND ANDREW WITKIN, "A Modeling Testbed," *Graphics Interface '88*, pp. 127-137 (June 1988).
11. GREENE, NED, "Environment Mapping and Other Applications of World Projections," *IEEE Computer Graphics and Applications* **6**(11) pp. 108-114 (November 1986).
12. HALL, ROY A., "Color Reproduction and Illumination Models," pp. 194-238 in *Techniques for Computer Graphics*, ed. R. A. Earnshaw,Springer-Verlag (1987).
13. HALL, ROY A., *Illumination and Color in Computer Generated Imagery*, Springer-Verlag, New York (1989).
14. KAJIYA, JAMES T., "Anisotropic Reflection Models," *Computer Graphics* **19**(3) pp. 15-22 (July 1985).
15. KAJIYA, JAMES T., "The Rendering Equation," *Computer Graphics* **20**(4) pp. 143-149 (August 1986).
16. KAJIYA, JAMES T. AND TIMOTHY L. KAY, "Rendering Fur with Three Dimensional Textures," *Computer Graphics* **23**(3) pp. 271-280 (July 1989).
17. KERNIGHAN, BRIAN W. AND DENNIS M. RITCHIE, *The C Programming Language*, Prentice-Hall (1978).
18. LEWIS, JOHN P., "Algorithms for Solid Noise Synthesis," *Computer Graphics* **23**(3) pp. 263-270 (July 1989).
19. MASSALIN, HENRY, "Superoptimizer: A Look at the Smallest Program," *Proceedings of ASPLOS (Architectural Support for Programming Languages and Operating Systems)*, pp. 122-127 (October 1987).
20. MILLER, GENE S. AND C. ROBERT HOFFMAN, "Illumination and Reflection Maps: Simulated Objects in Simulated and Real Environments," in *Siggraph '84 Course Notes: Advanced Computer Graphics Animation*, (July 1984).
21. MILLER, GAVIN S. P., "From Wire-Frames to Furry Animals," *Graphics Interface '88*, pp. 138-145 (1988).
22. NORTON, ALAN, ALYN P. ROCKWOOD, AND PHILIP T. SKOLMOSKI, "Clamping: A Method of Antialiasing Textured Surfaces by Bandwidth Limiting in Object Space," *Computer Graphics* **16**(3) pp. 1-8 (August 1982).
23. PEACHEY, DARWYN, "Texture On Demand," *(submitted for publication)*, (1990).
24. PERLIN, KEN, "An Image Synthesizer," *Computer Graphics* **19**(3) pp. 287-296 (July 1985).
25. PERLIN, KEN AND ERIC M. HOFFERT, "Hypertexture," *Computer Graphics* **23**(3) pp. 253-262 (July 1989).
26. PORTER, THOMAS AND TOM DUFF, "Compositing Digital Images," *Computer Graphics* **18**(3) pp. 253-260 (July 1984).
27. STEELE, GUY L., *Common Lisp*, Digital Press, Burlington, MA (1984).
28. UPSTILL, STEVE, *The RenderMan Companion*, Addison-Wesley (1989).
29. VERBECK, CHANNING P. AND DONALD P. GREENBERG, "A Comprehensive Light-Source Description for Computer Graphics," *IEEE Transactions on Computer Graphics and Applications* **4**(7) pp. 66-75 (July 1984).
30. WHITTED, TURNER, "An Improved Illumination Model for Shaded Display," *Communications of the ACM* **23** pp. 343-349 (1980).
31. WHITTED, TURNER AND DAVID M. WEIMER, "A Software Testbed for the Development of 3D Raster Graphics Systems," *ACM Transactions on Graphics* **1**(1) pp. 44-58 (January 1982).

ATI Ex. 2002
IPR2023-00687
Page 10 of 10