5.8

Real-Time Shadows on Complex Objects

Gabor Nagy

This article presents an efficient algorithm capable of creating realistic shadows in real-time applications. The algorithm can take advantage of today's fast texture-mapping and 3D transformation hardware.

Introduction

Shadows are among the most important depth cues in human vision. In computer graphics, they can give an image the final touch of realism. Without shadows—even with realistic lighting and texturing effects—computer-generated images look artificial; the objects appear to float in space, even when they are lying on a surface. This inability for humans to sense the relative position and depth in computer graphics is especially apparent when the camera is not moving (no parallax information).

Until recently, only computationally expensive algorithms such as ray tracing and radiosity could produce accurate shadows, in which both the objects casting the shadows and the ones receiving them are of arbitrary complexity.

The algorithm presented here is optimized for real-time applications. It provides a very good balance between realism and rendering performance while being easily extendible to all situations. With the always performance-hungry game programmer in mind, this article highlights points at which significant optimization for performance is possible using hardware features.

Some of the basic ideas in this article have been around for a while, but most papers describing them don't deal with the important implementation details we cover here.

The Light Source, Blocker Object, and Receiver Object

Consider the simple example shown in Figure 5.8.1. The torus (*blocker object* or *blocker*) blocks some of the light coming from the light source, casting a shadow on

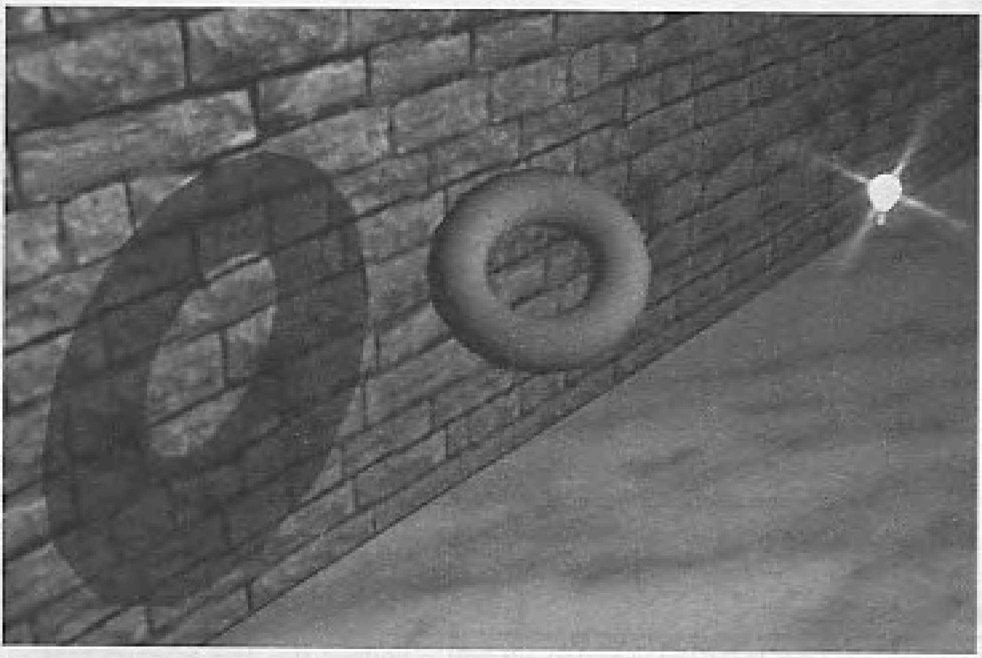

FIGURE 5.8.1. Shadow, receiver, blocker, and light source.

the wall. The wall receives the shadow, or "lack of light"; therefore it is called the *receiver object*, or *receiver*.

If the light source is a point light (infinitely small), the blocker object blocks the light of that light source in a well-defined volume, usually referred to as a *shadow volume* (see Figure 5.8.2). A shadow is created on a receiver object where its surface intersects with the shadow volume. As Figure 5.8.2 shows, the shadow volume has a truncated, cone-like shape, starting at the blocker object and continuing to infinity. Whereas the shadow volume really starts at the contours of the blocker object, its cone-like shape originates from the light source.

Let's examine how the cross section of the shadow volume changes as we get further from the light source. We call the point on the blocker's surface that is nearest to the light source P_n and the one farthest from it P_f.

We can divide the shadow volume into three regions:

1. Between the light source and P_n
2. Between P_n and P_f
3. From P_f to infinity

It's easy to see that in Region 3, the cross-section of the shadow volume has a constant shape, but it increases in size as we get further from the light source.

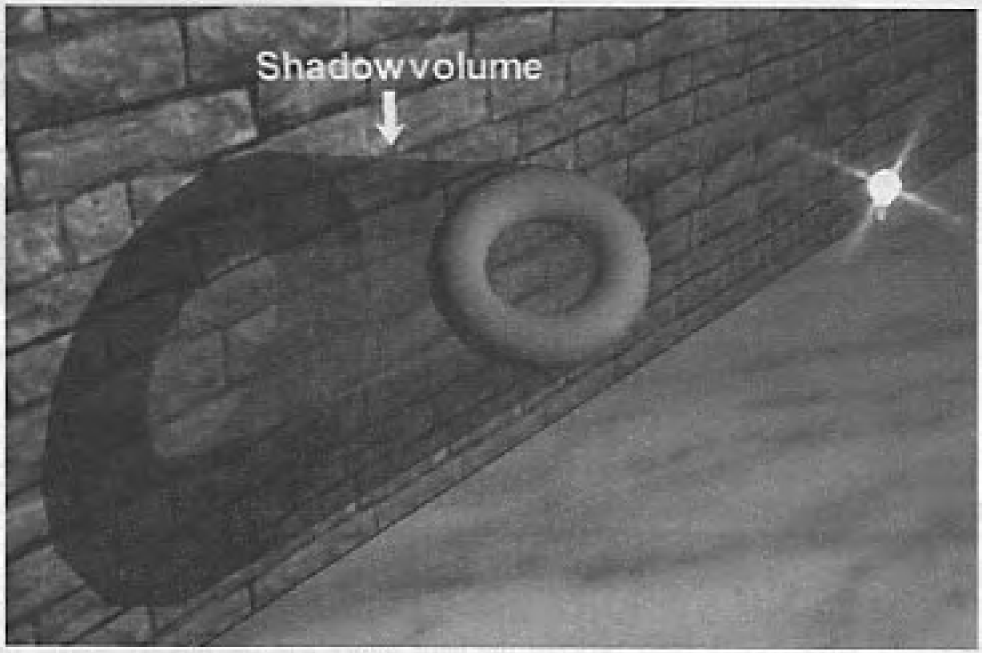

FIGURE 5.8.2. The shadow volume.

Because of this phenomenon, unless one or more receiver objects are in Region 2, the shadow volume can be accurately modeled by projecting a two-dimensional mask from the position of the point light source. Consequently, using the same projection, we can map this 2D mask on the receiver objects to define the shadowed areas! This 2D mask is called the *shadow map*, and it can be simply derived by drawing the blocker object's silhouette as seen from the light source.

Notice that we cannot see the shadow cast by the torus in Figure 5.8.3a because the torus exactly obscures it! This is a good indication that indeed, we can simply use a properly projected 2D image or mask (see Figure 5.8.3b) to define the shadow volume. This method is usually referred to as *projective shadow mapping*.

The Objectives of This Article

To draw shadows using the method introduced, we need to do the following:

1. Create a shadow map for each light/blocker object pair.
2. Calculate the shadow map (texture) coordinates to use on the receiver object's vertices.
3. Render the receiver objects with the shadow map applied as a 2D texture.

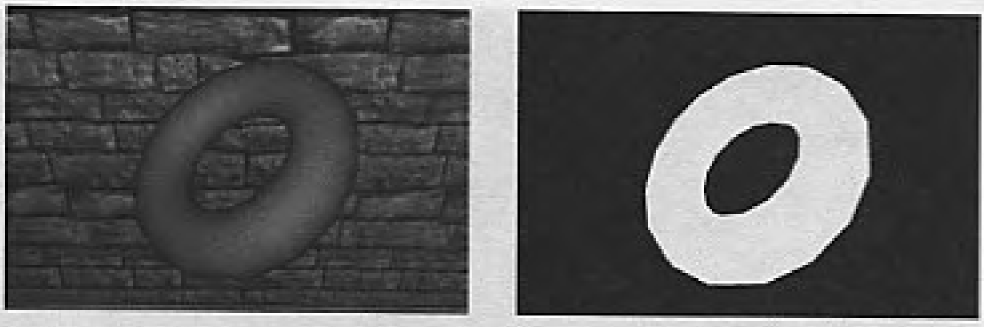

FIGURE 5.8.3. The blocker object as seen from *a*, the light source, and *b*, its silhouette.

Creating the Shadow Map

The first thing to do in order to render the shadow map is set up a perspective projection originating at the light source. This projection projects the blocker object onto a *virtual screen plane* between the light source and the blocker object, yielding the shadow map shown in Figure 5.8.4.

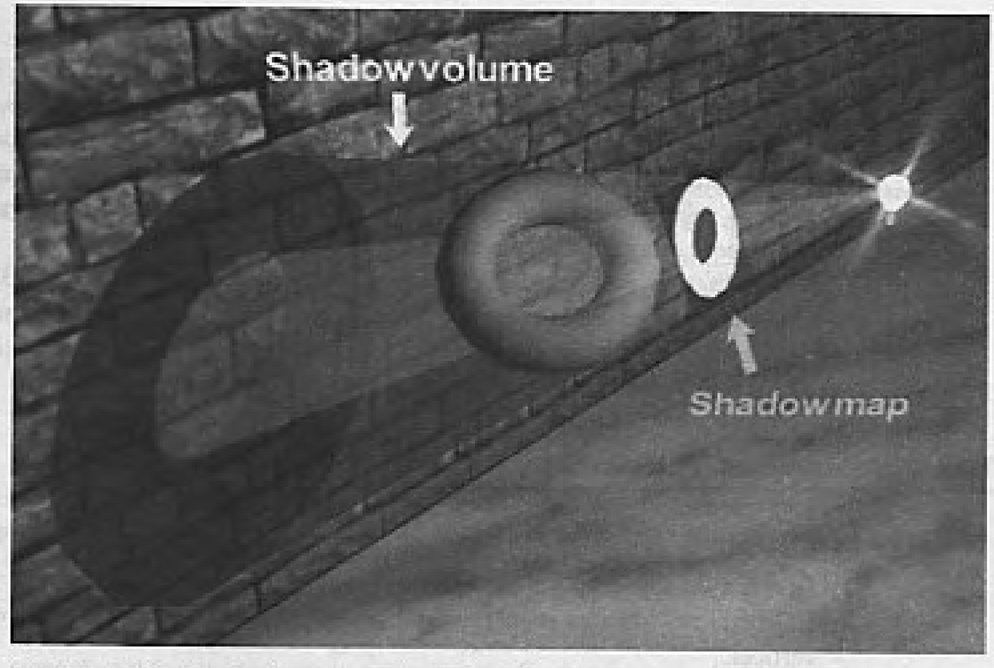

FIGURE 5.8.4. The shadow map projection (also Color Plate 6).

The Light Coordinate System

First, we define a new coordinate system with its origin at the light source and its *Z*-axis pointing at the blocker object. The *Z*-axis of this coordinate system determines the center line of the perspective projection, while its *XY*-plane defines the orientation of the screen plane on which we project the shadow map. If we transform the blocker object into this *light coordinate system*, illustrated in Figure 5.8.5., we can easily project it onto this plane.

To define an arbitrary coordinate system, we need to know the position of its origin and its orientation. We already know the position of the origin: It's the position of our light source. We can describe the orientation of the light coordinate system by the direction of its three axes: X_{light}, Y_{light} and Z_{light}, all 3D unit vectors in *world* coordinates.

Finding Z_{light}

Starting with the Z_{light} axis, we can easily find X_{light} and Y_{light}. Z_{light} is a direction vector that starts from the light source and points at the blocker object. Let's assume that the blocker object is polygonal, and we have an array of all the polygon vertices that are used in rendering this object. Now we have a set of "target" points in 3D space (the vertices of the blocker) and another point: the position of the light source. A fast and

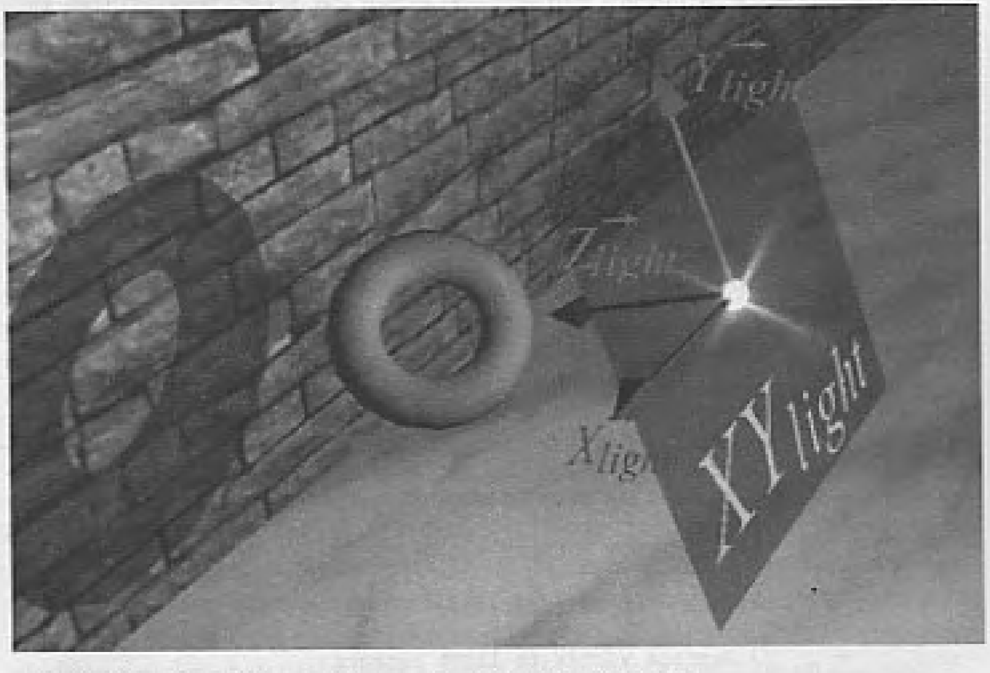

FIGURE 5.8.5. The light coordinate-system (also Color Plate 7).

efficient way to obtain a "good" direction vector is to average the vectors that start at the light source and point to each vertex. (We discuss a better approach later.)

We call the vector we have computed the *mean direction vector*, or MDV:

$$M\bar{D}V = \frac{\sum_{i=1}^{N_v}\left(V_i - P_{light}\right)}{N_v}$$

where N_v is the number of vertices considered in the blocker and P_{light} is the position of the light source.

Normalizing *MDV* yields Z_{light}:

$$\bar{Z}_{light} = \frac{M\bar{D}V}{\left|M\bar{D}V\right|}$$

Optimization Tip #1

Since we will normalize *MDV*, we don't have to divide the sum of light-to-vertex vectors by N_v, saving one divide operation. We can also calculate $P_{light}{}^{*}N_v$ in advance and avoid the $-P_{light}$ in the vertex loop, because:

$$\frac{\sum_{i=1}^{N_v}\left(V_i - P_{light}\right)}{N_v} = -P_{light}N_v\,\frac{\sum_{i=1}^{N_v} V_i}{N_v}$$

Here is the C code to compute Z_{light}:

```
typedef struct
{
  E3dType  X,Y,Z;
  short Flags;
} E3dVertex;

void ShadowMatrix(Matrix LBlockerLocalToWorldMatrix)
{
  unsigned long   LVn, LVC, LN, LC;
  float        Mx, My, Mz, LPlightX, LPlightY, LPlightZ,
  float        LMDVX, LMDVY, LMDVZ,   // Mean Direction Vector
  float        LZlightX, LZlightY, LZlightZ, // Zlight vector
  // Initialize Mean Direction Vector to (0.0, 0.0, 0.0)
  //
  LMDVX = LMDVY = LMDVZ = 0.0;

  Lvertex = LMesh->Vertices;

  // Average vertex-to-light vectors
  //
```

```
    LVn = LMesh->NumOfVertices;

    LMDVX = LPlightX * LVn;
    LMDVY = LPlightY * LVn;
    LMDVZ = LPlightZ * LVn;

    for(LVC = 0;L VC < LVn; LVC++, LVertex++)
    {
      Mx=LVertex->X; My=LVertex->Y; Mz=LVertex->Z;
      E3dM_MatrixTransform3x4(LBlockerLocalToWorldMatrix, LX, LY, LZ);

      LMDVX -= LX;
      LMDVY -= LY;
      LMDVZ -= LZ;
    }

    // Normalize Mean Direction Vector (MDV)
    //
    LVF = sqrt(LMDVX*LMDVX+LMDVY*LMDVY+LMDVZ*LMDVZ);
    LVF = 1.0 / LVF; // We can save 2 divisions by doing this in
                     // advance...

    LZlightX = LMDVX * LVF;
    LZlightY = LMDVY * LVF;
    LZlightZ = LMDVZ * LVF;
    …
  }
```

`E3dM_MatrixTransform3x4` is a macro function that transforms a 3D vector given by *Mx*, *My*, and *Mz* with a 3×4 matrix (actually the top-left part of a 4×4 matrix).

For the rest of the source code, please refer to the example program on this book's companion CD-ROM.

Finding X_{light} and Y_{light}

The projection to map the shadow-map texture on the receiver object is the same as the one used to draw the shadow map; therefore, the orientation of the shadow map (rotation around the Z_{light} axis) does not matter. In other words, rotating the *XY-light* plane around Z_{light} does not make any difference. This means that for the X_{light} axis, we can use any unit vector that is perpendicular to Z_{light} (refer back to Figure 5.8.5). We can get that vector as the cross-product of Z_{light} and any other vector that is not parallel with Z_{light}. Let's call this "helper" vector *V*.

We know that at least two of the *X*, *Y*, or *Z*-axes of the *world* coordinate system meet these criteria, so for simplicity, we use a unit vector *V(x,y,z)*, with one coordinate being 1, the others 0.

A vector's largest component (*X*, *Y*, or *Z*) determines its dominant direction; therefore, to get a vector that points far enough away from Z_{light}, we set to 1 the component of *V* that has the *smallest* absolute value in Z_{light}. This eliminates floating-point precision worries when performing a vector cross-product operation on Z_{light} and *V*.

For example:

If Z_{light}=(0.381, 0.889, 0.254), V will be: (0.0, 0.0, 1.0) = Z_{world}

If Z_{light}=(–0.889, 0.254, 0.381), V will be: (0.0, 1.0, 0.0) = Y_{world}

and so on.

The cross-product of Z_{light} and V yields a third vector that is perpendicular to both of them. After normalization, this vector yields X_{light}, the X-axis of the light coordinate system:

$$\vec{X}_{light} = \frac{\vec{Z}_{light} x \vec{V}}{\left|\vec{Z}_{light} x \vec{V}\right|}$$

With X_{light} and Z_{light} given, the Y_{light} axis is just another cross product away:

$$\vec{Y}_{light} = \vec{X}_{light} x \vec{Z}_{light}$$

Note that this step gives us a unit vector, so we don't have to normalize Y_{light}, since:

$$\left|\vec{X}_{light}\right| = 1 \text{ and } \left|\vec{Z}_{light}\right| = 1 \text{ and } \vec{X}_{light} \perp \vec{Z}_{light} \Rightarrow \left|\vec{X}_{light} \times \vec{Z}_{light}\right| = 1$$

With X_{light}, Y_{light} and Z_{light} and P_{light} known, we can create the matrix that transforms a point from *world* coordinates to *light* coordinates by simply filling in these values:

$$M_{WorldToLight} = \begin{bmatrix} X \text{ of } \vec{X}_{light} & X \text{ of } \vec{Y}_{light} & X \text{ of } \vec{Z}_{light} & 0.0 \\ Y \text{ of } \vec{X}_{light} & Y \text{ of } \vec{Y}_{light} & Y \text{ of } \vec{Z}_{light} & 0.0 \\ Z \text{ of } \vec{X}_{light} & Z \text{ of } \vec{Y}_{light} & Z \text{ of } \vec{Z}_{light} & 0.0 \\ -X \text{ of } P_{light} & -Y \text{ of } P_{light} & -Z \text{ of } P_{light} & 1.0 \end{bmatrix}$$

This is why we used X_{light}, Y_{light}, and Z_{light} to describe the orientation of the light coordinate system.

The next step is to pre-multiply this matrix with the blocker object's *local-to-world* matrix. This step gives us the *local-to-light* matrix for the blocker.

$$M_{BlockerLocalToLight} = M_{BlockerLocalToWorld} * M_{WorldToLight}$$

As the name implies, the matrix transforms a point defined in the *local* coordinate system of the blocker into the light coordinate system. Such transformed X and Y coordinates define the *parallel* or *orthogonal* projection of the blocker object onto the shadow-map plane (which is parallel with the XY plane of the light coordinate system).

Defining the Perspective Projection

To make this a perspective projection, we need a field of view, or the X and Y "projection ratios." We can find the projection ratios (R_X and R_Y) for each vertex of the blocker object by transforming the vertex with $M_{BlockerLocalToLight}$ and dividing the resultant X and Y coordinates by the Z coordinate:

$$R_Y = \left|\frac{Vtx_Y}{Vtx_Z}\right|$$

Adaptive Projection

We could always use the same ratio for the projection, but that would lead to the blocker object's silhouette changing size in the shadow map if we move the light source closer or farther away from it. The same problem arises if we change the size of the blocker or if the light source looked at it from a different angle. These changes could result in a tiny image of the blocker in the middle of the shadow map or an oversized image that doesn't fit in the shadow map (see Figure 5.8.6). In the first case (Figure 5.8.6a), we get a low-resolution shadow map with bad artifacts on the receiver objects. This is a waste of shadow-map memory.

The latter (Figure 5.8.6b) causes incorrect shadow shapes and possibly "shadow leaking" (see the section "Texture Coordinates and Shadow-Map Coordinates"). In this case, the shadow-map size is not large enough.

Instead of one fixed value, we use the largest R_X (R_{Xmax}) as the horizontal ratio for the projection, whereas the largest R_Y, (R_{Ymax}) gives the vertical ratio. This makes the perspective projection *adaptive* for both the X and Y direction, meaning that the blocker's silhouette always properly fills the shadow map, making the best use of all the pixels in it.

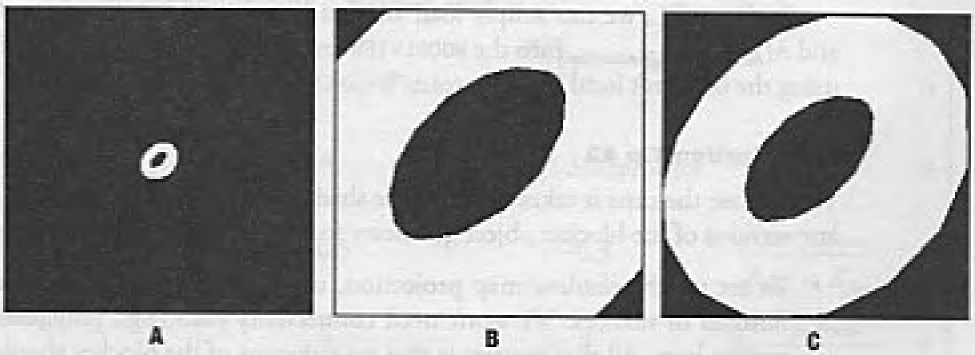

FIGURE 5.8.6. Non-adaptive (*a* and *b*) and adaptive blocker projection (*c*).

This concept is very important because we want to use the minimum necessary texture size, for the following reasons:

- Texture memory is always a scarce resource, and the maximum texture size might be limited by other factors.
- On some hardware, after drawing the shadow map image, we have to transfer it from the frame buffer to a dedicated texture memory, and the speed of this transfer is limited by bus and memory bandwidth.

Now we can fill out a standard perspective projection matrix for the blocker object:

$$M_{BlockerProjection} = \begin{bmatrix} \frac{0.98}{R_{X\max}} * SMapWidth & 0 & 0 & 0 \\ 0 & \frac{0.98}{R_{Y\max}} * SMapHeight & 0 & 0 \\ 0 & 0 & \frac{Z_{far} + Z_{near}}{Z_{near} - Z_{far}} & -1 \\ 0 & 0 & 2\frac{Z_{far}Z_{near}}{Z_{near} - Z_{far}} & 0 \end{bmatrix}$$

where *SMapWidth* and *SMapHeight* are the horizontal and vertical resolutions of the shadow map in pixels. Z_{near} and Z_{far} are the distances of the near and far clipping planes of the viewing frustum from the light source.

Pre-multiplying this matrix with $M_{BlockerLocalToLight}$ yields the 4 × 4 matrix that performs a perspective projection from blocker local coordinates to shadow-map coordinates:

$$M_{BlockerLocalToShadowMap} = M_{BlockerLocalToLight} \cdot M_{BlockerProjection}$$

In OpenGL, we can simply load an identity matrix into the `PROJECTION` matrix and $M_{BlockerLocalToShadowMap}$ into the `MODELVIEW` matrix and start drawing the shadow map using the blocker's local coordinates.

Optimization Tip #2

To decrease the time it takes to create the shadow map, we can use two or three different versions of the blocker object geometry for the various rendering stages:

- To set up the shadow map projection, use blocker geometry with a minimum number of vertices. We won't need connectivity data (e.g., polygons) or normal vectors here. All that matters is that no polygons of the blocker should be outside the projected shape of this volume, no matter the angle from which we look at it, because that would draw on the one-pixel edge of the shadow map, ruining texture clamping and causing "shadow leaking." We might even use a "good" bounding volume such as the blocker's bounding-box, which could eliminate the need to compute *MDV* (just use a vector from the light source to the center of the bounding box).

- To draw the shadow map, use blocker geometry with a minimum or no *surface detail* but necessary *contour detail*.
- And, of course, to draw the blocker object, we need the geometry with all the surface detail and surface properties (e.g., normal vectors) to make the object look spiffy.

Optimization Tip #3

If the rendering engine is programmable, we can use a very simple (and possibly fast) renderer code to draw the shadow map:

- No lighting needed; a simple "flat-color" renderer will do.
- No clipping needed (the blocker's image always fits in the shadow map!).
- No depth testing (Z-buffering) needed.

Projecting the Shadow Map on a Receiver Object

Now we have a shadow map associated with a blocker object and a light source. This shadow map can be projected on any number of receiver objects, and because it is applied as a texture, the receiver objects can have any complex shape (curves, holes, ridges, and so on).

As mentioned before, we use the same projection to project the shadow map on a receiver as we used to create the shadow map. The only differences are the image offset and scaling factors, because we use the [0..1] coordinate range as opposed to the [0..*SmapWidth*] or [0..*SmapHeight*] ranges.

This is the appropriate projection matrix:

$$M_{BlockerProjection} = \begin{bmatrix} \frac{0.49}{R_{X\,max}} * SMapWidth & 0 & 0 & 0 \\ 0 & \frac{0.49}{R_{Y\,max}} * SMapHeight & 0 & 0 \\ -0.5 & -0.5 & \frac{Z_{far} + Z_{near}}{Z_{near} - Z_{far}} & -1 \\ 0 & 0 & 2\frac{Z_{far} Z_{near}}{Z_{near} - Z_{far}} & 0 \end{bmatrix}$$

The next thing to do is to pre-multiply this matrix with $M_{WorldToLight}$:

$$M_{WorldToShadowMapST} = M_{WorldToLight} * M_{ReceiverProjection}$$

The resulting matrix transforms a point from world coordinates to shadow-map texture coordinates (the resulting *X* and *Y* give *S* and *T*, respectively).

Pre-multiplying $M_{WorldToShadowMapST}$ with the receiver's local-to-world matrix yields the matrix that we need to go from receiver local space directly to shadow-map texture space.

$$M_{WorldToShadowMapST} = M_{WorldToLight} * M_{ReceiverProjection}$$

Texture Coordinates and Shadow-Map Coordinates

The shadow map is an image with a finite number of pixels and integer coordinate values—for example, 256 × 256. However, texture coordinates are usually normalized floating-point values, meaning that the range [0..1] refers to pixel coordinates [0..255] horizontally and [0..255] vertically. So what happens outside the [0..1] range? We have to make sure that the texel used on the receiver is the color used for "no shadow" (black in Figure 5.8.7).

On most 3D hardware with texture mapping, you have at least two options:

- **Texture repeat.** Outside the [0..1] range, the texture is simply repeated—so, for example, in the [–1..0] range of texture coordinates, the texture produces the same image as the [0..1] range.
- **Texture clamping.** The pixel on the edge of the texture image is repeated everywhere outside the [0..1] range, or you can define a specific "border color" that is repeated outside the normal range.

It's easy to see that we have to use texture clamping because we want a uniform effect on the receiver object outside the [0..1] texture coordinate range.

Texture clamping effectively saves us from testing the receiver objects for intersection with the shadow volume. Because not all 3D hardware and APIs provide a separate texture border color, we have to leave a one-pixel-thick border on the shadow map. If we accidentally draw in this border, that pattern would be repeated on the receiver, producing a leaking effect ("shadow leaking"). To make sure that nothing is drawn in this border, we have to slightly decrease the *X* and *Y* scaling factors in the projection matrices (elements (0, 0) and (1,1)). This is the reason for using the value 0.98 (instead of 1.0) in $M_{BlockerProjection}$ and 0.49 (instead of 0.5) in $M_{ReceiverProjection}$. Note that these values depend on the resolution of the shadow map. (See the example program on the CD for the proper formulas to calculate them.)

Rendering the Receiver Objects

There are many different ways to draw the object receiving the shadow. The two most common methods are:

- **Single-pass rendering.** If there is no other texture on the receiver object, we can draw it in one pass, applying a black-on-white shadow map as a texture and using the light source to illuminate the object.
- **Multipass rendering with subtractive blending.** If a receiver already has a texture on it and the hardware doesn't support multitexturing, we need multiple passes:
  1. Draw the receiver object normally.
  2. Draw the shadow pass with subtractive pixel blending, using a white-on-black shadow map. This successively decreases the surface color intensity where there is a shadow cast. Use "GREATER-OR-EQUAL" or "LESS-THAN-OR-EQUAL" Z comparison functions for drawing multiple passes. This way, if you pass the same primitive, it overwrites or blends the current pass with the previous one.

For a brief description of pixel blending, please refer to the "Convincing-Looking Glass for Games" article elsewhere in this book.

Extensions and Enhancements to the Basic Algorithm

Simplicity and high performance usually come at a price. This projective shadow-mapping algorithm is no exception to that rule: It has some limitations. However, most of these limitations are very easy to overcome, and the algorithm can be extended to handle these cases.

Back-face Shadow Elimination

One side effect of projective shadow mapping is that it normally maps a shadow on the side of the receiver facing away from the light source.

We can correct this problem by either:

1. Determining whether a triangle is facing away from the light source and, if it is, assigning out-of-range shadow map coordinates for all its vertices. (The example code on the CD-ROM that accompanies this book does this.)
2. Setting up the rendering of the receiver in such a way that the receiver is completely black on the side facing away from the light source (no ambient lighting). This is the proper method because it is closer to what happens in reality. However, if there is more than one light source in the scene, the "back" face of the blocker can be lit by another light. In this case, we have to use multipass rendering and add the ambient light and light coming from other light sources in separate drawing passes.

Receiver Is Behind the Light Source

You have to explicitly check for this case and not map a shadow on the receiver object.

Multiple Light Sources

This case requires multipass rendering with subtractive blending on the receiver object. Use a receiver rendering pass for each shadow map. The multiple passes successively decrease the intensity (RGB values) in the shadowed areas on the surface of the receiver, making even the shadow intersections look correct.

Multiple Blockers

This case also needs multiple passes. There is one difference, though: The cumulative effect of shadow intersections is incorrect because the two blockers block the light of the *same* light source. Use the stencil buffer to *not draw* in the screen area where there is already a shadow drawn.

References

[Blinn88] Blinn, James, "Me and My (Fake) Shadow," *Jim Blinn's Corner*, pages 53–61, January 1988.

[Foley90] Foley, et al., *Computer Graphics Principles and Practice*, second edition, pages 745–753, Addison-Wesley, 1990.

[Blythe96] Blythe, David, and McReynolds, Tom, *Programming with OpenGL: Advanced Rendering*, SIGGRAPH '96 Course Notes, August 1996.

[Heckbert96] Heckbert, Paul, and Herf, Michael, *Fast Soft Shadows*, SIGGRAPH '96 Visual Proceedings, page 145, August 1996.

[Heckbert97] Heckbert, Paul, and Herf, Michael, *Simulating Soft Shadows with Graphics Hardware*, CMU-CS-97-104, Computer Science Department, Carnegie Mellon University, January 1997.

[Woo97] Woo, Mason, Neider, Jackie, and Davis, Tom, *OpenGL Programming Guide*, second edition, Addison-Wesley Developers Press, Silicon Graphics, 1997.