1 INTRODUCTION

Procedural shading is a proven graphics rendering technique in the production animation industry, and has been effectively used for years in commercials and feature films. For procedural shading, 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. These animations are created with an off-line renderer, taking anywhere from seconds to hours per frame. The resulting frames are replayed at 24–30 frames per second.

Polygon-per-second performance has been the major focus for *interactive* graphics hardware development during most of the time that procedural shading has been in use. Only in the last few years has attention been given to surface shading quality for interactive graphics. Today, even low-end PC products include image-based texture mapping as part of their fixed shading model. Instead of creating more and more complex fixed shading models, we would like to use procedural shading for images rendered at interactive rates (defined, for our purposes, as at least 10 frames per second and preferably 30 frames per second). Similar procedural techniques have also been used in off-line rendering at other points in the graphics process. We would like to use these same techniques throughout the interactive graphics rendering pipeline.

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.

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, we would like to present a simplified view of the interactive graphics system. We do this in two ways. First, we create an abstract pipeline of procedural stages. This abstract pipeline gives the user a consistent 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. 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.

The purpose of this dissertation is to design an abstract pipeline for interactive graphics; to demonstrate that this pipeline can be mapped onto an interactive graphics system; and to demonstrate that a special-purpose language can be used to write procedures that run at interactive rates.

1.1. Thesis Statement

> The decomposition of the graphics pipeline into a coherent set of user-programmable procedures provides valuable new tools to the interactive graphics programmer. In addition, a special-purpose language for writing the procedures insulates the programmer from the details of the graphics system while providing the system designer the opportunity to perform optimizations. This can be implemented efficiently on a graphics machine using current technology to yield a system that maintains interactive frame rates

1.2. 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 1.3). 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 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 (the full shader appears in Appendix A). 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.

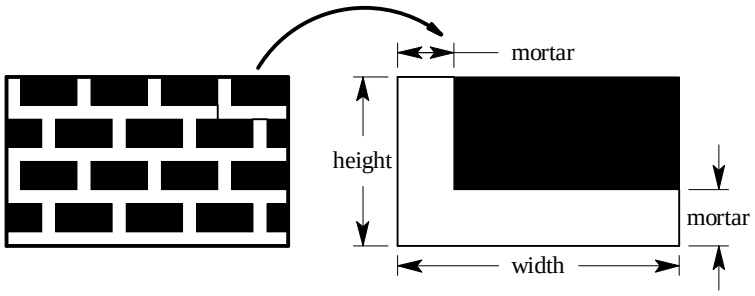


Figure 1.1. Size and shape parameters for brick shader

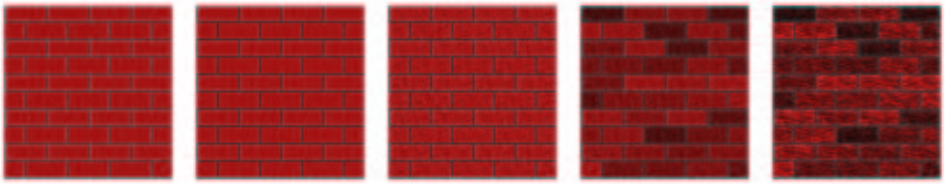
```
   // find row of bricks for this pixel (row is 8-bit integer)
fixed<8,0> row = tt/height;
   // offset even rows by half a row
if (row % 2 == 0) ss += width/2;
   // wrap texture coordinates to get "brick coordinates"
ss = ss % width;
tt = tt % height;

   // 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 1.3b-e show a series of changes from the

simple brick shader to a much more realistic brick. The code for the final resulting shader appears in Appendix A. 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.



a b c d e

Figure 1.3. Brick shader images.

a) simple version. b) with indented mortar. 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.

1.3. Abstract pipeline

We are interested in procedural techniques for more than just surface shading, so we decompose the rendering process into a pipeline of procedural stages. This pipeline casts the tasks of a traditional graphics pipeline into a form where every stage can be replaced by a user-written procedure. We call this pipeline of procedural stages the *abstract pipeline* because it provides a model of the graphics process. It is unlikely to exactly match the

organization of any particular graphics system, yet can be mapped onto the true structure of the system. Thus, it provides a convenient mental model for the user who will be writing new procedures.

The stages are independent and orthogonal, so the introduction of a new procedure for one stage does not require any changes in the other stages. This is an advantage for the procedure writer since it allows them to only write one procedure at a time. The independence and orthogonality also are an advantage for the system designer. A particular graphics system does not need to support user-written procedures for every stage of the abstract pipeline to be useful. It can support procedures at any number of the stages. Even a system that supports procedures at only one stage still appears to follow the abstract pipeline.

The stages in the abstract pipeline are: model, transform, primitive, interpolate, shade, light, atmospheric, and image warp. Procedures in the *model* stage correspond to objects in the scene. A modeling procedure creates the rendering primitives necessary to draw these objects. The *transform* stage handles positioning of objects as well as bending, twisting, and other changes to the object shape. A transformation procedure converts a point, vector, surface normal, or plane into a new one in its new position. *Primitive* procedures are responsible for drawing the geometric primitives. *Interpolation* procedures interpolate shading parameters across the rendered primitives. *Shading* procedures have already been mentioned, they determine the color and shading of a surface. *Light* procedures compute the direction and color of light that hit a surface, and can include effects like shadows or refraction. An *atmospheric* procedure changes the color of a pixel based on fog or other atmospheric effects that happen between the surface and the eye or camera. Finally, an *image warping* procedure can warp, filter, or modify the colors of the final image. Chapter 2 presents each of these stages in more detail, and uses the abstract pipeline as a taxonomy for the previous work.

1.4. Procedure language

Procedures can be written using one of two interface styles, which we call *testbed* and *language* interfaces. With a testbed interface, the user writes new procedures using the

internal library and data-structures of the graphics system. With a language interface, the user writes new procedures using a special-purpose high-level language. For either style of procedure, the abstract pipeline allows the user to ignore the details of the rest of the graphics system.

It is easier to provide a testbed interface for writing new procedures, but it requires the user to learn the internal details of the system. Because testbed interfaces rely on internal system libraries and architectures, testbed procedures are difficult to port to other graphics systems, even others that use the abstract pipeline. In contrast, the same language interface can be shared by many different graphics systems. The language provides an additional layer of abstraction, and the work of porting to the new graphics system is done by the procedure language compiler. This compiler can hide details of the graphics system while performing optimizations to improve performance.

We have created a language, *pfman,* for writing shading, lighting, or primitive procedures. Pfman is based on the RenderMan shading language, though we have added some features to the language. For shading and lighting, most of the features we added now appear in newer revisions of the RenderMan language standard. Because users often have a prior familiarity with the RenderMan shading language, we recommend future efforts use it for all procedure stages that RenderMan supports. The only language feature that we would still add to the current RenderMan language is a fixed point type. The pfman language and its features for shading and lighting are covered in Chapter 4.

We also have defined new pfman language features to support primitive and interpolator procedures, given in Chapter Primitives and Interpolation. This is an area not covered by RenderMan or the RenderMan shading language. Even if a future system uses the RenderMan language instead of pfman, it would still need these additions for the extra procedure types.

1.5. Demonstration

We have demonstrated several of these ideas on the PixelFlow graphics system. PixelFlow demonstrates the effectiveness of a procedure language interface for shading and

lighting computations, shows that the results can run at real-time rates of up to 30 frames per second, and explores some of the possible optimizations that can be performed.

PixelFlow supports a testbed interface for the primitive, surface shading, lighting, atmospheric, and image warp stages of the pipeline. It also supports a language interface for the shading and lighting stages and has preliminary support for a language interface for primitives. The pfman compiler produces C code for the existing testbed interface for each of these procedures.

As a special-purpose language, pfman, provides a high-level view that hides the system details from the user. The pfman compiler can do optimizations the user would otherwise have to do by hand. Memory allocation is the most critical optimization performed by the pfman compiler. Memory is a scarce resource on PixelFlow; without the pfman compiler's memory optimizations, shaders would not run. Other key areas of optimization deal with the communication bandwidth limits and the time limits imposed by the interactive performance goal. All of these optimizations are covered in more detail in Chapter 4.

Having a special-purpose language and compiler also introduces an extra layer between the user's intentions and the system. This introduces a potential source of inefficiency. The PixelFlow implementation of the language interface for primitive procedures exhibits some of these problems.

Thus far (March 1998), six people, other than the author, have written new procedural shaders in pfman. Having actual users of the system has guided us in the decisions of which features and optimizations to explore. It has also provided a great sense of satisfaction when a user produces new results that are not possible on any other current graphics machine.

1.6. Organization

The remainder of this dissertation will be organized as follows:

- Chapter 2 covers our abstract pipeline.
- Chapter 3 provides some details on the PixelFlow hardware, necessary to understand our PixelFlow demonstration system.

- Chapter 4 gives details on the surface shading and lighting stages on PixelFlow.
- Chapter 5 gives details on our implementation of the primitive and interpolation stages.
- Chapter 6 covers the experiences of our users who have created shading procedures on PixelFlow.
- Chapter 7 presents our conclusions and lists some areas of future research.