5.4

Efficient Omnidirectional Shadow Maps

Gary King and William Newhall

Introduction

Shadows are an important visual cue in 3D scenes; they are the primary mechanism for giving users information about the relative positions of objects. Unfortunately, integrating shadows into real-time applications is difficult and substantially increases performance requirements. Due to their cost, it is critical that any shadow implementation be as efficient as possible. This means that only the minimum amount of geometry (and pixels) required to add shadows to a scene be rendered, and rendering these objects as efficiently as possible. This article focuses on algorithms that will improve the performance and quality of omnidirectional shadow depth maps. For clarity, we will limit the scope of this article to implementations using cube maps [Greene86], although most of these techniques are directly applicable to other environment map types, such as dual-paraboloid and tetrahedral.

Review of Omnidirectional Shadow Mapping

In order to implement the optimizations that will be discussed later in this article, we need a basic cube map shadow map implementation as a starting point. For an excellent step-by-step description of cube map shadow maps (including fallbacks for older hardware and drivers), see Philip Gerasimov's article in *GPU Gems* [Gerasimov04], or see the source code for the demo accompanying this article (contained in R32FCubemap.[cpp/h/fx]).

For our quick and dirty shadow cube map implementation, we will use single-component 32-bit floating-point cube maps (`D3DFMT_R32F` in DirectX 9-speak). At the time of writing, this format was supported on Radeon 9500+, GeForce FX-series[1], and GeForce 6-series GPUs. Floating point cube maps naturally lend themselves for use as shadow maps: at each texel, store the radial distance (or squared radial distance) of the occluder from the light source, in light-space. To transform a vertex into light-space, simply subtract the light's position from the vertex's position, where both positions are defined in a common space (e.g., world-space).

[1] `D3DFMT_R32F` exposed in Forceware 60-series and later drivers for GeForce FX GPUs.

To render the shadow map, iterate through the scene six times (one for each face), and transform the light-space vertex by the face's view-projection matrix (remember to clear the Z-buffer for each face).

Finally, to use the shadow map when rendering the final scene, first transform the rasterized fragments into light-space. Since this transform is a linear operation, it can be computed per-vertex and interpolated per-pixel. Compare the distance of each pixel to the values stored in the shadow map (plus a small bias, to avoid precision problems that can lead to self-shadowing). If a pixel's distance from the light is farther than the distance stored in the shadow map, it is in shadow.

The last thing we will add to this basic shadow cube map implementation is shadow filtering. As you can see in Figure 5.4.1, point-sampling leads to noticeable stair-stepping artifacts dividing shadowed and unshadowed regions. These artifacts are known as shadow aliasing, and can be hidden by super-sampling the depth comparison in the final pixel shader. Simple implementations perform depth comparisons with multiple shadow texels and average the results (percentage closer filtering); more advanced implementations perform a true reconstruction filter on the results to get a smoother shadow edge (e.g., bilinear PCF). Because true bilinear filtering on cube maps is an extremely expensive process, we will use an approximation described by Arkadiusz Waliszewski that uses eight shadow comparisons and three vectorized LERPs [Waliszewski03].

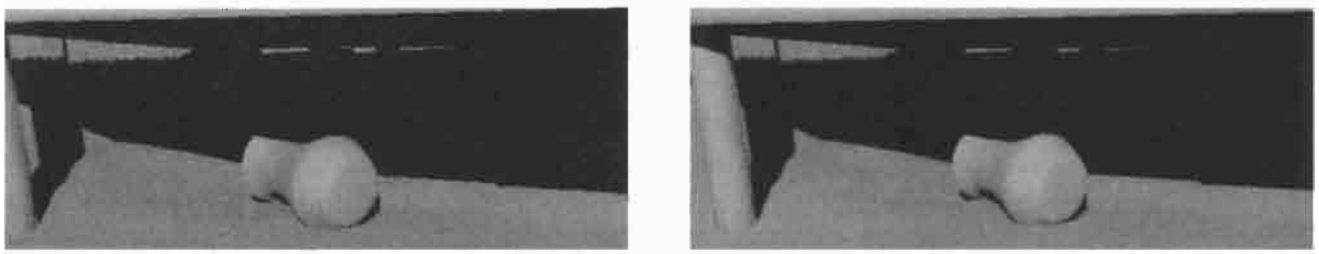

FIGURE 5.4.1 *Point-sampling the depth comparison yields a harsh transition at shadow boundaries.*

Optimizing Shadow Texture Computation

Looking at this example, the first thing you should notice is that omnidirectional shadowing is extremely expensive. Even with per-object view frustum culling implemented for all rendering passes, the sample application increases from 18 milliseconds per frame without shadows to 125 ms with filtered shadows on a Quadro FX Go1000—rendering the shadows is more than four times as expensive as rendering the rest of the scene! If a game is going to include fully dynamic shadowing, this has to be improved if real-time frame rates are going to be achieved.

Culling Cube Map Faces—Intersection of Two Frusta

The most effective optimization we can perform is not rendering faces of the shadow map, which saves both shadow map fill and vertex processing. The question then is how to cull portions of the shadow map without introducing differences in the resulting image. In order to do this, we are going to extend the simple axis-aligned bounding box frustum culling in the sample application to also cull against other frusta.

Figure 5.4.2 illustrates how the six shadow map frusta are located relative to the view frustum for a variety of light positions. In all cases, if the light source is outside of the view frustum, then we can skip rendering at least one face of the shadow map.

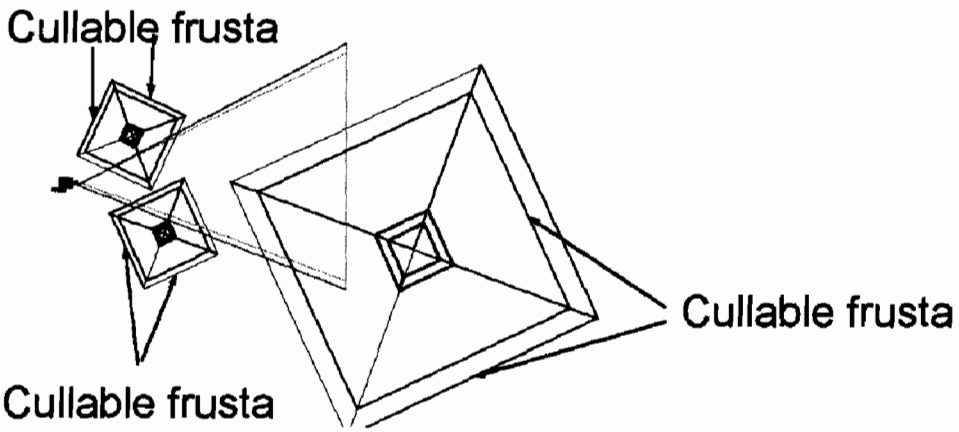

FIGURE 5.4.2 *If the light source is outside of the view frustum, at least one face is cullable.*

Frustum culling an arbitrary frustum, F, against a view frustum, V, is only slightly more complicated that frustum culling an axis-aligned bounding box. First, we need to determine if the two frusta intersect. We only need to handle four cases in order to determine this, and these cases can be divided into two symmetric tests: a boundary point of one frustum is inside the other frustum, or a boundary edge of one frustum intersects one or more clip planes of the other frustum. A frustum has 8 boundary points and 12 boundary edges. A boundary point is the intersection of any three clip planes, and a boundary edge is the intersection of two neighboring clip planes. In pseudocode, this looks like:

```
HalfIntersection( Frustum A, Frustum B )
P = points(A)
    For all p in {P}
            If inside(p, B)
                    Return intersection
    E = edges(P)
    For all e in {E}
            If intersect(e, B)
                    Return intersection
    return !intersection
```

```
Intersection( Frustum F, Frustum V )
return HalfIntersection(F,V) ||
       HalfIntersection(V,F)
```

Applying this to shadow frusta is straightforward: for each face of the shadow map, compute if the view frustum intersects the shadow face frustum (or vice-versa). If the two frusta do not intersect, we know that the shadow face doesn't affect the final image and we can skip all additional processing of it.

Improved Frustum Culling for Shadow Passes

We can apply this improved frustum-frustum culling code for shadow-casting objects, too. In most scenes, it is reasonable to assume that a small percentage of objects are casting visible shadows. However, simply performing bounding-box culling between shadow casters and the view frustum will not behave correctly—the shadows will "pop" into view. Similarly, simply culling shadow caster bounding boxes against the shadow frustum may result in invisible shadows being drawn, spending GPU cycles unnecessarily. To understand why this happens, look at Figures 5.4.3 and 5.4.4.

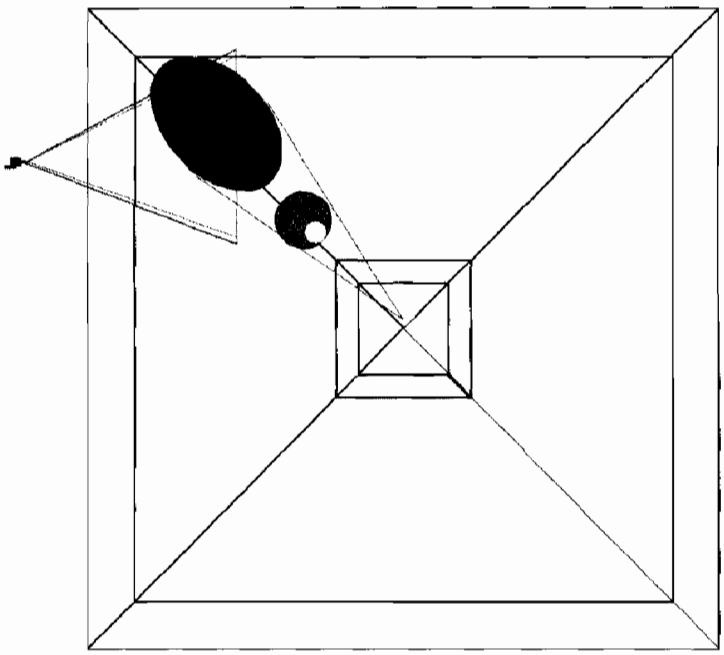

FIGURE 5.4.3 *An off-screen object casts a shadow into the visible scene.*

In order to properly cull a projected shadow against the view frustum, we need a representation of the projected shadow. Since we are casting shadows from point lights, we know that all shadow rays emanate from the same point. This is analogous to all rays converging to the same point, as is the case with perspective projections.

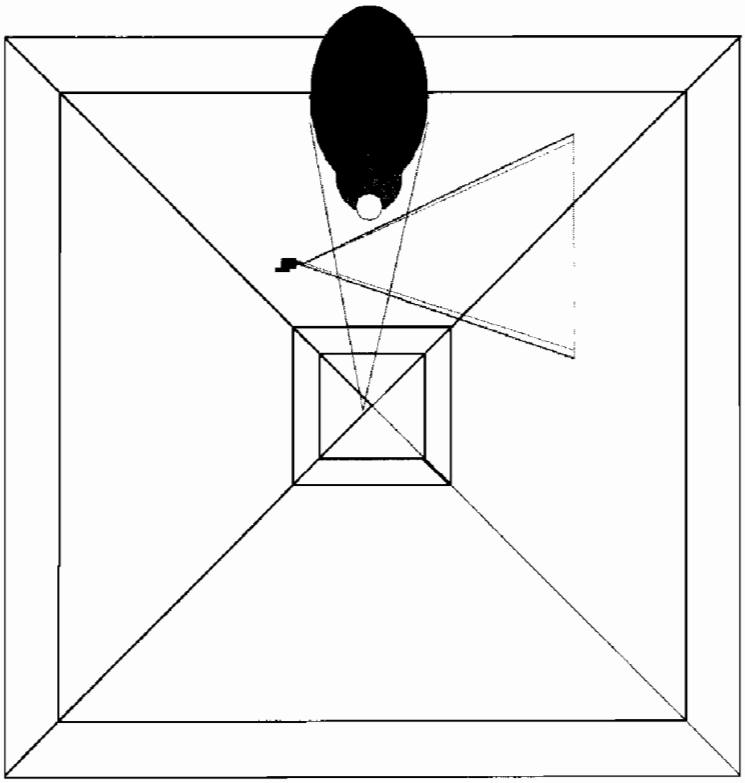

FIGURE 5.4.4 *An object within the shadow frustum casts an invisible shadow.*

Therefore, if shadow casting objects are represented as bounding boxes, a frustum can be used to represent the projected shadows, and we can reuse the frustum-frustum culling test developed above.

To build this frustum, the demo computes a tight bounding cone surrounding the light position and each shadow caster's world-space bounding box. This cone is trivially converted into a centered frustum. Note that better algorithms for building frusta from bounding boxes exist [Schmalstieg99], but for our purposes, this algorithm works fine.

There is one special case that needs to be handled for robust shadow-caster culling: when the light is inside a shadow caster's bounding box. In this case, one frustum is insufficient for representing the projected shadow (see Figure 5.4.4). However, if the light is attenuated, a bounding box bounding the entire light range[2] can be used instead, and simple bounding box culling can then be used.

At this point, our improved omnidirectional shadow-mapping algorithm is:

[2]Although the actual light range is infinite, a good approximation is to define the light range as the distance were the light intensity falls below a minimum (visible) threshold.

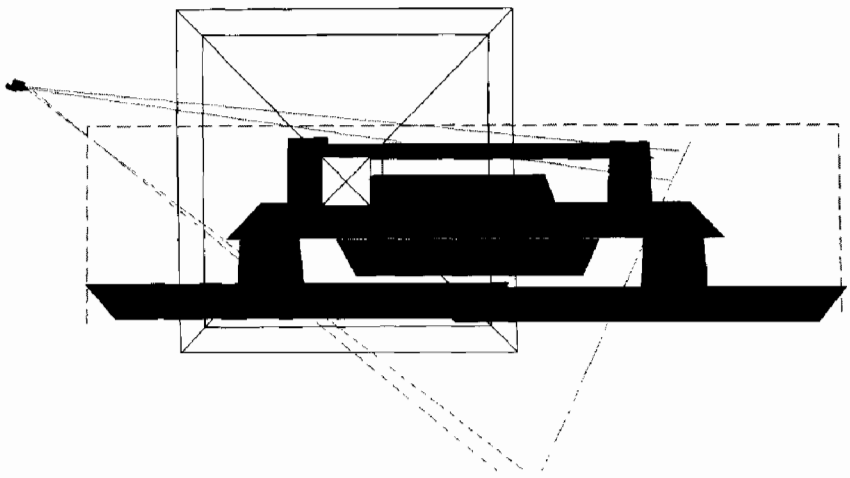

FIGURE 5.4.5 *Light source inside a shadow caster's bounding box casts a shadow on multiple cube faces.*

```
for all I in { cubemap faces }
        if intersect(view_frustum, frustum(I))
                set render target (I)
                for all O in { scene objects }
        if intersect(frustum(O), view_frustum)
        draw(O)
```

Optimizing Rendering With Shadow Textures

Scissoring

So far, the optimizations we've implemented have primarily helped cases where the light source is outside of the view frustum, and focused on reducing the cost of generating the shadow maps. However, lights are frequently inside the view frustum. Applying a shadow map in a pixel shader is costly, too, especially if filtering is applied.

In many cases, when a light source is inside the view frustum, it affects only a small number of pixels on the screen; Figure 5.4.5 demonstrates such a case. Using a brute-force algorithm, we might spend millions of GPU cycles computing per-pixel lighting with filtered shadows for every pixel in the frame, even though the light affects only a small percentage of those pixels! We can take advantage of a feature recently added to Direct3D called the scissor test to quickly cull away most of these pixels.

To use the scissor test, we need to define a *scissor rectangle* representing the region of the screen affected by the light source. Any pixels inside this rectangle will pass the scissor test and get fully shaded, other pixels are quickly rejected. To compute this rectangle, build an axis-aligned bounding box around the light source with side length equal to double the light's radius of effect. Transform this bounding box into post-projective view space, and find its bounding rectangle. The scissor rectangle is this

bounding rectangle, clipped to the screen extents (see Figure 5.4.6). To avoid problems with negative w coordinates, when the viewer is inside the light's bounding box, set the scissor rectangle to the full screen size.

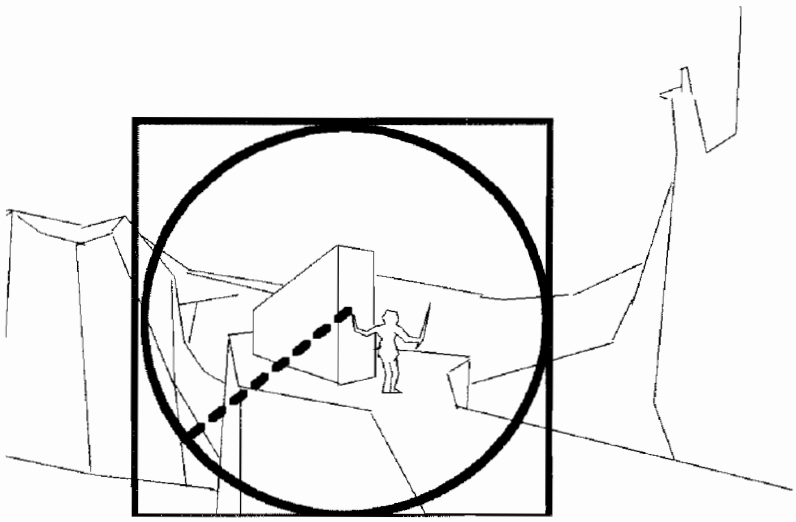

FIGURE 5.4.6 *Circle represents light range from torch, scissored to rectangle.*

Optimizing Shadow Texture Resolution

The last algorithmic optimization we will add builds upon some of the code developed for the scissoring optimization, and addresses a related observation: given a cube map edge length of 512, rendering all 1.2 M texels in the cube map when the light affects only 20,000 pixels in the final image is overkill. In addition to costing more pixel fill to render the shadow map, using an unnecessarily high resolution shadow map will reduce the texture cache hit rate when performing shadow comparisons, which also hurts performance on the shading pass.

In order to perform this optimization, we will need multiple cube maps with different resolutions. GPUs already have a mechanism for supporting this, mipmaps [Williams81]. In the demo, we exploit mipmaps to reduce the number of unique textures we need to create and manage; we do not use mipmap filtering. Because we will only render 1 mip-level per shadow map, we need to force the GPU to texture from a specific LOD. This can be accomplished trivially using the `texCUBElod` intrinsic in 3.0 pixel shaders. However, for non-shader model 3.0 hardware, something else is required. We achieve this "something else" by some rather unintuitive use of the LOD bias and max mip-level sampler states (`D3DSAMP_MIPMAPLODBIAS` and `D3DSAMP_MAXMIPLEVEL`)[3]. Isotropic texture LOD is computed as:

$$lod' = \max(MAXMIPLEVEL, \log_2(rho) + MIPMAPLODBIAS)$$

[3]In OpenGL, setting both `GL_TEXTURE_MIN_LOD` and `GL_TEXTURE_MAX_LOD` to the desired LOD will have the same effect.

This will force the hardware to texture from `MAXMIPLEVEL` level if we can guarantee that `MAXMIPLEVEL` is always greater than the computed LOD. This will be the case if `MIPMAPLODBIAS` is set to a large negative number, equal to at least the number of levels in the mipmap pyramid.

Now that we can texture from and render to specific mipmap levels, how do we decide which level is appropriate for the frame? A straightforward approach is to loosely apply the Nyquist theory to shadow map resolution: if `N` faces of the cube map are visible, covering `S` pixels on screen, choose the coarsest mipmap level with edge length `M` such that `N*M2 >= c*S`, with `c` equal to 2.0. This is not correct, since shadow maps aren't sampled uniformly with respect to the viewer, but it is a reasonable approximation (if your application demonstrates aliasing problems, try increasing `c`).

Virtual Shadow Depth Cube Texture Mapping

We can further improve the performance of our shadow cube texture maps if we can reduce the cost of supersampling and filtering when texturing from a shadow cube texture. The problem with our existing FP32 approach is that each filter tap requires a separate texture instruction, and each shadow compare requires arithmetic instructions and bilinear filtering in the fragment program is just too expensive in terms of instructions and register utilization.

An ideal approach would be to use the D16 and D24 shadow depth texture formats (as supported by Geforce3, Xbox, GeForce4, GeForceFX, and GeForce6) to perform four-tap bilinear percentage closer filtering for each texture fetch. These formats support shadow compare and bilinear percentage closer filtering for each texture instruction.

As an added benefit, shadow textures rendered using the D16 and D24 shadow texture formats take considerably less time to create for a variety of reasons:

- Less memory bandwidth required (especially in the case of D16)
- Fewer transactions, giving better bandwidth efficiency (no color buffer writes)
- Modern GPUs have accelerated depth-only rendering and compressed depth buffers

There is only one problem with D16 and D24 shadow depth texture formats: They don't work with cube maps. Neither DirectX 9 nor OpenGL support depth texture cube maps.

The solution is to render the six faces of the cube map as six subrectangles within a single 2D depth texture which we call the *virtual shadow depth cube texture,* or VSDCT. The fragment program then performs the necessary calculations to map the 3D light vector to the 2D VSDCT texture coordinates, computing the shadow depth used for the shadow map compare. There are three phases to these calculations:

- Cube map transformation (mapping a 3D vector to a 2D vector and a cube face)
- Texture coordinate remapping, scaling, and cube face address offset
- Shadow depth projection, perspective, and viewport transformations

While there are many calculations going on here, the actual implementation of these calculations can be substantially optimized by using an *indirection cube map* to accomplish the first two phases and all of the transformations in the final phase can be combined with a scalar reciprocal and a scalar multiply-add (see Figure 5.4.7).

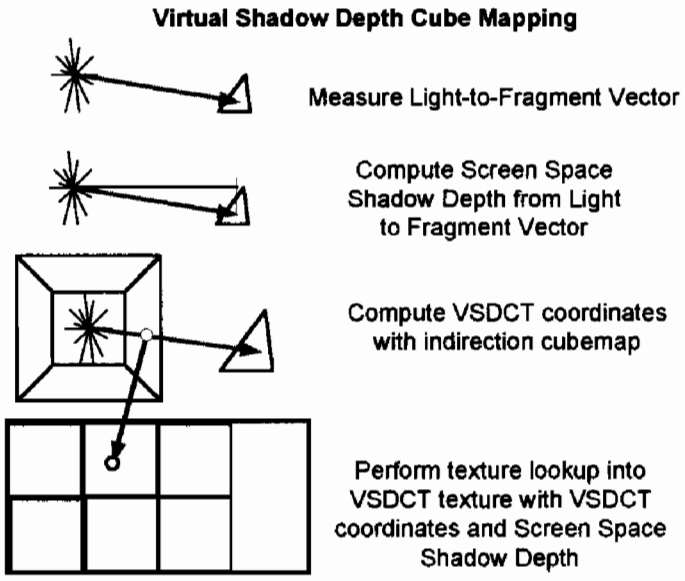

FIGURE 5.4.7 *Conceptual diagram of Virtual Shadow Depth Cube Texturing.*

Because we are using an indirection cube map to map cube texture coordinates to arbitrary points on a 2D texture we now have the freedom to size our shadow textures to arbitrary resolutions. In terms of pixel shader instruction efficiency it is best to use a VSDCT in which all the virtual cube map faces have the same resolution.

Cube Map and Depth Calculations for Rendering with VSDCTs

To combine elements of conventional cube mapping and shadow depth texturing to achieve omnidirectional shadow depth texturing the fragment program computes must now perform a number of calculations that are ordinarily performed by fixed-function hardware.

Cube Mapping Calculations

To correctly map the unnormalized 3D light vector to 2D texture coordinates on the VSDCT we need to accomplish the same sort of calculations that are performed by the texture unit in applying fixed-function cube mapping. Many hardware cube map implementations are conceptually arrays of six 2D textures. The process of mapping 3D vectors to texture coordinates is divided into three steps: face selection, coordinate transformation, and coordinate scaling.

Cube map face selection determines which of the six faces the 3D vector intersects. This can be determined by analyzing the relative magnitude and signs of the three components. After selecting the appropriate face, the texture coordinates are projected onto the unit cube (by dividing by the absolute value of the largest component), and rotated into the face's normalized device coordinates by conditionally swizzling and negating the other two texture coordinates. Finally, these coordinates are remapped to texel locations by applying a simple scale and offset. The indirection cube map mentioned above is used to implement these fixed function operations for VSDCTs, since performing the actual math inside a fragment program requires over 30 instructions.

Depth Coordinate Calculations

We need to apply the same projection, perspective, and viewport transformations to the R coordinate shadow depth that were applied to the depth values by the hardware when the shadow map was created. This is performed in two steps: *shadow depth ordinate selection*, and *projection transformation*.

To properly compare the shadow depth texture, the fragment program needs to compute an eye-space shadow depth from the 3D light vector components. This is known as shadow depth ordinate selection. Because each cube map face projection is exactly 90°, it can be shown that the eye-space depth coordinate is equal to the absolute value of the largest magnitude light vector component. We will call this *MA*. Because the VSDCT texture is storing screen Z, rather than eye-space Z, the same projection transform used for rendering the VSDCT faces needs to be applied to the shadow ordinate. In Direct3D, the canonical perspective projection matrix is:

$$P = \begin{bmatrix} \frac{1}{\tan(fovx * 0.5)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(fovy * 0.5)} & 0 & 0 \\ 0 & 0 & \frac{Z_{far}}{Z_{far} - Z_{near}} & 1 \\ 0 & 0 & \frac{-Z_{far} Z_{near}}{Z_{far} - Z_{near}} & 0 \end{bmatrix} \tag{5.4.1}$$

But for the shadow comparison, we only care about how the projection transform affects the Z component, Z_p.

$$Z_p = Z_{eye} \times \frac{Z_{far}}{Z_{far} - Z_{near}} - \frac{Z_{far} Z_{near}}{Z_{far} - Z_{near}} \tag{5.4.2}$$

and the value of the homogenous term W_p:

$$W_p = Z_{eye} \tag{5.4.3}$$

It is vital that we use a projection matrix with a value of 1 in element (3, 4) because we need W_p to equal eye space Z, which equals the *MA* value we measured from our light-to-fragment vector.

Dividing Z_p by W_p produces screen space Z_s (Z_p normalized to a range of [0–1.0]):

$$Z_s = \frac{Z_p}{W_p} \tag{5.4.4}$$

This value will be used for the shadow map compare within the texture pipeline. Since $W_p = Z_{eye}$ we can find Z_s by dividing both terms of Z_p by Z_{eye}:

$$Z_s = \frac{\dfrac{Z_{eye}Z_{far}}{Z_{far}-Z_{near}} - \dfrac{Z_{far}Z_{near}}{Z_{far}-Z_{near}}}{Z_{eye}} = \frac{Z_{far}}{Z_{far}-Z_{near}} - \frac{Z_{far}Z_{near}}{Z_{eye}\times\left(Z_{far}-Z_{near}\right)} \tag{5.4.5}$$

reordering and substituting *MA* for Z_{eye}:

$$Z_s = \frac{-1}{MA}\times\frac{Z_{far}Z_{near}}{Z_{far}-Z_{near}} + \frac{Z_{far}}{Z_{far}-Z_{near}} \tag{5.4.6}$$

Thus, our fragment program can compute shadow depth by computing the reciprocal of *MA* and applying a constant scale and a constant bias using a Multiply-Add instruction. This can be seen in VSDCT.fx.

Constructing Virtual Shadow Depth Cube Map Textures

Allocate the VSDCT Depth Buffer Render Target

When initializing your application (or at level load time) allocate the VSDCT surface with a power-of-two resolution that is large enough to accommodate six virtual cube faces at a reasonable resolution. The indirection cube maps will constrain our sampling to prevent cube seam crossings, so there really is no requirement to arrange the virtual cube face subrects contiguously.

In Direct3D, VSDCT surfaces can be created by allocated using `CreateTexture()` with a depth format (e.g., `D3DFMT_D24X8`) with `D3DUSAGE_DEPTHSTENCIL` specified. This is exactly like creating a 2D shadow map surface; for more details, see the comments in the accompanying source code (`VSDCT.cpp`).

Render the Virtual Cube Faces

Projection Matrices

To simplify the depth calculations required by the fragment program when applying the VSDCT we will use the same projection matrix for all of the cube faces. To maximize precision the near and far depth plane values should be selected by checking the bounding boxes of objects within the effective range of the light. The near depth plane should be as large as possible and the far depth plane should be as small as possible.

For maximum precision and efficiency, this matrix must have a 90° field of view in X and Y and must have a 1 in element (3, 4) so that the W value computed by the projection matrix will equal eye-space Z. If element (3, 4) is not equal to 1.0 then elements (1, 1), (2, 2), (3, 3,), and (4, 3) must be divided by the reciprocal of element (3, 4) and element (3, 4) must be replaced with 1.0.

Readying the Render Target and the Viewport

Set the render target write mask to disable color and stencil reads and writes. Clear the depth buffer. For each virtual cube face set the viewport to the rectangular region of the VSDCT that has been allocated for the virtual cube face.

Indirection Cube Map

The complete chain of cube face selection, coordinate rotation, side divide, (S, T) remap, (U, V) scaling, and cube face offset can all be accomplished with a single texture lookup into an *indirection cube map*. This cube map is created offline and can be used with multiple VSDCT textures provided that all faces of the virtual shadow cube map have the same resolution and that every virtual shadow cube map that employs this indirection cube map will use the same subrect configuration on every frame.

The format of an indirection cube map is `G16R16` and its resolution should be selected based on the resolution of the virtual shadow cube map and the desired dynamic range of the shadow. We can trade dynamic range to reduce indirection cube map memory bandwidth consumption by selecting a lower number of grey levels than the bilinear filtering hardware supports. This is accomplished by computing the resolution of the indirection cube map this way:

$$N_i = 2^{\lceil \log_2(N_s) \rceil + \min(\lceil \log_2(G) \rceil, D) - D} \tag{5.4.7}$$

where N_s is the resolution of the shadow map cube face, G is the number of desired grey levels from a bilinearly filtered shadow map comparison, and D is the number of bits of precision that the underlying hardware uses for texture derivatives (on GeForce FX and GeForce 6-series GPUs, $D = 8$). So, as an example, for a shadow cube map with 256 × 256 texel faces, and 16 desired grey levels from the final shadow comparison, $N_i = 16$.

Issues and Potential Enhancements

Shadow Texture Aliasing Artifacts

No depth shadow-texture mapping paper would be complete without acknowledging the great unsolved problem of shadow-texture mapping. Because shadow-texture mapping involves a threshold test between two entirely different sets of point samples, the aliasing artifacts are significantly more pronounced in the form of self-shadowing and light leaks. Taking more samples, using larger filtering kernels, and applying depth bias or slope-scale depth bias can reduce the visibility of these defects. However, the root cause of the problem is that we are attempting to compare measurements taken with two entirely different sample sets and reconcile them with a filter.

Sampling across Cube Map Seams

An unavoidable sampling artifact occurs when sampling quads that straddle multiple cube map faces. Ideally, the hardware filtering would walk across the boundary, sampling all the right texels along the way. Unfortunately, there is no way to arrange the virtual cube faces within the VSDCT such that all seam crossings are correct (see Figure 5.4.8). Our workaround for this issue is to build the `indirection` and `uvScaleBias` textures to avoid sampling across seams. This has the side effect of reducing texture lookups on seams to point sampling. Thankfully, this artifact is rarely noticeable, especially when multiple VSDCT samples are used to perform higher-order PCF.

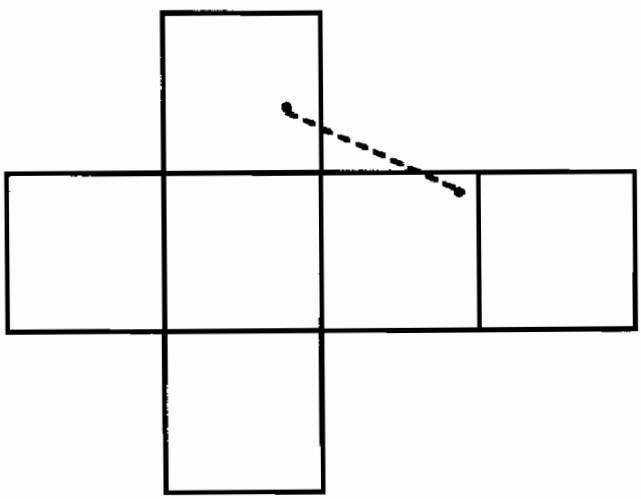

FIGURE 5.4.8 *Incorrect sampling across cube seams. The blue dots are adjacent texture samples, separated by unused VSDCT texels.*

Additional Sampling and Filtering

Many applications perform multiple, offset shadow map lookups to mask objectionable shadow map aliasing. For the same reasons that cube seam lookups need to be

constrained to individual VSDCT faces, performing simple 2D offsets is insufficient. The correct way to perform higher-order PCF on VSDCTs is to offset the original 3D vector, perform the indirection lookup using this vector, and use the result to fetch from the VSDCT.

Nonuniform Cube Map Sampling

Texels on a cube map (or VSDCT) are not distributed uniformly across the sphere—the sample resolution at the center of each cube map face is more than 5× the resolution at the corners. A potential way to reduce visible shadow map aliasing without increasing the resolution is rotating the cube map to provide maximal texel density near the focal point. This is left as an exercise to the reader.

Conclusion

Despite its visual impact, real-time dynamic shadowing has been avoided in most games due to its huge performance cost. However, most of this performance cost can be avoided—we have provided a number of techniques that dramatically improve the performance (and quality) of omnidirectional shadow maps. By applying all of these techniques to our demo, it ran nearly three times faster on GeForce FX products, and over one and a half times faster on GeForce 6-series products. In our demo, VSDCTs provided the largest individual benefit for lights filling the full screen, but were also the most complicated to implement. For smaller lights, scissoring and reducing shadow texture resolution provided performance gains comparable to VSDCTs, with significantly less implementation complexity. Due to this performance increase, using these optimizations in games will allow users without the highest-spec PCs to enjoy dynamic shadowing.

References

[Gerasimov04] Gerasimov, Philipp, "Omnidirectional Shadow Mapping," in Randima Fernando, ed., *GPU Gems*, Addison-Wesley, pp. 193–203, 2004.

[Greene86] Greene, Ned, "Environment Mapping and Other Applications of World Projections," *IEEE Computer Graphics and Applications*, 6(11), pp. 21–29, Nov. 1986.

[Schmalstieg99] Schmalstieg, Dieter and Robert F. Tobler, "Fast Projected Area Computation for Three-Dimensional Bounding Boxes," *Journal of Graphics Tools*, 4(2), pp. 37–43, 1999.

[Waliszewski03] Waliszewski, Arkadiusz, "Floating Point Cube Maps," in Wolfgang Engel, ed., *ShaderX²*, pp. 319–324, 2003.

[Williams78] Williams, Lance, "Casting Curved Shadows on Curved Surfaces," *Computer Graphics (Proc. Siggraph '78)*, 12(3), pp. 270–274, August 1978.

[Williams81] Williams, Lance, "Pyramidal Parametrics," *Computer Graphics (Proc. of SIGGRAPH 83)*, 17(3), pp. 1–11, 1983.