Shading Languages

Markus Kummerer

ABSTRACT

Shading Languages provide a highly flexible approach for creating visual structures in computer imagery. The RenderMan Interface provides an API for scene description, yet its true power lies in the integration of a C-like shading language. This special programming language describes how light sources emit light, surfaces reflect and transmit it, and the atmosphere affects it. Although programmable shading has gained great popularity in production animation, it is not suited for interactivity. Interactive Multi-Pass Programmable Shading provides a solution to this drawback.

1 INTRODUCTION

When creating a computer graphics image, two basic concerns arise. On the one hand, the user specifies a scene by positioning objects, light sources and the viewing device. This task is called modeling and requires high flexibility. On the other hand, simulating physical laws for light emission, reflection, transmission and projection is a static process. This turning of the internal representation into a two-dimensional image is called rendering.
The key idea behind the RenderMan Interface is to provide a separation between the modeling and the rendering domains. While the Interface scene description facilities are complete, its true power lies in its possibilities for describing the appearance of objects. Shading, an essential part of the rendering process, calculates the appearance of an object in a scene under a set of light sources, by setting up a function of light leaving a surface to light striking the surface. Visual complexity arises much more from surface variations than from the surfaces´ overall geometry, called shape. The second purpose of the RenderMan Interface is to bring this kind of surface variety to synthetic imagery. The key is flexibility in shading the surfaces in the scene, it is governed by a special programming language for describing how light sources emit light, surfaces reflect and transmit it, and the atmosphere affects it. Procedures written in this shading language are associated with individual surfaces and called during rendering, so that the shading can vary with position of surface, angle of view, time, etc.

2 THE RENDERMAN INTERFACE

The RenderMan Interface, currently at version 3.1, defines an API of 90 procedure calls. Due to the strict specification of the interface, the user is able to change the implementation of the renderer, as long as the new software adheres to the interface. The API provides:

Small, powerful set of primitive surface types
The types of the geometric primitives are basic, meaning that they can not be reduced to simpler primitives without sacrificing either efficiency or information useful in rendering. Three types of surfaces are supported: quadric, polygon and parametric. Quadric surfaces are surfaces of revolution, in which a curve in two dimensions is swept in three-dimensional space about one axis. This includes spheres, cones, cylinders, hyperboloids, paraboloids and tori. The second type of surfaces supported are polygons. They usually appear faceted and are best suited for flat objects. However, functions for convex, concave, planar and non-planar objects are provided. The most powerful type is the parametric. Bicubic surfaces and non-uniform rational B-Splines (NURBs) are part of the interface definition.

Hierarchical modeling, geometry
Single patches, polygons and quadric surfaces don´t have much visual appeal due to their limited complexity. They become interesting only when grouped to form more realistic items. A primitive object is just a single primitve, and an aggregate object consists of more than one object. Any aggregate object may be made part of another one just as a primitive object can. Thus, the RenderMan Interface supports a hierarchical model approach.

Also, a full set of capabilities for transforming object geometry within an explicit hierarchy defined by the geometric transformation environment is provided. This allows to give each part of an object its own local coordinate system.

Constructive solid geometry
CSG allows an object to be defined as a Boolean combination of other objects. A solid can be described as a set of points in three-dimensional space, with a surface separating points in the set from those not in the set. Using solids, one can either combine them, or use only their intersection, or subtract one from another.

Camera model
By default, a simple pinhole camera model is provided. Procedures for orthogonal and perspective projections are part of the interface. Nevertheless, other projections may be set by specifiing appropriate transform matrices. „Motion blur“ and „Depth of field“ calculations may also be applied to give the resulting image a more realistic look.

3 TYPES OF SHADERS

For each visible point on a surface, shading it involves the following tasks. First, the intensity and color of light, arriving from various light sources and even other surfaces, must be determined. A simulation of the interaction of the arriving light with the material follows. The third step consists of the modification of the light´s color as it travels from the object to the viewpoint. In other words, the result is defined by the colors of the surface and the light source(s), the position and orientation of the surface relative to the light and by the roughness of the surface. Figure 1 provides a graphical representation of the process. The RenderMan Interface provides the following types of shaders:

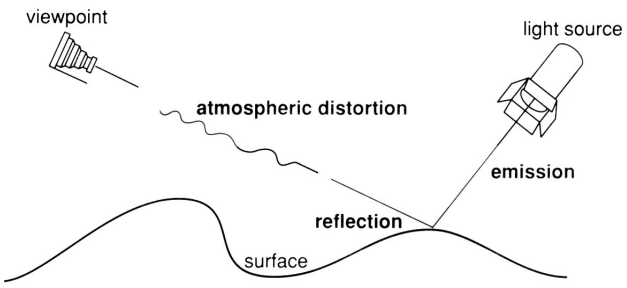


Figure 1: Three basic parts of the shading process

Light source shaders
A Light source shader calculates the intensity and color of light sent by the light source to a point on a surface. This type is the only shader that may have multiple instances available at the same time. That is, when a light source is instanced, it does not replace any existing light source, but is added to the light list. Figure 2 shows a graphical representation of the shader purpose.

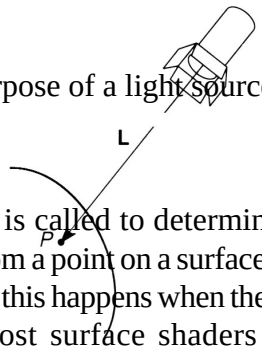


Figure 2: The purpose of a light source shader

A surface shader is called to determine the color of light reflecting from a point on a surface in a particular direction. Usually this happens when the point is found to be visible. Most surface shaders use the light arriving at the surface and the nature of the surface itself to calculate the color and intensity of the reflected light. A surface shader also has the opportunity to set the opacity of the surface at the point. With the cooperation of the renderer, the shader can thus let a surface reveal, either partially of completely, the surfaces behind it. Figure 3 depicts the geometric factors in surface reflection.

Figure 3: Geometric factors in surface reflection

Volume shaders

The idea of atmosphere affecting light passing through space between a surface and the eye is generalized in the concept of volume shaders. A volume is a region of space filled with any material that affects light passing through it. A volume can be as simple as a fog bank, or as complex as a human body captured by a CAT scan. Each has different effects on the light passing through them. The RenderMan Interface specifies that volume calculations are performed by volume shaders. They are invoked by the renderer and given the intensity, color and direction of light entering a volume. They calculate the intensity and color of light leaving that volume. Figure 4 shows how the volume shaders are related to the rendering process.

Figure 4: Volume shaders in the rendering process

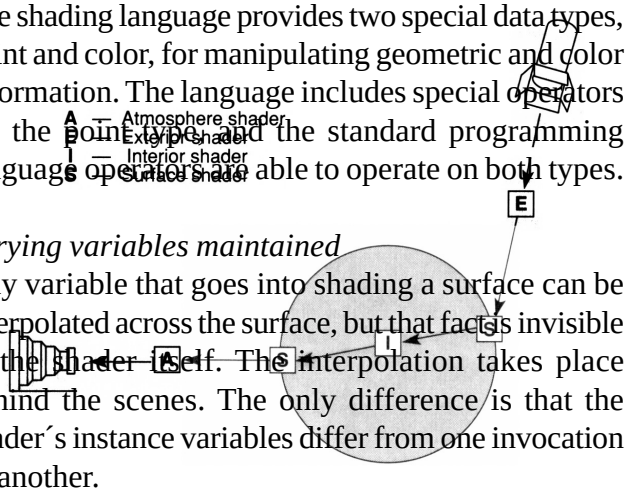
4 SHADING LANGUAGE

A key difference between shaders and standard programming languages is that the RenderMan Interface dramatically simplifies the task of writing shaders, both by an appropriate design and by setting out an extensive array of support services. Thanks to the environment shaders operate in, they are often just a few lines of code. Especially the shading language supports:

Orthogonal definition
One aid for shader writers is the RenderMan Interface´s definition of a canonical rendering process. Each shader has a well-defined role, and the shaders as a group are functionally independent.

Rendering environment
When a shader is called, geometric data is fully transformed, and a variety of relevant information is available in a set of global variables, thich are preset every time the shader is called.

Special data types
The shading language provides two special data types, point and color, for manipulating geometric and color information. The language includes special operators for the point type, and the standard programming language operators are able to operate on both types.



Varying variables maintained
Any variable that goes into shading a surface can be interpolated across the surface, but that fact is invisible to the shader itself. The interpolation takes place behind the scenes. The only difference is that the shader´s instance variables differ from one invocation to another.

Integration constructs
In addition to the usual programming language control constructs, the RenderMan shading language includes special constructs for spatial integration of light emerging from light sources and impinging on a surface.

Filtered map access
Texture maps and other forms of mapped data are accessed from shaders as simple function calls, with prefiltered return values. Any multichannel value can be stored in a map, accessed by a shader and used to control any aspect of shading.

Function library
The shading language includes a large library of mathematical, color, optical, geometric and noise functions. The common shading functions for calculating ambient, diffuse, specular and Phong reflections are also predefined.

5 EXAMPLES

First I provide a simple surface shader:

```
surface metal (
     float Ka = 1, Ks = 1,
     roughness = .25)
{
     point Nf =
     faceforward(normalize(N), I);

     Oi = Os;
     Ci = Os * Cs * (Ka * ambient() +
     Ks * specular(Nf, -I,
     roughness));
}
```

The specular() function implements a surface reflection that is not diffuse, but is rather concentrated around the mirror direction. The instance variable roughness is passed to specular to control the concentration of the specular highlight.

The next example uses texture mapping:

```
surface txtplastic (
     float Ks = .5, Kd = .5, Ka = 1,
     roughness = .1;
     color specularcolor = 1;
     string mapname = "")
{
     point Nf = faceforward(N, I);

     if (mapname != "")
          Ci = color texture(mapname);
     else
          Ci = Cs;
     Oi = Os;
     Ci = Os * ( Ci *
          (Ka * ambient() + Kd *
          diffuse(Nf))
          + specularcolor * Ks *
          specular(Nf, -I, roughness));
}
```

Instead of the surface color Cs, txtplastic() uses a texture map to provide the surface´s diffuse reflection.

The last example uses a displacement shader:

```
displacement pits (
     float Km = 0.03; string marks = "")
{
     float magnitude;

     if (marks != "")
          magnitude = float
          texture(marks);
     else
          magnitude = 0;

     P += -Km * magnitude * normalize(N);
     N = calculatenormal(P);
}
```

Figure 6: metal(), txtplastic() and pits() shader applied

CONCLUSION

While the RenderMan Interface is a complete scene description methodoloy, its true power arises from the use of shaders. Procedures written in the RenderMan shading language are associated with individual surfaces and called during rendering, so that the shading can vary with position of surface, angle of view, time etc. This flexibility has had a widespread impact on production animation. Pixar´s PhotoRealistic RenderMan has become the de facto standard in photorealistic rendering.

REFERENCES

Steve Upstill: The RenderMan Companion, Addison Wesley 1989

Pixar Webpage: www.pixar.com

Mark S. Peercy, Marc Olano, John Airey, P. Jeffrey Ungar: Interactive Multi-Pass Programmable Shading, Computer Graphics Proceedings 2000