Under the Shade of the Rendering Tree

The goal of computer graphics has always been to create increasingly realistic images. Faster processors and more sophisticated rendering techniques have allowed computer artists to create scenes that come very close to simulating reality. In particular, computer graphics are good at rendering architectural scenes. Raytracing and radiosity renderings often fool me into believing I am seeing actual photographs. Even in the real-time game market, techniques such as multi-pass rendering and precomputed lighting have enabled game players to run around in a world complete with reflections, shadows, dynamic lights, and an impressive amount of texture detail.

As I write this, QUAKE 3: ARENA has been unleashed upon the world. This impressive game takes real-time rendering technology to a new high. Once again, the graphics capabilities of this game {**Do you mean as each of id's games has done before it?** } stretch real-time rendering to its limit, bringing even the latest "graphics processing units" to their knees. QUAKE 3 also marks the industry debut of programmable "shaders," which are used for describing the look of a real-time rendered image.

A shader is a form of programming language that describes the look of a particular surface in a rendered world. In its most abstract sense, it is a function that is given a series of properties and then returns the color of the light leaving any position on the shaded surface. Normally, the properties given to a shader include such things as the lights in the scene, the color of the surface, and some measure of the roughness of the surface.

Game programmers and artists don't normally think of the rendered world in these terms. However, even the most basic 3D rendered scene can be described in these {**the preceding or the following terms?**} terms. A texture map that is applied to a 3D polygon simply describes the color of the light that leaves that polygon at any point on its surface. Likewise, the Gouraud shading model is a series of parameters that controls the interaction of the lights in the scene with the color and roughness of the polygon surface. The power of a shader language, however, goes way beyond what we have traditionally done with real-time 3D rendering. Since a shader describes the color leaving the surface of polygon, it can be used to generate a complex pattern of colors without texture maps.

Most of you are familiar with procedural textures. This is where a texture map is created by some form of mathematical formula instead of being drawn in an art package. Procedural textures are commonly used for patterns such as noise (like TV "snow"), lava, water, marble, or fire. UNREAL implemented procedural texture techniques for several effects used throughout its environments. This allowed the designers to have a nearly unlimited variety of certain types of textures without having to store all those bitmaps on the game CD. However, the textures still needed to be generated in order to load them onto the 3D card for rendering.

Wouldn't it be nice if those textures never had to be generated at all? What if I could simply upload a small program that handled all my procedural textures? Then all I would need to provide to the rendering hardware would be a few variable settings for each different material. Sounds kind of futuristic, right?

It is not as far out as you think. Shading languages have been around for quite a while. The first, and still most commonly used, is Renderman. First described in the late 1980s, this rendering language has been used to create some of the most memorable computer graphics scenes of all time, including the recent movie *Toy Story 2*. While it may seem that we are a long way from creating scenes this complex for real-time games, you may be surprised.

Listing 1 describes a Renderman shader that creates a checkerboard pattern on a surface. The shader takes three float variables and two colors and creates a checkerboard of any size and frequency. This is done without any texture map. For a checkerboard, this may not seem very impressive; however, it's the idea of controlling the look of an individual pixel on an individual surface that makes Renderman so powerful. A shader doesn't need to be as simple as a checkerboard. Shaders can be used to create all kinds of surfaces, everything from highly-detailed wood, marble, and fire to even a moldy cue ball (my favorite Renderman shader).

A closer examination of the checkerboard shader reveals that the only other thing the shader really needs to know about is the position of the view and the lights in the scene. Interestingly, the new generation of 3D graphics hardware such as Nvidia's GeForce 256

When not ditching work to catch the latest animated feature film, Jeff can be found at Darwin 3D trying to convince clients that things can't look any better. Tell him how wrong he is at jeffl@darwin3d.com.

keeps these positions in hardware already. I can't help but think that the hardware manufacturers are thinking of the implications in the same way that I am. I don't know how long it will take, but I am going to dust off my *Renderman Companion* and start thinking about how to integrate programmable shaders into my art production pathways. Since I can't really see envision many artists learning to program Renderman, I think there are going to be a lot of tools that will need to be created. However, until I get my ultimate shader language written, I am stuck with the traditional texturing and lighting methods to get the results I want.

Welcome to Toon Town

I have lamented before in this column that creating 3D characters is very difficult. I can take some comfort from the fact that even Pixar, with its terrific Renderman shading system and all the money and talent possible, has trouble getting human characters right. They have hit upon one of the great ironies of computer graphics. When rendering 3D environments, the technology has enabled increasingly realistic final images. With each advance in modeling or lighting, the images take a step closer to what we see around us in the real world.

With human characters, on the other hand, the story is entirely different. In my experience, as a 3D computer-generated human is rendered in an increasingly realistic manner, it paradoxically looks increasingly strange to viewers. They can't really say why it looks odd, just that it's not quite right. This is especially noticeable when the texture maps for the character faces are created from photographs of real people.

Particularly frustrating is the fact that people are able to look at a stick figure performing an animation and appreciate the lifelike motion. However, when that same motion is applied to a synthetic 3D character, those same people get hung up on the look of the character. They no longer regard the motion of the character as realistic simply because it looks "odd." If the character is a monster or something else nonhuman, this problem seems to go away. Well, that's great if you're creating a shooter filled with mutated zombies and uncontrollable robots. However, if you're creating a realistic scene filled with average people, you're in trouble.

This observation has led me to think that for now, at least, the focus for real-time 3D characters should not be on trying to achieve realism. Instead, we should be looking at approaches to creating stylized characters. Perhaps Disney had the right idea. For years, its artists have seemed to understand and appreciate this paradox. They were able to create very realistically painted backgrounds full of color and depth. For the actual characters, though, they still rely on simple pen-and-ink drawings. Even when the first fully CG character was introduced in a Disney animated feature, the magic carpet in *Aladdin*, it was rendered in a style that matched the traditional methods. With this in mind, is it at all surprising that 3D animated series such as Mainframe Entertainment's *ReBoot* and *Beast Wars* focus on robotic and animal characters?

LISTING 1. *A Renderman checkerboard.*

```
//
//      Shader:                 Checkerboard Shader
//      Arguments:Diffuse and Ambient Coefficient
//                                      Number of squares
//                                      Two colors to alternate
//
surface checkerBoard(
        float Kd, Ka;                   //  Specular and Ambient Lighting
        float frequency;        //  Number of Squares
        color c1, c2                    //      Two colors
        )
{
        // S and T vary from 0 - 1 across the surface
        float smod = mod(s*frequency,1);        // Interval in S direction
        float tmod = mod(t*frequency,1);        // Interval in T direction

        // Ci is the output color
        if (smod < 0.5) {                                       // Odd Columns
                if (tmod < 0.5)
                        Ci = C1;                                        // C1 Square
                else
                        Ci = C2;                                        // C2 Square
        } else {                                                        // Even Columns
                if (tmod < 0.5)
                        Ci = C2;                                        // C2 Square
                else
                        Ci = C1;                                        // C1 Square
        }

        Oi = Os;  // Opacity out = opacity in

        // ambient() returns ambient light value
        // diffuse(N, I) returns the sum of lights from
        // incident vector I and surface normal N
        Ci = Oi * Ci * (
                Ka * ambient() +
                Kd * diffuse(faceforward(N,I)) );

        // Ci is final color
}
```

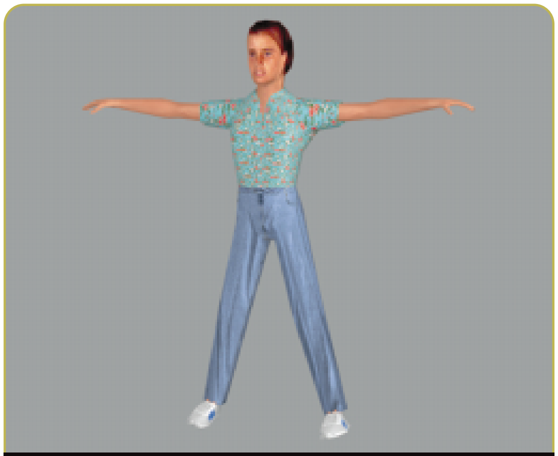

FIGURE 1. *Our original character, outfitted with textures created from photographic scans.*

Losing a D

Perhaps it is time to look at using 3D technology to create much more stylized animations instead of realistic ones. This technique, called non-photorealistic rendering (NPR) in academic circles, has emerged as a strong research field at industry conferences such as Siggraph. That means there are plenty of fresh, steaming piles of research to get me started.

The character in Figure 1 was created using textures created from photographs and scans. I wanted to get something much more stylized, so we had another model and set of textures created with a cartoon kind of look in mind. You can see the results in Figure 2. This character is much more typical of the kinds of characters you may see in a 3D game. However, it doesn't quite capture the 2D look I had in mind.

For one thing, the shading implies too much depth. The maps really need to be reduced to only a few colors. This is no problem to do in any image processing program as you can see in Figure 3. This is much closer to the idea of a cartoon rendering. However, the image is clearly missing the bold outlines that characterize cartoon images. To create those lines, I need to turn to some technology.

GL to the Rescue

The first lines that I need to create are the silhouette lines. These lines define the outline of the character. On a 3D model, the outline of a model is defined by the model's edges. Intuitively, I know that a silhouette edge must occur when an edge connects a polygon facing forward and a polygon facing backward. This can be expressed mathematically as:

$$\left(N_1 \bullet (V-E)\right)\left(N_2 \bullet (V-E)\right) \le 0$$

where N_i are the two face normals for the adjacent polygons, V is a vertex on the edge, and E is the eye point. When this statement is true, the edge is part of the silhouette.

As you can imagine, this would be a rather time-consuming process on a model that had any significant number of faces, but this method has the benefit of identifying the actual edges that define the silhouette. This could be useful if I wanted to apply some other effects to the silhouette lines. But for now, I want to look for a faster way that makes use of my existing 3D hardware.

I can start by drawing the front-facing polygons with texture. I can then draw the back-facing polygons in line mode. Since the Z-buffer is already filled for front-facing pixels, the only pixels drawn will be those pixels along the edge. However, in order for this to work, I need to set the depth test so it draws pixels that are at the same depth as those in the Z-buffer. In OpenGL, this setting is `glDepthFunc(GL_LEQUAL)`. This gives me a rendering algorithm like this:

1. Draw front facing textured polygons
2. Set depth test to LEQUAL
3. Draw back facing lines.

Or in OpenGL:

```
glPolygonMode(GL_FRONT,GL_FILL);   // Draw Filled Polygons
glDepthFunc(GL_LESS);        // Don't draw shared edges
glCullFace(GL_BACK);         // Draw front facing polygons only
DrawModel();              // Call my draw routine
glPolygonMode(GL_BACK,GL_LINE); // Draw Lines
glDepthFunc(GL_LEQUAL);      // Draw shared edges
glCullFace(GL_FRONT);        // Draw back facing edges only
DrawModel();              // Call my draw routine
```

You can see the result in Figure 4. The first frame shows just the resulting silhouette lines and the second frame shows the combined image.

With this technique, I can use OpenGL to enhance the effect. I can make the lines thicker or even anti-alias the lines with alpha blending (or even anti-aliased hardware lines, if available). It is even possible to make the lines pop out off the

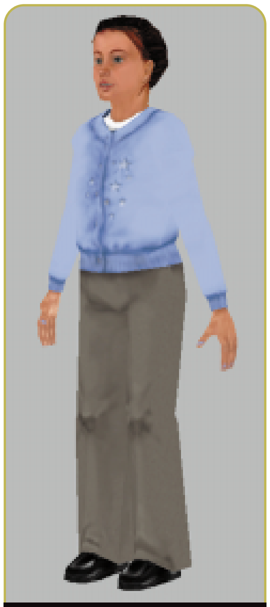

FIGURE 2. *A more stylized, cartoonish version of the character.*

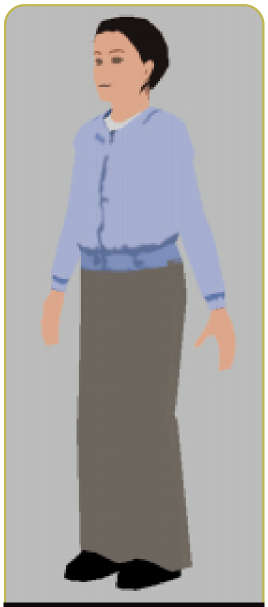

FIGURE 3. *Reducing the depth of colors gives a more 2D look.*

model using glPolygonOffset(). However, this can lead to a mess, so you need to be careful.

Another approach that may make cleaner lines requires the use of an extra pass and the stencil buffer. In this technique, the algorithm is:

1. Draw front facing textured polygons
2. Set draw mode to stencil only
3. Draw front facing edges in line mode
4. Draw back facing lines where stencil is set.

This will ensure that only edges that are shared front and back are drawn. However, since the stencil buffer is not commonly available across consumer hardware, it may be wise simply to test for it and use it when possible.

In addition to the silhouette lines, I probably want to add interior lines that define changes in the material of the character. This cannot really be done easily with just rendering tricks. This requires a pass through the object to detect edges that share polygons with different materials. These edges are marked as material boundaries and are drawn after the render. Luckily, the material edges are not viewer-dependent so they can be calculated only once as a preprocess.

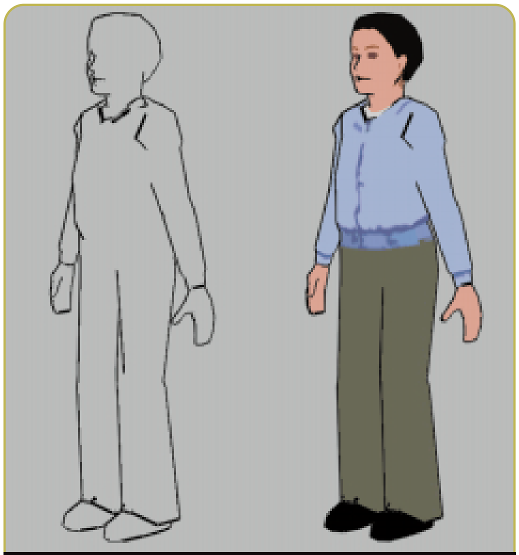
FIGURE 4. *To complete the cartoon look of the character, black outlines were added with OpenGL.*

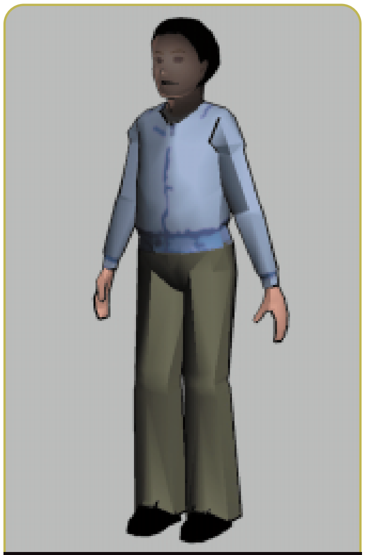
FIGURE 5. *Finally, shading is added with our lighting model.*

FOR FURTHER INFO

Siggraph 1999 had an entire course on Non-Photorealistic Rendering. If you are interested in the topic you should definitely get the notes or the courses CD: "Non-Photorealistic Rendering," Course 17 (Siggraph 1999, ACM Siggraph).

- Markosian, Lee, and others. "Real-Time Non-Photorealistic Rendering," *Proceedings of Siggraph 97*. New York: ACM Siggraph, pp. 415–420.

- Uphill, Steve. *The Renderman Companion*. New York: Addison Wesley, 1990.

Some Shadier Business

Now that I have a nice method for creating cartoon-style characters with silhouette lines, I need to think about shading. Applying typical Gouraud shading to these characters would ruin the effect I am trying to achieve. I need to change the lighting model to make this work.

In the Gouraud shading system, the angles between the viewer, light, and surface normal are used to determine the shade of the vertex. For my simple cartoon rendering, I only want two shades for each material, light and dark. In order to do this, I need to calculate the vertex colors myself. The formula I applied is:

$$\left(N_V \bullet (V - E)\right) < \varepsilon$$

where N_v is the vertex normal, V is the vertex position, E is the eye point, and ε is the shading threshold. This is very similar to the silhouette-detection formula. However, in this case, when the result of the formula is less than a certain threshold, ε, the vertex is shaded with the "dark" color. Otherwise, the standard color is used. The ε value can be changed to allow for more or less shading on the character. You can see the results of this shading in Figure 5.

Acknowledgements

Thanks to Lisa Washburn at Vector Graphics (http://www.vectorg.com) and Tom Knight at Imagination Works (http://www.imagination-works.com) for the models and textures used in this article.

The Squashy and Stretchy Show

I think these techniques provide a new way of thinking about real-time 3D animation. It's a classic example of embracing your limitations. There is lots of room for experimentation and exploration. Creating the ideal texture to work with non-photorealistic rendering will require some creativity on the part of artists. Some experimentation with brush patterns and other artistic styles might be interesting as well.

Another intriguing idea would be to apply some of the soft-body deformation techniques that I described in my March 1999 column to the models ("Collision Response: Bouncy, Trouncy, Fun"). These new squishy objects can be rendered using NPR techniques to get a real Road Runner feel. For this month, play around with the simple cell shader and begin exploring the world of the less-than-realistic. Grab the source code and the application from the *Game Developer* web site at http://www.gdmag.com. ■