Ca i g Shad V l me

by Hubert Nguyen

As I recall, the very first 3D game with real projected 3D shadows was VIRTUA FIGHTER, a Sega AM2 arcade game based on a Model1 board. That was the beginning of a new era of fighting games. Each character was composed of more than 1,500 flat-shaded triangles. Their shadows were projected onto a perfectly planar ring. Sega AM2's developers had several reasons for choosing a simple planar ring.

1. A planar area was sufficient for a terrific 30 FPS game play.
2. Casting shadows on a relief is CPU-hungry because it's not hardware-accelerated.
3. The main processor (a 680x0 family) was too slow to handle the inverse-kinematics mathematics needed for characters on a nonplanar area (which didn't appear until arcade machines started featuring the Model3 board).

For years, fighting games were the only games to use real-time shadows. These games typically featured only two characters, each casting a shadow on the floor. But these games lacked interobject shadow casting, such as the first player's shadow projected onto the second player's. Nowadays, with the power of arcade systems such as Sega's Model3 family, games such as VIRTUA FIGHTER 3 are capable of more elaborated special effects. Nonplanar shadows are now possible, though interobject shadow casting is still missing.

My love of fighting games inspired my efforts to come up with a way to create shadows that could be projected onto any object, even onto the other fighter (Figure 1). As with many techniques, the one I'll describe in this article has advantages and disadvantages, but I think the idea is worth sharing.

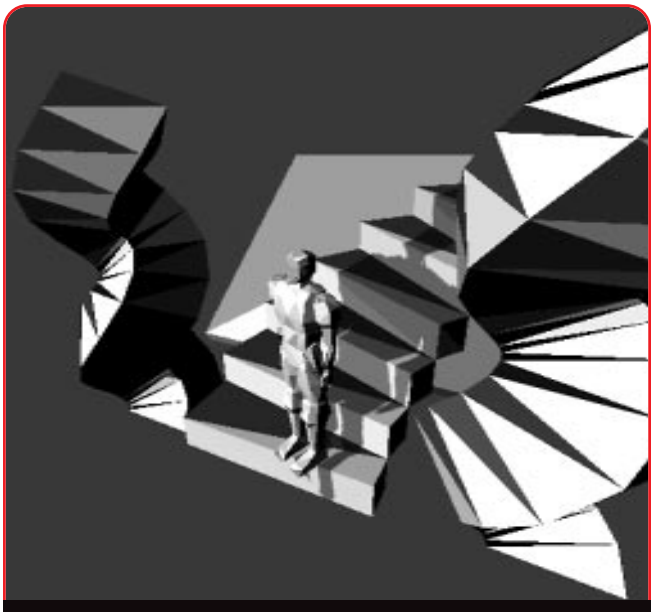


FIGURE 1. *An example of projection shadow mapping, in a screenshot taken from the sample application accompanying this article.*

The Easy Answer?

There are several ways to create 3D shadows. The most popular technique is to project the whole mesh (or at least the part that's visible from the light) onto a single plane (Figure 2). Some developers are using a simplified mesh to cast the shadow (for example, TEKKEN 3 from Namco). This technique dramatically reduces the number of triangles that must be processed, and usually gives acceptable visual quality. Proceeding with polygons does offer benefits such as

Nguyen Hubert Huu lives in Nanterre, France. He built demos for Impact Studios between 1993 and 1995, and is now working for a French game publisher. You can contact him via e-mail at nguyenhub@aol.com

FIGURE 2. *An example of shadow created with triangles projected on a plane.*

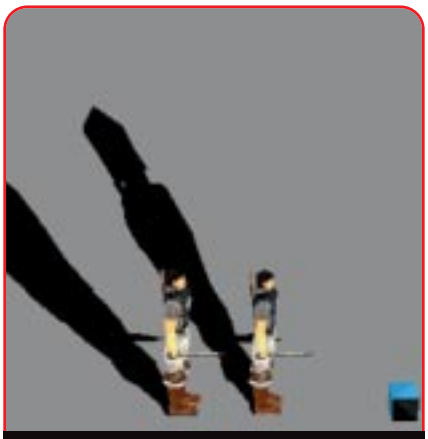


FIGURE 3. *Note that edges are sharp even if the shadow is stretched.*

LISTING 1. *Establishing the relationship between* **SrcObject** *and* **DstObject.**

```
01    loop on SrcObject polygons
02    {
03    loop on DstObject polygons
04       {
05          project SrcObject polygon on DstObject polygon plane
06          if ( projected SrcObject polygon shadow in DstObject polygon)
07          {
08             Newpolygon <- project SrcObject polygon on DstObject polygon plane
09             Shadowpolygon <- clip Newpolygon against DstObject polygon edges
10          }
11       }
12    }
```

razor-sharp edges (Figure 3) and speed (the game can render the shadow in flat-shading, which is usually faster than fully lit texture-mapped rendering). Projecting and clipping polygons on an infinite plane is pretty fast.

Also, on a nonplanar surface, creating polygons for a shadow is more difficult. First, you have to perform the projection onto a general plane, rather than the typical single-axis–aligned plane (vertical or horizontal). We could say that casting a shadow onto something other than a plane is equivalent to projecting one object onto another, both composed of polygons. Let's call the shadow-caster object **SrcObject** and the receiver of the shadow **DstObject**. For each polygon of **SrcObject**, we'll have to determine which polygons of **DstObject** will receive the shadow. One could represent the process with the pseudocode shown in Listing 1.

Of course, this algorithm is a little "brutal." It can be optimized to reduce the number of polygons that must be processed. For example, we could skip those polygons that are light-face–culled. Nevertheless, the overall complexity of this algorithm will always be pretty high. Lines 05 and 06 can be quite CPU-intensive compared to a simple planar shadow casting without clipping. We could further optimize this technique's performances by precalculating and storing certain data, such as the planes equation for each **SrcPolygon**, **DstPolygon**, and so on. But this algorithm is still a pretty complex process.

The code in Listing 1 will give us a brand new set of polygons with which to build the shadow. In most cases, the number of triangles in the shadow will be greater than or equal to the number of triangles in the **SrcObject** that are visible from the light source because a **SrcObject** polygon can be projected onto more than one **DstObject** polygon. This calculation is probably the costliest part of this algorithm. Fortunately, alternatives do exist.

Projecting Shadows

I first realized that I could use a texture to cast a shadow when I saw a 3Dfx demo featuring a spotlight implemented as a projected texture. I imagined how easy it would be to render a shadow into the projected texture. Projecting textures doesn't require CPU-intensive operations such as projecting onto general planes and clipping polygons. All we have to do is calculate the texture coordinates for applying a rendered texture onto an object. Projecting shadows employs a similar technique. The principle is to generate textures dynamically using a light source as the point of view.

The idea is pretty simple. Let's imagine that we want to project a picture onto an object. we simply transform each vertex of the object into the projector space (the light source), and use its new coordinates (x',y',z') to create the texture coordinates (s,t). This technique is even more intuitive when you can see what's happening. Figure 4 is

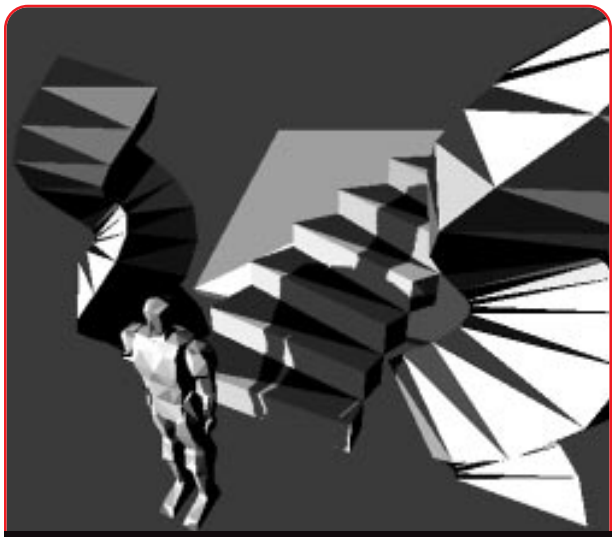


FIGURE 4. *Let's see how to achieve this effect...*

FIGURE 5. *This is the scene from the light's field of view.*

FIGURE 6. *These objects will receive the character's shadow.*

FIGURE 7. *The shadow texture map.*

the final result that we want to obtain. Figure 5 shows the scene and the figure from the light's point of view. You can see in Figure 5 that the shape of the shadow in Figure 4 is exactly hidden by the figure of the man when viewed from the light's point of view. The scene will receive the shadow. Figure 6 shows the scene objects from the light's point of view; we need the coordinates of these objects in order to calculate the (s,t) coordinates for the shadow-mapping. Figure 7 shows just the shadow texture — we've rendered the objects for which we want to create shadows (in this case, the character only). By projecting the texture in Figure 7 onto the objects in Figure 6, we get the final effect (Figure 4). In theory, casting shadows using textures is as simple as that.

Now let's look at the code. A sample application that performs these calculations is available from the *Game Developer* web site. The most important part of the algorithm computes the texture shadow coordinates for all the objects onto which we want to cast shadows. This calculation basically remaps the (x,y,z) coordinates of the object into (s,t) coordinates in the shadow map (Listing 2).

Here, the (x,y,z) coordinates are computed in the light's space coordinates (the object has been previously transformed by the `Rotate()` function). Figure 8 helps to visualize how the mapping is performed. The projection is planar — we only use (x,y). Because the object has been projected with its perspective relative to the light's point of view, the mapping takes care of the perspective. Figure 9 illustrates the result of this operation. Once we've remapped the (x,y) coordinates as (s,t) coordinates, we simply use (s,t) as normal texture coordinates. In our case, remapping was as simple as reusing the (x,y) coordinates as (s,t) coordinates.

Obviously, we need to consider the size of the shadow texture and the size of the viewport used to transform the object into the light's point of view. The sample application uses Glide, which has texture coordinates in the range

LISTING 2. *Remapping the (x,y,z) coordinates of the object into (s,t) coordinates in the shadow map.*

```
void CastOnMesh(Mesh &m)
{
float prod;

for (int i=0; i<m.nTriangles; i++)
{
   int v1= m.TriangleArray[i*3+0];
   int v2= m.TriangleArray[i*3+1];
   int v3= m.TriangleArray[i*3+2];
   m.FlagArray[i] = 0;     // by default, reset the flag

   // Backface Culling
   //((v3.x - v1.x) * (v2.y - v1.y)) - ( (v2.x-v1.x)*(v3.y-v1.y) )
   prod =
   (
   ((m.VertexArray[v3].px-m.VertexArray[v1].px)*(m.VertexArray[v2].py-m.VertexArray[v1].py))
   -((m.VertexArray[v2].px-m.VertexArray[v1].px)*(m.VertexArray[v3].py-m.VertexArray[v1].py))
   );
   if (prod<0.0f) continue;       // reject by BackFace Culling

   if ((m.VertexArray[v1].flags &
      m.VertexArray[v2].flags &
      m.VertexArray[v3].flags) != 0) continue; // reject if the triangle is completely
         // outside of the light POV
   m.VertexArray[ v1 ].u = m.VertexArray[ v1 ].px; // (x,y,z) are in Light FOV
   m.VertexArray[ v1 ].v = m.VertexArray[ v1 ].py;

   m.VertexArray[ v2 ].u = m.VertexArray[ v2 ].px;
   m.VertexArray[ v2 ].v = m.VertexArray[ v2 ].py;

   m.VertexArray[ v3 ].u = m.VertexArray[ v3 ].px;
   m.VertexArray[ v3 ].v = m.VertexArray[ v3 ].py;
   m.FlagArray[i] = 1;        // 1= draw this triangle for the current frame
   }
}
```

FIGURE 8. *Superimposition of the scene's position and the texture from the light's point of view.*

FIGURE 9. *The scene has been displayed using the shadow texture.*

[0..256] instead of the range [0..1.0] used by most of APIs. Because the shadow texture is a square 256×256, we don't have to scale the (x,y) coordinates when using them as (s,t) coordinates.

Crossing Hurdles

We need to be careful of a couple of things in regard to texture coordinates. Some (s,t) couples will have values outside of the range [0.0..1.0]. In order to avoid a tiling effect, which would distort the result by repeating the shadow in an undesired place, the rendering device must be configured into clamp mode.

Every texel can be addressed by a couple of texture coordinates (s,t) in the range [0.0..1.0]. In theory, giving a value outside of this range to s or t would cause an addressing error. Most people who've programmed a software rasterizer have encountered this problem. To avoid it, many hardware manufacturers have methods of keeping texture coordinates inside of the texture. These include tiling and clamping. Both modes modify the texture coordinates after they've been interpolated by the chip for each texel. When configured for tiling, the chip removes the integer part of the (s,t) coordinates (for example, 1.7 becomes 0.7). Clamping, on the other hand, brings the values that exceed the range to the nearest bound (for example, -2.3 becomes 0, and 1.3 becomes 1.0).

Figure 10 shows a clamped square that has texture coordinates greater than 1.0 and less than 0.0. Figure 11 shows the opposite, with tiling enabled. If you look closely at Figure 10, you can see that if the borders of our texture aren't clean, we'll get an unwanted effect on the display of our shadow. Figure 12 is an example the sort of problems that can arise in clamp mode. This bug is caused by the object being clipped by one of the texture's borders (Figure 13). To avoid those effects, we must make sure that our object is completely inside the point of view and isn't being clipped by the texture's border. We must be sure that our textures have clean borders, with a width of at least two pixels. In the sample application that accompanies this article, I've attached the light to the object and adjusted the point of view to avoid any clipping. A piece of code could automatically check whether or not the object is inside the point of view. A good adjustment will maximize the accuracy of the shadow in the scene by increasing the surface taken by your shadow in the texture. We'll talk about that at the end of the article.

Implementation

Now that we understand how texture projection works, let's look at how it fits into the rest of the program. I've divided the process up into five steps:

1. Render the object for which we want to create a shadow from the light's point of view.

FIGURE 10. *Clamping example.*

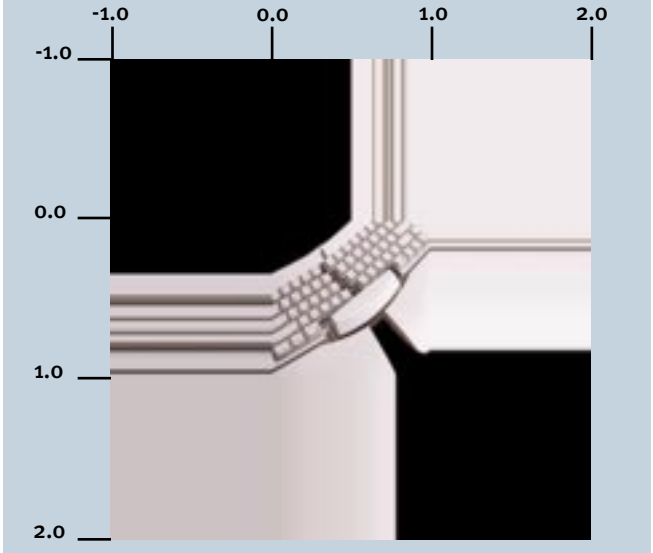


FIGURE 11. *Tiling example.*

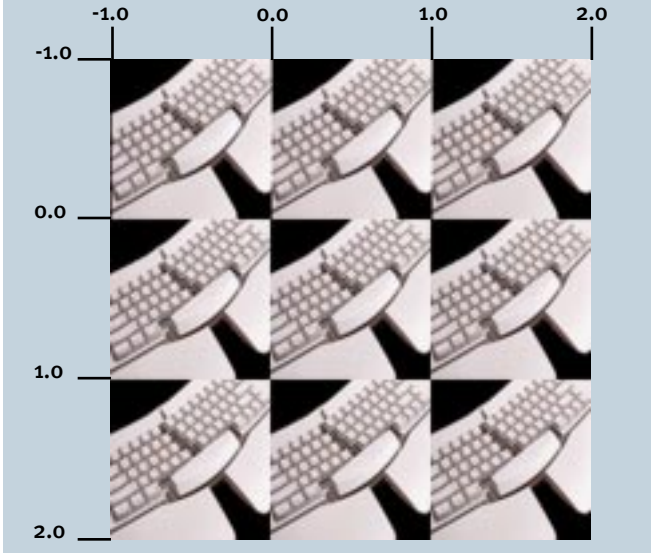


FIGURE 12. *Clamping the texture coordinates can cause strange display effects.*

FIGURE 13. *The bug in Figure 12 is due to this clipping of the character into the shadow texture.*

2. Create a texture using the result of Step 1.
3. Project the texture by calculating (s,t) coordinates for each object onto which we want the shadow to be projected.
4. Render the scene from the camera's point of view.
5. Render the triangles that are visible from the light's point of view using the (s,t) calculated during Step 3 and the texture from Step 2.

This is the basic method behind shadow projection. Now let's look at each step in detail.

STEP 1. We have to render the figure of the object that will cast a shadow in our scene. We'll need to transform, project, and render the object into the light's point of view. (A flat rendering is sufficient to achieve a simple shadow, but you can imagine a lot of special effects that could be implemented with this technique.) We can speed up this operation by using a simplified mesh (a level of detail mesh, for example); the effect on the final result won't be too great, especially if the object is animated (a character, for example). The drawing of the triangles is pretty fast because of the simplicity of a flat rendering (particularly for software rasterizers). The fastest rendering, of course, can be achieved with the help of a 3D accelerator. Because we're only interested in a figure of the object, we can deactivate the Z-buffer or any other sorting technique — the result will be the same, and the overall performance of this step will improve. As usual, any optimization aimed at reducing the number of triangles to be rendered would help speed things up (backface culling and such).

STEP 2. Copying the rendered figure from Step 1 into a texture is currently the slowest part of the sample application, due to the slowness of the LFB (linear frame buffer) access on the Voodoo chipset. Normally, this operation should be a lot faster. Some chips allow hardware blitting between the frame buffer and texture memory. Better still, some boards — such as the 3Dfx Voodoo Banshee — let you render directly into texture memory. In the latter two cases, you can consider this step as almost free, and expect a gain of 50 percent in performance for the sample application provided with this article.

STEP 3. Now, we transform all of the objects upon which we want to cast the shadow into the light's point of view. This is a good time to flag triangles that aren't visible in the light's point of view, so that we can reject them during Step 5. We then calculate the new (s,t) coordinates to perform the projection of the shadow. The speed of this step depends on the number of vertices in the mesh. In general, the complexity of the mesh isn't detrimental to the final frame rate. This step should typically account for about 15 to 20 percent of the overall frame. So, although it's not a critical issue, one could easily speed up this process. For example, we could use occlusion techniques (potentially visible sets, portals, and so on) that will decrease the number of vertices to process in the scene. Backface-culling can also divide the number of vertices by two for most objects.

STEP 4. We render the whole scene as we normally would. We don't need to worry about shadows for now.

STEP 5. Now we're going to render all the triangles that we determined were visible in the light's point of view during Step 3 (Figure 6). We'll texture these triangles with the texture map produced in Step 2 and we'll use the texture coordinates computed in Step 3. All of these triangles should have been rendered in Step 4. We are therefore using multipass rendering to render them a second time for the final result. Figure 14 shows the triangles that have been rendered twice to perform a multipass rendering. To setup multipass rendering, we have to configure the Z-buffer or W-buffer in Less or Equal comparison mode or the second pass (Step 5) won't be visible because the pixels will be rejected by the depth-sorting.

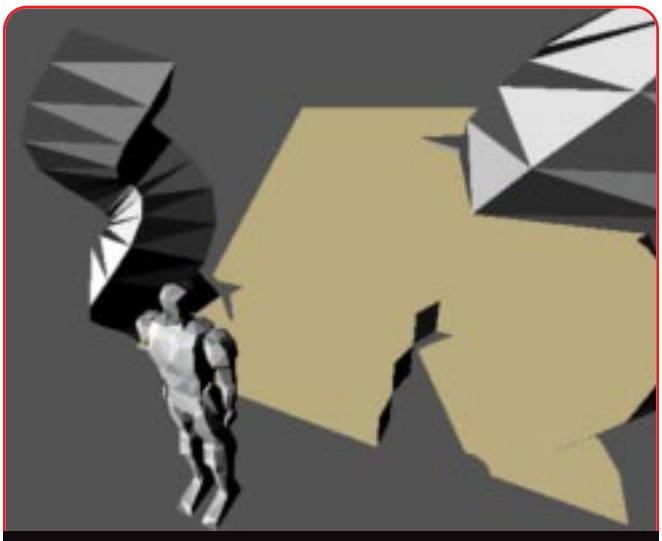


FIGURE 14. *The triangles that are visible in the light's field of view are displayed with colored flat rendering.*

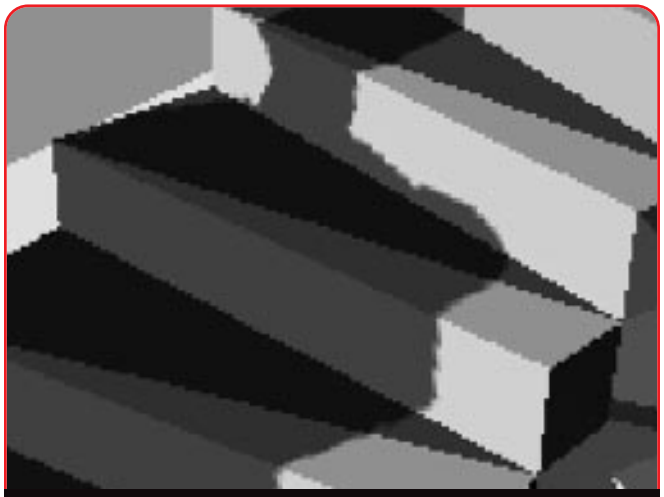


FIGURE 15. *Using an alpha shadow texture allows filtering on the edges of our shadow.*

FIGURE 16. *A RGB 565 pixel.*

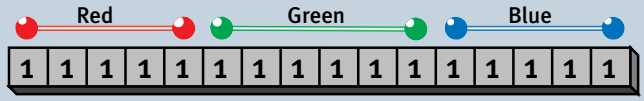
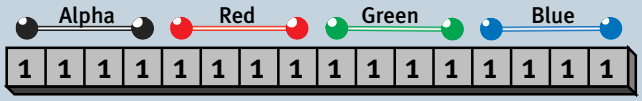


FIGURE 17. *A 16-bit pixel again, but in ARGB 4444 format.*



We also need to set our hardware to skip pixels that are of the background color in the shadow texture. The simplest way to do this is to render the shadow with an RGB color of 0x00000001 (ARGB value) on a 0x00000000 background. To reject the background but not the shadow, we can use the **grChromakeyValue()** function (or its equivalent in whatever API you happen to be using) to configure the rasterization device to reject the background pixels. However, the chroma-keying solution doesn't take advantage of bilinear filtering, which helps smooth out the otherwise pixilated texture.

A still better technique is to use a texture with an alpha channel (ARGB-4444 format in most hardware; I'll explain later how I generate an ARGB-4444 texture in a 565 frame buffer). We can use the alpha value in the texture to discard pixels of the background color. The point in using an alpha texture is to take advantage of filtering to obtain smoother edges. The alpha value will be interpolated in the same way as the colors. The edges of our shadows will appear smoother than if we'd used a simple color-key test. In some cases, we can even get an effect that resembles antialiasing. Figure 15 demonstrates the filtering capabilities of this technique. We can get sharper edges by increasing the size of the shadow texture, although this will slow down its generation. With higher fill-rates and Unified Memory Architecture (as on the Banshee, for example) the texture's size will be less of an issue.

Listing 3 shows how to configure Glide before drawing the shadow. Listing 4 shows the settings, which we perform once at the beginning of the program. Some of these may influence the rendering of the shadow texture.

Odds and Ends

The alpha combine is set to use both texture and iterated alpha at the same time to adjust the transparency of the shadow. The alpha of the texture encodes the mask of the shadow mask, and the iterated alpha allows for dynamically adjusting the transparency of the shadow. Using different degrees of transparency allows us to simulate variable light intensity. For example, if the spotlight is near the object, we would set the iterated alpha to 255.0 to get a black (opaque) shadow. On the opposite end of the spectrum, if the spotlight is far off or the light intensity is low, we would set alpha closer to 0.0 to get a very soft shadow.

I'm currently using the same alpha value for all vertices, but one could consider using more exotic per-vertex alpha calculation (such as computing an alpha value that is a function of the distance between the vertex and the light source).

Previously, I talked about generating an ARGB 4444 texture from a RGB 565 frame buffer. This is a simple trick. We need a black texture with an alpha mask representing the figure of our object. I have a RGB 565 texture. Figure

LISTING 3. *Configuring Glide to draw the shadow (continued on page 52).*

```
void DrawShadow(Mesh &m)
{
   guAlphaSource( GR_ALPHASOURCE_TEXTURE_ALPHA_TIMES_ITERATED_ALPHA );

   grAlphaBlendFunction( GR_BLEND_SRC_ALPHA,
         GR_BLEND_ONE_MINUS_SRC_ALPHA,
         GR_BLEND_ONE,
         GR_BLEND_ZERO );

   grTexCombineFunction(GR_TMU0, GR_TEXTURECOMBINE_ADD);
   guColorCombineFunction(GR_COLORCOMBINE_TEXTURE_TIMES_ITRGB);
   grColorCombine(
      GR_COMBINE_FUNCTION_SCALE_OTHER,
      GR_COMBINE_FACTOR_LOCAL,
      GR_COMBINE_LOCAL_ITERATED,
      GR_COMBINE_OTHER_TEXTURE,
      FXFALSE);

   grTexClampMode(GR_TMU0,GR_TEXTURECLAMP_CLAMP,GR_TEXTURECLAMP_CLAMP);
```

LISTING 3. *Configuring Glide to draw the shadow (continued from page 51).*

```
   for (int i=0; i<m.nTriangles; i++)
   {
      int i1 = m.TriangleArray[i*3+0];
      int i2 = m.TriangleArray[i*3+1];
      int i3 = m.TriangleArray[i*3+2];

      if ( m.FlagArray[i] == 0) continue;

      if (
         ((m.VertexArray[ i1 ].flags | m.VertexArray[ i2 ].flags | m.VertexArray[ i3
].flags)
         & Vertex::CLIPZ) == 0)
      {
         v1.x    = m.VertexArray[ i1 ].px;
         v1.y    = m.VertexArray[ i1 ].py;
         v1.oow  = m.VertexArray[ i1 ].pz;
         v1.tmuvtx[0].sow = m.VertexArray[ i1 ].u * v1.oow;
         v1.tmuvtx[0].tow = m.VertexArray[ i1 ].v * v1.oow;
         v1.r = 255.0f;
         v1.g = 255.0f;
         v1.b = 255.0f;

         v2.x    = m.VertexArray[ i2 ].px;
         v2.y    = m.VertexArray[ i2 ].py;
         v2.oow  = m.VertexArray[ i2 ].pz;
         v2.tmuvtx[0].sow = m.VertexArray[ i2 ].u * v2.oow;
         v2.tmuvtx[0].tow = m.VertexArray[ i2 ].v * v2.oow;
         v2.r = 255.0f;
         v2.g = 255.0f;
         v2.b = 255.0f;

         v3.x    = m.VertexArray[ i3 ].px;
         v3.y    = m.VertexArray[ i3 ].py;
         v3.oow  = m.VertexArray[ i3 ].pz;
         v3.tmuvtx[0].sow = m.VertexArray[ i3 ].u * v3.oow;
         v3.tmuvtx[0].tow = m.VertexArray[ i3 ].v * v3.oow;
         v3.r = 255.0f;
         v3.g = 255.0f;
         v3.b = 255.0f;

         v1.a = 170.0f;// use the "Iterated Alpha" to adjust the
         v2.a = 170.0f;    // "transparency" of the shadow for performing
         v3.a = 170.0f;// special effects like gradients and so on...

         guDrawTriangleWithClip( &v1, &v2, &v3);
      }
   }
     grColorCombine( GR_COMBINE_FUNCTION_LOCAL,   // turn OFF the "texture mapping"
                 GR_COMBINE_FACTOR_NONE,
                 GR_COMBINE_LOCAL_CONSTANT,
     GR_COMBINE_OTHER_NONE,
                 FXFALSE );

   grAlphaBlendFunction( GR_BLEND_ONE,         // disable the "alpha blending"
         GR_BLEND_ZERO,
         GR_BLEND_ONE,
         GR_BLEND_ZERO );
}
```


16 shows the RGB values in a 16-bit 565 pixel. Figure 17 shows the same thing in an ARGB 4444 pixel. We need an alpha mask with a value of 1111b for each pixel. If we compare the bits of the 565 texture, it corresponds to a part of the 565-Red, 11110b exactly. So, all we need to do is to render a flat shape with the value 1111000000000000b and copy it as an ARGB 4444 texture. This gives us an alpha mask with all alpha pixels with a value of 1111b. Listing 5 shows how it's written the sample application.

The **grTexDownloadMipMap()** function loads the memory from the ShadowBitmap buffer into the Voodoo texture memory. Because 4444 and 565 textures are both 16 bits per pixel, the function doesn't care about how bits are organized inside. It copies a block of memory to the address specified on the board (as it happens here, zero). The interesting part of this operation is that we can set up the hardware for a specific texture format with the **grTexSource()** function. All we have to do is to fill the **GrTexInfo** structure with the constant **GR_TEXFMT_ARGB_4444** for the format member instead of **GR_TEXFMT_RGB_565**. The function will configure the texture mapping unit to decode an ARGB 4444 texture. Most boards now support this format, so don't worry about the compatibility of this technique.

Pros and Cons

As good as it is, this technique is far from perfect. Its most significant drawback is the low accuracy of the shadow (As compared to a polygonal shadow, which has sharper edges). In order to increase the shadow's accuracy, we have to increase the size of the shadow texture. Changing the size of the shadow texture from 256×256 to 512×512 would multiply the number of texels by a factor of four, which is significant with actual rasterizers (software or hardware), but is still possible. This is the main inconvenience of any sampling technique.

Furthermore, because we need a direction for doing the projection in the light's point of view, this technique can only handle directional lights. Omni lights, which cast light in every direction, cannot be used. The closest we could get to simulating an omni

light would be to create shadow maps for each object in the scene by pointing the spot towards each of them, and later compounding the effect of all the shadow maps on each object. This approach could be quite costly and may not be practical. Another constraint is the distance from the light to the object. Remember that we need to avoid clipping the object against the border of the texture. Thus, we can't place the object and the light too closely to one another. We must maintain a minimum distance between the light and the objects casting the shadows.

But this technique does offer significant advantages. The most important of these is its simplicity. Any 3D programmer could implement this method in a very short time. There are no complex mathematics or data to manipulate. It's just an intuitive technique that can provide great effects. And it's fast. Rendering the shadow texture doesn't cost much CPU time if it's handled by a hardware accelerator. A hardware **blit()** between the frame buffer and the texture memory will significantly speed up our shadow casting operation. We can even improve the performance by using a simpler mesh, although generating shadows from complex objects only slows down Step 1. The rest of the code (drawing the shadow) will run at exactly the same speed regardless of the complexity of the original mesh. In comparison, polygon-based shadow techniques take a bigger performance hit for higher-complexity objects.

The shadow map texture is a scalable technique. For scenes in which multiple objects are casting shadows, we can use differently sized texture shadow maps. For the main object, or for the object that is closest to the camera, we would use a high-resolution texture shadow map (512×512) and scale down to 32×32 or smaller for less important objects or objects that are farther away from the camera (small on the screen). It's analogous to an LOD technique, applied to the shadow texture. Finally, certain special effects could be based on this technique. One of these might consist of using opacity maps to create holes in the shadow maps (skeletons are a typical case). I trust *Game Developer*'s readers to adapt this technique to their needs, and I look forward seeing good-looking shadows in future games. ■

Acknowledgements

The author would like to thank the following people for their great support : Eliane Fiolet, Eric Smolikowski, Xavier Gerbier, Alexandre Macris, Miky Larsen.

Special thanks to Denis Amselem at 3Dfx for Glide support.

LISTING 4. *Our Glide settings.*

```
grCullMode(GR_CULL_POSITIVE);
grDepthBufferMode( GR_DEPTHBUFFER_WBUFFER );
grDepthBufferFunction( GR_CMP_LEQUAL );
grDepthMask( FXTRUE );
grTexFilterMode(GR_TMU0,GR_TEXTUREFILTER_BILINEAR,GR_TEXTUREFILTER_BILINEAR);
grGammaCorrectionValue((float)0.8);
grDitherMode(GR_DITHER_DISABLE);
```

LISTING 5. *Generating an ARGB 4444 texture from a RGB 565 frame buffer.*

```
// STEP-2
// read the rendered picture from STEP-1
// and use it as a texture
//-----------------------------------------
grLfbReadRegion(GR_BUFFER_BACKBUFFER,    // copy from the LFB to a buffer in
          0,                             // x                // SYSTEM MEMORY
          0,                             // y
          256,                // width
          256,                // height
          256*2,              // stride in bytes
          ShadowBitmap
          );
LoadTextureOnVoodoo();                              // download from the buffer in SYSTEM
MEMORY

This reads the 565 framebuffer in a system memory buffer allocated by the program, but the
real trick is in the LoadTextureOnVoodoo() function.

void LoadTextureOnVoodoo(void)
{
          GrTexInfo info;

          info.smallLod        = GR_LOD_256;
          info.largeLod        = GR_LOD_256;
          info.aspectRatio     = GR_ASPECT_1x1;
          info.format          = GR_TEXFMT_ARGB_4444;
          info.data            = ShadowBitmap;

          grTexDownloadMipMap(
                    GR_TMU0,
                    0,                                          // start addr
                    GR_MIPMAPLEVELMASK_BOTH,
                    &info
                    );

          // this should be done only once only since
          // I always use the same adress and texture format
          grTexSource(GR_TMU0,
                    0,                                          // startaddr
                    GR_MIPMAPLEVELMASK_BOTH,
                    &info
                    );
}
```