Abstract

Shadows are important elements in three-dimensional computer graphics. They enhance the level of realism and provide visual cues that help determine spatial relationships. Point light sources generate only fully shadowed regions, i.e., there is an abrupt transition from no shadow to full shadow. These shadows are often referred to as hard shadows. Soft shadows on the contrary, are produced when area or volumetric light sources are present in a scene. Each shadow can have a fully shadowed region, called the umbra, and a partially shadowed region, called the penumbra. Soft shadows are generally preferable, because they let the viewer know that the shadow is indeed a shadow and not actual geometrical features. Soft shadows are also more realistic, and they further help in understanding distances in a scene.

This thesis focuses on an OpenGL implementation of the soft shadow volume algorithm presented by Assarsson and Akenine-Möller [1, 2]. The algorithm extends the well-known shadow volume algorithm for hard shadows presented in 1977 by Crow [3], and consists of two passes. The first pass uses the classic shadow volume algorithm to generate hard shadows as an approximation of the umbra region; the second pass provides the softness of the shadow, the penumbra region. This is achieved by generating penumbra wedges and rasterizing them using a fragment program that computes the light intensity for each pixel from the uncovered area of the light source.

The result of this thesis is an OpenGL implementation capable of real-time soft shadow volumes. The implementation produces high-quality shadows for area and volumetric light sources that cast properly on arbitrary surfaces. A simple load-balancing scheme that enables trade-off between quality and performance is also presented.

Sammanfattning

Skuggor är ett viktigt inslag i rendering av tredimensionell datorgrafik. De förstärker realismen och förser användaren med viktig information om spatiala förhållanden och positioner. Punktformiga ljuskällor genererar enbart fullt skuggade områden, vilket innebär att det finns en diskontinuerlig övergång mellan skugga och icke-skugga. Dessa skuggor kallas ofta hårda skuggor. Mjuka skuggor uppstår när ljuskällor har en utbredning, i form av antingen en area eller en volym. Varje skugga kan ha en fullt skuggad del, kallad kärnskugga, och en delvis skuggad del, kallad halvskugga. Mjuka skuggor är oftast att föredra, eftersom hårda skuggor lätt kan misstas för geometriska detaljer.

Den här rapporten fokuserar på en implementation i OpenGL av den algoritm för mjuka skuggor som utvecklats av Assarsson och Akenine-Möller [1, 2]. Algoritmen består av två renderingspass och bygger på Crows välkända algoritm för skuggvolymer från 1977 för hårda skuggor [3]. Det första passet genererar en överapproximation av kärnskuggan med hjälp av ovanstående algoritm, varefter det andra passet bidrar med den mjuka delen av skuggan, halvskuggan. Denna rastreras med ett fragmentprogram som beräknar ljusintensiteten för varje pixel utifrån den synliga delen av ljuskällan.

Resultatet av detta examensarbete är en OpenGL-implementation av mjuka volumetriska skuggor i realtid. Implementationen genererar högkvalitativa skuggor som kastas på godtyckliga ytor för ljuskällor med area eller volym. Dessutom presenteras en enkel lastbalanseringsstrategi som möjliggör avvägning mellan kvalitet och prestanda.

Table of Contents

1 Introduction

Shadows enhance realism and provide important visual cues about spatial relationships. Without shadows, images tend to look flat and it is difficult to determine sizes and positions of objects in the scene.

Currently most shadow algorithms for real-time applications are limited to hard shadows, which are generated by point light sources. Real world light sources have an area or volume, thus hard shadows do not occur in reality. It should be noted though, that the soft region of the shadow can be small, and therefore hard shadows can sometimes be a reasonable approximation.

Area and volumetric light sources produce soft shadows, which have a smooth transition from no shadow to full shadow. The fully shadowed region is called the umbra, and the transition is called the penumbra. The umbra region of a soft shadow (Figure 2) is not equivalent to a hard shadow generated by a point light source (Figure 1); instead the umbra region of a soft shadow is decreasing in size the larger the light source.

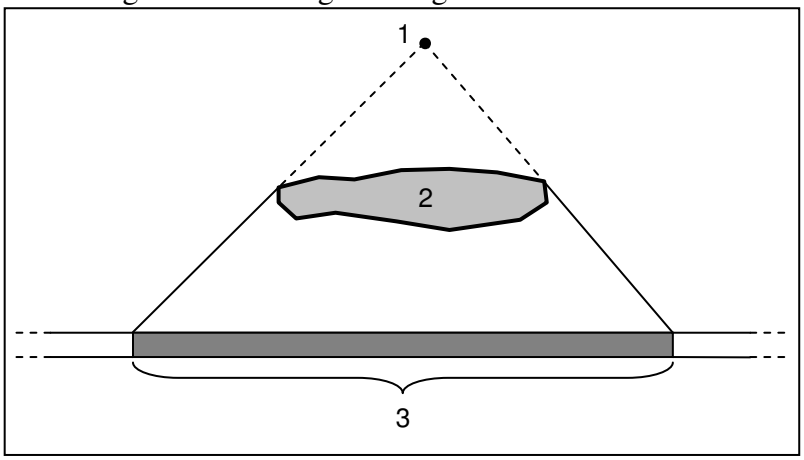

Figure 1: Hard shadows. 1) Light source. 2) Shadow caster. 3) Umbra region.

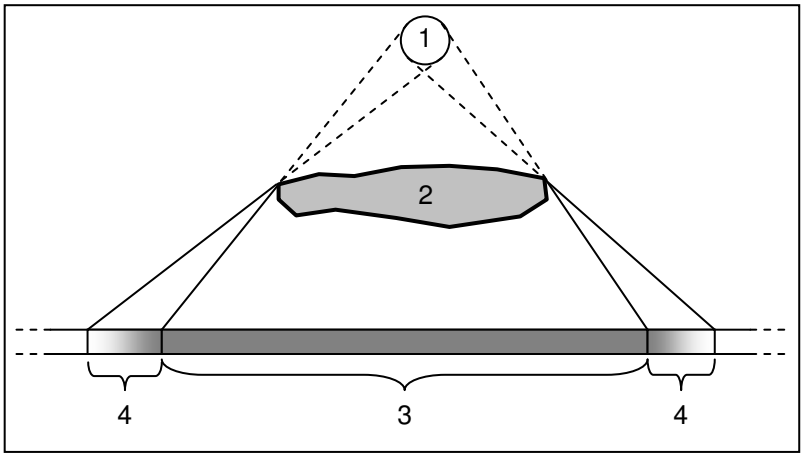

Figure 2: Soft shadows. 1) Light source. 2) Shadow caster. 3) Umbra region. 4) Penumbra region. Note how the umbra region decreases in size the larger the light source.

The visual quality of soft shadows compared to that of hard shadows is very high, as shown in Appendix A. It is thus desirable to be able to apply real-time soft shadows in real-time applications such as computer games.

This report presents an optimized implementation using OpenGL of the soft shadow volume algorithm presented by Assarsson and Akenine-Möller [1, 2], and a suggestion for a simple speed-up technique. Our main goal has been to show that the algorithm is well suited for games and other real-time applications were many calculations such as physical simulation and AI occupy the CPU.
In writing this thesis, we have assumed that the reader is familiar with common computer graphics (CG) terms and concepts.

2 Thesis Organization

First we give a brief presentation of previous work and real-time shadow techniques, and then follow a section describing our implementation. Then follow a discussion, a summary of results and suggestions for future work that would improve the soft shadow volume technique.

3 Previous Work

Various rendering techniques such as ray tracing [4], photon mapping [5] and radiosity [6] compute approximated lighting by considering a visibility function. For these techniques the visibility functions are simple to implement, making it trivial to produce computer generated images where shadows are included. However, the only feasible solution for real-time rendering is currently rasterization of triangles.

In this chapter we shall see that various algorithms exist that extend the rasterization approach with a visibility function for direct illumination. We first give an overview of the most common real-time shadow algorithms. We focus on the shadow volume algorithm, which currently is used in many real-time applications.

3.1 Real-Time Rendering of Shadows

3.1.1 Hard Shadows

Overview

There are many different shadow techniques with real-time performance. Simple shadows such as a dark rectangle under an object has been used in games for a long time since any shadow often is better than no shadow at all.

One of the major problems with the approach above is that the shadows do not cast properly on objects or walls. There are many algorithms that behave this way, and operate in scenes with very restrictive assumptions. Another example of this is the projective shadow algorithm [7], which assumes that the shadow receiver is a plane with known orientation and position.

The two most widely used techniques that are able to cast shadows on arbitrary geometry are *shadow volumes* and *shadow mapping*. The shadow mapping algorithm [8], renders a depth image, called a shadow map, as seen from the light source. Shadows are then created by rendering the scene from the eye, and for every pixel its distance to the light source is compared to the depth value in the shadow map, which determines whether the point is in shadow or not. The shadow mapping algorithm was first introduced by Williams in 1978 [8], and one of the most important additions to it has been the paper by Segal et al. [9], in which it is noted that the required computations are very similar to the computations already implemented in hardware. Since shadow maps can be hardware-accelerated it is fast, and many computer games use it. The technique has one major drawback; the discretization and limited precision of the shadow map can result in aliasing artifacts in the form of jagged shadow edges. The pixel precision seen in shadow volumes have not yet been achieved with shadow maps although several attempts have been made [10, 11, 12]. A detailed description of the shadow volume algorithm [3] follows in the next section.

Hard Shadow Volumes

A method based on Crow's shadow volume algorithm [3] uses the stencil buffer to cast shadows onto arbitrary objects. The technique is called shadow volumes, or volumetric shadows. To understand what a shadow volume is, imagine a point light source positioned above a triangle. Extending vectors from the light source through the vertices of the triangles yields a pyramid. The part of the pyramid under the triangle describes a volume in which all points are in shadow,

hence the name, shadow volume.
The classic version of the algorithm works as follows [13]. First, the stencil buffer is cleared. Then the whole scene is rendered using only ambient lighting, and the depth information of the scene is rendered into the Z-buffer. A pass follows where each silhouette edge of an object as seen from the light source creates a shadow volume quadrilateral which is rendered to the stencil buffer, using only Z-buffer testing. Quads facing the viewer that pass the depth test add one to the stencil buffer, while back facing quads subtract one. This means that the stencil buffer contains a mask where zero indicates no shadow, and non-zero indicates shadow. Finally the whole scene is rendered with specular and diffuse lighting where the stencil buffer is zero.
The shadow volume technique works well in most cases, but there are some limitations. Since the stencil buffer only stores one object's shadow state per pixel, transparent objects cannot receive shadows correctly. The algorithm also assumes that shadow casters consist of opaque triangles and that light sources are modeled as point light sources.

Problems

The stencil shadow volume algorithm as described above is quite fast, but it has a serious drawback that has limited its use for years. The problem with the original algorithm for shadow volumes [3], is that it does not properly handle the case when the viewer is inside a shadow volume, since shadow volumes will be clipped against the near plane, resulting in incorrect shadows. Below follows a description of how this problem can be solved. For further improvements of the algorithm, we refer to the techniques presented by Everitt and Kilgard for optimizing shadow volumes [14].
For shadow volume algorithms to work properly, the silhouettes of the shadow casting objects with respect to the position of the light source must be closed loops. This means that the objects must be closed, sometimes referred to as 2-manifold. Most polygonal objects are not closed; hence they must be capped. In section 4.3 we suggest a simple approach to cap triangular meshes in order to avoid erratic shadows. Another method is to add/subtract 2 for non-open edges and add/subtract 1 for open edges [15].

Eye Inside Shadow Regions: Z-fail vs. Z-pass

In 2000, John Carmack presented a solution to the problem, known as the *Z-fail algorithm*, or *Carmack's reverse* [16]. The idea is to render the shadow volumes that are obscured by visible geometry. Z-fail was also independently discovered by Billodeau and Songy [17].
In the first pass, all back facing shadow volume quadrilaterals are rendered and the stencil buffer is incremented when the polygon is equal to or farther than the stored depth value. The second pass renders all front facing quads and decrements the stencil count when the polygon is equal to or farther than the stored depth value. The algorithm is called Z-fail because the shadow volumes are drawn only when the Z-buffer test has failed.
In the original algorithm, called the Z-pass algorithm, a point is found to be in shadow if the number of intersections with frontfacing polygons and a virtual ray from the eye is larger than the number backfacing polygons intersecting the same virtual ray. In the Z-fail version, a point is in shadow if the number of backfacing polygons not seen is larger than the number of frontfacing polygons not seen. The difference is that all shadow volume quads in front of surfaces, including the ones which could encompass the eye, are not rendered, so most viewer location problems are solved.
Now that the depth test has been changed, it is necessary to render the original polygons on the

caster generating the quadrilaterals to properly maintain the count. The shadow volumes must be closed, *capped*, at their far ends, and this cap must be inside the far plane to avoid causing shadowing errors.
Generating the required top and bottom cap for an arbitrary mesh can be complicated, but assuming a closed mesh it is much simpler. For a closed mesh we can use the front facing triangles, relative to the light, as the top cap. The bottom cap can be generated from the back facing triangles by projecting them away from the light.
The changes to the original algorithm described above enable correct shadow calculation when the eye is inside a shadow volume and is a very robust and practical algorithm. The only drawback using Z-fail is that in most cases, z-pass will fill fewer pixels overall. Therefore, Z-fail may be slower, and Z-pass should be used whenever possible.

3.1.2 Soft Shadows

Overview

The shadows generated by the techniques described in the previous sections do not handle soft shadows; there is no smooth transition from no shadow to full shadow. This is a consequence of the limitation of the techniques: light sources must be points. Real world shadows are in most cases soft, with a smooth transition from no shadow to full shadow, since light sources have an area or a volume. Such light sources give rise to soft shadows since there are points where only a part of the light is visible. These points constitute the penumbra region. The set of points where the light is completely obscured and the geometry is in full shadow is called the umbra region.
Chan and Durand at MIT have an approach that builds on the shadow map algorithm by attaching geometric primitives, called *smoothies* [18], to the objects' silhouettes. These smoothies give rise to fake shadows that appear like soft shadows, at a low cost. The soft shadow edges hide some of the aliasing artifacts that are noticeable with ordinary shadow maps. Wyman and Hansen approximate soft shadows by introducing the *penumbra map* [19] that also extend the shadow map technique. This penumbra map is generated by using the objects' silhouette edges as seen from the center of an area light. Both algorithms allow arbitrary dynamic models that shadow themselves and their environment.
The soft shadow volume algorithm using penumbra wedges [20] extends the hard shadow volume algorithm so that area or volume light sources can be used. The algorithm renders soft shadows in real time from arbitrary shadow casters on arbitrary surfaces. We will present a brief description of the soft shadow volume algorithm. A great survey on real-time soft shadow algorithms is presented by Hasenfratz et al [21].

Soft Shadows Using Penumbra Wedges

The algorithm introduces a new rendering primitive, the penumbra wedge, illustrated in Figure 3. These wedges are used in the computation of a visibility buffer (V-buffer), which is used as a mask to add shadows to the final image.
To compute this V-buffer, the hard shadow quadrilaterals are rendered into the buffer by using the standard shadow volume algorithm for hard shadows. The silhouette edges are then used to create penumbra wedges, which encompass the penumbra volumes. The wedges are split in two by the hard shadow quads, which yields an inner and outer half.
For all points located in the inner half of the wedge, a fragment program computes how much each point can see of the light source, with respect to the silhouette edge of the wedge. This

fraction is added to the V-buffer, and compensates for the over-estimated umbra created by the hard shadow pass.
For points in the outer half of the wedge, the fragment program computes how much of the light source that is covered with respect to the silhouette edge, and this fraction is subtracted from the V-buffer.
The effect of this calculation is a visibility mask that represents the shadow containing both umbra and penumbra regions.

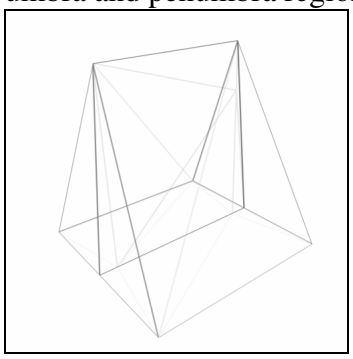

Figure 3: Penumbra Wedge.

4 A Soft Shadow Rendering Framework

4.1 Overview

One goal of our work has been to implement the soft shadow volume algorithm efficiently in OpenGL, and show its fitness for real-time applications such as games. Software of this type requires a lot of supporting code, e.g. window and input management, timing, camera handling, and scene representation. This functionality is implemented in a separate framework, independent of the actual shadow algorithm. Much of our work has been focused on developing this framework. The implementation is based on a series of articles, [1, 2, 22, 23], in which the technique described above as the *soft shadow volume algorithm* was presented by the authors.
In this section follows a description on how to implement the soft shadow volume algorithm using OpenGL. Our implementation is based on the shaders from the original DirectX implementation of the algorithm by Dougherty and Mounier[1], and handles non-textured spherical and rectangular light sources as well as textured rectangular light sources. The main difference between these three types is the fragment programs rasterizing the penumbra wedges. These fragment programs will be presented in detail. We suggest an automatic capping algorithm which enables correct shadows from arbitrary polygonal objects. We also suggest a simple speedup technique which can greatly reduce the number of pixels rasterized for each penumbra wedge.

4.2 Soft Shadow Volumes in OpenGL

4.2.1 Introduction

This section describes a way to implement soft shadow volumes in OpenGL. Our implementation runs in real time, but there are some problems related to limitations on current graphics hardware that have serious impact on performance. However, these problems are likely to disappear within the next few generations of graphics cards. In this section we discuss some of the identified problems along with the solutions we have applied to implement the algorithm on current hardware.
The standard stencil shadow volume algorithm requires a stencil buffer that holds information for every pixel whether it is in shadow or not. The soft shadow volume algorithm requires a similar buffer, but it must be able to represent several levels of intensity for the shadow. This visibility buffer (V-buffer) is the essential part of the algorithm and we will therefore focus on how it is computed and used.
In the following sections we will describe the implementation of the V-buffer, the V-buffer computation steps, and finally describe how the V-buffer used when rasterizing the scene. When computing the V-buffer, access to the world positions of the fragments representing the shadow receivers is required. Since this information is not available in fragment programs on current hardware, it must be provided through a buffer accessible in the fragment programs. We start by describing how this position buffer is implemented.

4.2.2 Position Buffer

The position buffer is a two-dimensional lookup texture required when rasterizing the penumbra

[1] Full source code available from “http://www.ce.chalmers.se/staff/tomasm/soft/”.

wedges. Since there are no information available on which pixels that are in shadow, the texture must be updated for all visible pixels. This texture is implemented as a floating point pixel-buffer, where the *rgb* components of each pixel hold the world space coordinates (*xyz*) for the corresponding pixel in the framebuffer.
Computing this position buffer is the first step of each frame. This is accomplished by rendering the entire scene into the buffer using a combination of one vertex and one fragment program. The fragment program receives the interpolated world position of each fragment through the texture coordinates sent to it from the vertex program. This information is then written to the buffer as color values. An example of the contents of the position buffer is shown in Figure 4.

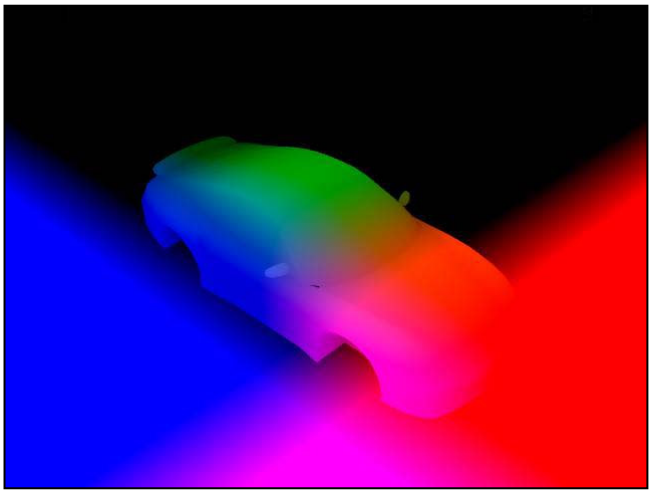

Figure 4: Position buffer, *rgb* values represent the *xyz* world coordinates of each fragment.

4.2.3 Visibility Buffer

The visibility buffer (V-buffer) holds a mask with coverage values, where a value of 0.0 represent full shadow and a value above or equal to 1.0 represent no shadow. The V-buffer is implemented as two 8-bit per component *rgba*-buffers, in the form of a double-buffered pixelbuffer. One of the buffers holds the additive luminance contribution, and the other holds the subtractive contribution.
Since blending with more than eight bits is not supported by today's graphics hardware, splitting values across several eight bit components is necessary to increase precision in order to avoid banding effects. Note that using split values will not be necessary when hardware has evolved. We reserve the highest five bits of each color channel for overflow, and use the three lowest bits for the coverage value, see Figure 5. These five overflow bits allow addition of split values (See Figure 6) and overlapping shadow geometry. A one-dimensional texture is used to split the coverage values across four 8-bit components. Indexing the texture with a value between zero and one gives the corresponding split value. The split values are recombined with a dot product that scales each channel and adds them together. See Figure 7.

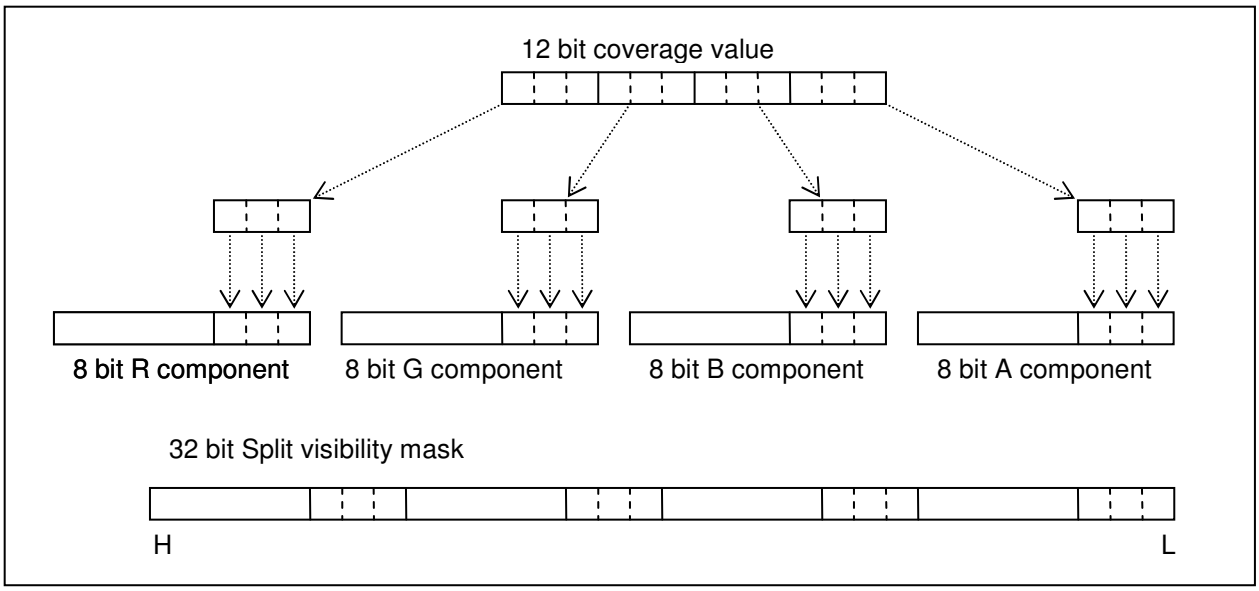

Figure 5: A 12 bit coverage value is split across four channels. The highest 5 bits of each component is reserved for overflow, and the lower 3 bits are used for the coverage value.

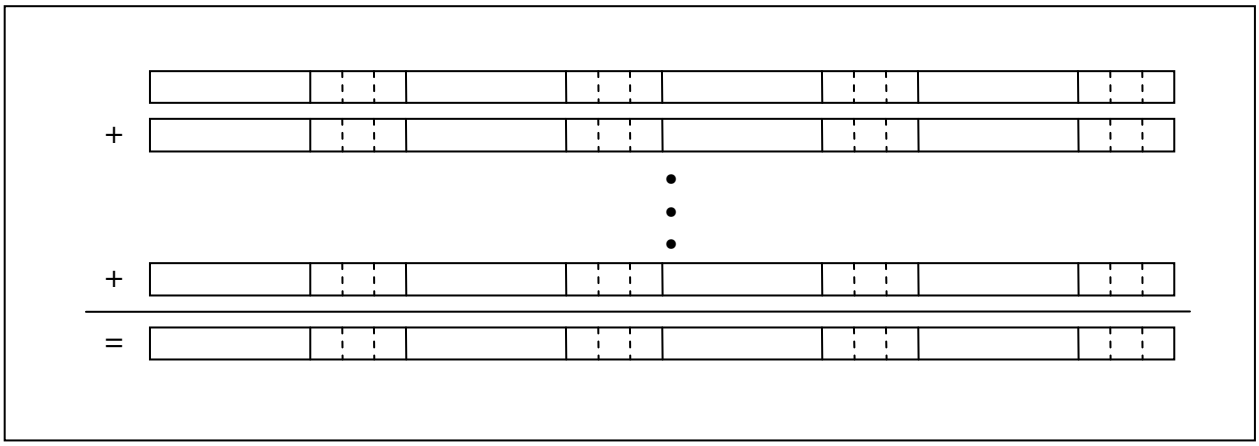

Figure 6: The split visibility masks are accumulated into a single split visibility mask.

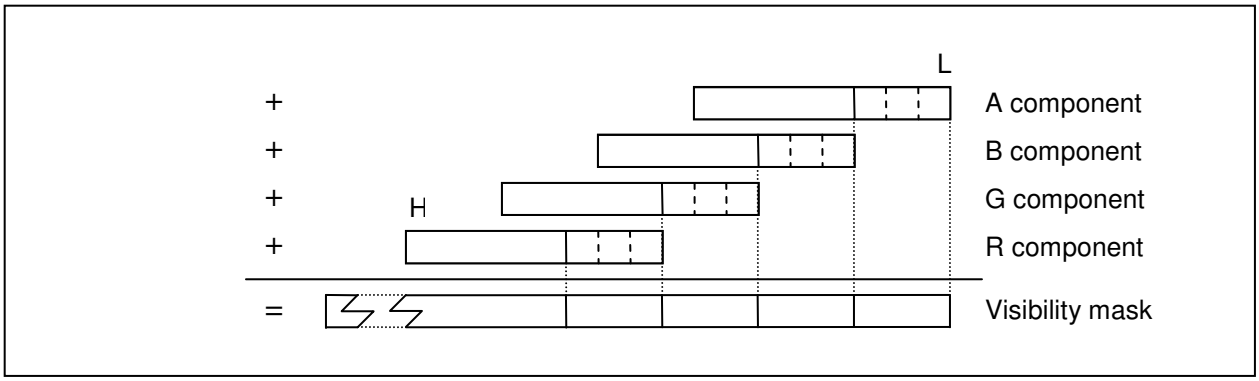

Figure 7: The split value is recombined by shifting the four components and adding them to yield a visibility mask. This operation can be performed with a single dot product.

Every frame before rendering the shadows of the scene, the V-buffer is cleared to the value one, representing no shadow. This is achieved by clearing the subtractive layer of the V-buffer to zero,

and the additive layer to one. Figure 8 is a visualization of how the V-buffer is derived from the additive and subtractive contributions.

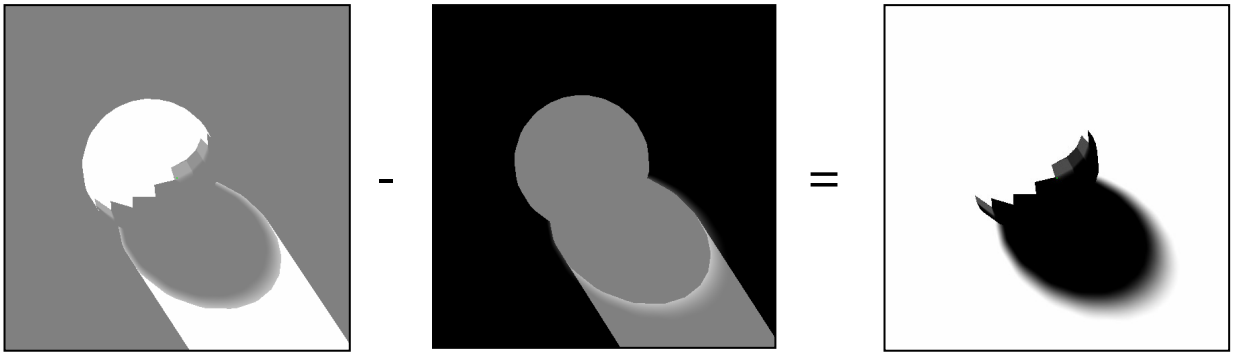

Figure 8: Subtracting the subtractive contribution from the additive yields the final visibility values. (Note that the intensities in the additve and subtractive contributions have been divided by two for visualization purposes)

4.2.4 Silhouette Edges

We have described how shadow volume algorithms use the silhouette loops of a 2-manifold shadow caster to create the shadow volume geometry. An edge is part of the silhouette if one of the triangles connected to it faces the light, while the other does not. Once the silhouette has been calculated, it is used to create the geometry of the penumbra wedges and the shadow volume.

4.2.5 Constructing Penumbra Wedges

The penumbra region for a given edge and a light source is found by sweeping a cone from one vertex of the edge to the other. The cone is generated by reflecting the light source through the sweeping point on the edge. But instead of calculating the exact penumbra volume in real time, a bounding volume for the penumbra region is created.

A wedge is created as follows: let a silhouette edge e be defined by the two vertices e_0 and e_1. First it is determined which of the two vertices is furthest from the light source center, l_c. This vertex is moved towards the light center until the two vertices are at the same distance from l_c. Assume that e_0 was at the shortest distance from the light source. We then denote the translated vertex e_1'. The two vertices e_0 and e_1' defines a new edge above the original edge, which will be the top of the wedge. This edge guarantees that the wedge contains the entire penumbra volume of the original edge, but the original edge is still used for visibility computation. Second, an orthogonal base is formed with the edge as x-axis and the vector from the edge to the light source as z-axis.

An axis-aligned bounding rectangle is created for the light source in this xy-plane. Each edge in this bounding rectangle together with one or both of the edge vertices define the front, back and side planes of the wedge. An additional middle plane separating the wedge in two halves is defined by the edge and the center of the light source, l_c.

The inner half is used for rasterizing the additive luminance contribution, and the outer half is used for the subtractive contribution. The middle plane coincides with the quadrilaterals generated with the standard shadow volume algorithm. Figure 9 shows the triangles that constitute a penumbra wedge. All triangles are necessary to avoid cracks between the hard

shadow quadrilateral and the wedge geometry.
In order to extend the soft shadow volume algorithm to handle the case when the eye is in shadow, we have implemented a Z-fail version. The solution is to add bottom caps to the penumbra wedges and modify the way wedges are rasterized in the same manner as for standard stencil shadow volumes (See section 3.1.1). The cap, visualized in Figure 9 c, consists of eight additional triangles.

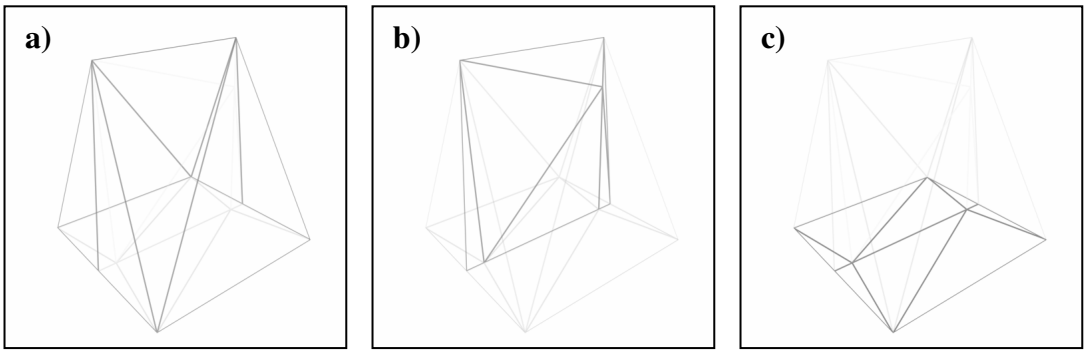

Figure 9: a) The eight triangles constituting the standard penumbra wedge. b) The six triangles representing the middle plane that separates the wedge in its front and back halves. c) The eight Z-fail bottom cap triangles.

4.2.6 Rasterizing Umbra Approximation

The V-buffer is initially cleared to 1, which indicates no shadow. Using the standard shadow volume algorithm, hard shadows are then rendered to the V-buffer. Front facing shadow volume quadrilaterals are additively blended to the subtractive luminance texture, thus subtracting 1.0 from the V-buffer for every pixel covered by the quad. Back facing quads are rendered to the additive luminance texture, thus adding 1.0 for every rasterized pixel. The result of this step is an over-approximation of the umbra region.

4.2.7 Rasterizing Penumbra Wedges

Rendering the penumbra wedges compensates for the over-estimated umbra, and computes the visibility factor for all points inside the wedges. The visibility factor for a point in the penumbra is defined as the area of the light source visible from the point divided by the total light source area. Note that the inner half of a wedge is rendered to the additive layer of the V-buffer while the outer half is rendered to the subtractive layer.
To reduce the amount of expensive per-fragment operations we first determine the visible pixels inside the penumbra wedge and mark these in the stencil buffer using the same technique [2] as for stencil shadow volumes. This requires that the depth information is present. The procedure is described in Figure 10 for Z-pass and Figure 11 for Z-fail.

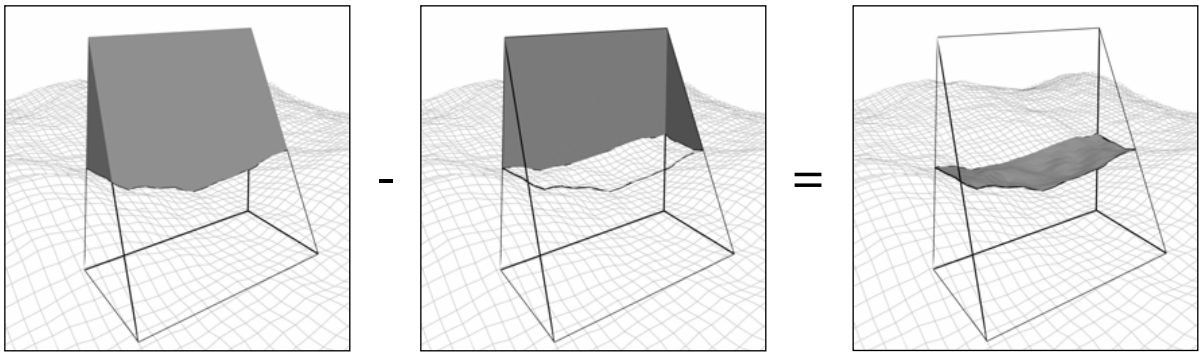

Figure 10: Z-pass stencilling. Front faces of the wedge half that pass the depth test, are rendered to the stencil buffer. Back faces that pass the depth test are then subtracted from the stencil buffer, which yields the penumbra region for the wedge half.

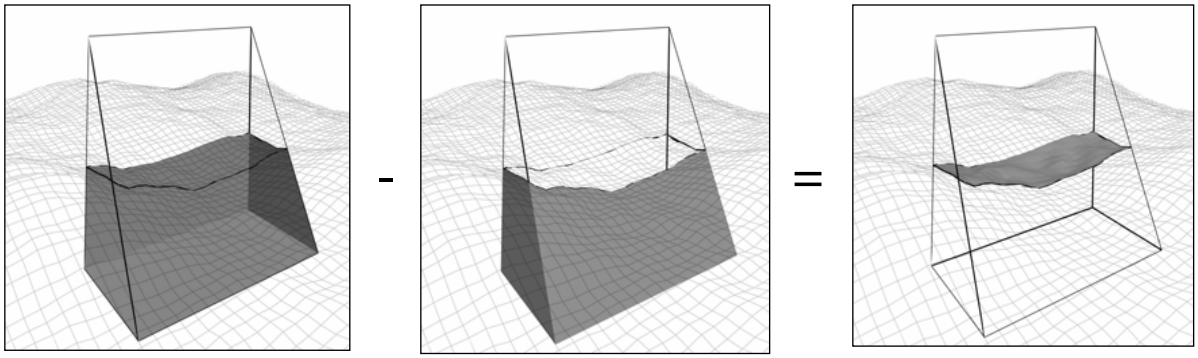

Figure 11: Z-fail stencilling. Back faces of the wedge half that fail the depth test are rendered to the stencil buffer. Front faces that fail the depth test are then subtracted from the stencil buffer. This results in the same mask as for Z-pass stenciling.

Our implementation handles three different types of light sources; spherical, rectangular, and textured rectangular light sources. The penumbra wedges originating from these three types have to be rasterized using three different fragment programs. These fragment programs, along with descriptions of the visibility computations are described below.

Fragment Program: Spherical Light Source

The fragment program first transforms the silhouette edge into light space. In light space, the light source is represented by a circle (orthogonal projection of the sphere along the z-axis) with its center located at z=1 on the z-axis, the radius is one, and the point to be shaded is located at the origin. An orthogonal basis for the transform is constructed with a z-axis directed from the point to be shaded to the light source. This basis combined with a scaling matrix transforms the edge points to light space. The scaling matrix multiplies the z coordinate by the inverse distance between the light source and the point to be shaded, and multiples the x and y coordinates by the inverse light radius.

A cone which encompass the light source and its top located at the point to be shaded now has the equation $x^2 + y^2 = z^2$. The line described by the transformed points is clipped against this cone in homogenous space by solving the equation of the intersection of the line and cone. Points below the xy-plane are rejected. A division by z projects the clipped points onto the z=1 plane.

The resulting points are used to do a texture lookup in an *atan2* cubemap to obtain the two angles

used when calculating the coverage. These angles characterize the minor arc defined by the intersections of the rays from the light center to the points and the unit circle. The angles are used in another texture lookup to obtain the area defined by the circle center and the minor arc, Figure 12 a. This area is subtracted by the area resulting from the cross product of the two points (Figure 12 b) and divided by the area of the unit circle to yield the final coverage (Figure 12 c).
In a final step, the coverage value is converted to a split value, as described in section 4.2.3, and added to either the additive or subtractive layer of the V-buffer. The currently active rendering target determines which layer is written to.

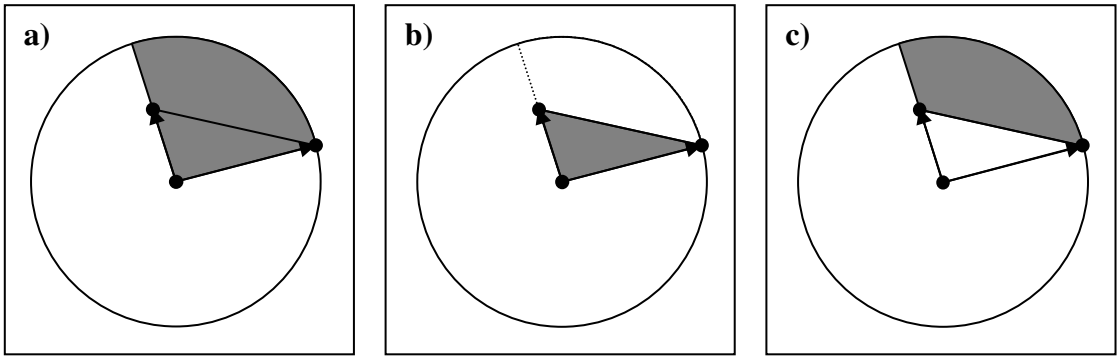

Figure 12: Coverage calculation for spherical light sources. a) Area defined by the light source center and the minor arc. b) Area of triangle defined by the light source center and the two clipped and projected edge points. c) The resulting coverage.

Fragment Program: Rectangular Light Source

The fragment program for the rectangular light source first transforms the point to be shaded into light space coordinates. As we have already described, the center of the light source is located at z=1 on the z-axis and the point to be shaded is located at the origin. A shear and scale matrix is constructed from the point to be shaded and the light extents (Figure 13). The silhouette edge is transformed into light space by this matrix and clipped against planes in homogeneous space. These planes coincide with the edges of the light source and the point to be shaded.

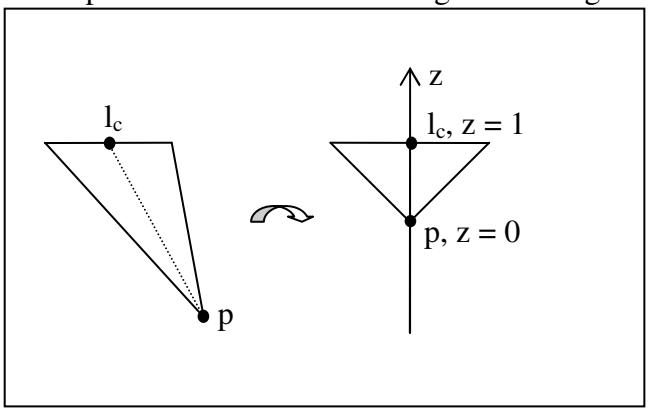

Figure 13: Transformation to light space.

As with the spherical light source the clipped points are projected onto the z=1 plane by a division by z and used to do a texture lookup that yields the two angles described in the previous section. These angles are used with a pre-calculated texture to lookup the area defined by the intersection between the rays from the light center to the points and the unit square (Figure 14 a).

Subtraction of the area of the triangle defined by the light source center and the clipped and projected edge points (Figure 14 b) yields the final coverage value (Figure 14 c).
The fragment program ends by converting the computed coverage value to a split value, and update the V-buffer in the same manner as for the fragment program for spherical light sources.

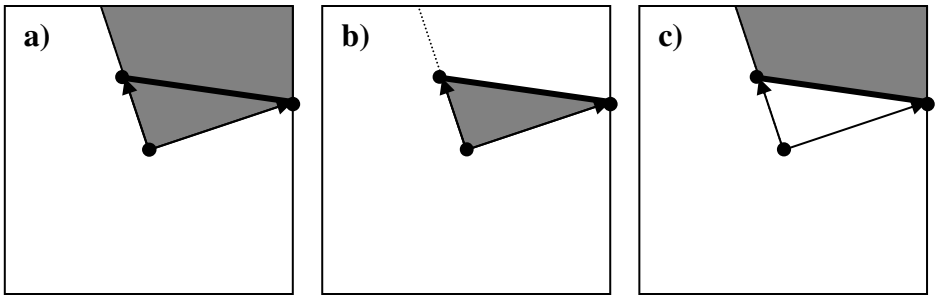

Figure 14: Coverage calculation for rectangular light sources. a) Area defined by the light source center and the minor arc clipped against a square. b) Area of triangle defined by the light source center and the two clipped and projected edge points. c) The resulting coverage.

Fragment Program: Textured Rectangular Light Source

This fragment program is very similar to the fragment program for non-textured rectangular light sources. The difference is that instead of computing the angles, we use the projected edges to do a lookup in a four-dimensional pre-calculated coverage texture. Since there is no support for four-dimensional textures, the texture is implemented as a two-dimensional texture, see [1] for details.

4.2.8 Putting it all together

The result of the steps above is a visibility buffer (V-buffer) containing the information of shadow intensities for the scene. To enhance the visual quality of the final image, we combine the V-buffer with per-pixel lighting derived from the previously created position buffer. Adding the diffuse color contributions from the textures of the scene yields the final image. Figure 15 shows how the V-buffer is combined with per-pixel lighting. This results in a complete lighting representation including visibility function. Figure 16 shows how the diffuse contributions from the textures of the scene are added to yield the final image.

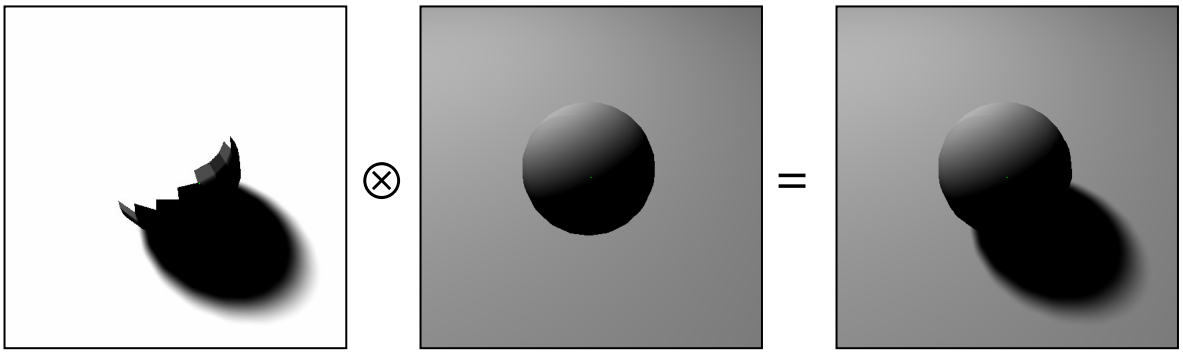

Figure 15: Adding per-pixel lighting.

Figure 16: Adding diffuse texture contributions.

4.3 Capping of Triangle Meshes

Since shadow volume algorithms require shadow caster objects to be two-manifold, unless Bergeron's trick is used, holes in non-closed meshes must be capped. We present an automatic capping algorithm that is part of the framework that handles triangular meshes, which enables correct shadows from arbitrary polygonal objects. Even though Bergeron's algorithm might be possible to adapt for soft shadow volume usage, the double incs and decs of the stencil buffer are likely more expensive than capping the geometry.

A triangle mesh consists of at least one triangle. The front- and back faces of a triangle are determined by the winding order of its edges. Without loss of generality we will throughout the rest of this section assume we have a counter-clockwise (CCW) winding for all triangles. See Figure 17.

To connect two triangles, their winding order must be identical. When two edges are connected they have opposite direction, thus forming a bidirectional edge. The two connected triangles form a triangular mesh where any unidirectional edges form a CCW directed loop.

If more triangles and/or meshes are connected in the same manner, the result is always a mesh where any unidirectional edges form one or more CCW directed loops. As long as these loops exist they form holes in the surface and the mesh is not 2-manifold. In order to remove the holes we cap them with triangles.

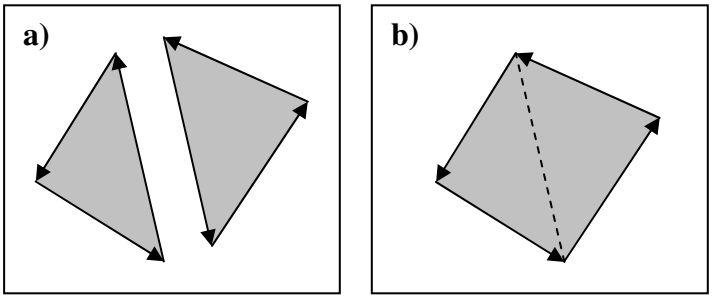

Figure 17: Connecting two triangles. a) Two triangles. b) Resulting mesh. Note the winding order of the edges.

We choose from a loop two edges that share a vertex and duplicate them. Their directions are reversed and a third directed edge is added to form a new triangle. This triangle has a CCW winding and is added to the mesh by connecting it to the two original edges (Figure 18 b). If the hole is triangular the third edge is also connected (Figure 18 c, d).

The method above has the effect of replacing two unconnected edges with at most one and thereby decreasing the total amount of edges in the loop. By repeatedly applying this method the holes are guaranteed to be capped with a time complexity of O(n) where n is the number of edges, since we remove at least one edge for each repetition. Note that this requires a structure that stores information on adjacent edges.
In our implementation we have chosen only to use the capping triangles in the silhouette calculation since automatic texture coordinate generation does not give the desired result in most cases.

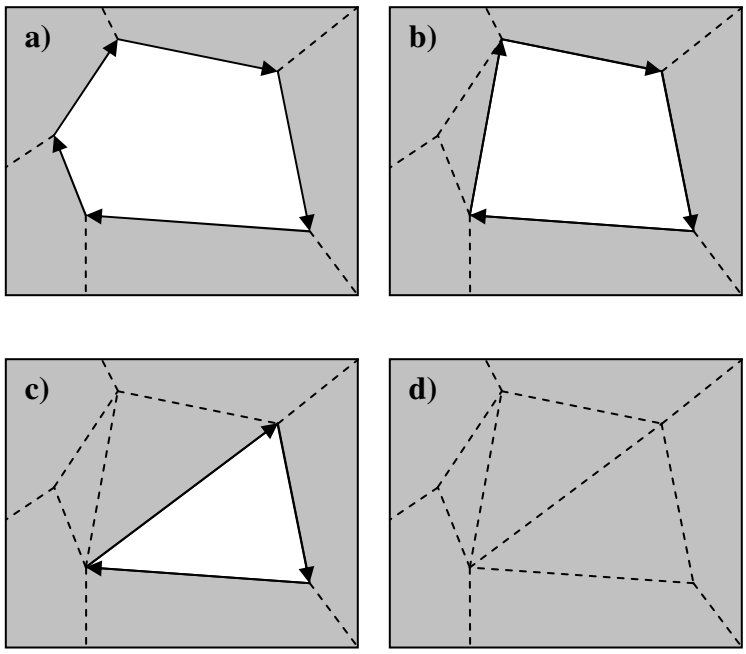

Figure 18: Capping of a hole in an arbitrary mesh. a) The hole is a loop of directed edges. b) A capping triangle has been added. c, d) The last three edges are connected with a triangle and the hole is closed.

4.4 A Simple Load Balancing Scheme

Due to the complexity of the fragment programs, the frame rate of our implementation depends on how many pixels containing penumbra that are rasterized in the V-buffer. In order to limit the number of rasterized pixels, we have experimented with resizing the V-buffer. The results are presented in Figure 20, Figure 21 and Figure 22. This technique could be extended to adaptively resize the V-buffer in order to achieve a constant frame rate.
One way of implementing a V-buffer with variable size is to use several layers with different resolution and switching between them. Another way could be to use one full resolution buffer, but drawing only to parts of it. The first method is easily implemented, but since several buffers are needed memory usage is increased. The second method is harder to implement, but since we need only one buffer less memory is needed and any V-buffer size can be chosen as long as it is smaller than the original buffer.
Bilinear interpolation of the sized V-buffer could reduce artifacts, but since split values cannot be interpolated directly, we have chosen to use point sampling to avoid a second rendering pass.

As seen in the result images, “shadow bleeding” artifacts appear around the edge of objects not in shadow in front of shadowed regions (Figure 19). This is because texels in the V-buffer generally covers more than one pixel in the frame buffer, causing shadow/non-shadow to appear where it should not. Despite this, scalability is sometimes an important property, why scaling of the V-buffer can prove to be useful. See Appendix A.

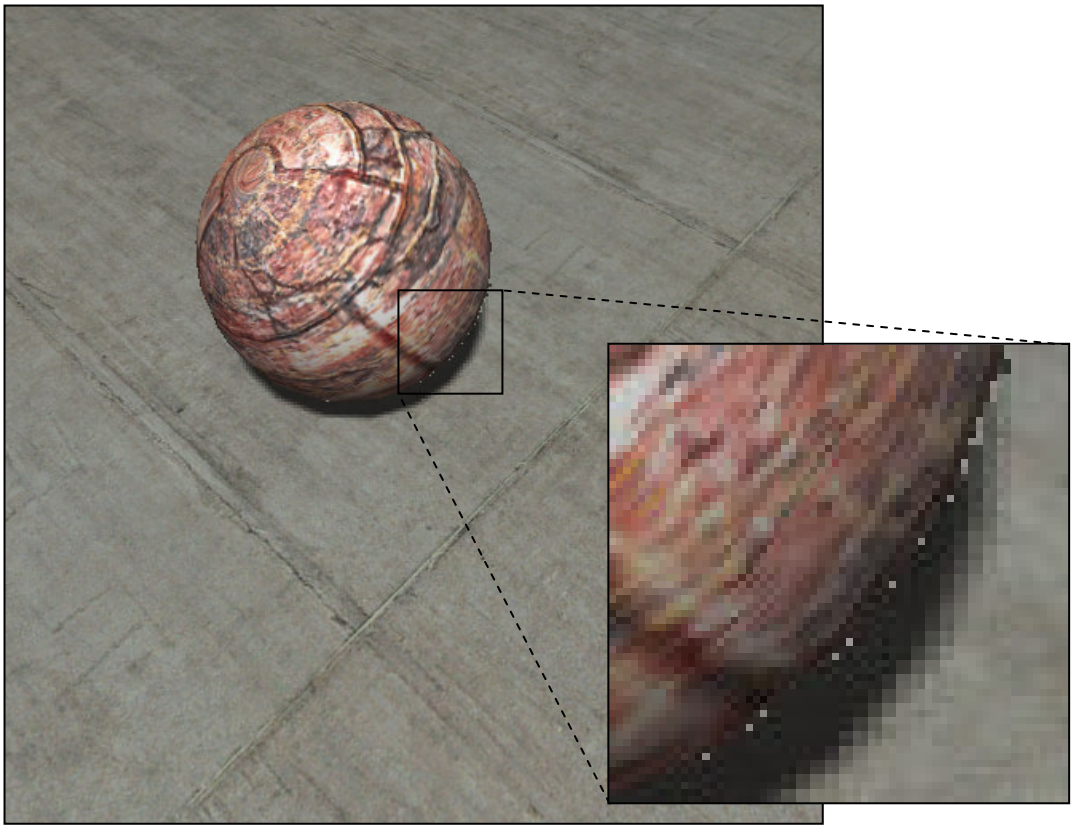

Figure 19: Shadow bleeding.

5 Discussion and Results

In this thesis we have investigated the theoretical and practical aspects of both hard and soft real-time shadows, and an implementation of the soft shadow volume algorithm [1, 2, 22, 23] using OpenGL has been presented. This algorithm extends the standard shadow volume algorithm, and implements a general solution for real-time soft shadows with arbitrary shadow casters and receivers. We have suggested a simple technique for load balancing in section 4.4.
Furthermore, we have shown that the algorithm fits well for real-time applications such as games and other virtual reality simulations by implementing the algorithm in real game settings as shown in Appendix B. The images render at real-time frame rates. By implementing a Z-fail version of the algorithm, we have improved the robustness of our implementation.
The algorithm has some limitations that cause artifacts. One such limitation is that a silhouette is calculated with respect to the center of the light source, which works fine as long as the silhouette is identical for all points inside the light source. Artifacts in the form of “popping shadows” appear when the light source center moves between two positions which generate different silhouettes. Another limitation comes from the fact that edges are treated independently, and too much shadow can be applied in the penumbra region when different edges cover the same portion of the light source.
We hope that our work will inspire developers to implement and further improve the soft shadow volume algorithm to enhance realism in games and other real-time applications. Soft shadows in real-time are here to stay, we can never go back.

6 Future Work

In the near future, several new features are expected to be implemented in graphics hardware. These features include, but are not limited to, *superbuffers* [24] and more than eight bits blending. Superbuffers allow textures and vertex buffers to be handled in a uniform way, enabling for instance rendering directly to vertex buffers, which may be useful for generating wedges. Our current implementation is limited to a 12-bit V-buffer, enabled by splitting values over four 8-bit values in a 32-bit texture. This splitting will be unnecessary and therefore increase performance when more than eight bits blending will be allowed. The higher bit-depth will also make more overlap of shadow volumes possible, preventing overflow. Implementation of a textured colored rectangular light source will also be possible with increased bit depth in blending operations since several color channels can be used.

Currently, the depth information of the scene has to be rendered once for each pixelbuffer. If the depth-buffers can be shared between these pixelbuffers, performance will be improved. To further improve performance, the techniques described in the GDC presentation “Optimized Stencil Shadow Volumes” [14] should be utilized.

An interesting subject to further investigate is the use of pre-rendered lightmaps in conjunction with the soft shadow volumes for static environments to improve performance and/or visual appearance. This would be particularly useful for indoor scenes with complex static geometry, as is common in many first person shooters.

7 List of References

[1] Assarsson, Ulf, and Akenine-Möller, Tomas, ”A Geometry-based Soft Shadow Volume Algorithm using Graphics Hardware”, ACM Transactions of Graphics (Proceedings of ACM SIGGRAPH), vol.22, no. 3, pp. 511-520, July 2003.

[2] Assarsson U., and Dougherty M., and Mounier M., and Akenine-Möller T., “An Optimized Soft Shadow Volume Algorithm with Real-Time Performance”, Graphics Hardware 2003, pp. 33-40, p. 131, July 2003.

[3] Crow, Frank, “Shadow Algorithms for Computer Graphics”, *Computer Graphics (Proceedings of ACM SIGGRAPH 77)*, July 1977.

[4] Whitted, Turner, “An improved illumination model for shaded display”, Communications of the ACM, v.23 n.6, p.343-349, June 1980.

[5] Wann Jensen, Henrik, and Christensen, Niels, “Photon Maps in Bidirectional Monte Carlo Ray Tracing of Complex Objects”, Computer & Graphics vol. 19(2), p.215-224, March 1995.

[6] Goral, Cindy M. et al, “Modelling the Interaction of Light Between Diffuse Surfaces”, Computer Graphics (Proceedings of SIGGRAPH 84), July 1984.

[7] Blinn, James F., “Me and my (fake) shadow*”, IEEE Computer Graphics and Applications*, January 1988.

[8] Williams, L. “Casting Curved Shadows on Curved Surfaces”, Computer Graphics, Proceedings of ACM Siggraph), August 1978.

[9] Segal, M., and Korobkin, C., and van Widenfelt, and R., Foran, J., and Haeberli, P., “Fast shadows and lighting effects using texture mapping”, Procedings of the 19th annual conference on Computer graphics and interactive techniques, pp. 249-252, ACM Press 1992.

[10] Fernando R., and Fernandez S., and Bala, K., and Greenberg D.P., ”Adaptive shadow maps”, proceedings of SIGGRAPH 2001, p. 387-390, 2001.

[11] Low Kik-Lim, and Ilie Adrian, “Computing a View Frustum to Maximize an Object’s Image Area”, Journal of Graphics Tools, vol 8, No. 1, p. 3-15.

[12] Stamminger Marc, and Drettakis George, “Perspective shadow maps”, ACM Transactions on Graphics (TOG), volume 21, issue 3 July 2002, special issue: proceedings of ACM SIGGRAPH 2002, p. 557-562, 2002.

[13] Heidmann, Tim, “Real shadows, real time”, Iris Universe, No. 18, p.23-31, Silicon Graphics Inc., November 1991.

[14] Everitt, Cass, and Kilgard, Mark J., “Optimized Stencil Shadow Volumes”, GDC 2003 Presentation http://developer.nvidia.com/docs/IO/8230/GDC2003_ShadowVolumes.pdf, 2003.

[15] Bergeron, P., “A General Version of Crow’s Shadow Volumes”, IEEE Computer Graphics and Applications, 6(9): 17-28, September 1986.

[16] Carmack, John, Unpublished material, 2000.

[17] Bilodeau Bill, and Songy Mike, “Real Time Shadows”, Creativity 1999, Creative Labs Inc. sponsored game developer conferences, Los Angeles, California, and Surrey, England, May 1999.

[18] Chan, Eric, and Durand, Frédo, “Rendering Fake Soft Shadows with Smoothies”, Proceedings of the Eurographics Symposium on Rendering 2003.

[19] Wyman, Chris, and Hansen, Charles, “Penumbra Maps: Approximate Soft Shadows in Real-Time.”, Proceedings of the 2003 Eurographics Symposium on Rendering.

[20] Assarsson, Ulf, “A Real-Time Soft Shadow Volume Algorithm”, PhD thesis, Department of Computer Engineering, Chalmers University of Technology, October 2003.

[21] Hasenfratz, J.-M, and Lapierre M., and Holzschuch N., and Sillion F.X., “A Survey of Real-Time Soft Shadows Algorithms”, Eurographics State-of-the-Art Report, Eurographics, 2003.
[22] Akenine-Möller, Tomas, and Assarsson, Ulf, ”Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges.”, 13th Eurographics Workshop on Rendering 2002, pp. 309-318, June 2002.
[23] Assarsson, Ulf, and Akenine-Möller, Tomas, ”Occlusion Culling and Z-fail for Soft Shadow Volume Algorithms”, The Visual Computer 2002.
[24] Mace, Rob, “OpenGL ARB Superbuffers”, GDC 2003 Presentation http://developer.nvidia.com/docs/IO/8230/GDC2003_OGL_ARBSuperbuffers.pdf, 2003.

Appendix A: Resizing of the V-buffer

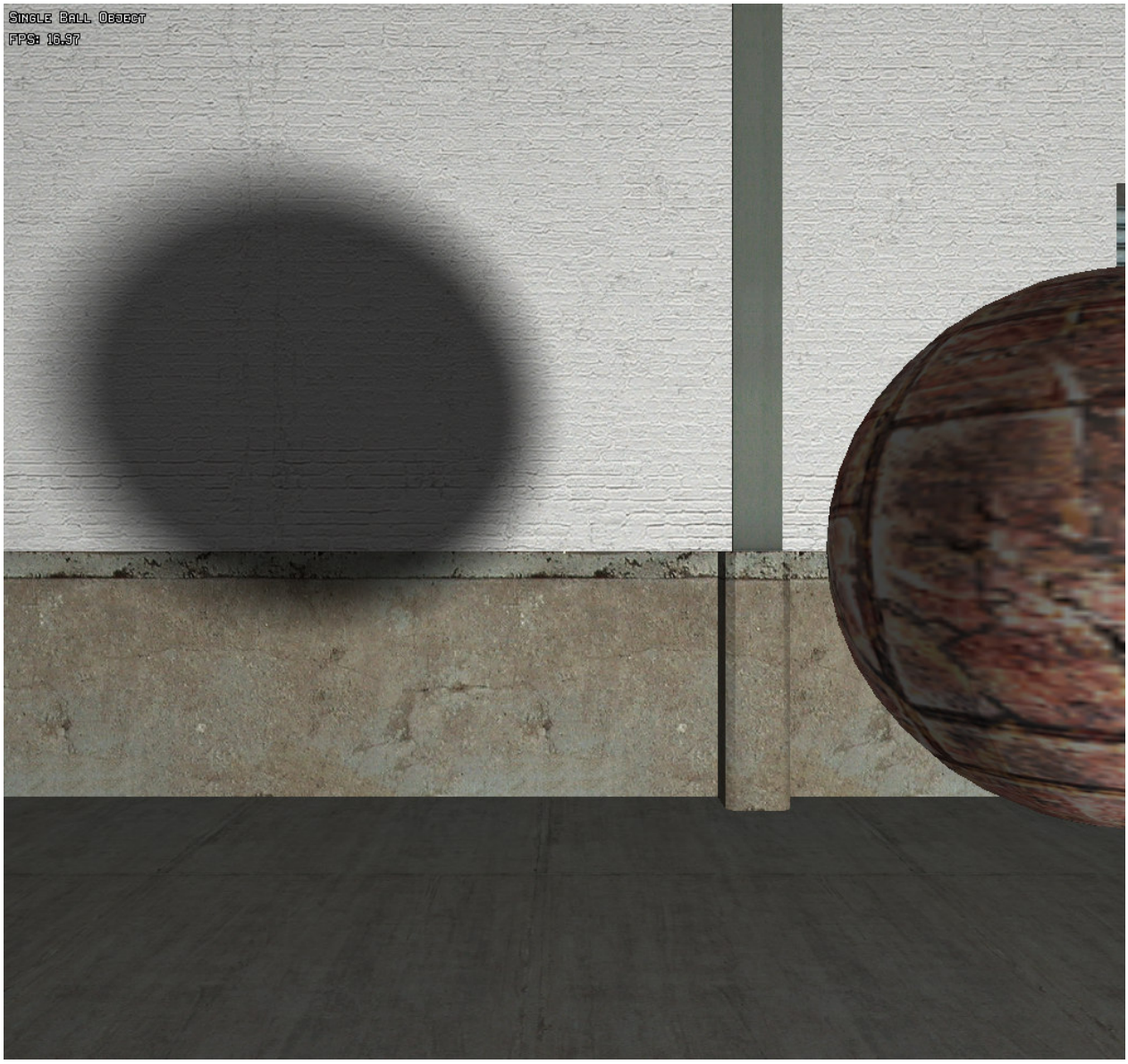

Figure 20: Full resolution V-buffer, 1024x1024 pixels. Frame rate ~16 Hz.

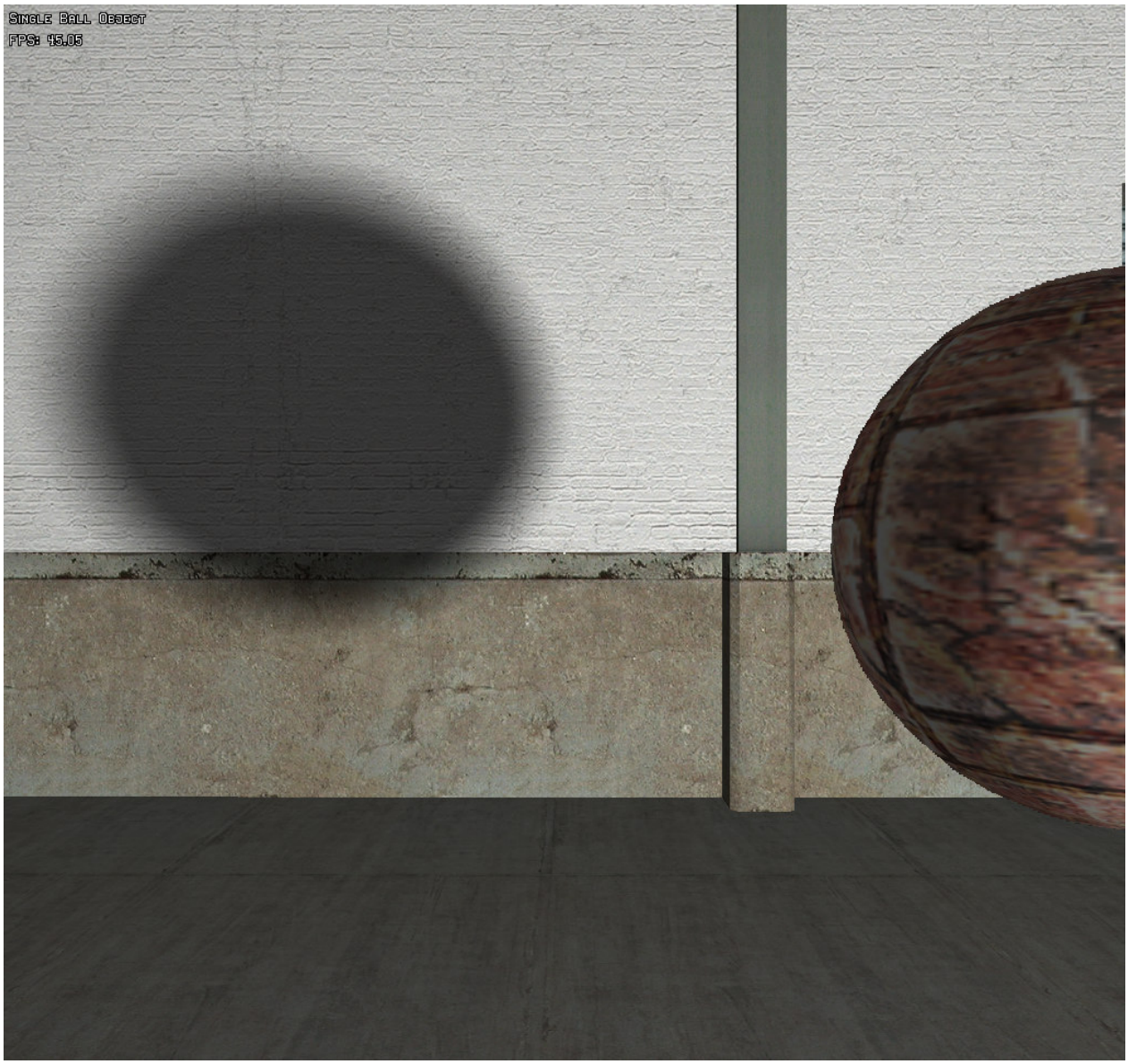

Figure 21: Resized V-buffer, 512x512 pixels. Frame rate ~45 Hz.

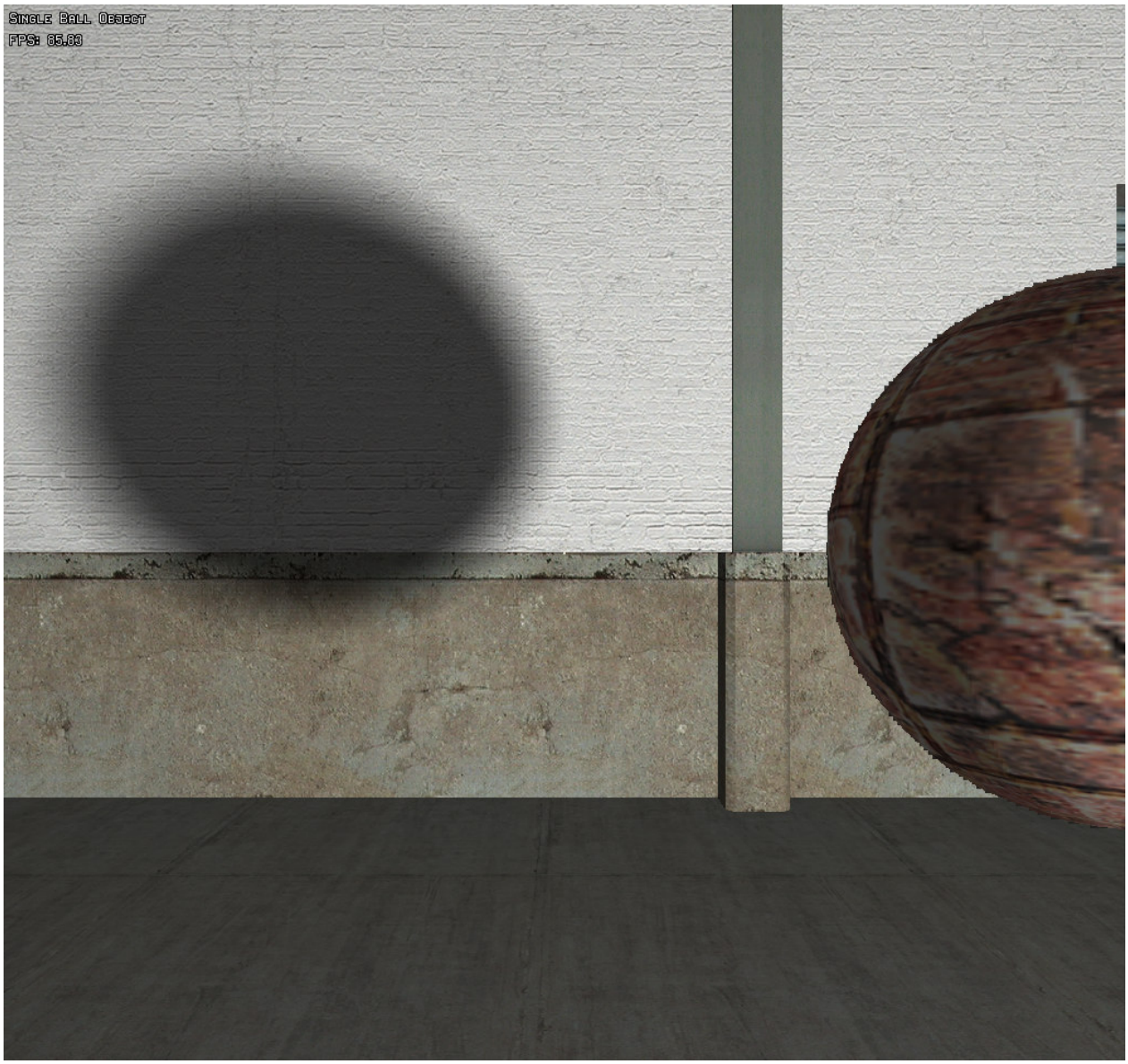

Figure 22: Resized V-buffer, 256x256 pixels. Frame rate ~85 Hz.

Appendix B: Screenshots

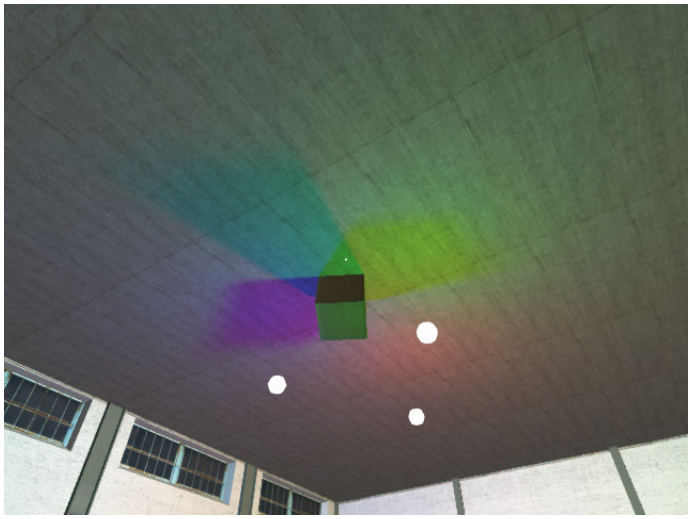

Figure 23: Multiple colored dynamic light sources

Figure 24: Multiple shadow casters

Figure 25: Rectangular light source

Figure 26: Arbitrary shadow receivers

Figure 27: Arbitrary shadow receivers

Figure 28: Multiple shadow casters

Figure 29: Dynamic colored light source