Introduction

Marc Olano
University of Maryland, Baltimore County

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.

Interactive graphics machines are complex systems with relatively limited lifetimes. Just as the RenderMan shading language insulates the shading writer from the implementation details of the off-line renderer, 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.

1. 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, 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, detailed in Chapter 3). 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.

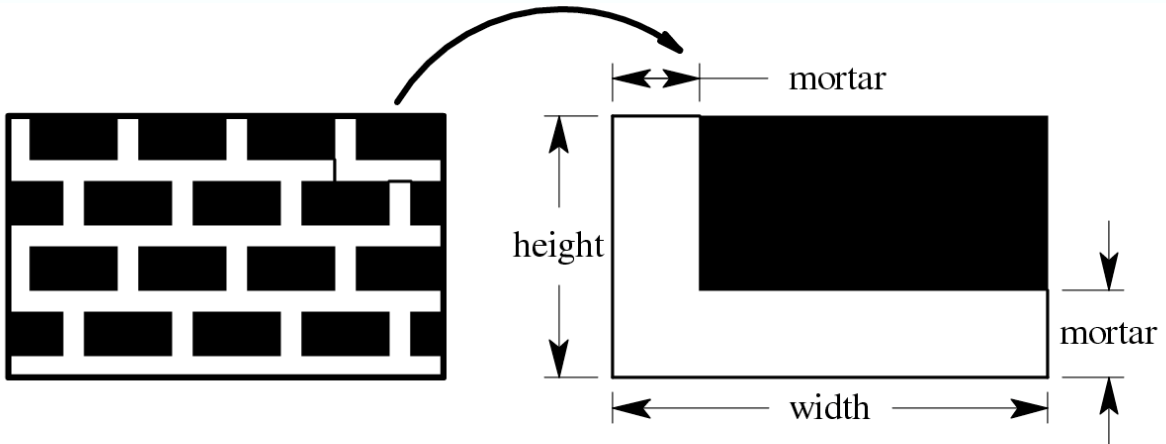

Figure 1.1. Size and shape parameters for brick shader

```
// 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

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.

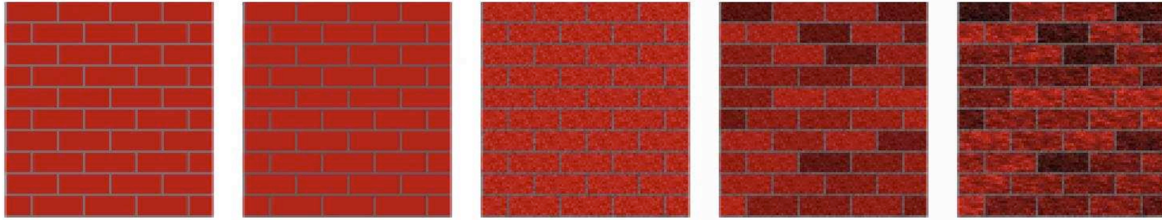

Figure 1.3. Evolution of a brick shader. a) simple version. b) with indented mortar computed bump map. c) with added graininess. d) with variations in color from brick to brick. e) with color variations within each brick.

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 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.

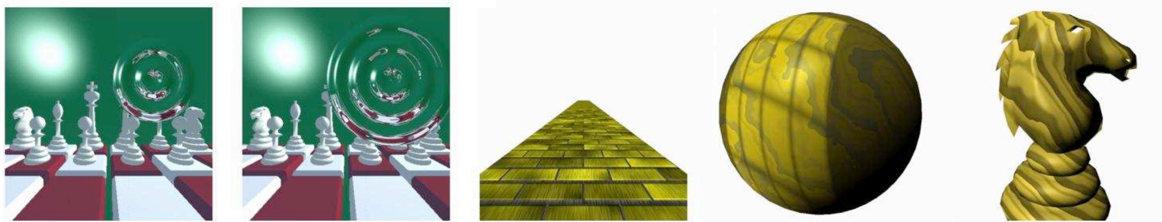

Figure 1.4. Examples of shaders. a+b) two frames of rippling mirror. c) yellow brick road. d+e) wood volume shader.

2. 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, todays 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.

3. What's to Come

These notes are divided into thirteen chapters, divided into four main areas. The first part covers models for shading, ideas that help us understand and use procedural shading hardware. The second part covers specific techniques for real-time shading. Here you will find some of the algorithms and techniques that

have been developed to run well in real-time shading systems. The third part focuses on specific shading systems, with contributions from three vendors of real-time shading hardware. The final part lifts the hood to explore in a little more depth the hardware and compilers that make real-time shading work. We provide the following as a rough guide to the connection between chapters in these notes, the course presenters, and what you might expect to find there:

Chapter	Presenter	Contents
Chapter 1	(Marc Olano):	This introduction.
	Part I: Models for Shading	
Chapter 2	(Marc Olano):	How procedural shading can allow us, its users, to ignore differences between very different hardware.
Chapter 3	(Randi Rost):	The OpenGL Shading Language is designed to be a cross-platform shading language that will remain useful for years and across generations of graphics hardware.
Chapter 4	(John C. Hart):	A way to understand and characterize the differences between different shading techniques.
	Part II: Techniques for Real-Time Shading	
Chapter 5	(Wolfgang Heidrich):	Hardware shading effects, the building blocks for later procedural shading systems.
Chapter 6	(Wolfgang Heidrich):	What it means to sample a procedural shader into a texture.
Chapter 7	(John Hart):	Several methods for producing solid textures on hardware.
	Part III: Shading Systems	
Chapter 8	(Randi Rost):	Shading at 3DLabs
Chapter 9	(Jason L. Mitchell):	Shading at ATI
Chapter 10	(Kurt Akeley):	Shading at NVIDIA
	Part IV: Under the Hood	
Chapter 11	(Bill Mark):	Characteristic of current graphics hardware that allow real-time shading.
Chapter 12	(Marc Olano):	A collection of papers on the design and development of shading compilers and what we can expect the compilers to do for us now and in the future.
Chapter 13	(All):	A collected bibliography of some of our favorite papers.