Shades of Disney: Opaquing a 3D World

Like many people who work with computer images, I am a huge fan of classic animation. The amount of labor that goes into creating an animated feature film has always amazed me. Consider for example Disney's *The Jungle Book*. The film is 78 minutes long. At 24 frames per second, that's 112,320 frames of animation. Each of those frames needed to be drawn, cleaned up, inked, painted, aligned with the background, and finally shot to film. Each of these steps required a great deal of skill and patience on the part of the artists involved.

Look at the job of the opaquer. This person is responsible for receiving the final cels from the ink department and coloring them using opaque paint. This job is essentially coloring in between the lines using a paint-by-numbers key known as a color model. While it seems like a fairly straightforward, though repetitive, job; opaquing was very time consuming during the early years of animation. Shamus Culhane, who was deeply involved in the process at Disney for many years, estimated that his opaquing department could average about 25 cels per day. At that rate, it would have taken his team 12 years to opaque the cels for *The Jungle Book*. Clearly, the staff for this Disney classic worked their little animated tails off.

Fortunately for the animation industry, computers have come along. Through the use of a digital ink and paint system, a single artist can opaque several hundred cels in a single day. As an extra benefit, the computer eliminates many of the problems artists had matching colors painted on various layers of acetate. Painting an animated feature is still a major issue in animation, but the job has gotten much easier.

Enter the Next Dimension

In the digital world of 3D real-time animation, I have some opaquing problems of my own. Last month I looked at methods for creating cartoon-style rendering on 3D objects. I was able to deal with creating the silhouette and material lines, however I had yet to get the cartoon look for the surface of the object. I suggested that I would need to look to texturing techniques to get that part of the job done.

You can see the situation I would like to end up with in Figure 1. Given one light shining on the model, I want there to be a clear separation of the light and dark halves of the model. A classic model for illumination gets me most of the way. I want the shade to be a function of the surface normal and the light position. In the Lambertian reflection model, the brightness of a surface position depends only on the angle between the direction to the light source, *L*, and the surface normal, *N*. Mathematically, that would be

$$I = k_d(N \bullet L)$$

The dot product is taken between the surface normal and the light source direction and is multiplied by a diffuse lighting constant. Since the dot product will vary from 0 to 1, this would

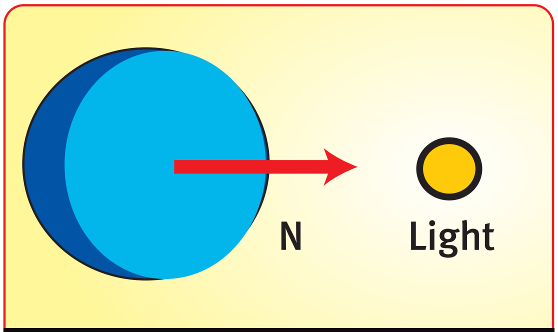


FIGURE 1. *A nice cartoon shading showing a clear delineation between light and dark portions of the model.*

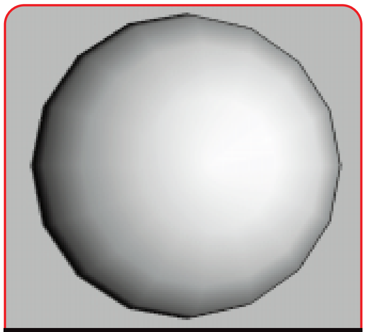


FIGURE 2. *A plain Lambertian reflection gives a Gouraud-shaded look.*

When not glued to his TV watching the Cartoon Network, Jeff can sometimes be found at Darwin 3D. Send him a message at jeffl@darwin3d.com and we will slip it to him during a commercial break.

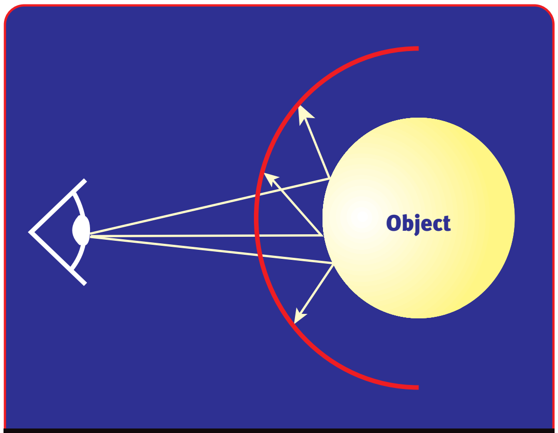


FIGURE 3. *The spherical environment-mapping technique can be used to make an object look as if it is reflecting light.*

just give me the basic Gouraud-shaded-ball look where the illumination value goes smoothly from white to black as you can see in Figure 2.

What I need to create is a cutoff where the illumination is "light" or "dark." The ideal formula would be

$$I = \left(k_d\left(N \bullet L\right) < \varepsilon\right)$$

where ε is the shading threshold. As I discussed last month, I could compute the vertex colors at every vertex using this formula. However, this wouldn't get the desired results. Graphics hardware interpolates the vertex color across each triangle. Since the cutoff point could potentially occur in the middle of a triangle, a simple interpolation would not look correct.

It's tempting to consider using environment-mapping techniques to create the effect. Spherical environment-mapping calculates the ray from your eye that reflects off the surface to the point that it strikes on a hemisphere around the object. You can see this illustrated in Figure 3.

This technique is used to make things look reflective, like shiny metal. It's also a method for creating a specular highlight on an object. I could create an environment map that transitions from light to dark, as seen in Figure 4, then apply that to the object. This gives me exactly the result I was looking for but has a few problems. For one, the coordinates are calculated from the eye point. In order to get the look I want, I will need to calculate the environment map from the light. This is possible, but kind of a pain.

The second problem is that if I wish to change the shading threshold or the number of in-between values, I would need to recalculate the environment map completely. That would be a bit of a burden on the CPU.

Using a Texture as a Lookup Table

Another thing you may have noticed is that the map in Figure 4 is a bit wasteful. The same color gradient is repeated around the circle radiating from the center. Let's look again at the formula I am trying to reproduce. I know that the dot product term will vary from 0 to 1. I can calculate the value for I for each dot product from 0 to 1 and store it in a table.

$$I = \left(k_d\left(N \bullet L\right) < \varepsilon\right)$$

$$\sum_{u=0}^{1} ShadeTable[u] = \left(k_d\left(u\right) < \varepsilon\right)$$

For example, suppose $\varepsilon = 0.375$. The table would look like Figure 5.

Now I can take this table and convert it into a one-dimensional texture (I know you've probably always wondered how those could be used). I set up the 1D texture in OpenGL with a couple of easy function calls that are almost identical to their 2D equivalents.

```
glGenTextures(1,&m_ShadeTexture);
glBindTexture(GL_TEXTURE_1D, m_ShadeTexture);
// Do not allow bilinear filtering
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB, m_ShadeWidth, 0,
GL_RGB , GL_FLOAT, (float *)m_ShadeSrc);
```

You will notice that I turned off filtering. This is because I actually want to get a banded, shaded look. If filtering were on, the colors would be blended in a way that would not look at all right for my purposes. Since filtering can slow things down on some cards, this ends up being beneficial for performance as well.

In order to use this new 1D texture, I simply need to calculate the dot product and index that result into the table as a texture-map coordinate. For an object that can rotate, the vertex normal will need to be rotated by the object matrix before the dot product is calculated. The code for

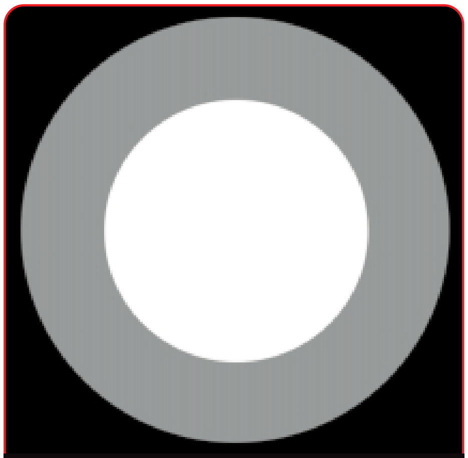


FIGURE 4. *A cartoon environment map, transitioning from light to dark.*

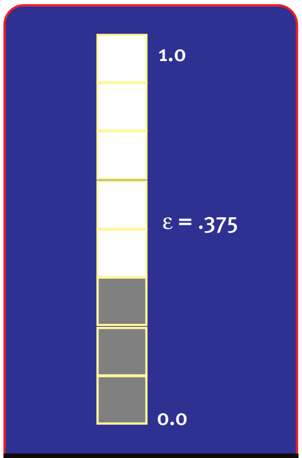


FIGURE 5. *A lookup table for the given value.*

calculating the table index is given in Listing 1.

This new lookup map is applied just like a normal texture map except for the fact that it is 1D and requires only one texture coordinate. The results of this process can be modulated with the object's surface color to get the final image. This gives me a great deal of flexibility in how the shadow is applied across the image. I can control the cutoff level and the number of levels of shading across the surface, and I can even add a highlight by brightening the top of the table. To recalculate the values, I only need to update the table — quite a bit easier than the entire 2D texture map. The table can be of any resolution your graphics card can handle. If you use too few shades, the resulting surface will appear very banded and blocky. I found that for most objects, a 32-pixel table looks pretty good. You can see a variety of shade tables and their respective results in Figure 6.

I now have a fast and flexible real-time cartoon renderer. The whole concept of using the texture-mapping capabilities of 3D graphics hardware to apply arbitrary functions across a surface is very powerful. You can create a very complex and completely nonlinear shade table and then apply it to the surface and let the hardware interpolate it.

LISTING 1. *Code for calculating the texture coordinates.*

```
///////////////////////////////////////////////////////////////////////////////
// Function:  CalculateShadow
// Purpose:   Calculate the shadow coordinate value for a normal
// Arguments: The vertex normal, Light vector, and Object rotation matrix
// Returns:   An index coordinate into the shade table
///////////////////////////////////////////////////////////////////////////////
float COGLView::CalculateShadow(tVector *normal,tVector *light, tMatrix *mat)
{
//// Local Variables ////////////////////////////////////////////////////////
   tVector post;
   float dot;
///////////////////////////////////////////////////////////////////////////////
   // Rotate the normal by the current object matrix
   MultVectorByRotMatrix(mat, normal, &post);
   dot = DotProduct(&post,light);  // Calculate the Dot Product

   if (dot < 0) dot = 0;           // Make sure the Back half dark
   return fabs(dot);               // Return the shadow value
}
```

On hardware that supports texture-coordinate generation and features such as cubic environment mapping, these methods are definitely worth exploring.

Other Methods

Obviously I'm not the only person exploring the use of non-photorealistic techniques for real-time rendering in games. Sim Dietrich of Nvidia has been exploring the use of hardware-accelerated transformation and lighting for non-photorealistic rendering. Methods such as my use of the normal and dot product require the CPU to perform calculations on each vertex. Sim's goal is to minimize the use of the CPU by using features found on Nvidia's GeForce 256 GPU.

The GeForce 256 supports cubic environment mapping and texture-coordinate generation. By using a cubic environment map and the D3DTOP_DOTPRODUCT3 operation to generate texture coordinates for the environment map, Dietrich can create a cartoon rendering with very limited CPU impact. In addition, by applying more rendering passes, he is able to add some texture to the shaded part of the image. You can see some examples of Sim's work in Figure 7.

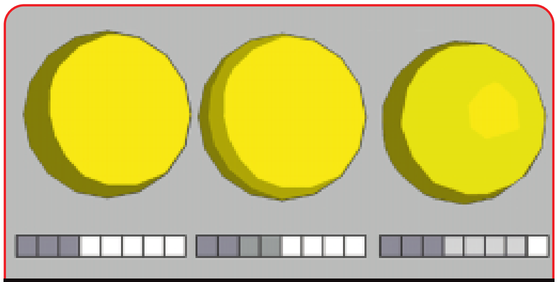


FIGURE 6. *Some shade tables and results.*

Intel Goes to Toontown

Intel has been creating a variety of impressive technologies available to the game development community. They have been working on a licensable real-time non-photorealistic rendering algorithm as part of the Intel 3D Software Toolkit that they are announcing at this year's Game Developers Conference. The software allows you to specify line settings such as thickness, color, and type. For the shading, you can set the shadow cutoff level and brightness as well as a highlight level and value. Intel has also been work-

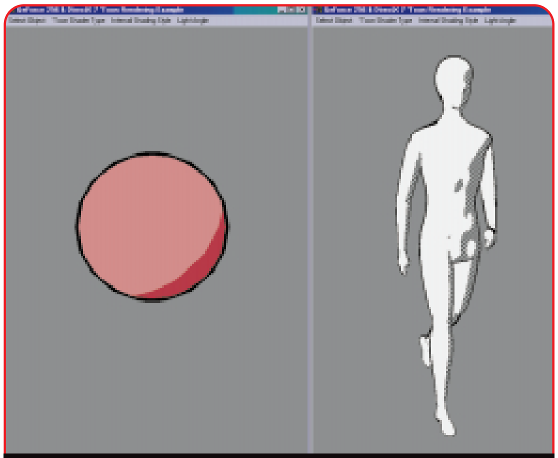
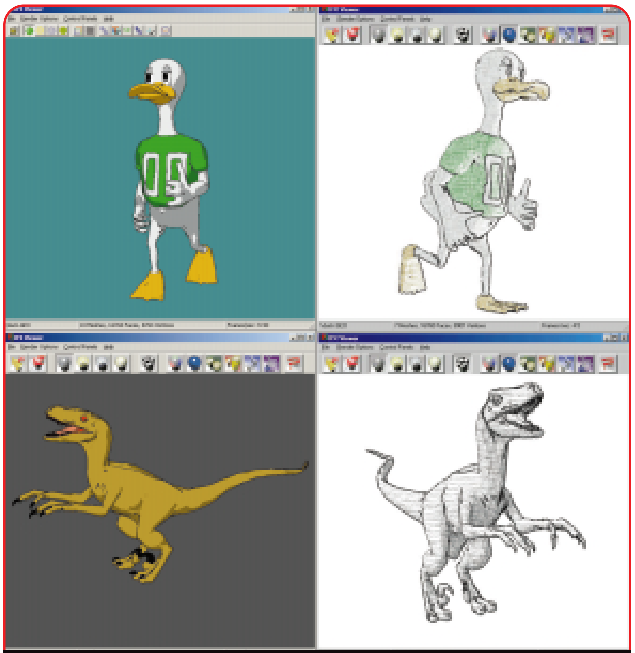


FIGURE 7. *This Nvidia cartoon rendering optimizes the GeForce 256 architecture, minimizing CPU impact.*

ing on creating a variety of rendering styles such sketch and pen-and-ink to go along with the cartoon rendering.

The 3D Toolkit will integrate this renderer with other 3D technology such as the multi-resolution mesh, subdivision surfaces, and a skeletal deformation system. The package will be available for use in a variety of real-time projects.


The Intel 3D Software Toolkit integrates a number of 3D technologies.

A Word About Digital Cinematography

Of course, even after all this work, creating the rendered look for your characters is only part of the battle. You also need to know how to display them effectively. In many real-time 3D games, the camera is tied directly to the character. It bobs along behind the character bumping off walls (or going through them in some cases) and wedges itself in a place guaranteed to block the thing you are trying hardest to see. Camera control has become a major part of the 3D game creation process. Poor camera operation and control is sometimes enough to cause a game to receive a poor rating in game magazines. Clearly it's time to start thinking about the camera as an integral actor in the scene. Perhaps games have matured to the point that it's time to look to film production techniques and assign cinematographers to camera control in 3D interactive games.

Not long ago, most cinematics in games were created using traditional filmmaking and animation methods. These movies could break the immersive experience by pulling the player out of the action. These days, however, it seems like more developers are creating their cinematic sequences using the game's real-time engine. This trend has brought a variety of new problems with it. Many of these games use pre-scripted sequences for camera control to display the action in a pleasing way. This is fine for noninteractive sequences or dialogue trees. But if we want to have truly interactive sequences that the players can enjoy the way the project director intended, we need to take a serious look at the art of real-time camera control.

Consider the example of the action/adventure game. Many of these games use a tethered camera under complete user control. Game designers must be content letting the player manipulate the camera in order to show the action. Anyone who has played a game like this knows it can be very difficult to manipulate the character and the camera at the same time. Many times the player will get the camera into a "good enough" position and continue with the action, but this position will probably not be the best one for displaying the action.

One alternative approach I have seen is never to give players control of the camera in the first place. This can be frustrating to players as they may have a different idea of what is important in an interactive situation. There are then the hybrid methods which yank control away from players to show them something "dramatic." This can be jarring and totally pulls players out of the interactive experience, leaving them no longer in control. It is clear to me that the interactive medium requires some fresh thinking about storytelling.

Filmmakers have been telling stories with the visual medium for a long time now. They have certainly learned a few things along the way. Out of those experiences a certain visual style has formed that guides basic filmmaking. I am not saying that these rules are not or should not be broken. However, when they are broken it is to achieve a desired effect, not simply out of ignorance of their very existence. This cinematic style, sometimes called continuity style, describes shot framing and staging methods that enhance storytelling without confusing audiences.

Next month, I'll be looking at methods for shooting an interactive story. Till then, think about the best and worst camera control you have seen in a game and let me know about it. ■

FOR FURTHER INFO

Traditional Animation

Culhane, Shamus. *Animation from Script to Screen*. New York: St. Martin's Press, 1988.

This book is an excellent source for all aspects of traditional animation — a must-have for animators and pretty useful for technical types. Covers everything from the walk cycle to facial expressions to starting your own studio. The only book I know that describes how a dodo bird walks.

Nvidia

http://www.nvidia.com

Sim Dietrich should have posted his document and application for cartoon rendering by now. If not, drop Nvidia's developer support an e-mail.

Intel 3D Software Toolkit

http://www.intel.com

Watch for a major announcement at the Game Developers Conference, March 8–12, 2000.