337

S2-60

RenderMan Design Principles

N90-20653

Tony Apodaca
Pixar
3240 Kerner Blvd.
San Rafael, CA 94901

Tom Porter
Pixar
3240 Kerner Blvd.
San Rafael, CA 94901

ABSTRACT

The two worlds of interactive graphics and realistic graphics have remained separate. Fast graphics hardware runs simple algorithms and generates simple-looking images. Photorealistic image synthesis software runs slowly on large expensive computers. The time has come for these two branches of computer graphics to merge.

The speed and expense of graphics hardware is no longer the barrier to the wide acceptance of photorealism. There is every reason to believe that high quality image synthesis will become a standard capability of every graphics machine, from superworkstation to personal computer. The significant barrier has been the lack of a common language, an agreed-upon set of terms and conditions, for 3-D modeling systems to talk to 3-D rendering systems for computing an accurate rendition of that scene.

Pixar has introduced RenderMan *to serve as that common language. This paper examines* RenderMan, *specifically the extensibility it offers in shading calculations.*

NASA has been at the forefront of developments in computer graphics. One area in particular has been the quest for realism in synthetic image generation. Voyager animations done at JPL a decade ago captivated many with the notion that the process of scientific discovery and the popular understanding of that process could both benefit from visually accurate computer generated imagery.

Computers have sped up since those animations were made. Tools for modeling and controlling the animation have also improved. Yet too often, the ability to produce complex and accurate renditions is relegated to specialized labs. The challenge that we face is in bringing this technology to the desktop, running it on every graphics platform, linked across the standard networks, fed from the common databases.

The goal is to unify the often divergent methodologies used in the computer-aided-design of a 3-D object, the analysis of that object during simulation, and the accurate representation of the object.

What is RenderMan?

RenderMan is an interface between 3-D modeling systems and photorealistic rendering systems. Modeling is the process of describing objects to a computer. We use modeling here to refer to all aspects of describing a scene, including its dynamics. Rendering is the process of generating an image of the scene from a given viewpoint. RenderMan is an interface proposal which will permit a large variety of geometric modelers to talk to a large variety of renderers with a straightforward, common format.

The central problem in making such a proposal is to accommodate the needs of advanced rendering in a clean way, while allowing standard CAD databases to feed the interface. Only then can photorealistic image synthesis be brought under the same wing, integrated into the same computing environment as other aspects of CAD and simulation.

Shape and Shading

An overriding principle in the design of RenderMan used to solve this problem is a recognition that an interface proposal must distinguish clearly between shape and shading, between the geometry of the scene and the visual characteristics of the geometry. The visual complexity of real world imagery is not found in the general shape of objects, but rather in the textures and materials and lighting and dynamics. In fact, the graphics community already *has* sufficient CAD tools to specify the shapes of things. We lack the tools to describe visual qualities, such as atmospheric conditions, reflectivity of materials, and characteristics of light sources.

A second principle is that shading computations need to be far more general than the Gouraud and Phong interpolation set forth in the textbooks. The world is not all plastic. We need rendering systems that can wrap an atmospheric texture around a spherical planet, that can compute a noise function to simulate the bumpiness of a surface, that can handle surface properties other than color, perhaps to compute renditions outside the visible spectrum.

The RenderMan interface is a specification for approximately 100 subroutines with which a modeler can completely describe all of the information that a renderer might need to generate an image of a scene. It provides entry points for geometric information, transformation hierarchies, color and material property information, camera parameters and output image characteristics.

The RenderMan interface supports a rich variety of geometric primitives. For example, convex polygons, concave polygons (with and without holes), polyhedral models, and a large number of quadric surfaces are supported. RenderMan includes a very comprehensive bicubic patch primitive, specified with an arbitrary basis matrix. RenderMan also supports non-uniform rational B-spline surfaces (NURBS).

Support for primitives such as these guarantees that most standard CAD packages can feed the RenderMan interface quite easily. There are two significant capabilities of the interface in extending the common notions about geometry:

First, RenderMan supports *procedural primitives*. One of the biggest problems in modeling natural phenomena (such as mountains, plants, fire, etc.) is that the geometric complexity is enormous. This problem is usually solved by writing programs which generate all of the tiny detail, rather than model it by hand. However, it can still be very expensive for the modeler to generate a huge complex model and then pass it to the renderer, particularly if the modeler doesn't know how much of it the renderer really needs. RenderMan's procedural primitives permit the user to give the renderer a pointer to a subroutine which will expand simple objects into more complicated ones, such as converting a triangle into a fractal mountain or a sphere into a particle system explosion. Using procedural primitives, the modeler can download a very complex model such as a fractal into the renderer in a carefully controlled way, so that only the required amount of detail is sent through the interface.

Secondly, RenderMan has a very general interface for specification of the arbitrary parameters on a surface. This permits the user to specify not simply the position and color, but also the surface normals and texture map coordinates on a per vertex basis. In addition, the vertex structure can actually be extended by the user at run-time, to include arbitrary information of his choosing such as temperature or stress or density or any other values that might be interesting to his particular application. These parameters can then be used to control the shading calculation.

Shading Language

Most software renderers have a subroutine which determines the color of the surface of an object. Typically, it will implement a single mathematical equation which uses a simple model of the reflection of light in order to calculate the contributions of the light sources and texture maps upon the surface color. The equation often has a lot of parameters (5 to 20, depending on the renderer) which the user tweaks to control the appearance of different kinds of materials (plastic, metal, chalk, etc.).

Very often, however, you want the surface to have some characteristic which you can't achieve with the fixed equation, such as the use of a texture map to modify some shading parameter. If you are fortunate enough to have the source code, you can add your function and recompile. If not, you are out of luck.

RenderMan changes this model, by providing the facility of the shading language, a C-like programming language which has new functions and data types that are specifically designed for the purpose of calculating colors based on geometric information. Programs which users write in the shading language are typically small (10 to 20 lines), and are loaded into the renderer at run-time when they are requested by some part of the scene geometry. These programs then replace the built-in shading equations. Users can use this language to customize the shading on a per-object basis. This new freedom gives the user the power to model the appearance of objects as carefully as he models their shape.

The shading language supports three basic data types, the `float`, the `point` and the `color`. `point` and `color` are abstract data types which are actually vectors of floating point values. The standard C arithmetic operators (`*`, `+`, `/`, etc.) work on these data types. In addition, there are some new operators for vector dot and cross product. The familiar C conditional and looping constructs are available (except `switch`), as are subroutine definitions and calls. There is a rich library of mathematical functions, as well as a library of functions which implement common shading operations such as normalizing vectors, transforming points between coordinate systems, calculating diffuse and specular lighting, interpolating colors, splining and calculating pseudorandom numbers.

RenderMan actually permits the user to define up to four separate shading language programs which provide different material characteristic information about each object: a *surface* shader, which determines what color we see when light reflects off the surface; a *displacement* shader, which can move the surface small amounts to add dents or fillets which are too small or too complex to model geometrically; a *light* shader, which describes how luminous objects emit light; and a *volume* shader, which describes how light is attenuated as it passes through the interior of a translucent object. This may seem a bit complicated, but it actually quite a straightforward way to think about the material properties of objects, particularly once you've seen them in action.

Shaders

The renderer calls the appropriate shading language program (*shader*) every time a light intensity, surface color, etc., is required. When a shader is called, it has available to it a large number of global variables which are provided by the renderer. These variables include all of the geometric information that the renderer knows about the surface being shaded, such as the position `P`, the surface normal `N`, the color `Cs` and opacity `Os` that the user specified, the texture coordinates `s,t` and others. The variables that the user applied to the vertices of his primitives are also available inside the shaders. Each type of shader accomplishes its specified task by calculating and modifying a specific part of this global state. For example, a surface shader is responsible for calculating and setting `Ci`, the color that the eye sees. A light shader is responsible for setting `Cl`, the light color.

Listing 1 shows an example of a simple surface shader. This shader calculates the reflectivity of a metallic object, using a simple equation. It makes use of the standard library functions `ambient`, `diffuse` and `specular` to determine the amount of light arriving on the surface from the light sources. These functions implement three customary equations based on the direction and strength of the incoming light. If those functions had not been appropriate, the surface shader has access to the lights and could have calculated whatever values it pleased from them. The shader then calculates a weighted average of the incoming light intensities and multiplies by the color of the object. Notice also that the shading language took care of the multiplication of float values by color vectors automatically, freeing the user from having to write the ugly loops which would have been present in most other languages.

The type of the shader (in this case `surface`) indicates its intended function. Parameters to the shader are specified using a syntax similar to ANSI C. This shader demonstrates another other unique feature of the shading language, the presence of default values in the parameter list. When a modeler requests this shader, it specifies the parameters it wishes to override by name. Any parameter not mentioned is left with the default value.

```
surface metallic (float Ka = .4,
                  Kd = .4, Ks = .6,
                  roughness = .25;)
{
   N = faceforward(normalize(N));
   Ci = Cs * (Ka * ambient() +
              Kd * diffuse(N) +
              Ks * specular(N,
              -normalize(I), roughness) );
}
```

Listing 1. A simple shader which simulates the reflection of light off of metallic objects.

Listing 2 demonstrates a displacement shader. The purpose of a displacement shader is to move the position of the surface around a little bit to simulate tiny fillets, dents and other minor surface perturbations. This greatly adds to the visual interest of an object, and makes it look much more realistic. This particular shader calculates a fractal dentedness using several iterations of *noise*, a function which produces a semirandom value which changes slowly over the surface of the object (using a purely random value would distort the surface beyond recognition, since adjacent points would have no relationship to each other). Getting the same effect by trying to model the intricate surface dents would be extremely difficult.

```
displacement dent (float scale = 1.0;)
{
    float size = 1.0, displace = 0.0;
    for (i=0; i<6; i+=1.0) {
      /* Calculate a simple
      fractal 1/f noise function */
      displace += abs(.5 - noise(P * size))
            / size;
      size *= 2.0;
    }
    /* Displace the surface and
      recalculate surface normals */
    P += N * pow(displace,3.0) * scale;
    N = calculatenormals(P);
}
```

Listing 2. Shader which simulates dents by moving the surface a small amount. This adds visual complexity which is very difficult to model convincingly using standard geometric modeling techniques.

Sensor Simulations

RenderMan can generate output much more general that the simple pinhole camera/RGB images provided by current systems. RenderMan can, for example, compute color in multichannel spectral spaces. Landsat data can be used as input texture maps to control multiple surface parameters mapped onto a planet surface. Shading language procedures can be written to use surface parameters such as temperature; in this way, multichannel sensor image acquisition can be simulated.

RenderMan allows the user to specify other parameters of the simulated camera, in order to provide information to renderers which support advanced rendering features. For example, the user can set the shutter time as well as the focal length, focal distance and f-stop of the camera, to simulate motion blur and depth-of-field. RenderMan allows the user to specify the positions, shapes and colors of the objects at multiple times during the shutter interval, so that sophisticated renderers that can simulate motion blur will know how the objects are moving.

High quality rendering requires a lot of attention to the sampling and filtering which is performed on the output pixels, in order to avoid *aliasing*. RenderMan gives the user independent control over the number of shading samples per pixel and the number of hidden surface samples per pixel, as well as the size and shape of the pixel filter function. In addition to the standard display parameters of output image name and device type and image resolution, RenderMan supports gamma correction and exposure control. These functions compensate for a monitor's phosphors' tendency to glow with exponentially increasing brightness as voltage increases linearly. It also contains the new concept of an *imager* shader, another shading language program which permits the user to implement various color manipulations on final pixels just before they are put into the framebuffer or file.

Conclusion

The RenderMan interface is a powerful interface between 3-D modeling systems and photorealistic rendering systems. It is designed to bring the highest quality in image synthesis into widespread use. Modern CAD modeling tools can feed RenderMan from their standard database of geometry. RenderMan provides simple built-in shading language procedures to provide for a range of standard material properties.

RenderMan provides a shading language for far-reaching extensibility in user specification of specific visual characteristics of the scene. The interface exposes a great deal of control over the shading process; modelers are encouraged to offer user-defined shading language procedures for renderers to execute. By partitioning the modeler/renderer interface in this way, high-quality rendering can be made accessible to a vast array of modeling systems and CAD databases.

RenderMan is the only graphics interface proposal to deal with issues in high-quality synthetic image generation such as antialiasing, texture mapping, motion-blur, shadows, spectral color models and programmable shading languages. These advanced features are not available on most of the rendering software and hardware that is currently available. As such, RenderMan represents a goal for sophisticated new graphics hardware and rendering software to shoot for.

Users of graphics workstations and personal computers will be the biggest winners, as photorealism becomes inexpensive, commonplace and compatible across a wide range of platforms.

Copies of *The* RenderMan *Interface, Version 3.0* are available from Pixar, 3240 Kerner Blvd., San Rafael, CA, 94901. Please enclose $15 to defer the cost of printing and mailing.

Further Reading

Foley, James D., and Andries VanDam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA, 1982.

Hall, Roy, *Illumination and Color in Computer Generated Imagery*, Springer-Verlag, New York, 1988.

Joy, Kenneth I., et al, (ed.) *Image Synthesis*, Computer Society Press, Washington, DC, 1988.

Pixar, *The* RenderMan *Interface, Version 3.0*, May 1988.

Rogers, David F., *Procedural Elements for Computer Graphics*, McGraw-Hill, New York, 1985.