Real-time Soft Shadows in a Game Engine

Kasper Fauerby (kasper@peroxide.dk)
Carsten Kjær (carsten@peroxide.dk)

14th December 2003

Abstract

In this thesis we explore the possibilities of using various real-time shadow techniques in a 3d game engine. We describe a technique known as the stencil shadow algorithm and show how it can be extended to produce soft shadows from volume light sources using penumbra wedges. The penumbra wedge technique allows for real-time soft shadows in relatively simple scenes.

We present a novel coverage calculation technique for spherical light sources, which significantly reduces the amount of pixel shader instructions and the amount of texture memory required for look-up tables.

We identify a performance bottleneck in the algorithm which prevents the achievement of real-time performance in complex scenes, and we present a new version of the algorithm that eliminates this bottleneck for a limited class of shadow casting objects.

We have implemented both versions of the soft shadow algorithm in our game engine, and we compare their respective performance on different hardware. Some implementation details are given, including the CG source code for the vertex and pixel shaders we have used.

We discuss how to effectively manage a large number of shadow volumes in a dynamic game scene where both lights and shadow casters move around freely. Finally, we give an overview of some of the limitations in graphical hardware anno 2003 that introduce unnessesary work loads on the algorithm, thus degrading performance.

Contents

Chapter 1

Introduction

A realistic light setting with proper shadows is very important in 3d graphics. Without shadows, images tends to look flat and it is difficult (or even impossible), to determine the size and spatial relation of the objects in the scene. In the upper-left corner of figure 1.1 a simple scene is rendered without any shadows. Without changing the camera angle it is hard to determine where exactly the bench is located in the scene, but at first glance it would seem that it is standing on the ground, a bit behind the lamppost. Indeed that is one possible interpretation of the image, as can be seen in the bottom-left rendering where shadows have been enabled. Another interpretation of the image could be that a slightly smaller version of the bench is floating in the air a short distance in *front* of the lamppost, as shown in the upper-right rendering in the figure. Without shadows it is impossible to tell which of the two interpretations is correct, but as soon as the shadows are included there really is no doubt.

In a computer game shadows are important as well, not just because they increase the level of realism and overall quality of the graphics, but also because they can affect the game-play significantly. F.ex. if the player is required to jump onto a platform or dodge a moving object, shadows provide very important visual clues required for the player to determine *when* to press the jump or dodge button. Without shadows such tasks can quickly become frustrating and annoy the player to the point where he stops playing the game. As a result, an enormous amount of research has been done on the topic of real-time shadow algorithms, fast enough for use in an actual computer game set in a complex 3d environment.

Currently most real-time shadow algorithms have been limited to *hard shadows*, like those in figure 1.1. Hard shadows are the result of light sources being modeled as a single point with no area and can be recognized by a very sharp transition from light into shadow. If light sources are modeled with an actual shape with an area or volume, (as f.ex. a sphere), *soft shadows* can be produced. Soft shadows can be recognized by their inclusion of a penumbra region: an area that is

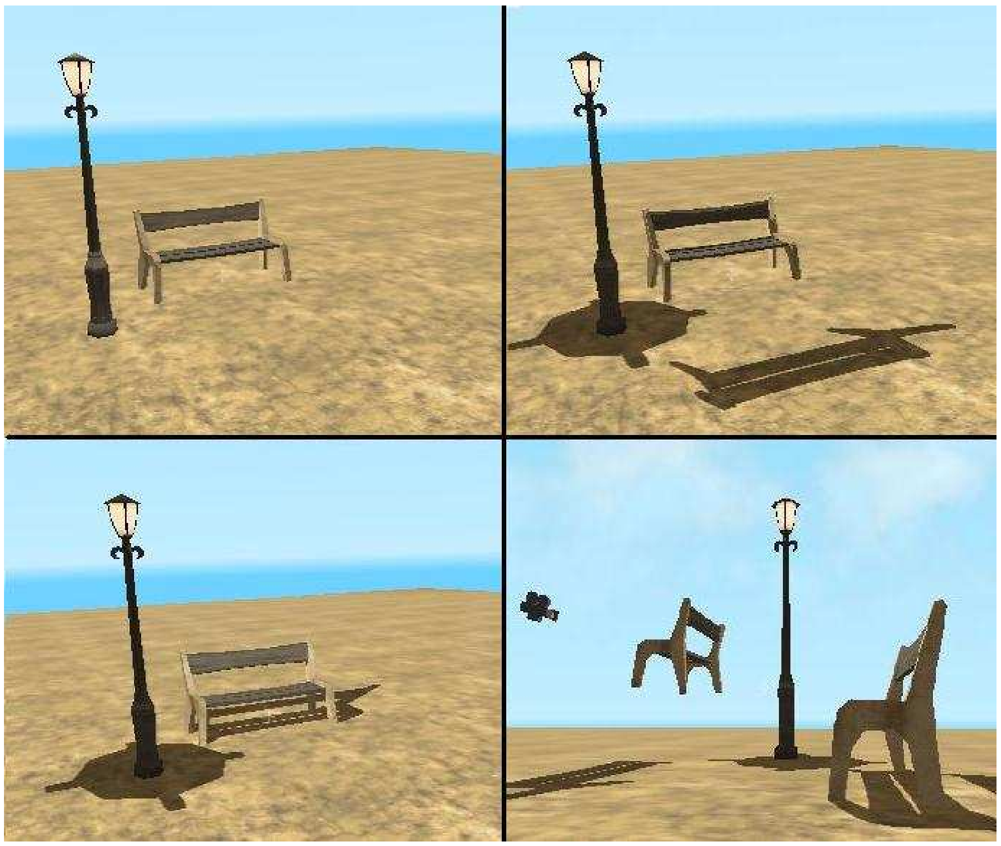

Figure 1.1: The importance of shadows in a 3d image

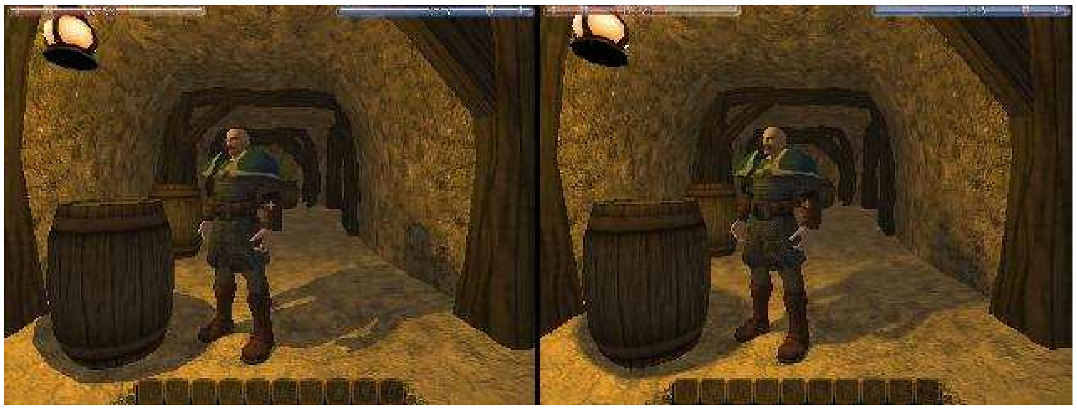

Figure 1.2: Hard vs. soft shadows

neither fully lit nor fully in shadow. The visual quality of soft shadows compared to that of hard shadows is very high, as demonstrated in figure 1.2. It is thus highly desirable to be able to apply real-time soft shadows to computer games. Unfortunately, the computations required for soft shadows are much more complex than those required for hard shadows. To the best of our knowledge, no released game has utilized true, real-time, dynamic soft shadows[1].

Key results

In this thesis we explore the possibilities of applying true and fully dynamic soft shadows to game scenes. We have implemented, as well as developed several optimizations for, a recent soft shadow algorithm and applied it to our game engine. Our contributions include a novel technique for calculating *coverage values* for spherical light sources. With this technique we are able to significantly reduce the length of the *pixel shader*, used for rendering soft shadows, as well as the amount of texture memory required for the technique. We also discuss some unresolved problems that still remain with regard to the technique, and we identify a serious performance bottleneck in the algorithm, which will have to be addressed before the technique can be applied to actual game scenes. Finally we present and discuss an outline for a new algorithm, which overcomes this bottleneck for a limited class of shadow casting objects.

Definitions and assumptions

In writing this thesis we have assumed that the reader is familiar with common terms and concepts used in 3d computer graphics. This includes concepts such as the color buffer, z-buffer and stencil buffer. Furthermore, it is assumed that the reader understands how the graphics pipeline operates on 3d meshes and moves them through a chain of 3d spaces, (often referred to as the model-space, world-space, view-space and projected-space), before actually rasterizing them into the color buffer. An understanding of homogeneous coordinates and how they solve the problem of being able to implement translation as well as rotation and scaling through a 4x4 transformation matrix is also assumed. Finally, the reader is assumed to have a thorough understanding of vertex and pixel shaders. Refer to Appendix A for a brief introduction to all these concepts.

Whenever we refer to 'current graphics cards' or 'the latest graphics hardware', throughout the thesis the intended meaning is DirectX 9.0 based graphics cards such as nVidias nv30-based and ATIs R300-based cards. All these chipsets has support for vs2.0 and ps2.0 shaders which are the minimum requirements for

[1] 'Fake' soft shadows have been applied to certain games where a hard shadow is simply blurred somewhat along the edge.

our implementation of the soft shadow algorithm. Specifically we have used DirectX 9.0b with a Radeon 9700Pro graphics card.

Thesis organization

In chapter 2 we briefly discuss how light works in the real world and in computer graphics. We introduce something called the *rendering equation*: a compact formulation of how to calculate lighting that gives us a framework against which we can compare our real-time solutions. In chapter 3 we give a short overview of the different real-time shadow solutions and then we elaborate on *stencil shadows*, the technique our soft shadow implementation is based upon. In chapter 4 we introduce a technique for rendering soft shadows in real-time with the use of a rendering primitive called a *wedge*, and we present our novel extensions to the soft shadow algorithm, followed by a discussion of the unresolved problems with the technique. In chapter 5 we discusses how a large amount of shadow casting objects in a game scene is efficiently managed, so that only those shadow volumes that affects the visible image are processed and rendered. In chapter 6 we provide an overview of our game engine which has been the framework for our implementation of the soft shadow technique, and we present some details of the implementation which was left out in the earlier chapters. Finally, in chapter 7, we summarize our results, draw conclusions and give suggestions for future work that would improve the soft shadow technique.

Chapter 2

Lighting

One of the goals of real-time 3d applications such as a computer game is to simulate a world and to generate real-time images which, to a certain degree, tricks us into believing that we are actually 'inside' this virtual world. Many real-time applications have been created where the user feels immersed in the virtual world and consequently, in some sense, believes that the generated images are real. This is not a result of photo-realistic images, as these are generally impossible to produce in real-time today, but because the human brain is capable of filtering away the flaws and inconsistencies in computer generated images and recognize what the image is supposed to represent. Photo-realism is therefore not necessarily an absolute requirement and in some applications, cartoons for instance, not even desired. In other applications though, f.ex. movies, games with a realistic look and architectural visualization applications, it is desirable to generate images as close to reality as possible and for such applications it is important to study why the real world looks the way it does.

In this chapter we first give a theoretical overview of local and global lighting models, introducing something called the *rendering equation* as well as the concept of a *BRDF*. Then we discuss how this theory can be approximated and applied to real-time graphics. In doing so, we introduce a framework that we can later compare our various light and shadow methods with.

2.1 Light models

Light models are the mathematical formulas used when calculating the color of a point on the surface of an object. Many different models have been proposed and the most important distinction between them is whether they are *local* or *global* models.

2.1.1 Local models

Local light models compute the color of a point on a surface by considering the position of the point, the properties of the surface that it is a part of, and the properties of any light sources that shines on it. This means that no other objects in the scene, except light sources, are considered neither as blocking light nor as reflecting light. This is clearly a crude approximation, and it will f.ex. make no difference whether there is an opaque object between the point and a light-source or not. In a more realistic light model such an object would cause a shadow. In spite of this, local light models are often used in real-time applications because of the minimal amount of computations required, and because only local knowledge of the scene geometry is needed.

We start by considering a point x'. If x' receives light from another point x'', f.ex. a light source, then we are interested in how that incoming light is reflected. If the eye point is placed at x we want to know how much light emitted at x'' towards x' is received by x. Figure 2.1 shows this setup.

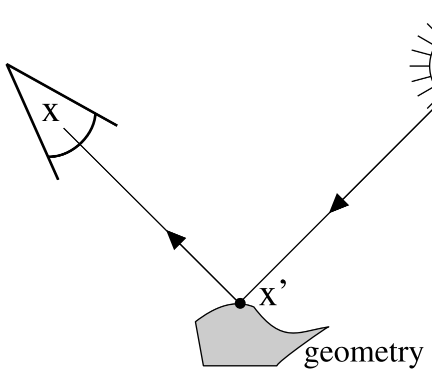

Figure 2.1: The three points involved in the local light model.

Since no more light can be reflected than is received, and since there is no such thing as negative light, the light received by x' and the light reflected towards x is assumed to be related by a factor in $[0..1]$. The reflection is dependent on the geometric relationship between the three points, an example being that the farther the three points are apart the less light x will receive. The reflection is also dependent on the orientation of the surfaces on which the points are located. F.ex., if the surface normal at x' is pointing towards x'', x' will receive more light than if it had a different orientation. Reflection is also highly dependent on the wavelength of the incoming light. All these factors can be combined into a single function: the *BRDF* or *Bidirectional Reflectance Distribution Function*[1]. We can

[1]BRDF is also called *spectral reflectivity coefficient*. It was introduced by Nicodemus et al.

describe the light passing from x' to x as a result of light passing from x'' to x' as:

$$L_{x''}(x, x') = L(x', x'')BRDF(x, x', x'') \tag{2.1}$$

Notice that the wavelength of the light is not mentioned explicitly. This omission is made on purpose because the relation is then independent of how we represent colors. For the usual RGB representation of colors all the elements are 3-vectors and the multiplication operation is per-component multiplication as described above.

Light is actually a stream of photons, and since a particle can only be reflected in one direction, the BRDF actually describes the *chance* of reflecting a photon in a certain direction or the *percentage* of all incoming photons that are reflected in that direction. This is not an important distinction as we do not model photons directly.

A different formulation of the BRDF is possible. Instead of using 3 points we can define a BRDF function which takes as input a single point, a direction for the incoming light and a reflection direction[2]. Under the direction formulation the local light model would look like this:

$$L_{\vec{\omega}_i}(x, \vec{\omega}_o) = L_i(x, \vec{\omega}_i)BRDF(x, \vec{\omega}_i, \vec{\omega}_o) \tag{2.2}$$

Where $\vec{\omega}_i$ is the direction of the incoming light, and $\vec{\omega}_o$ is the outgoing direction of the reflected light. It is sometimes more convenient to use this formulation of the BRDF but the basic idea is the same.

A visualization of a BRDF for a single incoming direction is seen in figure 2.2. The distance of the curve from x' represents the amount of reflection. The farther away the curve is, the more light is reflected in that direction.

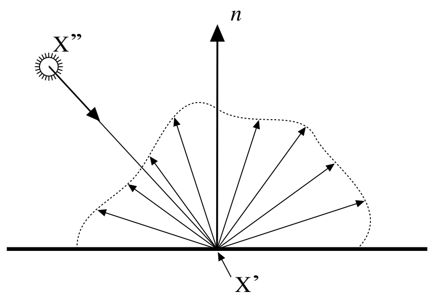

Figure 2.2: Visualization of a BRDF for a single incoming direction.

in 1977[NRH+77]. Note that in the manner we present it here it is actually the *unoccluded three point transport reflectance* as described in [Kaj86].

[2]This is the original formulation of the BRDF.

The BRDF is a very general description of reflection, and in real-time applications a general BRDF is often too expensive to evaluate so instead a simpler and cheaper model is often used. We will elaborate on this in section 2.2.

2.1.2 Global models

Global light models take into account the entire virtual world. This means that every object in the entire scene can potentially influence the color of a point. Examples of such influences are objects blocking direct light from a light source to the point or objects reflecting additional light onto the point. The result of using a global light model is often called *global illumination*.

The 'standard' model is *the rendering equation* introduced by Kajiya in 1986[Kaj86]. Kajiya says:

> *"[The model] subsumes a wide variety of rendering algorithms and provides a unified context for viewing them as more or less accurate approximations to the solution of a single equation."*

The rendering equation is:

$$L(x, x') = v(x, x')\left[L_e(x, x') + \int_S BRDF(x, x', x'')L(x', x'')dx''\right] \tag{2.3}$$

The function v encodes *visibility*. v is 1 if x and x' are mutually visible and 0 otherwise. L_e is the light passing from x' to x because of the emission of light at x'. Unless x' is a light source this factor will be zero. S is the union of all surfaces of all objects in the entire scene. The rendering equation therefore states that the light passing from x' to x is zero if they are not visible to each other; otherwise they are the sum of the emitted and the reflected light from x'. The emitted light is a property of the object, which x' is a part of, and the geometric relationship between x and x'. The reflected light from a *single* source point, or a single incoming direction, has already been described in equation 2.1, so the total amount of reflected light must be the sum (integral) of the contribution from *every* point in the scene.

The rendering equation cannot be directly evaluated since L occurs on both sides of the equation. However we can reformulate it as Kajiya[Kaj86] does. We start by writing it in a compact form:

$$L = vL_e + vTL \tag{2.4}$$

where

$$< Tf > (x, x') = \int_S BRDF(x, x', x'')f(x', x'')dx''$$

By evaluating equation 2.4 recursively we get:

$$\begin{aligned} L &= vL_e + vTL \\ &= vL_e + vT(vL_e + vTL) \\ &= vL_e + vT(vL_e + vT(vL_e + vTL)) \\ &= vL_e + v(Tv)L_e + v(Tv)^2L_e + v(Tv)^2TL \\ &= \sum_{n=0}^{\infty} v(Tv)^n L_e \qquad (2.5) \end{aligned}$$

An intuitive interpretation of this is that the light from a point is the sum of light reflected 0,1,2,3,... times from the point. This allows a reformulation of the distinction between local and global models. Local models only calculate light which is reflected zero and one times, i.e. light emitted from the point and light received directly from a light source, and disregard visibility for the reflections. Global models calculate light reflected *any* number of times, using a local model when calculating a single reflection.

2.2 Lighting in real-time computer graphics

The most general way of representing a BRDF is to sample it for a number of incoming and outgoing directions and then use the samples as a lookup table, interpolating the values. Since most BRDFs are not smooth quite a lot of samples of this six-dimensional function[3] is required. In real-time applications, this is often too expensive. To develop a simpler model we will look at special types of BRDFs that are interesting from a performance point of view. The *standard lighting model for real-time applications* is a result of attempts to model special types of BRDFs and the multiple reflections of light described by the rendering equation.

2.2.1 Diffuse BRDFs

A perfectly diffuse[4] surface reflects incoming light equally in all directions. Examples of this are dull, mat materials such as chalk or soot. The BRDF for such surfaces is constant under a change of outgoing direction (a direct consequence of reflecting equally in all directions). A change in the incoming direction will still

[3]A BRDF in the direction formulation is six dimensional: two values for the coordinates of point on a surface and two values for each direction since a direction can be represented in spherical coordinates.

[4]Also known as a Lambertian surface.

change the BRDF since the light received by x' depends on the orientation of the surface which x' is a part of. In figure 2.3 we see a beam of light with unit width hitting a surface at an angle θ. The area covered by the beam is equal to $1/\cos\theta$. We can therefore see that light hitting x' should be scaled by $\cos\theta$, where θ is the angle between the surface normal $\vec{n}$ and the direction towards the source point. If we call that direction $\vec{\omega}_i$, as in the direction formulation of the BRDF, and assume that both $\vec{n}$ and $\vec{\omega}_i$ are normalized, then $\cos\theta$ can be calculated as $\vec{\omega}_i \cdot \vec{n}$ (the dot product of $\vec{\omega}_i$ and $\vec{n}$).

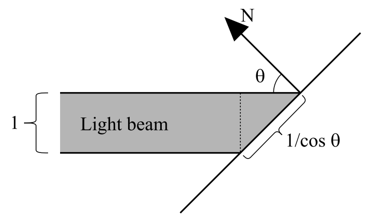

Figure 2.3: An incoming beam covers an area of $1/\cos\theta$.

This $\cos$ factor is fundamental in the sense that all physically correct BRDFs must include this. Except for the cos factor the BRDF for a perfectly diffuse surface is constant.

2.2.2 Specular BRDFs

Another special case of BRDFs is the perfectly specular surface. Here the incoming light is always reflected exactly in the mirror direction[5]. Examples of perfectly specular surfaces include mirrors and still water surfaces. If we disregard the $\cos$ factor on the incoming direction, a BRDF for a perfectly specular surface in the direction formulation is:

$$BRDF(x, \vec{\omega}_i, \vec{\omega}_o) = \left\{ \begin{array}{ll} \rho & \text{if } \vec{\omega}_o = \text{mirror}(\vec{\omega}_i) \\ 0 & \text{otherwise} \end{array} \right\}$$

Where ρ is a constant which tells how much of the light is reflected versus absorbed. Since few materials are perfect, light is often reflected in a small cone around the mirror direction. A surface, which is mostly specular, but not perfect, is called *glossy*. In figure 2.4 we see a visualization of a perfectly diffuse, a perfectly specular and a glossy BRDF.

[5]The mirror direction is where the incoming angle equals the outgoing angle, and it can be calculated as: $\text{mirror}(\vec{\omega}) = 2(\vec{\omega} \cdot \vec{n})\vec{n} - \vec{\omega}$.

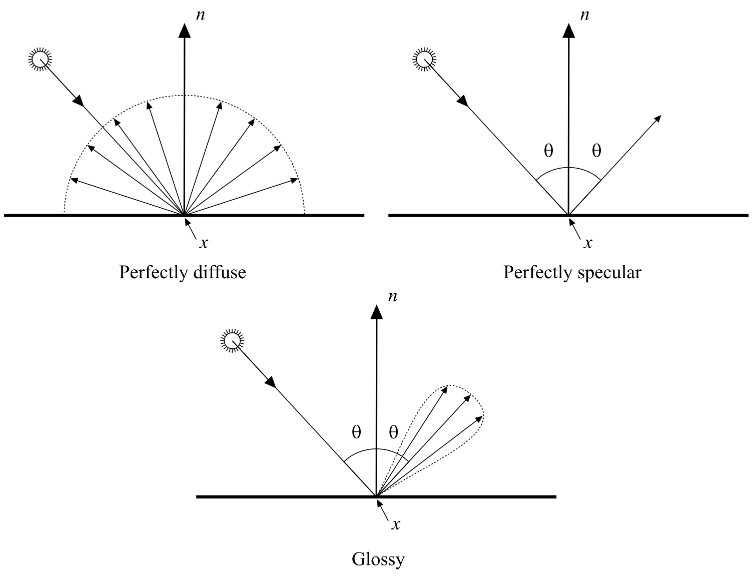

Figure 2.4: Special cases of BRDFs.

2.2.3 Ambient light

If we look at the rendering equation as written in equation 2.5 it is the sum of light reflected 0,1,2,3,... times. The local model we are about to describe makes a fair approximation of the 0th and 1st reflection but disregards the rest. These missing reflections would give a subtle illumination in a real environment, even on surfaces that are not directly illuminated by any light sources. This is because it is usually possible to find a path which, when reflected enough times, eventually reaches a light source from any one point.

A very crude but cheap approximation to this is to introduce an *ambient* term. The global ambient light is defined as a constant amount of light that illuminates *all* objects in the scene. Furthermore, light sources emit ambient light which is received by objects independent of their orientation. Different objects can still reflect this light differently however and the ambient term for a point is therefore the ambient light multiplied by an *ambient reflection coefficient* for the point.

2.2.4 The standard lighting model for real-time applications

The three different terms introduced above can be combined to form a local light model, which is used by allmost every real-time application[6]. It can be written as:

[6]We shall not describe all the details of the standard light model. For a more thorough description see f.ex. [WND+99] pp. 211 - 215 or [FvDFH91] section 16.1.

$$
\begin{aligned}
L(x) &= x_{emission} \\
&+ global_{ambient} * x_{ambient} \\
&+ \sum_{i=0}^{N} L^i_{attenuation} * \left(\begin{array}{l} L^i_{ambient} * x_{ambient} + \\ L^i_{diffuse} * x_{diffuse} * (\vec{n} \cdot \vec{\omega}_i) + \\ L^i_{specular} * x_{specular} * (\vec{n} \cdot (\vec{\omega}_i + \vec{\omega}_o))^{x_{shininess}}) \end{array} \right)
\end{aligned}
$$

Where x_p means property p of the point x, and L^i_p is property p of the ith light source.

The equation states that the color of a point x is the sum of the emission, global ambient and the ambient, diffuse and specular contribution from all N light sources in the scene. The contribution from each light source is dependent on the distance between the light source and the point. This attenuation is called $L^i_{attenuation}$ in the equation and is defined as:

$$L^i_{attenuation} = \min\left(1, \frac{1}{L^i_c + L^i_l d + L^i_q d^2}\right)$$

Where d is the distance between x and the ith light source. L^i_c, L^i_l and L^i_q are the *constant*, *linear* and *quadratic attenuation parameters* respectively. They can be adjusted individually for each light to achieve a certain attenuation behavior. The total attenuation factor is clamped to the range $[0..1]$ as light cannot be negative, and considering that the attenuation should never make a light stronger than its original power.

The ambient contribution from a light source is only dependent on the distance to the light source, whereas the diffuse contribution is scaled by the cos factor as described above. The specular contribution from a light uses a model proposed by Phong Bui-Tong[BT5x]. It models glossy surfaces by assuming that the light specularly reflected is dependent on the angle between the outgoing direction and the mirror direction, and a material dependent *shininess* factor ($x_{shininess}$). The dot product between the normal vector and the mirror vector, which are normalized before use, returns a number in $[0..1]$. Raising this number to the power of the shininess factor, which is typically between 1 and 256, gives a function which is 1 when the angle to the the mirror direction is 0 and falls of quickly with increasing angle. In early implementations the true mirror direction was not used for efficiency reasons, instead the *halfway vector*, calculated as $\vec{\omega}_i + \vec{\omega}_o$, was used. Current implementations can easily afford to calculate the true mirror direction and use that as input to the Phong model.

The standard lighting model is a highly empirical model. It uses approximations which work well in many cases, but it has no grounding in any theoretical model of light interaction. Phong's model for specular reflection is a exactly such an imperial model. Furthermore, the calculations for the ambient, diffuse and specular contributions are totally separate. This allows light sources to emit red 'specular' light and green 'diffuse' light for example. This wouldn't be physically correct, but it gives a lot of artistic freedom to achieve a certain look.

The reason for the popularity of the standard lighting model is probably a combination of four things: it is relatively cheap to compute, it is conceptually simple, it models many of the most important aspects of light/object interaction, and it allows a great degree of artistic freedom. The two most used real-time APIs, OpenGL and DirectX, implement the standard model in their fixed function pipeline, and previously applications were forced to use the standard model if they wanted to take advantage of hardware acceleration. This has changed with the introduction of the programmable pipeline which allows you to implement any model you desire.

2.2.5 Vertex vs. pixel based lighting

The standard lighting model, described above, can be evaluated per fragment on current hardware. Previously this was too computationally expensive and a different approach was used. The light model was only evaluated per vertex and the resulting color was then interpolated across the triangle. This is called *Goraud Shading*, (see [FvDFH91] section 16.2.4). The two approaches are also called *per-vertex lighting* and *per-pixel lighting*. Per-vertex lighting is of course computationally cheaper but it has several visible deficiencies, where the most important ones stem from the fact that the interpolation cannot produce higher values than the values at the vertices. Therefore a fragment in the center of a triangle cannot be brighter than the fragments at the vertices. Consequently areas where the lighting changes rapidly, such as specular highlights and light sources very close to the geometry, will exhibit visual artifacts, especially in animated scenes. These problems will become less noticeable if objects are highly tessellated, (using more vertices and triangles), but as mentioned with current hardware it is possible to use per-pixel lighting which of course produces the best results.

Figure 2.5 shows a simple scene rendered with per-vertex lighting on the left, and with per-pixel lighting on the right. The left side also shows a wire-frame view of the wall segment being illuminated. As is seen, the two different approaches results in very different images. With per-vertex lighting the light does not seem to have any effect at all. And indeed, with this setup, the light does not affect the image since none of the vertices that make up the wall segment fall within its sphere of influence. With per-pixel lighting, an attenuation value for the light is calculated at each fragment and as a result, the wall is correctly lit.

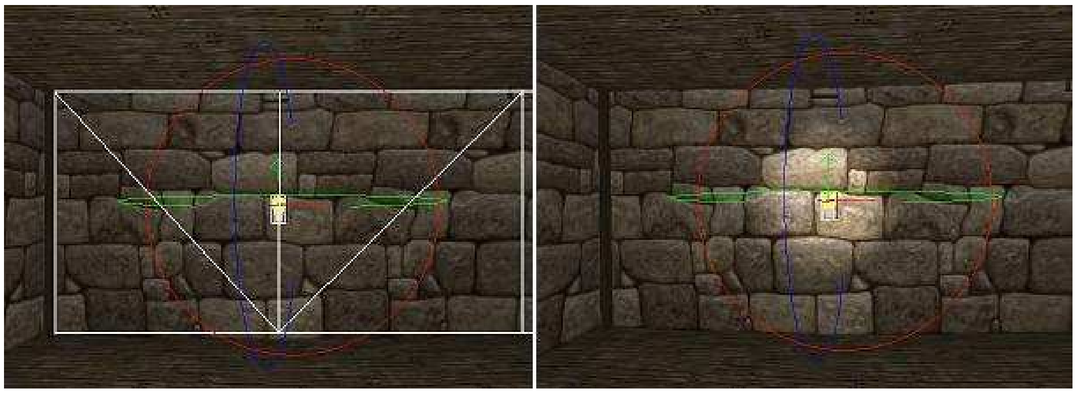
Figure 2.5: Per-vertex and per-pixel lighting.

Chapter 3

Shadow techniques

An exact solution to the rendering equation discussed in the previous chapter is not possible. Various off-line rendering techniques such as raytracing, photon mapping and radiosity give approximations to the rendering equation where the visibility functions are taken into consideration. This means that these techniques automatically produce images where shadows, sometimes even soft shadows, are included. However, the only feasible solution for real-time rendering is currently rasterization, which in its basic form uses the local light model described in section 2.2.4. By definition, using a local light model means that shadows are not included, but as we shall see in this chapter various algorithms exist that extend the rasterization approach with a visibility function for direct illumination.

In this chapter we first give an overview of the most important real-time shadow algorithms. We then focus on a particular technique called the stencil shadow algorithm, which is used by many real-time applications today. Several optimizations, improvements and versions of the stencil shadow algorithm are described as is how they can be viewed as approximations to the rendering equation.

3.1 Overview of shadow algorithms

For real-time applications shadow algorithms can be split into three groups that we have named limited, static and general.

Limited algorithms operate in environments with very restrictive assumptions. An example of this is the projective shadow algorithm [Bli88], which assumes that the object receiving shadow is a plane with a known orientation and position and that no objects are positioned between the shadow caster and the plane. Limited algorithms serve a purpose in specific environments, e.g. a CAD system, but are not widely used today.

Static approaches are characterized by their assumptions that objects and lights are stationary and it is therefore possible to precalculate shadow and light information. An example of a static algorithm is lightmaps, (see [AMH02] section 5.7.2), which precalculates light, and thus shadow, information to textures. Objects are then rendered with this additional texture modulated on top of their usual texture map. Since the precalculation can be a global illumination calculation, very good image quality can be achieved. Furthermore, as the rendering of an extra texture is something that all modern graphic cards excel at, the algorithm is very fast. Yet the algorithm also has its drawbacks, such as the static nature of the textures, which prevents non-stationary objects from casting shadows. Furthermore, the storage space required for the lightmapping algorithm can be fairly large since a lightmap must be stored for each triangle in the scene. Despite all of this, the lightmap algorithm is heavily used in many real-time applications, often to good effect.

General algorithms try to calculate shadows in a general environment where very little can be assumed about the nature of the shadow casters or the shadow receivers, and where all objects and lights can move freely around the scene.

This division of shadow algorithms into three groups is quite crude, and a lot of research has been conducted to create algorithms that cross these boundaries to reach a good compromise between the advantages and drawbacks of each group. See [AMH02] section 6.12 for an overview of real-time shadow algorithms. In recent years, research has focused on generel algorithms because developments in hardware have rendered the limited and static algorithms relatively simple to execute. Our focus for the remainder of this thesis will be on general algorithms only.

The two most successful general algorithms are *stencil shadows* and *shadow maps*. Stencil shadows will be explained in detail below. The basic observation with regard to shadow maps[Wil78] is that between the view-space of the observer and the view-space of a light source there exists a linear mapping (expressible by a 4x4 matrix and therefore cheap to calculate). The algorithm has two passes: In the first pass the scene is rendered into the light's view-space and the depth information, (i.e. how far away every fragment is from the light), is stored in a *shadow map*. In the second pass the scene is rendered normally and it is now possible to determine whether a fragment can 'see' the light source. This is determined by transforming the fragments position into the lights view-space and comparing its depth to the stored depth value in the shadow map. If the fragment is farther away than the stored depth value another object must be blocking the path from the fragment to the light source, and it is therefore in shadow.

The shadow map algorithm was first proposed by Williams in 1978 [Wil78], and perhaps the most important addition to it has been the paper by Segal et al. [SKv$^+$92], in which they noted that the required computations are very similar to the ones required for perspective correct texture mapping and therefore already implemented in hardware. See [ERCwn] for a detailed description of how this is accomplished. Since shadow maps can be hardware-accelerated it is quite fast, and it is used in many real-time applications today. Its most important advantage is its versatility: it can cast shadows from and onto everything that can be rendered by the application, the only exception being objects with semi-transparent areas[1]. It has one major drawback however: the discretization and limited precision of the shadow map can result in very visible artifacts, for example in the form of jagged shadow edges. Even though many improvements has been suggested the pixel precise shadows, seen in for example the stencil shadow algorithm, have not yet been achieved.

3.2 Stencil shadows

Crow presented the stencil shadow algorithm in 1977 [Cro77] under the name *projected shadow polygons*. In 1991 Heidmann suggested[Hei91] to use the stencil buffer to implement Crow's original algorithm which gave the algorithm the name by which it is best known today. Stencil shadows belongs to the group of *volumetric shadow algorithms* as the shadowed volume in the scene is explicit in the algorithm.

The basic idea in the algorithm is to generate, for each object and light pair, the volume which is in shadow from the light. When shading a fragment it must then be determined if it is inside any of these volumes. This idea is depicted in figure 3.1. The shadow volumes are the gray areas where the two spheres cast shadow. The volumes should ideally extend to infinity, but it is sufficient that they extend to the far side of the geometry. We will elaborate on the extension of the volumes later. When shading a pixel we trace a line from the eye to the fragment and count the number of entries into shadow and the number of exits out of shadow. If the number of entries are greater than the number of exits the point must be in shadow, otherwise it is lit by the light[2]. Take f.ex. the point p_1. This point is in shadow since the number of shadow volume entries (1) is greater than the number of exits (0). Point p_2 on the other hand is not in shadow since the number of entries and exits are both 1. Note how this approach also works across multiple shadow volumes where the ray passes all the way through one or more shadow volumes before reaching the pixel. Point p_3 is correctly classified as being in shadow since the number of entries (2) is greater than the number of exits (1). This approach assumes that the view point is outside any shadow volume. An

[1]Transparent areas in a texture are handled correctly if they are 100% transparent.

[2]Other methods for determining whether a pixel is inside a volume or not are possible but the line-trace algorithm is close to the one suggested by [Cro77] and lends itself nicely to hardware acceleration.

improvement that removes this restriction will be described in section 3.3.

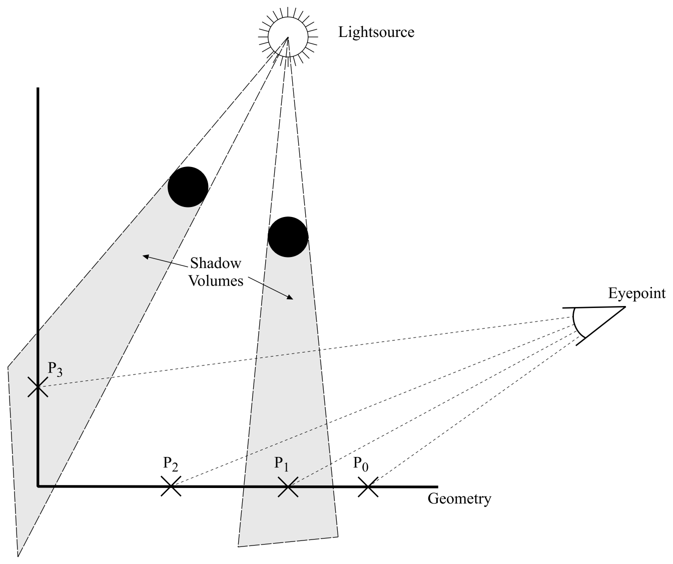

Figure 3.1: Stencil shadows: Rays from eye to pixel

As we are only interested in a fragment's color if it is visible, (i.e. not covered by fragments closer to the eye), we can think of the lines from the eye to the fragments as *view rays* beginning at the eye, going through the center of a pixel on the screen, and hitting the first visible fragment in the scene. The view rays count the number of entries and exits and we can therefore determine whether the ray is in shadow or not when hitting the fragment. By emitting view rays from the eye through *all* pixels on the screen we would then have the shadow information for the final image. Fortunately, it is not necessary to do actual ray tracing to implement this, but that is the conceptual idea of the algorithm. Before we describe a different implementation, we first examine how to model the shadow volume.

3.2.1 The shadow volume

The stencil shadow algorithm assumes that shadow casters consist of an *opaque* triangle mesh and that light sources are modeled as points, (i.e. have zero radius). A *shadow mesh* can the be build consisting of ordinary, but invisible, geometry which models the actual shadow volume. For a single triangle it would consist of three quads[3], each extending from a triangle edge to infinity, away from the light. More precisely, the line where two quads meet extends through the corresponding

[3]*Quadrangle* is usually shortened to *quad*.

vertex in the exact opposite direction of the vertex to light direction. Figure 3.2 is an illustration of this.

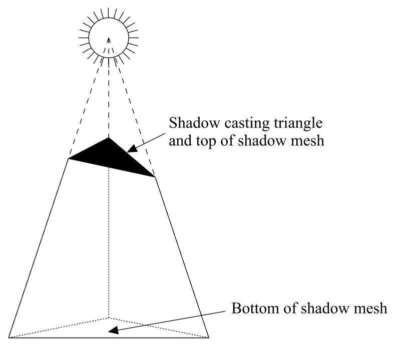

Figure 3.2: Shadow volume for a single triangle.

For a general mesh it is not necessary to create quads from all edges. Consider two triangles sharing an edge. If both triangles face the light then a quad extending from the shared edge would be unnecessary since it would represent neither an entry nor an exit from the shadow volume. At first glance it would appear that the edges which ought to generate the shadow mesh should be those on the silhouette[4] of the mesh, (as seen from the light). However, there are at least three reasons why this is not a good approach. Firstly, the silhouette is, in general, a collection of *parts* of edges and this would complicate the generation of the shadow mesh. Secondly, if we only create quads from the silhouette we would not calculate correct self-shadowing[5] in all cases. Finally the computation of the silhouette is quite expensive. We therefore use a simpler approach and generate the shadow mesh from the *contour* edges. These are the edges which have either only one front-facing neighboring triangle, or where two neighboring triangles have different orientations toward the light, (i.e. where one is facing towards and the other away from the light). To calculate the contour edges we start by creating a list of all edges and a data structure through which we can find the neighboring triangles for a given edge in constant time. This is a precalculation step whose result remains valid as long as the connectivity of the mesh does not change, (the vertices can change position without affecting connectivity, so it is possible to have animated meshes). To calculate a shadow mesh we then make a single pass through all edges and calculate whether they are a contour edge or not. Given the previous definition and data structure we see that this can be done in constant time

[4]The silhouette is the outer edge of an object as seen from a particular point

[5]Self-shadowing occurs when a part of an object cast shadow on another part of the same object.

for a single edge. The calculation is therefore linear in the number of edges in the mesh.

Note that for the algorithm to work it is not necessary for the shadow mesh to be closed at the top or bottom. It would seem that view rays could then enter and exit the volume without counting entries and exits correctly, but this is not the case. A view ray can never pass through the missing bottom of the shadow mesh since the bottom is at infinity and the fragment is therefore closer. It cannot pass through the missing top either because we have assumed that the shadow generating mesh is opaque. The fragment will therefore be on the shadow generating mesh, or possibly in front of it, stopping the ray before it enters the shadow volume.

3.2.2 Using the stencil buffer

Emitting view rays and tracing them through the world would suggest a ray-tracing implementation but, as Heidmann suggested in 1991 [Hei91], given both a stencil and z-buffer it is possible to use a rasterization approach.

Since we are not interested in the actual number of shadow entries and exits only whether the first is greater than the latter, we introduce the *shadow value* which is the difference between the two. A shadow value greater than zero then means that a fragment is in shadow. The basic idea in the stencil shadow algorithm is to let the shadow value be stored in the stencil buffer and update its value for all pixels covered by a shadow mesh triangle, instead of calculating it for a single pixel before moving on to the next. The z-buffer is used to determine whether the fragment on the shadow mesh triangle is in front of or behind the corresponding fragment on the geometry.

Figures 3.3, 3.4 and 3.5 show how the stencil values are updated. In the figures we see, in a 2d 'side view', some geometry, a shadow caster, a light source and a view point. The stencil buffer is visualized on the far left. To make the figures simpler we have used a parallel projection onto the stencil buffer, and the stencil value for a fragment can therefore be found by moving horizontally to the left. In figure 3.3 we see the initial setup where the stencil buffer is cleared to zero. In figure 3.4 we see the effect on the stencil buffer after the first shadow mesh triangle, (the fat line), has been rendered. This triangle is front facing and all values between the lines l_0 and l_1 have therefore been incremented to one as a front facing shadow mesh triangle represents an entry into shadow. Values below l_1 have not been affected as the geometry was in front of the shadow mesh triangle and consequently the z-buffer test has culled away those fragments. In figure 3.5 the second shadow mesh triangle has been rendered. This triangle is back facing and will therefore decrement the stencil values since it represents an exit from shadow. This has been done between the lines l_2 and l_3. Again, values below l_3 has not been affected because of the z-buffer test. We end up with the stencil-

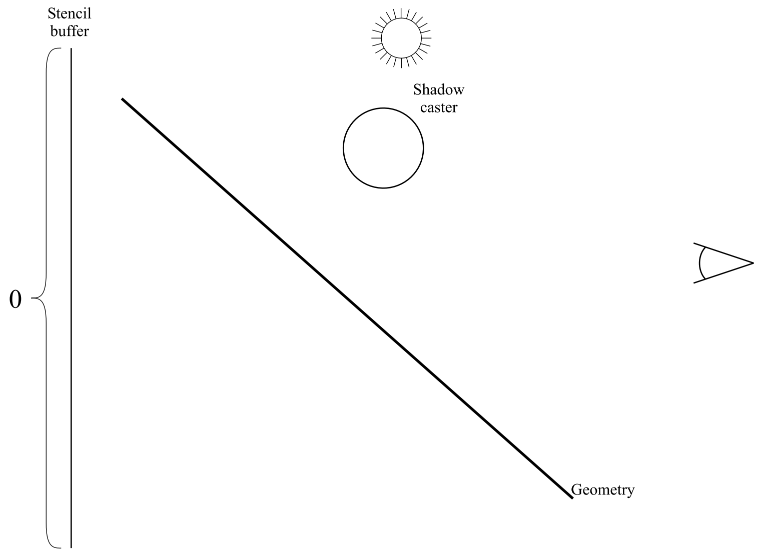

Figure 3.3: Stencil values: before rendering shadow triangles.

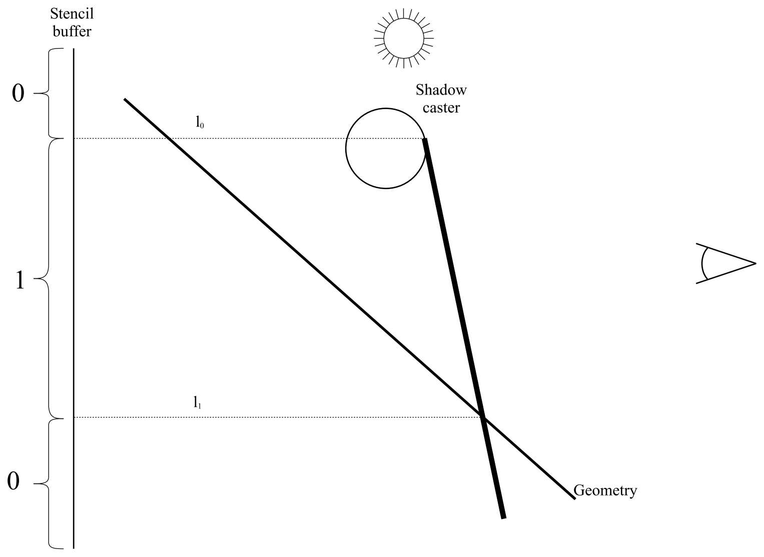

Figure 3.4: Stencil values: after rendering one shadow triangle.

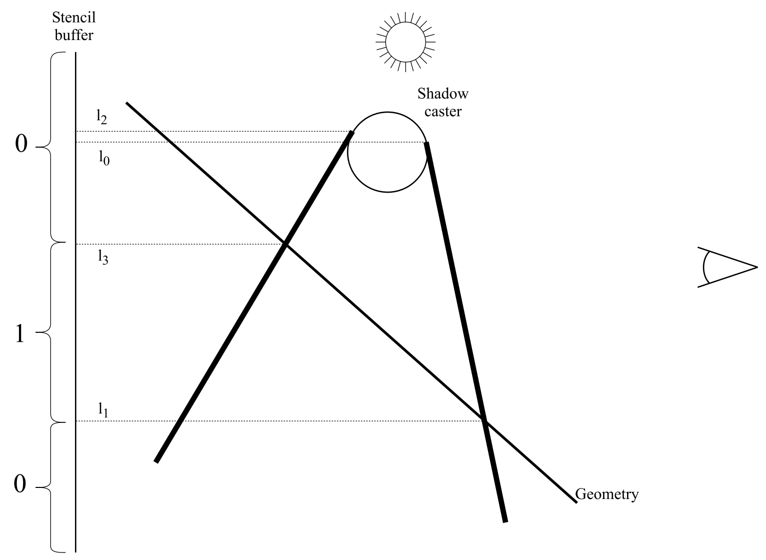

Figure 3.5: Stencil values: after rendering two shadow triangles.

buffer containing the value one in the area between the lines l_3 and l_1, and the part of the geometry that is in shadow corresponds to this area exactly.

The algorithm that uses this basic idea is a multi-pass algorithm. In the first pass it renders the scene once to fill the z-buffer. In the second pass it renders the shadow meshes to fill the stencil-buffer, as described above. In the final pass it renders the scene once again to add the light contribution from the light source. However in this pass the stencil test is used to cull away fragments where the stencil value is less than or equal to zero. This prevents the rendering from taking place in the shadowed areas. A more detailed description is:

1. Clear color-buffer, z-buffer and stencil-buffer.
2. Render the scene with only ambient and emissive lighting.
3. Disable writing to color-buffer and z-buffer, enable stencil-buffer.
4. Render all *front facing* shadow mesh triangles, *incrementing* the stencil value when passing the z-test.
5. Render all *back facing* shadow mesh triangles, *decrementing* the stencil value when passing the z-test.
6. Re-enable writing to color-buffer, set z-buffer test to equal, set stencil test to pass when value is less than 0 and use additive blending.

7. Render the scene with only diffuse and specular lighting.

Step 2 ensures that all fragments, including those in shadow, have both ambient and emissive lighting and serve to fill the z-buffer. Step 3 ensures that the shadow mesh rendered next does not affect the color-buffer, (directly at least), and that we can use the values in the z-buffer without overwriting them. Steps 4 and 5 are the essential ones that perform the counting of entries and exits as described above. Step 6 sets the z-buffer test to equal, ensuring that only the exact same fragments, that were visible in the first pass, are rendered. Step 6 also enables additive blending, meaning that the calculated color will be added to the previous content of the color-buffer, and sets up the stencil test so that only the pixels, (or rather their corresponding fragments), which are outside shadow will be rendered again.

One way to render only front facing or back facing triangles is to use the CPU to classify the triangles into these two categories and then only render the correct subset of the mesh. Another way is to use the GPU's capabality of rejecting triangles based on the order the vertices appear in when projected to the screen. If the triangles of a mesh are generated with a consistent ordering, either clockwise or counterclockwise, the projected order of their vertices determines whether they are front or back facing. Triangles are usually generated using a clockwise ordering of their vertices. Rendering only front facing triangles can then be accomplished by letting the GPU cull away all counterclockwise triangles. The entire shadow mesh can then be rendered in steps 4 and 5, and while this may seem wasteful at first, it allows the hardware to perform the orientation calculation and minimizes state changes as well as the amount of data sent to the GPU. More importantly, it allows the use of *two sided stencil* as described below.

So far we have assumed that there is only one light-source in the scene, which is rarely satisfactory. Fortunately it is easy to generalize the algorithm to handle multiple light-sources. The steps involved are:

1. Clear color-buffer and z-buffer.

2. Render the scene with only ambient and emissive lighting.

3. For all lights l:

    (a) Clear stencil-buffer, disable writing to color-buffer and z-buffer, set z-buffer test to less-than.

    (b) Render all *front facing* shadow mesh triangles generated by l, *incrementing* the stencil value when passing the z-test.

    (c) Render all *back facing* shadow mesh triangles generated by l, *decrementing* the stencil value when passing the z-test.

(d) Re-enable writing to color-buffer, set z-buffer test to equal, set stencil test to pass when value is 0 and enable additive blending.

(e) Render the scene with only diffuse and specular lighting from l.

This is a simple extension of the algorithm on page 24. After the first pass, which fill the z-buffer, we loop over all lights in the scene. Since the shadowed areas for one light is completely independent from those of other lights we must clear the stencil buffer before each light pass, which then proceeds exactly as previously defined. The additive blending ensures that we end up with the sum of the light contributions from each light plus the ambient and emissive light.

3.3 Improving stencil shadows

The stencil shadow algorithm, as described above, is easy to implement and by using the stencil buffer it can be hardware-accelerated and is therefore quite fast. Unfortunately it has a very serious drawback, which limited its use in applications for years: it does not work when the eye point is inside, or very close to, a shadow volume since the volume will be cut open by the near clip plane, resulting in the view rays missing a shadow-volume entry. We will now describe how to overcome this problem along with some optimizations for the algorithm.

3.3.1 Carmacks reverse

In 2000 Carmack suggested[Car00] a slightly different approach which entails that the view rays are traced from infinity towards the eye, stopping when encountering the pixel on the geometry that is closest to the eye. This reversal of the view rays' direction has given the algorithm the name *Carmacks reverse*. The two different approaches have also been named *zpass* and *zfail*, as the stencil buffer in the original algorithm is changed only when a fragment *passes* the z-test. As we will show below, Carmacks reverse can be implemented by changing the stencil values only when the z-test *fails* for the fragment, i.e. by using the *z-fail* stencil operation.

When using zfail the shadow mesh must be closed in both top and bottom. Figure 3.6 shows two cases where the lack of a top and bottom in a shadow mesh would result in incorrect shadow count for the shown pixels. In the leftmost part we see an example where the reversed view ray enters the shadow volume through the missing bottom and therefore fails to count a shadow entry. The fragment will therefore be shaded as if it was affected by the light, even though it is clearly in shadow. In the rightmost part of the figure, we see the reversed view ray entering the shadow volume through the sides and therefore counting an entry correctly.

The ray then proceeds through the shadow casting object, which is exactly where the top of the shadow mesh ought to be and the ray therefore fails to count an exit. The shadow-casting object will therefore always appear to be in shadow and even the areas of the object facing the light will be shadowed, which is obviously wrong.

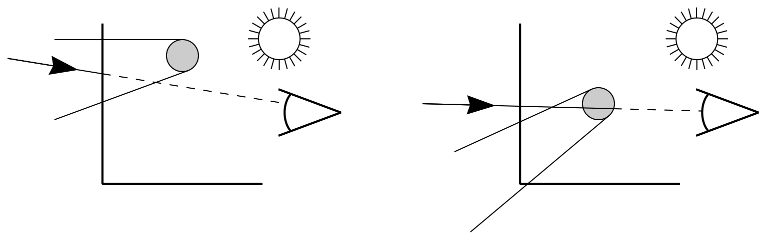

Figure 3.6: Lack of top and bottom in the shadow mesh

Generating the required top and bottom 'cap' for a general mesh can be complicated, but assuming a *closed*[6] mesh it is much simpler. For a closed mesh we can use the front facing triangles, as seen from the light, as the top cap. The bottom cap can then be generated from the back facing triangles by extruding them away from the light, as described in section 3.2.1. Since the mesh is assumed to be closed, all edges have exactly two triangles connected to it. The silhouette edges are therefore those whose triangles have different facings with regard to the light. From the above we know that the back facing triangle has been extruded away from the front facing one, tearing open the mesh. To close the shadow mesh we insert a quad at every silhouette edge connecting the top, which is in its original position, and the bottom, which is now at infinity.

Given the closed shadow mesh, we can implement the zfail algorithm by using the steps described on page 24 with a few changes. When rendering front facing triangles we *decrement* the stencil value when a fragment *fails* the z-test, and when rendering back facing triangles we *increment* the stencil value when the fragment *fails* the z-test. This implements the tracing of a line from infinity towards the eye since all shadow mesh triangles *behind* the visible geometry now affect the stencil values, and since those triangles that are back facing to the view position are now front facing to the view ray.

Assigning a vertex a position at infinity is not a problem, we simply use homogenous coordinates and set the w component of the vertex to zero. However,

[6]What we actually need, is that each edge in the mesh is connected to exactly two triangles, but this is the same as having the intuitive property of being closed: There is an 'inside' and an 'outside' of the object and we cannot move between them without passing through one of the triangles.

with a regular projection matrix those vertices would be clipped by the far clip plane and in this case it would mean that the bottom of the closed shadow mesh would be clipped away, resulting in the reversed view ray missing a shadow volume entry. There are at least three ways to correct this problem. As Everitt and Kilgaard suggest[EK02], it is possible to create a projection matrix which places the far plane at infinity, meaning that it will never clip any triangles. In the same paper they suggest the use of *depth clamping*[7] which achieves the same thing without using a special projection matrix.

The third solution to the problem is to use an *extrusion distance* less than infinity. The extrusion distance is the distance the vertices of the bottom cap of the shadow mesh are extruded away from the light. We can use an extrusion distance which places the bottom of the shadow mesh so far away from the light that the contribution from the light behind the bottom is zero or negligible. This makes sense for attenuated light sources whose light contribution decrease with increasing distance. Directional lights on the other hand, which are conceptually located at infinity, are of course not attenuated and as a result, it is hard to calculate an extrusion distance that is guaranteed to be big enough. Even for attenuated point lights it is easy to construct cases where any finite extrusion distance is too short.

In figure 3.7 we see a light source, a shadow casting object and the corresponding shadow mesh. The sphere of influence is the distance beyond which the light does not affect the shading of a fragment. The extrusion distance has been chosen so the vertices are outside the sphere of influence but the shadow mesh still leaves a non-shadowed region, which is lit by the light although it should not be. Choosing any finite, (and fixed), extrusion distance we see that it is possible to move the light so close to the shadow casting object that we still have a non-shadowed region. Of course it is possible to calculate the required extrusion distance for each case, but this requires that we, at each vertex, have information about all triangles which the vertex is part of. This complicates an otherwise very simple algorithm, and more importantly, this information is not available in a vertex shader. This implies, that vertex shader shadow volumes, as will be described in section 3.3.3, cannot be used.

By choosing a large extrusion distance it is also possible to make a shadow mesh big enough to be clipped by the far plane. This would then require us to use either a far plane at infinity or the depth clamp approach anyway. The finite extrusion distance solution is therefore not without problems but there is one aspect that makes it interesting: performance. Making the shadow volume smaller means that its projected screen size will also be smaller which reduces the number of stencil operations. This can have a significant impact on performance. Furthermore, the

[7]Depth clamping is currently only available in OpenGL using the *NV_depth_clamp* extension

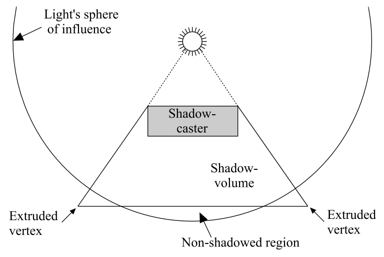

Figure 3.7: Shadow mesh extrusion distance.

artifacts introduced are usually not very obvious since the non-shadowed region is usually small and in an area where the light has been attenuated so its contribution is negligible.

The changes to the original algorithm described above enable correct shadow calculation when the eye is inside a shadow volume and is actually a very robust and practical algorithm. The extra cost incurred by the added top and bottom of the shadow volume is well spent in most applications.

3.3.2 Two-sided stencil testing

Another change to the algorithm is the use of *two-sided stencil testing*[8]. This is a new feature in recent GPUs that allows the application programmer to set up and use different stencil states and operations for back facing and front facing triangles respectively. With this functionality it is possible to exchange steps 4 and 5 of the algorithm on page 24 with a single step that renders all triangles just one time, without any orientation based culling. The GPU then decides which triangles are front and back facing and uses the correct stencil tests and operations according to the orientation. This reduces the CPU load of issuing render calls to the driver, reduces the amount of vertices that have to be transformed through a vertex shader, and reduces the amount of mesh data that has to be sent to the graphics card over the AGP bus. The net result is a significant performance gain.

3.3.3 Vertex shader calculation of shadow mesh

With the addition of programmable vertex shader functionality another significant change to the algorithm is possible. The calculation of the shadow mesh can be moved from the CPU to the GPU, which of course lifts a burden from the CPU but, more importantly, it also means that the shadow mesh data can be sent to the

[8]First proposed in [EK02]

GPU once instead of every frame as was previously necessary[9]. The idea is to give the GPU a copy of the shadow casting mesh, but one where it is possible for every edge to stretch and become a quad on the side of the shadow mesh. This is accomplished by inserting a quad of zero width between every edge of the mesh. Figure 3.8 shows four triangles and how the edges between them are replaced with quads. The leftmost picture shows the original triangles. In the middle picture all edges have been replaced with quads, which should be of zero width but since that represent some visualization difficulties we have shown them stretched a bit. In the rightmost picture, we see the leftmost triangle extruded a bit away from the others. The quads bordering this triangle have been stretched accordingly while all the others have zero width and are therefore invisible.

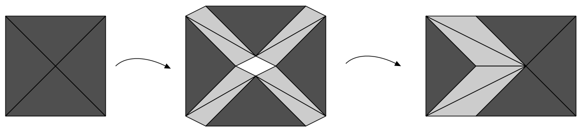
Figure 3.8: Vertex shader shadow mesh.

Since a vertex shader calculates the projected-space position of a vertex, the vertex shader can take all the points belonging to back facing triangles, (as seen from the light), and extrude them away from the light. All points belonging to front facing triangles are projected to their usual positions. To determine whether a point belongs to a front or back facing triangle the vertex shader must know the face normal of the corresponding triangle. This is necessary since the shader cannot access the other points that constitute the triangle and therefore cannot calculate the normal itself. This is not a problem as the face normal is simply stored in the vertex data in the same manner as when using the normal to calculate lighting. For the edges that contain quads which are not stretched, the quad will still have zero width and therefore covers no pixels and contributes nothing to the shadow calculation. Section 6.5 shows the vertex shader code for extruding a mesh.

The only problem with this approach is that the number of triangles in the shadow mesh grows a lot. We will now analyze exactly how much. A triangle is bounded by exactly three edges and, since the mesh is closed, an edge is incident to exactly two faces. Therefore a single edge 'generates' two times one third of a face. Assuming t is the number of triangles and e the number of edges we have:

$$t = \frac{2e}{3} \iff e = \frac{3}{2}t$$

[9]As long as a mesh is unchanged the graphics card can cache it in AGP memory, but to change it the application has to modify a version in system memory and upload it again.

A VS shadow mesh[10] contains two new triangles for each edge plus the original triangles. So the number of triangles in the shadow mesh t' is related to the original number of triangles as:

$$t' = t + 2e = t + 2\frac{3}{2}t = 4t$$

The shadow mesh must also contain new vertices. In fact, the original triangles can no longer share vertices since it must be possible for each of them to be extruded separately. The quads, however, share the vertices with the triangles they separate, hence the number of vertices in the shadow mesh v' is:

$$v' = 3t$$

The number of vertices in the original mesh can be calculated from Euler's formula, which states that:

$$v - e + t = 2$$

so we have that:

$$v - \frac{3}{2}t + t = 2 \iff v = 2 + \frac{1}{2}t \iff t = 2(v - 2)$$

and therefore:

$$v' = 3t = 3 * 2(v - 2) = 6v - 12$$

This means that there are four times as many triangles and about six times as many vertices in a VS shadow mesh as in the original mesh. Of course a regular CPU-calculated shadow mesh also has additional triangles, but a VS volume is the 'worst-case scenario'. However, given the performance characteristics of current GPUs where vertex processing is cheap compared to sending data from the CPU to the GPU, the VS shadow volumes still performs better than regular CPU volumes. The table below shows performance measurements in FPS for the screen-shots shown in figures B.1 to B.4[11]:

Location	CPU volumes	VS volumes	Difference
B.1	7.5	18.5	146%
B.2	24.0	33.5	39%
B.3	14.0	61.0	335%
B.4	38.0	39.0	3%

[10]Vertex shader shadow mesh.

[11]The test was carried out on a 900MHz AMD Athlon with an ATI Radeon 9700 Pro in a 1024x768 resolution with per-pixel lighting.

As is seen in the table, the performance difference varies a lot: from almost nothing to a four times increase. This is the result of the number of shadow volumes in the four scenes. In scenes with very few shadow volumes the work load introduced by either method is so small that the difference in performance is mimimal. This is the case in the fourth location in the table above. But in scenes with lots of shadow volumes (animated volumes in particular) the VS volumes lead to great performance gains due to the fact that they need not be calculated and transferred to the graphics card every frame.

3.4 The single-pass stencil shadow algorithm

The stencil shadow algorithm described on page 25 is the 'correct' version of the algorithm in the sense that it adds contributions from a light source to a fragment only if that fragment is not in shadow from the light. This implies that the algorithm must render the scene one time for each light-source[12]. There exists another algorithm[13] which renders the scene only once. It can be described as:

1. Clear color-buffer, z-buffer and stencil-buffer.
2. Render the scene with all lights enabled.
3. Disable writing to color-buffer and z-buffer, enable stencil-buffer.
4. Render all *front facing* shadow mesh triangles from *all* lights, *incrementing* the stencil value when passing the z-test.
5. Render all *back facing* shadow mesh triangles from *all* lights, *decrementing* the stencil value when passing the z-test.
6. Re-enable writing to color-buffer, disable z-buffer test, set stencil test to pass when value is less than 0 and set additive blending.
7. Render a dark full-screen overlay.

This algorithm first identifies all those areas on the screen that are in shadow from one or more light sources, and it then darkens these areas by a constant amount as a post process. It is usually referred to as the *single-pass* stencil shadow algorithm while the other version is often called the *multi-pass* stencil shadow algorithm. Note that the improvements described earlier, (Carmacks reverse, two-sided stencil etc.), concern how we *find* shadowed areas from a particular light

[12]This is not completely correct, see section 5.4 for optimizations.
[13]We have been unable to find the inventor of the algorithm.

source. These improvements are valid for both algorithms since the distinction between the single-pass and multi-pass algorithm is how we *use* the shadow information.

The multi-pass algorithm identifies the areas which are in shadow before adding the contribution from a light-source. The single-pass algorithm approximates this by adding the contribution from light-sources both in shadowed and non-shadowed areas and later compensates for this by subtracting a constant amount of light from the shadowed areas. The multi-pass algorithm is obviously the most correct of the two as the single-pass algorithm assumes that the contribution from all light-sources to all fragments is a constant, which can subsequently be removed by subtractions. This assumption is wrong for several reasons: different light sources can have different colors, light sources are usually attenuated, shadows from different light-sources can overlap each other, etc. All these factors contribute to a bad approximation, where shadows tend to look unnatural. See figure B.7 for a comparison of the two techniques.

Notice that the single-pass rendering has a constant colored shadow region, covering all pixels that are in shadow from at least one light source (but not necessarily both). As a consequence the sides of the box are darkened, even though a light shines directly on both of them. Another problem is that the shadow is constant colored. Where the two shadows from the barrel cross there should be a darker area, since none of the lights affect this region. Using the multi-pass algorithm all these problems have been rectified.

The only redeeming property of the single-pass algorithm is the performance characteristics. The single-pass algorithm renders the scene once, and for every fragment it calculates the light model for all the lights. The multi-pass algorithm renders the scene once for every light source, each time calculating the light model for a single light only. This means that the single-pass only calculates the transformation and projection of vertices once, whereas the multi-pass does this for every light. Both algorithms calculate the light model approximately the same number of times. Since vertex processing was a limiting factor on older systems multi-pass was very expensive. Current hardware, however, has relatively higher vertex processing power and the use of multi-pass is therefore possible. We have collected performance numbers for both algorithms on the scenes, viewed in the screen-shots in figures B.1 to B.4. The table below shows the results[14]. The second and third column show the number of FPS for the two algorithms, and the last column shows the performance drop when going from single to multi-pass.

[14]The experiment was performed on a 900MHz AMD Athlon with an ATI Radeon 9700 Pro GPU with per-vertex lighting.

Location	Single-pass	Multi-pass	difference
B.1	20.0	18.5	8%
B.2	36.0	28.5	21%
B.3	75.0	61.0	19%
B.4	50.0	43.0	14%

As the table shows there is a performance penalty for using the multi-pass algorithm, these measurements suggests about 15%. However, the performance is extremely dependent on the amount of optimization for both algorithms, and these numbers are therefore valid only for our engine in its current version. But the numbers do show that it is possible to use the multi-pass algorithm in a full-fledged game engine, and we are convinced that the increase in visual quality is well worth the performance penalty.

3.5 Approximations to the rendering equation

As described in section 2.1.2, the rendering equation can serve as a standard for other light models to be measured against. Equation 2.5 is the formulation most suited for this purpose. In this section we will examine how the standard lighting model described in section 2.2.4 and the single and multi-pass shadow algorithms described above can be seen as approximations of this equation. The rendering equation is formulated at a very high level of abstraction, giving us a compact description of a lighting model which captures only the essential elements. In the description below we will link the equations to the more implementation-minded description given above.

The standard lighting model in real-time applications described in section 2.2 can be formulated in the spirit of the rendering equation as:

$$L = v_0 L_e + v_0 T L_{e_0}$$

The first addend is the direct light from a point, i.e. the light reflected zero times. If we look at equation 2.6, L_e corresponds to the $x_{emission}$ and the global ambient term. The visibility function v_0 is the hidden surface removal calculated by the z-buffer. The zero subscript means that it is the visibility for the *last* reflection of light, i.e. it is the visibility between fragments and the camera.

The second addend is the light reflected one time. v_0 still operates on this factor, otherwise we would see light reflected off fragments which the z-buffer has determined to be invisible. L_{e_0} is the emitted light, but the zero subscript means that we allow only point light sources[15]. The T operator is therefore not

[15]Usually only a fixed number of lights are allowed. The fixed function pipeline allows eight.

an integral, as in the rendering equation, but simply a sum over the contribution from these light-sources which corresponds to the sum over the N light sources in equation 2.6. Note that the reflected term is missing a visibility operator which would otherwise make it $v_0 T v_1 L_{e_0}$, where v_1 is the visibility function for the *next-to-last* reflection of light. The standard lighting model does not include this term, which means no shadows are calculated.

The single-pass algorithm can be formulated as:

$$L = v_0 L_e + v_0 T L_{e_0} - (1 - \tilde{v_1})K$$

where K is a constant that determines how 'dark' shadows are. It resembles the standard lighting algorithm, the only difference being the subtraction of a constant where there is 'shadow'. The shadowed regions are determined by the $\tilde{v_1}$ function which approximates the true v_1 function. If $\tilde{v_1}$ is zero, (and $1 - \tilde{v_1}$ is one), we 'darken' the fragment by subtracting a certain amount of light. The $\tilde{v_1}$ function is calculated by the stencil buffer as described in section 3.4, and the reason that it is an approximation is that it computes visibility for *all* light sources in a single pass. This approximation is wrong since it entails that a fragment will be darkened by the same amount regardless of how many light sources that cannot affect it because of shadow. As described above this results in shadows that are notably different from the correct shadows of the multi-pass algorithm.

The multi-pass stencil shadow algorithm can be formulated in the following way:

$$L = v_0 L_e + v_0 T v_1 L_{e_0}$$

Since the multi-pass algorithm calculates visibility for each light source separately, the visibility function for reflected light is therefore the 'true' v_1 function. This is the essence of the differences between the single- and multi-pass algorithms: the single-pass uses the stencil buffer once to calculate an approximate visibility for all light sources in one pass, whereas the multi-pass uses the stencil buffer once for each light, calculating the true visibility function each time. The multi-pass algorithm is therefore a good approximation to the first two terms of the rendering equation. The only restrictions are, that light sources are only allowed to be points and that, for performance reasons, we can only handle a small number of them.

The soft shadow algorithm will allow a better approximation, although the difference in this formulation seems small:

$$L = v_0 L_e + v_0 T v_1 L_{e_1}$$

where L_{e_1} means (a few) *spherical* light sources with individual radii. This extension to spherical volume lights means that we now consider infinitely many points as light emitters. However the points must be located on the surface of spheres, each sphere conceptually being a single light source. Since a visibility function is evaluated for every point on the light source the result is soft shadows. In practice a single visibility function for the entire light sphere is implemented giving a percentage value representing how much of the conceptual light source is visible from the light-receiving point. An algorithm that implements this is described in chapter 4.

Chapter 4

Soft shadows

The shadows generated by the techniques described in the previous chapter have one flaw in common: they are *hard*. The transition from light to shadow happens over just two pixels: one is fully lit by the light source, the next is in full shadow. This is not due to a flaw in the calculations as such, but is a consequence of the limitation of the techniques: light sources must be points and only direct illumination is calculated. Real world shadows are usually *soft* with a smooth transition from full light to full shadow. This happens for two reasons: indirect illumination and volume light sources, exactly what the previous algorithms could not incorporate. Light sources with a non-zero volume cause soft shadows because there are points where *a part of* the light is visible, this is called the penumbra region and is illustrated in figure 4.1. The area where *nothing* of the light is visible and the geometry is in full shadow is called the umbra region.

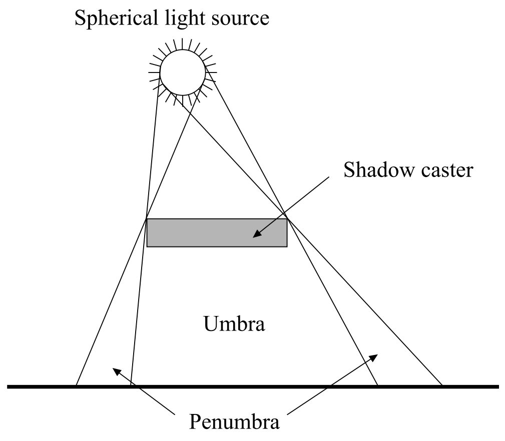

Figure 4.1: Volume light sources produce penumbra regions.

Indirect illumination also tends to ’soften up’ a shadow. This is illustrated in figure 4.2, where the dashed lines represent indirect illumination which bounces off the wall and into the umbra region. Different areas of the umbra will receive different amounts of indirect illumination. In most cases the intensity of the received indirect light will grow weaker when moving from an area near the shadow boundary to an area further away from it, and the shadow will thus appear less hard.

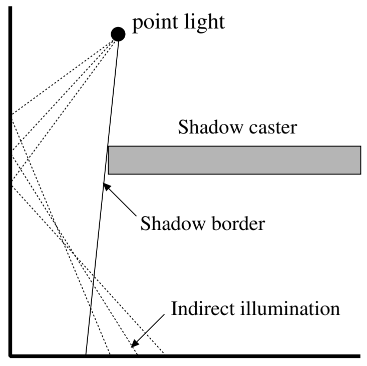

Figure 4.2: Indirect illumination 'softens' an otherwise hard shadow.

Simulating full global illumination, including the indirect illumination shown above, is very hard to do in real-time but, as we shall see in the following, it is possible to render shadows from volume lights to produce soft shadows.

Our work with soft shadows has been based mainly on a series of articles, [AMA02], [AAM03] and [ADMAM03], in which the authors first suggested and later refined and implemented a technique for rendering soft shadows from arbitrary shadow casters onto arbitrary surfaces with real-time performance using pixel shaders. When we began our research only the first of the three papers had been published and, as a result, our own implementation differs from theirs on several key points. On our own, we did come up with some of the same improvements and implementation techniques that they suggested in the later papers, and we interpret this as an indication that the ideas and techniques are sound.

We have also developed several new optimizations which greatly reduce the length of the required pixel shaders, the number of rendering calls made to the graphics driver, and the amount of texture memory required for look-up tables used in the shaders.

In this chapter we first describe the soft shadow technique as it appears in its final incarnation set forth by Akenine-Möller and Assarsson in [ADMAM03]. Then we describe our optimizations and discuss their impact on the overall performance of the technique. Finally, we discuss a number of problems that remain unsolved due to hardware limitations.

4.1 Soft shadows using penumbra wedges

The hard shadow algorithm, as described in section 3.2, uses the stencil buffer to mask out those regions of the screen that are in shadow. The problem with this approach is that the stencil buffer gives a sort of on/off write mask: either a pixel is rendered with full lighting or it is skipped entirely. To render soft shadows we must instead modulate each pixel with a *light intensity factor* that ranges from zero, when the pixel is in full shadow, to one, when it is fully lit. So the main goal of the soft shadow algorithm is to efficiently calculate a screen sized *light intensity buffer*, from now on referred to as the LI buffer, as described in [AMA02]. Once the LI buffer has been calculated the scene is rendered with diffuse and specular lighting, with each pixel being modulated by the corresponding value in the LI buffer. In a final pass, ambient lighting is added to all pixels in the image. In the following we will assume that only one object casts a shadow from the light.

4.1.1 Overview

As described in [AAM03], the soft shadow algorithm is an extension of the standard stencil shadow algorithm, and the calculation of the LI buffer starts by clearing it to one, indicating that all pixels are fully lit by the light. Next, the hard shadow for the object is rendered into the buffer in the usual way, setting the intensity to zero for all pixels that are inside the hard shadow. After this step, if we used the LI buffer to modulate the lighting without any further processing, the result would be hard shadows identical to those produced by using the stencil buffer.

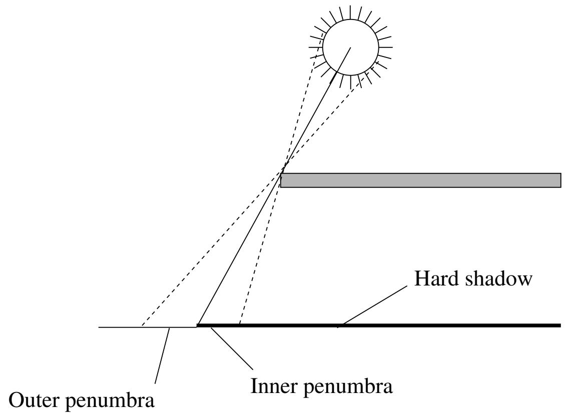

Figure 4.3: The hard shadow splits the penumbra

As seen in figure 4.3 the hard shadow splits the penumbra region into an inner and outer penumbra, and the idea is now to 'soften up' the shadow around this hard shadow edge. In the inner penumbra region we must add light to compensate for the hard shadow algorithm which has set the intensity to zero, even though the pixels can 'see' up to half of the light at the hard shadow edge. Similarly, in the outer penumbra region we must subtract light from those pixels the hard shadow algorithm has deemed fully lit, even though some of them can 'see' as little as half the light. Eventually we would like to end up with a gradient that decreases from 1 on the outside of the shadow to 0 when it enters the umbra region. At the hard shadow edge we should have an intensity of 0.5, as this indicates the border where exactly half the light is visible.

This adjustment of the LI buffer is made using pixel shaders and a special rendering primitive called a *penumbra wedge*, (from now on referred to as just a *wedge*), as described in [AMA02]. A wedge is created for each edge on the shadow silhouette and is a closed piece of geometry constructed to bound the penumbra region generated by that particular edge. By rendering the wedges with a special pixel shader, we are able to adjust the LI buffer as required to soften up the hard edge. We will cover exactly how this is done in a later subsection, but first we describe the wedge and its creation in more detail.

4.1.2 Wedge creation

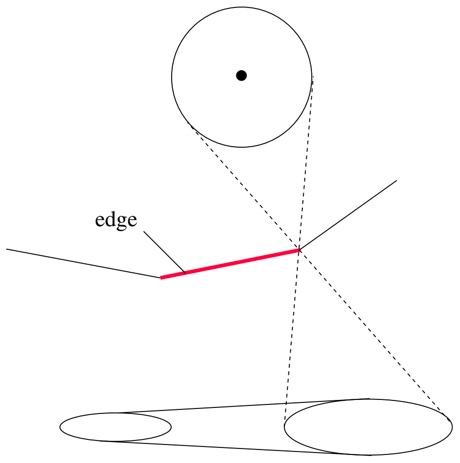

Figure 4.4: Hyperbolic penumbra volume for a spherical light source.

The exact penumbra region for a given edge and light source can be found by sweeping a general 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. It is not feasible to calculate the exact penumbra volume in real-time, nor is it necessary, as we shall see later. It is worth noticing that, assuming a spherical light source, the exact penumbra volume will generally have hyperbolic sides, (see figure 4.4), which also makes it unsuitable for tessellation into triangles, a process that would be required to render the volume. Instead, the wedge is created as a bounding volume for the penumbra region and a robust method for calculating this is presented in [AAM03].

The wedge is generated 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 closest to the center of the light; assume that this is e_0. Then the other one, e_1, is moved towards the center of the light until the distance to the center is the same for both vertices. We denote this new vertex e_1'. The two vertices, e_0 and e_1', define a new edge, e', above the original silhouette edge which will be used as the top of the wedge structure. Note that this new edge is only used for rasterization purposes. The original silhouette edge is still used for the actual visibility calculations in the pixel shader, as we shall see later. Now, because both endpoints of e' are the same distance from the light, the swept cone from e_0 to e_1' will generate planar front and back sides for the wedge and, because we chose to keep the closest vertex fixed, the wedge will fully contain the penumbra region, see figure 4.5.

Rather than actually sweeping a cone over e', the front and back planes of the wedge are calculated by rotating the hard shadow plane around e' so it exactly touches the light source on the other side. The planes for the left and right sides of the wedge are calculated in a similar way by rotating a plane around axes that are perpendicular to e'. The intersection of these four planes, along with a bottom plane some fixed distance away from the light, creates a closed hull: the wedge, illustrated in figure 4.6. Notice how the wedge structure fully contains the actual penumbra volume as swept over the original silhouette edge.

4.1.3 Culling away unnecessary fragments

A wedge is a 3d primitive and when seen in perspective from the side, the rendering of its front facing triangles will trigger the pixel shader for more pixels than those actually residing within the penumbra region on the screen. The greater the angle between the view direction and the light-to-edge direction is, the greater the amount of wasted pixels will be. However, using the stencil buffer it is possible to cull away most of those unnecessary pixels, as described in [ADMAM03]. An important observation here is that, as the penumbra lies on the geometry of the scene, it is only necessary to execute the pixel shader where the wedge volume intersects the scene geometry. To mask out this area, the front faces of the wedge are rendered into the stencil buffer, setting the stencil value to 1 where the pixels pass the z-test. This masks out the grey area seen in the upper-left part of figure 4.7. To actually execute the pixel shader in the intersection region, the back faces

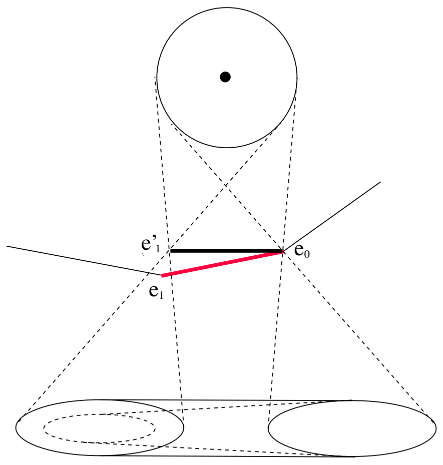

Figure 4.5: Planar front and back sides of penumbra volume.

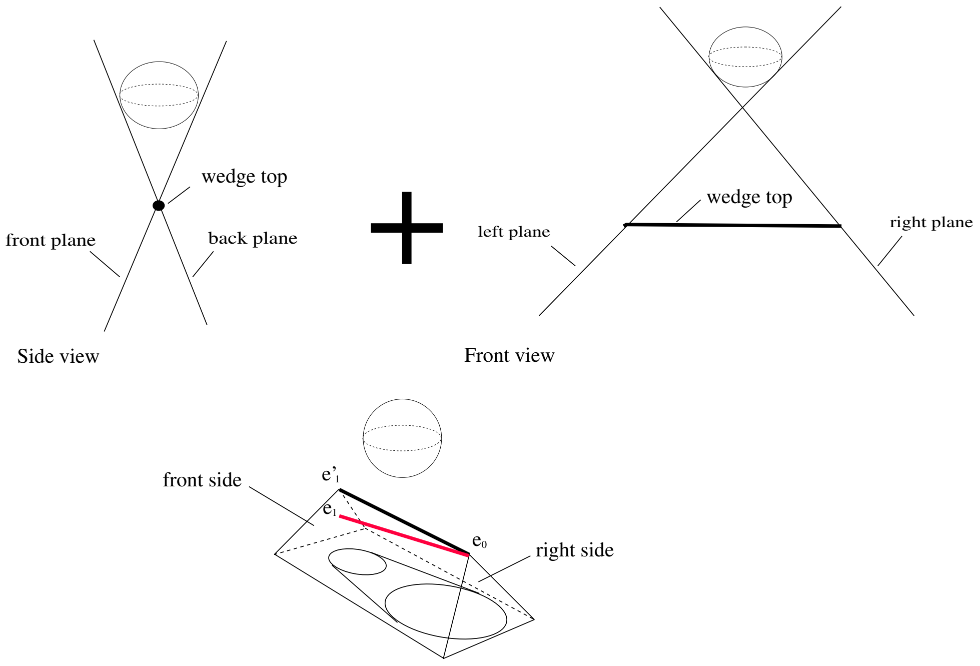

Figure 4.6: Generation of a wedge.

of the wedge are rendered with the z-test set to ’greater’, and the stencil buffer configured to only draw where the stencil value is 1. Rendering the back faces with this z-test is the same as rendering those pixels on the back faces that fail the ordinary z-test, as shown in the upper-right part of figure 4.7, and by enabling the stencil buffer we have effectively masked out the intersection of the two regions. As a result, the pixel shader is only executed where the wedge intersects the geometry, as shown in the bottom part of the figure. Note that as the wedge does not represent the exact penumbra volume, the pixel shader can still be executed for pixels outside the penumbra region and care must be taken to leave such pixels unchanged.

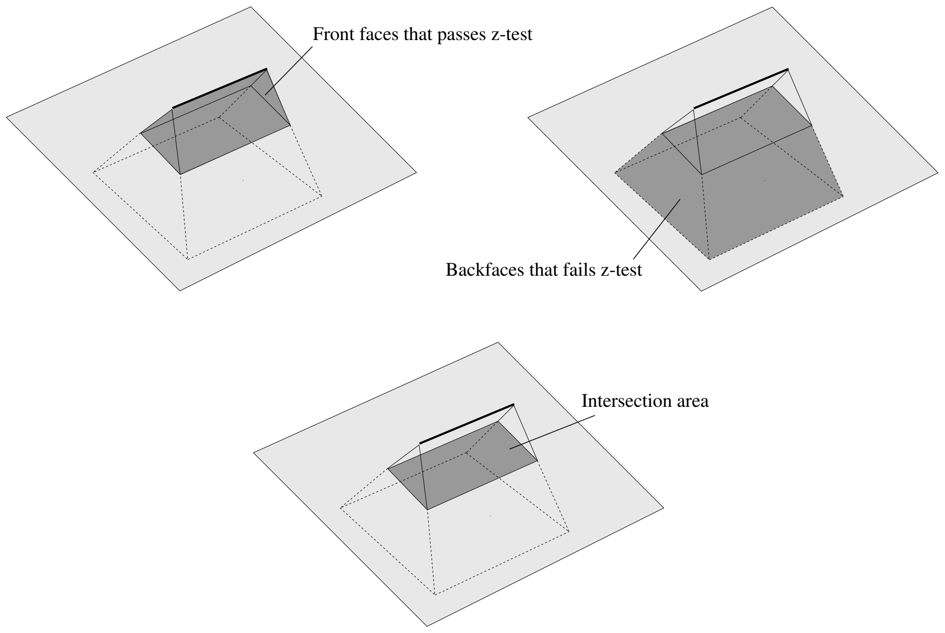

Figure 4.7: Masking out the intersection between a wedge and the scene.

4.1.4 Modifying the LI buffer

To modify the LI buffer we must somehow calculate the light visibility factor for each fragment of the scene geometry that falls within the penumbra region. The first things we need for this calculation are the position of the fragment, the silhouette edge and the light in some common space. As the light position is fixed for all fragments in a particular frame, we can easily upload it as a constant to the pixel shader in any space we want. The positions of the vertices of the edge are also fixed for an entire wedge and can either be calculated in the vertex shader, or, if the wedges are rendered one at a time, be uploaded as constants to the pixel shader in the space we want. Therefore, the challenge lies in finding the position of the scene fragment behind the wedge fragment currently being shaded, in *any* space. For technical and performance reasons we have chosen to do the calculations in view-space, but [ADMAM03] presents a solution where the calculations are done in world-space instead. In view-space, the position of the scene fragment behind a certain fragment on the wedge is formed as the tuple, (x, y, z), where x and y are the same as for the wedge fragment position and z is the depth value stored in the z-buffer. See figure 4.8. Once all these positions are known in view-space, it is possible to calculate a visibility factor for the fragment, as we shall see in the following.

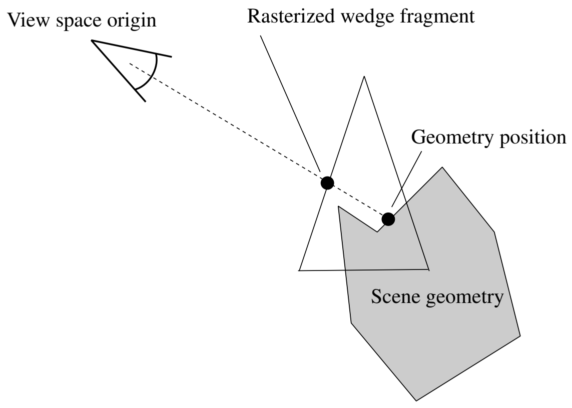

Figure 4.8: Calculating geometry position behind wedge.

To calculate how much of the light is visible from a particular fragment in the penumbra with respect to just a single silhouette edge, we project the hard shadow quad up onto the light, as seen from the fragment. Assuming a spherical light source, this is the same as first projecting the edge onto a circle and then tracing lines from the center of the circle through each projected edge vertex. A *coverage value* can now be defined as the percentage of the light source that is covered by this projection, and it is simply calculated as the area covered by the projection divided by the total area of the light circle. See figure 4.9 for examples of this projection.

Notice that for any fragment within the penumbra region this coverage value will lie between 0 and 0.5, being 0.5 for fragments on the hard shadow edge. For fragments in the outer penumbra region the coverage value defines how much of the light source is hidden by the occluding geometry while, in the inner penumbra region, it actually defines how much of the light is *visible* from the fragment. This means that the coverage value can be used to modify the LI buffer to create the

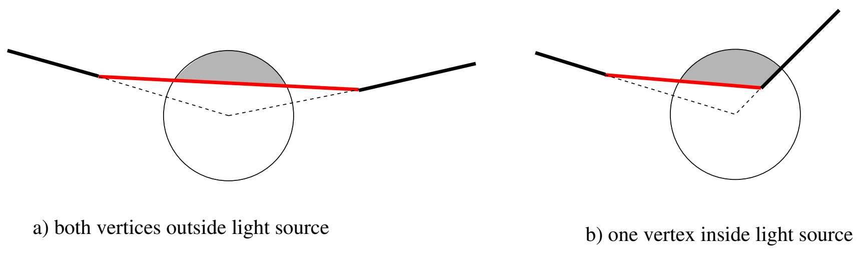
a) both vertices outside light source
b) one vertex inside light source

Figure 4.9: Projecting the hard quad onto the light source.

gradient. In the outer penumbra the coverage value defines how much light that needs to be subtracted and, in the inner penumbra, it defines how much light to add. See figure 4.10.

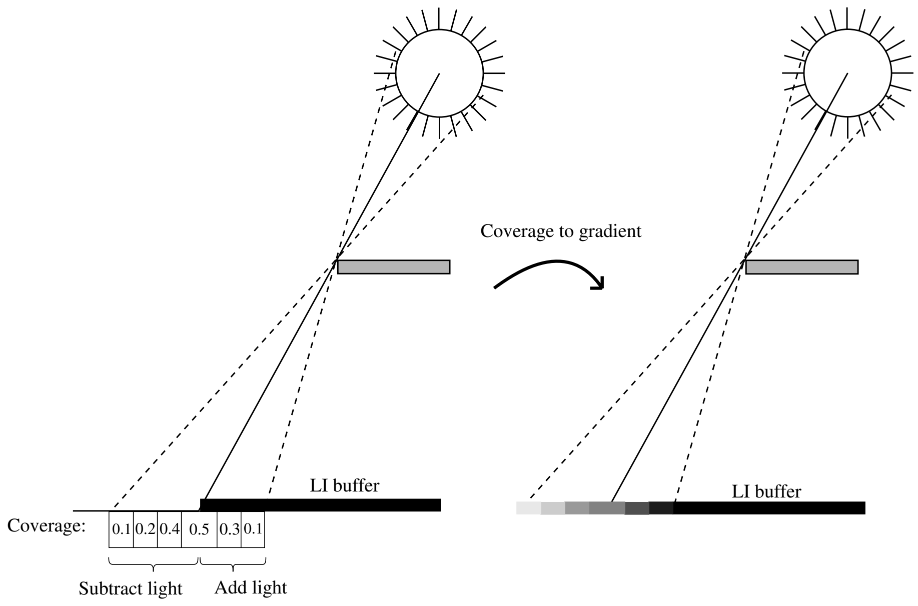

Figure 4.10: Modifying the LI buffer based on coverage.

4.1.5 Summing up coverage contributions

Until now, we have only considered a single silhouette edge when calculating the coverage value for a fragment. In reality, the silhouette form loops, (see [Ass03] pp. 133–135), and, to properly calculate the final coverage for a fragment, it is necessary to project the entire silhouette onto the light source. This is unfortunate since, as described earlier, the pixel shader cannot access any other information than what is given to it through constant registers and the rasterizer. This makes it impossible for the pixel shader to have anything more than local knowledge of the silhouette. Fortunately we can use Green's theorem, as described in [AAM03], to calculate the final coverage as a sum of coverage contributions, evaluated at one silhouette edge at a time. To add or subtract each contribution, when evaluating Green's theorem, is based on the fragments position in the penumbra, i.e. whether light should be added or subtracted as described above. As the penumbra regions for neighboring silhouette edges will overlap in an area around the shared edge vertex, the pixel shader will be executed multiple times for each fragment in that part of the penumbra region, (one time for each edge). As a result, after all the wedges have been rasterized the final coverage value is available as the sum of the contributions. Refer to figure 4.11 for two examples of how to calculate the final coverage with Green's theorem. In the first example, one edge is front facing to the fragment while another is back facing. This means that the fragment is in the outer penumbra region of the first edge and in the inner penumbra region for the other. Coverage values for the outer penumbra are added to the total coverage value, while coverage values for the inner penumbra are subtracted, (since the coverage value for fragments in the inner penumbra defines how much of the light is visible instead of how much is covered). In the second example, both edges are front facing to the fragment, so both are added to the final coverage value.

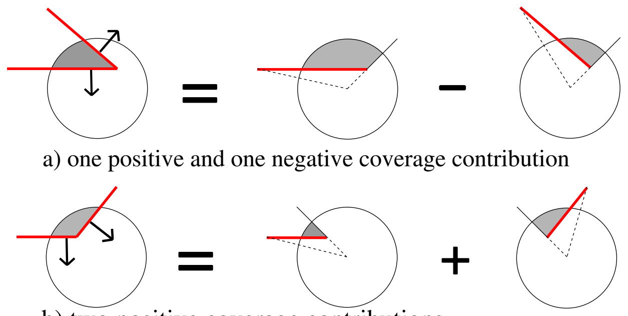
a) one positive and one negative coverage contribution
b) two positive coverage contributions

Figure 4.11: Calculating coverage with Green's theorem.

4.1.6 Summary

To summarize, the steps for creating the LI buffer are:

1. Clear the LI buffer to one, indicating that every pixel is fully lit.

2. Render the hard shadow into the LI buffer, setting the light intensity to zero for pixels inside the hard shadow.
3. For each silhouette edge in the object, create its wedge geometry.
4. Render the wedges with a special pixel shader, one at a time, and mask out the intersection area between the scene geometry and the wedge to minimize the amount of rendered fragments outside the penumbra region.
5. For each wedge fragment rendered, (pixel shader steps):
    (a) Calculate the view-space position of the scene fragment behind the wedge.
    (b) Determine if this fragment is in the inner or outer penumbra region.
    (c) Project the silhouette edge for the wedge up onto the light source, as seen from the fragment.
    (d) Calculate a coverage value from this projection.
    (e) Based on the position of the fragment, either add or subtract the coverage value from the LI buffer.

The interesting steps in the pixel shader are those where the edge is projected into the light source to calculate the coverage value, and this is also where the majority of the pixel shader instructions are spent. We will not go into any further details on how Akenine-Möller and Assarsson choose to implement those steps, but the interested reader can refer to their papers for some implementation details. We will instead focus on our own implementation and in the next section, we describe how we have optimized those steps to greatly reduce the amount of pixel shader instructions and the amount of texture memory required for look-up tables.

Due to certain hardware problems and limitations it is not possible to implement the algorithm exactly as specified in the pseudo-code above. Like Akenine-Möller/Assarsson, we have been forced to work around some of these hardware limitations. We refrain from mentioning them in the algorithm outline above as they do not affect the basic idea in the algorithm, and as some of them will be overcome by newer and better hardware. We will discuss the problems in section 4.3.

4.2 Fast coverage calculation for spherical light sources

In this section we describe our novel technique for highly efficient coverage calculation for spherical light sources. In [ADMAM03], a technique for coverage calculation for spherical light sources is presented. The technique relies on clipping the projected silhouette edge to the boundaries of the light source in the pixel shader. The two clipped 2d points can be used as texture coordinates for a lookup in a 4d texture which implements the coverage function. Using our technique we can avoid these clipping operations in the pixel shader and let the texture sampler do the clipping for free. We also reduce the dimension of the coverage function from 4 to 3, which enables us to encode the coverage function in a smaller texture. In [ADMAM03] Akenine-Möller/Assarsson also present coverage calculation techniques for rectangular and even textured rectangular light-sources. Our new technique cannot be used in these cases since it is a fundamental requirement that light-sources are spherical, but for most applications this is a reasonable limitation.

4.2.1 Unit sphere space

For our coverage calculation technique to work we need to work in a space where the light source is not just spherical but actually a unit sphere. This is done by applying a change-of-basis matrix, (from now on referred to as the CBM), to the light position, the silhouette edge and the fragment position. This CBM simply scales the three coordinate axes in R_3 with the inverse light radius and looks like this:

$$CBM = \begin{bmatrix} \frac{1}{R} & 0 & 0 \\ 0 & \frac{1}{R} & 0 \\ 0 & 0 & \frac{1}{R} \end{bmatrix} \tag{4.1}$$

Because the CBM only has entries on the diagonal, it can be reduced into a scaling of the vector it is applied to, like this:

$$CBM * V = (\frac{1}{R}, \frac{1}{R}, \frac{1}{R})V = \frac{1}{R}V \tag{4.2}$$

This scaling can be made in a single shader instruction per point that needs to be transformed. And as described above only the fragment position needs to be calculated in the pixel shader so the cost of working in unit sphere space is just a single pixel shader instruction. In the following we will assume that everything has been transformed into unit sphere space.

4.2.2 Coverage calculation

To calculate the coverage value for a specific fragment we first calculate the plane through the silhouette edge and the fragment. This can be thought of as a 'tilted hard shadow plane' and we refer to it in the following as the $geoPlane$. Next, the signed distance from the light source to the $geoPlane$ is calculated, call this distance for d_0. If $d_0 \in [-1..1]$ then the plane intersects the light source, and the fragment is actually within the penumbra region and not just inside the wedge. The sign of d_0 also determines whether the fragment is in the inner or outer penumbra region. If d_0 is positive the fragment is in the outer penumbra, if it is negative it is in the inner penumbra. If d_0 is exactly zero the fragment lies on the hard shadow edge. This classification determines whether we should add or subtract light to the LI buffer[1]. Then, the light source is projected onto the $geoPlane$ to a point we will call the $basePoint$. Now we can define the $lightPlane$ as having origin at the $basePoint$ and a normal in the direction from the $basePoint$ towards the fragment. The intersection between the light source and the $lightPlane$ gives us the 2d circle onto which we want to project the silhouette edge, as described in earlier sections. Refer to figure 4.12 for a 2d side view of the planes and points described above.

Note that if we project the infinite line on which the silhouette edge lies onto the light, then the $basePoint$ lies somewhere on this projection. In addition, when we project the two ends of the silhouette edge onto the light they will also lie on the line, either on the same side of the $basePoint$ or one point on each side. Now we calculate the distances from the $basePoint$ to each of the two projected ends and denote them d_1 and d_2. We can calculate the coverage value given four pieces of information: the absolute value of d_0, the value of d_1 and d_2, and whether the two projected ends are on the same side of the $basePoint$ or not. Figure 4.13 illustrates this. Note that due to symmetry we are able to rotate the original coverage area around the circle center as well as mirror it around the two coordinate axes without changing the actual coverage value. As a result of this, we can represent any coverage area as one of the two forms in the figure. Also note that even though the figure might indicate it, it is not a requirement that both or even any of the projected points are actually within the light source.

Each of the three parameters has valid values only when they are within the range $[0..1]$. When d_0 reaches 1 the coverage value will be 0, no matter what the distances to the end points are, and the same can be said for any value greater than 1. Consequently it is safe to simply clamp d_0 to be at maximum 1. To calculate the coverage value the two projected ends are first clipped against the light source. As the light is a unit sphere, the maximum vertical distance from the $basePoint$ to the circle edge is 1, and this only occurs when d_0 is equal to zero. For all other values of d_0, the vertical distance from the $basePoint$ to the circle edge will be less than one. So also the two other parameters, d_1 and d_2, can be clamped to be within the range $[0..1]$.

[1]In reality, due to hardware issues, another technique is currently used for this classification. We will cover this in a later section.

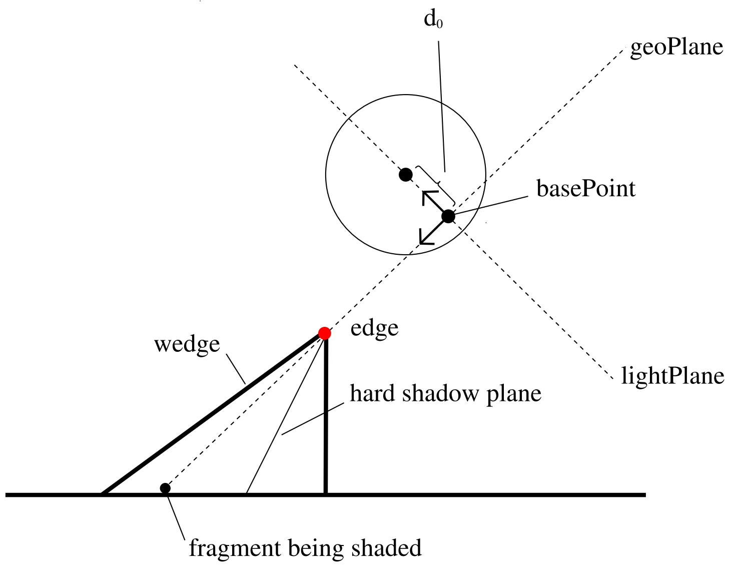

Figure 4.12: The geoPlane, lightPlane and basePoint.

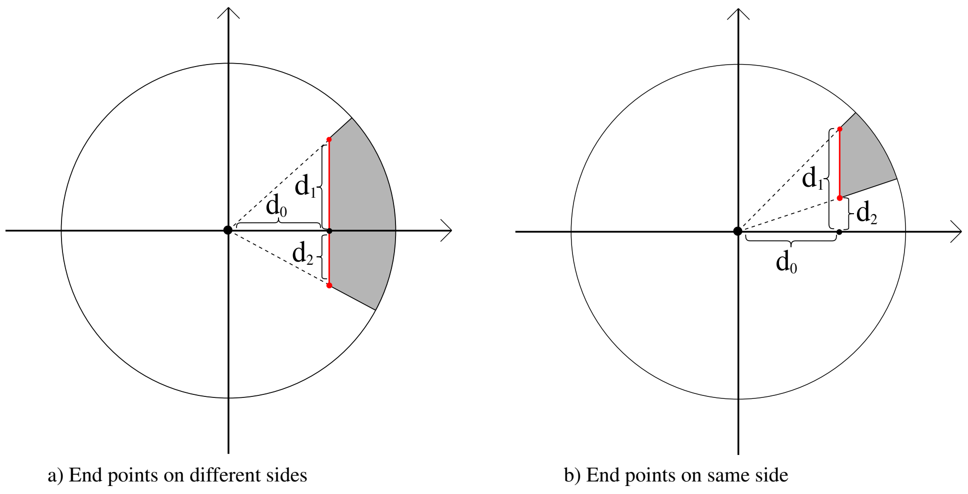

Figure 4.13: Parameterized coverage calculations.

Assuming that each parameter lies within its valid range, we can now encode the coverage function into two 3d textures: one texture for the case where the points are on the same side of the $basePoint$; and another for the case where they are on different sides.

The clamping of the three texture coordinate components *could* be done in the pixel shader before sampling the look-up texture but that is not necessary. Instead we use the built-in clamping option in the texture sampler which performs the desired clamping for free.

To determine which of the two coverage maps the pixel shader should sample, it calculates the dot product between the two vectors going from the $basePoint$ to each of the two projected edge points. If the two points are on the same side of the $basePoint$, the angle between these two vectors will be zero and therefore the dot product will be positive. If, on the other hand, the two points are on different sides of the $basePoint$, the angle will be 180 degrees and the dot product negative. If one or both of the edge end points should be projected exactly to the $basePoint$ this will result in one or both of the vectors being the zero vector and, as a result, the dot product will be zero. In that case, either of the two coverage textures can be used, so it does not really matter much which branch the pixel shader takes. In our implementation a dot product of zero would use the coverage texture for points on different sides of the $basePoint$.

4.2.3 Optimization summary

In [ADMAM03] Akenine-Möller/Assarsson report that their latest hand-optimized version of the pixel shader for spherical light sources requires 59 arithmetic pixel shader instructions and 4 samples into look-up textures.

Our version uses just 40 arithmetic pixel shader instructions and 3 texture samples on graphics cards without dynamic branching in the pixel shaders. On cards with dynamic branching the amount of texture samples are just 2. With dynamic branching the pixel shader can simply sample the correct texture, but if there is no dynamic branching the pixel shader must sample both textures, and choose the correct value afterwards.

As mentioned earlier our pixel shaders are all written in CG and we have not attempted to optimize the output from the compiler by hand. Consequently, it might be possible to save a few instructions this way to improve performance further.

In [AAM03] Akenine-Möller and Assarsson presents a method for parameterizing the coverage calculation as a four-dimensional function using two 2d points as indices, so each index is in the form (x_1, y_1, x_2, y_2). As 4d textures are not supported by any current graphics card, the function is encoded into a 2d texture where (x_1, y_1) determines which region, (or sub texture, as they call it), to sample in, and (x_2, y_2) looks up the actual coverage value from that region. They report that a discretization of the light source into 32×32 regions provides acceptable precision in the coverage function and, as a result, the size of their look-up texture is 1024×1024 pixels. Storing each coverage value as a 16-bit floating-point value, the amount of texture memory required for their look-up texture is thus 2MB. In addition to this 2d coverage texture they also use a cube map that implements the function $atan2(x, y)$. They do not report the size of this cube map but whatever it is it must be added to the total texture memory cost.

Using our new technique the coverage function is reduced from a four-dimensional into a three-dimensional function and, with a similar discretization of the light source, our two 3d textures uses just 65KB each (32*32*32*16bit for each texture). The small size means that the textures can easily be created at load time and need not be precalculated and stored in a file.

4.3 Problems with the soft shadow technique

Several problems have yet to be solved before this technique for creating soft shadows can be applied to a general game scene with real-time performance. Some of these problems are related to limitations in current hardware, and will likely disappear within the next few generations of graphics cards. Other problems are related to the technique itself and changes to the algorithm are required to overcome them. In this section we discuss each identified problem along with the temporary solution or work around we have applied to implement the technique on today's hardware.

4.3.1 Access to the z-buffer

One of the first steps in the pixel shader is to calculate the view space position of the scene fragment behind the wedge fragment that is currently being rasterized. The x and y components of this position is copied from the view-space position of the wedge fragment being rasterized, while the z component is the value in the z-buffer at the current pixel location.

The problem is that the z-buffer value at the current pixel location isn't available through the pixel shader API. At the moment, the only feasible solution for using the z-buffer's data in a pixel shader is to do an extra pass over the entire scene, using shaders to output the depth of each fragment to a texture. The texture can then be sampled from the pixel shader that needs the depth information. Some GPUs can render to multiple render targets at the same time and in such cases the 'extra' z-buffer can be rendered during the normal rendering of the scene, thus avoiding the extra pass. However, a screen-sized texture is still required and it uses a significant amount of texture memory. In addition, the extra fill-rate used to fill the depth texture has a negative impact on the overall performance of the application.

No specification exists for how depth information must be stored within a z-buffer surface and, as a result, the different graphics card manufacturers use all sorts of tricks to compress and pack the z-buffer to achieve maximum performance. This also means that it is expensive to access the z-buffer data, as is evident in for example DirectX where it gets increasingly difficult to lock and access the z-buffer with each new version. Still a read-only access to the depth value of the current pixel might be available from pixel shaders in a future generation of graphics card[2]. When, or if, this happens it can be used to optimize the soft shadow algorithm and save some much needed bandwidth for the rendering of the wedges.

4.3.2 Limited blending

With the latest generation of graphics cards the concept of floating point textures were introduced. These allows the application programmer to create textures where each channnel contains a 32-bit signed floating point value. A texture with just a single float channel would be perfect for the LI buffer in the soft shadow algorithm since the light visibility factor is just a single float value in the range 0 to 1. To update the LI buffer we would need to be able to add or subtract the contributions from the different wedges. Presently the only way to let the output from a pixel shader depend on the previous value in the render target is to use the fixed-function blending operation. However the blending capabilities of current hardware is quite limited. It *is* possible to both add and subtract values from a render target through the blending operation, but not without changing a render state in the driver. In other words, it is not possible to decide whether an addition or subtraction is to be performed for each seperate pixel. A solution to this might be to set up the card to always do addition and then output negative values for those pixels where a subtraction is required. Unfortunately this is not feasible as the output from a pixel shader is automatically clamped to lie within the range $[0..1]$.

To overcome this problem one could use a texture format with two channels, for example two channels with unsigned 16-bit floating-point values is also possible. All positive visibility contributions could then be accumulated, using normal additive blending, in the first channel while all negative contributions could be added together in the second channel. The final light visibility factor could then be calculated in a pixel shader as the first channel minus the second channel. Us-

[2]Conversation with Richard Huddy from ATI, ShaderDay 2003 at DTU

ing this approach the light visibility factor would still have 16 bit precision which is enough to avoid 'banding' in the gradient.

Unfortunately, on current graphics cards blending is not supported at all on render targets with more than 8 bits per color channel, a limitation which effectively means that the new floating-point textures cannot be used for the LI buffer.

Due to this we have used a standard four channel ARGB texture format with 8 bits per channel as the LI buffer in our current implementation. Two of the channels are used to hold the integer contributions from the hard shadow pass, much like a stencil buffer would, and with 8 bits per channel, we can thus have 256 overlapping hard shadow volumes. As suggested above the other two channels are used to hold the positive and negative gradient contributions from the wedge pass. A certain number of the bits in each penumbra channel must be reserved for overlaps and in our implementation, we use 3 bits for overlap and 5 bits for each coverage contribution. Using just 5 bits for the coverage values means that only 32 different shadows shades are available, which can lead to visible 'banding' effects when viewing the penumbra region of a shadow up close. Still, using 5 bits for the gradient offers a decent image quality. For complex shadow casters more than 8 overlapping wedges can occur and more bits will have to be reserved for the carry, leaving even less for the actual gradient. See figure 4.14 for a close up section of the penumbra gradient using 8, 5 and 3 bits.

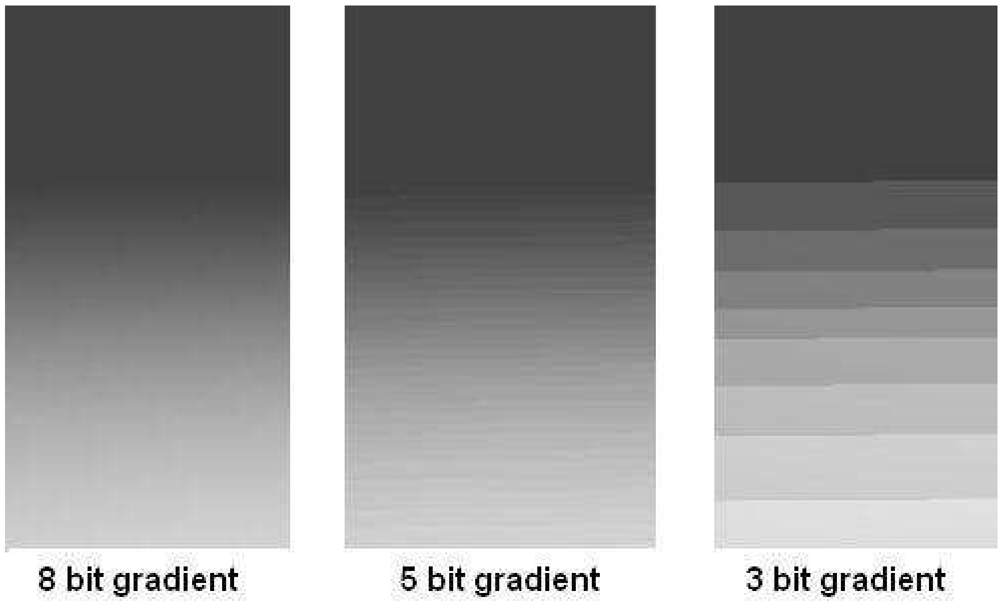

Figure 4.14: Banding artifacts

In [ADMAM03], a method is briefly mentioned by which the authors split each coverage contribution into multiple channels, obtaining a 12 bit gradient with 16 possible overlaps at the expense of extra pixel shader instructions and additional bandwidth use. We have chosen not to use this technique as we will instead await the next generations of graphics cards which will hopefully allow for blending to floating point render targets. Even better, the frame buffer blending might become a truly customizable component like the vertex and pixel shaders already have[3]. Once better blending is available, the soft shadow algorithm can easily be changed into using a single channel 32-bit float texture, as outlined in the beginning of this subsection, allowing for high precision gradients with a virtually unlimited amount of overlapping edges.

4.3.3 Splitting the wedges in two halves

As described above, the classification of a fragment into the inner or outer penumbra region decides whether light should be added or subtracted from the LI buffer. For every pixel this classification must match the hard shadow classification performed in the hard shadow pass. If for example, a pixel is classified as being in the outer penumbra in the wedge pixel shader, but the same pixel has had its light visibility set to zero during the hard shadow pass, then light will be subtracted from a pixel whose LI value is already zero. Similarly, it could happen that light is added to a pixel the hard shadow pass has not marked as being in shadow. Such errors result in very visible artifacts where pixels appear overly bright or overly dark within the penumbra region.

It ought to be impossible for such an error to occur if the classification of each fragment was made in the same way during both the hard shadow and the wedge pass, but in reality it *does* occur for pixels at or very near the hard shadow plane. The reason is that the hard shadow pass is made by rendering a normal shadow volume into the LI buffer. Each triangle in the hard shadow hull is sent to the rasterizer, which discretizes the otherwise mathematically continuous surface into a finite number of fragments, each with integer coordinates. This means that the hard shadow edge, as rendered into the LI buffer, is not the mathematical correct intersection between the shadow volume and the underlying geometry. As a result, it is impossible to mathematically classify a certain fragment as being inside or outside the hard shadow from the pixel shader.

A solution is presented for this problem in [ADMAM03] where each wedge is split into two halves, one for each of the inner and outer penumbra regions. By 'embedding' the hard shadow hull in the planes that split the wedges, it is now

[3]According to Richard Huddy from ATI a mechanism for implementing custom frame buffer blending operations will appear in future generations of graphics cards.

possible to ensure that, when rendering for example the inner half of each wedge, the pixel shader will not be run for pixels near the hard shadow edge in the LI buffer that hasn't been set to zero. Similarly, when rendering the outer wedges the pixel shader will only subtract light from pixels that are left fully lit by the hard shadow pass.

The intersection between each wedge half and the scene geometry can be stenciled out to cull away unnecessary pixels exactly as described earlier, and the classification step is now no longer necessary. Instead, a constant can be uploaded to the pixel shader that determines whether to add or subtract light for all the rendered fragments.

4.3.4 Rendering one wedge at a time

Perhaps the most severe problem with the algorithm in its current form is that the wedges must be processed and rendered one at a time. This is a consequence of the problem described above where each wedge is split into two halves to avoid rendering artifacts near the hard shadow edge. This solution removed the classification step, and the algorithm now relies solely on the stencil and depth buffer to determine which pixels the shader should add light or subtract light from. However, as seen in figure 4.15, this can lead to problems when wedges overlap.

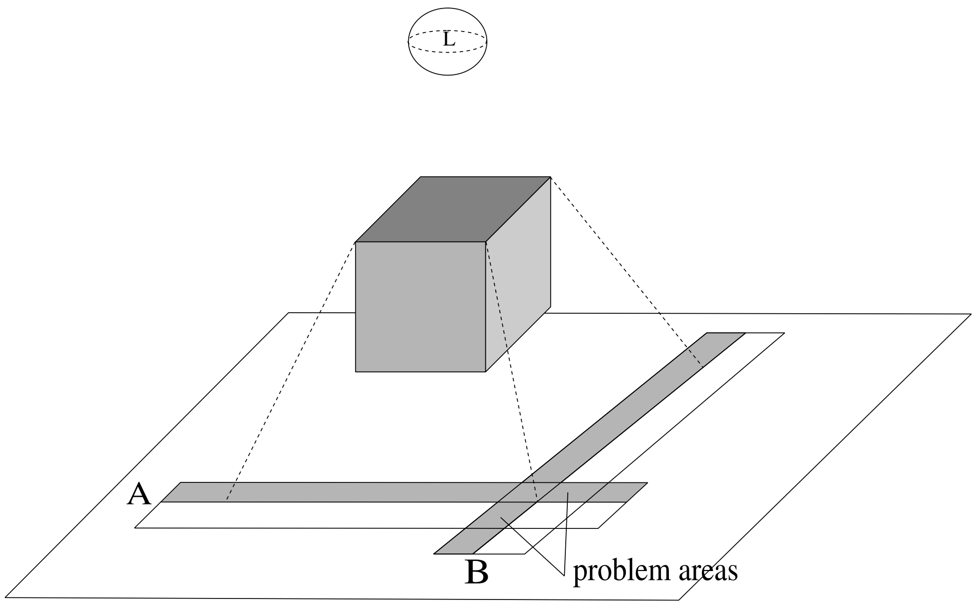

Figure 4.15: Problem with overlapping wedges.

Figure 4.15 shows the intersection between two whole wedges, A and B, and the scene geometry. If both inner wedge halves were to be rendered at the same time, the first step would be to stencil out the intersection between them and the scene, this area is marked out in gray on the figure. Now, when rendering the inner wedge half of wedge A, the pixel shader will also be executed for pixels in A's *outer* penumbra region that intersects with B's inner wedge half, (marked as the left problem area in the figure). Since these pixels are actually within the penumbra region for wedge A, the pixel shader will calculate a non-zero coverage value for them. Moreover, since we are currently rendering inner wedge halves, a constant will have been uploaded to ensure that this coverage value is *added* to the LI buffer. As a result, light is added where in fact it should have been subtracted. For other wedge configurations than the one shown in the figure a similar problem can be identified when rendering multiple outer wedge halves at the same time.

Therefore the wedges must be rendered one at a time. In fact, there are multiple rendering steps involved in rendering just a single wedge. To stencil out the intersection each wedge half is first rendered to the stencil buffer only and then rendered once more with the pixel shader enabled. So a total of four render calls are made for each wedge.

This is a very serious problem because, as described in [Wlo03], there is a small but still significant CPU cost to each draw call made to the graphics driver. It is reported that a 1GHz CPU can issue just around 25000[4] draw calls at 100% CPU usage. With four draw calls per wedge and a desired real-time frame rate of 30FPS that gives us about 200 wedges, (or silhouette edges), per frame per gigahertz of the CPU. In general, game scenes generate many times this number of silhouette edges[5], even with just a few light sources visible, and the CPU thus becomes the major bottleneck in the algorithm.

This is unsatisfactory for several reasons. First, the speed of current CPUs are magnitudes too slow and it is therefore unlikely that increases in CPU speeds will overcome the problem anytime soon. Moreover, as GPUs currently evolve faster than CPUs, the graphics detail, and thus the number of silhouette edges, will likely increase faster than the CPU speed thereby making the problem worse over time. Secondly, even if the CPU power was available it is unfortunate to spend a large amount of CPU power on just issuing draw calls. In a game, the CPU is needed for many other things such as visibility determination, AI, sound, collision detection and game scripts.

From the discussion above we conclude that if the soft shadow technique is to be used in a real game an algorithmic change that allows a large number of wedges to be rendered at the same time, with a single draw call is necessary. We have begun work in this field and have come up with a technique that reduces the number

[4]The exact number might vary slightly for different graphics cards and drivers.

[5]We easily reach 5000 silhouette edges or more for simple scenes in our game engine.

of draw calls from one per wedge to one per silhouette loop. Unfortunately, our current implementation of this new technique works only for a limited group of objects, namely those with convex silhouette loops. We describe this technique in more detail in section 4.4 and present benchmark results in section 4.5 to back these claims.

4.3.5 Fill-rate problems

For simple scenes, where the number of draw calls is low, we have identified another bottleneck, this time on the GPU. Our observation is that performance is quite dependent on screen resolution. From this we conclude that the algorithm is either limited by the amount of pixel shader instructions executed or on the amount of fill-rate used. We have already put forth a solution to reduce the number of pixel shader instructions, (see section 4.2), so there is not much we can do regarding the pixel shader, except await new generations of graphics cards with faster and better pixel shader components.

Regarding the fill-rate we have identified a problem caused by hardware limitations, which our current implementation suffers from. As described above we use a 32-bit four-channel ARGB texture as our LI buffer in our current implementation. Two of the channels are used during the hard shadow pass while the other two are used in the wedge pass. None of the passes reads or writes to channels used by the other pass. The final light visibility factor is calculated from all four channels but the calculation is performed in yet another pixel shader, which only reads from the LI buffer.

This means that we could split the LI buffer up into two 16-bit textures with two channels each, one texture for each of the two passes. If this was possible, we could reduce the fill-rate from 32 bits to 16 bits for each rendered fragment in both the hard shadow and wedge pass, in effect cutting the total fill-rate in half. The price for this optimization would be that the shader calculating the final light visibility factor would have to sample two LI buffer textures instead of one, but the sum of the sampled data would still be 32 bits per fragment.

In fact, a suitable texture format exists on current graphics hardware but unfortunately it cannot be used as render target with support for blending, which is a requirement to implement our LI buffer. Again, in future generations of graphics cards we expect to be able to customize the blending step from within the pixel shader allowing us to implement the LI buffer as a single-channel floating point surface with either 16 or 32 bits precision thus reducing the required fill-rate.

4.4 The per-loop algorithm

In this section we outline our new technique which allows us to batch together the rendering of multiple wedges into a single draw call, thus overcoming the major CPU bottleneck we identified in section 4.3.4. The new technique also allows us to use the original non-split wedges, described in section 4.1.2, without sacrificing a pixel-precise classification of whether a fragment is located in the inner or outer penumbra region. This reduces fill-rate, vertex transformations, and the size of the data transferred over the AGP bus each frame, (in the case of animated geometry or lights). Unfortunately, the method does not work for general shadow casters.

The original version of the soft shadow algorithm performs two different classifications regarding each fragment: whether it is in hard shadow or not, this is determined in the hard shadow pass; and whether it is in the inner or outer penumbra region, which is decided in the pixel shader used for the wedge pass. As explained in section 4.3.3, these two classifications must match exactly or visible artifacts will occur. Splitting the wedges in halves solves the problem, but consequently each wedge *must* be rendered separately.

Another solution to the problem is to perform both classification steps in the same place, namely in the wedge pixel shader, using the same data to make the classifications. That way we can make sure the two classifications will match.

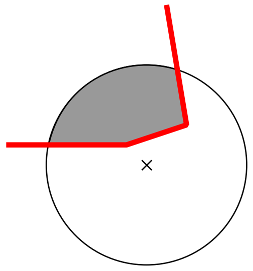

Figure 4.16: Silhouette with three edges projected onto the light plane.

In figure 4.16 we see three connected silhouette edges, projected onto the light plane as described in section 4.2.2. The determination of whether a fragment is in the inner or outer penumbra region, with respect to a single edge, is determined by the center of the sphere being in front of the single projected edge or not, as explained above. The determination of whether a fragment is inside the *hard shadow* area or not is determined by whether the center of the sphere is covered by the projection of all the connected silhouette edges onto the light. Figure 4.16 therefore shows the projection for a fragment *outside* hard shadow.

We see that while the classification of a fragment being in the inner or outer penumbra region is an 'edge local' property, the classification of being inside or outside hard shadow is not. However hard shadow is not a global property either. For a particular light source a fragment can be covered by shadows from many different objects. Being inside the hard shadow region from one shadow is independent of other shadows. The silhouette edges from a shadow-casting object form loops, (see [Ass03] pp. 133–135), and each loop can be thought of as a single shadow, each with a hard shadow region. Hard shadow is therefore a 'silhouette loop local' property.

If we assume that the silhouette loops form *convex* shapes, when projected onto the light plane, then a fragment is in hard shadow if, and only if, it is in the inner penumbra region for all the edges of the loop. We can therefore let the wedge pixel shader use a channel in the render-target for 'hard shadow data'. Concretely, we can let the shader add 1 to this channel if the fragment is in the *outer* penumbra region for a wedge, thus flagging that the fragment cannot possibly be in hard shadow. After rendering all the wedges of the silhouette loop, a subsequent pass can then determine whether a fragment is in hard shadow simply by checking if the 'hard shadow data' channel is still zero, the value it is cleared to. Besides this hard shadow data value, the wedge pixel shader also outputs a coverage value, calculated as in the original algorithm, that it either adds or subtracts from the LI buffer, based on its classification. Using this approach, the wedge pixel shader effectively performs both classification steps and no rendering artifacts appear near the hard shadow edge.

The considerations above form the basis of our new algorithm, in which we reduce the number of render calls made to the graphics driver by batching together all wedges for each silhouette loop. Since the number of edges in a silhouette loop is at least 3, (and often much higher), this addresses the CPU bottleneck of the original algorithm, as identified in section 4.3.4.

We now describe some of the details that are necessary for an actual implementation of our idea. First of all, the calculation of the final LI value used to modulate the light is a bit more complex than in the original algorithm. The subsequent pass mentioned above that checks for the hard shadow property must be implemented in a pixel shader which we will call the *coverageTransfer* pixel shader. This pixel shader calculates the *loop local* LI value from the output of the wedge pixel shader, and *transfers* it to the final LI-buffer. In our implementation, the loop local LI values are simply added together to form the final LI value[6].

The coverageTransfer pixel shader has two main cases: either a fragment is in the penumbra region or it is not. This can be determined by checking if the coverage value, the difference between the positive and negative coverage contributions from the wedge pixel shader, is non-zero. If it is non-zero, the fragment must be

[6]As described in [ADMAM03] section 5.1, this is not entirely correct and it would be possible to use the suggested average value instead.

within the penumbra region since the wedge pixel shader has only been run on fragments inside this region. Now we check whether the fragment is in the hard shadow region, which can be done by determining whether the hard shadow data value is still zero, as explained above. If the fragment is in hard shadow then 1 must be added to the coverage value, (in the original algorithm this is performed in the hardshadow pass). If we are outside the penumbra region then the coverage value is calculated purely based on the hard shadow pass, which must still be performed in our new algorithm to shadow the fragments in the umbra region. Section 6.4 shows the CG code for the coverageTransfer pixel shader.

From the account above we see that the wedge pixel shader can no longer render directly into the final LI-buffer. Instead we use a buffer called the *Loop-Buffer* to hold the loop local LI values. As with the LI buffer, it must currently be implemented with a 4-channel 32-bit ARGB surface since we need to be able to blend (add) values to it. The first channel is used for the hard shadow pass to flag those fragments that are in the umbra region, exactly as in the stencil buffer algorithm. The second channel is used for the hard shadow data flag and the last two channels are used for the positive and negative coverage contributions from the wedges. The LoopBuffer is set as render-target when rendering both the hard shadow pass and the wedges, and is used as a texture when running the coverage-Transfer pixel shader.

To summarize the above, a step-by-step description of the per-loop algorithm is given here:

1. Clear final LI-buffer.
2. For each silhouette loop L:
   (a) Clear the LoopBuffer and set it as render-target.
   (b) Render the hard shadow for L.
   (c) Render the wedges in L.
   (d) Set the final LI-buffer as render-target and the LoopBuffer as texture.
   (e) Render a screen-sized quad with the coverageTransfer pixel shader.
3. Use LI-buffer to modulate lighting as usual.

Unfortunately, there are several problems with this algorithm. Firstly, we have assumed that the projection of silhouette loops is convex, and this is generally not the case. Non-convex loops make the determination of being inside or outside the hard shadow much harder and currently we have no solution for this problem. Secondly, the rendering of a silhouette loop is followed by the coverageTransfer pass, which is expensive since it executes a pixel shader for every pixel on the screen. In addition, the LoopBuffer, a screen-sized render-target, has to be cleared for each silhouette loop. Finally, the number of render calls can still be high, as complex models can have many silhouette loops.

The many extra pixel shader executions and the clear operation for each silhouette loop become the performance bottleneck of the new algorithm, and, because of this, the per-loop algorithm is quite slow, generally slower than the original algorithm. However the per-loop algorithm has an interesting property: it is GPU limited, whereas the original algorithm is CPU limited because of the large number of render calls. As described above, since the graphics cards currently evolves much faster than CPUs, this might be a good tradeoff.

Note also that some optimizations could be implemented to improve the performance of the per-loop algorithm. An example of this is that, it is not necessary to execute the coverageTransfer pixel shader for every pixel on the screen, only for those affected by the rendering of the silhouette loop, which might only be a small fraction of the pixels on the screen.

4.5 Performance analysis

We have implemented both the original and our new per-loop version of the soft shadow algorithm, using our optimized coverage calculation technique for both versions. In this section, we present some performance measurements that show how the bottleneck is indeed found in different places for the two techniques. Four test scenes, as shown in figure B.6, are rendered in two different screen resolutions, (640x480 and 1024x768), and on two different CPUs, (an AMD Athlon 900MHz and an Intel Pentium4 3GHz). An ATI Radeon 9700Pro graphics card was used for all tests. The test have been constructed to gradually increase the number of lights and objects, and consequently the number of wedges. Below, two tables summarize the measured performance in FPS.

1024x768		900MHz		3GHz	
Scene	#wedges	Orig.	Per-loop	Orig.	Per-loop
1	4	71.0	68.0	78.0	73.0
2	64	55.0	42.0	59.5	44.0
3	538	13.0	9.0	19.0	8.5
4	756	8.5	6.5	15.0	6.5

640x480		900MHz		3GHz	
Scene	#wedges	Orig.	Per-loop	Orig.	Per-loop
1	4	175.0	168.0	190.0	179.0
2	64	82.0	103.5	136.5	108.0
3	538	13.0	21.5	35.0	22.0
4	756	8.5	16.0	22.5	16.5

As can be seen from the tables, the original algorithm quickly becomes totally CPU bound on the 900MHz CPU, and a change in resolution has no effect on the performance. This is the case for test scenes 3 and 4.

With enough CPU power available, the original algorithm is instead GPU bound, which is why a similar pattern cannot be found on the 3GHz CPU or in the two simplest scenes on the 900MHz machine. Still, in the lower resolution where the technique is less likely to be GPU bound, performance is predictable for the original algorithm. The drawing of a single wedge requires four passes: two for each wedge half. Therefore, in test scene 4, there are 3024 render calls just for the wedges, and the rendering of the hard shadow and the objects themselves must be added to this number. As explained in section 4.3.4, it is possible to perform somewhere around 25000 render calls per 1GHz CPU, if the CPU is used for nothing else. Assuming this is the case, and rounding the 900MHz machine to 1GHz, we can expect maximum possible frame-rates of 1GHz*25000/3024 = 8.2FPS and 3GHz*25000/3024 = 24.8FPS for the two CPUs respectively, numbers remarkably close to those measured for the original algorithm.

The per-loop algorithm, on the other hand, appears to be totally GPU limited. For all the test scenes, a decrease in resolution results in a large performance gain. In addition, performance is comparable for the two CPUs in both resolutions.

Chapter 5

Shadow management

Rendering a shadow volume is relatively expensive and in an environment with many light sources and where all objects cast shadows, the amount of shadow volumes in the scene quickly grows to very large numbers. Consequently it is very important to be able to cull away the shadow volumes that are outside the viewing frustum, and thus do not affect the final image, *before* they are processed by the graphics card.

In this chapter we first describe a culling technique called 'frustum culling', which rejects objects outside the viewing frustum based on their bounding volumes. We then present a method for calculating a bounding volume for VS shadow volumes. Next, a data structure called a 'scene tree' is presented. The scene tree is based upon a quad-tree but has been modified so that it is able to handle dynamic scenes with moving objects in an efficient way. We describe how to use the scene tree to accelerate intersection queries between the objects in the scene and various volumes such as a frustum, sphere or box. Finally, we conclude the chapter by presenting an optimized version of the multi-pass stencil shadow, using the culling techniques to speed up the overall rendering of the scene.

5.1 Frustum culling

Along with the near and far clipping plane, the camera defines a frustum shaped volume that encloses all visible geometry for a particular frame. Only those triangles that are fully contained in or intersect this volume are visible on the screen and must be rendered by the graphics card, the rest can be culled away and their processing skipped.

As turns out, it is too expensive to perform this culling check for each individual triangle because in the time the CPU spends on culling away a triangle the GPU can easily manage to render it. Instead, a bounding volume is calculated for

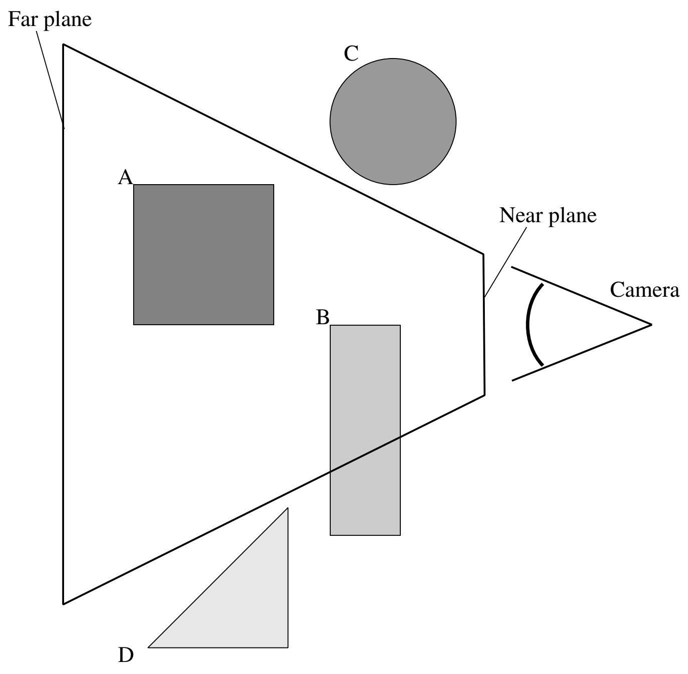

Figure 5.1: The viewing frustum

a group of triangles that is likely to be relatively close to each other. In [Wlo03], Wloka reports that anno 2003 the maximum triangle throughput for nVidia graphics cards is achieved by rendering triangles in batches of 500 or more triangles at a time, so this is also a good size for the triangle groups used for frustum culling.

The idea is that if the bounding volume for such a group is fully outside the viewing frustum then so are all the triangles it contains. If, on the other hand, the bounding volume is fully contained in or intersects the frustum then all triangles inside it are rendered, even if a few (or even most) of them are actually outside the frustum. In large and complex scenes, the frustum culling technique can quickly cull away a large percentage of the triangles that are outside the viewing frustum and thus it can accelerate the rendering of the scene considerably.

Two types of bounding volumes that are often used for frustum culling are the bounding sphere and the axis-aligned bounding box[1].

The intersection test between a sphere and a frustum is simpler and thus quicker than the test between an AABB and a frustum so it would seem reasonable to use a sphere as the bounding volume. And it is the best choice indeed for those meshes that are properly approximated by a sphere, but often an AABB

[1]Often referred to as an AABB, it is a box with sides that are parallel to the X,Y or Z plane. As its orientation is fixed, it can be described by just two points - a minimum and maximum point.

gives a much tighter fit around a mesh making it a more suitable bounding volume as illustrated in figure 5.2. The more 'air' there is inside a bounding volume, the greater is the chance that the bounding volume will intersect the frustum without *any* of its contained triangles doing so, which is the worst-case scenario in frustum culling. When this is combined with the fact that while it is easy to calculate *a* bounding sphere for a mesh it is not trivial to calculate the *minimum* bounding sphere, it becomes clear why the AABB is the most common choice for bounding volumes.

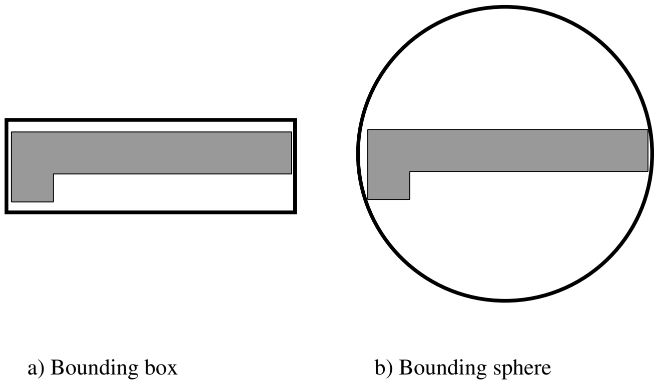

Figure 5.2: Bounding volumes

To perform intersection tests between the view frustum and bounding volumes the frustum is represented as six planes, all with normals pointing towards the inside of the frustum. In order to check if a single point is within the frustum we just have to determine whether the point is in front of all six planes. If this is the case, then the point must be within the volume. Checking a sphere against the frustum isn't much harder, we just have to check that the sphere isn't fully behind any of the planes. In other words, we must check that the signed distance from the sphere center to each plane is greater than the negative sphere radius. Determining whether an AABB and a frustum intersects is a bit more complicated as the AABB consists of two points that can appear in many different configurations relative to the view frustum planes. But the main idea is still to check the spatial relations between the two points and the six frustum planes.

5.2 Bounding a vertex shader shadow volume

For any rigid object with a constant size and position in world-space it is easy to calculate a bounding volume, and use this for culling. However,as described above in section 3.3.3, a VS shadow volume is extruded by the vertex shader in a direction that depends on the position of the light. This makes it harder to calculate a bounding volume that encloses the shadow volume because we do not have the extruded version in system memory. Therefor we cannot simply loop through the vertices to measure their extent.

What we *can* do is calculate the AABB for the collapsed shadow volume in object-space. By simulating the vertex shader, which performs the extrusion of the shadow volume on this bounding box, we get an extruded box, and the AABB of *this* box is guaranteed to enclose the extruded shadow volume also, thus making it a valid bounding volume; see figure 5.3.

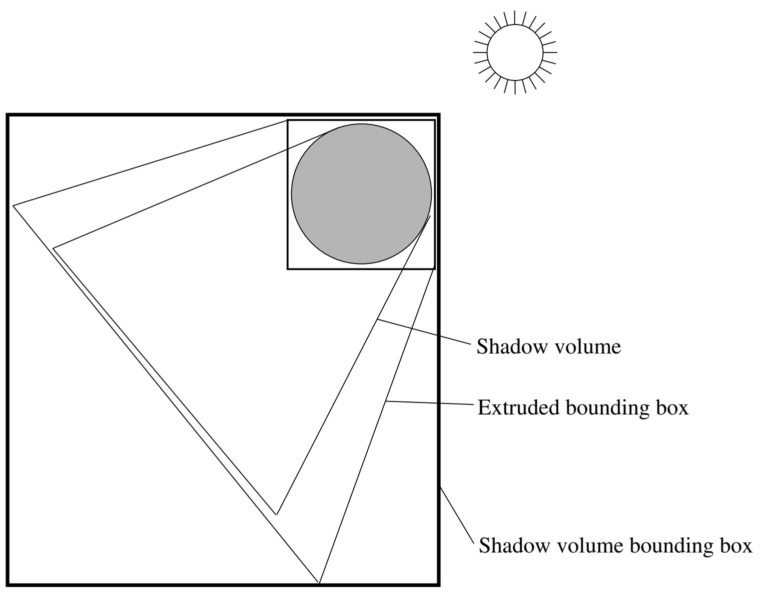

Figure 5.3: Bounding a vertex shader shadow volume

Actually, since we do not need a closed shadow hull for the extruded bounding box, (we will not render it, we just need its points so we can measure its extent), we do not need to create a special version of it with degenerate triangles at each edge as was the case with the normal geometry. Nor do we need to determine which edges are on the silhouette in order to extrude it. Instead we just extrude all eight points and calculate our final bounding box by taking the minimum and maximum points of the eight original points and the eight extruded ones. The resulting bounding box may be slightly larger than what we would have gotten from the true extruded box, but it will never be smaller, so it *is* a valid bounding box for the shadow volume. The reduced complexity of extruding the box outweighs the fact that our bounding volume is not as tight a fit as it could be.

5.3 Scene tree

A scene tree is a data structure into which one can insert the objects in a scene. The scene tree supports efficient queries to obtain the objects that intersects different volumes such as a frustum, sphere or box. Often a scene is dynamic, with one or more animated and moving objects. This implies that our scene tree should provide fast remove and insert operations so that we can efficiently re-insert an object when it has moved.

The scene tree we present here is based on a quad-tree[2] but has several key differences which we will describe later. A quad-tree is a rooted tree where each node has four children, hence the name. Each node corresponds to a 2d square, and the four children of a node correspond to the four quadrants of this square. As in any tree structure, the nodes that do not have children are called leaves and is usually where the data is stored. One application of a quad-tree is to store a set of points in the plane. In that case, the square of the root node is equal to a bounding square to all the points, and the tree is then subdivided until no more than one point resides in each leaf; see figure 5.4.

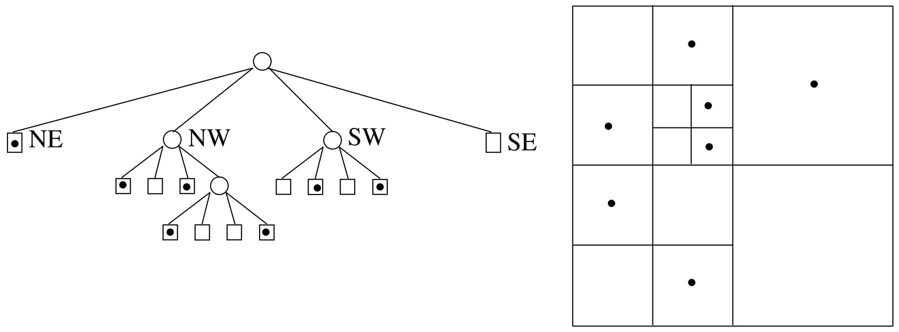

Figure 5.4: Storing points in a quad-tree

Even though a quad-tree is a 2d structure we can easily extend it to storing points in 3d. We simply assign a fixed top and bottom y value to each node, f.ex. taken from the 3d bounding box of all the points, making each node represent a 3d box instead of a 2d square. The tree is still only subdivided along two axes, namely the X and Z-axis, so each node still has exactly four children[3].

In its basic form, a quad-tree has several properties that make it unsuitable for use as a scene tree though. For example it is not balanced and the depth of a particular branch depends on the density of the stored items in the region the

[2]Many sources describe the quad-tree f.ex. [dBvKO00]

[3]A variant of the quad-tree exists which also subdivides along the Y-axis. As this results in each node having eight children instead of four, this variant is appropriately called an octree.

branch covers. This means that when an item is moved and re-inserted in a new region of the tree new subdivisions may occur which involve allocation of new leaves, splitting items into these new leaves, deletion of the old leaf etc.

However, we want to be able to set up the scene tree once and for all at load time and then be able to move items around without making any changes to the overall structure of the tree. We also want to use the scene tree to store objects, instead of just points. This means that it is not always possible to subdivide a node since the objects inside it may overlap each other or the boundaries between two children. An object can also be fully contained in a node but be unable to fit inside any of the children. In figure 5.5 objects A and B spans multiple child nodes and cannot be put into either of them while object C is small enough to be put into the lower left subdivision.

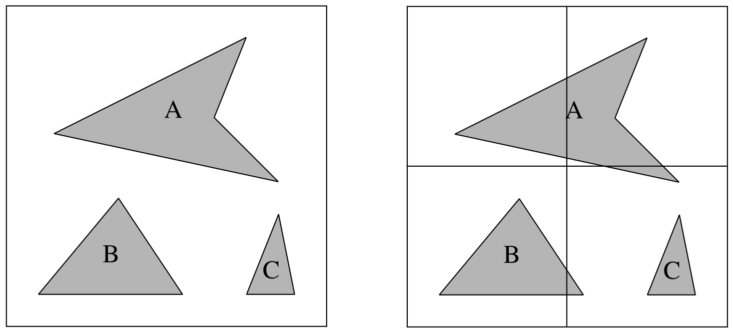

Figure 5.5: Splitting objects

To overcome the allocation problems, we set up our scene tree as a full hierarchy of preallocated and initially empty nodes and leaves. As with the quad-tree, we start with a bounding box for all the objects we want the scene tree to contain. However, rather than subdividing the tree until the amount of objects in each leaf is small enough, we subdivide the tree until the *size* of each leaf is small enough. What this size is exactly depends on the objects being stored in the tree, but it should be at least as large as the smallest object in the scene, (any leaf smaller than this will not be able to contain any objects anyway). As the tree is intended to accelerate culling queries, it is actually not desirable to subdivide it all the way down to the individual objects, so in practice the leaves can be large. In our implementation each leaf has a side length of 5 meters in our virtual world.

To overcome the splitting problem, we extend our scene tree so it can store objects in the nodes as well as in the leaves. To insert an object into the tree we 'push' it as far down the tree as possible, putting it in the tightest fitting node or leaf. As the root node in the scene tree has a box the same size as the bounding box of the scene it can contain any object in the scene and, as a result, an insertion into the scene tree can never fail.

As the removal of an object does not affect the overall structure of the scene tree it can be performed by simply removing the object from the containing node or leaf. If objects are assigned a handle to their scene tree node at insertion time, removal can be done in time $O(1)$. Insertion of an object does not change the overall structure either, and can be done in time linear in the depth of the tree. This is $O(log_2(m))$, where m is the maximum of the X and Z side lengths of the scene bounding box.

Using the implementation described above, it is possible to handle scenes with dynamic objects efficiently and in typical game scenes, where the amount of moving objects is relatively small, there is no significant performance hit involved in re-inserting the moved objects. As a further optimization it is worth noticing that, over a single frame, a moving object will often only have moved within the node or leaf that it already resides in, making a re-insertion unnecessary. A check for this case can performed in constant time using the objects handle to its scene tree node. Also instead of removing an object and re-inserting it from the root, it is possible to push the object upwards in the tree until it is fully contained in a node, and then push it downwards as far as possible. In some cases this method faster than doing a full re-insertion, but in the worst-case scenario the cost is doubled.

The scene tree can be used to accelerate intersection queries between the objects in the scene and volumes such as a frustum, a sphere or a box. The main idea is that if a node's box is fully outside the volume then all its children must be outside as well and we can totally skip the sub-tree. If a node's box is *fully* contained in the volume then so are all the objects that the node and its children contain. Thus, we can include all objects in this node's sub-tree without any further intersection checks. In the final case, where the volume intersects the node's box, each object in the node is checked for intersection and the algorithm is called recursively on each of the four children.

Assuming a roughly equal distribution of objects throughout the scene tree this means that with a single intersection check large parts of the scene can be culled away. In figure 5.6, just three checks between the sphere and the node boxes at a certain level in the scene tree culls away $\frac{3}{4}$ of all objects in the sub-tree.

5.4 Efficient shadow rendering

In the following we will assume that all lights are omni-directional point lights with a finite range. This means that each light source has a *sphere of influence* with its center at the position of the light and a radius equal to the light range. The light cannot affect objects outside its sphere of influence.

The multi-pass stencil shadow volume is described in section 3.2.2. One of the

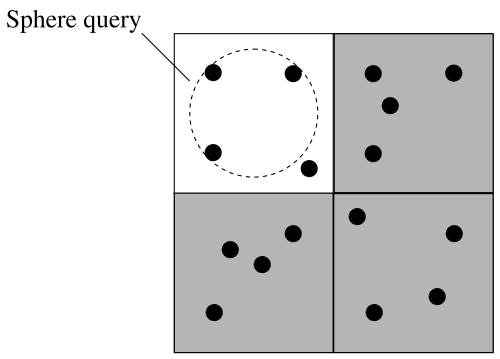

Figure 5.6: Culling grey areas with three intersection checks

sub-routines in the algorithm is to render the scene for every light-source while using the stencil-buffer to allow drawing solely in areas that are illuminated by the light. If we follow this procedure blindly, we could end up doing a lot of work that would have no effect on the final image. These are the three main cases where superfluous work often occur:

1. A light-source does not affect anything in the view frustum. In this case, both the rendering of shadow volumes for the light and the extra pass over the geometry is redundant.
2. An object is so far away from a visible light-source that the object receives no light from it. In this case, the rendering of the object and its shadow volume is redundant.
3. An object is affected by a light source, but the shadow volume is outside the view frustum. In this case the rendering of the shadow volume is redundant.

As described in section 5.1, there is an efficient algorithm for checking a sphere against a frustum so to avoid case number 1 in the list above we simply have to check the light source's sphere of influence against the view frustum. If the sphere is fully outside the view frustum then neither the light source nor its shadow volumes can affect the final image and we can skip any further processing of the light. The cost of finding the lights that affect the view frustum in this way is linear in the total number of lights in the scene, and whether or not to accelerate the process through a tree structure depends on the application. In practice, however, most scenes will probably have relatively few light sources[4] and it is unlikely that simply checking all light sources linearly will result in a significant performance drop as the 'sphere vs. frustum' check is fairly cheap.

[4]Usually less than 100.

If a light affects the final image, we must render all the shadow volumes it has caused. Naturally, only those objects in the scene that lie within the lights sphere of influence can cast a shadow from it. We can therefore find the set of objects that intersects the lights sphere of influence and render only the shadow volumes for those. In doing so, we have avoided case number 2 in the list above. Again, there is an efficient intersection algorithm for AABB vs. sphere checks that we can use to check the object's bounding box against the sphere of influence. However, a scene often contains many objects, and a linear algorithm that checks each object against the sphere of influence will be too slow. Instead we set up a scene tree, as described in section 5.3, and use it to quickly find all objects that intersect the light's sphere of influence.

Once we have the set of objects that intersects the light's sphere of influence, and thus the set of potentially visible shadow volumes, we note that it is possible for a visible light to cause a shadow volume that is fully outside the view frustum. To avoid rendering these we simply check each shadow volume's bounding box against the view frustum. For shadow volumes calculated on the CPU it is easy to maintain a bounding box, and for shadow volumes extruded in a vertex shader we calculate a bounding box as described in section 5.2. Through this last culling mechanism we have avoided case number 3 in the list and should be rendering only those shadow volumes that actually affect the final image.

We have used the ideas described above to modify the multi-pass stencil shadow algorithm and make it capable of rendering large scenes efficiently:

1. Clear color-buffer and z-buffer.

2. Render the scene with only ambient and emissive lighting.

3. For all lights l:

   (a) Check l's sphere of influence against the viewing frustum. If l does not intersect the frustum we skip it.

   (b) Query the scene tree for the set of objects o that intersects l's sphere of influence. For every object in o, check the AABB of the l-generated shadow volume against the view frustum.

   (c) Clear stencil-buffer, disable writing to color-buffer and z-buffer, set z-buffer test to less-than.

   (d) For all visible shadow volumes v:

      i. Render all *front facing* triangles of v generated by l, *incrementing* the stencil value when passing the z-test.

      ii. Render all *back facing* triangles in v generated by l, *decrementing* the stencil value when passing the z-test.

(e) Re-enable writing to color-buffer, set z-buffer test to equal, set stencil test to pass when value is 0, and set additive blending.

(f) Render all objects in o with only diffuse and specular lighting from l.

The modification does not affect optimizations of the innermost loop such as the two-sided stencil technique and Carmacks reverse, described in section 3.3.

Using the above culling procedures does not guarantee that a scene does not generate too many visible shadow volumes for the graphics card to handle at an acceptable frame rate. In that case, we can start culling away *visible* shadow volumes. As this will result in visual errors, it is important first to cull away the shadow volumes that contribute least to the final image. We suggest calculating an 'importance value' for each visible shadow volume, based on some heuristic, and then sorting the shadow volumes and rendering the most important ones first. It is then possible to allot a certain amount of time or triangles to each light and to stop rendering shadow volumes once that amount has been exceeded.

Two factors can be used in calculating an importance value for a shadow volume: the distance from the center of the volume to the camera, and the projected size on the screen of its bounding box. The idea behind the first factor is that we would rather cull away a distant shadow volume than a close one, as it is likely the viewer will focus more on geometry in the foreground than in the background. The idea behind the second factor is that we would rather cull away smaller shadow volumes as they contribute less to the final image than larger ones.

Determining how to weigh these two factors requires some tweaking and depends on the scene in question. Sometimes a large shadow in the background is much more important than a small one close to the camera, whereas in scenes where each shadow volume is approximately the same size one might want to sort the volumes purely by distance to the camera.

Chapter 6

Implementation details

To actually implement the techniques discussed so far in this thesis can be a daunting task with lots of potential pitfalls and caveats. In this chapter we present some implementation details which was left out in the previous chapters for clarity reasons. We begin by giving the reader an overview of the game engine we have incorporated the techniques into. Then we present a solution to the seemingly simple problem of sampling a screen sized texture map at coordinates corresponding to the current pixel being shaded in the render target. Finally, we present the full CG source code for the pixel and vertex shaders required to rasterize the penumbra wedges into the LI buffer.

6.1 The Peroxide engine

As one of our main goals with this thesis was to test the applicability of soft shadows in a true game environment, we implemented our version of the soft shadow algorithm within our game engine - the *Peroxide Engine*. A full game engine is a very complex application and a detailed description of its components is beyond the scope of this thesis. Still it is important to realize that the added complexity of a game engine compared to a simple test application can significantly affect the measured performance of the techniques. In this section, we give a short overview of the different components and features of our game engine to give the reader an understanding of the framework in which our experiments have been conducted.

Platform and API independence

The Peroxide Engine has been developed in a platform independent way, which means that it can run on multiple platforms using several different APIs for graphics, sound and input. At the moment, the engine runs on Linux, using SDL and OpenGL for graphics and input, as well as under Windows where the DirectX framework is used instead. This cross platform feature is achieved by wrapping all API and platform specific code in 'drivers' that expose all functionality through a common interface to the main game engine. A number of such drivers exist in order to provide access to the different categories of platform specific code:

OS Provides OS specific code for timing functions, file selection dialogs, dialog boxes etc.

GfxDriver Provides an abstraction to everything having to do with graphics. Examples of this includes wrappers for vertex buffers, state management on the graphics card, drawing code, and shader management.

InputDriver Provides an interface to the mouse and keyboard. Does also implement functionality to bind callback functions to key or mouse events.

SoundDriver Provides an interface to the playback of music files, as well as to 2d and 3d sound effects.

NetDriver Provides an interface to networking code, which is required for a client/server application.

The main game engine should compile on any platform with a C++ compiler and STL. The drivers described above are the only components which must be reimplemented to support a new platform.

This portability comes at a price though. Each call to code within one of the drivers is wrapped, typically with a virtual function call, and is thus a bit more expensive than a similar call in a platform specific application. In a properly optimized game engine there will be relatively few calls to the drivers per frame, but if for example the graphics driver is used in a suboptimal way (lots of render calls for example) the abstraction layer causes a performance drop.

The rendering framework

The Peroxide Engine uses an *effect* framework for rendering. Before rendering can take place an effect must be obtained from the graphics driver. If the desired effect exists in a version compatible with the detected hardware the graphics driver returns a pointer to the effect. The effect provides *begin* and *end* methods along with a mechanism for setting parameters that varies with the objects being rendered. Any geometry rendered between the 'begin' and 'end' calls is drawn as specified by the effect. If multiple passes over the geometry are required for a certain implementation of an effect, the effect interface will specify exactly how many passes are required and, if different versions of the geometry are involved, the effect will specify the order in which the application should draw them.

One benefit of this system is that multiple *render paths* can be implemented for various generations of graphics cards, each using the latest features on the hardware for the fastest and best looking implementation of the desired effect. For example in our engine we have three different implementations of a water effect, ranging from a simple texture mapping to a complex pixel shader implementation that only runs on cards with ps2.0 or better. As better hardware becomes available it is easy to implement a new version of the effect without making any changes in the actual game engine. In addition, if a certain effect does not exist for the detected hardware the game engine will know this and will skip the rendering of the geometry that needs the effect.

Dynamic worlds

The Peroxide Engine is built with large, fully dynamic, indoor and outdoor environments in mind. As a result, no assumptions are made about the relationships between the entities that make up the world. Every object or light can freely be moved around, without any significant performance decrease. This entails that all shadows are dynamic. For outdoor scenes, a fully dynamic landscape component has been implemented that allows for real-time modifications to shape, texture and color among other things. This allows for game-play effects such as craters that appear if a bomb is dropped, or permanent scorch marks caused by fires. A dynamic day-cycle has been implemented where a number of key-frames specify properties such as the ambient light, the sun's position, color and strength as well as various fog settings for specific times of the day. This dynamic world is efficiently managed through a scene tree structure, as described in chapter 5.

Script languages

Two custom scripting languages have been developed for the Peroxide Engine. The first is called *PxdScript* and is a programming language with a C-like syntax. Through PxdScript it is possible to manipulate the game engine and the entities in a scene. PxdScript is typically used to implement game-play events, AI scripts for the NPCs[1], and *services*. Services are scripts that run continuously in the scene for example to rotate the cogs of a machine or the wings of a windmill. A virtual machine executes PxdScript programs and allow pseudo-parallel execution and saving and loading of running programs. The second language in the engine is used for scripting dialogs with the NPCs in the world through a very high-level syntax.

[1]Non Player Characters: characters controlled by the engine as opposed to the character controlled by the player.

Additional features

In addition to the features mentioned above, the Peroxide Engine includes a few components that we will only mention very briefly, as they have limited or no relevance to the topics covered in this thesis. Still, they are mentioned here as they indirectly affect the measured performance of the soft shadow algorithm by imposing a constant CPU overhead each frame that would not be present in a test application.

We have implemented a GUI framework, using accelerated 3d graphics, in which it is possible to set up and to render windows with different 'skins' or 'looks'. We use this GUI toolkit for some of our in-game windows as well as for some of our editing tools. We have also implemented a flexible and highly parameterized particle system as well as a system for cloth animation. The Peroxide Engine also includes an animation system that allows for skeletal animation with up to four weights per bone, and a system for mixing animations allowing for smooth blends from one animation to another. All 3d models and animations are exported from 3d studio max to our custom file formats using our own export plug-ins.

The editing of our worlds is conducted in real-time using editing features built into the game engine. The editing can be performed either offline on maps that reside locally on the client machine or online through an editing server called *NetEd*. By connecting to NetEd it is possible for multiple users to edit the same map simultaneously. This is useful since our game maps, typically, are too big and complex for a single world builder to handle by himself. When the client connects to the NetEd server the current state of the map is saved to a buffer and sent to the connecting client. After this point any changes a client might make to the map is propagated to all connected clients to keep them synchronized.

6.2 Calculating screen-space coordinates in a shader

Certain graphical multi-pass algorithms first render some sort of information to a screen sized texture and then, in a later pass, they sample the texture to retrieve the information stored in the pixel corresponding to the one currently being shaded in the render-target. A good example of this is the soft shadow algorithm which uses this technique twice: the wedge pixel shader retrieves depth information from the extra z-buffer, and the light pixel shader retrieves the light intensity from the LI-buffer.

To read from a texture, texture coordinates have to be available. So, reading from a screen sized texture presents us with the problem of finding texture coordinates that will lookup the texel corresponding to the pixel that is about to be shaded[2].

In order to calculate these texture coordinates, it is necessary to look at the *viewport transformation* which transforms a vertex from projected-space onto the screen. This is an additional step in the transformation process shown in figure A.4. The viewport transformation converts the coordinates from the range $[-1..1]$ in projected-space to actual pixel coordinates in the image, f.ex. to the range $[0..1023] \times [0..767]$. As it is possible to map projected-space to any rectangular area of the screen, the image resolution does not necessarily have to match the screen resolution.

When the viewport is equal to the full screen size, and has full depth range, the viewport transformation matrix V is given as (see [Mic] Viewport Scaling):

$$V = \begin{bmatrix} \frac{W}{2} & 0 & 0 & 0 \\ 0 & -\frac{H}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{W}{2} & \frac{H}{2} & 0 & 1 \end{bmatrix} \tag{6.1}$$

where W and H are the width and height of the screen in pixels.

Let p be a point in projected-space ($p_{xyz} \in [-1..1]$). The final screen-space position p' (in pixels) is calculated by applying the viewport transformation matrix to the homogeneous projected-space coordinates. The result is finally homogenized into Cartesian coordinates by dividing with p'_w:

$$p' = \frac{pV}{(pV)_w} = \frac{pV}{p_w} \tag{6.2}$$

For a transformation matrix V as given in equation 6.1, and from the definition of p' in 6.2, we find that:

$$\begin{aligned} \frac{pV}{p_w} &= \frac{\left[p_x\frac{W}{2} + p_w\frac{W}{2}, -p_y\frac{H}{2} + p_w\frac{H}{2}, p_z, p_w\right]}{p_w} \\ &\Downarrow \\ p' &= \left[\frac{1}{2}(\frac{p_xW}{p_w} + W), \frac{1}{2}(-\frac{p_yH}{p_w} + H), \frac{p_z}{p_w}, 1\right] \end{aligned} \tag{6.3}$$

To sample a texture, the supplied texture coordinates are transformed into texel coordinates t as follows (see [Mic] Directly Mapping Texels to Pixels):

$$t_{xy} = c_{xy}s_{xy} + 0.5 \tag{6.4}$$

[2]OpenGL fragment programs have these coordinates accessible as a built-in variable but this is not the case for DirectX pixel shaders.

where c is the texture coordinates and s is the texture size. Since we want to sample a screen sized texture we have that $s_x = W$ and $s_y = H$. To sample the correct texel in the screen sized texture we need to calculate texture coordinates c_{xy} so that $p'_{xy} = t_{xy}$:

$$\begin{aligned}
p'_{xy} &= t_{xy} \\
&= c_{xy}s_{xy} + 0.5 \\
&\Updownarrow \\
c_{xy} &= \frac{p'_{xy} - 0.5}{s_{xy}} \\
&= \left[\frac{\frac{1}{2}(\frac{p_x W}{p_w} + W) - 0.5}{W}, \frac{\frac{1}{2}(-\frac{p_y H}{p_w} + H) - 0.5}{H}\right] \\
&= \left[\frac{\frac{p_x W}{2p_w} + \frac{W}{2} - 0.5}{W}, \frac{(-\frac{p_y H}{2p_w} + \frac{H}{2} - 0.5}{H}\right] \\
&= \left[\frac{p_x}{2p_w} + 0.5 - \frac{1}{2W}, -\frac{p_y}{2p_w} + 0.5 - \frac{1}{2H}\right] \\
&= \left[\frac{p_x}{2}/p_w + 0.5 - \frac{1}{2W}, -\frac{p_y}{2}/p_w + 0.5 - \frac{1}{2H}\right] \qquad (6.5)
\end{aligned}$$

Unfortunately, we have to evaluate equation 6.5 in the pixel shader for each pixel, rather than in the vertex shader for each vertex. It is not possible to calculate the equation in the vertex shader because linearly interpolating a and b and then calculating a/b is not the same as interpolating a/b. However it is possible to calculate the expressions $\frac{p_x}{2}$ and $-\frac{p_y}{2}$ in the vertex shader and interpolate these values. Furthermore, the expressions $0.5 - \frac{1}{2W}$ and $0.5 - \frac{1}{2H}$ are both constant and can thus be uploaded to a constant register in the pixel shader. All the pixel shader must do is to divide the interpolated $\frac{p_x}{2}$ and $-\frac{p_y}{2}$ by the interpolated p_w, and then add the constant uv displacement. This is trivially vectorizable and can be implemented in two assembler instructions (one reciprocal and one multiply-and-add instruction, see [Mic] Instructions - ps_2_0).

The CG code that implements the above is shown here:

```
// Vertex shader
struct appin {
  float4 position : POSITION;
};

struct wedgeOut {
  float4 position: POSITION;
  float4 posData: TEXCOORD0;
};
```

```

wedgeOut main(appin IN,
              uniform float4x4 worldViewProj : register(c0))
{
  wedgeOut OUT;
  OUT.position = mul(IN.position, worldViewProj);

  // The line below implements the per-vertex part of the lookup technique:
  OUT.posData = OUT.position;
  OUT.posData.y = -OUT.posScreenSpace.y;
  OUT.posData.xy *= 0.5; return OUT;
}
```

```
// Pixel shader
struct appIn {
  float4 position: POSITION;
  float4 posData: TEXCOORD0;
};

float4 main(appIn IN, uniform float2 uvOffset : register(c0))
{
  // Calculate screenspace UV coords
  float4 temp = IN.posData;
  float2 screenSpaceUV = (temp.xy/temp.w) + uvOffset;

  // sample the screen-sized texture:
  ....
}
```

6.3 The soft shadow algorithm

In this section we show the application code issuing the rendering calls and the vertex and pixel shaders used in the soft shadow algorithm. The algorithm is described in chapter 4, and an overview of it is given in section 4.1.6, so we will not repeat the description here, but we will clarify some of the more obscure details in the implementation.

Application code

```
// the hard shadow part:
Effect* hardShadowEffect = gfxdriver->getEffect("hardWedge");
if (hardShadowEffect) {
  // for all passes (=2):
  for (int i = 0; i < hardShadowEffect->getNumPasses(); i++) {
    hardShadowEffect->begin(i);

    for (int j = 0; j < shadowVolumes.size(); j++) {
      shadowVolumesj->renderHardShadow();
    }
  }


  hardShadowEffect->end();
15  }
16
17  // the wedges:
18  wedgeEffect = gfxdriver->getEffect("wedge");
19  if (wedgeEffect) {
20    // parameters for wedge
21    VECTOR viewSpaceLightPos = gfxdriver->getViewMatrix() * light->GetPosition();
22    float lightRadius = light->getRadius();
23
24    gfxdriver->clear(COLOR(), CLEAR_STENCIL_ONLY);
25
26    // Draw all volumes:
27    for (int i = 0; i < shadowVolumes.size(); i++) {
28
29      // render each wedge:
30      for (int j = 0; j < shadowVolumesi->getNumWedges(); j++) {
31
32        // Inner wedge half
33        wedgeEffect->begin(0);
34        shadowVolumesi->renderInnerWedgeNr(j);
35        wedgeEffect->begin(1);
36        wedgeEffect->setParameter(EP_LIGHT_POSITION, viewSpaceLightPos/lightRadius);
37
38        wedgeEffect->setParameter(EP_LIGHT_RADIUS, VECTOR4D(lightRadius,0,0,0));
39        shadowVolumesi->renderInnerWedgeNr(j);
40
41        // Outer wedge half
42        wedgeEffect->begin(2);
43        shadowVolumesi->renderOuterWedgeNr(j);
44        wedgeEffect->begin(3);
45        wedgeEffect->setParameter(EP_LIGHT_POSITION, viewSpaceLightPos/lightRadius);
46        wedgeEffect->setParameter(EP_LIGHT_RADIUS, VECTOR4D(lightRadius,0,0,0));
47        shadowVolumesi->renderOuterWedgeNr(j);
48      }
49
50    }
51    wedgeEffect->end();
52  }
```

The application code uses the *effect* framework described in section 6.1, which is why the state settings are not visible in the code below. The shadow volumes are represented by C++ objects with a number of convenience methods. F.ex. it is possible to get the number of wedges and to render the inner and outer half of each wedge separately.

The application code renders two things: the hard shadow and the wedges. The wedge rendering, (lines 33–47), shows that four render calls are used per wedge, two for each wedge half. The two render calls for each half accomplish the culling of fragments that are not in the penumbra area, as it is described in section 4.1.3.

Wedge vertex shader

```
struct appin {
  float4 position : POSITION;
3    float4 edgePoint0 : TEXCOORD0;
4    float4 edgePoint1 : TEXCOORD1;
5  };
6
7  struct wedgeOut {
8    float4 position: POSITION;
9    float4 posData: TEXCOORD0;
10   float3 posViewSpace : TEXCOORD1;
11   float3 edgePoint0 : TEXCOORD2;
12   float3 edgePoint1 : TEXCOORD3;
13   float  r3DepthViewSpace : TEXCOORD4;
14 };
15
16 wedgeOut main(appin IN, uniform float4x4 worldViewProj : register(c0),
17                         uniform float4x4 worldView     : register(c4),
18                         uniform float rcpLightRadius   : register(c10)) {
19
20   wedgeOut OUT;
21   OUT.position = mul(IN.position, worldViewProj);
22
23   // Calculate screenspace position
24   OUT.posData = OUT.position;
25   OUT.posData.y = -OUT.posData.y;
26   OUT.posData.xy *= 0.5;
27
28   OUT.posViewSpace = mul(IN.position, worldView).xyz;
29   OUT.r3DepthViewSpace = OUT.posViewSpace.z;
30   OUT.posViewSpace.xyz *= rcpLightRadius;
31
32   OUT.edgePoint0 = mul(IN.edgePoint0, worldView).xyz * rcpLightRadius;
33   OUT.edgePoint1 = mul(IN.edgePoint1, worldView).xyz * rcpLightRadius;
34
35   return OUT;
36 }
```

The vertex and pixel shaders use the technique described in section 6.2 to calculate the screen-space coordinates, which is seen in vertex shader lines 24–26 and pixel shader line 19. The edge points are passed to the pixel shader via the vertex data (in texture coordinates 0 and 1, line 3 and 4). Since the vertices are not shared between wedges, these data are constant for all triangles involved in a wedge, which is why there is no risk that interpolation will change these values.

The `rcpLightRadius` variable is the reciprocal value of the radius of the current light, and is used to transform the view-space position and the edge point 0 and 1, (see lines 30–33), into unit sphere space as described in section 4.2.1. We use the reciprocal value because it allows us to use multiplication instead of division, saving an assembly instruction.

Wedge pixel shader

```
1  struct appIn {
2    float4 position: POSITION;
3    float4 posData: TEXCOORD0;
4    float3 posViewSpace : TEXCOORD1;
```

```
  float3 edgePoint0 : TEXCOORD2;
  float3 edgePoint1 : TEXCOORD3;
  float  r3DepthViewSpace : TEXCOORD4;
};

float4 main(appIn IN, uniform sampler2D depthMap   : register(s0),
                      uniform sampler3D visMapSameSide : register(s1),
                      uniform sampler3D visMapDiffSide : register(s2),
                      uniform float3 lightPos   : register(c0),
                      uniform float2 uvOffset   : register(c3),
                      uniform float rcpLightRadius : register(c4),
                      uniform float4 resultModulator: register(c5)) : COLOR {

  // Calculate screenspace UV coords
  float2 screenSpaceUV = (IN.posData.xy/IN.posData.w) + uvOffset;

  // Sample depth value
  float depthValue = tex2D(depthMap, screenSpaceUV).r;

  // Find position in view-space of geometry behind this wedge pixel
  float3 geometryPos = IN.posViewSpace * (depthValue / IN.r3DepthViewSpace);

  // Find plane through edge and geometryPos
  float3 geoPlaneNormal = normalize(cross(IN.edgePoint0 - geometryPos, IN.edgePoint1 - geometryPos));

  // distance from plane to lightPos (if within range -1,1 it intersects the light):
  float d0 = dot(lightPos - IN.edgePoint0, geoPlaneNormal);

  // Project lightPos to geoPlane:
  float3 basePoint = lightPos - d0 * geoPlaneNormal;

  // Find normal of lightPlane
  float3 lightPlaneNormal = normalize(geometryPos - basePoint);

  // Project e0 and e1 onto lightPlane
  float distToPlane = dot(IN.edgePoint0 - basePoint, lightPlaneNormal);
  float3 edgePoint0Proj = IN.edgePoint0 - distToPlane * lightPlaneNormal;

  distToPlane = dot(IN.edgePoint1 - basePoint, lightPlaneNormal);
  float3 edgePoint1Proj = IN.edgePoint1 - distToPlane * lightPlaneNormal;

  // Determine if the projected points are on the same side of base point or not.
  float3 baseToE0p = edgePoint0Proj - basePoint;
  float3 baseToE1p = edgePoint1Proj - basePoint;

  // Calculate distance from base point to the two projected points.
  float d1 = length(baseToE0p);
  float d2 = length(baseToE1p);

  // The look-up texture coordinate:
  float3 uvw = float3(abs(d0), d1, d2);

  // Sample the correct map
  float coverage;
  if (dot(baseToE0p, baseToE1p) > 0) {
    coverage = tex3D(visMapSameSide, uvw).r;
  }
  else {
    coverage = tex3D(visMapDiffSide, uvw).r;
  }

  // Calculate the changes to make according to coverage.
```

```
67    float4 result = float4(0,coverage,0,coverage) * resultModulator;
68
69    return result;
70  }
```

Conceptually the pixel shader can be divided into three parts: finding the geometry (or fragment) position (until line 25), calculating the coverage value based on the geometry position (until line 65), and using the coverage value to give the desired output. The geometry position is found by using the pixel position and reading the depth value from the depth buffer as described in section 4.1.4.

Calculating the coverage value is the most expensive part. A high-level description of the computation is found in section 4.1.4, and here we will describe it in further detail. First, we find the geoPlane, or rather the normal to it (`geoPlaneNormal` at line 28). With this, we can find d_0: the signed distance from geoPlane to the light source. We can then use d_0 to find the basePoint by projecting the light source onto the geoPlane (line 34). The normal to the lightPlane can now be found as the normalized vector from the basePoint to the geometry position (line 37). We then project the edge points onto the lightPlane and form the two vectors from the base point to each of the projected edge points (`baseToE0p` and `baseToE1p`). These two vectors are used for finding d_1 and d_2 as well as for calculating whether the projected points are on the same or different sides of the basePoint. The projected edge points are on the same side if the dot product between the two vectors are greater than zero (line 59). Knowing this, and knowing the values of d_0, d_1 and d_2 we can sample the correct function map to lookup the coverage value.

Using the coverage value to give the desired output is very simple. We want the coverage value outputted to either the y or the w channel, depending on whether the fragment is in the inner or outer penumbra region. Since this version of the soft shadow algorithm uses the split wedge geometry we know where the fragments are located without having to make any calculations. Consequently we could have made two slightly different versions of the pixel shader; one that outputs the coverage value to the y and one that outputs to the w channel. Instead we use a little trick. The `resultModulator` variable contains either $(0, 1, 0, 0)$ or $(0, 0, 0, 1)$ and line 67 therefore masks out the correct output channel. This trick costs an extra pixel shader instruction but allows us to use the same shader for both halves, thus saving the overhead of switching shaders and allowing us to maintain one pixel shader instead of two.

6.4 The per-loop soft shadow algorithm

This section shows some of the code for our implementation of the per-loop soft shadow algorithm described in section 4.4. Some of the code is identical to the code of the original algorithm and we will only discuss the code that is different here. As the vertex shader in this version is identical to the original it is not shown.

Per-loop application code

```
1  Effect* wedgeEffect = gfxdriver->getEffect("wedge");
2
3  if (wedgeEffect) {
4    gfxdriver->setRenderTarget("coverageTexture");
5    gfxdriver->clear(COLOR(0,0,0,0), CLEAR_COLOR_ONLY);
6
7    // For every shadow volume...
8    for (int j = 0; j < shadowVolumes.size(); j++) {
9      ShadowVolume* volume = shadowVolumesj;
10
11     // for every silhouette loop...
12     for (int loopNum = 0; loopNum < volume->getNumLoops(); loopNum++) {
13
14       // Clear softDataTexture
15       gfxdriver->setRenderTarget("softDataTexture");
16       gfxdriver->clear(COLOR(0,0,0,0), CLEAR_COLOR_ONLY);
17
18       // Hardshadow
19       wedgeEffect->begin(0);
20       volume->renderHardLoopNr(loopNum);
21       wedgeEffect->begin(1);
22       volume->renderHardLoopNr(loopNum);
23
24       VECTOR viewSpaceLightPos = gfxdriver->getViewMatrix() * light->GetPosition();
25       float lightRadius = light->getRadius();
26
27       // Stencil out the penumbra area
28       wedgeEffect->begin(2);
29       volume->renderWedgeLoopNr(loopNum);
30
31       // Run PS in the stenciled out area calculating loop-local coverage into softDataTexture
32       wedgeEffect->begin(3);
33       wedgeEffect->setParameter(EP_LIGHT_POSITION, viewSpaceLightPos/lightRadius);
34       wedgeEffect->setParameter(EP_LIGHT_RADIUS, VECTOR4D(lightRadius,0,0,0));
35       volume->renderWedgeLoopNr(loopNum);
36
37       // Transfer the calculated value to coverageTexture
38       wedgeEffect->begin(4);
39
40       wedgeEffect->end();
41     }
42   }
43 }
```

The application code is a lot different from the original algorithm since we render per silhouette loop; the for-loop in line 12 accomplishes this. The shadow volume objects used here are also different from the ones in the original algorithm as they must support per-loop operations instead of per-wedge operations. Also note that the coverage transfer step is accomplished in the `wedgeEffect->begin(5)` call in line 38 which makes the appropriate render call (a screen-sized quad).

Per-loop wedge pixel shader

```
/ Pixel shader for wedges
struct appIn {
  float4 position: POSITION;
  float4 posData: TEXCOORD0;
  float3 posViewSpace : TEXCOORD1;
  float3 edgePoint0 : TEXCOORD2;
  float3 edgePoint1 : TEXCOORD3;
  float  r3DepthViewSpace : TEXCOORD4;
};

float4 main(appIn IN, uniform sampler2D depthMap   : register(s0),
                      uniform sampler3D visMapSameSide : register(s1),
                      uniform sampler3D visMapDiffSide : register(s2),
                      uniform float3 lightPos   : register(c0),
                      uniform float2 uvOffset   : register(c1)) : COLOR {

  // Calculate uv coords for screen position
  float2 screenSpaceUV = (IN.posData.xy/IN.posData.w) + uvOffset;

  float coverage = 0;

  // Sample depth value
  float depthValue = tex2D(depthMap, screenSpaceUV).r;

  // Find position in view-space of geometry behind this wedge pixel
  float3 geometryPos = IN.posViewSpace * (depthValue / IN.r3DepthViewSpace);

  // Find plane through edge and geometryPos
  float3 geoPlaneNormal = normalize(cross(IN.edgePoint0 - geometryPos, IN.edgePoint1 - geometryPos));

  // Does plane intersect with light sphere?
  // distance from plane to lightPos:
  float distLightToGeoPlane = dot(lightPos - IN.edgePoint0, geoPlaneNormal);

  // Project lightPos to geoPlane:
  float3 basePoint = lightPos - distLightToGeoPlane * geoPlaneNormal;

  // Find normal of lightPlane
  float3 lightPlaneNormal = normalize(geometryPos - basePoint);

  // Project e0 and e1 onto lightPlane
  float distToPlane = dot(IN.edgePoint0 - basePoint, lightPlaneNormal);
  float3 edgePoint0Proj = IN.edgePoint0 - distToPlane * lightPlaneNormal;
  distToPlane = dot(IN.edgePoint1 - basePoint, lightPlaneNormal);
  float3 edgePoint1Proj = IN.edgePoint1 - distToPlane * lightPlaneNormal;

  float3 baseToE0 = edgePoint0Proj - basePoint;
  float3 baseToE1 = edgePoint1Proj - basePoint;

  float dotProd = dot(baseToE0, baseToE1);
  float distToE0 = length(baseToE0);
  float distToE1 = length(baseToE1);

  float3 uvw = float3(abs(distLightToGeoPlane), distToE0, distToE1);
55    if (dotProd > 0) {
56      coverage = tex3D(visMapSameSide, uvw).r;
57    }
58    else {
59      coverage = tex3D(visMapDiffSide, uvw).r;
60    }
61
62
63    // Use coverage value calculated
64    float4 result;
65
66    // Let distLightToGeoPlane decide whether we are in inner or outer region
67    if (distLightToGeoPlane > 0) {
68      // Outer region - add coverage and tell that we are outside
69      result = float4(0,1,coverage,0);
70    }
71    else {
72      // Inner region - subtract coverage
73      result = float4(0,0,0,coverage);
74    }
75
76    return result;
77  }
```

The pixel shader is almost identical to the original algorithm, the only difference is how the calculated coverage value is used, (see line 67). If the fragment is in the outer penumbra region, we add the coverage value to the z channel and add one to the y channel. The y channel is later tested by the coverage transfer shader to determine whether we are inside or outside hard shadow. The method for doing this is described in section 4.4. If the fragment is in the inner region, we add the coverage value to the w channel.

The coverageTransfer pixel shader

```
struct appin {
 float4 position : POSITION;
 float2 texCoords : TEXCOORD0;
};

float4 main(appin IN, uniform sampler2D softDataMap : register(s0)) : COLOR
{
  float4 softDataValues = tex2D(softDataMap, IN.texCoords);

  float coverage = softDataValues.z - softDataValues.w;

  if (coverage == 0)
    coverage = softDataValues.x;
  else if (softDataValues.y == 0) {
    // Inside hardshadow
    coverage += 1;
  }

  coverage = saturate(coverage);

21    return float4(coverage, 0, 0, 0);
22  }
```

The coverageTransfer pixel shader calculates the LI value for a single silhouette loop. The LI value is the difference between the positive and negative coverage contributions (the z and w channels respectively, see line 10). If the LI value is zero, we assume that we are outside the penumbra area and we use the umbra (or hard shadow) value which is found in the x channel (line 13). If the LI value is different from zero we are inside the penumbra area and the y channel tells us whether or not we are in hard shadow as described above. When in hard shadow, we add one to the LI value (line 16).

6.5 Vertex shader shadow volumes

Here we show the vertex shader code which extrudes the VS shadow volumes as described in section 3.3.3.

```
struct appin {
  float4 position : POSITION;
  float4 normal : NORMAL;
};

struct vertout {
  float4 position : POSITION;
};

vertout main(appin IN,
             uniform float4x4 worldView       : register(c4),
             uniform float4x4 proj            : register(c8),

             uniform float4 viewSpaceLightPos : register(c12),
             uniform float lightRange         : register(c13))
{
  // Calculate view-space position and normal
  float4 viewSpacePos = mul(IN.position, worldView);
  float3 viewSpaceNormal = mul(IN.normal.xyz, (float3x3)worldView);

  // Calculate extrusion vector
  float4 extrusion = viewSpacePos - viewSpaceLightPos;
  extrusion.w = 0;
  float distToPoint = length(extrusion);
  extrusion = normalize(extrusion) * max(0, lightRange-distToPoint);

  // Calculate final position:
  float dotProd = dot(viewSpaceNormal, extrusion.xyz);
  float4 finalViewSpacePos = (dotProd<0) ? viewSpacePos : viewSpacePos + extrusion;

  // Project final view-space pos
  vertout OUT;
  OUT.position = mul(finalViewSpacePos, proj);
  return OUT;
}
```

Lines 22–25 calculate the extrusion vector for the vertex. Line 29 chooses between letting the vertex stay at its normal position or extruding it based on whether it is front or back facing to the light. Note that the extrusion is performed in view-space and the projection matrix is applied afterwards.

Chapter 7

Conclusion

In this thesis we have investigated the theoretical and practical aspects of both hard and soft real-time shadows, and we have implemented them in a full-fledged modern game engine. In this chapter, we present a compact summary of our key results and suggest future work that would speed up the presented soft shadow algorithms as well as expand the class of volume light sources that can be used.

7.1 Results

We have implemented the soft shadow algorithm suggested by Akenine-Möller and Assarsson in [AMA02], [AAM03] and [ADMAM03] and described in chapter 4. The algorithm calculates penumbra wedges for each silhouette edge from a given light source. The penumbra wedges are rasterized into the LI buffer using a pixel shader. The LI buffer holds a visibility factor for each pixel on the screen, and this factor is used in a subsequent pass to modulate the contribution from the light. The penumbra wedge algorithm implements a general solution for real-time soft shadows in simple scenes with arbitrary shadow casters.

Assuming spherical light sources we have developed a novel technique for calculating the coverage value as described in section 4.2. The coverage calculation is the most time consuming part of the pixel shader, but with our optimization the length of the pixel shader is reduced from 63 to 43 instructions. Furthermore, the amount of texture memory required for look-up tables is reduced from 2MB to 128KB.

We have identified several problems in the algorithm, the most important being that each wedge must be rendered seperately. This is a consequence of the need to split each wedge in halves as described in section 4.3.3. The large amount of render calls results in a severe CPU overhead that becomes the bottleneck in the algorithm for complex scenes. To overcome the CPU bottleneck, we have developed a novel version of the algorithm that is able to render all wedges in a silhouette loop as a single batch. The per-loop algorithm is described in section 4.4. As described in section 4.5, this version of the algorithm is GPU limited rather than CPU limited and as GPUs currently evolve faster than CPUs, we believe this is an interesting trait. In its current form the per-loop algorithm unfortunately only allows shadow casters that produce convex silhouette loops.

We have implemented both the original and our per-loop algorithm in our game engine, and we have tested the techniques on real game scenes as demonstrated in screenshots B.1 to B.5. The images render at interactive, but not real-time, frame rates[1]. To efficiently manage the large amount of shadow volumes in the game scenes we have developed several culling techniques which ensure that only visible volumes are processed, as it is described in chapter 5.

7.2 Future work

From our work with the soft shadow algorithm we conclude that it is not ready for general use in its current form. Further research is necessary before it can be applied to games the way stencil shadows are today.

Performance

The most important contribution to the algorithm would be to increase its performance. In section 4.3 we identified a list of problems with the soft shadow algorithms, some of which had to do with the limited blending functionality of current hardware. It is possible that new generations of graphics hardware will allow custom blending operations from within pixel shaders and if so, work can be done to optimize the calculations performed on the GPU. But since the original algorithm is CPU limited this will not solve the performance problems.

Before our per-loop algorithm can be put to general use it must be able to handle arbitrary shadow casters. At this time we have no ideas for a solution to this problem.

We have not tried to optimize our per-loop algorithm, but there are several ways to reduce the number of the coverage transfer passes and clear operations. One such way is to split up the channel used for hard shadow data. This should make it possible to render two loops for every coverage transfer pass, and it would effectively halve the number of clear operations as well as the number of executions of the coverage transfer pixel shader. Furthermore, the coverage transfer pass always renders a screen-sized quad, even if the affected area is just a small fraction of the screen. This is wasteful, both in regard to the pixel shader executions

[1]Between 1.5 and 5 FPS in a 640×480 resolution.

and the bandwidth usage.

Ellipsoidal light sources

Another valuable contribution to the algorithm would be to extend our new coverage calculation technique to other light shapes than spheres. Akenine-Möller and Assarsson have implemented three variants of the original algorithm which allows them to cast shadows from both spherical, rectangular and even textured rectangular light sources[AAM03]. Our optimized coverage calculation technique is only valid for spherical light sources which should not be a problem in most game settings. However it is possible, to extend our algorithm to handle ellipsoidal light sources. In section 4.2.1 we describe how to transform geometry into a space where the light source is a unit sphere through the use of a change of basis matrix. For spherical light sources we show how this can be reduced to a simple division by a scalar. For axis-aligned ellipsoids a similar reduction into a division by a *vector* is possible but for generally oriented ellipsoids the entire CBM must be applied. The cost of applying a full matrix to a point is four pixel shader instructions whereas division by a scalar or a vector is possible in one.

Ellipsoids provide a good approximation to many shapes, and we would with this addition be able to handle soft shadows from for example strip lights. Strip lights are common in many environments and are quite poorly approximated by spheres.

Appendix A

Working with 3d graphics

A.1 Terminology

In this section we briefly introduce some of the most common concepts and terms used in 3d computer graphics. An understanding of these concepts is crucial for reading this thesis.

Color buffer

At the most basic level, images in computer graphics consist of an array of colors - one color for each pixel in the image. Each color is typically represented using three color channels, R, G and B, describing the intensity of each of the main color components: red, green and blue. Optionally, the color can also contain an alpha channel that can be used for auxiliary information such as the transparency value of the pixel. 8 bits are typically used for each channel, making a color 32 bits in size: an optimal size for a CPU as it matches the cache boundaries nicely and makes it possible to store an entire color value into memory with a single assembly instruction. As a result, 32 bits are usually used - even when an alpha channel is not needed. In such a case, each color simply contains a padding byte where the alpha information would normally be stored. The array of colors is usually referred to as the *color buffer*.

Depth buffer

For 2d graphics a color buffer is really all we need, but in 3d it is possible for several surfaces to be projected and rendered into the same pixels in the color buffer. Then it is necessary to keep track of the spatial order of all such pixels so only the front most pixel is shown[1]. As it is impractical to keep track of the

[1]This is known as the *hidden surface removal* problem

spatial order of *all* pixels in real-time rendering a *depth buffer* or *z-buffer* is used to keep track of the current depth of all pixels in the color buffer. So a depth buffer is simply a buffer (with the same width and height as the color buffer) that stores the depth value of each pixel currently in the color buffer. Whenever a new pixel is about to be rendered into the color buffer, its depth value is first compared with the z-buffer - this is often referred to as the *z-test*. Only if the new pixel has a depth value closer to the viewer[2] than the current one is it allowed to update the color and depth buffer. Since a fairly high precision is needed to sort the pixels correctly, 24 or 32 bits are typically used for each pixel in the depth buffer.

Stencil buffer

Current graphics cards are also equipped with a so-called stencil buffer. A stencil buffer can be thought of as a kind of mask that can be set up to define which regions of the color buffer that can be rendered to. If f.ex. the stencil buffer is cleared to zero and a circle is drawn in the middle of it, setting the stencil value to one for all pixels that the circle covers, then the stencil buffer can later be configured only to allow draws in the color buffer in those regions where the stencil value is one. In effect, we have masked out a circular region of the color buffer. Typically 8 bits are used per pixel in the stencil buffer, and for performance reasons it is usually coupled with a 24 bit depth buffer, resulting in a 32 bit combined *depth-stencil* buffer.

As the stencil buffer is a very important tool for our shadow rendering, we will cover its use a bit more in depth in the following. There are quite a few parameters involved in setting up the stencil buffer and two of them are the *stencil reference value* and the *stencil compare function*. The stencil reference value is a constant 8-bit value that is uploaded to the graphics card, and for each pixel the corresponding value in the stencil buffer is compared with it using the specified compare function. The result of this comparison is a Boolean value. If this Boolean is true then we say that the pixel *passes* the stencil test - otherwise it *fails* the stencil test. As explained above it is only if the pixel passes the stencil test that it is allowed to be drawn into the color buffer. So, given that we have already placed some values into the stencil buffer somehow, those two parameters are all that is needed to use the stencil buffer for masking out certain areas in the frame buffer. In the circle example above we would set the stencil reference value to '1' and the compare function to 'equal'.

It is not possible to render values directly into the stencil buffer though. Its values are modified with certain *stencil operations* that happen when any of three different conditions are met. These three conditions are: when the stencil test

[2]Actually it is possible for the application programmer to define the z-test to be something else than the usual 'less-or-equal' but this was the original idea behind the z-buffer.

passes; when the stencil test fails; and when the stencil test passes but the z-test fails. For each of the three cases: PASS, FAIL and ZFAIL a stencil operation must be specified. The exact list of available stencil operations depends on the graphics card but the basic ones available on all cards are:

- KEEP - leave the stencil value untouched.
- ZERO - set the stencil value to 0.
- ONE - set the stencil value to 1.
- INCR - increase the stencil value by 1.
- DECR - decrease the stencil value by 1.
- REPLACE - replace the stencil value with the stencil reference value.

So, to set the values of the stencil buffer the graphics card is typically configured not to draw in the color buffer. Then pieces of geometry are rendered as normal, with the stencil buffer turned on and the stencil compare function set to 'always'. As a result, all rendered pixels will pass the stencil test just as long as they pass the z-test.

Homogeneous coordinates

The basic geometrical transformations used in 3d graphics are rotation, scaling and translation. Both scaling and rotation in 3d can be expressed through a 3x3 matrix, and in order to scale or rotate a 3d vector it is simply multiplied with the corresponding matrix. Translation, on the other hand, is achieved by *adding* the translation vector to the source vector. This inconsistency in how to apply transformations is unfortunate - we would like to be able to treat all three kinds of transformations in a consistent way, namely through a vector/matrix multiplication. To overcome this problem most graphics APIs, including both DirectX and OpenGL, work with so-called *homogeneous coordinates*. In homogeneous coordinates an extra 'w' component is added to the vector. In 3d this means expanding each vector from three to four vector components: x, y, z and w. Thus the transformation matrices must also be expanded from 3x3 to 4x4 if they are still to be multiplied to the vectors. As explained in [FvDFH90] chapter 5, using homogeneous coordinates and 4x4 transformation matrices we are now able to also implement translations as matrix multiplications. The main benefit of this is that we now can concatenate a whole string of transformations into a single 4x4 matrix and apply all the translations to a vector simply by multiplying it with this single combined transformation matrix. This is very useful in a 3d graphics pipeline.

If we *homogenize* a point given in homogeneous coordinates by dividing each component with the w component, we get a vector of the form $(x, y, z, 1)$ and we call the point (x, y, z) for the Cartesian coordinates of the homogeneous point. The fact that we're using homogeneous coordinates, can be fairly transparent to the user of a 3d API such as OpenGL or DirectX since we can just define our 3d vectors as usual and have the API assume a default w value of 1. On the other hand, if we explicitly specify a w value different from 1, or a vector is transformed by the graphics pipeline described below so that it gets a w value different from 1, then the API will homogenize the coordinate before rasterising the triangle in which it is used. As one cannot divide by zero, homogeneous points with a w value of zero cannot be homogenized. However, as the value diverges towards infinity we define all such points to be infinitely far away, displaced along a ray originating at $(0, 0, 0)$ and with direction vector (x, y, z). As a result it is common to represent *direction* vectors as homogeneous coordinates with w=0 while *point* vectors uses the standard representation with w=1.

Fragments vs. pixels

There is a subtle but important difference between *fragments* and *pixels*. A fragment is the projection of a small part of a specific triangle to a certain coordinate on the screen while a pixel is the smallest unit in the image. The final color of each pixel is a combination of the colors of all fragments that are projected onto the pixel. Sometimes the projection of a fragment onto a pixel simply overwrites its current color but it is also possible to have the graphics card *blend* the new fragment's color with the current color of the pixel instead. This technique is called frame buffer blending, and various settings on the graphics card exist that allow the application programmer to specify how this blending should be done. Examples of different blend modes are additive blending and various forms of alpha blending. In additive blending the color of each new fragment is simply added to the current color of the pixel, while in alpha based blending modes the alpha channel of the new fragment is used to decide the weighting of a blend between the new and current color of the pixel.

A.2 The graphics pipeline

Current 3d cards and 3d APIs such as DirectX and OpenGL represent the geometry they render as meshes of triangles. A triangle mesh is built from a collection of vertices (points) in 3d and is defined by edges that connect those vertices into triangles. In this section we cover the various spaces in which the coordinates for such 3d meshes can be defined. We also give an overview of the pipeline that converts the geometry from its mesh representation to its final representation in the color buffer as pixels.

Transformations

Each mesh is typically defined in its own local coordinate space called *object-space* or sometimes *model-space*. In other words, the coordinates of the vertices are defined relative to a local basis that can be oriented in a way that makes it practical to define and edit the mesh. In the case of the box shown in figure A.1, a local coordinate space is chosen so that the sides of the box are parallel to the coordinate axes, and consequently it is easy to define the coordinates of the mesh.

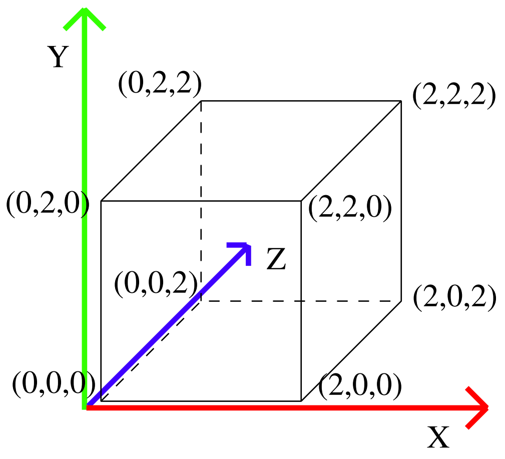

Figure A.1: Wire-frame box in object-space

A 3d scene generally consists of a number of meshes placed into a common space called *world-space*. To transform a mesh from object-space into world-space, a matrix called the *world matrix* is applied to all its vertices. The world matrix rotates, scales and translates the vertices into new coordinates, relative to the common basis. By associating multiple different world matrices to a mesh, it can be rendered multiple times into different positions and orientations in the scene. Each time a new *instance* of the mesh is said to be put into the world.

The world-space is infinitely large and only a small fraction of it can be visualized on a computer screen. To define what is seen, a camera is put into the world with a certain position, orientation and *FOV*[3].

The camera defines a third coordinate space called *camera-space* or *view-space* with origin at the camera position: a z-axis along the viewing direction;

[3]FOV is short for Field Of View and defines how wide the field of vision is. Typically, a FOV of 90 degrees is used, even though the human eye has a much wider field of view.

a y-axis along the 'up' direction of the camera; and an x-axis along the 'right' direction of the camera. From these three basis vectors it is possible to create a *view matrix* which has the effect of transforming a point from world-space into view-space. The view matrix is applied to all vertices after they have been put into world-space by their world matrices.

The view-space is still an infinite 3d space and thus, like the world space, only a small fraction of it can be visualized on a computer screen. The camera position, along with the FOV, defines an infinitely deep pyramid with a top at the camera position and spreading out, away from the camera, along the z-axis in the view-space. This pyramid is intersected by two planes, both orthogonal to the view direction, called the *near clipping plane* and the *far clipping plane*. The intersection between the pyramid and the near clipping plane defines a bounded 2d area that can be thought of as the computer screen, put into the 3d world. The far clipping plane is used to limit the visible region of the view-space to a closed volume, which is used in the projection step. The four side planes of the pyramid, along with the two clipping planes, define a frustum shaped volume called the *view frustum* and only geometry inside this frustum is deemed visible and will be projected onto the screen. Refer to figure A.2 for a visualization of the viewing frustum and the clipping planes.

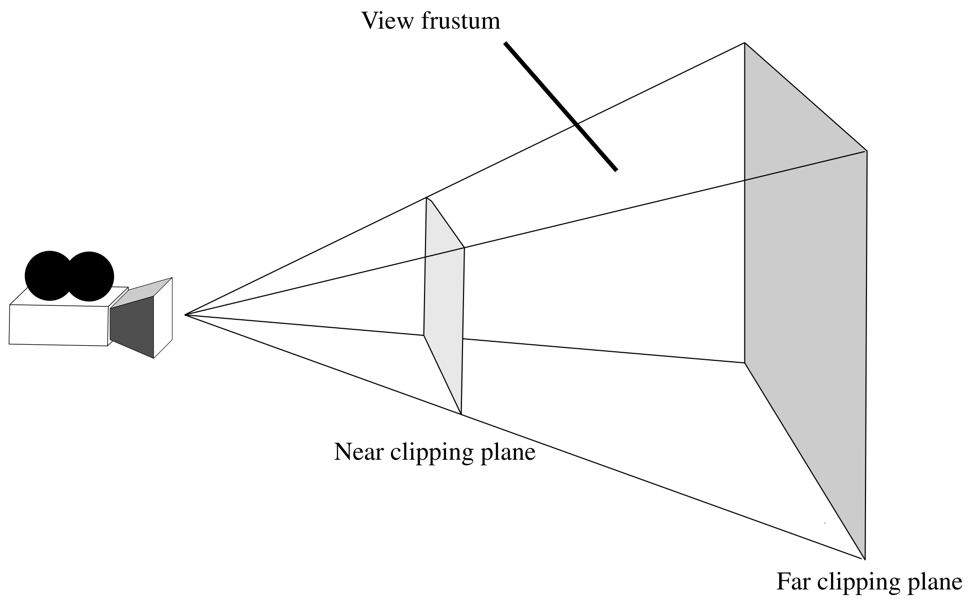

Figure A.2: The view frustum

Using a *projection matrix* the geometry is projected from view space onto the near clipping plane, which has the effect of scaling down points that are far away.

This is the way perspective is introduced to the image. Because it is possible for two different 3d points in the view frustum to be projected onto the same 2d point on the near clipping plane, the projection matrix also scales the z component of the point to be within the range [0..1]. A value of zero means that the point is on the near clipping plane, and a value of one means that it on the far clipping plane. This scaled depth value can then be used in the z-buffer test as described above. To have a fixed coordinate range of the projected points, independent of the FOV, the projection matrix also scales the x and y components of the points to lie within the range [-1..1], with the point (0,0) being at the center of the screen. So the total result of the projection matrix is to convert the view frustum into a bounded cubic space called *projected-space* with a fixed coordinate range as shown in a 2d top-down view in Figure A.3.

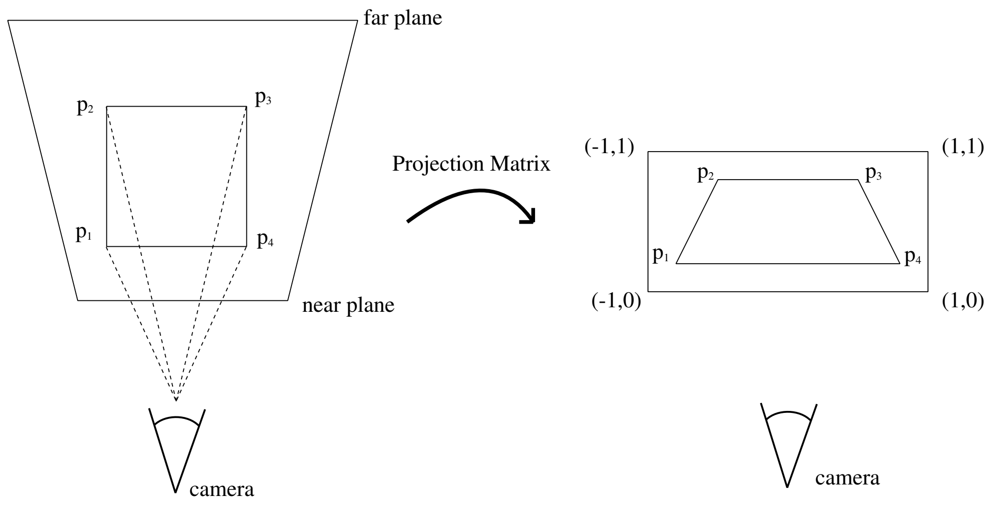

Figure A.3: Projected-space

The final conversion from projected-space into the actual color buffer is achieved by simply discarding the z component of the projected-space coordinate and then scaling the resulting 2d coordinate to the actual resolution of the color buffer - for example 1024x768 pixels.

To summarize: the meshes or objects in a 3d scene are initially defined in their own local spaces, and before they are actually shown on the computer screen they go through a chain of space transitions, as shown in figure A.4. In practice all these transitions happen in a single step, as the nature of matrices allows us to concatenate the world, view and projection matrix into one single matrix that takes a vertex all the way from object-space into projected-space.

Once the geometry has been projected onto the screen, each projected triangle is then *rasterized* into fragments and a color is calculated for each fragment, which

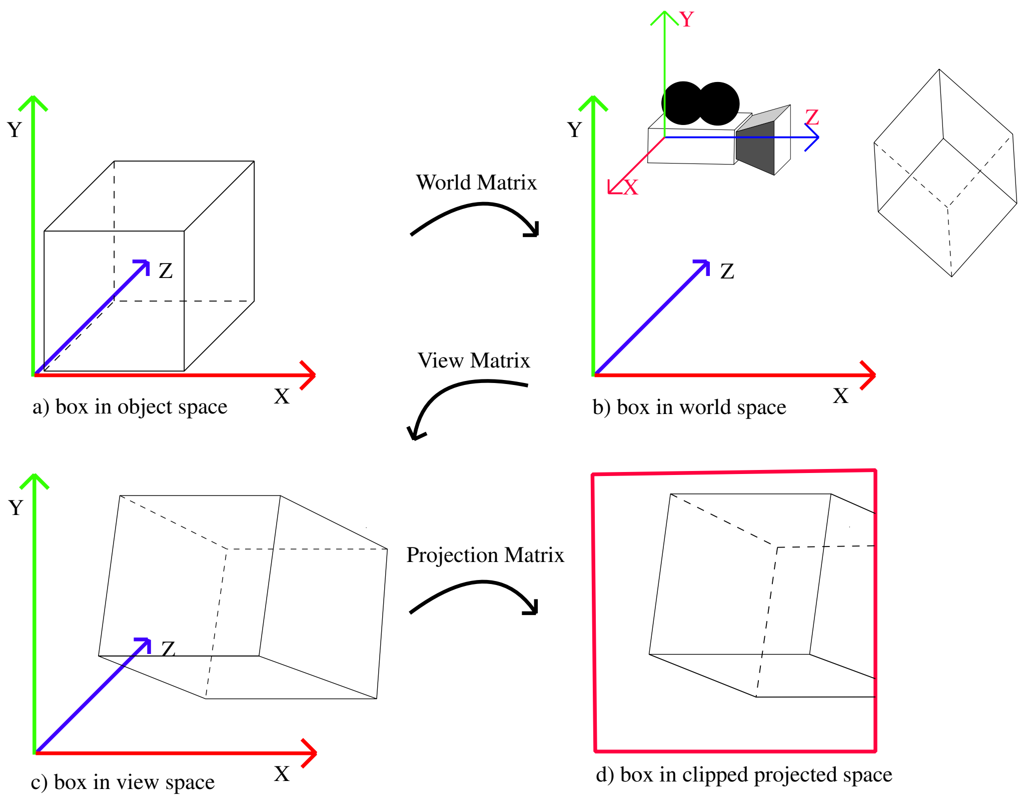

Figure A.4: Chain of transitions

is then finally written into the corresponding pixel in the color buffer. Once all fragments of all triangles have been rasterized into the color buffer the image is completed and can be shown on the screen.

Pipeline

The entire graphics pipeline can be visualized as a series of steps, each step being represented by a box as shown in figure A.5.

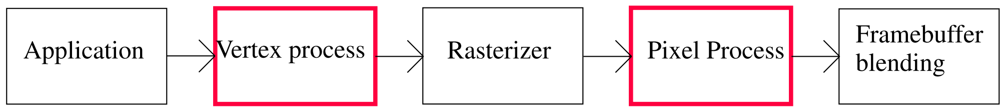

Figure A.5: The graphics pipeline

Each step is completely self-contained and depends only on its input. Therefore, if we wanted we could replace one or more of the boxes with our own custom components. As long as the output of our new components is valid as input to the next box in the chain everything would still work as it should. Previous graphics cards did not support custom components and operated solely through a so-called *fixed function* pipeline where the graphics APIs only allowed the programmer to set certain fixed parameters such as the different matrices, lights, materials etc. The actual steps of processing the vertices and shading the fragments were thus totally defined by those parameters, as described above.

However, on newer graphics cards it is now possible to install custom components for the vertex and fragment processing step and this gives the programmer the power to implement advanced vertex transitions and fragment shading programs, which is necessary for advanced graphics effects such as our implementation of soft shadows.

The custom components are called *vertex shaders* and *pixel shaders* and are small programs that are executed once per vertex or pixel respectively. The input to a vertex shader is the data for a single vertex of the mesh which is currently being rendered, as specified by the application programmer. This will typically consist of a position in object-space, a vertex normal, a diffuse color and one or more sets of texture coordinates but this isn't a requirement - the input can be anything that fits into a valid vertex format. As a position in projected-space is a crucial input to the rasterizer for it to be able to draw the triangles this is also a required output from any vertex shader. In addition to this position, the vertex shader can also output other things that are computed on a per-vertex basis.

The output from the vertex shader is, as explained above, the input to the rasterizer component, which will use the input positions to scanline convert the triangle into individual fragments. The rasterizer will also do a linear interpolation of all additional input values over the triangle surface, and for each pixel it will call the pixel shader with the interpolated values as input.

The pixel shader will then, based on the input, calculate a final color for the fragment and output it to the frame-buffer component, which will then blend it into the color buffer. The pixel shader can use a number of arithmetic instructions to do calculations on the input values as well as sample one or more texture maps for use in its computations - but in the end it *must* output at least one color, since that is required as input to the frame-buffer component.

Both the vertex shader and pixel shader component have an additional way of getting input, namely through a constant store where the application can upload settings that are constant for all pixels or vertices in a particular frame. The store consists of a number of 4d vectors with 32 bit float components and the size of the store depends on the hardware - but it is typically not very large[4]. Constant parameters include the matrices discussed above: the world matrix, view matrix

[4]Current hardware has a constant store size between 128 and 256 slots.

and projection matrix, as well as settings for materials and lights. All these things must be uploaded manually to the constant store by the application programmer. Figure A.6 summarizes the inputs to two shader components.

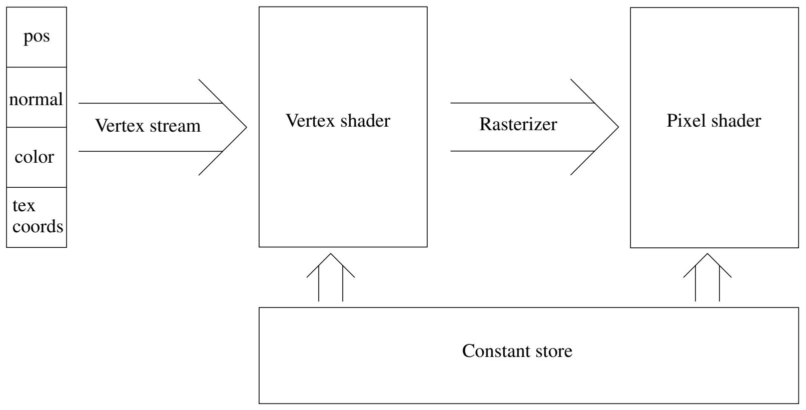

Figure A.6: The shader inputs

A.3 Vertex and pixel shaders

The introduction of vertex and pixel shaders gives the programmer much greater expressive power than the previous fixed function pipeline. Shaders are, however, still a very young technology with several severe limitations. An understanding of these limitations is necssary to be able to use them properly.

Shader instructions

A standard instruction set for shaders has not yet been established. With almost every new generation of graphics cards, new instructions are introduced that either expand on the core functionality or expose new features in the hardware. As a result both vertex and pixel shaders exist in many different versions, and while they are all backwards compatible it still means that shaders written for f.ex. ps2.0[5] cannot be run on hardware that only support an older profile. This makes it cumbersome to write software that both utilizes the latest features *and* runs on older hardware. Either a shader is written using the lowest possible version (possibly in a suboptimal way for the newest cards) or multiple versions of the same shader are written, one for each hardware profile that is to be supported.

[5]It is common notation to label the different versions of shaders with the prefix ’vs’ or ’ps’ for vertex shaders and pixel shaders respectively, followed by the version number.

Another limitation in current shader profiles is that there is a maximum number of instruction slots available for each shader. For vs2.0 the limit is 256 arithmetic instructions, and for ps2.0 the limit is 64 arithmetic instructions and 32 texture instructions. Also, since there is no true branching or looping in ps2.0 or vs2.0, loops must be unrolled with each iteration taking up a certain amount of the available instruction slots. With this in mind it becomes clear that heavy optimization is often required to keep a shader within its instruction limit.

In DirectX[6], the shaders are programmed through an assembly-like API. This API consists of a number of one-slot instructions and some macros that each take up multiple instruction slots. An example of a macro is the 'm4x4' instruction, which transforms a vector by a 4x4 matrix. This macro takes up four instruction slots since it can be implemented through four one-slot dot product instructions. However there is not a direct mapping between a shader in its assembly form and the actual implementation on the hardware, and the macros are not expanded by the runtime system. Instead, the shader is sent as a stream of tokens to a back-end compiler, implemented in the graphics driver. This compiler compiles the shader into native instructions, available on the particular graphics card, and runs it through an optimizer to do optimal register and instruction scheduling. If the hardware has native support for a macro, it will be able to execute it as it is, otherwise it will expand it into a series of simpler instructions.

The graphics driver usually does a good job of optimizing the shaders, and without very detailed knowledge of the underlying hardware there is not much one can do to facilitate the process, except keeping the shaders as short as possible. One way of reducing the instruction count is to exploit the fact that the GPU is a vector based processor, meaning that all instructions operates on 4d vectors with 32 bit float components. This is important to keep in mind when writing shaders, because often multiple scalar operations can be packed together in a single vector operation, if the operands are properly arranged in two vectors. As shown in figure A.7 it is possible to add four sets of two scalar values together in a single 'add' assembly instruction.

Using a technique called *swizzling* it is possible to access the individual components of each register. Swizzling refers to the ability to copy any source register component to any temporary register component, and it is done before the instruction that uses swizzling is run. An example of an instruction using swizzling is "*mov r1, r0.xxzy*", which has the effect of first creating a temporary register with both the x and y component set to r0.x, the z component to r0.z and the w component to r0.y - and then assigning this register to r1. Using explicit swizzling on both source and destination registers is good practice, since it provides optimization hints to the graphics driver. Thus it might be able to optimize the native code

[6]We have used DirectX 9.0b.

$$\begin{bmatrix} a_1 \\ ? \\ ? \\ ? \end{bmatrix} + \begin{bmatrix} a_2 \\ ? \\ ? \\ ? \end{bmatrix} = \begin{bmatrix} a_1+a_2 \\ ? \\ ? \\ ? \end{bmatrix}$$

a) Adding one set of floats.

$$\begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \end{bmatrix} + \begin{bmatrix} a_2 \\ b_2 \\ c_2 \\ d_2 \end{bmatrix} = \begin{bmatrix} a_1+a_2 \\ b_1+b_2 \\ c_1+c_2 \\ d_1+d_2 \end{bmatrix}$$

b) Adding four sets of floats.

Figure A.7: Vectorization of operations

for an instruction like the left one in figure A.7 if it is explicitly told that only the x components of the vectors needs to be added together. See [Rig02] for more information on how shaders operate on modern graphics hardware.

Working with shaders

Although the performance of current graphics cards seems impressive at first sight it is easy to write shaders that push them to the limit. It is especially easy to hit the instruction limit described above and optimization of shader code is therefore very important. As pixel shaders are run many times more than vertex shaders they should be the main target for optimizations and a good way to start is to make sure that nothing is calculated on a per-pixel basis that is actually constant for all pixels in the frame. Such values should be uploaded to the shader through the constant registers. Furthermore nothing that is constant, or can be interpolated linearly over an entire *triangle* should be calculated in the pixel shader, as it is better calculated on a per-vertex basis in the vertex shader and then interpolated by the rasterizer. Examples of values that are usually computed on a per-vertex basis and then interpolated over the triangle are texture coordinates and diffuse or specular colors.

Another thing that is important to understand when working with shaders is that vertex and pixel shaders work purely on the data they are provided with as input and that they cannot interact in any way with other vertices or pixels. For example it is not possible for a vertex shader to check the position of a neighbor vertex and use this information in its own calculations. On a similar note, a pixel shader cannot look up the color of another pixel and use this to decide its output. These are understandable and reasonable limitations, but they still put a limit on what kind of algorithms that can be implemented on the hardware through shaders. Skinned animation f.ex., where each vertex is animated using one or more transformation matrices without looking at its neighbors, is possible to implement in a shader while cloth animation currently isn't because it works through a spring system that relies on the ability to move neighbor vertices.

High-level shader programming

Traditionally shaders have been written in an assembly-like language, as described above. As a result, writing shaders was a cumbersome and slow process with lots of debugging required to make the shaders work properly. Recently Microsoft and nVidia has cooperated in developing a high-level language for programming shaders, making shader development much easier for the application programmer. nVidia has dubbed their language ’CG’, which is short for ’C for Graphics’ and Microsoft has dubbed their version ’HLSL’ for ’High Level Shading Language’. This has led to a great deal of confusion among developers, but in truth the two languages are close to identical and the two compilers can compile the same high-level shader code. The differences mainly lie in the runtime systems provided to manage the shaders. To avoid further confusion we will refer to high-level shader code in general as ’CG shaders’ for the remainder of this thesis. All the shaders we have written for our soft shadows implementation have been written using this high-level language.

Appendix B

Screen-shots

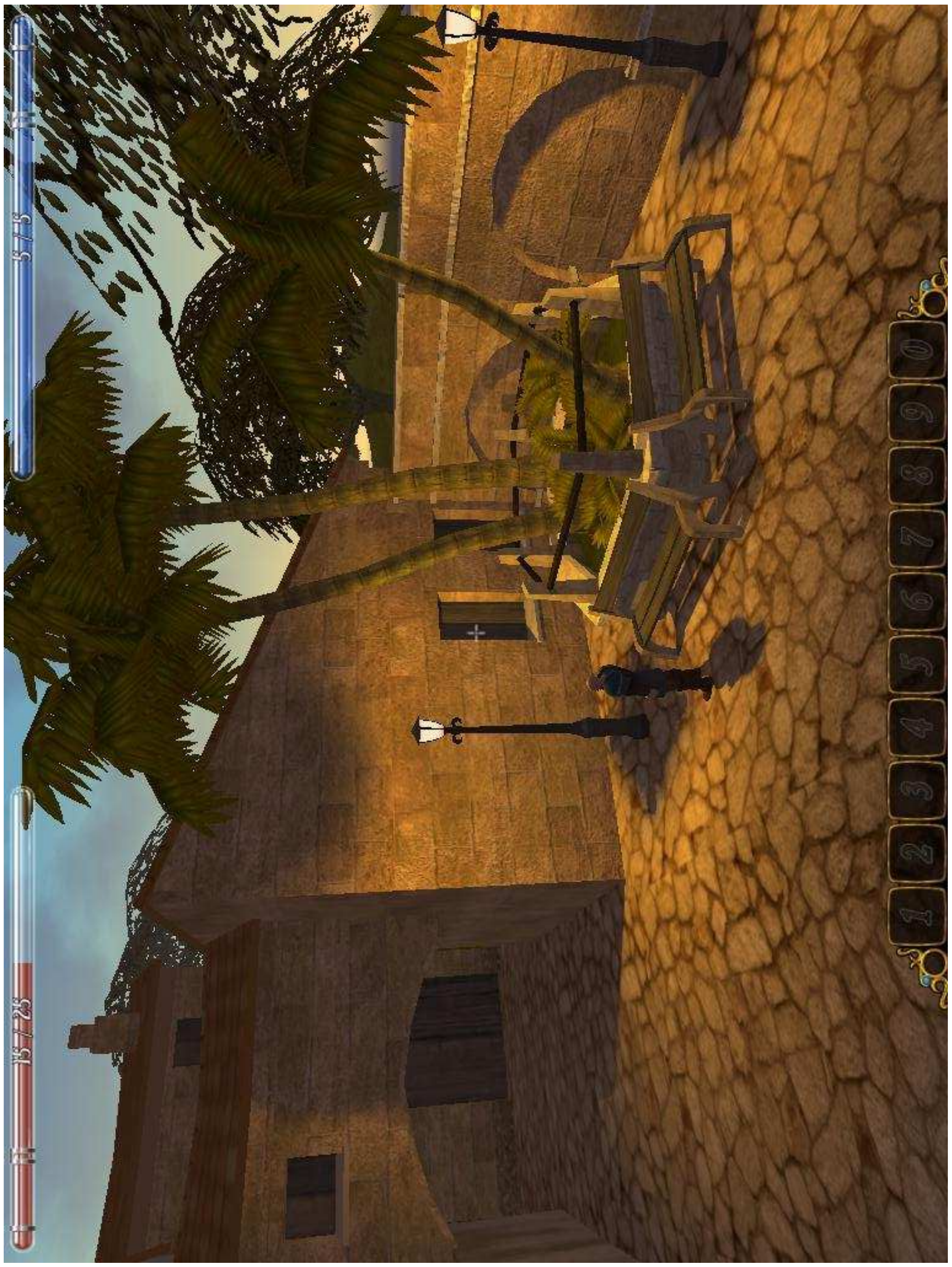

Figure B.1: Cosy back yard in the city

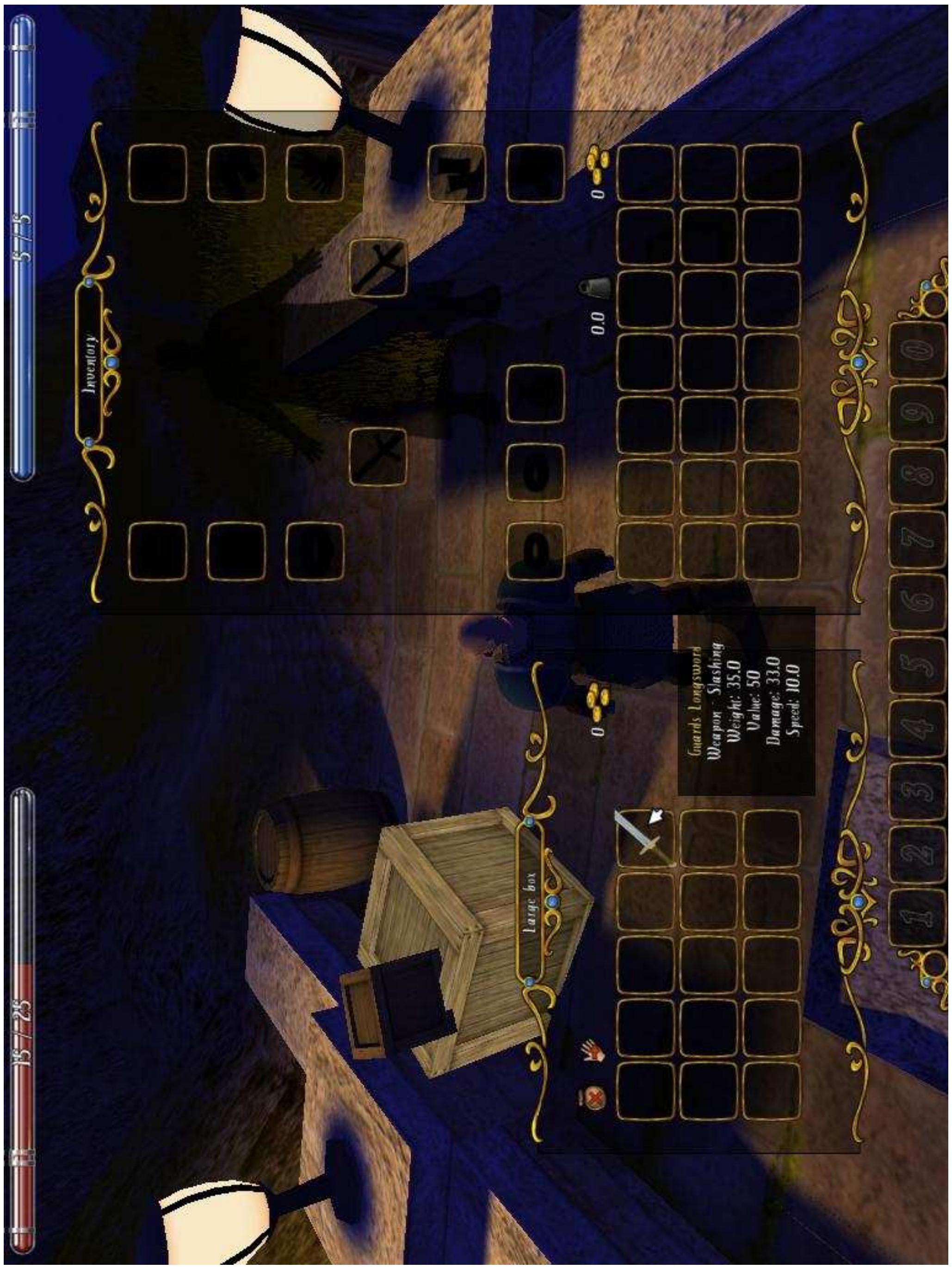

Figure B.2: Examining a box

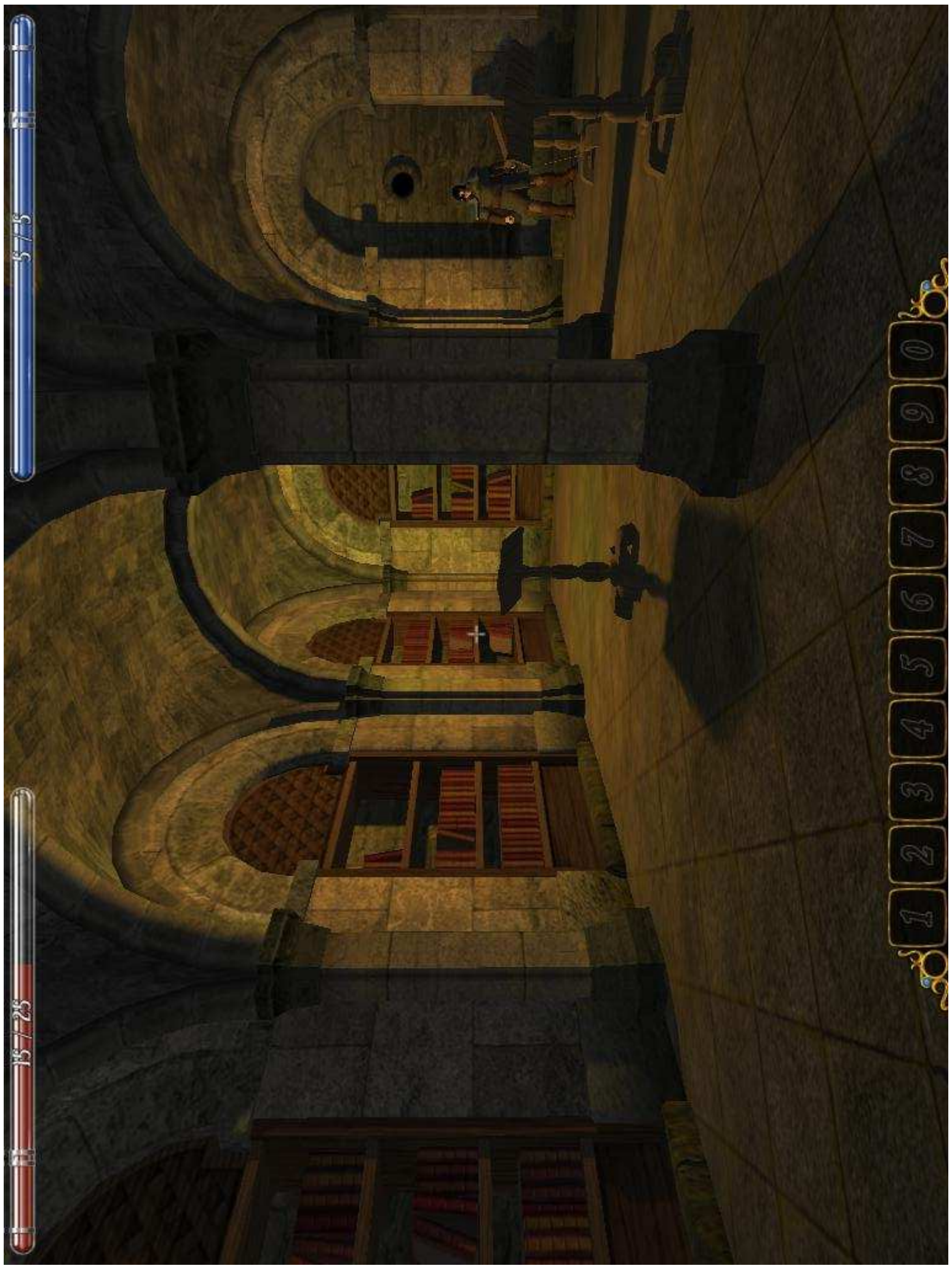

Figure B.3: In the library

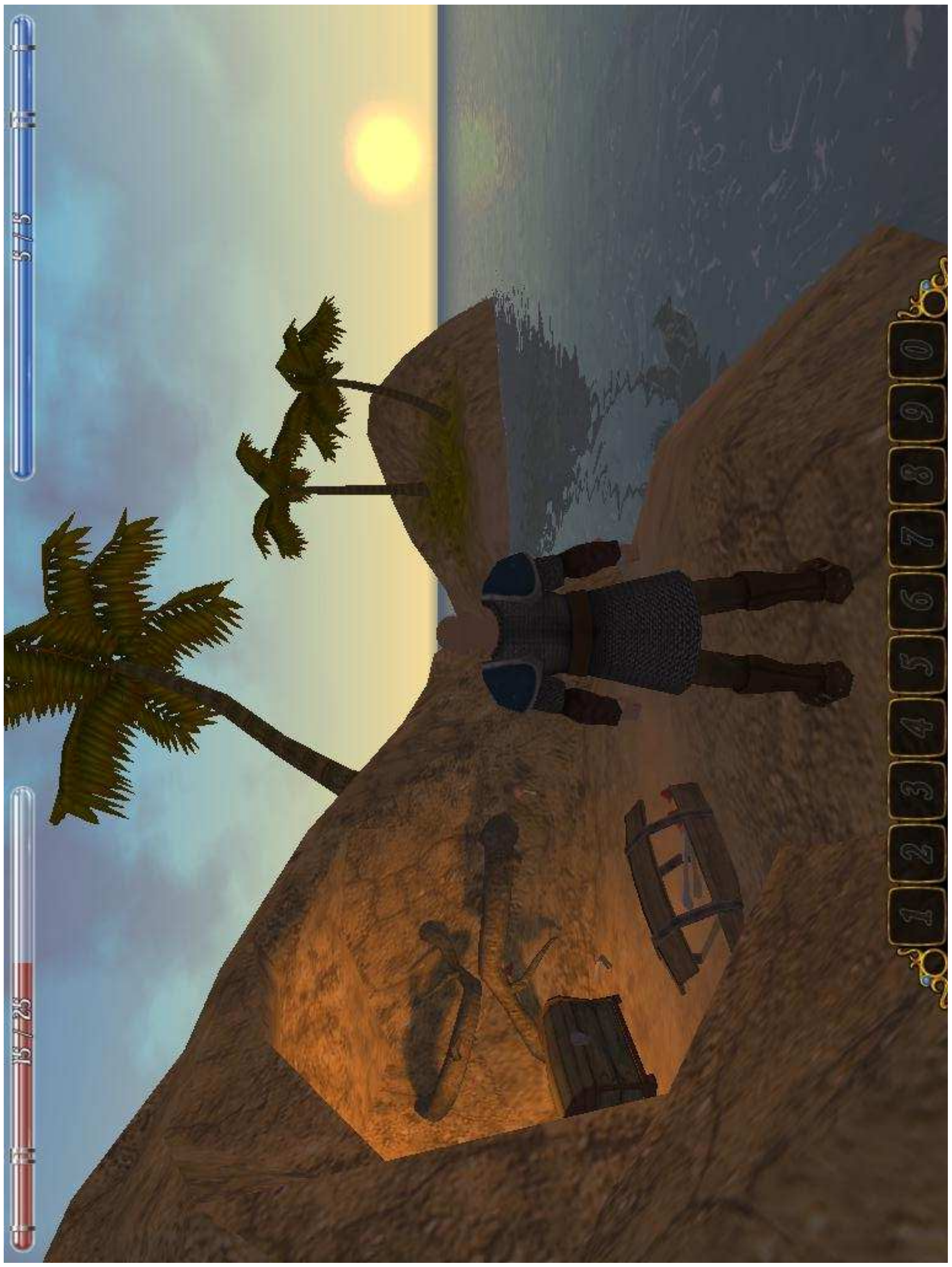

Figure B.4: Pirates’ treasure on a small island

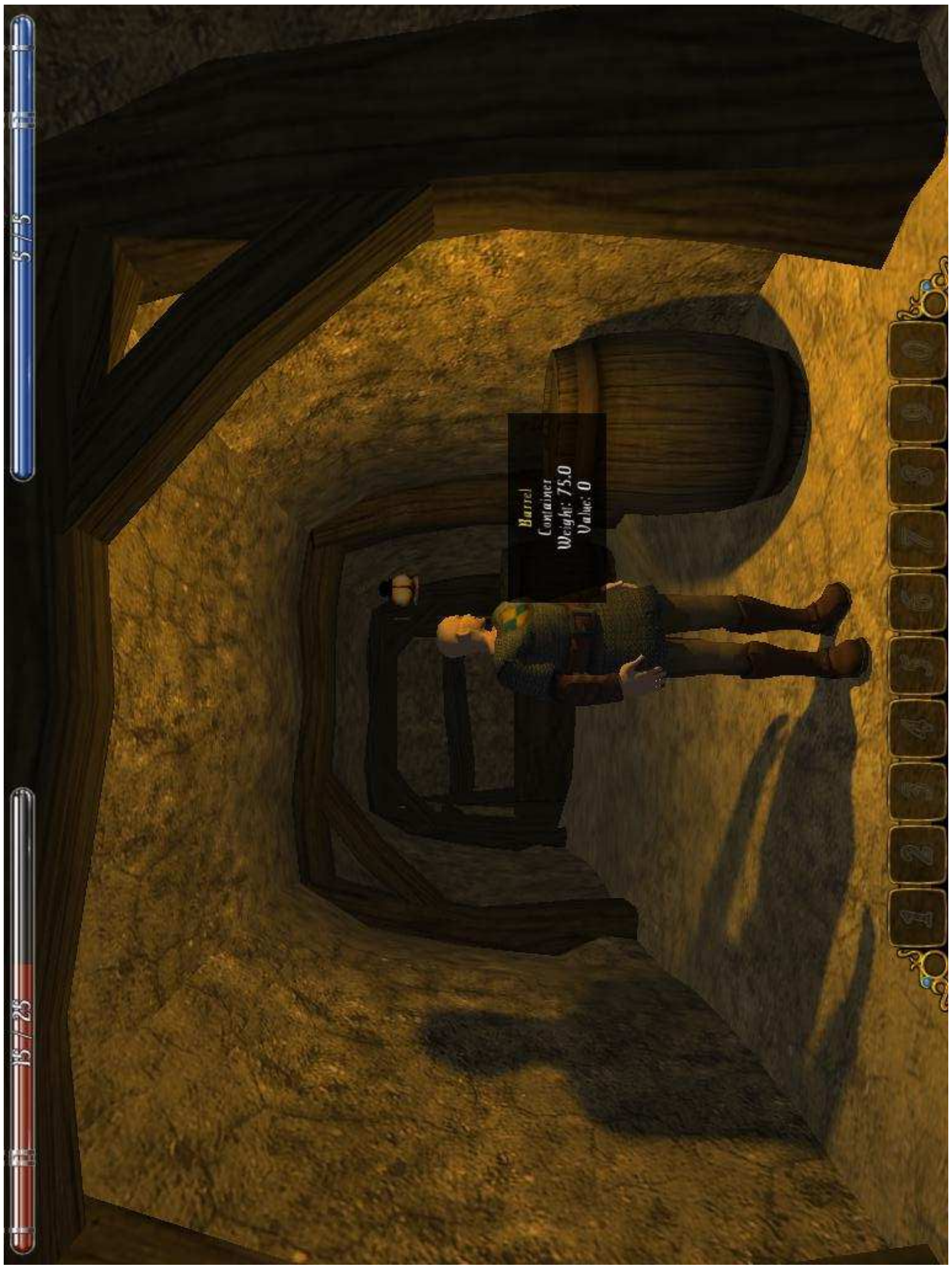

Figure B.5: In the mine

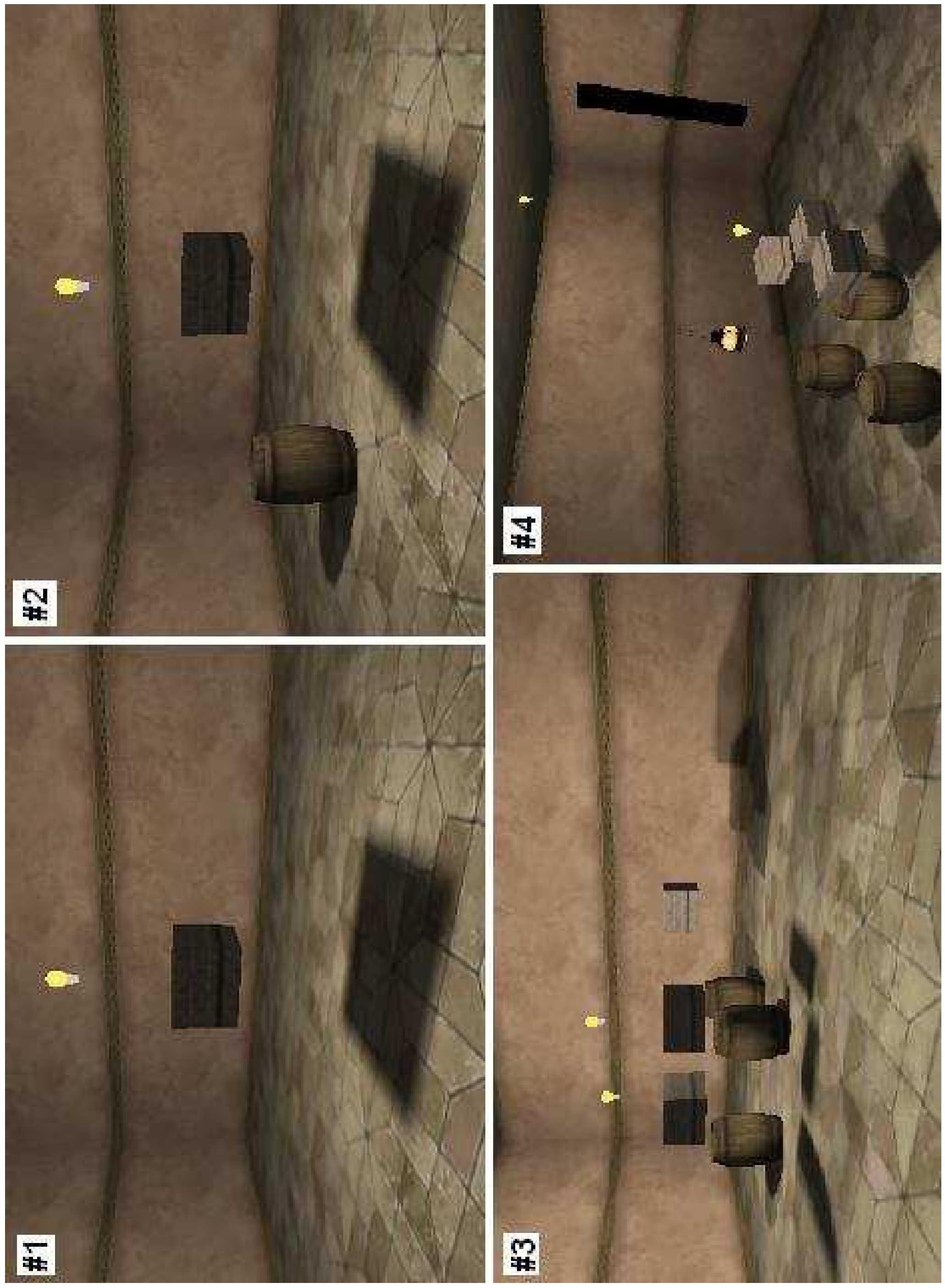

Figure B.6: Benchmark scenes

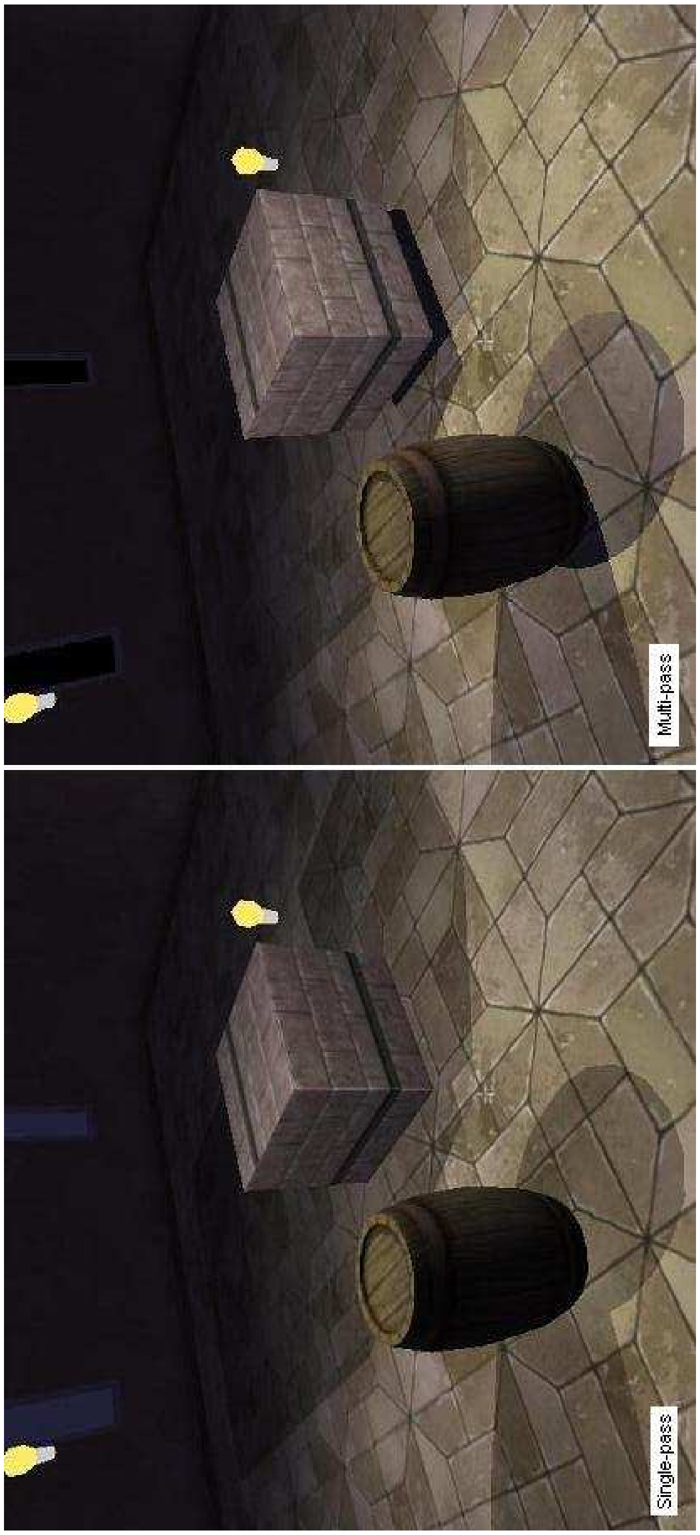

Figure B.7: Single-pass vs. multi-pass shadows

Bibliography

[AAM03] Ulf Assarsson and Tomas Akenine-Möller. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Transactions on Graphics (TOG)*, 22(3):511–520, 2003.

[ADMAM03] Ulf Assarsson, Michael Dougherty, Michael Mounier, and Tomas Akenine-Möller. An optimized soft shadow volume algorithm with real-time performance. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 33–40. Eurographics Association, 2003.

[AMA02] Tomas Akenine-Möller and Ulf Assarsson. Approximate soft shadows on arbitrary surfaces using penumbra wedges. In *Proceedings of the 13th Eurographics workshop on Rendering*, pages 297–306. Eurographics Association, 2002.

[AMH02] Thomas Akenine-Moeller and Eric Haines. *Real-Time rendering*. A.K. Peters Ltd., second edition, 2002.

[Ass03] Ulf Assarsson. *A Real-Time Soft Shadow Volume Algorithm*. PhD thesis, Chalmers University of Technology, 2003.

[Bli88] James F. Blinn. Me and my (fake) shadow. *IEEE Computer Graphics and Applications*, 8(1):82–86, January 1988.

[BT5x] Phong Bui-Tong. Illumination for computer generated pictures. *Communications of the ACM*, 1975x.

[Car00] John Carmack. Unpublished correspondance. 2000.

[Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 242–248. ACM Press, 1977.

[dBvKO00] M. de Berg, M. van Kreveld, and M. Overmars. *Computational Geometry - Algorithms and Applications*. Springer-Verlag Berlin, second edition, 2000.

[EK02] Cass Everitt and Mark J. Kilgard. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. Published at http://developer.nvidia.com, Mar 2002.

[ERCwn] Cass Everitt, Ashu Rege, and Cem Cebenoyan. Hardware shadow mapping. Published at http://developer.nvidia.com, Year unknown.

[FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 1990. second edition.

[FvDFH91] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics*. Addison-Wesley Publishing Company, second edition, 1991.

[Hei91] Tim Heidmann. Real shadows real time. *IRIS Universe*, 18:28–31, 1991.

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150. ACM Press, 1986.

[Mic] Microsoft Corporation. *Microsoft DirectX 9.0 Documentation*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/directx9cpp.asp.

[NRH$^+$77] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. Geometric considerations and nomenclature for reflectance. Monograph 161, National Bureau of Standards (US), October 1977.

[Rig02] Guennadi Riguer. Performance optimization techniques for ati hardware with directx 9.0. 2002. Available from http://www.ati.com/developer.

[SKv$^+$92] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 249–252. ACM Press, 1992.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274. ACM Press, 1978.

[Wlo03] Matthias Wloka. Batch, batch, batch: what does it really mean? 2003. Available from http://developer.nvidia.com.

[WND$^+$99] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, and OpenGL Architechture Review Board. *OpenGL Programming Guide: The official Guide to Learning OpenGL, Version 1.2*. Addison Wesley Longmann, Inc., Reading, MA, third edition, 1999.