Real-Time Programmable Shading

Anselmo Lastra, Steven Molnar, Marc Olano, Yulan Wang

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

Abstract

One of the main techniques used by software renderers to produce stunningly realistic images is programmable shading—executing an arbitrarily complex program to compute the color at each pixel. Thus far, programmable shading has only been available on software rendering systems that run on general-purpose computers. Rendering each image can take from minutes to hours.

Parallel rendering engines, on the other hand, have steadily increased in generality and in performance. We believe that they are nearing the point where they will be able to perform moderately complex shading at real-time rates. Some of the obstacles to this are imposed by hardware, such as limited amounts of frame-buffer memory and the enormous computational resources that are needed to shade in real time. Other obstacles are imposed by software. For example, users generally are not granted access to the hardware at the level required for programmable shading.

This paper first explores the capabilities that are needed to perform programmable shading in real time. We then describe the design issues and algorithms for a prototype shading architecture on PixelFlow, an experimental graphics engine under construction. We demonstrate through examples and simulation that PixelFlow will be able to perform high-quality programmable shading at real-time (30 to 60 Hz) rates. We hope that our experience will be useful to shading implementors on other hardware graphics systems.

1 INTRODUCTION

The bulk of research in computer graphics has been directed toward making computer-generated images appear as realistic as possible. Since much of this effort was motivated by film making, the term "photorealistic" has been used to describe a very well rendered image, presumably one that couldn't be distinguished from a photograph of a natural scene. The latter has rarely been true, but certainly the quality of the images has improved dramatically. Practitioners generating these high-quality images have been content to wait moderately long periods of time for the rendering computations that it took to achieve these excellent results. Quality was the primary goal.

At the same time, other researchers have been striving to render images at interactive rates. The computations necessary just to determine visibility are demanding enough that, at first, only simple flat shading was possible. As technology has improved, the standard shading model on high-end commercial machines has progressed to Gouraud shading and, fairly recently, to image-based texturing. Still, rendering more geometry within tight time constraints has been most important. Interactivity was the primary goal.

We believe the time has come when one can achieve both high quality shading and interactivity. Advances in technology have made it possible to render, at interactive rates (15 Hz or greater), images that just a few years ago were considered "photorealistic". We don't claim that all of the techniques used for high-quality shading can now be done interactively, but a very large class of renderers, those dealing with local lighting effects, can be computed in real time. A notable example of this class is the Reyes renderer [1].

As evidenced by the quality of the work produced at Pixar, local effects can produce striking images. Cook, et. al. observed that many global effects can be approximated using tables, such as environment and shadow maps [1]. If a rendering system can be designed to fit the traditional rendering pipeline, communication patterns can be kept well structured, and global communications can be limited, very complex geometry with complex shading models can be rendered to produce very high quality images.

This paper consists of two main parts. The first examines the key requirements of programmable shading and explores how current hardware and software architectures can be adapted to meet these needs. The second half of the paper describes a real-time hardware/software shading architecture we have designed for PixelFlow, an experimental graphics machine currently under construction. We describe the decisions and tradeoffs in the design and give a detailed example of a complex shader that will run in real time, together with performance simulations that justify this claim.

2 TOWARD REAL-TIME SHADING

In order to achieve real-time programmable shading, we must identify the crucial requirements of software renderers and combine them with the real-time capabilities of hardware renderers, as indicated in Figure 1.

2.1 Programmability

A number of computer graphics researchers [2, 3, 4] have argued that a fixed shading model, even with adjustable parameters, is not sufficiently powerful to shade realistic images. The wide variety of surfaces makes it difficult, if not impossible, to create a single, comprehensive, shading program. Programmability allows the practitioner to create any desired effect. As a result, most software

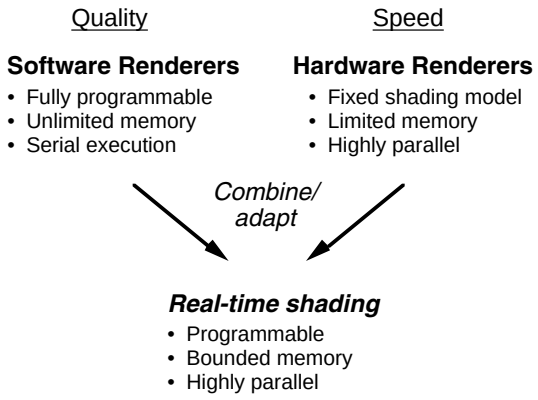

Figure 1: Merging the capabilities of software and hardware renderers.

systems designed for high-quality rendering allow users to specify the shading algorithm either in a traditional high-level language [2], or in a language specifically designed for shading [3, 4, 5].

On the other hand, computers designed for interactive graphics typically have powerful hardware for interpolating color and depth values, and more recently for computing image-based textures, but only support a fixed shading algorithm with a few adjustable parameters—most commonly linear interpolation of colors or intensities between vertices.

Some of these computers have hard-wired processors. Ironically, many have programmable processors. Even though the processors may be capable of performing adds, multiplies, and linear interpolations—the basic operations necessary for shading—they are not available for general shading because users are not given access to this level of code.

The reason for this is twofold. First, programming at this level is difficult, since pixel processors tend to be specialized and arcane, and the documentation and programming environment for them often is poor. More importantly, code written at this level is not portable. It is specific to a particular system implementation, which may change between successive machine models and even upgrades of a single model. Manufacturers also desire software to be compatible over a range of machines spanning prices and generations (and more recently, different vendors). Typically, system firmware writers are the only ones who are granted direct access to the pixel processors.

These considerations are useful and important, but they need not preclude programmable shading. Just as access to geometry rendering hardware is provided by portable graphics libraries, such as PEX and OpenGL, programmable shading can be provided by a portable language, such as the RenderMan shading language [4]. The shading language can be compiled for the particular hardware, or in low-end machines, software execution. As with current machines, more powerful, expensive workstations will shade at interactive rates, while cheaper models will produce the same images much more slowly.

We believe that programmability at the pixel level is essential to meet the goal of high-quality shading and that it can be provided in machines sufficiently powerful to shade at interactive rates.

2.2 Memory

Programmability goes hand in hand with storage. The norm in software renderers is to provide full access to the memory of the workstation, which in many ways is virtually unlimited, especially when coupled to secondary memory, such as a disk.

Systems designed for interactive graphics, on the other hand, have very limited amounts of memory for shading. A minimal frame buffer with space for *z* and color at each pixel is often the only memory available. High-end systems provide more, but the limitations are obvious when one considers that frame buffer sizes are still typically measured in *bits*, not *bytes*. Recently, graphics workstations have added memory for image-based texturing, but normally it is accessible only in regimented ways.

For high-quality shading at interactive rates, we need more memory than is available on current graphics engines, but it does not need to be as voluminous as on a workstation, and we certainly do not need advanced features such as virtual memory and access protection.

One way to make implementation feasible is to observe that memory requirements for most shaders fall into two categories: storage for local variables used during the shading calculations, and storage for tables containing texture maps, shadow maps, environment maps, etc. The local memory can be simple since the computational units only need access to their own memories, not to those of other processing elements. It must, however, be very fast because the processor can only execute as fast as it can access the local memory. On the other hand, the global memory for table lookup is not used very often, so access can be slower than that of the local memory but it must be accessible from essentially all of the processing elements.

We can take advantage of these distinct uses by dividing our memory into those two classes, memory local to a pixel and global memory that must be accessible from all of the pixels. These divisions by function are common in special purpose hardware because the technique of specializing memory can increase speed and decrease cost.

Local Memory. Let us examine the demand for local memory first. In our experience, procedural texturing is the operation that consumes the most local memory. For example, Figure 2 illustrates the amount of memory used by several of the shaders that are shipped with the RenderMan software package [6].

Shader	Local variables
carpet	24
marble	26
stippled	33
stone	23

Figure 2: Local memory requirements for RenderMan shaders.

Our experience is that storage for 30 to 40 local variables is adequate, though this does not count all of the necessary global parameters such as normals, intrinsic color, etc. Space must also be provided for a program stack used for function calling and for the temporary variables of the called functions. On a workstation this much storage, say a total of 100 floats or 200 bytes seems like a very small amount of space.

However, a frame buffer with 1600 bits of storage per pixel is very rare indeed. An observation on the nature of shading can help us solve this storage dilemma. The large memory demand can be thought of as the *peak*, temporary usage, necessary only when a surface is being shaded. Once shading is complete, only a small amount of memory is necessary, just enough to store color and

perhaps depth. It is not necessary to instantiate this amount of memory for every pixel at once. We need this working memory only for the pixels that are being shaded at any one time. One can imagine doing shading calculations one pixel at a time and saving only the final color. Realistically, however, given the amount of computation that is necessary for rendering, calculating one pixel at a time is impractical. More likely, a system will shade a number of pixels in parallel, but not necessarily the complete screen.

Table Memory. The second type of memory that is necessary is that used for table lookups, not only to apply image-based textures but also to transfer global information to surfaces by means of intermediate images, such as shadow, and environment maps. We also wish to look up stored information to use during local computation, for example, to modify lighting for local effects such as bump mapping.

The characteristics of this memory are very different from that used to hold local variables. It only needs to be accessed occasionally, but the access patterns are very general. Table memory is an exception to the modest memory requirements of a shader. This global storage pool needs to be much larger than any single local memory.

Since we want to apply several visual effects to each pixel and filtering is often required, we need to accommodate multiple table lookups per pixel. Three or four accesses per shader is probably the minimum. If we can accommodate eight to ten, we open the way for more interesting visual effects. Furthermore, since shadow and environment maps may be recalculated as often as every frame, we must be able to either render directly into table memory, or load a map from the frame buffer very quickly. Address calculations and table access patterns should be flexible since, in our experience, it is difficult to predict what a programmer may wish to do.

2.3 Computational Power

The key to real-time shading is to combine the programmability and memory requirements above with the tremendous computational power needed to shade images in real time.

To get a feel for the magnitude of the calculations involved, consider simple Phong shading. To do this for a million pixels at 30 times per second, requires a billion or more operations per second. More sophisticated algorithms, such as bump mapping, shadow mapping, procedural textures, and antialiasing, can multiply these requirements by an order of magnitude or more.

Large-scale parallelism. The only way to achieve computation rates such as these is to employ large-scale parallelism. Current graphics engines employ dozens to hundreds [7] (or even thousands [8]) of processors to perform visibility and relatively simple shading. Even more are needed for programmable shading.

Fortunately, many features of general-purpose processors are not needed here, so processors can be specialized for rendering. For example, pixel-level processors can have tightly bound local memory, specialized datapaths and functional units, separate code stores, and may even share control and address paths (i.e. operate in SIMD). Such specialization can reduce the cost and size to a small fraction of that of a standard processor.

Even with specialization, a parallel processor for shading will be expensive because of the great computational demands. To make real-time shading practical, we must also reduce the workload as much as possible through optimization. We now consider several optimization techniques that can provide significant speedups.

Deferred Shading. One optimization is to shade only pixels that will be visible in the final image. Figure 3a illustrates the normal rendering pipeline. Shading is done as primitives are rasterized. If the pixel is visible at the current time, the pixel is shaded and a final color value is stored in the frame buffer. However, many pixels may be covered by later primitives, particularly in scenes with high depth complexity. The shading performed on non-visible pixels is wasted.

```
// Rasterization/shading pass
for each primitive
   for each pixel {
      calculate depth;
      if (pixel is visible) {
         shade;
         store color;
      }
   }
```

a) Immediate shading.

```
// Rasterization pass
for each primitive
   for each pixel {
      calculate depth;
      if (pixel is visible)
         store appear. params;
   }

// Deferred shading pass
for each pixel
   shade;
```

b) Deferred shading.

Figure 3: Two variations of the rendering pipeline.

This wasted work can be avoided by delaying shading calculations until after primitives have been rasterized, a technique known as deferred shading [9, 10]. Figure 3b illustrates a pipeline modified for deferred shading. The only work that is performed in the loop over primitives is to determine visibility and to store the raw data the shader will need to compute the pixel colors later. This data typically consists of constants, such as intrinsic colors, or interpolated parameters, such as surface normal vectors, texture coordinates, etc. (Cook refers to these as *appearance parameters* [3]. We will use this term in the remainder of the paper). A second pass of the algorithm loops over the pixels, shading each one.

Deferred shading divides the cost of shading by the depth complexity of the image. This can be substantial for complex scenes. Deferred shading constrains the rendering algorithm in a number of ways, however. It requires additional storage in the frame buffer for appearance parameters, which require more space than simply color and *z*. Also, shading cannot affect the visibility of objects, since visibility is completely determined before the shading pass.

Uniform vs. varying parameters. In the design of the RenderMan shading language, Hanrahan recognized that a potentially powerful optimization is to calculate expressions that are independent of position on a surface only once [4]. To take advantage of this, the language allows the programmer to specify whether a variable is *uniform*, its value is constant across a surface, or *varying*, its value depends on position. Expressions or subexpressions that consist of only uniform variables may be calculated once and, in a uniprocessor, cached away.

This optimization extends to MIMD parallel shading, only the potential savings are not as great. as for a uniprocessor. Since each processor shades only a fraction of the pixels, the calculation of uniform parameters cannot be amortized over as many pixels.

However, this optimization fits the SIMD paradigm quite well. The uniform expressions can be calculated on the control processor to generate a single, position-independent result that can be broadcast to all of the processing elements. Of course varying computations are local and must be performed in parallel across the processor array.

Fixed-point vs. floating-point arithmetic. In order to save Silicon area and cost, most of the pixel-level calculations in graphics workstations are carried out using integer arithmetic. In

contrast, most calculations in high-quality software renderers use floating point. Can we use fixed-point arithmetic for shading?

Most shading parameters, such as surface normals, light source direction vectors, ambient, diffuse, and specular coefficients, are numbers in the range from zero to one. We can analyze the numerical errors that may occur in a particular computation, such as that for Phong shading, in order to determine the necessary precision. For example, if we would like to obtain 12 bits of precision for color, the Phong lighting and shading computation will require:

- 2 bytes for intrinsic color
- 3 bytes for normals
- 2 bytes for the illumination model coefficients
- 3 bytes for intermediate colors

We can use four-byte integers for convenience as well as overflow protection during the calculations and still perform our computations an order of magnitude faster (or cheaper) than we could with floating-point arithmetic.

The problem with fixed-point integer arithmetic, of course, is that we cannot determine the necessary number of significant digits if we don't know *a priori* the magnitudes of the equation parameters. This is the case with global effects, such as shadow maps. In order to write generally usable and robust procedures, we may have to use floating-point arithmetic in some critical parts of shaders, such as matrix transformations. However, for speed on a hardware-supported shader, most shading calculations can be done in fixed-point arithmetic.

Optimizations such as these, combined with parallelism and fast processors, make it possible to build a system that can render high-quality images at interactive rates.

3 THE PIXELFLOW SHADING ARCHITECTURE

We are building a hardware and software system to demonstrate the feasibility of real-time programmable shading. In this section we describe the architecture of the system and show how it can meet our performance goals. We begin by briefly describing the architecture of PixelFlow, the hardware on which the system is built. We then explain the techniques that we use to achieve interactive programmable shading. Finally, we outline the programming models of the system: the existing low-level model, and a high-level language we are implementing that is similar to the RenderMan shading language [4].

We believe that this system, when the hardware is complete, will be able to render the types of images previously seen only on software renderers, at interactive rates.

3.1 Hardware Overview

PixelFlow consists of a set of nodes, each of which is essentially a complete graphics computer. All of the nodes are identical, although some have additional video input or output capability on daughter cards to allow them to act as frame grabbers or frame buffers. The PixelFlow nodes are connected by a linear network that provides fast dedicated pixel-level communication and built-in *z*-buffer compositing. General purpose communication between the PixelFlow nodes is provided by a message passing network. Figure 4 shows a block diagram of a PixelFlow system.

There are two types of computational resource on each PixelFlow node. A SIMD array of 128x64 (8,192) pixel processors and a pair of general-purpose RISC processors (GPs). The pixel processors perform most rasterization and shading calculations, while the

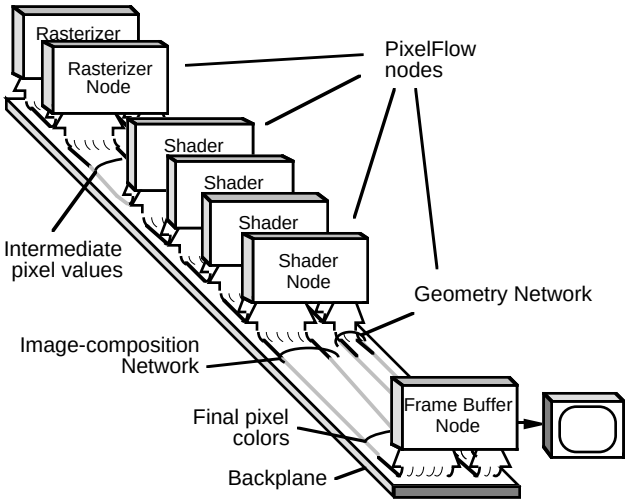

Figure 4: PixelFlow system block diagram.

GPs generate instructions for the SIMD array. These instructions are stored in GP main memory and are fetched by an instruction sequencer that controls the array. Figure 5 shows a block diagram of a PixelFlow node.

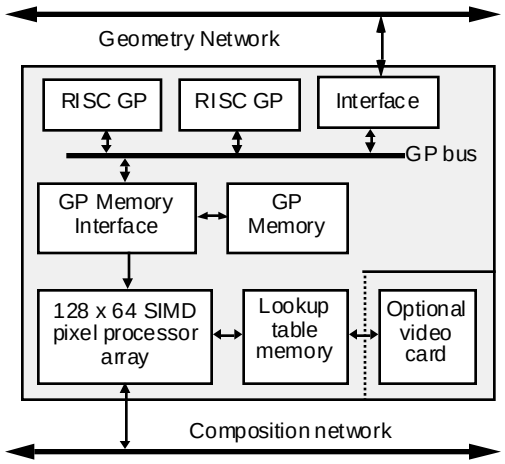

Figure 5: PixelFlow node block diagram

A SIMD architecture was chosen for the pixel processors to maximize the amount of compute power that could be placed on a node. The advantage of SIMD is that a single instruction sequencer, an expensive resource, is shared by a large number of processors. The individual pixel processing elements are simple and there is no direct pixel to pixel communication. This makes it possible to put 128x64 processors on one board.

Each pixel processor contains an 8-bit ALU which performs a standard set of integer instructions, such as addition, subtraction, multiplication, and shifts. Most of the instructions allow operand sizes to vary from one to eight bytes in length. Single-precision floating-point operations, based on the IEEE standard, are implemented as sequences of integer operations.

Fixed-point arithmetic. Earlier, we discussed the tradeoffs between fixed-point (integer) and floating-point arithmetic. Figure 6 shows the instruction execution times, per pixel processor, of integer vs. floating-point instructions. Even though up to 8K

processors execute these instructions at once, the lower execution times of some of the integer operations make them very attractive.

Operation	2-byte short	4-byte long	4-byte float
Addition	0.07 µs	0.13 µs	3.94 µs
Multiplication	0.50 µs	2.00 µs	2.53 µs
Division	1.60 µs	6.40 µs	7.04 µs
Square Root	1.22 µs	3.33 µs	6.98 µs

Figure 6: Execution time of integer versus floating-point instructions.

Conversion from floating point to 4-byte integer format takes 1.35µs, and from 4-byte integer to floating point takes 1.57µs. This makes it feasible to convert representations to use whichever is more advantageous. Whenever possible, we use fixed-point or integer representations.

Memory. Each processor has 256 bytes of local memory and 128 bytes of communication register that may also be used as local memory. Each node can store 16MB of texture information in table lookup memory. This memory may be read or written from each of the pixel processors, thus serving as global storage.

3.2 Achieving interactive shading

Each PixelFlow node possesses an enormous amount of computational power—over 40 billion integer operations or 2 billion floating-point operations per second. In addition, the processors are programmable in a very general way, and we believe that the 256 (+128) bytes of local memory at each processor is sufficient to implement many interesting shading algorithms. However, even this amount of computational power is not enough to achieve our goal of real-time shading. We must harness multiple PixelFlow nodes in an efficient manner to multiply the power available for shading.

PixelFlow rasterizes images using a screen-subdivision approach, sometimes called a *virtual buffer* [11]. The screen is divided into 128x64-pixel regions, and the regions are processed one at a time. When the rasterizers have finished with a particular region, they send appearance parameters and depth values for each pixel onto the image-composition network, where they are merged and loaded into a shader.

If there are *s* shaders, each shader receives one of every *s* regions. While it shades the region, it has full use of the local memory at each pixel processor. With this method of rendering, even a small machine can support an arbitrary sized screen. Of course, the more complex the problem, the more nodes that are needed to achieve interactive performance.

Deferred shading. As stated in Section 2.3, deferred shading is a powerful optimization for scenes of high depth complexity. It has an even bigger payoff for a SIMD architecture such as PixelFlow. We implement deferred shading on a machine-wide basis by giving each node a designated function: rasterization or shading. The rasterization nodes implement the first loop in Figure 3b, while the shading nodes implement the second.

As specified in Figure 3b, the rasterization nodes scan convert the geometric primitives in order to generate the necessary appearance parameters. Multiple rasterization nodes can work on a single region of the screen as described by Molnar, et. al. [12]. The composition network collects the rasterized pixels for a given region (including all necessary appearance parameters), and delivers it to the shading node that has been assigned to process that region.

Deferred shading provides an additional computational advantage on PixelFlow because of the SIMD nature of the pixel processors. Consider how a SIMD machine might behave if shading is performed during rasterization (immediate shading—Figure 3a). For each primitive, the processors representing the pixels within the primitive are *enabled*, while all of the others are *disabled.* The subsequent shading computations are performed only for the enabled pixels. The processors representing pixels outside of the primitive are disabled, so no useful work is performed.

Since most primitives cover only a small area of the screen, we would make very poor use of the processor array. The key to making effective use of the SIMD array is to have every processor do useful work as much of the time as possible.

With deferred shading, all of the pixels in a region that require the same shader can be shaded at one time, even if they came from different primitives. This is especially useful when tessellated surfaces are used as modeling primitives. These can be rasterized as numerous small polygons but shaded as a single unit. In fact, disjoint surfaces can be shaded at once if they use the same shading function.

Factoring out common calculations. We can go even further than executing shading functions only once per region. Shading functions tend to be fairly similar. Even at a coarse level, most shading functions at least execute the same code for the lights in the scene even if their other computations differ. All of this common code need only be done once for all of the pixels that require it. As illustrated in Figure 7, if each shading function is executed to the point where it is ready to do lighting computations, the lighting computations for all of them can be performed at once. The remainder of each shading function can then be executed in turn.

```
// Shader-specific code
for each surface shader
   pre-light shading;

// Common code
for each light source
   accumulate illumination;

// Shader-specific code
for each surface shader
   post-light shading;
```

Figure 7: Factoring out common operations for multiple shading functions.

Currently, we code this manually, but this is yet another reason to have a high-level compiler. A suitably intelligent compiler can identify expensive operations (such as lighting and texture lookups) among several shading functions and automatically schedule them for co-execution.

Table lookup memory. Each shader node has its own table-lookup memory for textures but, since it is not possible to know which textures may be needed in the regions assigned to a particular node, the table memory of each must contain every texture. For interactive use this not only limits the size of the textures to the maximum that can be stored at one node, but it also presents a problem for shadow map and environment map algorithms that may generate new textures every frame. After a new map is computed, it must be loaded into the table-lookup memories of every shader node. This aspect of system performance does not scale with the number of nodes: a maximum of 100 512x512

texture maps can be loaded into table-lookup memory per second (2-3 in a 33 ms frame time).

Uniform and varying expressions. For efficiency, expressions containing only uniform shader variables (those that are constant over all of the pixels being shaded) are computed only once on the RISC GP. Varying expressions (those that vary across the pixels), or those containing a mix of uniform and varying variables, are executed on the pixel-processor array.

Shader parameters. There are two ways to communicate parameters to a shader node. One is to send the parameters over the composition network. The other is to send the parameters over the front-end geometry network. Obviously, a varying parameter that must be interpolated over the pixels, such as color or surface normal, is produced on a rasterization node, and should be sent over the composition network.

A uniform parameter that is used at the GP and does not vary from primitive to primitive should be sent over the geometry network because composition network bandwidth is a valuable resource. An example is something like the roughness of a surface which is a fixed parameter for a particular material. If the parameter is needed in the local memory of the pixel processors, it can be broadcast locally at a shading node. We allow the programmer to choose the best way to transmit each parameter.

3.3 Shader programming model

Low-level model. Since instructions for the pixel processors are generated by the GP on a PixelFlow node, the code that a user writes is actually C or C++ code that executes on the GP. The low-level programming model for the pixel processors (called *IGCStream*) consists of inline functions in C++ that generate code for the SIMD array. Some of these functions generate the basic integer operations; others, however, generate sequences of instructions to perform higher-level commands, such as floating-point arithmetic.

We have written a library of auxiliary shading functions to use with this programming model. It provides basic vector operations, functions to support procedural texturing [5, 13], basic lighting functions, image-based texture mapping [14], bump mapping [15], and higher-level procedures for generating and using reflection maps [16] and shadow maps [17, 18]. It is perfectly feasible to program at this level. In fact, we currently use this programming model to write code for testing, and to produce images such as those in the example video. We would prefer, however, to work at a higher, more abstract level.

High-level model. We are implementing a version of the RenderMan shading language that is modified to suit our needs. Our goal in using a higher-level language is not solely to provide architecture independence. That may be useful to us in the future, of course, but since PixelFlow is an architectural prototype it is not necessary. We are more interested in the shading language as a way to demonstrate feasibility and to provide our users with a higher-level interface that they've had [19] in order to encourage wide use of the shading capabilities of our system. Also, as mentioned earlier in this section, a high-level shading language provides opportunities for compiler optimization, such as co-executing portions of several shader functions.

The RenderMan specification has only float, point, and color arithmetic data types. Since we need to be frugal in our use of floating-point arithmetic, we have added integers and fixed-radix-point numbers to the data types of our language. A compiler for the shading language will accept shader code as input, and emit C++ with SIMD processor commands as output. This code will be linked with the auxiliary shading function library and finally with the application program.

API support. We also need some way for graphics applications to access our shading capability. Since one of our main goals for PixelFlow is interactive visualization of computations as they are executing on a supercomputer, we have chosen an immediate-mode application programmer's interface (API) similar to OpenGL [20]. An advantage of choosing OpenGL, and extending it to meet our needs is that students and collaborators are likely to be familiar with the its basic concepts Also, this will make it easier to port software between PixelFlow and other machines.

The current specification of OpenGL only incorporates the limited set of shading models commonly found on current graphics workstations: flat and linearly interpolated shading with image-based textures. We have extended the specification to allow users to select arbitrary shaders.

We do not plan to implement an official, complete OpenGL for two reasons. One is that some of the specifications of OpenGL conflict with our parallel model of generating graphics. The second is that we lack the resources to implement features that we do not use. Consequently, though our functions are similar to OpenGL, we use a *pxgl* prefix instead of OpenGL's *gl* prefix. Within these constraints, we have attempted to stick as closely as possible to the OpenGL philosophy. We intend to describe this API, and the problems involved in implementing it on PixelFlow, in a future publication.

Limitations. Although the PixelFlow shading architecture supports most of the techniques common in "photorealistic rendering," (at least in RenderMan's use of the term), it has a few limitations. Because PixelFlow uses deferred shading, shaders normally do not affect visibility. Special shaders can be defined that run at rasterization time to compute opacity values. However, these shaders poorly utilize the SIMD array and slow rasterization.

A second limitation is that shaders cannot affect geometry. RenderMan, for example, defines a type of shader called a *displacement shader*, which displaces the actual surface of a primitive, rather than simply manipulating its surface-normal vector, as is done in bump mapping. This is incompatible with the rendering pipeline in PixelFlow, as well as that of virtually all other high-performance graphics systems.

4 EXAMPLE

In this section, we present a detailed example of real-time high-quality shading on PixelFlow. The example—bowling pins being scattered by a bowling ball—was inspired by the well-known "Textbook Strike" cover image of the *RenderMan Companion* [6]. We cannot guarantee that the dynamics of motion are computable in real-time, but we are confident that a modest-sized PixelFlow system (less than one card cage) can render the images at 30 frames per second.

The accompanying video was rendered on the PixelFlow functional simulator. The execution times are estimates based on the times of rasterization and shading of regions, using worst-case assumptions about overlap. We simulated a PixelFlow machine containing three rasterizer nodes, twelve shading nodes, and a frame-buffer node. There are 10,700 triangles in the model. The images were rendered at a resolution of 640x512 pixels with five-sample-per-pixel antialiasing.

4.1 Shading functions

Three shading functions are used to render these images, one for the bowling pins, one for the alley, and one for the bowling ball. Two light sources illuminate the scene, an ambient light and the main point-light source which casts shadows in the environment.

Parameter	Number of bytes
Depth	4
Shader ID	1
Intrinsic Color	1 x 3
Normals	2 x 3
Texture coordinates, u, v	2 x 2
Texture gradients	2 x 2
dP/du, dP/dv	2 x 6

Figure 8: Appearance parameters used in bowling example.

Figure 8 shows the data for each pixel that is sent from a PixelFlow rasterizer node to a shader node, a total of 34 bytes. We actually plan to use 10 bits of color per channel on most PixelFlow applications, but 8 bits were used for this simulation. In addition to the appearance parameters used by the shaders, two other parameters are necessary, the depth and a shader identification number for each pixel. The shader ID is used by the shading control program to select the shader code for each pixel.

The bowling ball has a shadow-mapped light source with a Phong shader. The alley has a shadow-mapped light source, reflection map, mip-mapped wood texture, and a Phong shader. The pins have a shadow-mapped light source, procedural crown texture, mip-mapped label, bump-mapped scuffs, mip-mapped dirt, and finally a simple Phong shader. We factor out common lighting computations as described in Section 3.2. Each shader is divided into three parts, the part before the lighting computation, the common lighting computation, and the part after the lighting computation.

4.2 Multiple-pass rendering

The shadow and reflection maps are obtained during separate rendering passes. When each of these 512x512 images has been computed and stored, rendering of the final image begins. In this section we describe, in detail, the steps necessary to render and store the shadow map and to render the final camera-view image. Since computation of the reflection map is similar, we do not describe it in detail.

Shadow map. A shadow map is a set of depth values rendered from the point of view of the light source. We use three rasterizer nodes to rasterize all the primitives and compute the depth at each sample point. Since we do not need to calculate colors or other parameters, this is a simple computation. The worst-case time for this step is approximately 100 µs, although many map regions have very few polygons and take less time to rasterize.

The depth values are then z-composited over the composition network, and the resulting depth is sent to all of the shaders. Composition time is only 5 µs per region. Notice that data transfer and computation can proceed simultaneously.

As mentioned in Section 3.2, storing tables for shadow or reflection mapping is a point of serialization on our system. The combined time to store both the shadow and reflection map takes almost half the time for each frame. Since the hardware can store four values into table memory at one time, we take advantage of this intra-node parallelism by storing the depth map in units of four regions each. Thus, the shader nodes accept four regions of data before storing them (see Figure 11).

The total time to complete the shadow map pass is the time consumed by eight table writes, 6.08 ms, plus the time to rasterize the first four regions, for a total time of less than 7 ms.

Reflection Map. Rasterization for the reflection map can begin as soon as enough buffer space is available at the rasterization nodes. Shading for the reflection map can begin as soon as the last table write for the shadow map has begun. The reflection map can be generated and stored in less than 7 ms.

Final Image. The rendering time for the final image is a function of both the rasterization time and the shading time. If the time to rasterize a region is longer than the time to shade it, the shading nodes will be idle waiting for appearance parameters from the rasterizer nodes. The worst-case time will then be the total rasterization time plus the time to shade the final region. If the time to rasterize a region is less than the time to shade it, the shading nodes will always have regions waiting to be shaded. We will see that for this scene shading is the bottleneck, so the rendering time will be the total shading time plus the time to rasterize the first few regions (to get the shading nodes started).

First, consider the rasterization time. With all of the appearance parameters, each of the front-facing triangles in the model takes approximately 0.85 µs to rasterize. One of the busiest frames, with all of the pins visible, contains just under 6400 front-facing triangles (this includes the additional triangles that have to be rendered when triangles cross region boundaries). This total takes 5.4 ms to complete on one rasterizer node. If we also do five sample antialiasing, this becomes 27 ms. To achieve our performance goal we divide the polygons over 3 rasterizers to decrease the time to a little over 9 ms. Details on the use of multiple rasterizers in PixelFlow can be found in [12].

Section of code		Shading function		
		Pin	Alley	Ball
pre-light	procedural crown	2 µs		
	mip-map label	15 µs		
	bump-map scuffs	24 µs		
	mip-map dirt	15 µs		
	mip-map wood		15 µs	
light	shadowing	← 28 µs →		
post-light	Phong shader	12 µs		
	reflection		27 µs	
	Phong shader			12 µs

Figure 9: Shading times (1 node, 1 sample, 1 region) excluding table lookup.

Now, consider the shading time. In PixelFlow, the table lookup time is proportional to the number of pixels that need data, so it is not constant for a region but depends on how many total values are actually needed. The worst case for table lookup will occur if all of the pixels in a region use the bowling pin shading function since it needs to look up four different values: two mip-mapped image textures, one bump map, and one shadow map. To do one table lookup for all 8K pixels on a node takes 190 µs, so looking up four values for a full region requires 760 µs.

The worst-case time for the rest of the shader processing occurs for regions that require all three shading functions, bowling pin, alley, and ball. For regions without all of these elements, only

some of the shading functions need to be run. Figure 11 shows the processing time for the shading functions excluding the table lookup times. Note, however, that the time setting up for a lookup and using the results *is* included. The slowest time for a region is the sum of all the times in the figure or 150 µs.

This time is for only one sample of one region. Since we are doing five samples and a 640x512 video image has 40 regions, there are really 200 regions to shade. The total time comes to 182 ms. By distributing the shading among twelve shading nodes, we can cut the worst-case shading time to about 15.2 ms.

The 9 ms spent rasterizing is less than the shading time. Therefore, the shading time dominates. The total time to compute the final camera view is the shading time plus the time to rasterize the first regions, or about 15.7 ms.

Total frame time. A complete image can be rendered in under 29.7 ms. This includes 7 ms to generate a shadow map, 7 ms to generate a reflection map, and 15.7 ms for the final camera image. These times were computed with pessimistic assumptions and without considering the pipelining that occurs between the rendering phases. This results in a frame rate faster than 30 Hz. With more hardware it will be possible to run even faster.

Additional hardware will not significantly speed the shadow or reflection map computations since they are dominated by the serial time spent writing the lookup tables. But rendering time of the camera image is inversely proportional to the number of rasterization and shading nodes. For more complex geometry, we add rasterization nodes. For more complex shading, we add shading nodes. Note that the hardware for both of these tasks is identical. The balance between them can be decided at run time.

5 CONCLUSION

In this paper, we described the resources required to achieve real-time programmable shading—programmability, memory, and computational power—requirements that many graphics hardware systems are close to meeting. We explained how this shading power can be realized in our experimental graphics system, PixelFlow. And we showed with an example, simulations, and timing analysis that a modest size PixelFlow system will be able to run programmable shaders at video rates. We have demonstrated that it is now possible to perform, in real time, complex programmable shading that was previously only possible in software renderers. We hope that programmable shading will become a common feature in future commercial systems.

ACKNOWLEDGMENTS

We would like to acknowledge the help of Lawrence Kesteloot and Fredrik Fatemi for the bowling simulation dynamics, Krish Ponamgi for the PixelFlow simulator, Jon Leech for his work on the PixelFlow API design, Nick England for his comments on the paper, and Tony Apodaca of Pixar for RenderMan help and advice. Thanks to Hewlett-Packard for their generous donations of equipment.

This research is supported in part by the Advanced Research Projects Agency, ARPA ISTO Order No. A410 and the National Science Foundation, Grant No. MIP-9306208.

REFERENCES

[1] Cook, R. L., L. Carpenter and E. Catmull, "The Reyes Image Rendering Architecture", *SIGGRAPH 87*, pp. 95-102.

[2] Whitted T., and D. M. Weimer, " A Software Testbed for the Development of 3D Raster Graphics Systems", *ACM Transactions on Graphics*, Vol. 1, No. 1, Jan. 1982, pp. 43-58.

[3] Cook, R. L., "Shade Trees", *SIGGRAPH 84*, pp. 223-231.

[4] Hanrahan, P. and J. Lawson, "A Language for Shading and Lighting Calculations", *SIGGRAPH 90*, pp. 289-298.

[5] Perlin, K., "An Image Synthesizer", *SIGGRAPH 85*, pp. 287-296.

[6] Upstill, S., *The RenderMan Companion*, Addison-Wesley, 1990.

[7] Akeley, K., "Reality Engine Graphics", *SIGGRAPH 93*, pp. 109-116.

[8] Fuchs H., J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories", *SIGGRAPH 89*, pp. 79-88.

[9] Deering, M., S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics", *SIGGRAPH 88*, pp. 21-30.

[10] Tebbs, B., U. Neumann, J. Eyles, G. Turk, and D. Ellsworth, "Parallel Architectures and Algorithms for Real-Time Synthesis of High Quality Images using Deferred Shading", UNC CS Technical Report TR92-034.

[11] Gharachorloo N., S. Gupta, R. F. Sproull and I. E. Sutherland, "A Characterization of Ten Rasterization Techniques", *SIGGRAPH 89*, pp. 355-368.

[12] Molnar S., J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition", *SIGGRAPH 92*, pp. 231-240.

[13] Gardner G. Y., "Visual Simulation of Clouds", *SIGGRAPH 85*, pp. 297-303.

[14] Williams L., "Pyramidal Parametrics", *SIGGRAPH 83*, pp. 1-11.

[15] Blinn, J. F., "Simulation of Wrinkled Surfaces", *SIGGRAPH 78*, pp. 286-292.

[16] Greene N., "Environment Mapping and Other Applications of World Projections", *IEEE CG&A*, Vol. 6, No. 11, Nov, 1986, pp. 21 - 29.

[17] Williams L, "Casting Curved Shadows on Curved Surfaces", *SIGGRAPH 78*, pp. 270-274.

[18] Reeves W. T., D. H. Salesin, and R. L. Cook, "Rendering Antialiased Shadows With Depth Maps", *SIGGRAPH 87*, pp. 283-291.

[19] Rhoades, J., G. Turk, A. Bell, A. State, U. Neumann, and A. Varshney, "Real-Time Procedural Texture", *Proc. 1992 Symp. on 3D Interactive Graphics*, pp. 95-100.

[20] Akeley K., Smith K. P., Neider J., *OpenGL Reference Manual*, Addison-Wesley, 1992.