Fast, Practical and Robust Shadows

Morgan McGuire, John F. Hughes, Kevin T. Egan,
Mark J. Kilgard, and Cass Everitt

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-03-19
October 2003

October 27, 2003

Fast, Practical and Robust Shadows

Morgan McGuire, John F. Hughes, and Kevin T. Egan
Brown University

Mark J. Kilgard and Cass Everitt
NVIDIA Corporation

Abstract. We present a set of algorithms for rendering shadows using the stencil buffer and shadow volume geometry. It can achieve greater performance than previous methods by applying a series of techniques for culling, clipping, and simplifying shadow volume geometry. The performance of many 3D games is currently limited by pixel fill rate, and the clipping and culling techniques effectively reduce the fill rate consumption of shadow rendering by a constant factor. Scientific and engineering 3D applications can instead be performance limited by the hardware vertex processing rate and have unused fill rate. For a shadow caster with O(s) possible silhouette edges and O(f) faces, the simplification techniques reduce the triangle count needed for shadow determination for two common cases from O(f) to O(s), reducing the number of vertices processed. Finally, we use a vertex program and indexed vertex array to reduce the memory bandwidth requirement from 36 to 6 bytes per triangle on programmable hardware. The effectiveness of these techniques depends highly on the geometry of the 3D scene and the graphics hardware available. For example, the simplification method trades vertex rate for fill rate and will slow down applications that are fill rate bound.

Our algorithm builds on previously published algorithms by Crow, Everitt and Kilgard, and Lengyel. To facilitate implementation, we review their methods and give a complete shadow algorithm rather than just showing our improvements in isolation. Further implementation details include an overview of the data structures used and a pseudo-code implementation. An implementation in C++ using OpenGL is available on our website.

1. Introduction

Real-time shadow rendering techniques are currently a popular topic in computer graphics. This paper presents a new algorithm for rendering shadows using the shadow volume method. It is a both a research paper that presents a novel algorithm and a standalone document for teaching programmers how to implement our algorithm, including the parts originally published by other authors upon which we build. To our knowledge this is also the first paper to present a theoretical analysis of the performance of shadow volume methods.

In the physical world, the visual phenomenon of a *shadow* occurs when a *shadow caster* blocks most of the photons from a light source from reaching a *shadow receiver*. Our algorithm uses a two-pass strategy for producing this effect. We operate on a scene consisting of a single point light source and no ambient illumination. The algorithm then extends to multiple lights, light types, and ambient illumination. The first pass is called the *shadow determination pass*. It separates screen-space regions of light and shadow. The second pass illuminates only the "light" areas, allowing the shadowed regions to remain dark. We call this the *illumination pass* because it creates the actual illumination.

The majority of this paper is concerned with making the shadow determination pass fast, however in section 8 we present a simple rendering system incorporating our algorithm to demonstrate how to integrate the illumination and determination passes.

Our shadow determination algorithm uses shadow volumes, which are described in detail in the following section. It takes as input a scene containing a set of shadow casting models, a viewer, and a set of lights. The data structures used for these objects and constraints on them are described in section 4. The algorithm classifies each pixel as shadowed or illuminated and stores the result in the stencil bits corresponding to that pixel. The stencil value is zero for an illuminated pixel and non-zero for a shadowed pixel. Section 5 describes the stencil buffer and other hardware capabilities required by our algorithm and section 6 presents the algorithm itself. Section 7 contains an analysis of the number of triangles rendered by our algorithm. An appendix lists documentation available on the web for some of the newer hardware features used in our algorithm.

2. Shadow Volumes

In this section we briefly review shadow volume geometry and notation. A point is in shadow with regard to a light source if there is no line of sight between that point and the light. We construct geometry called a shadow volume and use it to identify such points.

The set of edges where a front face meets a back face is important in shadow volume creation. Crow and Sutherland called these *silhouette* edges [Crow77] because for a convex object they outline that object's silhouette. For a non-convex object some of these edges lie within the outline shape commonly called the silhouette, so we introduce the term *possible silhouette* edges to indicate that they are a super-set of the true silhouette.

The *shadow volume* of a model and a light is the set of all points that are occluded from the light by the model. It can be helpful to visualize this as the volume of darkness stretching away from an object held in front of a headlight on a foggy night. The shadow volume of a polygonal object is a closed polyhedron with infinite volume. We construct a shadow volume as follows.

The shadow volume of a triangle is a triangular frustum capped on top by the triangle itself and extending away from the light, to infinity. At infinity, it is closed by another triangle. The shadow volume of a model is the union of the shadow volumes of its component triangles.

Because our concern is the volume of space we can simplify the geometry of this construct by removing interior geometry. Within the volume there are pairs of triangles with the same vertices but opposite orientation that effectively cancel each other. These can be removed, leaving only a polyhedral shell of the volume. For models that satisfy the constraints described in section 4, we can construct this union directly. In this case it is bounded by the light facing polygons from the model, their counterparts at infinity, and a set of polygons connecting them. The boundary of the light facing polygons of a model is the possible silhouette of the model as observed from the light source, so these connecting polygons are referred to as the *extruded possible silhouette*. We call the three pieces of the shadow volume geometry the *light cap* (the light facing polygons), *dark cap* (their counterparts at infinity), and the *sides* (extruded possible silhouette) that stretch between the caps.

By definition the shadow volume for an object contains all of the points that it obscures from the light source. That is, a point in the shadow volume of *any* object in the scene must be in shadow. Shadow determination is thus reduced to a point-in-polyhedra test where the points are the visible points in the scene and the polyhedra are the shadow volumes of the models. A general algorithm for performing point-in-polyhedra is as follows.

Assume we know a point U that is outside all polyhedra. For each point P to be tested, choose a path between P and U. Find all intersections between this path and the polyhedra faces. Label an intersection +1 if the path *enters* (the dot product between the face normal and the path tangent is negative at the intersection) and –1 if the path *exits* the polyhedron at that intersection. Let faces contain their edges so that a glancing intersection perfectly perpendicular to a face must enter the polyhedron at one edge of that face and exit at a subsequent edge. The sum of the labels is the number of polyhedra containing P. For our purposes, it is sufficient to note that the sum is zero when P is outside all polyhedra, which corresponds to unshadowed in our algorithm, and non-zero when P is inside at least one polyhedron.

This test is performed simultaneously for all visible points by rendering the faces of the shadow volumes to the stencil buffer. They are specifically **not** rendered to the depth or color buffers and are not themselves visible in the final image.

In the hardware accelerated point-in-polyhedra test, we perform the test simultaneously for each pixel. Conceptually, each pixel is one point P. The processes of rasterization and projection makes it convenient to work with paths that travel along the line between P and the center of projection (viewpoint). For each P we therefore choose U, our known to be unshadowed point, along this line (the path between them along which we count intersections is on the line as well). There are two choices for U. The first choice is the point where the line intersects the viewport. This point is known to be outside all shadow volumes if and only if the viewport is not in a shadow. Figure 1 shows a 2D schematic of a scene with two models, one casting a shadow on the other. The viewer is represented by the "V" shape on the left and the viewport by the line in front of it. One visible point P is represented as a blue dot in the center of the diagram. The corresponding U point on the viewport is shown as U_V and marked by a blue dot.

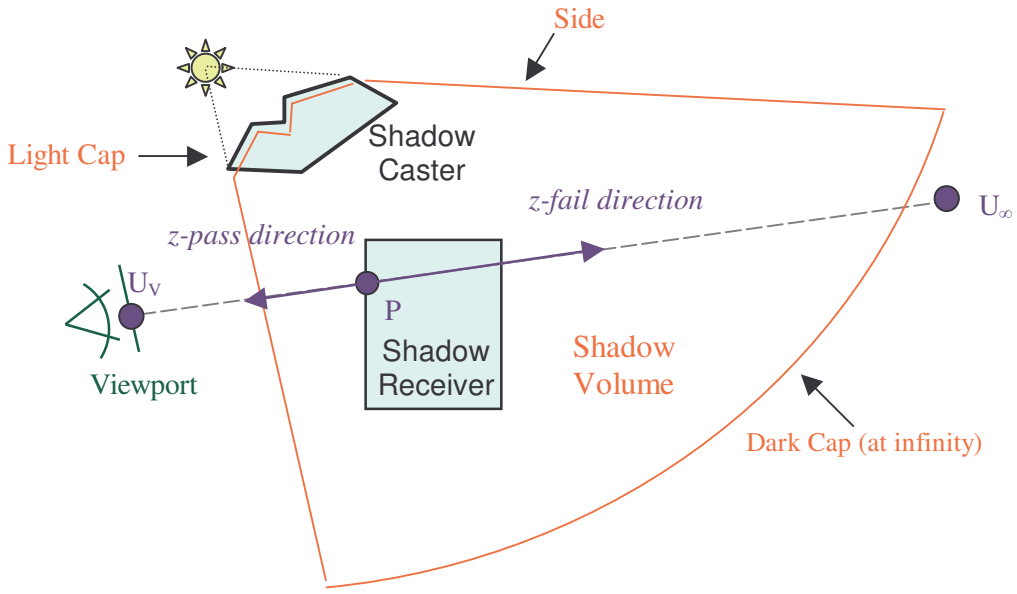


Figure 1. A 2D diagram showing shadow volume geometry

Another choice is the point at infinity at the far end of the line, shown in the figure as U_∞. This point is always outside all shadow volumes because it is infinitely far away from the scene (technically, it could be on the dark cap of some shadow volume at infinity, but we will use a strict definition of *inside* and consider that outside the polyhedron).

The intersections between the shadow faces and the line between P and whichever U we choose all lie along the along the dashed gray line through P and the center of projection. Note that entering intersections must correspond to shadow front faces (i.e. those triangles presenting positive counter clockwise area to the viewer) and exiting intersections must correspond to shadow back faces. We can therefore label and count all intersections in the stencil buffer by rasterizing the shadow faces. The stencil operation must be set to increment the stencil count when a front face is rasterized and to decrement the count when a back face is rasterized.

Of course, we only want to count intersections between P and U, not along the entire line. Because P is a *visible* point, these intersections can be discriminated by a depth test. For U_∞ at infinity, the shadow faces must *fail* the depth test. For U_V at the viewport, shadow faces must *pass* the depth test. Counting towards infinity is thus called the *z-fail* method and counting towards the viewport the *z-pass* method.

3. Related Work

Crow introduced shadow volumes in 1977, expressing shadow determination as a point-in-polyhedron problem [CROW77]. Crow's method explicitly clips shadow geometry to the view frustum, generating perfect caps where the volume crosses a clipping plane. Crow also used possible silhouettes to simplify shadow volume geometry. He recommends Sutherland's possible silhouette algorithm, which operates by identifying edges where a back face meets a front face. Barequet et. al [BARE01] propose a geometric for fast incremental determination of possible silhouette edges for static models to replace Sutherland's method.

Heidmann [HEID91] adapted Crow's algorithm to hardware acceleration by using the stencil buffer to compute the per-pixel count for the point-in-polyhedron test. This is what we now call the z-pass method. As discussed in section 6, the z-pass method does not require light and dark caps so it renders O(s) triangles. Although not mentioned in Heidmann article, his method produces incorrect results when the viewport cuts through a shadow volume. This problem of a clipping plane "opening up" a closed shadow volume and the promise of hardware-accelerated shadows motivated a number of papers.

One class of papers (including this paper) use z-fail testing to ensure robustness. Carmack [CARM00] and Bilodeau and Songy [BILO99] separately identified the problem with Heidmann's method and proposed z-fail testing, eliminating the problem at the near clipping plane. Reversing the count direction only moved the problem to the far clipping plane, however. Their z-fail methods produce incorrect results whenever the viewer has a clear line of sight to the far clipping plane. Everitt and Kilgard [EVER02] resolved this by moving the far clipping plane to infinity, producing the first hardware accelerated shadow volume method that gives correct results for all scenes. Everitt and Kilgard's method renders O(f) triangles, so it is slower than Heidmann's. However, they observed that a simple geometry test can determine when the viewport is in shadow and proposed that z-fail testing only be used when necessary.

Lengyel [LENG02] created a hybrid algorithm from this that uses faster z-pass rendering when the viewport is not shadowed and reverts to robust z-fail rendering when the viewport is shadowed. Lengyel also incorporated another suggestion by Kilgard, to use a rectangular clipping region to reduce the fill rate consumed when rendering shadows from point lights [KILG02]. The relevant observation here is that point lights only produce significant illumination within some finite radius. Lengyel computes the projected image plane bounding box of a 3D sphere of this radius and sets the OpenGL scissor region to that bounding box when determining shadows and rendering illumination from a point light source. This reduces the fill rate consumed by shadow determination and illumination passes.

The papers discussed thus far form a line of research that used z-fail testing to achieve robustness. An alternative line of research has pursued capping at the near plane, like Crow's original method. Diefenbach [DIEF96] presented capping methods that were not completely robust [EVER02]. Möller and Haines [MÖLL99] discuss a well-known but none-the-less incorrect method in their book. This method counts the number of shadow volumes enclosing the center of projection by casting a ray to infinity and solving for intersections with shadow faces. If the count is non-zero, indicating that the viewer is in shadow, all stencil values are incremented by one. This is effectively a full-screen cap. This method produces incorrect results because it approximates the viewport rectangle with a single point. If a shadow volume crosses partway through the viewport, no uniform adjustment of stencil values will give correct results because some points on the viewport are in shadow and some are not.

Shadow mapping [WILL78] is an alternative, image-based method for shadow determination. Unlike shadow volumes, shadow maps have the ability to shadow any renderable object (e.g. billboards, non-closed models). An interesting algorithm by McCool [MCCO98] generates shadow volumes from a shadow map image through edge detection. The advantage of this technique is that it allows easy generation of shadow volumes for arbitrary objects. It also incorporates some of the undesirable aspects of shadow maps, like difficulty handling point lights at finite locations, a singularity when the light and camera are directly opposed, and precision limitations. It is possible to apply our culling and dark cap simplification methods to McCool's technique directly since he effectively generates a possible silhouette. However, his reconstruction method is slow compared to direct possible silhouette computation on a geometric model and does not seem applicable to current consumer hardware.

Our algorithm follows the z-fail line of research and builds on Lengyel's algorithm. We improve performance over his algorithm in two ways. First, our aggressive culling and depth

bounds clipping can reduce the fill rate consumption of rendering shadow volumes to the stencil buffer. We also incorporate Lengyel's attenuation culling but use a cube bound for the light's effective illumination volume rather than a sphere. The fill rate reduction produces a large performance improvement when many lights shine over a limited area. For example, a scene with many dim point lights or spot lights pointed at the ground. It is hard to characterize the improvement formally because it is very scene and viewer dependent. Second, our culling and simplification reduce the number of triangles rendered, and therefore the number of vertices transformed. Specifically, our algorithm distinguishes between three cases (shown later in Figure 4). Lengyel's algorithm has two cases, z-pass and z-fail. In the z-pass case he renders O(s) polygons because the light and dark caps are not needed. His z-fail case encompasses our cases 2 and 3. For both of these he renders O(f) polygons because he reverts to Everitt and Kilgard's method. The probability of each case occurring is highly dependent on the scene and viewer position, so rather than provide a single bound we treat the cases independently and characterize his algorithm as O(s), O(f), O(f). Our algorithm renders fewer triangles in case 2 and is characterized as O(s), O(s), O(f).

Several new shadow volume improvements are suggested by papers to appear later this year. Assarsson et. al [ASSA03a, ASSA03b] describe how to create penumbrae for soft shadow rendering from shadow volumes. Brabec and Seidel [BRAB03] describe a method for performing possible silhouette edge determination in hardware. Their method is also compatible with our algorithm, however it requires a high-precision frame buffer and a fast method for converting an offscreen buffer to a vertex array for an ideal implementation.

4. Data Structures

This section describes the data structures used in our shadow determination algorithm and the constraints on those structures. It is assumed that these data structures will be augmented with additional information like texture maps, animation poses, and physics data when used in a real rendering system.

A scene consists of a viewer and corresponding viewport, a set of models, and a set of lights. The number of lights is not limited. However, areas with many dim lights will tend to have low color resolution and areas with many bright lights will saturate the color depth of the frame buffer. The number and position of the models is constrained by the bit depth of the stencil buffer as described below, although we note that this constraint is rarely a concern in practice.

A *model* consists of an array of indexed triangles, an array of indexed edges, and the array vertices on which they are indexed. For efficiency, vertices contain three components; the homogeneous component is implicitly $w = 1$. Vertices are in the model's object space and will be transformed to world space in graphics hardware by some transformation matrix T. An oriented, object space bounding box and sphere are stored for the model. All indices are assumed to be zero-based.

Struct Model:	
Triangle	triangle[f]
Edge	edge[3 f / 2]
Vector3	vertex[v]
Box	boundingBox
Sphere	boundingSphere

A model must be *closed* so that every edge connects exactly two triangles and there are no T-junctions. A model with f triangles that meets this definition is geometrically constrained so that

the size of the edge array is $3f / 2$. A *caster* is a model that has been designated to cast shadows. While every model may be a caster, it is sometimes desirable to disable shadow casting for certain models. For example, the shaft of a torch may cast such a large shadow on the ground that the result is aesthetically undesirable. In this case, the torch can be excluded from the set of shadow casters when determining shadows from the torch flame (in the real world, a torch flame is not a point light, so the shaft's shadow is substantially diminished).

Identical copies of the vertex array are stored both in main memory and in video card memory. This eliminates the need to transfer that data every rendering frame. However, this imposes an additional constraint for models that are animated by skinning (the process of blending different transformations for a vertex from nearby bones). The process used in hardware to skin the model must be identically replicated in software so the possible silhouette determination step can find the possible silhouette of the animated model, not the static pose. This is a problem common to all shadow volume techniques and cannot be eliminated until graphics hardware is sophisticated enough to support the adjacency data structures needed for possible silhouette determination or rendering returns to the main CPU. In this paper, the hardware vertex array[1] for a model is notated as the array VAR. This array should be allocated in video memory for maximum performance For a model with v vertices, VAR has length $2v$, and all elements are explicit homogeneous vectors.

Vector4	VAR[$2v$]

The value of VAR[i] is as follows. For $0 \le i < v$, VAR[i].xyz = vertex[i].xyz, VAR[i].w = 1. For $v \le i < 2v$, VAR[i].xyz = vertex[$i - v$].xyz, VAR[i].w = 0

A *triangle* stores the indices of its three component vertices and a Boolean field that is used as temporary storage during the shadow rendering algorithm.

Struct Triangle:	
bool	lightFacing
int	index[3]

Each *edge* contains the indices of its two component vertices and the two adjacent faces.

Struct Edge:	
int	vertexIndex[2]
int	triIndex[2]

Edges are directed from vertexIndex[0] to vertexIndex[1]. *Reversing* an edge changes the order of these indices. The edge is in face triIndex[0]; the reversed edge is in triIndex[1] and is not explicitly represented in the model.

The algorithm constructs a set of possible silhouette edges. This set may be efficiently implemented as dynamic array of edge indices. Note that the size of the array is bounded by the size of the model's edge array, $3f / 2$.

The OpenGL representation is used for *lights*. The position is represented by a homogeneous vector. For a point light, $w = 1$. A directional light is mathematically equivalent to a point light at infinity, where $w = 0$. The position of a directional light can also be interpreted as the direction *towards* the light from a point in the scene.

[1] See http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt for a discussion of OpenGL vertex arrays.

Each light has an associated color, bounding box, and radius. For an attenuated point or spot light the box bounds the volume over which the illumination contribution is non-negligible (i.e. is perceptible). For a directional or unattenuated light, this box extends to infinity in all directions and contains the scene. For an attenuated light it is finite. In practical rendering systems (e.g. Doom III) it is desirable to let artists design custom attenuation functions rather than restrict them to the procedural models supplied with the fixed-function rendering pipeline. Because a single 3D texture or multiple 2D textures are natural ways of implementing such custom attenuation functions, a box instead of a sphere is needed to tightly bound the attenuation function. For standard procedural lights the bounds can be computed as follows. Invert the attenuation algorithm and solve for the distance at which the largest intensity component is attenuated below the smallest representable value in the frame buffer.

Struct Light:	
Vector4	position
Color	intensity
Cone	cone
Box	bounds
double	radius

Spotlights and projective spotlights have a cone that bounds the illuminated volume. For all other light types, this cone is opened wide (angle = 2π) so as to encompass the whole scene. Area and volume lights are simulated by collections of many dim point lights. Aliasing artifacts can be reduced by assigning jittered (but still static) locations for these lights rather than arranging them in a grid.

The hardware frame buffer is assumed to consist of fragments (pixels) with at least stencil, color, and depth information.

Struct Fragment:	
int	stencil
Color	color
double	depth

The bit depth of the stencil buffer constrains the scene. Because modular stencil arithmetic is used, the algorithm cannot differentiate between a stencil value of 0 and any integer multiple of $2^{\text{stencil bits}}$. When the algorithm terminates, the stencil value is the number of objects shadowing a fragment. If precisely some integer multiple of $2^{\text{stencil bits}}$ objects shadow a point, it will appear unshadowed. Therefore no more than that many objects may be in line with each other and the light source. For an 8-bit stencil buffer, this limitation is unlikely to cause a problem for most scenes. One situation in which it may be a concern is that of a very long staircase with a light shining down the stairs. In this situation, the multiple folds of the staircase could cause every 256^{th} stair to appear unshadowed.

5. Hardware Capabilities

Our algorithm is designed for modern hardware, and therefore depends on certain features of that hardware. This section describes the required hardware capabilities and how to simulate them if they are not available.

The stencil buffer must support *two-sided stencil testing*, a feature available in both the DirectX 9.0 and OpenGL 1.4 APIs. This paper abstracts this functionality as a single function,

SetStencilOp(*FrontDepthFailOp, FrontDepthPassOp, BackDepthFailOp, BackDepthPassOp*), that has four parameters indicating how to update the stencil value. For each fragment, the rendering hardware selects the update method as follows:

Triangle is a front face, depth test failed for fragment	→ use FrontDepthFailOp
Triangle is a front face, depth test passed for fragment	→ use FrontDepthPassOp
Triangle is a back face, depth test failed for fragment	→ use BackDepthFailOp
Triangle is a back face, depth test passed for fragment	→ use BackDepthPassOp

The front/back face distinction is determined by the hardware face culling method and the vertex winding order and is relative to the viewer. The typical "stencil fail" parameter is ignored because the stencil test is configured to always pass for the algorithm. Each parameter assumes one of three values, KEEP, INCR, or DECR. KEEP leaves the stencil value unchanged. INCR increments the value using modular arithmetic. DECR decrements the value using modular arithmetic. Two-sided stencil testing can be simulated on hardware that does not support it by rendering all faces twice. The first rendering uses normal back face culling and sets the stencil operation for front faces. The second rendering reverses this; *front faces* are culled and the stencil operation is set for back faces.

We assume the presence of a simple programmable hardware vertex processing pipeline, without branches or loops. This is used to extrude the vertices on the dark cap. In the absence of such a pipeline, the model must be extruded to infinity away from the light every frame for every light on the main CPU and the rendering time is likely to be dominated by memory transfers for the resultant data.

We use the GL_EXT_depth_bounds_test extension to clip rasterization in depth and avoid rendering shadow volume geometry outside the effective radius of a light source. There is no way to efficiently simulate this extension on hardware that does not support it, however the test may simply be ignored without affecting correctness (forfeiting the performance gain, of course).

Finally, we require a rasterizer that correctly handles coordinates at infinity[2] ($w = 0$) and produces no underdraw (holes) or overdraw (doubly rendered pixels) along the edge between adjacent polygons. All of the capabilities described in this section are present in the current generation of consumer graphics hardware (e.g. NVIDIA GeForce and ATI Radeon graphics cards), except for the depth bounds test, which is available only on the NVIDIA GeForceFX 5900 cards.

6. Shadow Determination Algorithm

This section presents our algorithm for shadow determination for a single light source. The system should use the result to illuminate non-shadowed areas before iterating to the next light source in the scene. The algorithm assumes the following setup:

- Stencil buffer cleared to 0
- Entire scene rendered to the depth buffer
- Rendering to the stencil buffer enabled
- Rendering to the depth buffer disabled

[2] We note that triangles at infinity spanning more than 180 degrees about the viewer are ambiguous: they may be huge front faces or small backfaces, and suggest a workaround. Small triangles at infinity must still be rasterized correctly.

- Rendering to the color buffer disabled
- The EXTRUDE vertex program is loaded

The EXTRUDE vertex program transforms a vertex *V* to *V'* as follows:

$$V' = MT_{OC} \begin{cases} (V_x, V_y, V_z, 1) & V_w = 1 \\ (L_w V_x - L_x, L_w V_y - L_y, L_w V_z - L_z, 0) & V_w = 0 \end{cases}$$

where M is the projection matrix, T_{OC} is the object space to camera space matrix, and L is the light source position. The purpose of this vertex program is to extrude the second copy of the model geometry, which is stored in the latter half of the VAR array, away from the light source to infinity. Page 12 of Lengyel's Gamasutra article [LENG02] contains an implementation of this vertex program in the Cg shading language.

When the algorithm terminates, the result of shadow determination is stored in the stencil buffer. A fragment has stencil value of zero if it is illuminated and a non-zero value otherwise. The stencil test can be used to incorporate this result into a rendering system such as the one proposed in the following section.

To perform shadow determination for one light and many models:

```
void DetermineShadows(ModelArray shadowCasterArray, int lightID)
{
        // occlusion pyramid has its base at the viewport and tip at the light
1       Bounds occlusionPyramid = makeOcclusionPyramid(lightID);
2       Matrix projectionMatrix = getProjectionMatrix();
3       Bounds lightVolume = getLightBounds(lightID);
4       Point4 lightPos  = getLightPosition(lightID);

5       PointArray lightRegion = clipAndProjectToNearPlane(lightVolume);
6       PointArray scissorRegion = projectBoundsAgainstViewport(clippedLightVolume);

7       Bounds viewFrustum = getViewFrustum(scissorRegion);

8       if (scissorRegion.size() == 0) then return; else setScissorRegion(scissorRegion);

9       for (int i = 0; i < shadowCasterArray.size(); i++) {
10              Matrix worldTransform = shadowCasterArray[i].getWorldTransform();
11              Matrix objectToCamera = shadowCasterArray[I].getObjectToCameraTransform();
12              Bounds casterBounds = shadowCasterArray[i].getBounds();

13              if  (doBoundsIntersect(lightVolume, casterBounds)) then {
                        // This object's shadow may prevent illumination of other objects
14                      Point4 objectLightPos = worldTransform.invert().multiplyPoint(lightPos);
15                      EdgeArray silhouetteEdges = computeSilhouetteEdges(objectLightPos);

16                      loadExtrudeConstants(lightPos, projectionMatrix, objectToCamera);
17                      SetDepthBounds(projectionMatrix,
                                intersectBounds(lightVolume, casterBounds, occlusionPyramid));

18                      if  (doBoundsIntersect(casterBounds, occlusionBounds) == false) then {
                                // Case 1: Z-pass
19                              SetStencilOp(GL_INCR, GL_KEEP, GL_DECR, GL_KEEP);
```

```
20                              DrawSides(silhouetteEdges, lightPos,
                                        viewFrustum, shadowCasterArray[i]);
21                      } else {
                                // Cases 2 & 3: Z-fail
22                              SetStencilOp(GL_KEEP, GL_INCR, GL_KEEP, GL_DECR);
23                              DrawSides(silhouetteEdges, lightPos,
                                        viewFrustum, shadowCasterArray[i]);
24                              DrawDarkCap(silhouetteEdges, lightPos,
                                        viewFrustum, shadowCasterArray[i]);
25                              DrawLightCap(shadowCasterArray[i],
                                        casterBounds, viewFrustum);
                        }
                }
        }
}
```

Details of the helper procedures *ComputeSilhouetteEdges*, *SetDepthBounds, DrawSides*, *DrawDarkCap*, and *DrawLightCap* are presented following discussion of the algorithm.

The first few lines define several variables used in the equation. The only non-trivial ones are *lightRegion*, *scissorRegion* and *viewFrustum*. The *lightRegion* variable is the bounding volume of the light's useful illumination range, clipped to the near plane and projected onto the 2D image plane (viewport). This 2D region bounds the fragments that can receive illumination from the light—any fragment outside the region is definitely not illuminated. This region may extend beyond the boundaries of the screen, however. The *scissorRegion* is the intersection of *lightRegion* and the 2D viewport.

The bounding volume around the light is not visible if the *scissorRegion* is empty. Because the bounding volume contains all points that receive illumination from that light, if the bounding volume is not visible no illumination from the light will be seen on screen and all shadow determination for the light can stop. Line 8 performs this test and returns immediately if this is the case. If not, it calls *setScissorRegion*, effectively limiting the view frustum. Doing so can save significant fill rate when shadow volume geometry is rasterized because time is not wasted on shadow determination for fragments that cannot possibly be illuminated. In line 7 the viewFrustum variable is computed from this region so that geometry culling tests can have the benefit of the restricted frustum.

Line 14 begins the real work of the algorithm. In order for a caster's shadow to have an effect on the scene, the caster itself must receive some illumination. The conditional checks to see if the model is outside of the light's illumination range. If so then the shadow determination step can be skipped for this model. The exact method of determining the intersection of the bounding volumes depends on the types of volumes being used. Implementations of this and all other intersection computations used in the algorithm are provided in the C++ demo that accompanies this paper. If the conditional is satisfied, the algorithm proceeds to render the shadow volume of the caster. Otherwise it moves on to the next caster.

Line 15 computes the possible silhouette of the caster as observed from the light source. The result is a set of edges, which can be stored as an array of integer edge indices, since all possible silhouette edges are also in the model's edge array. Some edges are directed oppositely from the model. Our method makes copies of these edges but it is also possible to store the information without introducing new edges. Let a non-negative index e represent directed edge C.edge[e] and a negative index, $-e$ represent the reverse of directed edge C.edge[$-e-1$].

The possible silhouette computation is performed in the caster's object space because transforming the light to object space is faster than transforming the model to world space. This

possible silhouette determination step is the most computationally expensive part of the algorithm and is performed "in software" on the main processor. The remaining steps are dominated by hardware triangle rendering and place little demand on the main processor.

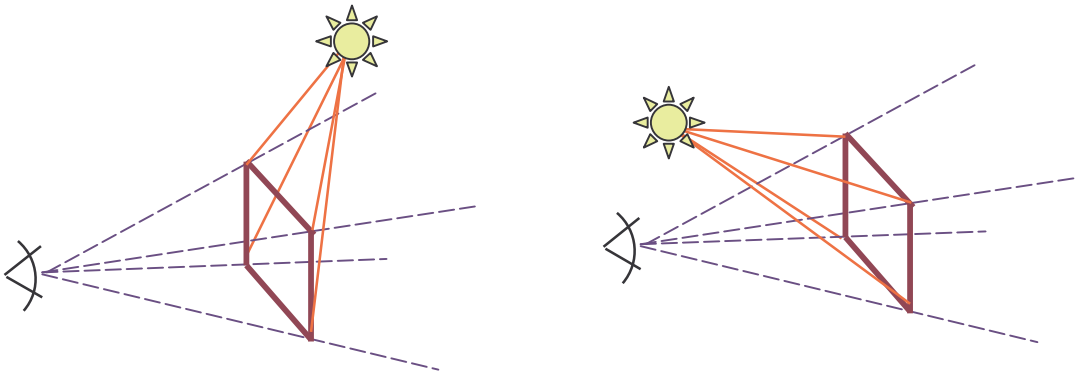


Figure 2: The viewport occlusion pyramid (solid red) extends from the light source to the viewport. Compare to the view frustum (dashed blue). As shown on the right, the occlusion pyramid may also be behind the viewport.

Recall that the vertex program EXTRUDE takes a vertex, projection matrix, and object to camera space transformation as input. Line 16 loads the current projection matrix and transformation into the vertex program "constant" inputs. The transformation objectToCamera variable is the OpenGL GL_MODELVIEW transformation that is the model's transformation composed with the world to camera space transformation matrix.

Line 17 sets a depth bound for stencil buffer writes as suggested by Everitt and Killgard [EVER03]. In *setDepthBounds* the nearest and farthest z-value for the visible illuminated region is recorded. This illuminated region can be computed by taking the intersection of the view frustrum, the light bounding volume, and the model bounding volume. Again computing the intersection of these volumes depends on the type of bounding volumes used. This depth test forces a check of the current depth buffer before updating the stencil buffer. All pixels that fail this check (pixels that are not influenced by the light) are skipped. This test can reduce the number of increments and decrements to the stencil buffer.

We are now ready to begin rendering shadow volume geometry. Line 18 distinguishes between case 1, where fast z-pass rendering can be used, and cases 2 and 3, which require robust z-fail rendering. If the intersection between the viewport's occlusion pyramid (Figure 2) and the caster's bounding box is empty it is impossible for the caster's shadow to fall across the viewport. In this case, z-pass rendering is used. Otherwise, the viewport may be partially or completely shadowed and z-fail rendering is required.

Lines 19 and 20 implement z-pass rendering. The stencil operation is set to **INCR**ement on shadow front faces that lie between the (known to be unshadowed) viewport and a visible point and **DECR**ement on back faces that lie in that region. When a shadow face fails the depth test it is beyond the visible point and the stencil value is unaltered. Only the sides of the shadow volume are drawn. With z-pass rendering the light cap will always fail the depth test because it is inside the caster, so it need not be rendered. The dark cap is always beyond every visible point in the scene. Thus, at every fragment it either fails the depth test or there is no visible point (i.e. the background, which cannot be shadowed, is directly visible) so there is no reason to render it either.

Lines 22 – 25 implement z-fail rendering. This covers both cases 2 and 3, which are discriminated only within *DrawLightCap*. The stencil operations are those of z-pass rendering, but with the sense of the depth test flipped so that we only alter a stencil value when the depth test

fails. All parts of the shadow volume may need to be drawn, so the final three lines render the sides, dark, and light caps. However, there is additional logic inside these procedures that attempts to cull unnecessary geometry.

We now examine the details of the helper procedures. *SetDepthBounds* finds the minimum and maximum z-values of the visible illuminated volume of a model. Any pixel values that have z-values out of this range are not affected by the light. As of the writing of this paper, the glDepthBoundsEXT functionality required by *SetDepthBounds* is available only on NVIDIA GeForceFX 5900 graphics card. If an implementation of this function is not available *SetDepthBounds* can be skipped, since this function is an optimization and does not affect the correctness of the algorithm.

```
void SetDepthBounds(Matrix projectMatrix,  Bounds bounds)
{
1       double minZ = 0, maxZ = 1;
2       PointArray hullVertices = bounds.getConvexHullVertices();
3       glEnable(GL_DEPTH_BOUNDS_TEST_EXT)

4       for (int i = 0; i < hullVertices.size(); i++) {
5               Point4 clipPoint = projectMatrix.multiplyPoint(hullVertices[i]);
6               minZ = MIN(clipPoint.z, minZ);    maxZ = MAX(clipPoint.z, maxZ);
        }
7       glDepthBoundsExt(minZ, maxZ);
}
```

ComputeSilhouetteEdges uses Sutherland's back face method for finding possible silhouettes, originally mentioned and proposed for this application by Crow [CROW77]:

```
EdgeArray ComputeSilhouetteEdges(Model caster, Point4 lightPos)
{
        // Note: the light is in object space
1       EdgeArray silhouetteEdges;

2       for (int i = 0; i < caster.tris.size(); i++) {
3               Triangle T = caster.tris[i];
4               T.lightFacing = dot(T.getNormal(), (lightPos.w * T.getPoint(0) – lightPos) ) > 0;
        }

5       for (int i = 0; i < caster.edges.size(); i++) {
6               Edge E = caster.edges[i];
7               if  (caster.tris[E.triIndex[0]].lightFacing  ^^
                                caster.tris[E.triIndex[1]].lightFacing) then {
8                       if  (E.triIndex[0].lightFacing) then {
9                               silhouetteEdges.append(E);
                        } else {
10                              silhouetteEdges.append(E.reverseIndices());
                        }
                }
        }
11      return silhouetteEdges;
}
```

Lines 2 - 4 find all faces of the caster that face the light (where a zero dot product is considered to face away from the light). The algorithm then iterates through all edges, selecting those that lie between a light face and a dark face. Edges are inserted into the possible silhouette set so that the curl of the possible silhouette is directed clockwise with respect to the light. When these possible silhouette edges are extruded to form the sides of the shadow volume, the shadow face normals will therefore be directed out of the shadow volume. As discussed previously, an alternative implementation is to store edge indices, appending index *e* if the edge is not reversed and index (-e-1) if the edge is reversed.

DrawSides takes the possible silhouette set, world space light position, and view frustum as input and renders the sides of the shadow volume to the stencil buffer. Line 2 computes a bounding volume for the shadow volume. In our implementation we use the convex hull of the union of the caster's bounding box and that box projected away from the light to infinity. Note that the shadow volume bound is large (one end is at infinity!) and if the light is near the caster the bound will contain nearly half the scene. If the light is inside the bounding box of the caster, it contains the entire scene.

The conditional on line 3 tests if the shadow volume sides can possibly affect the stencil values. This test has two parts. The shadow volume's bounding box must lie at least partly within the view.

```
void DrawSides(EdgeArray silhouetteEdges, Point4 lightPos,  Bounds viewFrustum,
                        Model caster)
{
        // Vertex index offset between projected and non-projected vertices in memory
1       int offset = caster.tris.size();
2       Bounds shadowVolumeBounds = caster.getShadowVolumeBounds();
3       if  (doBoundsIntersect(viewFrustum, shadowBounds) == false) then return;
4       if  (lightPos.w = 0) then {
                // Directional light.  The sides will converge to a single point, –L
5               for (int i = 0; i < silhouetteEdges.size(); i++) {
6                       DrawIndexedTri(caster.tris, E.vertIndex[0],
                                E.vertIndex[1], offset )
                }
        } else {
                // point light.  The sides extrude to infinity.
7               for (int i = 0; i < silhouetteEdges.size(); i++) {
8                       DrawIndexedQuad(caster.tris, E.vertIndex[0], E.vertIndex[1],
                                E.vertIndex[1] + offset, E.vertIndex[0] + offset)
                }
        }
}
```

If the conditional passes, the algorithm branches depending on whether the light source is a point light or directional light. Lines 5 and 6 handle the directional light case. A directional light produces a shadow volume that converges to a single point at infinity. That is, the entire second half of the vertex array will all transform to a single point under the EXTRUDE vertex program (recall that the vertex array has size 2**numCasterVertices*). The sides of the shadow volume are triangles between that point and the possible silhouette edges. To render them, we iterate over the possible silhouette edges and connect them to VAR[*numCasterVertices*], the first

point in the second half of the VAR array. Any other index greater than or equal to *numCasterVertices* would suffice.

If the light is instead a point light, the shadow volume diverges as it approaches infinity. In this case, the side faces are quadrilaterals constructed by connecting the possible silhouette edges to their oppositely oriented counterparts at infinity. Lines 7 and 8 draw these quadrilaterals.

DrawDarkCap is only invoked when z-fail rendering (cases 2 and 3) are used. It draws the cap that closes the shadow volume at infinity. Because a directional light produces a shadow volume that converges to a point, such a cap is only needed for point lights, which are discriminated by the L.w = 1 test on line 3. Furthermore, the dark cap need only be rendered when its bounding box intersects the view frustum. Line 2 computes the bounding box of the dark cap as the bounding box of the caster extruded to infinity away from the light.

If all tests pass, lines 4 - 6 draw the cap. Our method for rendering the dark cap draws only as many triangles as there are edges in the possible silhouette. The dark cap is at infinity, beyond everything in the scene. Therefore its shape does not matter because no object could possibly be inside it and be shadowed. Since we are free to change the shape (provided we keep it beyond infinity), only one constraint remains: the cap must actually close the shadow volume. A simple fan over the sides accomplishes this. Observe that for a non-convex possible silhouette, the fan will be convex but the pattern of front and back faces will produce a non-convex shape in the stencil buffer. This technique has previously been used for rendering filled silhouettes in the stencil buffer from possible silhouette edges for Silhouette Clipping [Sand00].

The process is identical to that of computing the signed area of a polygon by constructing a fan. Figure 3 shows an example. The concave polygon on the left is tessellated into a fan that is treated as two sets: front faces (shown yellow) and back faces (cyan). One of the triangles in the front face set is shown as dark yellow to represent the double coverage of that part of the front face set. When the back faces are subtracted from the front faces, the result is the original polygon with every part covered exactly once.

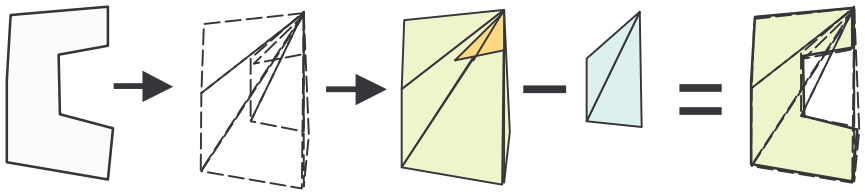


Figure 3. A signed area triangle fan covering a non-convex region.

```
void DrawDarkCap( EdgeArray silhouetteEdges, Point4 lightPos, Bounds viewFrustum,
                  Model caster)
{
1       int offset = caster.tris.size();
2       Bounds darkCapBounds = caster.getDarkCapBounds();

3       if  ((L.w == 1) && doBoundsIntersect(viewFrustum, darkCapBounds)) then {
                // point light; we don't draw for a directional light
4               for (int i = 0; i < silhouetteEdges.size(); i++) {
5                       Edge E = silhouetteEdges[i];
6                       drawIndexedTri(caster.tris, offset, E.vertexIndex[0] + offset,
                                E.vertexIndex[1] + offset);
                }
        }
```

```
}
```

Unfortunately, this technique can produce triangles at infinity with edges spanning 180 degrees or more of the sphere at infinity. These triangles will be interpreted as backfacing by OpenGL and will not be rendered like the other triangles. It is therefore necessary to implement *DrawIndexedTri* to detect such a case and subdivide the large triangles. The simplest method for this is to switch from the indexed vertex buffer to standard *glVertex* primitive calls for those triangles only.

Because the possible silhouette edges are oriented clockwise from the light's point of view, the dark cap uses the given direction edges, forming polygons where the surface normals point away from the light (out of the infinite sphere).

```
void DrawLightCap(Model caster, Frustum viewFrustum)
{
1       Bounds casterBounds = caster.getCasterBounds();
2       if (doBoundsIntersect(viewFrustum, casterBounds)) then {
                // Case 3
3               setDepthTest(GL_NEVER);
4               for (int i = 0; i < caster.tris.size(); i++) {
5                       Triangle T = caster.tris[i];
6                       if  (T.lightFacing) then {
7                               DrawIndexedTri(caster.tris, T.index[0], T.index[1], T.index[2]);
                        }
                }
8               setDepthTest(GL_LESS)
        }
}
```

DrawLightCap is invoked for z-fail rendering. It distinguishes between cases 2 and 3 in line 2 and returns immediately if the caster's bounding box is not inside the view frustum (case 2). The depth test can be disabled for the light cap (set to always fail, in this case) because we know that it will always fail. For non-convex geometry, the light cap cannot be simplified as can the dark cap because the shape is constrained. If we alter the shape of the cap it may change the set of visible points that are in shadow. Therefore we are left to simply draw all of the light facing triangles on the caster in lines 4 – 7.

7. Analysis

This section presents an asymptotic analysis of the number of triangles rendered by our shadow determination algorithm and some thoughts on the performance likely to be observed in practice. This analysis is interesting for applications with performance limited by vertex processing rate. This can occur when an application renders a huge amount of geometry (e.g. the oil rig model used in radiosity and level of detail papers), has a complex vertex shader (e.g. displacement mapping), or is running vertex shaders on older hardware. These situations can easily occur today for scientific and engineering applications. We observe that most 3D games produced today are limited by fill rate and not vertex rate. The analysis in this section is therefore not relevant to understanding the performance of modern games. However, trends indicate that games may also become vertex rate limited in the future. Very recent graphics cards provide massive fill rate capability compared to previous hardware, such that they can now render full-

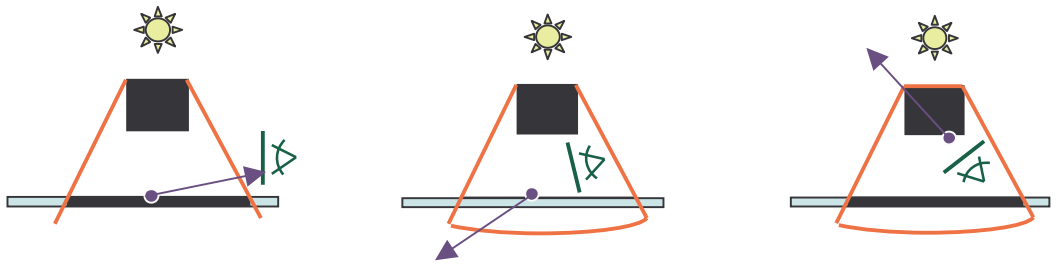
screen 3D scenes at 1280x1024 resolution with antialiasing at high frame rates. Higher resolutions may not increase the quality of rendered images substantially compared to more complex geometry. Increasingly interesting programmable capabilities encourage game developers to write more complicated vertex programs. Displacement mapping, lighting, skeletal animation, physics, and dynamic tessellation all consume vertex processing time.

Consider a single shadow-casting model composed of f triangles. It is expected that half the triangles are light facing. The possible silhouette of the model from the perspective of the light has s edges, where $s < f$ (and frequently, $s << f$). The sides of the shadow volume therefore contain s quads, the light cap contains expected $f / 2$ triangles, and the dark cap contains s triangles.

Figure 4 shows the three major cases of the algorithm. In *case 1* the viewport is unshadowed and z-pass rendering is used. At most only the shadow volume sides are drawn so O(s) triangles are rendered.

Case 2 occurs when the viewport is shadowed but the viewer is looking away from the light (i.e. the light cap is culled by the view frustum). At most, the dark cap and sides are drawn so again O(s) triangles are rendered.

Case 3 occurs when the viewport is shadowed and the viewer is looking toward the light. The light cap is not culled, so O(f) triangles are rendered. The dark cap and sides may contribute at most another O(s) triangles which cannot affect the asymptotic performance because $s < f$.



Case 1: Unshadowed viewport

Case 2: Shadowed viewport, looking **away** from light

Case 3: Shadowed viewport, looking **toward** light

Figure 4. Three cases distinguished by our algorithm.

A general theoretical analysis of the probability of each case occurring is difficult. If the light source, viewer, and shadow caster are placed randomly within a sphere that is large compared to the size of the caster, the probability of case 1 approaches unity. This is because the shadow volume of the caster is extremely narrow compared to the volume of the sphere bounding the scene. If the scene is constrained to fit within a sphere that is small compared to the caster, the probability of case 1 approaches zero because the light is most probably inside the caster, causing the shadow to encompass the entire scene.

Most scenes encountered in practice have a small number of overhead light sources with a small number of shadow casters above the viewer (and thus, potentially shadowing the viewer). Case 1 is extremely common for such scenes. When the viewport becomes shadowed it is typically because the viewpoint has moved inside a building or tunnel. In this case, the viewer is still unlikely to be looking upwards, so case 2 dominates for the building model. Note that objects inside the building are still in case 1. Only when the viewer looks toward a light source that is occluded (e.g. an object silhouetted against the sun) is case 3 invoked.

An exception to this is a natural scene with sunrise and sunset, when a light source (the sun) lies near the horizon. Shadow volumes will extend across the entire scene, and it is likely for

the viewer to be observing the rise or fall of the sun. When this occurs, any object along the path between the viewer and the sun is in case 3.

A triangle with vertices represented as three 32-bit floating point numbers requires 36 bytes to transfer between the main CPU and graphics hardware. We avoid transferring vertex data by using a double length vertex buffers and a vertex program to perform extrusion in hardware. This allows us to specify a vertex with a single 16-bit indices and each triangle in only 6 bytes.

We implemented our algorithm and applied to an interactive rendering of a scene. Our scene was constructed to stress the algorithm and is thus not unduly 'nice'. It contains animated models, large staircases, and a 3D grid of cubes with a light source moving between them. We found that most objects were still in the fast cases 1 and 2, and that we were able to effectively cull much shadow volume geometry in software, without the need to transfer it to the graphics hardware and rasterized it. Our algorithm performed strictly faster than Lengyel's algorithm for the same scene, which is unsurprising because we build on Lengyel's optimizations.

8. A Rendering System

This section describes how to incorporate a shadow determination algorithm into a rendering system by adding a corresponding illumination pass. The rendering system makes four kinds of rendering passes. An initial rendering pass renders the scene with ambient illumination only, and writes to the depth buffer. Subsequent passes will test against but not write to the depth buffer. A shadow determination and illumination pass are made for each light. A final rendering pass is used as a catch-all for effects like transparency that are incompatible with shadow rendering.

```
RenderFrame():
    AmbientPass()
    For each light L:
        DetermineShadows( L )
        Illuminate( L )
    FinalPass()
```

The ambient and illuminate passes are typical rendering code. They iterate through all models in the scene and render them using the current lighting, which is set to either ambient or a single light. The Illuminate pass should only illuminate areas that are not in shadow. Because the result of shadow determination is stored in the stencil buffer, the stencil test accomplishes this. We set the stencil test to pass when the stencil value is zero and fail when the stencil value is non-zero.

The observed illumination on a surface is the sum of the contributions from each light. We must therefore sum the results from the individual rendering passes to produces the net illumination for the scene. Haeberli [HAEB90] introduced the hardware accumulation buffer for performing this kind of operation. However, consumer graphics cards do not currently support accumulation buffers. We can emulate an accumulation buffer in for the purposes of illumination by using additive alpha blending. Setting the alpha blending function to $DST = SRC + DST$ (glBlendFunc(GL_ONE, GL_ONE) in the OpenGL API) causes the *Illuminate* passes to add their result to the frame buffer rather than overwrite the previous contents. Haeberli also noted that accumulation-buffer functionality can be used to model area and volume light sources (for casting soft shadows). These can be approximated with a large number of dim point lights or spotlights. A drawback to this method, and the multipass rendering method in general, is that color resolution is decreased with every rendering pass due to rounding off of intermediate results. We

have observed that about 20 light sources can illuminate a single point with an 8-bit per color channel frame buffer before the color banding becomes a significant artifact.

When the illumination pass completes it must clear the stencil buffer and set up all rendering state as described in the assumptions for the *DetermineShadows* procedure.

Some rendering effects do not integrate well with shadows. For example, transparency and 3D annotations (e.g. labels) that the programmer may not want to receive shadows. For these effects we add an additional *final pass*. Another use for the final pass is to limit texturing to once per frame instead of performing it once per illumination pass. This is especially important on older hardware that may not be enough texture units to perform lighting and color texturing in a single illumination pass. For diffuse surfaces we render the illumination passes with all models colored pure white. Then in the final pass, the accumulated illumination is modulated by the appropriate texture maps. This also preserves some color resolution. This process will not work for objects with specular highlights because those highlights are not be modulated by a model's texture map. To accommodate such surfaces, separate diffuse and specular images must be rendered and composed to form the final image.

Note that because this system uses shadow determination to suppress illumination, the "shadows" are never rendered. Rather illumination is added to the area around shadows. As a result, shadows from colored lights and lights with projected textures are rendered correctly. Figure 5 shows an image from our implementation that demonstrates the interaction of colored lights. Three animated Quake III format models walk down a short staircase outside a temple (there are much longer staircases inside the temple). The scene is illuminated by three point lights. A green light is to the characters' right, a white light immediately in front of them, and a red light to their left. The shadow volumes from the red light are rendered explicitly to help the viewer visualize them; normally shadow volumes are invisible. The source code for this demo can be found on our website listed at the end of this paper.

Figure 5. Three colored light sources and visualized shadow volumes.

Acknowledgments. Seth Block of Brown University worked with us on the C++ implementation. Eric Haines (Autodesk), Frank Crow (NVIDIA), and Ben Landon (Curl) provided feedback and further information on shadow techniques. John Carmack (id) and Max McGuire (Iron Lore) discussed their own experience with shadow volumes in their respective video games. Graphics hardware donated by ATI and NVIDIA was used to develop and test the algorithm. The Tick model was created by Carl Schell (carl@cschell.com) and is used with permission. The Tick character is owned by New England Comics.

References

AKEL93 K. Akeley. Simple capping demonstration. Appeared in SGI Developer's Toolbox, Release 3.0, January 1993.

ASSA03a U. Assarsson and T. Akenine-Möller, "*A Geometry-based Soft Shadow Volume Algorithm using Graphics Hardware*," to appear in SIGGRAPH 2003.

ASSA03b U. Assarsson, M. Dougherty, M. Mounier, and T. Akenine-Möller, "*An Optimized Soft Shadow Volume Algorithm with Real-Time Performance*", to appear in Graphics Hardware 2003.

BARE01 G. Barequet, C.A. Duncan, M. T. Goodrich, W. Huang, S. Kumar, M. Pop, "*Efficient Perspective-Accurate Silhouette Computation,*" Proc. 17th Ann. ACM Symp. on Computational Geometry (SoCG), Medford, MA, pp. 60-68, June 2001

BILO99 B. Bilodeau and M. Songy, talk at Creative Labs Inc. sponsored game developer conference, Los Angeles, May 1999.

BRAB03 S. Brabec and H. Seidel, "*Shadow Volumes on Programmable Graphics Hardware,*" to appear in Eurographics 2003 (Computer Graphics Forum).

CARM00 J. Carmack. E-mail to private list May 23, 2000. Published on the NVIDIA website "http://developer.nvidia.com/docs/IO/2585/ATT/CarmackOnShadowVolumes.txt".

CROW77 F. C. Crow. "*Shadow Algorithms for Computer Graphics*." Computer Graphics (SIGGRAPH '77 Proceedings), vol 11, no. 2, pp. 242-248, July 1977.

DIEF96 Diefenbach, P. *Multi-pass Pipeline Rendering: Interaction and Realism through Hardware Provisions,* Ph.D. thesis, University of Pennsylvania, tech report MS-CIS-96-26, 1996.

EVER02 C. Everitt and M. J. Kilgard. "*Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering*." NVIDIA Corporation, Austin, TX. March 12, 2002, "http://developer.nvidia.com/view.asp?IO=robust_shadow_volumes".

EVER03 C. Everitt and M. J. Kilgard. "*Optimized Stencil Shadow Volumes*." Slides from Game Developer Conference, San Jose, March 2003, http://developer.nvidia.com/docs/IO/4449/SUPP/GDC2003_ShadowVolumes.pdf.

GREE93 N. Greene, M. Kass, and G. Miller. "*Hierarchical Z-buffer visibility*." Computer Graphics (SIGGRAPH '93), vol 27, (Annual Conference Series), pp 231-238, 1993.

HAEB90 P. Haeberli P. and K. Akeley. "*The Accumulation Buffer: Hardware Support for High-Quality Rendering*." Computer Graphics (SIGGRAPH '90), vol. 24, no. 4, 309-318, August 1990.

HEID91 T. Heidmann.. "*Real Shadows, Real Time*," Iris Universe, No. 18, pp. 23-31, Silicon Graphics Inc., November 1991.
http://developer.nvidia.com/docs/IO/2585/ATT/RealShadowsRealTime.pdf

KILG02 M. J. Kilgard. "*Robust Shadow Volumes*." Presented at the Games Developer Conference 2002. http://developer.nvidia.com/docs/IO/3128/ATT/SIGGRAPH2002_RobustShadowVolumes.ppt

LENG02 E. Lengyel.. "*The Mechanics of Robust Stencil Shadows*." Gamasutra, October 11, 2002, "http://www.gamasutra.com/features/20021011/lengyel_01.htm".

MCCO98 M. McCool. "*Shadow Volume Reconstruction*." Technical Report CS-98-06, University of Waterloo Department of Computer Science, 1998.

MÖLL99 T. Möller, T. and E. Haines. *Real-Time Rendering*. A K Peters, 1999.

SAND00 P. V. Sander, X. Gu, S. J. Gortler, H. Hoppe and J. Snyder. "*Silhouette Clipping.*" SIGGRAPH 2000. July 2000.

WILL78 L. Williams. "*Casting Curved Shadows on Curved Surfaces."* Computer Graphics (SIGGRAPH '78 Proceedings), vol. 12. no. 3 August 1978.

Appendix: API resources

Our algorithm depends on some relatively new hardware rendering functionality. Rather than attempt to explain these capabilities ourselves, we refer the reader to the following reference material on the web.

Vertex programs

DirectX
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/programmingguide/Programmable/VertexShaders/VertexShaders.asp

OpenGL
http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt
http://oss.sgi.com/projects/ogl-sample/registry/NV/vertex_program2.txt
http://www.sci.utah.edu/~lefohn/work/shadingLang/oglLang.pdf

Cg (NVIDIA)
http://www.cgshaders.org/

Two-sided stencil

DirectX
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/programmingguide/advancedtopics/TwoSidedStencil.asp

OpenGL
http://oss.sgi.com/projects/ogl-sample/registry/EXT/stencil_two_side.txt

Depth Bounds

http://www.nvidia.com/dev_content/nvopenglspecs/GL_EXT_depth_bounds_test.txt

Vertex Array [Vertex Buffer]

DirectX

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/d3d/interfaces/idirect3ddevice9/CreateIndexBuffer.asp

OpenGL
http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt
http://www.opengl.org/developers/documentation/OGL_userguide/OpenGLonWin-15.html
http://oss.sgi.com/projects/ogl-sample/registry/ATI/vertex_array_object.txt
http://oss.sgi.com/projects/ogl-sample/registry/APPLE/vertex_array_object.txt
http://oss.sgi.com/projects/ogl-sample/registry/NV/vertex_array_range.txt

Web Information:

http://www.cs.brown.edu/research/graphics/games

Morgan McGuire, John F. Hughes, Kevin T. Egan, Department of Computer Science, Brown University, P.O. Box 1910, Providence, RI. (morgan, jfh, ktegan@cs.brown.edu)

Mark Kilgard, Cass Everitt, NVIDIA Corporation (mjk, ceveritt@nvidia.com)