Chapter 1: Introduction

Marc Olano, UMBC

1 Course Background

This is the seventh offering of a graphics hardware-based shading course at SIGGRAPH, and the field has changed and matured enormously in that time span. The first course, "Procedural Shading on Graphics Hardware" in 2000 focused on a handful of research projects that had emerged showing procedural shading actually could be accomplished on graphics hardware (if you tried really, really hard). The focus was on how you could possibly get those inflexible graphics machines to do something as flexible as procedural shading.

Since that early beginning, real-time procedural shading hardware and software has blossomed. This year's course focuses much more on current shading and rendering approaches on the GPU, both real-time and non-real-time.

The remainder of this chapter provides some basic shading background. If you are a long-time shading user, skip ahead. If you're just getting started and wonder what all this shading stuff is about, read on.

2 Shading Background

Procedural shading is a proven rendering technique in which a short user-written procedure, called a *shader*, determines the shading and color variations across each surface. This gives great flexibility and control over the surface appearance.

The widest use of procedural shading is for production animation, where has been effectively used for years in commercials and feature films. These animations are rendered in software, taking from seconds to hours per frame. The resulting frames are typically replayed at 24-30 frames per second. Since the frames are stored and played back later, rendering frame rate is totally decoupled from playback frame rate, instead being dictated by a combination of production schedule, maximum allowable preview wait time, and budget for building a large render farm.

One important factor in procedural shading is the use of a shading language. A shading language is a high-level special-purpose language for writing shaders. The shading language provides a simple interface for the user to write new shaders. Pixar's RenderMan shading language [Upstill90] is the most popular, and several off-line renderers use it. A

shader written in the RenderMan shading language can be used with any of these renderers.

Meanwhile, polygon-per-second performance has been the major focus for most *interactive* graphics hardware development. Only in the last few years has attention been given to surface shading quality for interactive graphics. Recently, great progress has been made on two fronts toward achieving real-time procedural shading. This course will cover progress on both. First, graphics hardware is capable of performing more of the computations necessary for shading. Second, new languages and *machine abstractions* have been developed that are better adapted for real-time use.

To support general procedural shading, a system must support the following:

1. Texture or table lookup
2. Arithmetic operations sufficient to implement all functions in a standard math library
3. Types with sufficient range and precision for shading computations (preferably floating point)
4. Flow control (at least looping, preferably also branching and function calls)

The first has been common at the fragment level for a couple of decades, but is only just appearing at the vertex level. Graphics hardware has had the second (in the guise of texture lookups and flexible blending) for a while, but only in the past 5-6 years has the interface to it been refined to treat them as generic arithmetic operations. The third is finally becoming an standard hardware feature after years of first fixed-point computation, then reduced-precision floating point. The last has been possible for years through multi-pass rendering with the application able to decide how many passes is sufficient for the required loop iterations, but has only become possible in fragment shading in the most recent generation or two of hardware. The recent NVIDIA gelato™ to accelerate production film rendering provides a concrete demonstration that we are finally reaching the point where graphics shading hardware has the same basic capabilities as CPU-based shading.

Interactive graphics machines themselves are complex systems with relatively limited lifetimes. The RenderMan shading language insulates the shading writer from the implementation details of the off-line renderer. A RenderMan shader writer does not know or care if the renderer uses the REYES algorithm, ray tracing, radiosity, or some other rendering algorithm. In the same way, a real-time shading system presents a simplified view of the interactive graphics hardware. This is done in two ways. First, we create an abstract model of the hardware.

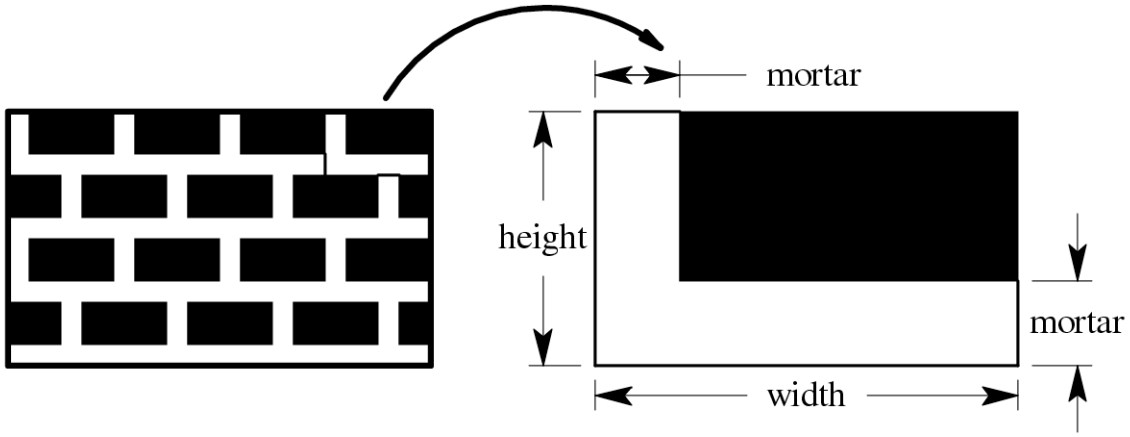

Figure 1. Size and shape parameters for brick shader.

This abstract model gives the user a consistent high-level view of the graphics process that can be mapped onto the machine. Second, a special-purpose language allows a high-level description of each procedure. Given current hardware limitations, languages for real-time shading differ quite a bit from the one presented by RenderMan. Through these two, we can achieve *device-independence*, so procedures written for one graphics machine have the potential to work on other machines or other generations of the same machine.

In the first incarnation of this course at SIGGRAPH 2000, there were as many single-platform shading languages as there were presenters, each with the same spirit, but incompatible in syntax. Now we have a selection of cross-platform languages. Where the choice of language used to be based on which hardware you were using, now it is based more on which graphics API and language syntax you prefer.

3 Procedural Techniques

Procedural techniques have been used in all facets of computer graphics, but most commonly for surface shading. As mentioned above, the job of a surface shading procedure is to choose a color for each pixel on a surface, incorporating any variations in color of the surface itself and the effects of lights that shine on the surface. A simple example may help clarify this.

We will show a shader that might be used for a brick wall (Figure 3). The wall is to be described as a single polygon with *texture coordinates*. These texture coordinates are not going to be used for image texturing: they are just a pair of numbers that parameterize the position on the surface.

The shader requires several additional parameters to describe the size, shape and color of the brick. These are the width and height of the brick, the width of the mortar between bricks, and the colors for the mortar and

```
// shader constants
// could be passed in to allow modification
float width=.25, height = .1, mortar = .01;
vec4 brick_color = vec4(1.,0.,0.,1.);
vec4 mortar_color = vec4(.5,.5,.5,1.);

void main() {
   // find row and column for this pixel
   float col = gl_TexCoord[0].x, row = gl_TexCoord[0].y;

   // offset even rows by half a row
   if (mod(row,2.*height)<height) col += width/2.;


   // wrap texture coordinates to get "brick coordinates"
   col = mod(col,width); row = mod(row,height);

   // pick color for this pixel, brick or mortar
   if (row < mortar || col < mortar)
     gl_FragColor = mortar_color;
   else
     gl_FragColor = brick_color;
}
```

Figure 2. Code for a simple brick shader.

brick (see Figure 1). These parameters are used to fold the texture coordinates into *brick coordinates* for each brick. These are (0,0) at one corner of each brick, and can be used to easily tell whether to use brick or mortar color. A portion of the brick shader is shown in Figure 2. In this figure, `row` and `col` are local variables used to construct the brick coordinates. The simple bricks that result are shown in Figure 3a.

One of the real advantages of procedural shading is the ease with which shaders can be altered to produce the desired results. Figure 3 shows a series of changes from the simple brick shader to a much more realistic brick. Several of these changes demonstrate one of the most common

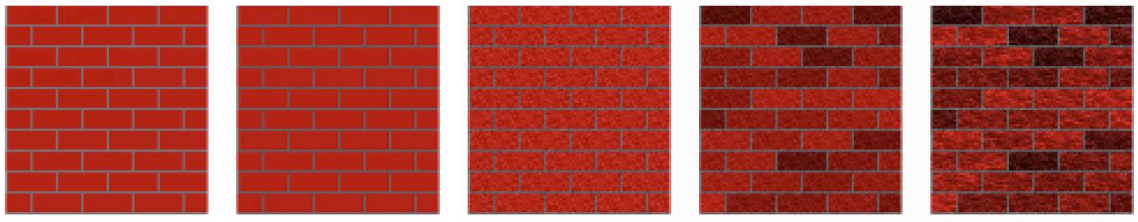

Figure 3. Evolution of a brick shader. a) simple version. b) with indented mortar computed bump map. c) with added graininess. d) with variations in color from brick to brick. e) with color variations within each brick.

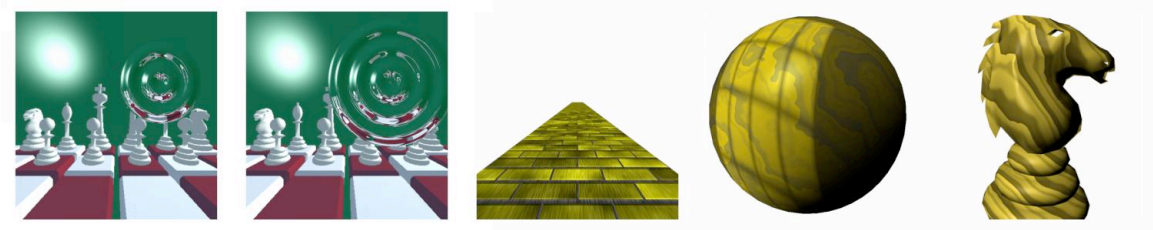

Figure 4. Examples of shaders. a+b) two frames of rippling mirror. c) yellow brick road. d+e) wood volume shader.

features of procedural shaders: controlled randomness. With controlled use of random elements in the procedure, this same shader can be used for large or small walls without any two bricks looking the same. In contrast, an image texture would have to be re-rendered, re-scanned, or re-painted to handle a larger wall than originally intended.

Procedural shading can also be used to create shaders that change with time or distance. Figure 4a and b are frames from a rippling mirror animated shader. Figure 4c shows a yellow brick road where high-frequency elements fade out with distance. **Error! Reference source not found.**d and e show a wood shader that uses surface position instead of texture coordinates. Figure 4d is also lit by a procedural light, simulating light shining through a paned window.

4 Shading for Interactive Rendering

The shaders in Figure 4 were written for interactive rendering on the PixelFlow graphics system [Olano and Lastra 1998]. This system had somewhat different performance characteristics than current shading hardware. Specifically, texture lookups on PixelFlow had a high latency (the time between when you started the lookup and when you absolutely had to know the result). This was reasonable if only a few textures were used in each shader, but made it generally preferable to do shading computations as explicit computations rather than many texture lookups. Even without that performance difference, shaders written for offline use (large RenderMan shaders for example), tend to include a fairly high ratio of computation to texture lookups. While textures may still play a large part in computing the shaded appearance, a computation-based shader is much more flexible than one that is more strongly texture-based. That flexibility translates into shaders that are useful in more situations without needing to be rewritten, and fewer design cycles trying to get the shader appearance just right. In contrast to both of these, today's shading hardware (at least the fragment shading hardware responsible for per-pixel computation) encourages the use of textures, including storing partial computations into textures, over raw computation alone. This has had a great impact on the way we write shaders for real-time use, and has created a whole area of graphics research on how to cast

different problems into a form requiring only combinations of functions of two variables that can easily be stored in a texture.

One of the great promises of real-time shading is the potential to have a single shading program that can run across a wide range of graphics hardware. While we don't yet have a single cross-platform shading language to satisfy everyone, there is ample evidence that it *is* possible. In this chapter, we discuss what is necessary to create a cross-platform shading language, how shading languages allow us to ignore hardware differences, what range of hardware can reasonably be represented by a single shading language, and what evidence exists now that it will really work.

5 Cross-Platform Shading

The key to a cross-platform shading language is to work with a common model of shading hardware rather than specifics of the hardware itself. The model is a mental picture of what's going on that shader-writers use to make sense of the code they write. The further you get into hardware specifics, the less general your model becomes.

Designing a good model for shading is the balance of three competing goals. The model should be simple enough for novice users to understand. It should be a good model of the problem domain, accurately describing what the shader is trying to do rather than exactly how to do it. This will allow the shading language compiler to map the shader onto the hardware in the way that is best for the specific hardware platform. Of course, it should also be possible to map it efficiently onto all the desired platforms.

The second goal is particularly important — the purpose of shading code (or any code) is to describe **what** you want done. The compiler can and will made different choices about **how** (within limits — it can't change the algorithms you use, but it can rearrange the execution order, unroll loops, decide if a certain operation should be computed or looked up in a texture, etc.).

5.1 Single Program, Multiple Data

Shading is inherently a very parallel task. Whether we are talking about vertices in an object, a surface diced into micropolygons, ray-traced intersection points, or screen pixels, there is always some relatively common set of operations being applied to a set of samples on the surface. It is this parallel nature that makes shading so approachable by hardware and allows us to even consider real-time shading.

The model that almost every shading system adopts is Single Program/Multiple Data (SPMD), with no processor-to-processor communication. You write a shader to describe what happens to a single

sample on the surface (single program). That same single program is run at every sample on the surface (multiple data). SPMD is closely related to the Single Instruction/Multiple Data (SIMD) model of parallel computation, but SIMD implies more about how the program will be executed. With SIMD, a set of parallel processors will runs the same set of instructions in lockstep, but with different data at each processor. SPMD runs the same program, but without any implication of whether the same path through the program must be taken by all processors. On a pure SIMD array of processors, conditional code is handled by disabling a subset of the processors, who must wait while the others process the conditional instructions. Contrast this with the Multiple Instruction/Multiple Data (MIMD) model, where every processor can be running a completely different program.

SPMD is sometimes referred to as “SIMD on MIMD” or “effective SIMD”, as it is uses a SIMD style of programming, but can include programs to run on a single processor, MIMD machine or SIMD machine.

5.2 No communication

One of the aspects of shading that has allowed the explosion of fast shading hardware is the independence of each shading sample from every other shading sample. One of the most difficult and expensive aspect of general-purpose parallel machines is the communication network allowing the processors or nodes to communicate with each other. If the need for this communication is removed, the need for synchronization between the processors disappears, as does the need for physical connections between processors. The processors can be packed much more densely and are free to execute on batches of samples, samples in a pipeline, samples one at a time — whatever is most efficient.

Communication costs are also generally so high relative to computational costs, and so dependent on the machine architecture, that introducing processor to processor communication into a SPMD model greatly reduces the kinds of hardware a program can use effectively. The longer we can avoid communication, the more general our shaders will be.

Shaders don’t need sample to sample communication because shaders are typically restricted to computing only local lighting models. Anything that makes the appearance of one point on the surface depend on points elsewhere on the surface introduces the need a sample to sample communication. Shadows, global illumination and subsurface scattering are all on the list of effects that break the model to some degree if they are allowed in a shader.

General purpose computations, on the other hand, often require significant processor to processor communication. As graphics hardware

becomes more powerful and flexible, there is an increasing desire to use it for other general purpose parallel computation. This comes at a cost in flexibility of the resulting code. I would argue that we need **two** computational models. A model including communication for general purpose computation on NVIDIA and ATI-style hardware, and a model for shading (possibly targeting the general model on hardware that supports it) that is task oriented and unifies vertex and fragment computations.

In the mean time, many people have succeeded in creating general purpose algorithms on GPUs with inter-processor communication. They achieve this feat through the use of multiple passes. On one pass, you write data into textures or buffers in the graphics card. On the following pass, **any** processor can read **any** data from this texture, not just its own. Even if you are willing to accept multiple passes through the hardware, this communication method isn't perfect for all uses. The reader decides what other processor's data to read, and can read at most a handful of data per pass. Some computational algorithms map well to this model, while others would prefer to have the *writer* decide where the data should go. All of that will be covered in more depth later – for now, we'll restrict the discussion to shading.

5.3 Languages for hardware abstraction

One of the best examples of a shading language for hardware abstraction is the RenderMan shading language. Shaders written in this language have been successfully targeted to a huge range of different hardware. Pixar's PhotoRealistic RenderMan targets a single processor running each step of the shader in a loop over the micropolygons in a diced-up surface as generated by the REYES algorithm [Cook 1987]. BMRT also targeted a single processor, but as a ray-tracer it ran each shader in its entirety on one ray-intersection sample before moving on to the next sample [Gritz and Hahn 1996]. SGI created a RenderMan implementation targeting multiple rendering passes on graphics hardware, assuming hardware with a fast render-to-texture/read-from-texture or copy framebuffer-to-famebuffer [Peercy et al. 2000]. ATI has created a RenderMan language compiler targeting current shading hardware as part of their ASHLI toolkit.

RenderMan may not turn out to be the best language for hardware shading, but it has done an admirable job at being adaptable to a wide range of hardware. In the model presented by RenderMan, the shader writer tags data as being either *uniform* or *varying*. Uniform data is the

same across a set of samples being shaded at once[1]. Varying data may change from sample to sample.

For compilation of RenderMan shaders, the most important uniform and varying designations are for the inputs to the shader. The shading compiler must derive for itself which intermediate results within an expression are uniform and which are varying. Expressions using only uniform arguments will have uniform results; expressions with any varying arguments will have a varying result. The compiler can use similar logic to decide whether any local variable in a shader is really uniform or varying regardless of how it was specified in the shading code.

Given an accurate idea of exactly which quantities vary across the shaded surface and which don't, the shading compiler can make several choices for actual execution. It can decide to still evaluate every computation at every sample (not using the uniform/varying distinction). It can evaluate the uniform computations once for a set of samples and for each varying computation, loop over the samples to evaluate it. It can execute the varying computations as SIMD instructions across a parallel array of processors. It can execute the entire shader or just the varying computations across a set of MIMD processors. It can create a pipeline of stream processors, each executing one or a few varying instructions on one sample before passing that sample on to the next.

5.4 Where should we break the portability?

There are several facets of the RenderMan shading language that are not well suited for graphics hardware. We can expect several of these to be the foundation of differences between real-time languages and the RenderMan shading language, or limitations of hardware-accelerated RenderMan implementations.

Since PRMan version 3.8, the RenderMan shading language has included the ability to call arbitrary code from within a shader. This code can do anything, from compute a specialized noise function to spawn a different style of renderer to download an image from a live camera on the south pole. Until graphics hardware has the ability to run arbitrary code, this won't really be an option for real-time shading.

RenderMan's has just one scalar data type, *float*. Graphics hardware supporting floating point data is now ubiquitous, but the size and precision of the floating point numbers vary. Fixed point or reduced-precision floating point numbers are also provided on some hardware as a faster option than pure 32-bit floating point. With no way to indicate

[1] One RenderMan trick that will tell you something about how many samples are shaded at once, breaking the illusion that all hardware is the same, is to assign a random color into a uniform variable in a RenderMan shader.

the range or precision of computations, a RenderMan compiler cannot know when to use these faster operations. Many of the candidates for a real-time shading language include some reduced precision types for efficiency: the OpenGL Shading Language [Kessenich et al. 2003], Direct3D HLSL [Microsoft 2002], and Cg [Mark et al. 2003].

RenderMan shaders have two computational frequencies (how often a computation happens), uniform and varying. Shading hardware has at least three — compute on the CPU, compute per vertex and compute per fragment, with no interleaving of computation between the levels. All of the languages mentioned above have chosen to break shading computation into separate procedures executing at each of these levels. That choice makes those shaders a poor fit to any hardware or software rendering system that does not follow the CPU/Vertex/Fragment breakdown. However, any alternative language that targeted all three stages must include new types for the new types of computation [Proudfoot et al. 2001].

The RenderMan shading language also includes no real means of communication from sample to sample. This is one of the strengths that allow RenderMan shaders to run on such a wide range of rendering systems, but is a serious restriction for the general computations that are becoming popular on graphics hardware. Communication between processors in current hardware seems best supported by rendering partial results to a texture then using the random access provided by the texturing hardware to find values from other processors in a later pass.

This form of communication is currently limited to fragment shaders and comes at a very high price of communication to instructions. Similar communication at the vertex shader level is possible, though considerably more complicated. The all but the final vertex shader pass can operate on a regular grid of vertices, allowing all vertex and fragment operations to be used (including any vertex and fragment texturing and rendering to texture for communication). In the next-to-last vertex shader pass, the vertex locations (or data necessary to do the final computation) can be rendered in to a vertex array for use in a final vertex shader pass. If multiple passes of fragment shading are needed, they must follow after all vertex shading passes, but need not repeat the multi-pass vertex shading computation.

Obviously, stretching the hardware beyond its intended limits like this introduces a significant burden on the shader developer! Because users want to write algorithms that use communication, better facilities will appear in real-time shading languages, but as they do they will limit the applicability of those shaders to the class of similar hardware.

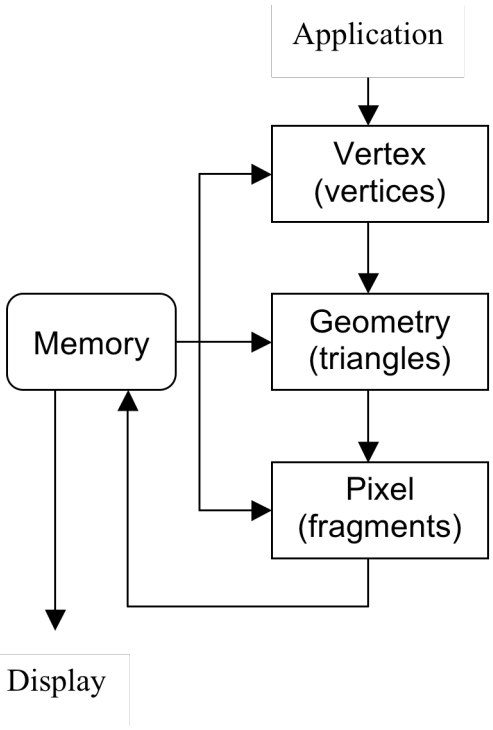

Figure 5. Hardware Shading Model.

5.5 The Vertex/Geometry/Fragment stream model

All current shading hardware presents a model in which three different types of shaders operate on a stream of primitives (Figure 5). The hardware and driver software promise to translate whatever code you give fitting this model into something the hardware can execute.

The *vertex shader* describes what operations should happen to a single vertex. This same shader is applied to each vertex in the stream of vertices (with several vertices being processed in parallel), producing a stream of transformed vertices. Then the hardware groups these vertices into triangles for the *geometry shader* (when supported, see Chapter 2).

The geometry shader describes what happens to a single triangle. Once again, this single function is applied to every triangle in a stream of triangles, except this time each triangle can add a somewhat arbitrary number of processed triangles to its output stream. None the less, the output is a new stream of triangles, and the hardware once again silently handles the conversion of a stream of triangles into a stream of fragments or pixels (OpenGL likes to call them fragments when in process, and reserve the word *pixel* for the single values stored in a frame buffer after z-buffering, antialiasing, etc. have been done).

The *fragment shader* or *pixel shader* describes what happens to a single fragment/pixel, computing the final color for display. There are typically many times more fragments than either vertices or triangles, so

Fragment shading is the most computationally expensive and uses the greatest degree of parallel execution.

All of the shader stages have access to texture memory, and the final stage writes to memory (either texture or framebuffer) for later use or display.

This model has seen some evolution over the past several years, but that evolution has consisted primarily of adding programmable stages. improvements in the ability to read or write memory, and generalization of the memory from very special purpose separation of vertex buffers, texture and frame buffer memory to a much more unified view. However, each of these changes improved or generalized the previous model, but did not break it. Thus, we should expect that shaders written to todays models will continue to work into the future.