Chapter 1. Introduction

Marc Olano

Course Background

This is the sixth 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.

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.

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 4-5 years has the interface to it been refined to treat them as generic arithmetic operations. The third has only really been available for the past couple of years. 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 hardware. The recent introduction of 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. 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

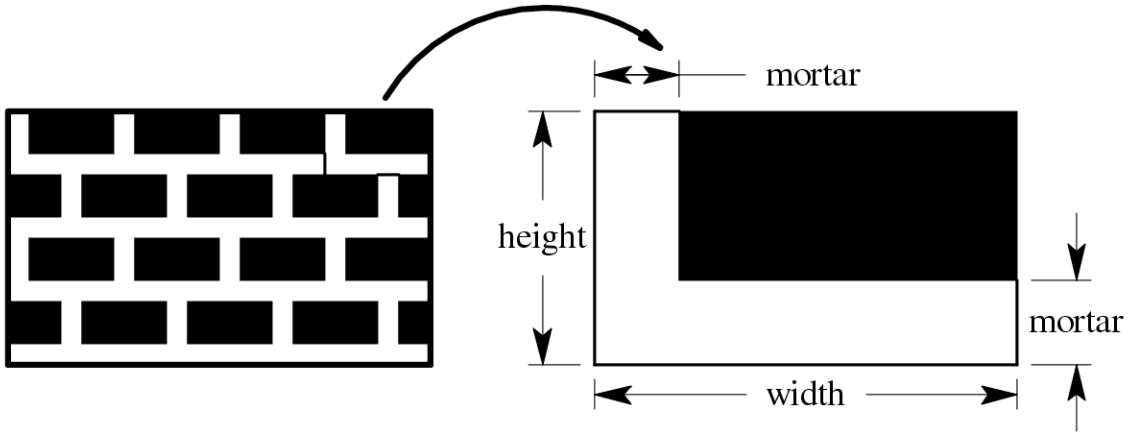

Figure 1.1. Size and shape parameters for brick shader

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.

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 1.1). 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,

```
// find row of bricks for this pixel (row is 8-bit integer)
fixed<8,0> row = tt/height;

// offset even rows by half a row
if (row % 2 == 0) ss += width/2;

// wrap texture coordinates to get "brick coordinates"
ss = ss % width; tt = tt % height;

// pick a color for this pixel, brick or mortar
float surface_color[3] = brick_color;
if (ss < mortar || tt < mortar)
      surface_color = mortar_color;
```

Figure 1.2. Portion of code for a simple brick shader

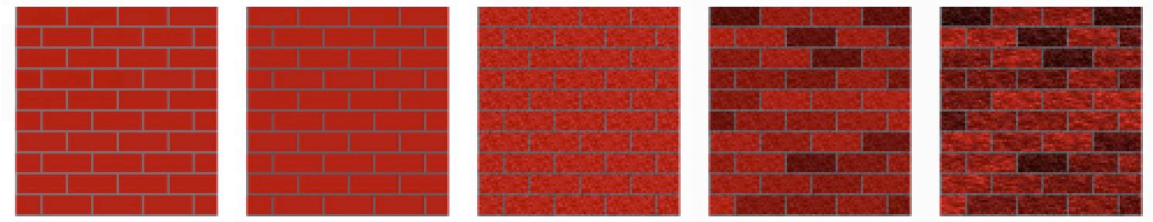

Figure 1.3. Evolution of a brick shader. a) simple version. b) with indented mortar computed bump map. c) with added graininess. d) with variations in color from brick to brick. e) with color variations within each brick

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 brick (see Figure 1.1). 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 1.2 (this shader happens to be written in the *pfman* language, the first real-time shading language). In this figure, `ss` and `tt` are local variables used to construct the brick coordinates. The simple bricks that result are shown in Figure 1.3a.

One of the real advantages of procedural shading is the ease with which shaders can be altered to produce the desired results. Figure 1.3 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 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 1.4a and b are frames from a rippling mirror animated shader. Figure 1.4c shows a yellow brick road where high-frequency elements fade out with distance. Figure 1.4d and e show a

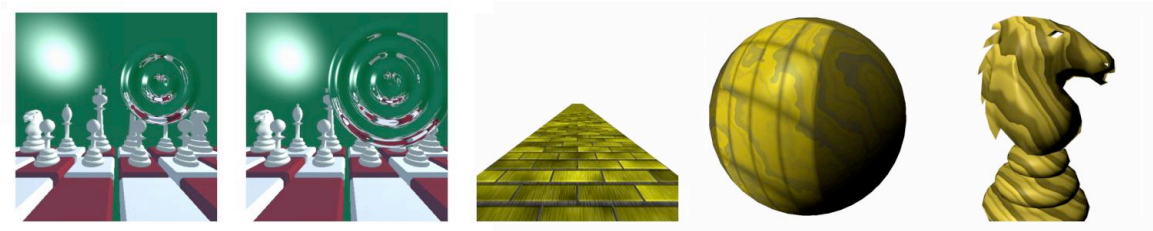

Figure 1.4. Examples of shaders. a+b) two frames of rippling mirror. c) yellow brick road. d+e) wood volume shader.

wood shader that uses surface position instead of texture coordinates. Figure 1.4d is also lit by a procedural light, simulating light shining through a paned window.

Shading for Interactive Rendering

The shaders shown above 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.

What's to Come

These notes are divided into a series of eleven chapters, each corresponding to one of the presentations during the course. Some chapters are longer, and others are only a couple of pages. The notes are roughly divided into four parts:

1. Shading Technology in chapters 1-3
2. Shading Languages in chapters 4-7
3. GPU Rendering in chapters 8 and 9
4. Shading Systems in chapters 10 and 11