3

REAL-TIME PROGRAMMABLE SHADING

WILLIAM R. MARK

INTRODUCTION

The materials and lighting effects in the real world are very complex, but for many years real-time graphics hardware could only support a few simple shading models. VLSI technology has now progressed to the point where it is possible for a single-chip real-time graphics processor to support complex, user-programmable shading programs with high performance.

This transition in graphics hardware has enabled new applications and has changed existing ones. For visual simulation applications such as flight simulators, real-time programmable shading enables much greater visual realism. In entertainment applications such as games, programmable shading allows artists to create a unique "look." For data visualization applications such as volume rendering, programmable shading allows the transfer functions that map data values to colors to be more complex and allows these transfer functions to be modified interactively. Finally, programmable graphics hardware is sufficiently flexible that it can be used as a general-purpose parallel computer, to run applications that its designers didn't even anticipate.

Many of the sophisticated shading and lighting algorithms that were originally developed for offline rendering can now be used for real-time rendering. However, there are some significant differences between real-time programmable shading and offline programmable shading. A few of these differences are transitory and will disappear as graphics hardware continues to evolve. But many of them are more fundamental and are likely to persist for many years. One major goal of this chapter is to explain these fundamental differences and their implications—in other words, to provide a bridge between the established domain of offline programmable shading and the newer domain of real-time programmable shading. The other major goals of this chapter are to describe the current state of the art in real-time programmable shading and to provide examples of shaders that run on real-time hardware.

After this introduction, the remainder of this chapter is organized as follows:

- First we describe why graphics hardware is fast and the limitations that it imposes on programmability in order to provide this high performance.
- Then we provide an initial example of a real-time shading program and show that it can be written in two very different styles—one style is a RenderMan-like style, and the other style is closer to the underlying hardware representation of the program.
- The next two sections discuss the factorization of shading into "surface" and "light" computations and describe the interface between applications and shading programs.
- We then provide two longer shading programs that demonstrate useful real-time shading techniques.
- We conclude with some strategies for developing real-time shaders and some thoughts on the future of real-time programmable shading.

The remainder of this introductory section will cover three topics. First, we will summarize the fundamental differences between real-time shading and offline shading. Then, we will explain our reasons for using a high-level language to program real-time graphics hardware, instead of an assembly-level language. Finally, we will list some of the topics that are *not* covered in this chapter because they are very similar to the corresponding topics from offline procedural shading and are thus well covered earlier in this book and elsewhere.

What Makes Real-Time Shading Different?

Although real-time programmable shading is similar in many respects to offline programmable shading, there are several unique characteristics of real-time programmable shading:

- *Most applications are interactive.* As a result, the shader writer usually does not know which viewpoints will be used to view an object and may not even know which lights will be near an object.
- *Performance is critical.* Real-time rendering requires a greater emphasis on performance than offline rendering. Furthermore, performance must be consistent from frame to frame.

- *Shaders execute on graphics hardware.* Graphics hardware provides high performance at low cost, but imposes certain restrictions on shading programs in order to obtain this high performance.

We will discuss each of these points in more detail.

In offline rendering, the viewpoints used with any particular shader are known prior to the one-time final rendering, so shaders and lighting can be tuned for these viewpoints. In contrast, most interactive applications (e.g., games) give the user control over the viewpoint, either directly or indirectly. Thus, the shader writer does not know in advance what viewpoints will be used with the shader. As a result, the tasks of scene lighting and shader antialiasing are more difficult than they are in the offline case. Programmers must insure that their solutions work for any viewpoint, rather than just for a particular set of viewpoints that were specified by the movie director.

One likely consequence of this difference is that the adoption rate of physically based illumination techniques will be more rapid in real-time rendering than it has been in offline movie rendering.

Although real-time shading presents new challenges to the shader programmer, it also provides one very important advantage: Scenes can be re-rendered in fractions of a second rather than minutes or hours. As a result, it is much easier to experiment with changes to a shader until the desired visual effect is achieved. The importance of this rapid modify/compile/render cycle cannot be overstated. More than most other types of programming, shader development is inherently iterative because the algorithms being implemented are ill-defined approximations to complex real-world phenomena. The ultimate test for a shader is "Does it look right?" so the development process is most efficient when it can be rapidly driven by this test.

Rendering performance is an issue in almost any type of rendering, but it is especially important for real-time rendering. The final frames for a real-time application may be rendered billions of times. In contrast, the final frames for a movie that is generated with an offline renderer are only generated once. The economics of these two cases are very different, requiring that much more effort be expended on optimizing rendering performance for the real-time case.

The consequences of this difference are apparent when we examine the shader-writing styles for offline rendering and real-time rendering. Many offline shaders are designed in part as modeling tools. As a result, these shaders have many parameters and option flags and can be thousands of lines long. These shaders provide great flexibility to artists, but at a cost in complexity. In contrast, real-time shaders are more highly specialized. Real-time shaders also rely more heavily on performance-tuning techniques such as the use of table lookups for vector normalization.

Real-time rendering imposes one additional performance requirement that is not present for offline rendering: the rendering cost of every frame must be below a certain threshold to provide acceptable interactivity. In contrast, in a computer-generated movie, it is acceptable for a few frames to take an order of magnitude longer to render than the average frame.

Offline shading systems use only the CPU, but real-time shading systems perform most of their computations on the graphics hardware (GPU), because the GPU's peak performance is about two orders of magnitude greater than that of the CPU. But this high performance comes with a penalty—the GPU programming model is more restricted than that of the CPU. These restrictions are necessary to allow programmable GPUs to provide high performance at a low cost. The section "Real-Time Graphics Hardware" describes these restrictions and explains how they enable high performance.

Why Use a High-Level Programming Language?

Most graphics hardware supports one or more low-level programming interfaces, generally at the assembly language level. It is possible to program the GPU using these low-level interfaces, but we won't describe them in this chapter. Instead, this chapter uses a high-level shading language for GPU programming. Programs written in this high-level language (often referred to as *shaders*) must be compiled into assembly language before being run.

A high-level shading language provides a number of advantages over an assembly language interface:

1. It is easier and more productive to program in a high-level language. Ease of programming is especially important when developing shaders because the best approach to shader development is to try something, look at it, and then iterate. In contrast, programming at the assembly language level is sufficiently painful that it discourages exploration of ideas.
2. Writing programs in a high-level language makes it easy to create "libraries" of shaders. More importantly, the shaders from such a library can be easily modified and/or combined to meet specific needs.
3. A high-level language provides at least some degree of hardware independence. In contrast, GPU assembly languages are often completely different for

products from different hardware companies, and in many cases are even different for different products from the same company.

4. A high-level language (and associated compiler) can virtualize the hardware to hide hardware resource limits. For example, a compiler can use multiple rendering passes to hide limits on the number of textures, to hide limits on the number of temporary variables, or to hide limits on the size of a shader. With the proper hardware support for this virtualization, it can be completely invisible to the programmer.

5. A high-level language has the potential to provide *better* performance than typical handwritten assembly language code. The reason is that the compiler can optimize shaders using detailed information about the GPU that would be too tedious to use when writing assembly language code by hand. When the compiler is developed by the hardware vendor, the compiler may also use information about the GPU design that the hardware vendor would be unwilling to disclose publicly for competitive reasons.

For these reasons, we expect that high-level languages will become the standard GPU programming interface in the future, and we focus on them in this chapter.

What You Need to Learn Elsewhere

We've already discussed some of the differences between real-time and offline programmable shading, but the two are also similar in many respects. Many of the features of real-time shading languages have been adopted from the RenderMan shading language, and many of the techniques used in offline shaders are equally useful in real-time shaders. Since there is already a wide variety of high-quality information available about offline procedural shading, this chapter largely avoids repeating this information. Instead, we encourage you to use these other resources, then rely on this chapter to learn about the unique characteristics of real-time shading.

For readers who are unfamiliar with procedural shading, Chapter 2 of this book provides essential background material. The important topics in Chapter 2 include the basic introduction to procedural shading; the distinction between "image-based" and "procedural" approaches to shading; and the discussion of antialiasing for procedural shaders.

The standard procedural shading language for offline rendering is the RenderMan shading language. Although current real-time shading languages differ in important ways from RenderMan, almost all of the basic shader-writing strategies that are used for RenderMan shaders are equally applicable to real-time shaders. There are two books that describe the RenderMan shading language and discuss shader-writing techniques. The first is *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics,* by Steve Upstill (1990). The second is *Advanced RenderMan: Creating CGI for Motion Pictures,* by Tony Apodaca and Larry Gritz (2000). The *Advanced RenderMan* book is a particularly good complement to the material in this chapter.

There are several different implementations of the RenderMan standard. The first one is Pixar's PhotoRealistic RenderMan (PRMan). Another is the Blue Moon Rendering Tools (BMRT). Most RenderMan shaders that are simple will run on any implementation of RenderMan, and these shaders are a valuable source of ideas for real-time shading.

If you are writing a complete real-time graphics application, you will need to understand the entire graphics pipeline, not just the programmable shading parts of it. There are two major interfaces for controlling the entire graphics pipeline—Direct3D and OpenGL. Historically, OpenGL has been the better-architected API, but it often requires the use of vendor-specific extensions to access the latest hardware features. In contrast, Direct3D has been more difficult to learn and use, but has evolved more rapidly to support leading-edge features with a common interface. OpenGL is described in the *OpenGL Programming Guide* (OpenGL ARB 1999), and in extension specifications available from the Web sites of graphics hardware companies. A variety of books are published every year or two to describe the latest version of Direct3D.

Real-Time Graphics Hardware

High-performance real-time programmable shading languages are designed to run on GPUs. To make effective use of these languages and to be able to predict how their capabilities will evolve with time, it is crucial to have some understanding of graphics hardware. This section discusses some of the characteristics of graphics hardware that are most important for programmable shading.

CPUs and GPUs are designed with very different goals. CPUs are designed to provide high performance on general-purpose, *sequential* programs. In contrast, GPUs are designed to provide high performance for the *specialized* and *highly parallelizable* task of polygon rendering and shading.

GPU architectures are organized around these two themes of specialization and parallelization. For example, rasterization and texture decompression are typically performed using specialized hardware. As a result, the programmability of these parts of the graphics pipeline is limited. For other parts of the graphics pipeline that are programmable, the GPU architecture imposes certain programming restrictions to facilitate massive parallelism at low cost. For example, the programmable pixel/fragment processors in 2002-generation GPUs do not support a store-to-memory instruction because supporting this type of memory access with reasonable ordering semantics would impose extra costs in a parallel architecture.

Object Space Shading versus Screen Space Shading

Hardware graphics pipelines perform some programmable shading in object space (at vertices) and some programmable shading in screen space (in effect, at pixels). In contrast, the REYES algorithm used by Pixar's PRMan performs all shading in object space, at the vertices of automatically generated *micropolygons*.[1] Hardware pipelines use the hybrid vertex/pixel shading approach for a variety of reasons, including the need for high performance and the evolutionary path that graphics hardware has followed in the past. We will explain the two approaches to programmable shading in more detail and discuss their advantages and disadvantages.

RenderMan uses curved surfaces (both tensor-product splines and subdivision surfaces) as its primary geometric primitives. To render these surfaces, PRMan dices them into grids, then uniformly tessellates each grid into "micropolygons" that are smaller than a pixel, as in Figure 3.1(a). The programmable shader executes at each vertex of these micropolygons. The programmable shader may change the position of the vertex, as well as calculate its color and opacity. Next, for each pixel in which a micropolygon is visible, its color is evaluated at one or more screen space sample points. The color at each sample point is determined using either flat shading or Gouraud shading. The algorithm is described in more detail in Cook, Carpenter, and Catmull (1987) and Apodaca and Gritz (2000).

In contrast, graphics hardware uses triangular polygons as its primary geometric primitive. One set of programmable shading computations is performed at the vertices of these triangles, prior to their transformation into screen space. Then, the triangles are rasterized into screen space samples, referred to as *fragments*. Next, a second set of programmable computations is performed at each fragment. These two sets of programmable computations are commonly known as per-vertex and per-fragment computations, respectively. This process is illustrated in Figure 3.1(b). Finally, if multisample antialiasing is active, the visibility of a fragment may be tested at multiple points in screen space (not shown in the figure).

1. Note that other RenderMan implementations, such as BMRT, use different approaches.

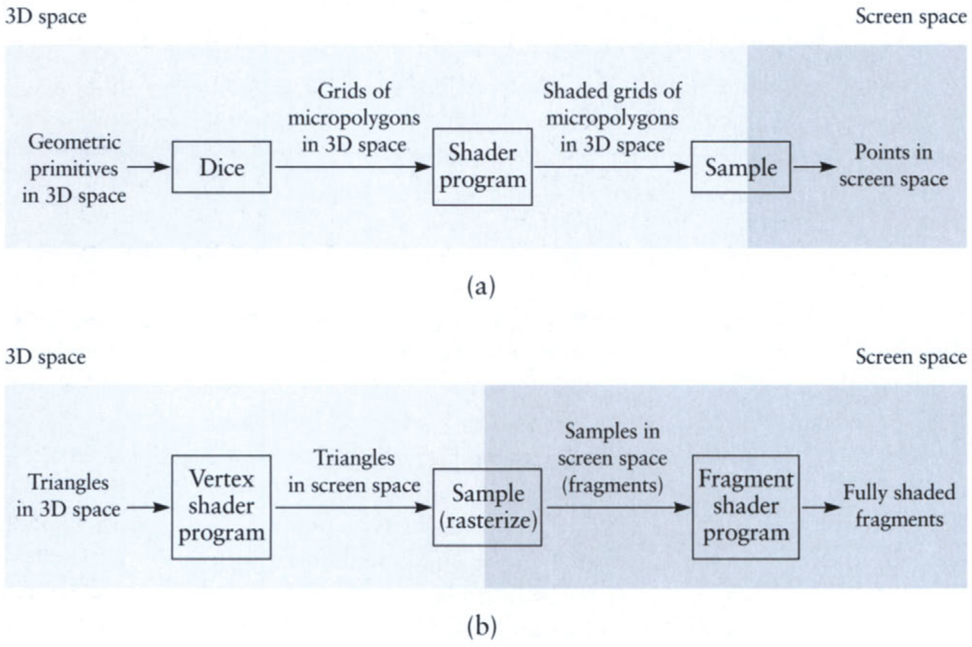

FIGURE 3.1 Comparison of (a) the REYES pipeline (used in Pixar's RenderMan) and (b) the

The most important difference between object space shading and screen space shading is that an object space shader may change the position of the surface—that is, it can displace the surface, using either a displacement map or a procedural computation. In contrast, a screen space shader is forbidden to change the three-dimensional position of a fragment because the (u,v) screen space location of the fragment has already been fixed by the rasterizer. A screen space shader is limited to changing the screen space depth of a fragment.

For rapidly varying shading computations such as specular lighting, it is important that the shading computation be performed approximately once per pixel to produce high-quality images. Screen space shading is automatically performed at this rate, but object space shading is not. The REYES algorithm solves this problem by automatically generating micropolygons of the necessary size from curved surfaces specified by the user. There are a number of reasons why this approach has not yet been used in real-time hardware:

- Historically, the performance of graphics hardware has not been high enough to allow the use of pixel-sized polygons. Transformation and lighting were performed relatively infrequently at the vertices of large polygons, and these results were interpolated in screen space. This interpolation is much more efficient than reevaluating the shading equation at each pixel. The performance of graphics hardware has now increased to the point where this optimization is less important than it once was, but because performance is so crucial in real-time graphics, it is still useful to be able to perform some computations less often than others.

- If all shading computations are performed at object space vertices, it is crucial that polygons be approximately the size of a pixel. Adopting this approach for real-time use would require automatic tessellation of curved surfaces and/or large polygons to form micropolygons. To avoid performance loss from the CPU-to-GPU bandwidth bottleneck, this tessellation must be performed by the graphics hardware. As of this writing, some hardware does support hardware tessellation of curved surfaces, but this hardware lacks the automatic adjustment of tessellation rate needed to guarantee pixel-sized micropolygons and has not been widely used by developers. Hardware implementation of adaptive tessellation is challenging; feedback paths of the type used in the REYES dicing algorithm are complex to implement, as are algorithms that avoid cracks at boundaries where tessellation rates change. However, we can expect that these challenges will eventually be overcome.
- Because graphics hardware has historically performed high-frequency shading computations at fragments rather than vertices, 2002-generation graphics hardware does not support the use of texture maps at vertices. A switch to 100% object space shading would require that texture mapping capability be added to the programmable vertex hardware.

There are several other differences between the pure object space shading model used by the REYES algorithm and the hybrid model used by graphics hardware:

- If a displacement map contains large, rapidly varying displacements, the object space shading model can create polygons that are large and flat-shaded in screen space. In contrast, the hybrid vertex/fragment shading model performs fragment computations once per pixel even for large polygons generated by the displacement process.
- The object space shading model allows motion blur and depth-of-field effects to be generated inexpensively. Shading computations are performed once in object space, and the shaded surfaces are sampled many times to generate the motion

blur and depth-of-field effects. In contrast, the accumulation buffer technique that is typically used to generate these effects in conjunction with screen space shading requires that the shading computations be repeated for each set of samples.

- Antialiasing of procedural shaders often relies on derivative computations, and computing these derivatives is very simple at fragments. If vertices are specified directly (rather than automatically generated by dicing, as in REYES), computing derivatives at vertices is more complex than doing so at fragments because the neighborhood relationships for vertices are less regular.

One minor, but sometimes annoying, implication of the different rendering approaches used by RenderMan and real-time hardware is that real-time hardware does not make the "geometric normal" available to vertex programs. In RenderMan, the geometric normal is available as the variable `Ng` and is directly computed from the local orientation of the curved surface being rendered. In 2002-generation graphics hardware, only the shading normal `N` is available in vertex programs. However, the geometric normal can be made available to fragment programs on some graphics hardware via two-sided lighting tricks.

Parallelism

Graphics hardware relies heavily on several forms of parallelism to achieve high performance. Typically, programming any type of parallel computer is difficult because most algorithms have dependencies among the different computations required by the algorithm. These dependencies require communication and synchronization between processors. The need for communication and synchronization tends to reduce processor utilization and requires expensive hardware support.

The programmable processors in graphics hardware largely avoid these costs by restricting the communication that is allowed. As a result, the hardware can provide very high performance at low cost, as long as the user's algorithms can tolerate the restrictions on communication.

On 2002-generation graphics hardware, the programmable vertex processor allows only access to data for a *single* vertex. As a result, the hardware can perform operations on different vertices in any order or in parallel. If multiple vertex processors operate in parallel, no communication or synchronization is required between them.[2] Likewise, the programmable fragment processor allows access to data for just

2. Note that some synchronization is required when vertices are assembled into primitives, but because this operation is not programmable, it can be performed by highly specialized hardware.

one fragment. High-level real-time shading languages that compile to this hardware must expose the restriction that each vertex is processed independently and each fragment is processed independently.

If both read and write access is provided to a memory, that memory provides another form of communication. The system must define (and support) ordering semantics for the reads and writes. To avoid the costs associated with supporting these ordering semantics, 2002-generation GPUs tightly restrict memory accesses. Fragment processors can read from texture memory, but not write to it. Fragment processors can write to frame buffer memory, but not read from it. The only read access to frame buffer memory is via the nonprogrammable read/modify/write blend unit. These read/modify/write operations are performed in the order that polygons were specified, but the hardware required to implement this ordering is simplified because the fragment processor is not allowed to specify the address of the write. Instead, the write address is determined by the (nonprogrammable) rasterizer. All of these restrictions are exposed in high-level real-time shading languages. Fortunately, the restrictions are straightforward and are also present in the RenderMan shading language.

One implication of the restricted memory access model is that a compiler must store all of a shader's temporary variables in hardware registers. On a CPU, temporary variables can be "spilled" from registers to main memory, but this simple strategy does not work on a GPU because there is no general read/write access to memory. The only way to perform this spilling on 2002-generation graphics hardware is for the compiler to split the shading computation into multiple hardware rendering passes. Then, temporary variables can be saved into the graphics memory in one pass and restored from that memory in a subsequent rendering pass. Note that the CPU software must issue a barrier operation between rendering passes to switch the region of memory used for the temporary storage from write-only access ("frame buffer mode") to read-only access ("texture mode").

At the fragment level, most 2002-generation graphics hardware uses a SIMD (single instruction, multiple data) computation model—the same sequence of instructions is executed for each fragment. At the instruction set level, the SIMD model is evident from the absence of a conditional branch instruction. Hardware that uses this SIMD computation model cannot execute data-dependent loops using a single hardware rendering pass. Shading languages that compile to this hardware must either forbid these operations or implement them inefficiently by using multiple rendering passes and stencil-buffer tests.[3] If/then/else operations can be executed within

3. This strategy is inefficient because it consumes additional memory bandwidth and requires retransformation and rerasterization of geometry.

a single rendering pass, but only by effectively executing both the "then" and "else" clauses for every fragment, and then choosing which result to keep.

The situation is different for vertex computations. Some 2002-generation hardware supports looping and branching for vertex computations. Thus, the vertex hardware uses an SPMD (single program, multiple data) computation model—different vertices execute the same program, but they may take different execution paths through that program.

The SIMD computation model is less expensive to implement in hardware than the SPMD model because the instruction storage, fetch, and decode hardware can be shared among many processing units. Furthermore, SIMD execution can easily guarantee that the computations for different fragments complete in the correct order. However, data-dependent looping is valuable for algorithms such as anisotropic filtering, antialiasing of turbulent noise functions, and traversal of complex data structures. For these reasons, it is likely that graphics hardware will eventually incorporate direct support for conditional branch instructions in fragment programs.

Hardware Data Types

Graphics hardware has historically used low-precision, fixed-point data types for fragment computations. The registers and computation units for smaller data types are less expensive than those for larger data types; for example, the die area required by a hardware multiplier grows approximately as the *square* of the number of bits of precision. But general-purpose shader programs require higher-precision data types, especially for texture coordinate computations (e.g., computing the index into an environment map). Shader programs are also easier to write if the hardware supports floating-point data types instead of fixed-point data types, since the availability of floating-point data types frees the programmer from having to worry about scale factors.

For these reasons, 2002-generation hardware supports 32-bit floating-point fragment arithmetic. However, because lower-precision arithmetic can be performed more quickly, 2002-generation hardware also supports lower-precision floating-point and fixed-point data types. Shaders that use these lower-precision data types will typically run faster than shaders that reply primarily on the 32-bit floating-point data type.

Pre-2002-generation graphics hardware only supports 8-bit or 9-bit fixed-point arithmetic for most fragment computations. If a shader that is written in a high-level language needs to run on this older hardware, its variables must all be declared using these older fixed-point data types. Even on newer hardware, shader writers must restrict themselves to using data types that are hardware supported on all of the hardware platforms that they are targeting. Compilers can effectively hide some differences between hardware instruction sets, but they cannot efficiently hide differences between hardware data types.

Resource Limits

In general, graphics hardware has limits on resources of various types. These limited resources include the following:

- Memory for storing instructions
- Registers for storing temporary variables
- Interpolators for vertex-to-fragment interpolants
- Texture units
- Memory for textures and frame buffer

Ideally, these resource limits would be hidden from the high-level language programmer by some combination of hardware-based and software-based virtualization techniques. As an analogy, CPUs hide limits on main memory size by using a virtual memory system that pages data to and from disk. Performance may degrade when this virtualization is activated, but programs produce exactly the same results that they would if more main memory were available. CPU hardware includes features that are designed to support this virtualization. Unfortunately, 2002-generation graphics hardware does not yet support resource virtualization as well as CPUs. Most resource limits can be circumvented by using multiple rendering passes, but multipass rendering is an imperfect virtualization technique. Generally, the need to resend geometry from the CPU to the GPU cannot be completely hidden from the application program. Furthermore, multipass rendering produces incorrect results for overlapping, partially transparent surfaces. As a result, shader programmers must sometimes concern themselves with limits on the resources that a program may consume.

Even if virtualization is implemented well, it may cause nonlinear changes in performance that are sufficiently large to be of concern to programmers. For example, adding one more statement to a shader can cause its performance to drop in half on some graphics hardware. Programmers who are concerned about performance on particular hardware must understand the resource limits and performance characteristics of that hardware.

Memory Bandwidth and Performance Tuning

Z-buffered rendering consumes an enormous amount of memory bandwidth for reading and writing the frame buffer. If surfaces are texture mapped, additional memory bandwidth is required to read the texels from texture memory. As a result, the performance of real-time graphics hardware has historically been limited primarily by memory bandwidth, especially when rendering large polygons.

As VLSI technology advances, this limitation becomes more pronounced because on-chip computational performance improves more rapidly than bandwidth to off-chip memories. One solution to this problem would be to put the entire frame buffer, plus a frame-sized texture cache, on chip. But as of this writing, this solution is not yet economical for PC graphics hardware.

Graphics hardware designers have instead taken a different approach—-they have placed the increased computational performance at the programmer's disposal, by adding the programmable hardware units that are the focus of this chapter. Besides enabling the generation of much more realistic images, these programmable units indirectly reduce the demand for memory bandwidth by allowing programmers to use a single rendering pass to implement algorithms that used to require multiple rendering passes. In processor design terminology, GPUs have become a specialized type of stream processor (Rixner et al. 1998). Stream processors are designed to support a high compute-to-bandwidth ratio, by reading a packet of data (e.g., a vertex), executing a complex computational kernel (e.g., a vertex program) on this data, and then writing a packet of data (e.g., a transformed vertex).

Unfortunately, the availability of these programmable units makes performance tuning very complex. At any one time, rendering performance may be limited by vertex-processor performance, fragment-processor performance, memory bandwidth (including texture fetches), or any one of a whole set of other factors, such as host-to-GPU bandwidth. Detailed approaches to performance tuning are hardware dependent, but we will describe two performance-tuning techniques that are broadly applicable.

First, it is often possible to trade compute performance for memory bandwidth and vice versa. The simplest example of such a trade-off is the use of a table lookup (i.e., texture read) in place of a complex computation. If a shading program is compute limited, then making this trade-off will improve rendering performance.

Second, for programs that are limited by memory bandwidth, it may be possible to improve performance by more effectively utilizing the hardware's texture cache. In general, reducing the size of a texture will improve performance, as will restructuring programs that use table lookups so that they are more likely to repeatedly access the same table entries.

SIMPLE EXAMPLES

Throughout the rest of this chapter, we will use a series of example shading programs that show some of the effects that can be achieved with a real-time programmable shading system. These shaders are written using the Stanford real-time shading language. The Stanford real-time shading system is an experimental system designed to show that a hardware-independent shading language can be efficiently compiled to programmable graphics hardware. At the time that we developed the examples in this chapter, it was the only high level language available for mainstream programmable graphics hardware.

As this book was in the final stages of publication, commercially supported programming languages for programmable GPUs were beginning to appear. These languages, including NVIDIA's Cg language, follow the philosophy of the C language in the sense that they are designed primarily to provide convenient access to all of the capabilities of the underlying hardware, rather than to facilitate any particular use of the hardware. In contrast, RenderMan, and to a lesser extent the Stanford shading language, are designed for the specific task of surface shading and include a variety of features designed to support that task.

All but one of the examples in this chapter avoid most of these specialized features and therefore can be easily ported to commercially supported GPU programming languages such as Cg. However, you may want to check the author's Web site for updated versions of the examples before porting them yourself.

Vertex and Fragment Code in the Stanford Shading System

Most of the commercially available GPU programming languages require that the user write two separate programs—a vertex program and a fragment program. One of the unique features of the Stanford shading language is the ability to combine vertex and fragment computations within a single program. The user specifies whether computations are per vertex or per fragment by using the `vertex` and `fragment` type modifiers. The following code shows how a simple program can use these type modifiers. The Stanford language also allows computations to be performed once for each group of primitives, using the `primitive group` type modifier. Typically, this capability is used for operations such as inverting and transposing the modelview matrix. On 2002-era hardware, these operations are executed on the CPU rather than the GPU.

```
//
// A very simple program written in the Stanford shading language. This program
// includes both vertex computations and fragment computations. The fragment
// type modifier indicates that a variable is instanced once for each fragment.
// Likewise, the vertex and primitive group type modifiers indicate that a vari-
// able is instanced once for each vertex or once for each group of primitives,
// respectively.
//
// Scale 'u' component of texture coordinate, and apply texture to surface
//
surface shader float4
applytexture (vertex float4 uv, primitive group texref star) {
    vertex float4 uv_vert = {uv[0]*2, uv[1], 0, 1}; // Scale texcoord
    fragment float4 uv_frag = uv_vert;              // Interpolate
    fragment float4 surf = texture(star, uv_frag);  // Texture lookup
    return surf;
}
```

The Stanford language uses simple "type promotion" rules to determine whether specific computations within an expression are mapped to the CPU, the vertex processor, or the fragment processor. For example, when a `vertex` expression is multiplied by another `vertex` expression, the computation is performed by the vertex processor and yields another `vertex` expression. When a `vertex` expression is multiplied by a `fragment` expression, the `vertex` expression is interpolated to produce a `fragment` expression before performing the multiplication on the fragment processor. The compiler is responsible for using these rules to split the user's single program into separate vertex and fragment programs that can be executed on the graphics hardware, and a separate primitive group program that is executed on the CPU.

This unified vertex/fragment programming model is very convenient for straight-line code, but it becomes unwieldly in a language that supports imperative looping constructs, such as "for" and "while" loops. The Stanford language doesn't support these constructs, but the newer commercially available languages do, and therefore they require the user to write separate vertex and fragment programs. The example shaders in this chapter are designed to be easily split into separate vertex and fragment programs for such languages.

Two Versions of the Heidrich/Banks Anisotropic Shader

One of the most important advantages of programmable graphics hardware is that it can be used to implement almost any lighting model. For anisotropic surfaces, one lighting model that is especially appropriate for real-time use is Heidrich and Seidel's (1999) formulation of the Banks (1994) anisotropic lighting model. Figure 3.2 illustrates this lighting model applied to a sphere.

Heidrich and Seidel's lighting model was designed to execute efficiently on graphics hardware. The vertex-processing hardware computes a pair of dot

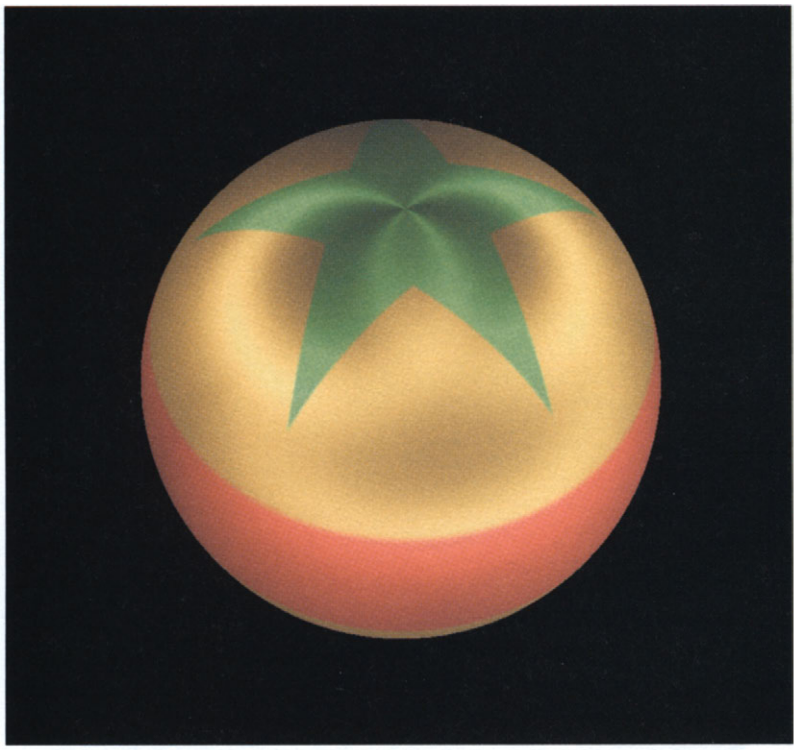

FIGURE 3.2 Banks/Heidrich anisotropic lighting shader applied to a sphere.

products, and these dot products are used as indices for a 2D table lookup at each fragment. The 2D table is precomputed and stored in a 2D texture map. Heidrich and Seidel have described a whole family of real-time lighting models that are based on the idea of factoring lighting models into independent terms that depend on only one or two angles. Each such term can be stored in a 1D or 2D texture map. The SIGGRAPH paper that describes their approach (Heidrich and Seidel 1999) is worthwhile reading for anyone who is interested in lighting models that are both realistic and efficient.

We will now look at two programs that implement the Banks/Heidrich lighting model using two different shader-writing styles. The first program expresses almost all of the transform and shading computations explicitly. For example, the first two statements transform the position from homogeneous model space to Cartesian eye space, and later statements transform the surface normal and binormal vectors (`Ndir` and `Bdir`) to eye space. These computations rely directly on parameters provided through the geometry API, such as the vertex position and the modelview matrix.

```
//
// The first of two implementations of the Heidrich/Banks anisotropic lighting
// model. This implementation explicitly expresses every computation that will
// be performed by the programmable vertex and fragment hardware, except for the
// final transformation of position into clip space. The anisotex texture must
// be precomputed as described in Heidrich and Seidel (1999). Adapted from a
// shader written by Kekoa Proudfoot; used with permission.
//
// Relies on the following predefined variables:
//  __position      = object space position for vertex (from API vertex calls)
//  __normal        = object space normal for vertex   (from API vertex calls)
//  __binormal      = object space binormal for vertex (from API vertex calls)
//  __modelview     = 4 × 4 modelview matrix
//  __invmodelview3 = inverse transpose of 3 × 3 modelview matrix
//  __ambient       = global ambient color
//
surface shader float4
anisotropic1 (texref star,         // Surface texture
              texref anisotex,     // Precomputed table for BRDF calc
              float4 lightpos,     // Light position
              float4 lightcolor) { // Light color
    //
    // Determine light's location and its intensity at surface
    //
    vertex float4 Peye4 = __modelview * __position; // calc eye space obj pos
    vertex float3 Peye3 = {Peye4[0], Peye4[1], Peye4[2]} / Peye4[3];
    vertex float3 Lvec  = rgb(lightpos) - Peye3;
    vertex float3 Ldir  = normalize(Lvec);
    vertex float4 light_intensity = lightcolor / (1.0 + .01*length(Lvec));
    //
    // Look up surface texture using computed texture coordinates
    //
    vertex float4 surf_uv = {0.5*__position[2]+0.5, 0.5*__position[0]+0.5, 0, 1};
    fragment float4 surf  = texture(star, surf_uv);
    //
    // Compute eye space normal and binormal direction vectors
    //
    vertex float3 Ndir = normalize(__invmodelview3 * __objnormal);
    vertex float3 Bdir = normalize(__modelview3 * __objbinormal);
    //
    // Heidrich/Banks anisotropic lighting
    // Uses a texture as table lookup
    //
    vertex   float3 Edir     =normalize(-Peye3); // Direction from surface to eye
    vertex   float4 aniso_uv = {0.5*dot(Bdir,Edir)+0.5, 0.5*dot(Bdir,Ldir)+0.5, 0, 1};
    fragment float4 anisotrp = light_intensity *
                               max(dot(Ndir,Ldir),0) * texture(anisotex, aniso_uv);
    //
    // Calculate ambient term; modulate lighting by surface texture
    //
    float4 Ma = {0.1, 0.1, 0.1, 1.0}; // Ambient coefficient
    return surf * (Ma*__ambient + anisotrp);
}
```

This program also explicitly specifies whether variables are instanced at each vertex or at each fragment. This style of programming makes it clear which computations will be performed in the vertex-processing hardware, and which will be performed in the fragment-processing hardware. It also makes it easy to port this program to shading languages that require vertex and fragment computations to be specified in separate programs.

The second version of the Heidrich/Banks shader program looks quite different, although it compiles to the same hardware operations:

```
//
// The second of two implementations of the Heidrich/Banks anisotropic lighting
// model. This implementation uses separate surface and light programs and
// relies on implicitly specified computations. Two unique features of the
// Stanford system are used here: The perlight variables in the surface shader
// are instanced once for each active light shader. The integrate() function
// performs a sum over all lights of its argument. The argument to integrate()
// must be a perlight expression, but the result is an ordinary expression.
// Adapted from a shader written by Kekoa Proudfoot; used with permission.
//

//
// Light shader with constant and linear terms
//
// Relies on the following predefined variable:
//  Sdist = distance from light to surface point
//
light shader float4
simple_light (float4 lightcolor, float ac, float al)
{
    return lightcolor * (1.0 / (al * Sdist + ac));
}

//
// Heidrich/Banks anisotropic shader; works with any light shader(s).
//
// Relies on the following predefined variables:
//   Pobj  object space surface position, w=1
//   N     eye space normal vector, normalized, w=0
//   B     eye space binormal vector, normalized, w=0
//   E     eye space local eye vector, normalized, w=0
//   L     eye space light vector, normalized, w=0  (per light)
//   Cl    color of light                           (per light)
//   Ca    color of global ambient light
//
surface shader float4
anisotropic2 (texref star, texref anisotex) {
    //
    // Look up surface texture using computed texture coordinates
    //
    float4 surf_uv = {0.5*Pobj[2]+0.5, 0.5*Pobj[0]+0.5), 0, 1};
    float4 surf = texture(star, surf_uv);
    //
    // Heidrich/Banks anisotropic lighting
    // Uses a texture as table lookup
    //
    perlight float4 aniso_uv = {0.5*dot(B,E)+0.5, 0.5*dot(B,L)+0.5, 0, 1};
    perlight float4 anisotrp = Cl *
                               max(dot(N,L),0) * texture(anisotex, aniso_uv);
    //
    // Calculate ambient term; modulate lighting by surface texture
    //
    float4 Ma = {0.1, 0.1, 0.1, 1.0}; // Ambient coefficient
    return surf *(Ma * Ca + integrate(anisotrp));
}
```

The differences between the two versions of the program fall into three categories. First, the second version relies on the shading system to implicitly perform some computations, such as the transformation of the normal vector from object space to eye space. Second, this program specifies "surface" and "light" computations in two different shaders, and relies on the shading system to combine these computations into a single hardware program at compile time. We will discuss this capability of the Stanford system in more detail later. Third, this program doesn't explicitly specify whether computations are performed per vertex or per fragment. Instead, it relies on the Stanford system's default rules to determine whether computations are performed per vertex or per fragment. As mentioned earlier, these default rules are similar to the type promotion rules in the C language.

SURFACE AND LIGHT SHADERS

Historically, real-time graphics APIs have allowed the application to manage surface properties separately from light properties. Surface properties include the texturing modes and surface color(s). Light properties include the number of lights, the type of each light, and the color(s) of each light. For example, in OpenGL, the `glColorMaterial` routine modifies surface properties, and the `glLight` routine modifies light properties.

This separability of surface and light properties corresponds to the behavior we observe in the physical world—any light in a scene will modify the appearance of any surface, as long as the two are visible from each other. The RenderMan shading language supports this distinction by providing two types of shaders—*light* shaders and *surface* shaders. A light shader is responsible for calculating the intensity and color of light that is incident on any surface element from a particular light. A surface shader is responsible for determining the final surface color (as seen from a particular direction), given a set of incident light intensities.

Unfortunately, light and surface computations are not cleanly separated in graphics hardware because a Z-buffered rendering pipeline is fundamentally just a surface renderer. If the application changes the number of active lights, the graphics system must change the computation performed by the hardware at every point on every surface. For a fixed-function rendering pipeline, the hardware driver can manage the required changes to the hardware configuration, since the fixed-function pipeline's lighting model is simple enough that the configuration changes are localized to the vertex-processing part of the pipeline.

When the hardware API supports full programmability, it becomes much more difficult for the hardware driver to maintain the illusion that surface and light computations are fully separable. As a result, 2002-generation low-level programmable APIs for graphics hardware eliminate the distinction between surface and light computations. These APIs expose one programmable vertex processor and one programmable fragment processor. These processors must perform all of the per-vertex and per-fragment computations, respectively. The user is responsible for combining surface and light computations to create these programs.

Even though the hardware does not directly support separate surface and light shaders, it is possible for high-level real-time shading language to do so. However, if a language supports separability, the separability is an illusion that must be maintained by the language's compiler and run-time system. The compiler and run-time system are responsible for combining each surface shader with the active light shader(s) to create a single vertex program and single fragment program that will run on the underlying hardware.

As with almost any binding operation, there is a trade-off between the performance of the combined code and the cost of the binding operation. If the binding is done early, prior to most of the compilation process, then the resulting code is very efficient, but the binding operation itself is expensive—it is essentially a recompile. If the binding is done late, then the binding operation is inexpensive, but the resulting code is likely to be less efficient because the compiler cannot perform global optimizations.

The Stanford shading system supports separate surface and light shaders using the early-binding model. The binding process is explicit: the run-time API requires that light shaders be bound to a surface shader before compiling the surface shader. The Stanford system could have used an implicit binding model instead, but implicit early binding is dangerous because the binding process is sufficiently expensive that the application should be aware of it. In addition to supporting surface and light shaders, the Stanford system also supports deformation shaders, which can modify the position and local coordinate frame of a surface point.

As of 2002, most other high-level real-time shading languages do not support separate surface and light shaders. One reason for this omission is that built-in support for separate surfaces and lights requires that the shading system impose an interface between the two types of shaders. For example, in the RenderMan shading language, lights return a single RGB color. In the Stanford shading language, lights return a single RGBA color. Interestingly, neither of these interfaces is sufficient to support the OpenGL lighting model! The reason is that the OpenGL lighting model requires that lights return three RGBA colors—ambient, diffuse, and specular. Although such lights are not physically realizable, the flexibility they provide has proven to be very useful in practice.

Since it is difficult to pick a single surface/light interface that is appropriate for all uses, it is likely that future real-time shading languages will provide a more general mechanism for specifying interfaces between cooperating routines. In the RenderMan shading language, it is common for users to define such interfaces indirectly, using extra shader parameters and the RenderMan message-passing routines.

THE INTERFACE BETWEEN SHADERS AND APPLICATIONS

A shader program is not a stand-alone program. It operates on data (e.g., vertices) provided through some external mechanism, it obtains its parameters (e.g., light positions) through some external mechanism, and it is enabled and disabled through some external mechanism. Typically, this external mechanism is an API that is either layered on top of the primary real-time API or fully integrated with it. For example, the Stanford shading system uses a set of API calls that are implemented on top of OpenGL, while the proposed OpenGL 2.0 shading language requires changes to the OpenGL API itself. Although the RenderMan standard is best known for its shading language, it also includes an entire API that is used to specify geometry, to set shader parameters, and to generally control the rendering process.

For programmers who migrate from fixed-function real-time graphics pipelines to programmable pipelines, it is useful to understand which parts of the fixed-function API are subsumed by shading programs and which are not. Most of the state management API routines are replaced by the shading language. For example, a shader program replaces the fixed-function API routines used to configure texture blending, to configure lighting, and to set texture coordinate generation modes. However, current shading languages do not replace the API commands used to load textures. The application program is responsible for loading textures before using a shader.

A programmable shading system retains the fixed-function API routines used to feed data down the graphics pipeline and augments them with new routines. For example, the API routines used to specify the position and normal vector of a vertex are retained. New API routines are provided to allow the specification of shader-specific vertex parameters (e.g., skinning weights) and to allow the specification of shader-specific state (e.g., "time," for an animated shader).

To use a more concrete example, the Stanford shading system provides API routines that allow an application to do the following:

- Load a shader's source code from a file
- Associate a light shader with a surface shader
- Compile a surface shader and its associated light shader(s)
- Bind to a compiled shader (i.e., prepare to render with it)
- Specify values of arbitrary shader parameters (either through immediate-mode style commands or through vertex arrays)
- Specify vertices to be rendered (again, either through immediate-mode style commands or through vertex arrays)

The following program shows the API calls required to render one face of a cube using the Stanford shading system.

```
//
// Pieces of a C program that make the API calls required to render one face of a
// cube using the Stanford shading system. All of the sgl* calls are shading API
// calls. sglParameterHandle() binds a numeric handle to a string name.
// sglParameter4fv() specifies the value of a shader parameter, using a numeric
// handle. sglBindShader() specifies which shader should be used for rendering.
// The other sgl* calls are wrappers around the similarly named OpenGL calls.
// Note that this listing omits the API calls required to initialize the shading
// system and to compile the shader.
//
// Setup
//
float green[] = {0.0, 1.0, 0.0, 1.0};
float a[]     = {0.0, 0.0, 0.0, 1.0};
float b[]     = {0.0, 1.0, 0.0, 1.0};
float c[]     = {1.0, 1.0, 0.0, 1.0};
float d[]     = {1.0, 0.0, 0.0, 1.0};
//
// Assign numeric handles to parameter names
// Only needs to be done once for a shader
// This shader requires a "surfcolor" parameter and a "uv" parameter.
//
const int UV        = 3;
const int SURFCOLOR = 4;
sglBindShader(200); // Bind a shader that was compiled earlier
sglParameterHandle("uv",        UV);
sglParameterHandle("surfcolor", SURFCOLOR);
.
.
.

//
// Render y=1 face of a cube
// Surface color is green; assign "uv" coordinates for texturing
//
sglBindShader(200); // Bind a shader that was compiled earlier
sglBegin(GL_QUADS);
sglParameter4fv(SURFCOLOR, green);
sglNormal3f(0.0,1.0,0.0);
sglParameter4fv(UV, a);    sglVertex3f(-1.0, 1.0, -1.0);
sglParameter4fv(UV, b);    sglVertex3f( 1.0, 1.0, -1.0);
sglParameter4fv(UV, c);    sglVertex3f( 1.0, 1.0,  1.0);
sglParameter4fv(UV, d);    sglVertex3f(-1.0, 1.0,  1.0);
sglEnd();
```

It is important to remember that most programmable shaders require significant support from the main application program. In addition to binding the shader, the application program must specify values for the shader's parameters, load textures, and provide any auxiliary data that is needed in conjunction with each vertex and/or triangle. When designing a shader, it is crucial to decide how much of a burden you are willing to impose on the application program. For example, will you restrict your shader to using a specific set of values that is already provided with each vertex by the application? Or are you willing to require that the application provide new per-vertex values just for your shader?

MORE EXAMPLES

This section discusses two shading programs that are substantially more complex than the previous examples. The first is a shader used for volume rendering. It requires hardware capabilities such as dependent texturing that were not supported on PC graphics hardware prior to NVIDIA's GeForce3. The second is a shader for procedurally generated flame. It consists entirely of fragment operations and requires an NVIDIA NV30 (fall 2002) or better to run.

Volume-Rendering Shader

Now that consumer-level graphics cards support both 3D textures and fragment-level programmability, it is possible to use these cards to perform volume rendering with a variety of classification and lighting functions. This type of volume rendering is implemented by rendering a series of 2D slices from a 3D texture and compositing the slices as they are rendered (Cabral, Cam, and Foran 1994; Van Gelder and Kim 1996). The volume shader is applied to each slice as it is rendered and composited.

The following program is an example of this type of shader. Figure 3.3 illustrates the use of this shader. Many variants on this theme are possible; for example, better-looking results can be obtained by using a double-sided lighting model in place of the single-sided lighting model used in the `lightmodel_gradient()` function.

```
//
// A volume-rendering shader written in the Stanford shading language.
// On each invocation, the shader classifies and shades one sample from the
// volume. Information about the volume is stored in 3D texture maps. This
// shader was written by Ren Ng; used with permission.
//
// functions to transform 3 and 4 vectors from
// eye space to object space
//
surface float3 objectspace(float3 V) {return (invert(affine(__modelview))*V);}
surface float4 objectspace(float4 V) {return (invert(__modelview)*V);}

// simple light shader
//
light shader float4
```

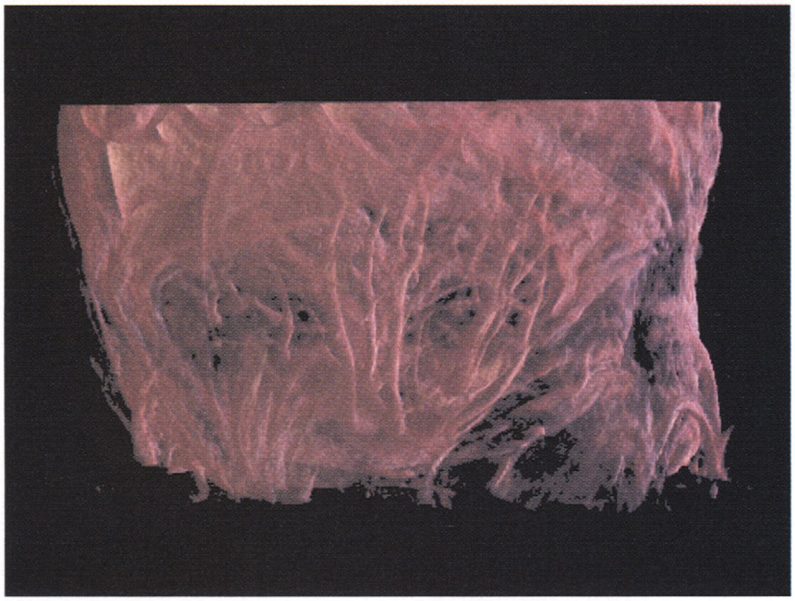

FIGURE 3.3 Using a volume-rendering shader to visualize a 3D MRI of a mouse abdomen. Ren Ng wrote the shader and captured this frame. The mouse data set is courtesy of G. A. Johnson, G. P. Cofer, S. L. Gewalt, and L. W. Hedlund at the Duke University Center for In Vivo Microscopy.

```
constant_light(float4 color) {
  return color;
}

// function to do specular & diffuse lighting based on a gradient field
//
// Hobj: the half angle vector in object space
// Lobj: the lighting vector in object space
// Nobj: the gradient vector in object space
// a: ambient term
// d: diffuse coefficient
// s: specular coefficient
//
surface float3
lightmodel_gradient (perlight float3 Hobj,
                     perlight float3 Lobj,
                     fragment float3 Nobj,
                     float3 a, float3 d, float3 s) {
    // Diffuse
    perlight float NdotL = dot(Nobj, Lobj);
    perlight float3 diff = d * clamp01(NdotL);
    // Specular exponent of 8.0
    perlight float NdotH  = clamp01(dot(Nobj, Hobj));
    perlight float NdotH2 = NdotH  * NdotH;
    perlight float NdotH4 = NdotH2 * NdotH2;
    perlight float NdotH8 = NdotH4 * NdotH4;
    perlight float3 spec  = s * NdotH8;

    // Combine
    return integrate(a + rgb(Cl) * (diff + spec));
}

// Surface shader for resampling polygons drawn through a volume.
// Note that compositing is set up by the calling application.
//
// density_plus_gradientmag: a 2-component, 3D texture containing
//    density in the red channel and the magnitude of gradient in alpha
// gradient: a 3-component, signed 3D texture containing the gradient
// color_opacity_transfer2d: a 2D texture that contains an RGBA value
//    (a base color and opacity), to be indexed by gradient magnitude
//     and density
// voxelsPerSlice: inverse of the average number of resampling slices
//    that pass through each voxel
// ambientColor: ambient lighting color
// specularColor: specular lighting color
// objToTex: a matrix that transforms from the object coordinates of
//    the volume to texture coordinates. This matrix provides the client
//    application with flexibility in exactly how the volume is
//    stored in an OpenGL texture
// opacityFactor: a dial for the application to easily change the
//    overall opacity of the volume.
//
surface shader float4
volume_shader (texref density_plus_gradientmag,
               texref gradient,
               texref color_opacity_transfer2d,
               primitive group float voxelsPerSlice,
               primitive group float4 ambientColor,
               primitive group float4 specularColor,
               primitive group matrix4 objToTex,
               primitive group float opacityFactor) {
  // texture coordinate set-up: convert from world
  // coordinates to object coordinates.
  matrix4 worldToTex = objToTex * invert(__modelview);
  float4 uvw         = worldToTex * P;

  // Classification: map (density, gradient magnitude) -> (base color, alpha)
  //                     using a dependent texture lookup
  fragment float4 density_maggrad = texture3d(density_plus_gradientmag, uvw);
  fragment float4 dep_uv = { density_maggrad[3], density_maggrad[0], 0, 1 };
  fragment float4 basecolor_alpha = texture(color_opacity_transfer2d, dep_uv);
  fragment float3 basecolor = rgb(basecolor_alpha);
  fragment float alpha = basecolor_alpha[3] * (4.0 * voxelsPerSlice);

  // Shading: Blinn-Phong model, evaluated in object space
  //  3D texture gives an object space normal vector
  //  transform eye space H and L vectors into object space
  fragment float3 Nobj = rgb(texture3d(gradient, uvw)); // uses signed texture
  perlight float3 Hobj = normalize(objectspace(H));
  perlight float3 Lobj = normalize(objectspace(L));
  float3 A = rgb(ambientColor);
  float3 D = basecolor;
  float3 S = rgb(specularColor);
  fragment float3 color = lightmodel_gradient(Hobj, Lobj, Nobj, A, D, S);

  float opacFact = clamp01(opacityFactor);

  fragment float4 rgba = {opacFact * (4.0 * alpha) * color, // RGB
                          opacFact * (4.0 * alpha) }; // ALPHA
  return rgba;
}
```

Noise-Based Procedural Flame

The following program is an animated flame shader. This shader is sufficiently complex that it is at the outer extreme of what is reasonable in a real-time shader in 2002. Figure 3.4 shows a single frame generated using this shader.

```
//
// An animated flame shader consisting entirely of per-fragment computations.
// This shader is written in the Stanford shading language and is designed to
// be applied to a single square polygon with (u,v) in the range [0,1]. It
// compiles to 122 NV_fragment_program instructions. This shader is ©2001
// NVIDIA; used with permission.
//
float abs(float x) {
  return(select(x < 0, -x, x));
}

//
// fastspline--Evaluate spline function
//            for spline parameters (1, 0.8, 0.1, 0, 0)
//
float fastspline(float x) {

  float t0 = x*2;
  float r0 = 0.8 + t0*(-0.45 + t0*(-0.8 + t0*(0.55)));
```

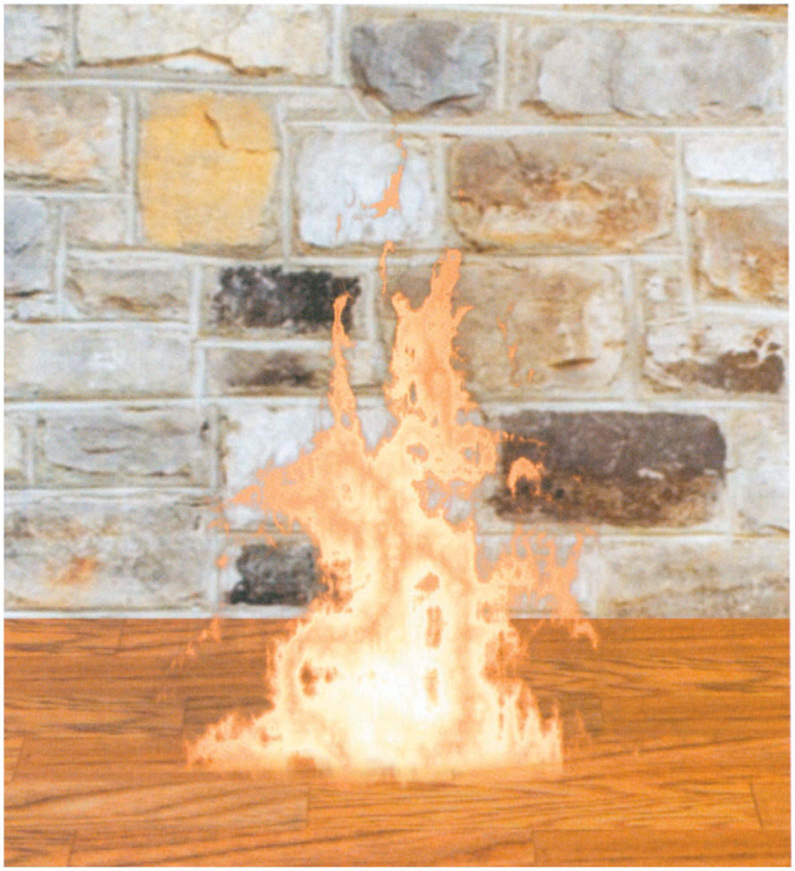

FIGURE 3.4 One frame generated using an animated flame shader. This scene consists of only three rectangles—one for the stone wall, one for the floor, and one for the flame. The rectangle with the flame shader is rendered last because the flame is partially transparent and because the flame shader uses the depth of the background at each pixel.

```
  float t1 = (x-0.5)*2;
  float r1 = 0.1 + t1*(-0.4 + t1*(0.55 + t1*(-0.25)));

  float r = select(x < 0.5, r0, r1);
  return r;
}

// Rotate 2D vector (stored in float3) by 30 degrees and scale by 2
float3 rotate30scale2(float3 x) {
  return {x[0]*2.0*0.866, x[1]*2.0*(-0.5), 0.0} +
         {x[0]*2.0*0.5,   x[1]*2.0*(-0.866), 0.0};
}

// Return one 3D noise value from a 2D noise texture
float noise3D(float3 T, texref noisetex,
              float3 L1offset, float L1weight,
              float3 L2offset, float L2weight) {
  float L1 = texture(noisetex, T+L1offset)[0];
  float L2 = texture(noisetex, T+L2offset)[0];
  float N = L2*L2weight + L1*L1weight; // Range of N is [0,1]
  return abs(N-0.5);
}

// Obtain background depth for this pixel from a texture
// that has been created with render-to-texture or equivalent.
// In this case, the depth is packed into the RGB values.
float lookup_olddepth(texref depthtex) {
    float4 dtex = texture(depthtex, rgb(xyz_screen())/1024);
    return dtex[0] + dtex[1]/256 + dtex[2]/(256*256);
}

//
// Notes on texture parameters:
//   * 'noisetex' holds random noise.
//     Its wrap mode must be set to REPEAT for both dimensions.
//   * 'permutetex' holds a 1D array of random values
//     Its wrap mode must be set to REPEAT, and it can be point-sampled.
//   * 'depthtex' holds the depth of the background,
//     with the floating-point depth value stuffed into RGB
//
surface shader float4
flame (float4 uv, primitive group float time,
       texref permutetex, texref noisetex,
       texref depthtex) {
  fragment float u = uv[0];
  fragment float v = uv[1];
  //
  // Calculate the coordinates (T) for turbulence computation
  //
  float framediff = time*0.2; // Animate--upward flame movement with time
  float3 T = {u, v+framediff, 0};
  //
  // Scale the turbulence coordinates.
  // 'freqscale' adjusts the spatial frequency of the turbulence
  // The 1/64.0 scales into the range of the 64 × 64 noise texture
  //
  constant float freqscale = 16;
  T = T * ({freqscale, freqscale/2, 0} * {1/64.0, 1/64.0, 0});
```

```
//
// To get a third (temporal) dimension of noise, we generate a pair of
// pseudorandom time-varying offsets into the 2D noise texture. We use
// a 64 × 1 (really 64 × 64) permutation texture to generate the offsets.
//
constant float changerate = 6.0;
float timeval  = time*changerate;
float timerem  = timeval - floor(timeval);
float timebase = floor(timeval); // Determines pair of samples
float L1weight = 1.0 - timerem;
float L2weight = timerem;
float ocoord1  = timebase/64.0 + 1.0/128.0;
float ocoord2  = ocoord1 + 1.0/64.0;
float3 L1offset = rgb(texture(permutetex, {ocoord1, 1.0/128.0, 0.0}));
float3 L2offset = rgb(texture(permutetex, {ocoord2, 1.0/128.0, 0.0}));


//
// Generate turbulence with four octaves of noise
//
float turb;
turb = noise3D(T, noisetex, L1offset, L1weight, L2offset, L2weight);
T    = rotate30scale2(T); // Rotate T by 30 deg and scale by 2
turb = turb +
       0.5 * noise3D(T, noisetex, L1offset, L1weight, L2offset, L2weight);
T    = rotate30scale2(T); // Rotate T by 30 deg and scale by 2
turb = turb +
       0.25 * noise3D(T, noisetex, L1offset, L1weight, L2offset, L2weight);
T    = rotate30scale2(T); // Rotate T by 30 deg and scale by 2
turb = turb +
       0.125 * noise3D(T, noisetex, L1offset, L1weight, L2offset, L2weight);
//
// Calculate the flame intensity in the "edge" (silhouette) region.
// It decreases both with the distance from the vertical axis
// and with height. It is also perturbed by the turbulence value.
//
float turbscale = 0.5 + 0.7*(1-v);
float x = (1.0/sqrt(v)) * (abs(2.0*u-1.0) + turbscale*turb);
float edgedensity = 12.5 * fastspline(clamp(x, 0, 1));
//
// Calculate the color for the edge region of the flame
//
float FlameTemperature = 0.6;
float3 FlameColor = {1.0, FlameTemperature, FlameTemperature-0.2};
float3 edgecolor  = FlameColor * edgedensity;
//
// Calculate the color for the interior region of the flame
// The flame interior is cooler near the top
//
float indensity = (2.85*turb+0.55) + v*0.35;
```

```
  float3 incolor  = FlameColor * indensity;
  //
  // Transition from the interior color to the edge color at the point
  // where the densities (and thus colors) are equal
  //
  float3 flamecolor = select(edgedensity > indensity, incolor, edgecolor);
  float density     = select(edgedensity > indensity, indensity, edgedensity);
  //
  // Clamp the color and density
  //
  flamecolor = clamp(flamecolor, 0, 1);
  density    = clamp(density, 0, 1);
  //
  // If no depth-based attenuation is desired, can exit the shader here:
  // return {flamecolor, density};
  //
  // ** Depth-based attenuation **
  // This attenuation reduces the "straight edge" effect where the flame
  // meets the floor. It is equivalent to a very simple volume
  // integration (in Z)
  //
  constant float depthscale     = 0.002;
  constant float flamethickness = depthscale; // flame thick in [0,1] Z units
  constant float depthperturb   = 1.75 * depthscale; // depth perturb scale
  constant float edgeperturb    = 3.5 * depthscale; // depth perturb at side
  float olddepth = lookup_olddepth(depthtex);
  //
  // Perturb current depth by turbulence value (to avoid uniformity):
  // and by abs(u-0.5) to give rounded appearance to flame base
  //
  float depth = xyz_screen()[2];
  depth = depth + depthperturb*turb + edgeperturb*abs(u-0.5);
  //
  // Attenuation is proportional to difference between
  // current depth (after perturbation) and background depth.
  //
  float atten = (olddepth - depth) / flamethickness;
  return {flamecolor, density*min(atten,1.0)};
} // flame
```

This shader is almost entirely procedural; although it uses three textures, these textures are used only to assist in the generation of pseudorandom noise and to provide the shader with the depth of the background geometry at the current pixel.

Pseudorandom 3D noise can be generated using a variety of techniques. At one extreme is an almost entirely procedural technique (used by most of the examples in earlier chapters in this book); at the other extreme is a single 3D texture lookup.

This flame shader uses an intermediate approach: for each 3D noise evaluation, the shader performs a pair of two 2D texture lookups and combines the results. In effect, one dimension of the 3D noise function is evaluated procedurally, and the other two dimensions are obtained from a texture. The best approach to evaluating a 3D pseudorandom noise function depends strongly on the performance characteristics of the target hardware, so this shader's approach will not be ideal for all hardware.

This flame shader consists of four basic parts. First, the shader computes a turbulence function from four octaves of pseudorandom noise. The pseudorandom noise varies as a function of `u`, `v`, and time. The dependence on time animates the flame. Second, the shader calculates the flame's intensity. The spatial dependence of this computation provides the flame with its inverted-V shape. The shader uses this intensity to determine the location of the flame's silhouette and to calculate the flame's color in the regions near the silhouette. Third, the shader calculates the flame's color in the interior (nonsilhouette) regions of the flame. This color is primarily based on the pseudorandom turbulence value, but is adjusted to make the flame look hotter at the base and cooler at the top. Finally, the shader reduces the opacity of the flame if the depth of the flame's polygon is similar to the depth of the background object. Without this depth-based attenuation, the flame would end abruptly where the flame's polygon intersects the floor.

STRATEGIES FOR DEVELOPING SHADERS

Developing complex shaders is difficult and is somewhat different from most other programming tasks. The task is partly a programming task and partly an artistic task. A real-time shader needs to achieve a visually pleasing result with the best possible performance, and it is often not clear what the best strategy is for reaching this goal. The *Advanced RenderMan* book (Apodaca and Gritz 2000) is worth reading for its insights into this process.

The following hints may also be useful:

- Find photographs and/or videos of the real-world effect you are trying to model. For example, I found several video clips and photographs of real fire before writing the preceding flame shader.

- Don't start from scratch. Find an existing shader that does something similar to what you are trying to do, and use that as a starting point. Almost all of the examples in this chapter used some other shader as a starting point. Offline shaders can often be adapted for real-time use, or at least mined for insights into the phenomenon that is being modeled. For example, the preceding flame shader evolved from a much simpler offline shader that I found in the ShaderLab2 package sold by Primitive Itch software. Graphics hardware companies usually maintain Web pages that provide examples of real-time shaders that run well on their hardware.

- Iterate! When you are working on a real-time shader, you have the luxury of being able to tweak it and see the result within seconds. Take advantage of this capability to look at the images produced by your shader as you develop it. Iteration is especially important for shaders that use a primarily phenomenological strategy (i.e., the computation is designed to look right), rather than a physically based strategy (the computation is designed to model some underlying physical process).
- Consider different approaches: in some cases an image-based approach (texture map) is best; in other cases a procedural approach works best. Most shaders use a combination of the two techniques. The light/surface interaction shader that John Carmack uses in his Doom game is an excellent example of combining computation with texture mapping to efficiently produce a desired effect on 2002-era graphics hardware.
- If shader performance is a major concern, make sure you understand which operations are efficient on the target hardware and which are not. Sometimes it can be helpful to examine the assembly language output from the compiler (if this is possible) to get ideas for improving the performance of a shader.

FUTURE GPU HARDWARE AND PROGRAMMING LANGUAGES

In 1999, GPUs were totally unprogrammable. Less than three years later, in 2002, GPUs included very general programmability for both vertex and fragment operations. Future changes are likely to be somewhat less dramatic, but we will continue to see an expansion of the generality and scope of GPU programmability. As GPU programmability matures, we can expect that hardware vendors will converge toward a common set of hardware data types and basic programmable features. This convergence will make it easier to use a high-level programming language to write shaders that are portable across a broad range of graphics hardware.

As graphics hardware becomes more general, it is likely that GPU programming languages will continue to evolve to look more like general-purpose programming languages and less like specialized shading languages. This distinction is subtle, but important. General-purpose C-like languages provide straightforward access to hardware capabilities, without making any assumptions about the kind of program that the user is writing. In contrast, shading languages make certain assumptions about the kind of code that the user is writing. For example, the RenderMan shading language includes data types for points, vectors, and normals; in some cases, the compiler automatically transforms variables from one coordinate system to another. These capabilities are convenient for writing a shader, but unnecessary and even awkward for writing other types of GPU code. Ultimately, it is likely that specialized languages like RenderMan will be available for GPUs, but that programs written in these languages will be automatically converted into a C-like language before being compiled and run.

As graphics hardware and programming languages become more general, we will also see a wider variety of algorithms implemented on GPUs. For example, ray-tracing algorithms can be implemented on 2002-generation GPUs with reasonable efficiency (Purcell et al. 2002). If nontraditional uses of the GPU such as ray tracing can be shown to be sufficiently valuable, they will in turn influence the future evolution of GPUs.

LITERATURE REVIEW

The RenderMan shading language (Hanrahan and Lawson 1990) is the standard for offline programmable shading. The definitive definition of the language is provided by the RenderMan specification (Pixar 2000). The two books on RenderMan mentioned earlier in the chapter (Upstill 1990; Apodaca and Gritz 2000) contain much information that is useful for real-time shading programmers.

Prior to the era of programmable PC graphics hardware, there were two major efforts to build real-time programmable shading systems: the PixelFlow project Olano and Lastra 1998; Leech 1998) and SGI's OpenGL Shader (Peercy et al. 2000). Many of the lessons learned from these systems are still valuable for anyone who is designing new programmable hardware or shading languages.

The first shading language for PC graphics hardware was developed as part of the Quake III game engine (Jaquays and Hook 1999). This simple language showed that game writers would be willing to use a shading language if it provided portability to different hardware platforms without an unreasonable performance penalty. The Stanford shading system was the first system designed to target highly programmable PC graphics hardware and is described in Proudfoot et al. (2001).

ACKNOWLEDGMENTS

Eric Chan, Matt Pharr, Kurt Akeley, and David Ebert read drafts of this chapter and made excellent suggestions for improvements.

The contents of this chapter were inspired by research done as part of the Stanford real-time programmable shading project. Kekoa Proudfoot, Pat Hanrahan, and I began building the Stanford real-time shading system in 1999. Svetoslav Tzvetkov, Eric Chan, Pradeep Sen, John Owens, and Philipp Schloter added key capabilities to it. Ren Ng demonstrated that the system could be used to enable interactive, programmable volume rendering, in collaboration with David Ebert, Marc Levoy, and other project members. Our industry partners ATI (Andy Grueber, Steve Morein, Evan Hart, and Jason Mitchell), NVIDIA (Matt Papakipos, Mark Kilgard, David Kirk, and many others), SGI (Mark Peercy and Marc Olano), SONY-Kihara Research Center (Takashi Totsuka and Yuji Yamaguchi), SUN (Michael Deering), and 3dfx (John Danskin, Roger Allen, and Gary Tarolli) provided information about future graphics hardware, access to prerelease hardware and drivers, and funding. Matt, Mark, and others at NVIDIA were particularly strong supporters of our project. Their comments had an influence on the research direction we chose, and they provided us with outstanding access to prerelease hardware and drivers.

More recently, Tim Purcell, Ian Buck, Pat Hanrahan, and I began investigating the use of programmable graphics hardware for ray tracing and scientific computation. My thoughts about future graphics hardware architectures have been influenced by Bill Dally's Imagine project (Rixner et al. 1998; Owens et al. 2000) and by discussions with Mark Horowitz and Kurt Akeley.

I'd like to thank Pat Hanrahan for inviting me to work in a great research environment at Stanford, for sharing his thoughts on procedural shading, and for letting me take the time to write this chapter. Finally, I thank NVIDIA, and David Kirk in particular, for allowing me to include up-to-date information about NV30 hardware in this chapter, and for granting permission to include the flame shader that I wrote for NVIDIA.