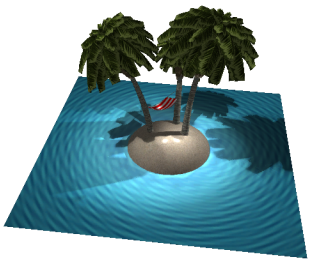


Real-time Shadow Generation for 3D simulations using modern hardware.

Maarten van Sambeek

Comittee:

Dr. J. Zwiers
Dr. M. Poel
Ir. D. Nusman

RE-liON

University of Twente
Enschede - The Netherlands

November 28, 2007

Preface

This thesis is based on the research I conducted from October 2006 to November 2007, performed at the Human Media Interaction chair at the University of Twente. It is the final report of my graduation project and marks the end of my Technical Computer Science studies.

I would like to thank Job Zwiers and Mannes Poel for supervising me during this project. I would also like to thank Daan Nusman, my supervisor at Re-lion, the company where this research was done. Re-Lion is a company that specializes in realistic real-time 3D simulations and serious gaming. Not only did they provide me with an assignment, but I also got to work with the newest available hardware.

I learned a lot about 3D graphics in the past year, but most of all I had a great time. Everyone at Re-lion was very interested in my work and helpful when I ran into problems. Steven, Chris, Alex, Paul, Oebele, Eddy and Bart, thank you guys!

Furthermore, I'd like to thank my (old) flat mates, my friends and family and everyone who helped me by filling in the survey or looking at 3D pictures with strange shadows.

Last but not least, I would like to thank Michou for reading my report over and over again, and supporting me the entire time. I couldn't have done it without you!

Maarten van Sambeek, Enschede, November 2007

Abstract

Shadows are important to create realism in 3D Simulations. They give extra information about spatial relationships of objects and they add to the overal atmosphere. Several techniques to create shadows exist, each with their advantages and disadvantages. Modern graphics hardware can process more graphics data than every in real time. Shadow algorithms that required preprocessing or could not be used in real-time now be implemented efficiently using the modern GPUs. With the new geometry shader, processing of polygons can be moved from the CPU to the GPU.

In this thesis real-time shadow generation using the Re-lion Renderer2 engine is presented. Several existing techniques have been adapted to make use of the capabilities of modern graphics hardware. These techniques have been implemented in a demo framework in the form of a shader library.

To compare the performance and quality of the techniques, they were evaluated and compared in the areas of performance, shadow quality and memory usage. Finally, recommendations are made to select the right shadow technique for the right situation.

Contents

Chapter 1

Introduction

1.1 3D Simulations

3D Graphics is an area of computer science that has gained more and more ground over the years. The graphics in modern computer games now look more realistic than ever. In these games a virtual world is presented where a player has almost as much freedom to move as in the real world.

3D Simulations is a field which is closely related to games. Simulations are meant to train or educate the user in a certain field. Until recently, the purpose of the simulation was more important than the appearance of the virtual world. The educational element of simulations was the most important aspect, so the 3D graphics used were mostly functional and not very detailed.

Nowadays modern game technology is used more frequently in 3D simulations . With this technology a new level of realism can be reached, making it easier for the user to perceive the virtual world as real. Because simulations are designed to simulate a real world situation, this is desirable. The new term for 3D Simulations that match the quality of modern games is *Serious gaming*.

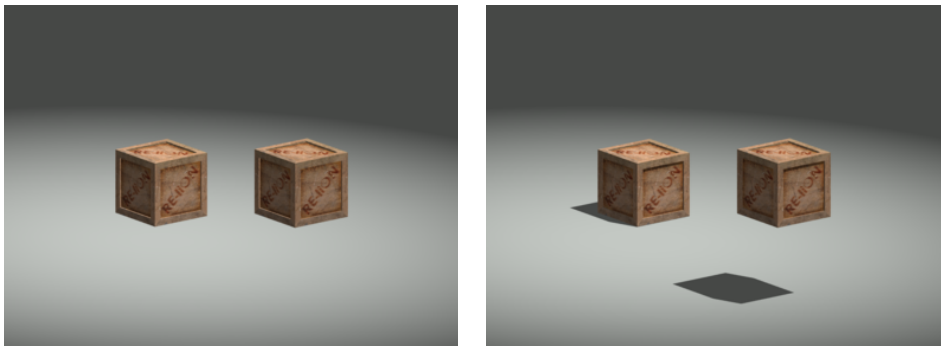
1.2 Shadows

An important part of creating this realism are shadows. 3D simulations try to approximate how humans perceive the world around them. In the world around us, light is cast by the sun and other light sources. The places this light cannot reach is shadow. This is why realistic simulations should have shadows.

But this is not the only reason for using shadows in simulations. Shadows also give important clues about the world that is visualized, as described in [HLHS03].

Position and size

In figure 1.1a two boxes are sitting on a gray plane without shadows. The picture is only a 2D view of a 3D scene. This means that one dimension of information of the scene is lost. In figure 1.1b the same scene is shown. This time the boxes cast shadow. The right box appears to be floating above the gray plane. Some of the information that was lost in the 2D projection of the 3D scene is regained. From the shadows one can deduct the position of the light source, and from the position of the light source, the position of the boxes in the 3D scene can be derived.



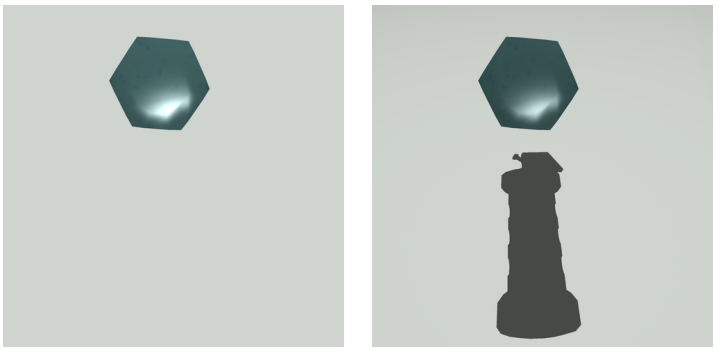
(a) Two boxes without shadows (b) Shadows show their real positions

1.1: Shadows give information about the size and position of objects.

Figure 1.1b shows that the right box floats above the plane, and that it is situated closer to the camera than the left box. Thus the right box must be smaller in comparison to the left box. This means that shadows also give information of the size of objects.

Shape

Another aspect of objects that can be lost in the 2D projection of the 3D scene is information about the shape of an object. Figure 1.2a shows a simple object that looks like a hexagon. When light is emitted from a light source above the object so it casts a shadow, more information about the shape of the object is given.



(a) A simple shape (b) Shadows give more information

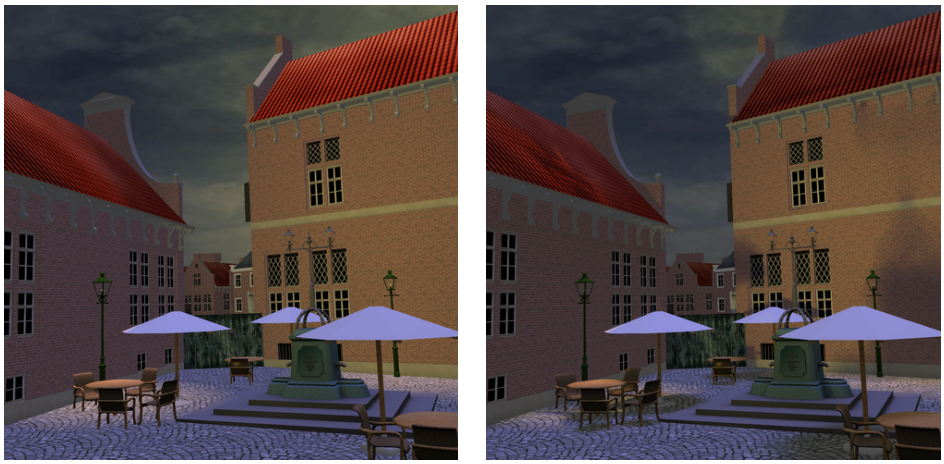
1.2: Shadows give information the shape of objects.

Visibility

Simulations are often used for training purposes; By using a simulator people are put in problematic situations to train skills. Most of the simulations are based on visual skills. In a simulation without shadows all objects are equally visible. In reality objects can be hidden in the shadows, making it harder to find them. Shadows in the simulation are needed to simulate situations like this.

Atmosphere

In a simulator that tries to reach a high level of realism, the atmosphere or feeling of a scene is important. This atmosphere is controlled by the user's perception. Users get 'sucked into' a simulation if the atmosphere is right. As in movies and in video games lighting effects in simulations play a very important part in creating the right feeling. Shadows contribute to this feeling, adding some depth to the scene. The way shadows influence the atmosphere cannot be measured objectively but, as can be seen in figure 1.3, shadows add a lot to the feeling of the scene.



(a) A scene without shadows (b) The same scene with shadows

1.3: Shadows can add to the atmosphere of the scene.

1.3 Shadow generation problems

Over the years many techniques have been developed to generate shadows. Almost every game on the market today uses some sort of shadow to make its virtual world more realistic. Why is creating shadows in simulations still a problem?

Shadows are a global effect. This means that to determine if a polygon is in shadow, information about the entire scene is needed because every object in the scene can be a light blocker for the polygon. Current graphics hardware draws polygons in a highly optimized way, one at a time. Whenever a polygon is drawn by the hardware, only the information about *that* polygon is available.

This is why there is need for a trick to have access to the necessary information about the significant polygons when rendering. Every shadow technique tries to solve this problem in its own way, resulting in either quality loss or an increase of rendering time.

Most computer games take this quality loss for granted by optimizing the techniques only in game specific situations. In a car-racing game for example, shadow resolution does not need to be high because the player will never be extremely close to a shadow receiver. Also, the camera position in car games will always be located just above the road so optimizations can be made for that specific camera position too. Another optimizations could be made by only creating shadows from the sun, which always has the same relative position to the car.

In a simulation engine that is meant for multiple types of simulations, no assumptions can be made about camera or light positions. This is why most of the optimizations used in games cannot be used in this field.

Older graphics hardware used shaders to transform vertices and calculate pixel colors. All these shaders could do was transform the data that was provided to them by the application. Every vertex that went into a shader was transformed and came out again at the other side. The shader could not generate extra vertices, nor could it destroy the unnecessary vertices.

Geometric algorithms for shadow creation depend on the adding or removing of vertices from a model. With old hardware this had to be done by the CPU. After the processing of the model the data was uploaded to the graphics card to be rendered. This happened every frame. A real-time application usually runs at frame rates higher than 20 frames per second, which leads to a lot of data that has to be uploaded to the graphics card. A CPU can only perform one task at a time. This means that the vertices were processed serially. Graphics hardware is optimized to process data in parallel.

1.4 New technology

As mentioned earlier, graphics hardware capabilities have improved significantly over the years. Modern GPUs can process millions of polygons per second. The increase in speed and processing power allowed for more complicated effects, but there was still a drawback: graphics hardware could only transform data. This meant that no new data could be created by it.

This has changed with the latest generation of hardware. Instead of only being able to transform data, new hardware can also dynamically generate or discard data.

To make this new kind of data processing possible, a new type of shader was introduced: The geometry shader [Geo07]. This shader is executed after the vertex shader and gets a primitive as input. A primitive can be a point, a line, a polygon or each of these with adjacency information. It can discard this primitive, create more primitives using the original data, or just keep the original primitive. Also, geometry shaders are run in parallel, making it possible to process many polygons at the same time. While this all happens the CPU can use its processing power for other purposes. This means that an application has more processing power available and the amount of data that has to be uploaded to a graphics card decreases drastically.

Geometry shaders move a lot of work away from the CPU onto the GPU. This means that techniques that needed preprocessing or a lot of CPU processing power using old hardware can now be done in real-time using the new graphics hardware.

1.5 Re-lion

The research described in this report was performed at Re-lion: a company that specializes in creating 3D simulations and serious games. Visual realism is an important aspect in these simulations. To visualize 3D graphics Re-lion uses an in-house developed engine named Lumo renderer. Recently this engine was completely redesigned. The new engine named Renderer2 focuses on the low-level aspect of 3D rendering. Simulation applications are responsible for the high level operations like scene management and animation.

The only shadows that were implemented in the simulations that re-lion created until now, were static shadows. Static shadows are generated offline and added to the textures of the scene. During the simulations these shadows never change. This means that if a dynamic object moves into a static shadow, it will still appear as if it is situated in light. To increase realism in these simulations, support for dynamic shadows is desirable.

Renderer2 is intended to be a generic engine. It is used in all kinds of simulations of all sizes and complexities. This is why a shadow method is needed for all these different situations.

Since 3D simulators consist of hardware and software, shadow methods can make use of the capabilities of the latest generation of hardware; no support for older hardware is necessary.

1.6 Research

Because no support is needed for older hardware, this research can focus on using the newest generation of GPUs and the new possibilities that they provide.

The new capabilities, combined with the need of a shadow implementation in Renderer2 have lead to the following question:

> *Which existing shadow techniques, when adapted for using the capabilities of modern hardware, produce the best results in the areas of performance and shadow quality and how can these techniques be implemented using the Renderer2 API?*

These adaptations for the use of the capabilities of modern hardware can be:

- Data an application has to provide to the graphics hardware. In the ideal case, an application needs to provide geometry data to the graph-

ics card at initialization time. This is possible if this data is only processed by the GPU during the simulation. Some shadowing techniques have to process the geometry data every frame. This processing of the geometry results in vertices being added or removed. Since this is was possible on older hardware, these calculations calculation were ususally done on the CPU. Every frame, this preprocessed data needed to be uploaded to the GPU. This took up a lot of bandwidth and slows down the application. With the new hardware, this preprocessing can be done on the GPU.

- Distribution of workload between the CPU and GPU. CPU processing power is needed to run a simulation. When a shadowing algorithm also uses a lot of CPU processing power, application performance may suffer. Moving tasks from the CPU to the GPU reduces the amount of CPU processing power needed by shadow algorithms, thus leaving more for the simulation.

- Real-time performance of the selected techniques. The different techniques generate shadows of different visual quality. Shadows that look better tend to cost more processing power. The different techniques will be evaluated on their performance versus the quality of the shadows.

In the next sections, previous research in this field will be summarized.

1.7 Shadow algorithms

Over the years, many shadow algorithms have been proposed. The most important real-time shadow techniques can be found in [WPF90] and [HLHS03]. In this research the focus in on two groups of shadow algorithms:

- *Image based algorithms.* For these algorithms, the scene is rendered to one or more textures. These textures are used in a final pass to determine what areas of the scene are in shadow or in light. Since textures cannot be infinitely large, image based techniques suffer from resolution problems; textures are stretched out over the scene, causing visual artifacts. Image based techniques scale well with scenes size but tend to use a lot of memory for the textures that is rendered to. Image based techniques are usually derived from shadow mapping [Wil78].

- *Geometry based algorithms.* These algorithms create or transform the geometry of the scene to determine what areas of a scene are in shadow. One example of a geometry based algorithm are projected shadows

where the scene geometry is projected onto a ground plane to visualize its shadow [Bli88]. Another geometry based algorithm creates volumes that contain the areas of the scene that are in shadow: the so-called shadow volumes [Cro77]. These volumes have to be calculated every frame when a light a dynamic object moves. Because this calculation requires polygons to be added to the geometry, this could only be done on the CPU. This is why geometry based techniques did not scale well with scene size; the bigger the scene, the more calculations needed to be done. These calculations used up the time that was needed to do the simulation calculations.

Both types of algorithms have their advantages and drawbacks. However, the second group of algorithms will greatly benefit from the new geometry shaders because the calculations that slow these algorithms down can now be implemented on the GPU.

1.8 Implementation

Several shadow techniques were implemented for this research. This implementation was done by using the Renderer2 API. At the start of this research, Renderer2 only supported the Direct3D 9 API. Unfortunately the new capabilities that are exposed by modern hardware are only supported in Direct3D 10 and OpenGL. The Renderer2 API is designed to support multiple graphics APIs through a driver model. To support the new techniques, a driver had to be implemented for Direct3D 10 or OpenGL. Since converting the driver from Direct3D 9 to Direct3D 10 is less work than creating a OpenGL driver from scratch, a Direct3D 10 driver was created. This driver was only to contain the core functionality, but while implementing shadow techniques more and more functionality was needed and thus implemented.

Because Renderer2 is a low level API it does not provide scene management. This is why a demo application was created to demonstrate the different shadow techniques. This application is responsible for the loading and saving of models and scenes. Simple scene manipulation like moving objects, cameras and lights can be done using the application.

In this demo application the different shadow techniques were implemented using shaders. This resulted in a shader library that can be used with Renderer2.

1.9 Evaluation

The implemented shadow techniques were compared on two areas. The first area, performance, was tested by looking at the frame times and the usage of different parts of the GPU. This was done by rendering a number of test scenes and measuring the frame times.

For every test scene, the amount of time spent in the different parts of the GPU was also measured. Using these results, the bottlenecks in the render pipeline can be found for each technique.

Another way to compare shadow technique performance is by looking at the amount of memory a technique uses. In situations where the available graphics memory is low, because a lot of textures are needed for the scene, a shadow technique is needed that does not require any extra memory.

The second area that the shadow techniques were compared in, was shadow quality. Rendered shadows were compared to a reference image which contained the correct umbras and penumbras. The difference between the reference image and the rendered shadows are a measure for shadow quality. The smaller the difference, the higher the shadow quality.

For realistic simulations, it is important that the shadows look and feel real. This cannot be measured objectively. A number of people was asked to rank images that were rendered using the shadow techniques according to realism. The results were used to analyse what technique is perceived as the most realistic technique.

Chapter 2

Shadowing techniques

Over the years, numerous shadowing techniques have been proposed. Many of these shadowing techniques are mentioned in [WPF90] and [HLHS03]. These techniques can be divided in real-time and pre- or postprocessing techniques. For this research only real-time techniques are important, so a selection of the available techniques is made. In this chapter, this selection of techniques will be presented.

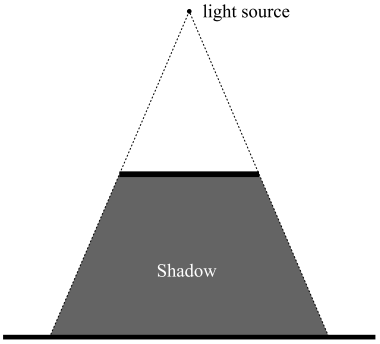
To use these techniques, some information about the geometry of shadows is necessary. This can be found in the following section.

2.1 Shadow geometry

Shadows are the areas of a scene that receive no light from light sources because the light is blocked by an object. Objects that block light cast shadow. From now on these objects are referred to as *shadow casters*. Objects that receive shadow will be referred to as *shadow receivers*. Note that a shadow receiver can also be a shadow caster and vice versa.

Point lights

In figure 2.1 a shadow is shown that is cast by an infinitely small point light. The light source emits light. This light falls onto the shadow caster, and is blocked by it. The objects behind the shadow caster will not receive light, so they are in shadow.

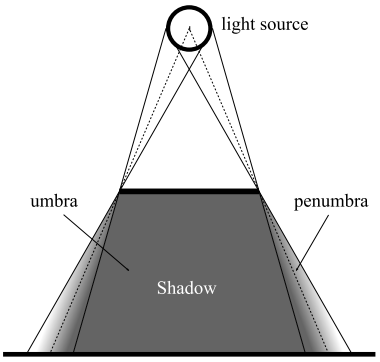


2.1: Point light source

The light source in figure 2.1 casts shadows with hard shadow borders. This is because the light source is a point light, an infinitely small point that emits the light. From any part of the scene, the light of the light source is either totally blocked or totally visible. This is because an infinitely small light source cannot be partially visible, since it is infinitely small. These point lights only exist in theory; in reality all light sources have an area that emits light.

Area lights

Figure 2.2 shows an area light source. The entire surface of the spherical area light source emits light on the scene. Since this surface isn't infinitely small, objects can be partially in shadow; a shadow blocker can block the light that is emitted from part of the light source. The shadow receiver behind it will not be entirely in shadow. This part of the shadow that is not entirely in the shadow is called *penumbra*, while the part of the shadow where the light source is totally blocked is called *umbra*.



2.2: Area light source

This concludes the explanation of shadow geometry. In the following chapters shadow techniques from the literature are discussed.

2.2 Shadow mapping

2.2.1 Algorithm

When looking at a scene from the position of a light source, all visible objects are in light. All other objects are in shadow. *Shadow mapping* is based on

this principle.

Shadow mapping was first proposed in [Wil78]. The technique is meant to create shadows for spot lights and directional lights, because it is possible to calculate a view and projection matrix for these types of lights. For point lights this is not possible, because they do not have a field of view. It is however possible to simulate a point light using multiple spot lights on the same position, pointing in different directions.

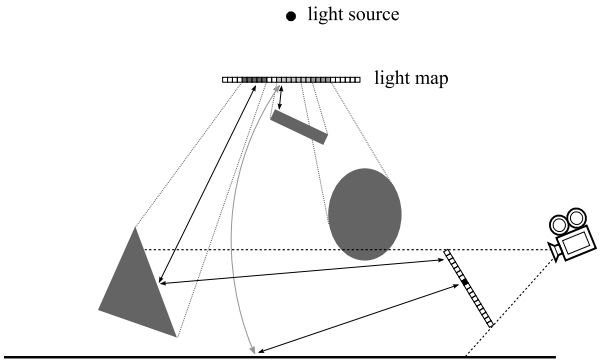
The shadow mapping algorithm for a single light source is as follows: Create a view matrix V_L and projection matrix P_L for the light source. In case of a directional light, the projection matrix will be an ortogonal projection. Using these matrices, the scene is now rendered from the position of the light source. Instead of storing the color values of the rendered geometry, the distance of the geometry to the light source is stored. The result of this pass is stored in a texture, the so-called light map.

After generating the light map, the scene is rendered one more time, now from the camera position. The light map is now projected onto the scene, and used for depth comparison: for every rendered point p, its position p_L in the lights projected space is calculated. Using this position, the texture position in the light map (u_p, v_p) and the distance to the lightsource in light projective space z_p for this point can be calculated.

$$\begin{aligned} p_L &= pV_LP_L \\ (u_p, v_p) &= (0.5 + \frac{p_L.x}{2p_L.w}, 0.5 + \frac{p_L.y}{2p_L.w}) \\ z_p &= \frac{p_L.z}{p_L.w} \end{aligned}$$

The value z_L of the light map at position (u_p, v_p) is fetched. It represents the distance of the first geometry blocking the light. When z_L is smaller then z_p, it means that p is not visible from the light source, so it must be in shadow.

Figure 2.3 shows this algorithm graphically. The light-blocking objects are rendered to the light map. Next, while rendering from the camera perspective, all visible geometry points are checked with the light map. The figure shows two rays from the camera, one looks at a point in shadow, the other one looks at a point in light.



2.3: Graphical representation of the shadow mapping algorithm

The above algorithm can be extended to support multiple light sources. For every light, a light map has to be created. In the final render, the point has to be looked up in all the light maps. Shadow mapping is a multi-pass technique. The scene needs to be rendered at least once for every light source and once more for the final render.

Benefits

Since shadow mapping does not depend on processing the geometry of the scene, scene complexity has no influence on performance of the algorithm.

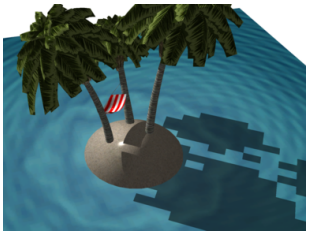
Most modern graphics hardware is optimized for rendering light maps. The pass that is done to create the light map only needs depth information. This means that color and lighting information do not have to be computed for this pass, which allows for a speed increase.

Shadow mapping can be implemented easily using projective texturing. It can be done in just a couple of lines of shader code.

Problems

Shadow mapping is a cheap, fast and scene complexity independant method of creating shadows. This is why it is used in many applications, from games to 3D simulations. Shadow mapping does have some drawbacks. The problems that occur using shadow mapping are described in the next section.

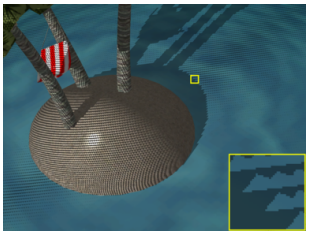
Resolution



2.4: Low shadow resolution

Because shadow mapping is an image-based technique, it is subject to resolution problems. The light map is projected over the entire area the light covers. If the area the camera covers is smaller, a big part of the available resolution will be wasted. Especially when the camera and the light source are very far apart, light map pixels will map to multiple screen pixels. This can be seen as the "blocky edges" in figure 2.4 that shadow mapping shadows often have. There are techniques to decrease or hide the wasted resolution. These techniques are described in Section 2.2.4 and further.

Floating point precision



2.5: Shadow acne

To check if a point is in shadow, it is projected to light space using the ightsources view and projection matrices. It is then compared to the value stored in the light map. Ideally, a point projected by the camera to light space would be equal to a point projected by the light to light space. As floating point numbers of finite precision are used, round-off errors can occur. This often leads to false self shadowing, also known as shadow acne. Figure 2.5 shows these artifacts. Shadow acne can be reduced by adding a bias to the shadow depth. Effectively this moves the shadows a little bit backwards, removing round-off errors. When this bias is too big, shadows will be moved too far backwards. This will make objects appear floating, or eliminate self shadows in places where they should appear.

Hard shadow borders

The shadow mapping algorithm tells us if a point is totally lit (1) or in shadow (0). This produces hard shadow borders as if they were created by an infinitely small point light, or a perfect directional light. In real life infinitely small point lights do not exist; they always have a size. This means that the shadows of this light will have an umbra and a penumbra, which would result in soft shadow borders.

Optimization

The paragraph before described the problems that arise using shadow mapping. To improve shadow quality, some measures can be taken, as described in [BAS02]. These measures are ways to improve standard shadow mapping, and most of them can also be applied to derived techniques.

2.2.2 Linear Z-buffer distribution

When rendering a scene, depth values are sampled non-uniformly ($\frac{1}{z}$). For scene cameras this is correct behavior. Objects in the foreground take up more space in the final render, so they should get more resolution. If this projection is used for light maps, objects close to the light will get more depth resolution than objects far from the light. This is because floating point numbers are used to store the distance, and small numbers have a higher precision than bigger numbers. Light map depth resolution should be equal over the entire scene, because the camera can be anywhere.

To make sure the depth resolution is divided equally, a change is needed in the way the depth value is calculated during the perspective transform. Normally, an eye point p_e, a 4D vector (x, y, z, w), will be transformed to the post-perspective space by multiplying it with the projection matrix. After this transformation, the vector is normalized by dividing it by w. The normalization of the z coordinate is responsible for the non-uniform distribution of the depth values. To distribute the depth values uniformly, after the projection z is replaced by $\frac{w(z_e - near)}{far - near}$ with far and $near$ the far and near planes of the light. After normalisation, this is equal to $\frac{z_e - near}{far - near}$. This is a uniform distribution between 0 and 1 (if $far > near$).

2.2.3 Calculating near and far planes

To decrease the effects of the floating point precision problems, measures must be taken to use the available shadow map precision for the objects that are visible to the camera. All precision should be used for the objects that are both in the lights and in the cameras frustum. The intersection i of these two frustums contains all objects that receive shadow. Objects that are in the light frustum but not in i can cast shadow, but not receive it. Because they can cast shadow, they still have to be rendered to the light map. This would mean that the near plane of the light should be moved back, which would decrease precision of the light map. A solution to this problem is depth clamping. All objects that are in front of the near plane are rendered as if they are exactly on the near plane. This enables these objects to block the light, but the near plane is kept as far back as possible, which increases precision.

Extensions

In the preceding section, standard shadow mapping is described. Over the years, a lot of extensions were proposed to increase shadow quality. Most of these techniques increase shadow quality by filtering the shadow map,

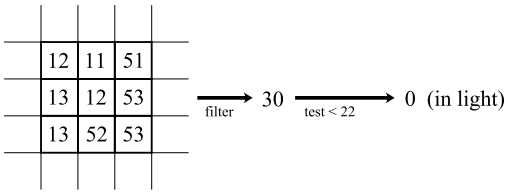
changing the projection of the scene or using multiple light maps for a single light. In the next section filtering methods will be described. These filtering methods hide the resolution problems that shadow mapping is subject to.

2.2.4 Percentage closer filtering

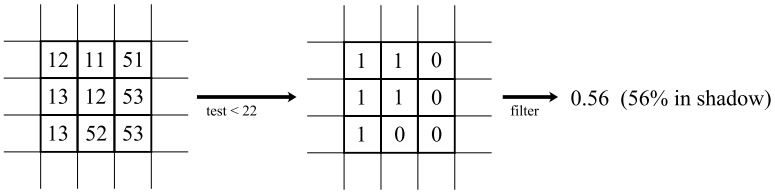
Filtering is used to decrease or hide the shadow aliasing due to resolution problems. It is a technique commonly used in computer graphics. instead of sampling just one point, the mean of multiple points are taken. This will reduce aliasing caused by undersampling an image. Filtering of shadow maps requires a different approach, suggested in [RSC87]. This approach, named **percentage closer filtering**, is explained in the following paragraphs.

When filtering a shadow map, taking the mean of the depth values at a point does not give the desired result. Take a look at figure 2.6a. This figure shows a small portion of a light map. The numbers in this light map represent the distance of the rendered geometry to the light source on that specific pixel. On this light map, a shadow test is performed for a point that is 22 away from the camera. When filtering the depth map values of the light map, a distance of 30 is obtained. The problem here is that *there is no object at distance 30*. There are just two objects at distance 11 and distance 53. Comparing to 30 would give a faulty result of 0 (not in shadow), even though our point is in shadow (the unfiltered shadow map distance is 11).

Figure 2.6b shows the correct way of filtering a shadow map. First, all depth values in the filter kernel are compared to the distance of the point (again, 22). These depth test results are then filtered. This leads to the result of 0.56, which means the point is 56% in shadow.



(a) Filtering the distances



(b) Filtering the depth tests

2.6: Filtering of depth maps: the incorrect and correct way.

Percentage closer filtering increases shadow border quality, but it still has some aliasing problems. One of these problems is banding. In the sample above, 9 samples are used to calculate the shadow value. This means that the outcome of the shadow calculation can only output ten values (0, $\frac{1}{9}$, $\frac{2}{9}$ ... $\frac{9}{9}$). A gradient of ten values looks better than one of two values (0 and 1), but the banding is still visible. To overcome this problem, linear interpolation between the depth tests is necessary.

This is how linear interpolating at a point with texture coordinate t is done: Texture coordinates are coordinates between 0 and 1. Multiply t by the light map size in pixels s. The result is the pixel offset of the point in the light map. The integer part i of this offset represents the texel that would normally be used to do shadow mapping. The fractional part f is the offset into this texel. Now the linear interpolated result l of the depth test can be calculated.

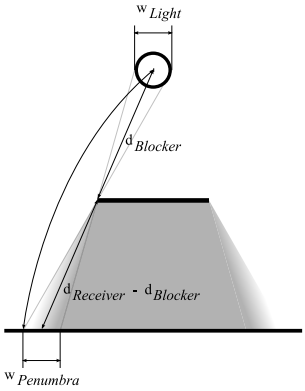
$$\begin{aligned} l &= (1 - f.y) \cdot ((1 - f.x) \cdot \text{sample}(i.x, i.y) + f.x \cdot \text{sample}(i.x + 1, i.y)) + \\ &\quad f.y \cdot ((1 - f.x) \cdot \text{sample}(i.x, i.y + 1) + f.x \cdot \text{sample}(i.x + 1, i.y + 1)) \end{aligned}$$

Linear interpolation of the depth test results totally removes banding aliasing, but it requires more light map lookups. The interpolated depth results can be used in percentage closer filtering, to increase shadow quality in exchange for even more lookups. Modern hardware does not have this penalty, because it provides instructions to do a hardware accelerated linear interpolation of the depth tests.

2.2.5 Percentage-closer soft shadows

Percentage closer filtering can improve shadow quality considerably. It creates the soft shadow borders somewhat resembling the soft shadows as they are seen in the real world. However, real shadows have umbras and penumbras, depending on the distance from the receiver to the blocker, the light size, and the distance to the light. The size of the percentage closer filtering borders depends on the available light map resolution. To overcome this limitation of percentage closer filtering, the filter size should depend on the distance from a receiver to a blocker. This is exactly what is done in *percentage-closer soft shadows* as proposed in [Fer05].

The percentage-closer soft shadows algorithm uses the same light map as standard shadow mapping. When rendering the final image it does some extra steps to determine the amount of shadow at a pixel. These steps are *blocker search*, *penumbra estimation* and *filtering*. During the *blocker search* step, the percentage-closer soft shadows algorithm searches a region in the shadow map for depth values that are closer to the light than the receiving point. These depth values are then averaged. In the *penumbra estimation* step, this averaged depth value is used as the distance to the blocker. Using this distance, the distance of the receiver to the light, and the light size, the penumbra width is calculated:



2.7: Penumbra calculation

$$w_{Penumbra} = \frac{(d_{Receiver} - d_{Blocker}) \cdot w_{Light}}{d_{Blocker}}$$

This calculation is illustrated in figure 2.2.5. The assumption is made that the blocker, receiver and light source are parallel planes. Although this is almost never the case, it works well in practice. For the final step, *filtering*, the penumbra width is used as the size of the percentage closer filtering kernel. This creates softer shadows at a distance, and harder shadows close to the blocker.

2.2.6 Variance shadow mapping

With percentage closer filtering, every shadow pixel is filtered when the light map is projected. This means that for every polygon drawn, the filtering is applied, and the filtering needs to sample the light map multiple times. This can be inefficient for scenes that have a lot of overdraw.

To overcome this problem, the light map has to be filtered before it is applied. One way to do this, called *variance shadow mapping*, is proposed in [DL06].

The variance shadow mapping algorithm works with depth distribution, not depth values. It is a statistical approach to shadow mapping. Instead of storing just the depth values in the light map, two values per point are stored: the depth of the point and the square of this depth. After that, the light map is filtered to average the depths and squared depths with their neighbours. Effectively, this filtering turns the pixels of the light map into weighted means over the area surrounding these pixels. Now, the two moments M_1 and M_2 can be obtained by sampling the texture. These moments are defined as follows:

$$
\begin{aligned}
M_1 &= E(x) &&= \int_{-\infty}^{\infty} x\, p(x)\, dx \\
M_2 &= E(x^2) &&= \int_{-\infty}^{\infty} x^2\, p(x)\, dx
\end{aligned}
$$

From these moments, the mean μ and variance σ^2 can be calculated:

$$
\begin{aligned}
\mu &= E(x) &&= M_1 \\
\sigma^2 &= E(x^2) - E(x)^2 &&= M_2 - M_1^2
\end{aligned}
$$

The variance is a quantitative measure of the width of a distribution. This means that it puts a bound on how much of the distribution can be far away from the mean. This bound is described in Chebyshev's inequality:

$$
P(x \geq t) \leq p_{max}(t) \equiv \frac{\sigma^2}{\sigma^2 + (t - \mu)^2}
$$

While testing if a point is in shadow, the distance z_p of the point to the light source is calculated as in standard shadow mapping. Now the probability l that z_p is inside the distribution obtained from the light map is calculated. This is only done when the distance z_p is greater than the first moment obtained from the light map, because shadows only appear behind blocking objects. The probability l gives a good estimation of the amount of light that reaches the point.

$$
l = \frac{\sigma^2}{\sigma^2 + (d - M1)^2}
$$

The quality of the shadows depend on the type of filter used to average the light map. Any kind of filters can be used, and multisampling anti-aliasing also helps to increase the shadow quality. The biggest benefit of applying filters to the light map texture is that it is relatively cheap, all hardware optimalisations can be used to do this very fast. Variance shadow mapping can only be used to filter the light map; it does not provide a way to create real-looking shadow umbras and penumbras.

Changing the projection

Another way to increase shadow quality is changing the projection. With shadow mapping the light map is created by rendering the scene from the viewpoint of the camera in world space. This light map can be filtered, but this does not change the available resolution. Creating the light map

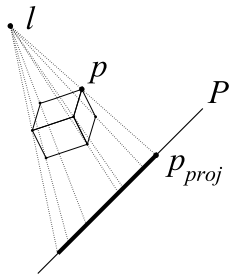
is nothing more than projecting the scene geometry to the unit cube; every 3D point is multiplied by a matrix to get the transformed position. When a light source is treated as a camera, every point is projected using the lights view and projection matrices. This gives resolution problems if the light is far away from the geometry the camera is viewing.

To increase shadow quality the scene can first be projected to another space where these resolution problems are less. Techniques that use different projections are perspective shadow mapping [SD02], light-space perspective shadow mapping [WSP04] and trapezoidal shadow mapping [WSP04]. Similar to this, multiple shadow maps [For07], or a tree structure of shadow maps [FFBG01], [LSK+05] can be used to increase the shadow quality.

These techniques require information about the scene that is not available in the general case. They can be implemented as optimizations in specific situations, but that is beyond the scope of this research.

2.3 Projected shadows

A fast way to create shadows is described in [Bli88]. The scene geometry is projected on the ground plane to create shadows. This technique can only be used to cast shadows onto a flat plane, and it is not suitable for self shadows.



2.8: Projecting geometry to a plane P

Projecting geometry onto a plane is achieved by projecting each individual point of the geometry to the plane from the light position, as shown in figure 2.8. Effectively this is a ray/plane intersection. This ray goes through the light l, and a point on the geometry p. The plane P is described using its plane equation $ax + by + cz + d = 0$. The projected point p_{proj} can be calculated using the following equation:

$$p_{proj} = l - (p - l)\frac{al_x + bl_y + cl_z + d}{a(p_x - l_x) + b(p_y - l_y) + c(p_z - l_z)}$$

This equation can be expressed in matrix form as projection matrix M:

$$M = \begin{bmatrix} bl_y + cl_z + d & -bl_x & -cl_x & -dl_x \\ -al_y & al_x + cl_z + d & -cl_y & -dl_y \\ -al_z & -bl_z & al_x + bl_y + d & -dl_z \\ -a & -b & -c & al_x + bl_y + cl_z \end{bmatrix}$$

When rendering the geometry, all objects are transformed using this projection matrix. The objects polygons will all be projected on P. To make these polygons appear as shadows, they should be drawn in a darker color.

Projecting the polygons like this will cast shadows on the entire infinite floor plane. To cast shadows on a finite floor plane, clipping has to be performed, to make sure shadows are only drawn to the correct area of the floor. Usually the stencil buffer is used to do the clipping.

2.3.1 Benefits

This way of shadow generating is fast, because all it takes is a simple transformation. No extra scene information is needed, the shadow casting geometry just has to be rendered multiple times, once for every light. The shadow can be cast on a plane or, if clipping is used, on a polygon.

2.3.2 Problems

Scene geometry consists of much more polygons than a simple plane. All this geometry can be used to create shadows, but none of this geometry can receive shadow. For the best results, realistic shadows should be present over the whole scene, not just on the floor.

This algorithm creates shadows with hard shadow borders, a point is either in shadow or in light. It does not support umbras and penumbras as they can be seen in the real world.

All geometry is projected to the same plane. If a depth buffer is used while rendering, this can lead to so called z-fighting. Z-fighting occurs when floating point rounding errors occur. As floating point numbers do not have infinite precision, the numerical value of a point projected to a plane can actually be just in front or just behind that plane. This means that, when using a depth buffer, some parts of the original plane seem to be in front of the shadow whereas some other parts will not. Figure (a) shows this problem. This problem can be solved by using an offset to place the shadows in front of the plane. The correct result is shown in Figure (b).

(a) Z-fighting (b) Correct projection

2.9: Projected shadows with and without z-fighting.

Self shadowing is not supported using projected shadows. It could be implemented by projecting the geometry to *every* polygon in the scene, but this means that every object has to be rendered $2n$ times, where n is the amount of polygons of the object. 3D Models today have thousands of polygons, so this is not a real-time solution.

2.4 Shadow volumes

A different approach to shadow generation are *shadow volumes*, proposed in [Cro77]. Every object in the scene is extruded in the direction of the light. If another object is inside the stretched object, the so-called *shadow volume*, it is in shadow, otherwise it is in light.

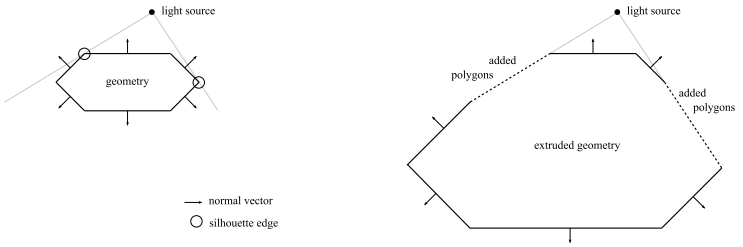
2.4.1 Brute force shadow volumes extrusion

The easiest way to stretch an object in the direction of the light is to find every polygon that is facing to the light, and extrude each edge in the direction of the light. This way, no extra information has to be available about the light blocker. This does however create a lot of extra geometry. For every light-facing polygon 6 extra polygons have to be created (2 for every edge to form a quad). There are smarter ways to extrude geometry available that use less polygons. How this is done is described in the next section.

2.4.2 Z-pass shadow volumes

To stretch the object in the direction of the light, first the silhouette of the object has to be found. This sihouette consists of all the edges between the

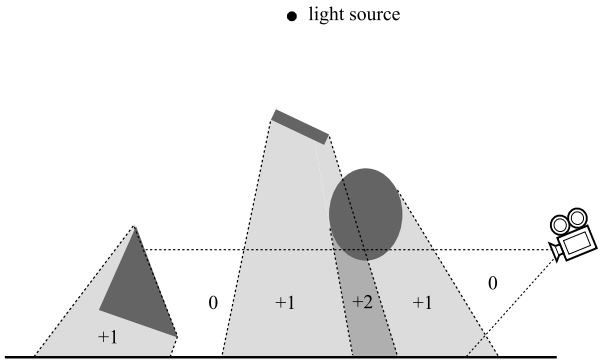
polygons that face the light and the ones that do not. To determine this, the dot product of the light vector and the surface normal is taken. If this dot product is greater than zero, the surface faces the light. This is shown in figure 2.10a. After the silhouette detection, all polygons that do not face the light are moved away from the light. This creates gaps in the geometry at the polygon edges. To fill these gaps, new polygons are added between the front and back facing polygons as in figure 2.10b.



(a) Edge detection (b) Extrusion

2.10: Two steps of shadow volume creation

With these generated shadow volumes, polygons can be classified as inside or outside shadow. Every polygon has to be tested against every shadow volume. Simply checking every polygon with every volume will not work in real-time. A solution to this problem was proposed in [Hei91]. This solution uses the stencil buffer to mark the shadowed areas. It can be seen as casting rays from the camera to the geometry, increasing and decreasing a counter every time a ray enters or exits a shadow volume. First, the scene geometry is rendered to the depth buffer. The stencil buffer is cleared. Now, all shadow volumes are rendered with depth and color buffer disabled for writing. All back-facing polygons are culled, so only the front facing ones are drawn. Every time a pixel is drawn, the stencil buffer is increased by one. The depth buffer is still enabled for reading, so only the shadow volumes that are in front of the visible geometry are drawn. This step is repeated, but now the front-facing polygons are culled, and the stencil buffer is now decreased by one. Again, only the shadow volumes that are in front of the visible geometry are drawn. After this pass, the stencil buffer contains zero where geometry is not in shadow, and any other number otherwise. This method increases and decreases the stencil buffer when the depth test (or Z-test) passes, so it is called *Z-Pass* shadow volumes.



2.11: Shadow volumes using the stencil buffer

Figure 2.11 shows the proces of shadow volumes. Every time a ray from the camera enters a shadow volume, the stencil buffer value of this ray is increased. Every time a ray exists a shadow volume, the value is decreased. Polygons that are rendered with a value of 0, are in light, all other polygons are in shadow.

Modern hardware has extensions that can combine the two stencil buffer write passes to one single pass.

When the stencil buffer is filled, the areas that contain shadows can be darkened, or left blank, while the area that is in light can be drawn using lighting and specular. A commonly used way to create the shadows is to not only draw the geometry to the depth buffer in the first pass, but to also draw it to the color buffer using only ambient lighting. In the last pass, the entire scene is drawn again, but only pixels that have a stencil value of zero are drawn using lighting and specular calculations.

This technique works as long as the camera is not inside a shadow volume. This is because only polygons in front of the camera are drawn, so the stencil buffer is not increased for the polygons behind the camera. This results in inverted shadows; everything that is supposed to be in light, is in shadow, and some shadows appear in light.

2.4.3 Z-fail shadow volumes

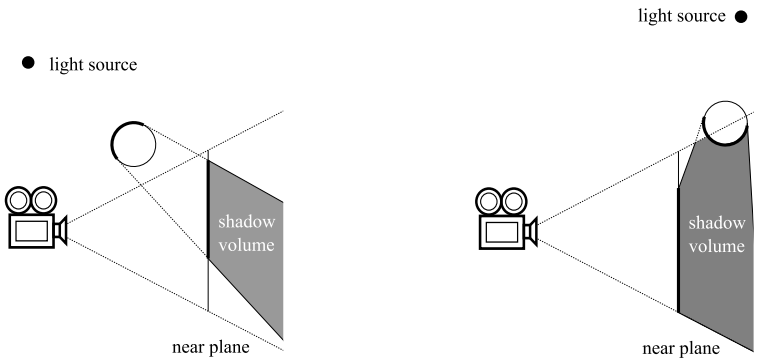
A solution to this problem is Z-fail shadow volumes [Car00]. Instead of rendering the shadow volumes *in front of* the visible geometry, the shadow volumes *behind* the visible geometry are rendered. Effectively, this is casting a ray from the geometry to infinity, not from the light to the geometry. Z-fail shadow volumes produce correct results only if the shadow volumes are capped. If they are not capped and the camera looks in the light direction,

shadowed areas will not be in shadow. Z-fail often increases the rendered polygon count dramatically. Every polygon behind the visible geometry is rendered, and none of them can be clipped by the depth buffer. Shadow volume caps are needed, so this is another increase to the polygon count. This is why most implementations switch from Z-pass to Z-fail rendering only if the camera is inside a shadow volume.

2.4.4 Z-pass+ shadow volumes

Z-fail shadow volumes are robust, but they are slower than Z-pass shadow volumes. In [HHLH05] an extension to Z-pass shadow volumes is proposed to make it robust, while still running faster than Z-fail shadow volumes. Problems with Z-pass rendering occur when a shadow volume intersects the near plane of the camera (which effectively means the camera is inside a shadow volume). A solution to this problem is putting caps on the shadow volume at the near plane of the camera. This can be done by adding extra geometry, but the position and shape of this geometry is dependent of the camera position, light position and the shape of the shadow volume. This information has to be calculated every frame, and is CPU intensive.

The Z-pass+ algorithm introduces another way to render the caps of the shadow volumes. When the light source and camera are on the same side of the near plane of the camera, all polygons of the occluder that face the light are rasterized to the near plane of the camera to initialize the stencil buffer. When the ightsource is on the opposite side of the near plane, the backfaces off the occluder are rasterized to the near plane of the camera. This is shown in figure 2.12. After the rasterizing the stencil buffer is initialized and normal Z-pass rendering is done.



(a) Light and camera at the same side (b) Light and camera at opposite sides

2.12: Projecting the geometry to the near-plane of the camera

Rasterizing the caps is done by projecting the geometry from the light to the near plane. Only the polygons that face the light (or face away, if the ightsource is on the other side of the near plane) should be rendered. To do this efficiently, a custom projection matrix is used. This projection matrix simply projects the scene from the position of the light source onto the near-plane of the camera. Because this projection is done from the position of the light source, front- and backface culling no longer culls polygons that face away from the *camera*, it now culls the polygons that face away from the *light*. Only the polygons that face the light are rasterized.

2.4.5 Problems

2.13: Leaks in the geometry

To use the shadow volumes algorithm, all geometry in the scene has to be *watertight*. This means that all polygons have to be connected to other polygons, and the model can not contain gaps. If geometry does contain gaps, the shadow will 'leak' into parts that are supposed to be in light. This is shown in figure 2.13. The highlighted box is supposed to be in light, but because one of the leaves is not watertight, an incorrect shadow is visible. Often, models used in games and simulations are not closed to decrease polygon count and thus increase rendering speed. Making these models watertight requires extra polygons in the models. A solution to this problem is creating shadow volumes that have a lower polygon count than the actual model, but this means creating twice the amount of geometry for a scene.

Using this technique, objects are classified either as *totally inside* shadow or *totally outside* shadow. This means it creates hard shadow borders as if the shadows were from a point light. Soft shadows are not possible using the standard shadow volumes algorithm. In section 2.4.7 discusses a way to create soft shadows using a shadow volumes derivative.

The shadow volumes technique renders all shadow volumes, which means that the scene is rendered at least twice. Also, extra geometry is added to extrude the shadow volumes. This means a lot of extra polygons are drawn. Especially using Z-fail shadow volumes, where the depth buffer cannot be used to throw away polygons behind the visible geometry, the actually drawn polygon count can be enormous. When rendering using modern hardware, one of the bottlenecks is the fill-rate, the number of drawn pixels per frame. Since shadow volumes needs all shadow volumes drawn, this fill rate is enormous, because of the large amounts of overdraw. Pixels cannot be thrown away, because shadow volumes do not write to the depth buffer and this slows down rendering. Ways to decrease the fill rate are discussed in section 2.4.6.

Every shadow casting object has to have a shadow volume. This means that the amount of shadow volumes increases when the scene complexity increases. Especially in dynamic scenes with many shadow casters, it can be a problem to use the algorithm in real-time.

2.4.6 Optimization

Using shadow volumes for shadows consumes a lot of fill rate. Especially in complex scenes, every pixel can be overdrawn more than twenty times. Even the fastest hardware has a hard time rendering so many pixels while still keeping a real-time frame rate. A way to reduce the fill rate while using shadow volumes is proposed in [LWGM04]. This technique uses scene information to cull and clamp the shadow volumes, so only a small part of the infinitely long shadow volumes have to be rasterized. There are three ways these CC shadow volumes remove unnecessary areas of shadow volumes.

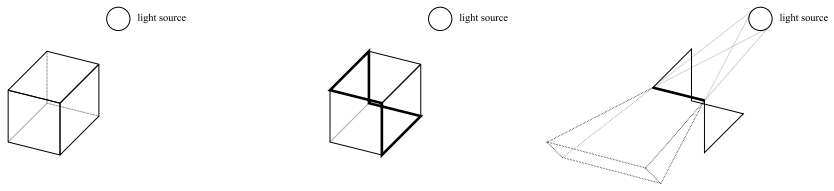
- *Culling.* All shadow volumes that are completely inside other shadow volumes are culled. This removes a lot of unnecessary shadow volumes that would have no effect on the final scene anyway.
- *Continuous Shadow Clamping.* The shadow is clamped to the part of the scene where shadow receivers are. To achieve this, the bounding boxes of the geometry are checked against the view frustum of the camera. Also, the minimum z_{min} and maximum z_{max} distance of the shadow receivers to the light are used to clamp areas that will not

contain any receivers. This is done by projecting the line from z_{min} to z_{max} to the view plane, Only the y components of the projected line are used to mark the area that can contain this receiver. In the areas that are not occupied by a projected line no shadow volumes have to be rasterized.

- *Discrete Shadow Clamping.* The camera space is split up into multiple regions using planes that face towards the light and pass through the view point. The part of the shadow volume between these two planes is checked to determine if there is any geometry that can receive shadow in this slice. If there is none, then the part of the shadow volume can be removed.

2.4.7 Penumbra wedges

Using standard shadow volumes, there is no way to create soft shadows. To overcome this limitation, an extension to shadow volumes is proposed in [AMA02]. This method uses extra geometry to detect the areas the penumbra is in. instead of extruding the silhouette edges of the geometry in the direction of the light, the edges are extruded in two directions, thus creating wedges. The process of creating the wedges is shown in figure 2.14. For normal shadow volumes, the edges are extruded over the shadow volumes plane formed by the light source and the two endpoints of the edge. To extrude the wedges, the shadow volumes plane is rotated around the edge. The amount of rotation depends on the size of the light source and the distance to the light.



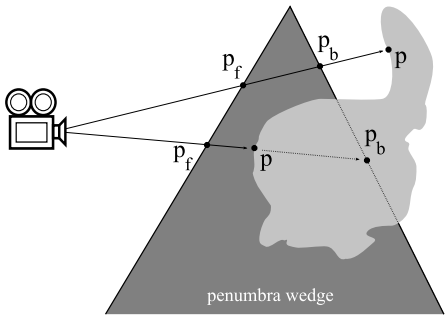
(a) Scene with area light (b) Edge detection (c) Constructing the wedge

2.14: Penumbra wedges algorithm

Now, the scene is rendered using diffuse lighting and specular. A depth buffer is used to store the depths of the rendered pixels. This information can later be used to obtian the 3D coordinates of the 2D rendered pixels. For normal shadow volumes the stencil buffer (usually with 8 bit precision) would be used. The penumbra wedges algorithm needs to store more information

in the stencil buffer, so the conventional stencil buffer does not have enough precision. This is why a 16-bit texture is used as a stencil buffer.

Next, the wedges are rasterized to this stencil buffer. No depth and color information is written during this pass. Every front facing polygon of the wedge is rendered. Per wedge, the front and back planes are known. For every rendered pixel, its depth value is looked up in the depth buffer. From the pixel coordinate and the depth value, the location of the original point p is calculated. The point p_f is the point where the ray from the camera to the current pixel intersects the front plane of the wedge. The point p_b is the point where the ray intersects the back plane. If p lies between p_f and p_b, the pixel is inside a wedge. This is shown in figure 2.15.



2.15: Determining if p is inside a wedge

Once it is known if p lies between p_f and p_b the light intensity of p can be calculated. A ray is constructed from p in the direction of the normal of the shadow volumes plane. The intersections of this ray with the front plane i_f and back plane i_b are the positions that are totally in light and totally in shadow. The shadow value can be interpolated using the distance between p and i_b divided by the distance between i_f and i_b. Other interpolations are possible to achieve better result on wedge sides.

The biggest benefit of using penumbra wedges is that the shadows have soft borders and no aliasing occurs. The method does however generate a lot of extra geometry, which can be a burden in complex scenes.

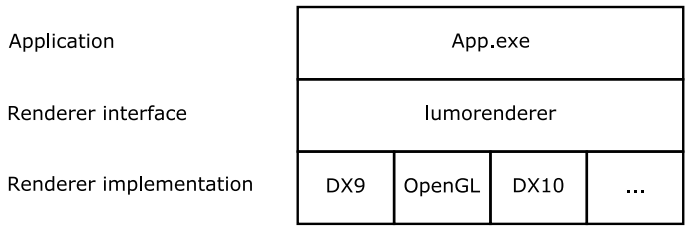
2.5 Available tools and software

This section describes the available tools at the start of this project. This research heavily depends on the newly available capabilities of modern hardware. To put these capabilities to use, one of the new graphics API's can be used: Direct3D 10 [Dir07] or OpenGL [Ope07] using extensions. These API's are also discussed in this section.

2.5.1 Renderer2

Previously, Re-lion used an in-house 3D engine called Lumo renderer. This engine was scenegraph based, and used many different shaders. A new, more basic 3D engine was in development. This engine is only to provide core 3D functionality. Scene management has to be done in a higher level library or the application. From now on, the new renderer will be referred to as *Renderer2*.

In order to create API independancy, the 3D engine is built up in two layers. Figure 3.1 shows these. The first layer is the implementation independant interface, which exposes the functions used by te application. The second layer is the API dependent layer. This layer contains the API-specific implementation of the functions. This creates the possibility to create multiple API drivers (Direct3D 9, OpenGL, Direct3D 10) without having to change the interface the program uses.



2.16: API layers in the new renderer

Renderer2 Functionality

The Renderer2 API is responsible for handling all graphics calls. It is designed to function the same no matter what graphics API driver is chosen. An instance of a Renderer2 can be created using a single function call. It is possible to create a renderer using a specific driver, or let the system choose one. The user can specify the display format, the display mode (full-screen or windowed), the refresh rate and the multisample settings.

When the renderer is created, it can be used until lumorenderer is unloaded or the renderer is destroyed. Using the renderer, resources can be created, destroyed and manipulated. On destruction, all resources of the renderer that are still in memory are released to the operating system. This ensures that there will be no memory leaks when the user does not free some resources.

Lumorenderer handles resources in an graphics API independant way. When resources are created, the application only gets a handle to the resource.

Allocation and deallocation of resources is handled by Renderer2 internally.

Renderer2 only has support for low level resources like vertexbuffers, indexbuffers and textures. This means that an application is responsible for higher level primitives like meshes.

Graphics API's

To put the capabilities of the modern hardware to use, a graphics API is needed that supports this modern hardware. Two graphics API's qualify for this: Direct3D 10 and OpenGL. These two API's are described in the following sections.

2.5.2 OpenGL

OpenGL is a graphics API that is supported on multiple platforms. Through extensions, it has support for geometry shaders [GLE07] and other new capabilities of modern hardware. The major advantage of using OpenGL is its support for multiple platforms. This would mean that the simulations can be developed platform independently. However, as mentioned before, a simulator constists of a complete system, so the operating system is usually chosen by the creator of the simulator.

Since Renderer2 only supported Direct3D 9 and no OpenGL when this research started, using OpenGL meant creating a driver from scratch. This seamed a lot more work than adapting the driver to Direct3D 10, so the choice for Direct3D 10 was made.

2.5.3 Direct3D 10

When this research started only a Direct3D 9 Renderer2 driver was available. Although most 3D graphic effect can be realized using Direct3D 9, Direct3D 10 introduces some new features that can improve performance and even do things that were never possible on the GPU. This section describes the major differences between Direct3D 9 and Direct3D 10. Consideration for porting from Direct3D 9 to Direct3D 10 can be found at [DXC07].

Backward compatibility

With the introduction of Direct3D 10, Microsoft has chosen to drop backwards compatibility between Direct3D versions. One of the reasons to do this, is the new driver model in Windows Vista (Direct3D 10 only runs on Vista). Another reason is to loose the capabilities system of Direct3D 9.

Graphics cards can *partially* support Direct3D 9. This makes programming for these cards harder. All capabilities of the graphics hardware have to be checked at run time, to make sure the graphics hardware supports what the software is trying to render. With Direct3D 10, the complete set of capabilities is always guaranteed. No more run-time checking is necessary. The Direct3D 9 driver for Renderer2 made a lot of assumptions on the capabilities of the graphics hardware, but checking whether they are supported should still be done. In the Direct3D 10 driver this is no longer necessary.

Another big change in Direct3D 10 is that everything that can be done at initialization time, will be done there. Most run time checking is dropped in favor of creation time checking. This means that the CPU load is less high while running (initialization can take longer though).

Geometry shaders

Modern graphics hardware can process huge amounts of data in hardware. An application provides the data to the graphics hardware, and then this data is processed in the background. This processing is done by using shaders, programs that are executed on the graphics hardware. First, a vertex shader is executed for every vertex. After this, the primitives formed by the processed vertices are rasterized to the screen, and a pixel shader is executed for every pixel.

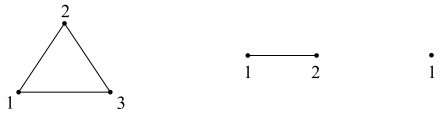
The first vertex and pixel shaders could only execute a small number of instructions, with limits to the number of texture fetching instructions in the pixel shader. The vertex shader did not support texture fetches. With the improvement of the graphics hardware capabilities, the need for longer shaders arrised. This resulted in shader models 1.1, 2 and finally shader model 3.

This was how rendering was done in Direct3D 9. With Direct3D 10, shader model 4 was introduced, with a new type of shader: the *geometry shader*. If present, this shader is executed before the primitives are rasterized and after the vertices are processed. The geometry shader gets an entire primitive as input, and can output a number of vertices. This means that in the geometry shader vertices can removed (by not outputting them), but they can also be created in hardware. After the geometry shader is executed the created vertices can be used to rasterize the data the primitives or they can be streamed into a vertex buffer using *stream out* so they can be used again later.

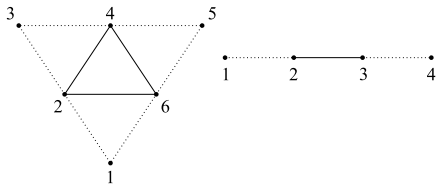
Primitive types

To make geometry shaders even more useful, new rendering primitive types are made available. In Direct3D 9 points, lines and triangles (lists, strips or fans) where the only supported primitives. Direct3D 10 adds the primitives with adjacency information to this list. This means that instead of presenting one triangle to the geometry shader, four triangles (the first and its three neighbors) are presented to the geometry shader.

Figure 3.2 shows how this adjacency information is presented to the geometry shader as six vertices (for a triangle) or four vertices (for a line). Note that a point does not have adjacency information; it is never connected to other points.



(a) Direct3D 9 primitives



(b) Primitives with adjacency information

2.17: New primitives to provide adjacency information to the geometry shader.

This adjacency information is necessary for detecting silhouette edges of geometry, an important part of shadow volume generation. It is the responsibility of the application to provide this adjacency information if it is needed.

Effect framework

To simplify the use of shaders, DirectX provides a framework that groups pixel and vertex shaders into effects. To make use of a vertex/pixel shader combination, only the effect has to be activated. Effects also handle the state configuration of the graphics hardware. State configuration are hardware settings like the use of a depth buffer, which polygons will be culled or the size of point sprites. On activation of an effect, these state settings were set

to the correct values, and on deactivation, the state was restored to what it was before.

In Direct3D 9 effects were part of a helper library named D3DX, a higher level API library. With Direct3D 10, effects have become a part of the Direct3D API. These effects also support the new geometry shaders. From an applications point of view, effect usage does not change with the presence of a geometry shader. In Direct3D 10, effects are part of the Direct3D API, so no high-level library is needed.

In the shader models before shader model 4, shader variables could be shared by all effects in an effect pool. An effect pool is just a collection of effects. Sharing these variables was done by putting the `shared` keyword in front of a variable. This also meant that if the value of one variable was changed, all values of the variables had be uploaded to the graphic card.

Direct3D 10 introduces a way of grouping the shared variables using constant buffers. Constant buffers are groups of variables that are updated at the same time. When a variable in a constant buffer is updated, only that constant buffer has to be updated. Typical application usage of constant buffers would be a constant buffer for variables that are almost never updated, another buffer for variables that are updated when special cases occur, and one constant buffer for variables that are changed every frame.

The Direct3D 9 Driver for Renderer2 used part of the effect framework. Effects allow for multiple techniques (different shader combinations in one file) and multiple passes (Rendering geometry multiple times in a row, but using different shader combinations for every pass). The Renderer2 driver did not encapsulate this functionality, multiple techniques require multiple shader files, and only one pass was supported.

Vertex declarations

Vertex buffers are buffers in memory that hold vertex data. The format of this vertex data does not matter when storing it in video memory. When rendering the vertex data however, the graphics hardware needs to now how to interpret the data. In Direct3D 9 vertex format declarations were used for this end. A vertex format declaration describes the layout, the semantics of every component and the usage of a vertex. Before rendering geometry, a vertex format declaration has to be made active.

On the graphics hardware, the vertex data is fed to the vertex shader. This happens by binding the vertex data to the vertex components in the vertex shader. In Direct3D 9, this binding happened every time a vertex format declaration was made active. Only one vertex format declaration was needed for one vertex format, and it could be used with all shaders.

With Direct3D 10, the binding is done using an input layout. An input layout can be compared to a shader specific vertex format declaration binding. It connects one shader to a vertex buffer. This means that binding and the vertex format declaration components to the shader vertex components and the validating of the binding only has to be done once, at the beginning of the program. This increases runtime performance.

Overal comparison

The previous section showed the differences between Direct3D 9 and Direct3D 10. Although there are some differences, big parts of the API have remained roughly the same. Although both OpenGL and Direct3D 10 support the same new hardware capabilities, the OpenGL API and Direct3D 9 have much less in common. This means that upgrading an application from Direct3D 9 to Direct3D 10 should be less work than rewriting it to OpenGL. This is why Direct3D 10 was chosen for implementation in this research.

Chapter 3

Implementing shadows

This chapter discusses the implementation of the various shadow techniques and the application framework. It also describes problems that occurred during implementation, and the solutions that were found for these problems. First, an overview will be given of all the available software and tools at the start of this research.

3.1 Available tools and software

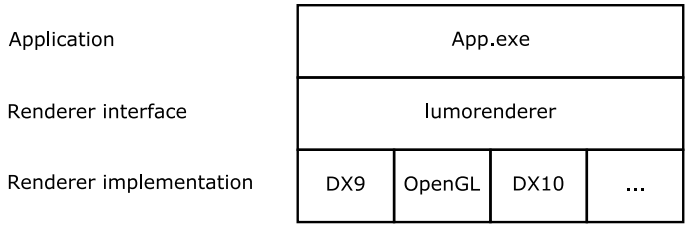
This section describes the available tools at the start of this project. This research heavily depends on the newly available capabilities of modern hardware. To put these capabilities to use, one of the new graphics API's can be used: Direct3D 10 [Dir07] or OpenGL [Ope07] using extensions. These API's are also discussed in this section.

3.1.1 Renderer2

Previously, Re-lion used an in-house 3D engine called Lumo renderer. This engine was scenegraph based, and used many different shaders. A new, more basic 3D engine was in development. This engine is only to provide core 3D functionality. Scene management has to be done in a higher level library or the application. From now on, the new renderer will be referred to as *Renderer2*.

In order to create API independancy, the 3D engine is built up in two layers. Figure 3.1 shows these. The first layer is the implementation independant interface, which exposes the functions used by te application. The second layer is the API dependent layer. This layer contains the API-specific implementation of the functions. This creates the possibility to create multiple

API drivers (Direct3D 9, OpenGL, Direct3D 10) without having to change the interface the program uses.



3.1: API layers in the new renderer

Renderer2 Functionality

The Renderer2 API is responsible for handling all graphics calls. It is designed to function the same no matter what graphics API driver is chosen. An instance of a Renderer2 can be created using a single function call. It is possible to create a renderer using a specific driver, or let the system choose one. The user can specify the display format, the display mode (full-screen or windowed), the refresh rate and the multisample settings.

When the renderer is created, it can be used until lumorenderer is unloaded or the renderer is destroyed. Using the renderer, resources can be created, destroyed and manipulated. On destruction, all resources of the renderer that are still in memory are released to the operating system. This ensures that there will be no memory leaks when the user does not free some resources.

Lumorenderer handles resources in an graphics API independant way. When resources are created, the application only gets a handle to the resource. Allocation and deallocation of resources is handled by Renderer2 internally.

Renderer2 only has support for low level resources like vertexbuffers, indexbuffers and textures. This means that an application is responsible for higher level primitives like meshes.

Graphics API's

To put the capabilities of the modern hardware to use, a graphics API is needed that supports this modern hardware. Two graphics API's qualify for this: Direct3D 10 and OpenGL. These two API's are described in the following sections.

3.1.2 OpenGL

OpenGL is a graphics API that is supported on multiple platforms. Through extensions, it has support for geometry shaders [GLE07] and other new capabilities of modern hardware. The major advantage of using OpenGL is its support for multiple platforms. This would mean that the simulations can be developed platform independently. However, as mentioned before, a simulator constists of a complete system, so the operating system is usually chosen by the creator of the simulator.

Since Renderer2 only supported Direct3D 9 and no OpenGL when this research started, using OpenGL meant creating a driver from scratch. This seamed a lot more work than adapting the driver to Direct3D 10, so the choice for Direct3D 10 was made.

3.1.3 Direct3D 10

When this research started only a Direct3D 9 Renderer2 driver was available. Although most 3D graphic effect can be realized using Direct3D 9, Direct3D 10 introduces some new features that can improve performance and even do things that were never possible on the GPU. This section describes the major differences between Direct3D 9 and Direct3D 10. Consideration for porting from Direct3D 9 to Direct3D 10 can be found at [DXC07].

Backward compatibility

With the introduction of Direct3D 10, Microsoft has chosen to drop backwards compatibility between Direct3D versions. One of the reasons to do this, is the new driver model in Windows Vista (Direct3D 10 only runs on Vista). Another reason is to loose the capabilities system of Direct3D 9. Graphics cards can *partially* support Direct3D 9. This makes programming for these cards harder. All capabilities of the graphics hardware have to be checked at run time, to make sure the graphics hardware supports what the software is trying to render. With Direct3D 10, the complete set of capabilities is always guaranteed. No more run-time checking is necessary. The Direct3D 9 driver for Renderer2 made a lot of assumptions on the capabilities of the graphics hardware, but checking whether they are supported should still be done. In the Direct3D 10 driver this is no longer necessary.

Another big change in Direct3D 10 is that everything that can be done at initialization time, will be done there. Most run time checking is dropped in favor of creation time checking. This means that the CPU load is less high while running (initialization can take longer though).

Geometry shaders

Modern graphics hardware can process huge amounts of data in hardware. An application provides the data to the graphics hardware, and then this data is processed in the background. This processing is done by using shaders, programs that are executed on the graphics hardware. First, a vertex shader is executed for every vertex. After this, the primitives formed by the processed vertices are rasterized to the screen, and a pixel shader is executed for every pixel.

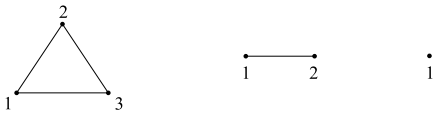
The first vertex and pixel shaders could only execute a small number of instructions, with limits to the number of texture fetching instructions in the pixel shader. The vertex shader did not support texture fetches. With the improvement of the graphics hardware capabilities, the need for longer shaders arrised. This resulted in shader models 1.1, 2 and finally shader model 3.

This was how rendering was done in Direct3D 9. With Direct3D 10, shader model 4 was introduced, with a new type of shader: the *geometry shader*. If present, this shader is executed before the primitives are rasterized and after the vertices are processed. The geometry shader gets an entire primitive as input, and can output a number of vertices. This means that in the geometry shader vertices can removed (by not outputting them), but they can also be created in hardware. After the geometry shader is executed the created vertices can be used to rasterize the data the primitives or they can be streamed into a vertex buffer using *stream out* so they can be used again later.

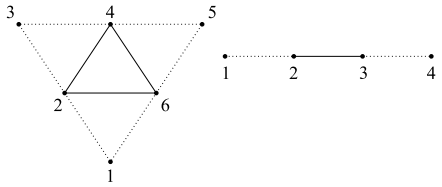
Primitive types

To make geometry shaders even more useful, new rendering primitive types are made available. In Direct3D 9 points, lines and triangles (lists, strips or fans) where the only supported primitives. Direct3D 10 adds the primitives with adjacency information to this list. This means that instead of presenting one triangle to the geometry shader, four triangles (the first and its three neighbors) are presented to the geometry shader.

Figure 3.2 shows how this adjacency information is presented to the geometry shader as six vertices (for a triangle) or four vertices (for a line). Note that a point does not have adjacency information; it is never connected to other points.



(a) Direct3D 9 primitives



(b) Primitives with adjacency information

3.2: New primitives to provide adjacency information to the geometry shader.

This adjacency information is necessary for detecting silhouette edges of geometry, an important part of shadow volume generation. It is the responsibility of the application to provide this adjacency information if it is needed.

Effect framework

To simplify the use of shaders, DirectX provides a framework that groups pixel and vertex shaders into effects. To make use of a vertex/pixel shader combination, only the effect has to be activated. Effects also handle the state configuration of the graphics hardware. State configuration are hardware settings like the use of a depth buffer, which polygons will be culled or the size of point sprites. On activation of an effect, these state settings were set to the correct values, and on deactivation, the state was restored to what it was before.

In Direct3D 9 effects were part of a helper library named D3DX, a higher level API library. With Direct3D 10, effects have become a part of the Direct3D API. These effects also support the new geometry shaders. From an applications point of view, effect usage does not change with the presence of a geometry shader. In Direct3D 10, effects are part of the Direct3D API, so no high-level library is needed.

In the shader models before shader model 4, shader variables could be shared by all effects in an effect pool. An effect pool is just a collection of effects. Sharing these variables was done by putting the `shared` keyword in front of a variable. This also meant that if the value of one variable was changed, all values of the variables had be uploaded to the graphic card.

Direct3D 10 introduces a way of grouping the shared variables using constant buffers. Constant buffers are groups of variables that are updated at the same time. When a variable in a constant buffer is updated, only that constant buffer has to be updated. Typical application usage of constant buffers would be a constant buffer for variables that are almost never updated, another buffer for variables that are updated when special cases occur, and one constant buffer for variables that are changed every frame.

The Direct3D 9 Driver for Renderer2 used part of the effect framework. Effects allow for multiple techniques (different shader combinations in one file) and multiple passes (Rendering geometry multiple times in a row, but using different shader combinations for every pass). The Renderer2 driver did not encapsulate this functionality, multiple techniques require multiple shader files, and only one pass was supported.

Vertex declarations

Vertex buffers are buffers in memory that hold vertex data. The format of this vertex data does not matter when storing it in video memory. When rendering the vertex data however, the graphics hardware needs to now how to interpret the data. In Direct3D 9 vertex format declarations were used for this end. A vertex format declaration describes the layout, the semantics of every component and the usage of a vertex. Before rendering geometry, a vertex format declaration has to be made active.

On the graphics hardware, the vertex data is fed to the vertex shader. This happens by binding the vertex data to the vertex components in the vertex shader. In Direct3D 9, this binding happened every time a vertex format declaration was made active. Only one vertex format declaration was needed for one vertex format, and it could be used with all shaders.

With Direct3D 10, the binding is done using an input layout. An input layout can be compared to a shader specific vertex format declaration binding. It connects one shader to a vertex buffer. This means that binding and the vertex format declaration components to the shader vertex components and the validating of the binding only has to be done once, at the beginning of the program. This increases runtime performance.

Overal comparison

The previous section showed the differences between Direct3D 9 and Direct3D 10. Although there are some differences, big parts of the API have remained roughly the same. Although both OpenGL and Direct3D 10 support the same new hardware capabilities, the OpenGL API and Direct3D 9

have much less in common. This means that upgrading an application from Direct3D 9 to Direct3D 10 should be less work than rewriting it to OpenGL. This is why Direct3D 10 was chosen for implementation in this research.

3.2 Implementation

After having described the available tools, we will now discuss the created software, the problems that occurred while implementing it, and the solutions that were found for these problems.

The main goal of this research is to compare shadow techniques and evaluate what the newly available techniques add to them, using re-lion's software. A way to evaluate these techniques is to implement them and compare their behavior, performance and visual results. An application will be created to visualize the shadow techniques. Implementation of this application will be done using the Renderer2 API. To do this, the different components of the application are identified and discussed in the next section.

3.2.1 Components

The application can be divided into a number of distinct components that need to be implemented in order to compare the shadow techniques. The following components can be identified:

- *The driver for Renderer2.* The application will make use of the Renderer2 API. This API currently only supports Direct3D 9. The application ought to use Direct3D 10, so a Direct3D 10 driver should be added to Renderer2.
- *The application framework.* This research is about comparing shadow techniques. In order to do this, a platform must exist in which scenes can be loaded and saved and parameters can be changed, to be able to test the shadow techniques in a number of different situations. This application will be developed on top of the Renderer2 API, and this API is solely responsible for the communication with the graphics hardware.
- *The shadow techniques.* To compare the shadow techniques, they should be implemented using the new API. Although every technique will have the same interface to the application, it handles shadows in its own way.

The following sections describe the implementation of these components.

3.2.2 Renderer2 Driver

Because the Renderer2 driver is required to use Direct3D 10 technology this Driver was the first thing to be implemented. At first, this did not seem such a substantial task because the application was only to use basic functionality.

The first approach was to hollow out the Direct3D 9 driver for Renderer2 and reimplemented it using Direct3D 10. Only the parts that were needed for the implementation of the application framework were filled in. This worked for most of the basic functionality like rendering meshes using different shaders.

However, some shadow techniques required extra functionality, for example the possibility to render to a texture. This meant the driver had to be adapted again. The implementing of the driver, which was supposed to be the first thing to be finished, turned out to take a lot longer than expected.

Finally, the Direct3D 10 Renderer2 driver almost had the same functionality as the Direct3D 9 one. Most of the time this was achieved by just rewriting Direct3D 9 code to Direct3D 10, but for some parts the API has changed considerately. The problems that arose because of these changes will be described in the following sections.

Shader

Renderer2 has a shader resource type. In this context, a shader consists of vertex- and pixel programs that are executed on the GPU when rendering the scene. Shaders in Renderer2 are created by passing the shader source code to the Renderer2 API. This source code is then compiled on the fly, by the Renderer2 driver. To use different drivers, different shaders have to be written because the shader language is graphics API specific.

In the Renderer2 driver implementation for Direct3D 9 and in the new driver for Direct3D 10, a shader corresponds with a Direct3D effect. If an effect contains a geometry shader, it is executed, and if it does not, only the vertex and pixel shaders are used. This means that for a Direct3D 10 enabled application using the Renderer2 interface just the shaders need to be adapted to support the Direct3D 10 functionality, not the application.

Still, a few changes had to the made to the Renderer2 API. Direct3D 10 comes with some new primitive types that need to be supported. Support for these new primitives was not available in the Renderer2 framework. The triangle fan primitive was dropped in Direct3D 10. These issues were solved by adding the extra primitive types to the primitive types enumeration, and an error code to inform the application when a primitive type is not supported.

The Renderer2 framework supports global shader variables. This means that by setting a variable that is marked as shared in an effect, all variables with the same name in all other effects will be changed. In Direct3D 9 this behavior was implemented using an effect pool. Whenever a shader was added to the effect pool, its shared variables were added. Direct3D 10 still supports effect pools, but it does not support the addition of new shared variables with the addition of shaders. This is because Direct3D 10 uses constant buffers.

The problem with shared constant buffers is that they can only be declared inside an effect pool. Child effects cannot declare shared variables. This means that all shared variables in constant buffers have to be known to the effect pool when compiling the child effects. The problem with Renderer2 is that when a effect pool is created, there is no information about the shaders that will be used during the application. The Direct3D 9 driver allowed effects to declare their own shared variables, but this will not work using the Direct3D 10 effect pool system.

For the created application all shaders and thus all constant buffers were known, so these variables were hard coded in the Direct3D 10 driver. When the Direct3D 10 driver is used in another application this behavior has to be changed.

One way to do this is recompiling the effect pool after loading a new shader. The Renderer2 framework does not force an application to load shaders at the start of a program, but creating and compiling shaders also took a long time in Direct3D 9, so it will probably not be something that is done while rendering. To recompile the effect pool, the shader sources need to be parsed and the variable and constant buffer names have to be extracted to build an effect pool.

Another way to let each shader have its own constant buffer layout is by not using effect pools. The Direct3D 10 Renderer2 driver is responsible for updating all variables in every known shader. This means that for each updated variable, all shaders that contain this variable have to be updated. Even in the best case this causes a performance penalty when using multiple shaders.

Vertex format declarations

Renderer2 works with vertex format declarations. Internally, the Direct3D 10 driver works with input layouts. The conversion between the two is solved by saving the vertex layout when creating a vertex format declaration. Whenever a shader is used in combination with a vertex format declaration, a look up is done to check whether or not an input layout for this specific combination already exists. If it does, the input layout is used. If it doesn't

exist an input layout is created and added to the list of combinations. When in the first frame multiple shader are used, a lot of input layouts are created. This can cause a performance hit. The frames after that only have to look up the right input layout, so this performance hit only happens once.

Driver implementation

The biggest problem of writing the driver was that the Renderer2 model was designed with the Direct3D 9 infrastructure in mind. Although Renderer2 provides some abstraction, a lot of the driver infrastructure has a one to one connection with the infrastructure of Direct3D 9. This means that every big change in the API has to be 'undone' using a workaround. Some of the bigger workarounds were presented in the last paragraphs.

With a new API, new functionality is exposed. Because Renderer2 was based on the Direct3D 9 infrastructure, this new functionality would be exposed through extensions. This would mean that the Renderer2 driver has to be queried about the capabilities of the hardware it is running on, enabling or disabling functionality when needed. Renderer2 was designed as an intermediate layer between the hardware and software that should work 'out of the box', no matter what driver is used internally. For this approach, capabilitly checking is not desirable.

Fortunately, the design of Renderer2 uses the effect framework. Direct3D 10 has a new and improved version of this framework, but it still uses the same basic idea: An effect is created, its variables can be changed, and the effect is then used while rendering. The inner workings of the effect are specified completely in a shader, instead of in the driver or Renderer2 library. This means that extensions added to the shader model in Direct3D 10 can be used without having to create extensions to the Renderer2 library.

Result

The created Renderer2 driver was needed to make use of the new technology. The bare-bone driver that was built first did not suffice for the functionality that some shadow techniques needed. As a result, the Direct3D 10 driver now has almost as much functionality as the Direct3D 9 driver had. It should not take much effort to add the missing functionality to the driver.

3.2.3 Application framework

With the Direct3D 10 functionality implemented in the Renderer2 driver, an application framework could be created. While implementing the shadow

techniques more and more features where added to the demo framework to make debugging easier and to visualize problems with the shadow creation. In the next paragraphs the functionality and the implementation of the framework will be explained.

Basic functionality

In order to create shadows a shadow caster and a shadow receiver are required. This is why functionality had to be added to load meshes. Before Renderer2, Re-lion used scene graphs to represent geometry and its position. Because scene graphs cause a lot of overhead if they are not optimized, this representation of the scene was taken out of the main engine when Renderer2 was designed. With Renderer2, it is an applications responsibility to sort and/or cull geometry if needed.

Renderer2 mesh data is represented by multiple files with the following extensions:

- *.vtx* A file that holds the vertex description combined with the vertex data
- *.idx* A file that holds the index data (if available for this mesh)
- *.tex* A file that holds a texture description combined with the texture data

To identify which files belong together they have the same file name, but a different extension. The files also have a unique MD5 sum over the data. This way duplicate data will only be loaded once. The framework has load functionality for this type of meshes and uses the MD5 sum to load identical meshes and textures only once. Only few texture formats are supported because the example models that were used all had the same texture format.

Now that meshes could be loaded, light sources had to be implemented. The framework only uses spot lights, but most techniques can be adapted easily to use directional lights or point lights. How these techniques need to be adapted for different types of light sources will be described in Section 3.2.4.

The reason for only supporting spotlights is that spotlights have a position and a cone in which they shine their light. Cameras also have a position and a cone (or frustum). This means that the scene as seen from the light can be made visible. This is implemented as the *light view*. Every light source in the scene can be treated as a camera, making it possible to visualize all geometry a light shines on. To enable the user to further manipulate the scene, all cameras and lights can be moved and their field of view can be adjusted.

Some shadow techniques create soft shadows like they are cast by an area light. To support this, light properties can be modified and the light source's size and color can be changed.

The framework has the possibility to take screen shots from a window or from the scene, using all implemented shadow techniques.

Finally one of the most useful features of the framework is to save and load scenes. All geometry, camera, lights and light settings are saved to a file and can be loaded later.

Technique implementation

As mentioned earlier, the application framework is meant to demonstrate the shadow techniques. To make the implementation of shadow techniques as convenient as possible, all shadow techniques are called by the demo application in the same way. This process is described in this section.

A shadow technique is responsible for its own resources. It can set up shader variables, and render extra geometry if necessary. The application calls a technique on three occasions:

- *At the beginning of a frame.* This enables the technique to initialize for a frame.
- *For every light.* The application calls the technique for every light in the scene that casts shadows. This enables the technique to do per-light calculations.
- *For every viewport that uses the specified technique.* Multiple viewports can show the same technique from a different camera angle. These techniques all use the same results from the per-light calculations.

While rendering the scene the scene can query the technique for light maps. This way these light maps can be visualized when a technique uses light maps, .

Result

The application framework is intended for loading and saving scenes and visualizing the different shadow techniques. More functionality was added as it became clear that in order to compare the shadow techniques, the user needs to be able to manipulate the scene. A separation between shadow techniques and the rest of the application was made to make implementation of techniques easier.

3.2.4 Implemented techniques

This section will describe the implementation of the shadow techniques in the previously described framework.

Techniques

The last section described how the techniques are called by the application framework. The technique is responsible for rendering the geometry with its different shaders. To allow for easy adaptation of techniques the shaders were split up into several parts that can be connected using include files. Every technique consists of a combination of the following parts:

- *Vertex shader* – Transforms the vertex and calculates distances to the light sources.
- *Geometry shader* – If a geometry shader is available, it checks if primitives should be rendered, and can create more primitives if necessary.
- *Pixel shader* – Uses the generated values from the vertex shader to calculate the final pixel color. For every light source the following functions are called:
  - *calculate_shadow* – Calculates how much of the light reaches the current pixel. This function can be overridden to implement other techniques.
  - *calculate_phong* – Calculates the color value of the current pixel when it is fully lit by the light source.
- *Render states and settings* – Specify all technique specific render states.
- *Technique description* – Specifies the vertex shader, geometry shader and pixel shader that are used for this technique. Also contains render state changes.

This separation of the parts mentioned above is done to be able to create multiple shadow techniques while reusing as much code as possible.

No shadows

This technique is purely intended to show the effect of the other techniques. The *No shadows* technique renders the scene with per-pixel Phong [Pho75] shading. The light sources only light the objects inside their cone. Areas that face away from the light are lit by an ambient light. All other shadow techniques use this same shading model. For this technique, the calculate_shadow just returns 1 for the pixels that are lit by the light source.

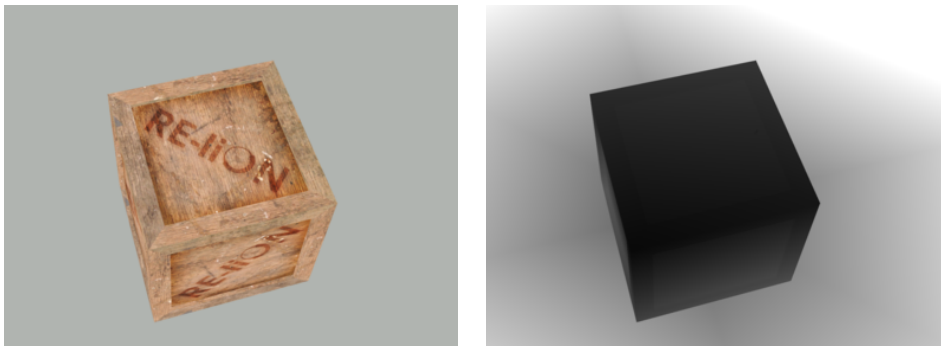
Projected shadows

The projected shadows technique was only added to illustrate this simple way of creating shadows. It solely supports shadows on a ground plane. This is done by first rendering this ground plane. For every light, the scene geometry is rendered in black using a transformation matrix that projects all geometry from the light source to the plane. After this all geometry is drawn using the *No shadows* technique.

Projected shadows can be used as a reference while casting shadows on a ground plane. They were only implemented as a simple example of shadows.

Standard shadow mapping

The standard shadow mapping technique is the base technique for all image based shadow algorithms. The technique implementation does the following: For every light source the scene is rendered to a light map, in this case an off-screen texture with 32-bit floating point precision. Instead of rendering the texture and lighting, every visible object's distance to the light source is saved. Figure 3.3 shows the scene as seen from the light without any shading and the corresponding depth map. Objects that are far away from the camera will show on the light map with a value close to 1 (white) and objects that are close to the camera will appear closer to 0 (black).



(a) The scene as seen from the light (b) The light map for this light

3.3: A light map stores the distance of an object to the light source.

After the light maps are created for every light source, the scene is rendered one more time from the camera's point of view. For each light a pixel shader looks up the light map value. If this light map value is bigger than the distance of the pixel to the light, this pixel is in light. When the pixel is in light, the color value of the pixel is calculated using Phong shading. When the pixel is in shadow, only the ambient color is used for this light.

All the contributions of the light sources in the scene are added to produce the color the pixel will have.

This technique uses one more pass than the number of lights in the scene. The final pass can be quite slow with a lot of lights because each vertex has to be projected onto the scene from every different light source. The light map generation is fast because no texturing or light calculations are necessary in these passes.

This technique can be implemented on hardware since pixel shader model 1.0. However, only a limited amount of texture look ups could be done using this model. Modern hardware no longer has this limit so many lights can be supported in the final pass.

Section 2.2.1 describes some optimizations to maximize depth buffer precision. The implemented technique uses a linear distribution of the depth values.

Implementing this technique was pretty straightforward using the application framework. Two shaders were created: one for the light map generation, and one for the final combining pass. The shader for light map generation just stores the distance from every rendered pixel to the light in the light map. The shader used for the final pass implements a different calculate_shadow function, which compares the camera distance to the value from the light map. If this distance is greater than the light map value, it returns 0 because these pixels are in shadow. Otherwise it returns 1.

The first implementation suffered a lot from surface acne. Surface acne can be prevented by adding a bias to the shadow test. The application framework allows this bias to be adjusted by hand.

Another way to eliminate surface acne is to only render the back-facing polygons to the light map. No polygons facing the light will suffer from surface acne but back facing polygons can still have this problem. However, because polygons that face away from the light are not lit by the light source, this problem is not visible. This works as long as an object is watertight. Objects with cracks in them (or with polygons left out to reduce the vertex count) produce holes in the shadows as in figure 3.4. Only rendering the back faces can be achieved by changing the culling method for the light map generation pass.

3.4: Holes in the shadow

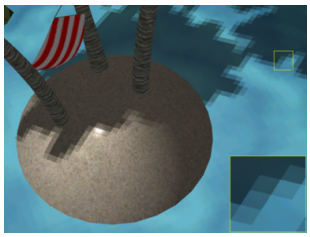
Results of both the standard shadow mapping and the back face-only shadow mapping can be found in appendix refapp:figures.

Percentage closer filtering

The previous technique creates shadows with hard shadow borders. Objects are either totally in light, or totally in shadow. When the light source is far away from the geometry, or a small light map is used, shadows can appear 'blocky'. To hide this effect, the shadow map can be filtered using percentage closer filtering.

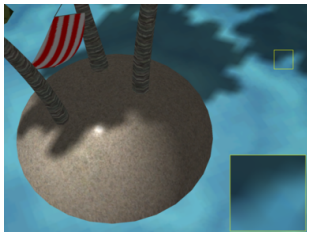
Percentage closer filtering was implemented by adjusting the calculate_shadow function. The rendered pixel is tested against 4 light map values instead of just to one. The results of these tests are added and divided by 4. The result of this division is returned by calculate_shadow.

Filtering the shadows in this way allows objects to be partially in shadow. Because only 4 samples are used, banding occurs. Figure 3.5 shows this. Because the combination shadow tests is divided by 4, only 5 possible amounts of shadow are possible. To decrease banding more light map samples can be used. Using 9 points percentage closer filteringhides some of the banding artifacts.



3.5: Banding

A better solution is linear interpolation of the light map samples. Figure 3.6 illustrates this. Linear interpolation can be done by sampling the light map points, and using the texture coordinates to calculate how much each sample contributes to the final result. Direct3D 10 also provides a sampler type that does exactly this, so this sampler is used in the implementation.



3.6: Linear interpolation

The best results can be obtained when using more light map samples in combination with linear interpolation. Appendix refapp:figures gives examples of all described percentage closer filtering variants.

Because percentage closer filtering uses a larger area of the light map, it can increase surface acne. This means the bias has to be increased, or back-face rendering should be used.

Percentage-closer soft shadows

Percentage closer filtering produces soft shadow borders. These soft borders somewhat resemble the penumbras that area light sources create. However, penumbras change in size when a shadow receiver is further away from a light blocker.

With percentage closer filtering, softer shadows can be created by increasing the filter kernel size. To simulate penumbras, shadows created by light blockers that are close to the receiver should use a smaller filter kernel than when the light blocker is further away from the receiver.

To achieve this varying filter size, the distance from the blocker to the receiver has to be known. That is why the percentage-closer soft shadows algorithm first does a blocker search.

To do this, the position of the current pixel on the light map is calculated. Now multiple points around this position are sampled, and the distance of the point that is closest to the light source is saved. When this distance is smaller than the distance of the current point to the light source, it means a blocker is found. The distance of the point to the light source and the distance of the blocker can be used with the light size to calculate the penumbra size. This penumbra size is then used to filter the current point using percentage closer filtering.

For this technique, the calculate_shadow function had to be adapted to do the blocker search, penumbra size calculation and variable filtering. Shader model 4 supports dynamic branching, which means that when a point is not in shadow it does not have to be filtered. This results in a performance increase.

This technique uses a lot of texture look ups in the blocker search and the percentage closer filtering. Because it can use really big filter sizes surface acne gets worse, so bigger biases have to be used.

Variance shadow mapping

In contrast to percentage closer filtering and percentage-closer soft shadows, variance shadow mapping attempts to filter the light map using a statistical approach. For this approach, the light map not only has to store the distance to the light, but also this distance squared. This means that the light map generation pass has to be adapted to hold this information.

The light map distances are stored in a R16G16F texture, that holds 16-bits floating point values for every red and green component. Light maps used in the filtering methods described before all used 32 bit floats, but since Direct3D 10 cannot filter R32G32F textures, the 16-bits variant is chosen. After the light map is generated, it is filtered using a Gaussian filter. Variance shadow mappingneeds the mean distance and the mean square distance around a point, so the more the light map is filtered, the smoother the results will be.

For the final pass, the calculate_shadow function has to be adapted. This function first samples the light map to get the mean distance d and the mean

squared distance s. A anisotropic filtering sampler is used to obtain these values, so the data is filtered even more. With these values, the variance is calculated. With this variance the probability that the pixel is in shadow can be approximated. This probability is used to calculate the amount of shadow the current pixel receives.

Variance shadow mapping uses a large amount of filtering. This ensures soft shadow borders and smooths out effects like surface acne. The drawback of this is that an object that occupies only a few pixels on the light map will cast a shadow that appears much lighter than objects that occupy a big part of the light map.

Variance shadow mapping filtering uses one or two extra passes per light to do the light map filtering, depending on the filter that is used. However, variance shadow mapping produces good results with much smaller light maps because of the heavy filtering.

Brute force shadow volumes

The techniques described before are all image based techniques. They are fast and scale well with scene complexity, but they all suffer from artifacts and resolution problems. An alternative to image based techniques are the geometry based ones, and especially the shadow volumes technique. To create shadow volumes in dynamic scenes, the objects have to be extruded from the light source every frame. This used to be done on the CPU, consuming a lot of processing time.

With the new geometry shaders in Direct3D 10, shadow volume generation can be done on the GPU. The brute force shadow volumes technique uses these geometry shaders.

This technique works in several passes. First, the scene is rendered from the camera view, using only ambient lighting. The stencil buffer is reset to zero. After this pass, the z-buffer is initialized for the scene.

Now, two passes are required for every light source.

The first pass extrudes the shadow geometry away from the light source. This is done in the geometry shader. The geometry shader gets a primitive as input, in this case a triangle. For this triangle, the normal is calculated. If the triangle normal points in the opposite direction of the light vector, this triangle faces the light. For every triangle that faces the light, six extra polygons are created by the geometry shader. These polygons form three quads that extend as far as the light can shine. Every triangle that does not face the light is discarded.

The rendering of these generated triangles is done without writing color or

depth information. For every polygon that faces towards the camera, the stencil buffer is increased, and for every polygon that faces away from the camera, the stencil buffer is decreased.

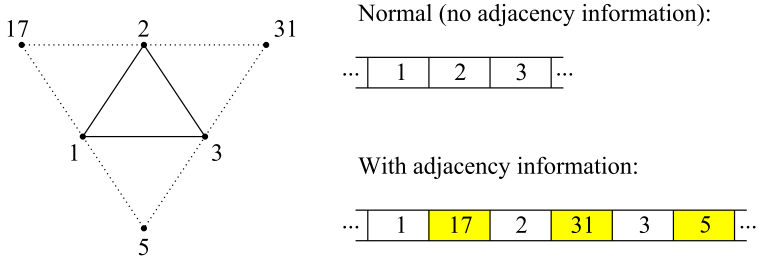
The second pass renders the scene, but only outputs the light that is cast by the current light source. When the stencil buffer is equal to 0, this light is added to the render target. All areas of the stencil buffer that are not equal to 0 remain untouched.

Now, the stencil buffer is cleared, and the two passes start from the top for the next light source.

After the last light, rendering is done. This technique produces hard shadows as if they were cast by a point light. This technique draws a lot of very large polygons. Graphics hardware can only draw a certain amount of points per frame. This is called the fill rate. Brute force shadow volumes eat a lot of fill rate, especially in large scenes, with lots of polygons. Many of the extruded polygons do not even contribute to the shadow in the scene, so they should not be drawn. The next sections describe smarter ways to do shadow volume extrusion.

Silhouette detection

To create shadow volumes using less fill rate and polygons, only the silhouette of an object has to be extruded in the light direction. To extrude the silhouette edges, these edges have to be detected first. This can also be done in a geometry shader. To do this, adjacency information is needed for the rendered geometry. This adjacency information is stored in the index buffer.



3.7: Storing adjacency information in the index buffer

Figure 3.7 shows how this is done. Normally, an index buffer only holds the vertex indices for the triangles. Adjacency information is added between these indices. When the scene is rendered, and the primitive type is set to triangles with adjacency information, the geometry shader receives information on all six vertices that belong to this triangle with adjacency. In stead of one triangle, four triangles are available in the geometry shader.

Silhouette detection using these four triangles is straight forward. When the triangle in the middle is facing the light, and one of the other triangles isn't, the edge between those two triangles is a silhouette triangle.

This silhouette detection is exactly what is needed to create shadow volumes that are more efficient. The only problem is creating the adjacency information.

For closed meshes that only contain edges that are shared by two polygons, the adjacency calculation is simple. For every edge, the other triangle is searched that shares the same edge. The point of the other triangle that is not in the shared edge is the point that needs to be inserted in the index buffer.

For meshes that contain holes or edges that are shared by more polygons, calculating adjacency information is more complicated. The application framework tries to solve this problem by first finding all possible neighbouring triangles, and then selecting the neighbour with the normal vector closest to the normal vector of the triangle.

When an edge is only connected to one triangle, the adjacency information will contain the same point twice. This can be detected in the geometry shader. An edge that only has one triangle attached to it is always a silhouette edge.

Z-pass shadow volumes using silhouette edges

Since the geometry shader has adjacency information available, it can detect silhouette edges. By only extruding the silhouette edges, the number of polygons that will be rasterized is decreased. The less polygons are rasterized, the smaller the amount of fill rate that is used, which means an increase in performance.

The geometry shader is responsible for the silhouette detection and generation of the extruded polygons. The next section will describe how this is done.

As input, the geometry shader receives a primitive with adjacency information. To the geometry shader, this is an array with 6 elements: the vertices of the triangle and its neighbours. Elements 0, 2 and 4 of this array are the vertices of the triangle. With these vertices, the triangle normal is calculated.

Next, the direction of the light to this triangle is calculated. Because the light is a spot light, this direction differs per triangle vertex. The light direction is for the entire triangle is approximated by the direction from the light source to the mean of the triangles vertices. This ensures that

a triangle always uses the same light direction. The first approach was to take the direction of the light to the first vertex of a triangle, because this requires less calculations. The problem here is that a triangle has to be checked multiple times, once as main triangle, and multiple times as neighbour of another triangle. In these checks, the order of the vertices can be different. This produces different light directions in different cases, and can lead to silhouette edges not being detected.

Next, the dot product of the triangle normal and the light direction is taken. If this dot product is greater than 0, the polygon is facing the light. When this is the case, the three neighbouring triangles are checked if they face the light. When a neighbouring triangle is found that does not face the light, it means that the edge between the main triangle and the neighbouring triangle is a silhouette edge.

When a silhouette edge is found, a new quad is created. This quad consists of the two points of the current edge, and two points that are moved away from the light source.

The geometry shader discards all polygons that face away from the light source. It keeps polygons that face the light source. Combined with the created silhouette quads, it outputs a shadow volume. This shadow volume is rendered using the standard Z-pass shadow volumes technique. For all polygons that pass the depth test the front facing shadow volume polygon increases a stencil buffer, while every back facing polygon decreases it. Area's that have a stencil value of zero are in light, everything else is in shadow.

This technique creates shadows with hard borders. It works in all cases where the camera is not inside a shadow volume. When the camera *is* inside a volume, this technique produces incorrect results. This is because the near plane clips the shadow volume the camera is in. To produce correct results, the front faces of this shadow volume should be drawn, but usually they are behind the camera. Z-pass+ shadow volumes forms a solution to this by capping the front of the shadow volumes in a separate pass. This was not implemented, because Z-fail shadow volumes also solves this problem.

Z-fail shadow volumes using silhouette edges

Z-pass shadow volumes gives incorrect results when the camera is inside a shadow volume. A solution to this problem is using Z-fail shadow volumes. The shadow volume generation of this technique is essentially the same, however, shadow volumes have to be capped at the back side. To do this, every vertex of every polygon that faces the light is moved away from the light source. This creates the caps for the extruded mesh. These polygons

have to be flipped, because their normals are pointing in the wrong direction. This is done by reversing the order of the vertices of the polygon.

Now that the geometry shader creates the capped shadow volumes, the only thing that has to be changed is the stencil behavior. Instead of increasing the stencil buffer when the depth test passes on a front facing polygon, the stencil buffer is increased when the depth test fails. This also goes fort he back facing polygons; when the depth test fails on such a polygon, the stencil buffer is decreased, otherwise it is kept the same.

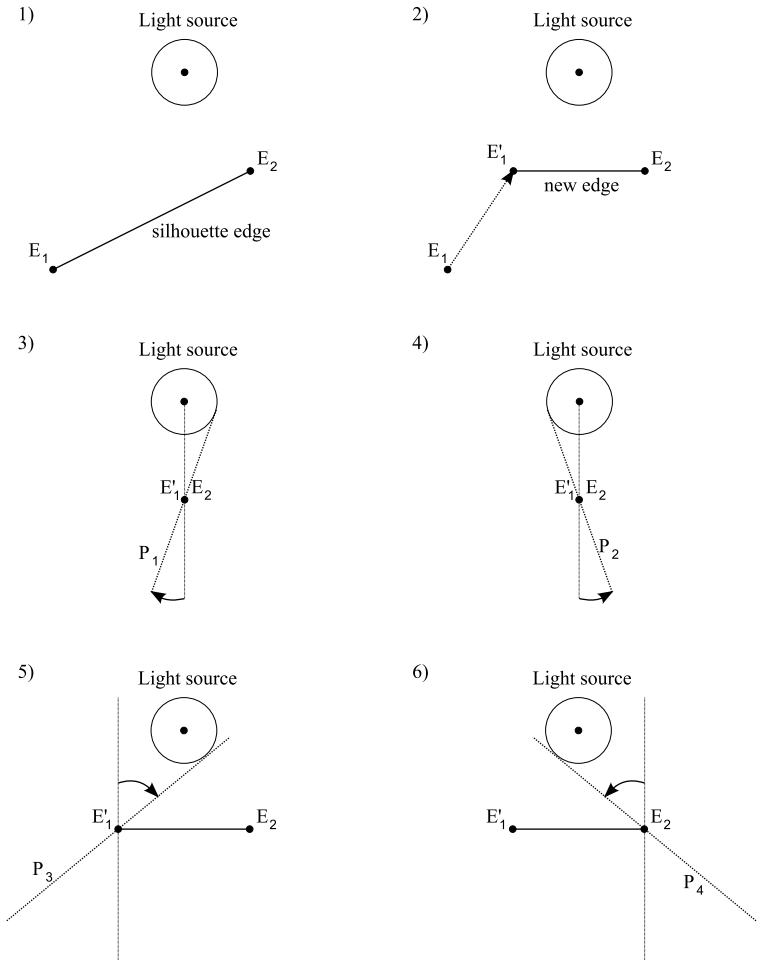
When rendering the shadow volumes, they should not be clipped by the far plane. Z-fail rendering moves the clipping problem from the near plane to the far plane. All shadow volumes that are clipped by the far plane produce incorrect results. There are two ways of solving this problem: moving the far plane further away from the camera, or turn off clipping from the far plane. Since Direct3D 10 can turn of far-plane clipping using a rasterizer state, this solution was chosen. Using this rasterizer state, the generated shadow volumes create correct results in all situations.

Penumbra wedges

Shadow volumes produces alias-free hard shadows using geometry. With geometry shaders, this can even be done in hardware. Penumbra wedges are an extension to shadow volumes that use geometry to create soft shadows.

First, the scene is rendered using standard shadow volumes. Depth information of the scene is kept in the depth buffer. Now, the depth buffer is bound to the render pipeline as a resource. This means that while rendering the penumbra wedges the depth information of the scene is available.

After this, the penumbra wedges are generated by the geometry shader. This is done by detecting the silhouette edges the same way it is done for shadow volumes. Using these silhouette edges, the penumbra wedges can be created. How this is done in the geometry shader is illustrated in figure 3.8.



3.8: Creating the penumbra wedges in six steps

1) First the distance is calculated between the light source and the vertex closest to it, in the figure this is E_2.

2) The other vertex E_1 is then moved towards the light source until its distance is equal to the distance of the closest vertex E_2. Vertex E_2 and the moved vertex E_1' will form the new edge that will be used to create the wedge.

3) Now the plane is calculated that contains both the new edge and the light source's center. This plane is rotated around the new edge, until its distance to the light source is equal to the light size. This rotated plane P_1 will form the front or back of the wedge. Note that the camera in the figure has rotated around the image so that the vertices of the new edge are behind each other.

4) The back or front of the wedge, plane P_2, is obtained by rotating the plane with the same angle in the opposite direction.

5) After this the plane with normal vector parallel to the new edge that contains the first edge vertex E_1' is calculated. This plane is rotated around the vector that lies in the plane and is perpendicular to the new edge until it touches the light. The rotated plane P_3 is the first side plane of the wedge.

6) The same is done for the other edge vertex E_2 resulting in plane P_4.

The area contained by these four planes make up a wedge that always contains the penumbra generated by the light source. The geometry shader calculates these planes, and generates polygons for the wedge. The equations of the four planes ($P_1 \ldots P_4$) and the edge vertices (E_1 and E_2) are stored in every vertex that is output by the geometry shader. This way, the wedge information is available for all rendered vertices.

Rasterizing the penumbra wedges is done in the pixel shader. This shader is executed for every pixel in a wedge. For wedge rasterization, depth writing is turned off.

In the pixel shader, the x and y values of the current pixel are available. With these values and the information stored in the depth buffer, the point p of the geometry that this wedge covers can be calculated.

Every wedge has information about the four planes it consists of. Using this information, a check is performed to see if p lies inside the wedge. If this is not the case, the pixel is discarded and the pixel shader is not executed further for this pixel.

If a pixel *is* inside a wedge, the amount of light that shines on this pixel has to be calculated. This is done by projecting the edge E_1E_2 from p to a plane that goes through the light source. This projected edge on the plane is clipped to $([-1,1],[-1,1])$. These two clipped edge point coordinates are used to look up the penumbra coverage in a precalculated texture. In the pixel shader, the plane formed by E_1E_2 and the light source is still available. This plane can be used to see if p lies inside or outside the shadow volume. When p lies inside a shadow the image is brightened with the penumbra coverage, otherwise, the image is darkened.

3.3 Encountered problems during implementation

This concludes the description of the implementation part of this research. The discussion of the implemented techniques and the results of this research will be presented in the next chapter. In this section, some general difficulties that occured during the implementation traject will be described.

3.3.1 Demo application

Because this research is about the shadow techniques, the plan was to create a small demo application to be able to start developing the shadow technique shaders as soon as possible. While developing the shadow techniques, more and more options were needed in the application framework. This started with the ability to move the camera, but more options had to be added, like the support for light sizes and colors. This caused the demo application to grow into a large program with a lot of options. This took a lot of time that was meant to be spent on the implementation of the techniques.

3.3.2 Hardware drivers

While rendering 3D graphics performance depends on the hardware that the graphics are rendered on, *and* on the drivers for this hardware. Direct3D 10 is only available for the Windows Vista operating system. This operating system is relatively new, and introduces a new driver model. In addition to this, Direct3D 10 provides a whole new list of capabilities that graphics hardware has to conform to.

This is why, with the release of the NVIDIA GeForce 8800, the graphics card that was used in this research, the drivers for the GPU were not quite ready yet. Drivers existed for windows vista, but a lot of the functionality that was exposed by Direct3D 10 did not yet work as fast as it was supposed to. These unfinished drivers also caused a lot of crashes that required that the graphics driver was restarted. While developing the demo application, this slowed down development drastically.

Fortunately new NVIDIA drivers come out frequently. After installing new drivers, performance gains of 10%, 20% or even 50% could be obtained (this was a good thing, because even demos from the Direct3D 10 SDK ran at 4FPS).

3.3.3 Renderer2

Renderer2 was developed to loose the overhead of the original Lumo Renderer. When this research started, it was still in development, so API changes could sometimes occur. Finally, this was solved by freezing the API for the implemented Direct3D 10 driver. This means however that when the the driver is to be used in the current version of Renderer2, some adjustments have to be made.

Chapter 4

Evaluation

In order to evaluate the implemented shadow techniques, they will be compared to each other. Different techniques respond differently in different situations. What technique responds best in what situation?

In order to be useful for a 3D simulation, generating the shadows should be possible in real-time (20 or more frames per second). The amount of memory the shadow technique uses can also be important, because enough memory should be available for the rendering of the rest of the simulation's graphics.

The shadows should appear realistic, but how is realism measured? An aspect of realism is whether the generated shadows conform to the light model described in section 2.1. This means that area lights should have umbras and penumbras. The light sources used in the test scene were all area lights, so penumbras should be visible in the results. The most important part of realism however, is that the shadows *look real*.

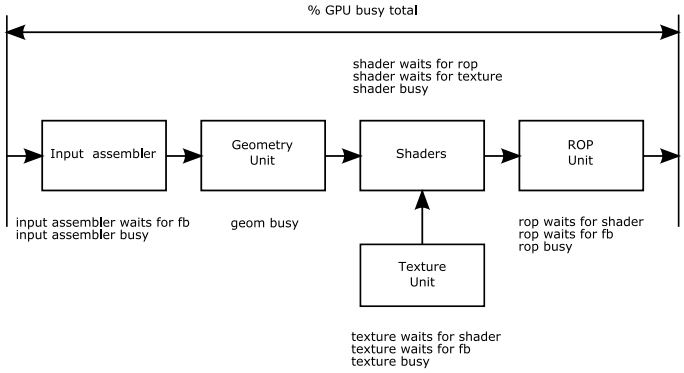
4.1 Tests

This section describes the tests that were taken to evaluate the shadow techniques.

4.1.1 Technique performance

As mentioned before, real-time performance of the shadow techniques compromises an important aspect for 3D simulations. In order to test for real-time performance, three measurements have been made for every shadow technique:

- *Frame time*, the time it takes to render one frame using a shadow technique. When it takes more than 50ms to render a frame, the technique is no longer has real-time performance.
- *GPU counters*, GPU's have built-in counters that store the amount of time spent in different parts of the rendering process. In 3D simulations shadows are not the only graphics that need rendering. There needs to be enough GPU time left to render the rest of the scene. This is why the GPU counters should give insight in how heavily the techniques burden the GPU.



4.1: GPU counters

Figure 4.1 shows the available counters. They return the percentage of the time that was spent in the different parts of the GPU. These parts are:

- The input assembler, turns user-supplied data into GPU readable vertices.
- The geometry unit, adds the vertices together into primitives.
- Shaders, the vertex shader, geometry shader and pixel shaders process the vertex data and create output.
- Texture unit, used by the shaders to do texture look ups.
- ROP unit, combines the pixel color values with the frame buffer to create the final image.

These counters give results for every part of the graphics pipeline, except for one: The pasteurisation stage. This stage, which lies between the geometry shader and the pixel shader (or the vertex shader and the pixel shader, if no geometry shader is available), calculates the pixels that have to be drawn for every rendering primitive. The amount of pixels that the rasterization stage can process is the available fill rate.

When many large polygons have to be drawn, the fill rate can get too high, and the rendering slows down. This means that the counters from the different parts of the render pipeline will return lower values, while the `% GPU busy total` counter remains high.

Shadow techniques can be used in a number of different situations. This is why multiple test scenes were necessary. These test scenes differed in 4 area's:

- *Number of objects in the scene.* The amount of time it takes to render a scene using a specific technique depends on the amount of objects in the scene. Scenes with 1, 5, 10, 25, 50, 100 and 200 objects were created. This way the influence the number of rendered objects has on the performance could be studied.

- *Number of polygons per object.* This aspect of the scene was changed to see how well shadow technique performance scales with increasing geometry complexity. Two types of scenes were created: scenes with objects of approximately 100 polygons, and scenes with objects of approximately 1600 polygons. These numbers were chosen because of the availability of two models with those polygon counts. Originally, the scenes were also to be rendered using a model of 10000+ polygons. However, using some techniques this slowed down the rendering process so much that testing all situations would take too much time.

- *Number of lights in the scene.* This aspect of the scenes was changed to see how well techniques scale with the number of lights. Most simulations use multiple light sources, so it is important to see how the techniques respond them. Scenes were created with 1, 2 and 3 lights.

- *Size of the rendered area in pixels.* When rendering to a large area more pixels have to be drawn. This increases the workload of the pixel shaders. To study the impact of this extra work on the performance, the scenes were rendered to the screen at different resolutions. The render targets were always square in size. The sizes that were taken for measurements are 25, 50 and every multiple of 100 up until 1200. A 1200 by 1200 square render target just fits in the screen, because the monitor that was used allowed for a maximum resolution of 1600x1200. Modern graphics hardware can render to a texture at much higher resolutions (even up to 4096 by 4096), but the measurements are done while rendering *to the screen.*

This resulted in 588 test scenes. For every shadow technique, all tests scenes were rendered using that technique. The used techniques can be divided in

groups of techniques that are expected to react to the performance tests in similar ways:

- *No shadows.* The scenes were rendered without shadows as a reference.
- *Standard Shadow mapping.* This is the basic implementation of shadow mapping.
- *Percentage closer filtering.* These techniques implement filtering by using percentage closer filtering. Different filter kernel sizes are available, so in total five versions of percentage closer filtering were tested: 2x2, 3x3, bilinear, bilinear 2x2 and bilinear 3x3. These techniques only differ in the number of texture fetches used for filtering. This means that they should react to the performance tests roughly the same. Techniques with bigger filter kernels should be slower than techniques with small ones.
- *Percentage-closer soft shadows.* This technique simulates penumbra size by using a variable filter kernel size.
- *Variance shadow mapping.* The variance shadow mapping technique uses a different approach to filtering than percentage closer filtering. This means that variance shadow mapping will react differently to the performance tests.
- *Shadow volumes.* Two versions of the shadow volumes technique were implemented: Z-pass shadow volumes and Z-fail shadow volumes. They use the geometry shader to determine which silhouette edges should be extruded, and extrude only these edges.
- *Soft Shadow volumes.* This technique creates soft shadows using a geometry based algorithm.

4.1.2 Shadow realism

Shadow realism has been tested in two ways. One way was testing test how much the shadows created by the different techniques differ from the shadows created by a reference technique. The other way was to create some scenes with shadows, and ask people which technique produced the most realistic shadows.

For these test, scenes were needed that have light sources in different situations. These are the scenes that were created:

- *Car (1)*, this scene shows a car standing on a box. There are two area light sources above the car that are relatively close to the scenes

geometry, so they produce soft shadows. Since the lights are close to the scene, shadow mapping artifacts will not be visible.

- *Boxes (2)*, this scene shows 50 boxes on a gray plane. The camera is situated a bit further away from the scene, so the depth complexity is higher. There is one area light source that is placed further away from the scene also, lighting the scene from above.
- *Disco box (3)*, this scene shows a box lighted by three area lights in the colors red, green and blue. The light sources are located close to the box.
- *Island (4)*, this scene shows an island in the see. The island is lit by one light source which is so far away from the scene that no umbra/penumbra effects can be seen.
- *Inside warehouse (5)*, this scene shows a warehouse that is lit by a single area light. Umbra and penumbra effects should be visible because the light is relatively close to the light blockers.
- *Outside warehouse (6)*, this scene shows the same warehouse from the outside. The camera and light source both are placed far away from the scene, but close to each other. This means that all shadow mapping resolution should not be a problem. Because the scene is big, it has a lot of depth complexity. This is why the bias for the shadow mapping related techniques had to be high.

Image quality

According to the lighting model described in section 2.1, area lights should cast shadows with umbras and penumbras. To test if the shadow techniques behave as correct area lights, several scenes were created and rendered using the different techniques.

These renders were compared to a reference image that contained the correct umbras and penumbras for the area lights used. An area light is a volume that shines light upon the scene. This volume can be seen as if it is filled with an infinite number of point lights. To create the reference image, 1024 renders were made using only point lights at random places inside the area light volumes. These renders were combined into one image by averaging the picture values. Because of the random distribution of the point lights in the area lights, the correct shadows are obtained.

To see how much the images created by the shadow techniques differ from the reference image, the color value of every pixel was used as a vector. Because the comparison is between dark and light (shadow and no shadow),

the colors were converted to YUV. The Y-component of a color in YUV space holds the luminance of a pixel, and the luminance is responsible for dark and light. The distances between the Y components of the reference image and the technique image each were squared and summed. From the total squared distance the square root was taken producing the Root Mean Square distance (or standard deviation between the images). The higher the RMS, the greater the distance between the images.

Note that the Root Mean Squares were taken for images with a resolution of 600x600 pixels. Increasing or decreasing the image resolution will change the differences between the images, but the order will stay the same. This is because some technique experience resolution problems that are better visible at higher resolutions.

Survey for realism

To get an idea of what people perceive as realism a web survey was created presenting the previously described scenes to people. This websurvey can be found in appendix A. For every scene, people were presented 9 images. These images consisted of 8 images created using the shadow techniques and the reference image. People who entered the web survey were asked to rank the different techniques from 1 (most realistic) to 9 (least realistic). They could also specify how hard it was to see a difference between the images.

4.1.3 Memory usage

For every technique, the memory that it uses is calculated. The total amount of memory used is compared to the amount of memory that would be used when the scene was rendered without shadows.

4.2 Results

In this section, the results of the tests are presented.

4.2.1 Technique performance

As stated before, the shadow techniques were divided into 6 groups. For every group is described how it reacts to changes in the number of objects rendered, the number of polygons per objects, the number of lights in the scene and the size of the render target.

The tests were performed using an NVIDIA 8800 GTS with 640 MB of video memory. The instrumented drivers for this GPU were used to be able to sample the GPU counters. Due to lack of time, all tests were performed twice. Some artifacts can be seen in the graphs. These artifacts appear randomly and are caused by the operating system and other applications running. By performing multiple tests, these artifacts can be filtered away.

Table 4.1 shows the results of the performance and bottleneck tests. On the left side are the abbreviations for the techniques: no shadows (No), standard shadow mapping (SSM), percentage closer filtering (PCF), percentage-closer soft shadows (PCSS), variance shadow mapping (VSM), shadow volumes (SV) and soft shadow volumes (SSV).

The next four columns are the results of the performance tests. The first performance limit is the maximum number of low poly objects that can be rendered while maintaining real-time performance. The resolution at which this is achieved is also specified: 200 at 900x900 means that a maximum of 200 objects can be rendered to a render target of 900x900 pixels in less than 50ms.

The second column is how many low poly objects can be rendered at the maximum resolution of the render target that was tested (1200x1200). The third and fourth columns show these values for the high poly objects. The last column, bottleneck, shows which part of the rendering pipeline is the bottleneck for this technique.

Technique	**Low poly performance limit 1**	**Low poly performance limit 2**	**High poly performance limit 1**	**High poly performance limit 2**	**Bottleneck**
No	–	–	–	–	shaders
SSM	–	–	200 at 500x500	140 at 1200x1200	shaders
PCF	200 at 900x900	135 at 1200x1200	200 at 200x200	82 at 1200x1200	shaders
PCSS	150 at 25x25	1 at 600x600	100 at 25x25	1 at 600x600	shaders
VSM	200 at 1000x1000	150 at 1200x1200	150 at 25x25	80 at 1200x1200	shaders
SV	150 at 25x25	75 at 1200x1200	10 at 25x25	8 at 1200x1200	fill rate
SSV	50 at 25x25	25 at 1200x1200	5 at 25x25	4 at 1200x1200	fill rate

Table 4.1: Results of the performance and bottlenecks test.

The following paragraphs illustrate how these results were obtained. For every technique, an overview is given of the performance using 3 light sources. The number of objects and the size of the render target in pixels are plotted against the frame times. The limit for real-time performance is set at 20 frames per second, which means that frame times should not rise above 50ms.

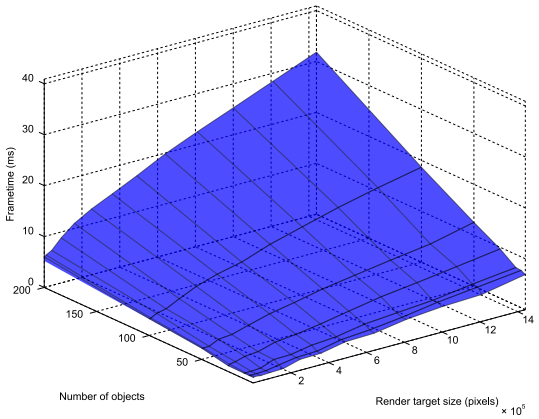
To see where the bottleneck for each technique lies, five graphs show how the different parts of the GPU respond to the techniques. Using the GPU counters, the bottlenecks of the rendering process can be found.

No shadows

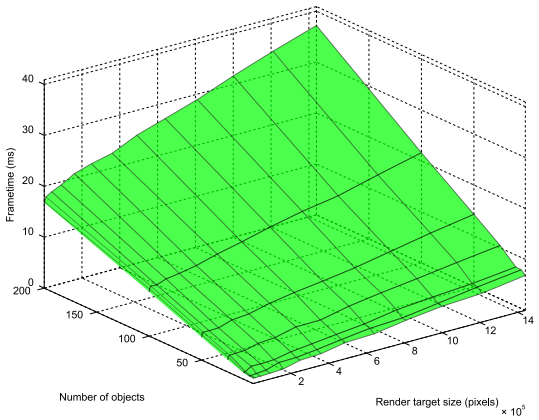
This technique is used as a reference technique. All objects are rendered lit by the light sources in the scene, but no shadows are created. The light sources are spot lights, they only light the parts of the scene that is inside their cone.

Figure 4.2a gives an overview of the effects that render target size and the number of objects have on the frame time for objects with approximately 100 polygons. The frame times were measured at a resolution of 1ms. This causes some noise at low resolutions or object counts. Since all the frame times are below 50ms, this is a real-time technique.

Figure 4.2b shows the frame times for the objects with the higher polygon count. The frame times are higher, but they are still below the 50ms, so the technique produces real-time results in all tested cases.



(a) 3 Light sources, low poly.



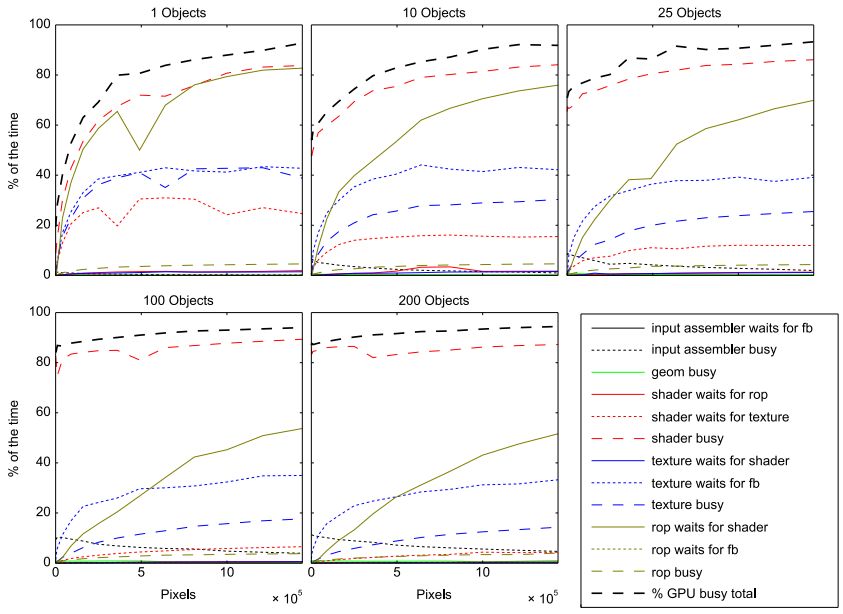
(b) 3 Light sources, high poly.

4.2: Frame times for the *no shadows* technique.

The graphs shows that the frame times depend on the number of objects rendered in an almost linear way. This is because a draw call is made for every object in the scene. These draw calls are sent from the CPU to the GPU. Since CPU/GPU synchronisation is needed for these calls, draw calls cannot be processed in parallel. Every draw call takes the same amount of work, so more draw calls cause a linear increase in the frame time.

The more objects in the scene, the more frame time needed per pixel. This

is because all the objects have to be rasterized, and for all the rasterized objects the pixel shader has to be executed.



4.3: Performance counters for the *No shadows* technique.

Figure 4.3 shows the GPU counters. When the scene does not contain many objects, the GPU is idle part of the time. There is not enough work to be done to keep the GPU busy. The graph shows that GPU spends the biggest part of the time executing the shaders.

The shaders are the bottleneck in this technique. This is because the ROP unit is waiting for them for a big part of the time. For low object counts, a significant part of the time is spent on the texture lookups. Also, the texture unit is waiting for the frame buffer.

When the number of rendered objects increases, the percentage of the time spent waiting decreases. Only the percentage of the time spent in the shaders stays roughly the same. The relative waiting time for the ROP decreases, because the shaders produce more output, so the ROP has more color data available.

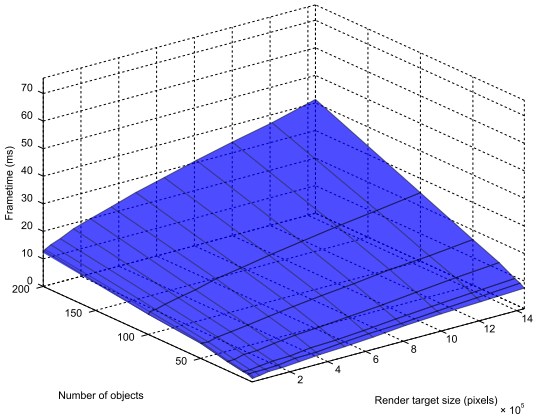
When more objects are rendered, the relative time spent in the input assembler increases too. This is another sign that the GPU is not fully used. More time in the input shader means more primitives, and should lead to more time spent in the shaders. This is not the case, so the GPU still has some available shader processing power left.

Standard Shadow mapping

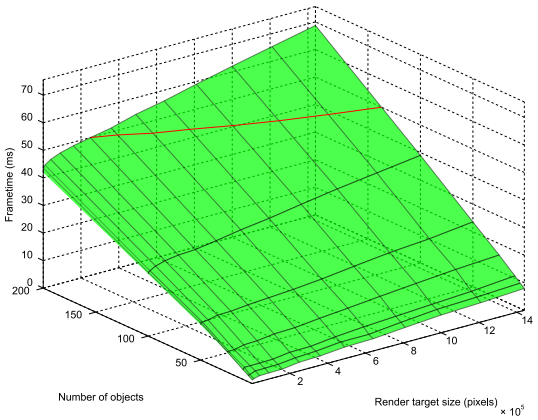
This technique implements the basic variant of shadow mapping. For every light, a light map of the scene is rendered. After this, the scene is rendered using the light maps to create the shadows. Figure 4.4a shows that this technique produces real-time results for the low poly objects.

With a small number of objects, the frame times measured for this technique are almost equal to the *No shadows* technique. With 200 objects however, the frame time increases with 6ms for small rendertargets to 10ms for the largest. This is because when more objects are rendered, more pixels must be shaded, and the pixel shader for this technique uses more instructions than the one for the *No shadows* technique.

Figure 4.4b shows how the technique responds to high poly objects. For large object counts, it cannot produce shadows in real-time anymore. The limit for real-time performance lies at a render target size of 500x500 with 200 objects, or at 140 objects using a render target size of 1200x1200.

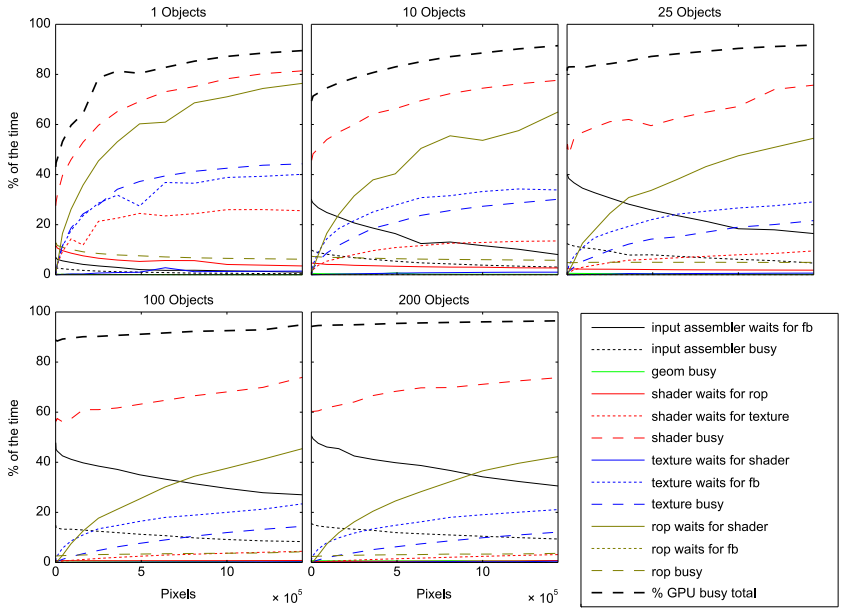


(a) 3 Light sources, low poly.



(b) 3 Light sources, high poly.

4.4: Frame times for the *standard shadow mapping* technique.



4.5: Performance counters for the *Standard shadow mapping* technique.

Figure 4.5 shows the performance counters for the standard shadow mapping technique. These graphs have a lot in common with the graphs of the *No shadows* technique. The bottleneck are still the shaders. There are however some differences.

With largers number of objects, the input assembler is waiting for the frame buffer for a relatively long time. This is caused by the rendering of the light maps. The GPU cannot continue rendering the next light map or the scene when the previous light map is still busy. This is why the input assembler has to wait.

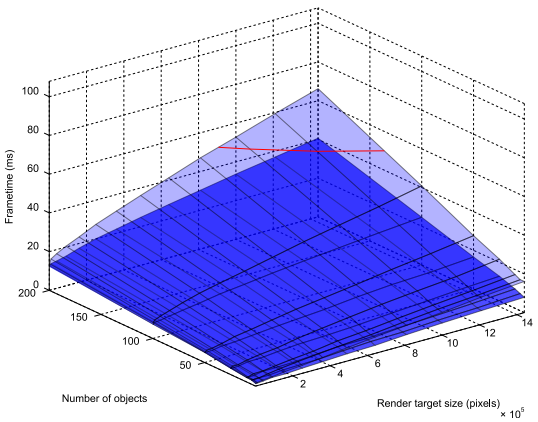
The graphs show the results of the *entire scene*. This includes the rendering of the light maps. Rendering these light maps uses a very simple pixel shader. This is why the relative time spent in the shaders is lower than that for the *No shadows* technique.

Percentage closer filtering

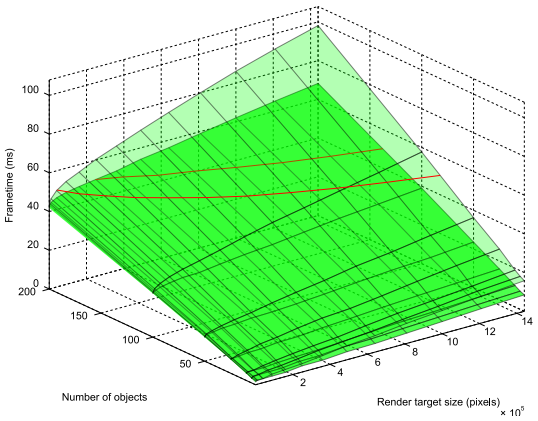
5 Variants of the percentage closer filtering technique were implemented. They all have different kernel sizes. Figure 4.6a shows the technique that needs the smallest frame time (percentage closer filtering through bilinear filtering) and the technique that needs the largerst frame time (percentage closer filtering with a 3x3 pixel kernel using bilinear filtering) with low poly objects.

The faster technique, bilinear filtering is the lowest blue surface. This technique produces real-time results in every tested situation.

The red line in the graph shows where the 3x3 percentage closer filtering technique no longer produces real-time results any more. With 200 objects, the largest render target size to produce real-time results is 900x900, and with the full 1200x1200 render target size, only 135 objects are supported to maintain real-time performance.



(a) 3 Light sources, low poly.



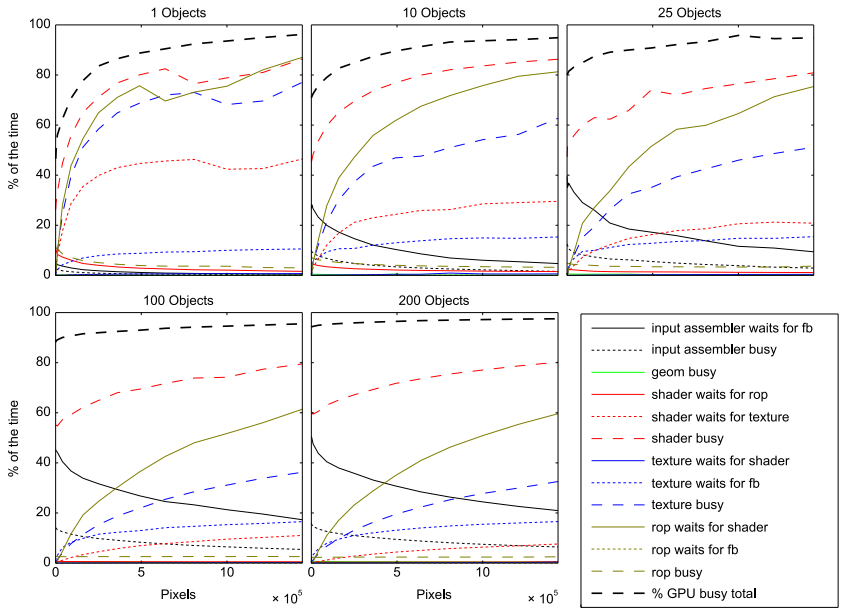
(b) 3 Light sources, high poly.

4.6: Frame times for the *percentage closer filtering* technique.

The bilinear filtering is implemented in hardware on the GPU that was used. Only one instruction is needed to do a bilinear shadow map compare. This explains why the frame times of the bilinear filtering variant of percentage closer filtering are almost equal to those of the standard shadow mapping technique; fetching a texture and fetching a texture using bilinear filtering takes the same amount of time.

The frame times for high poly objects is shown in figure 4.6b. For these objects, the fastest percentage closer filtering technique produces the same results as standard shadow mapping. For the 3x3 bilinear filtered version of

percentage closer filtering the limits for real-time performance lie at 200x200 pixels for 200 objects or 82 objects with a render target size of 1200x1200.

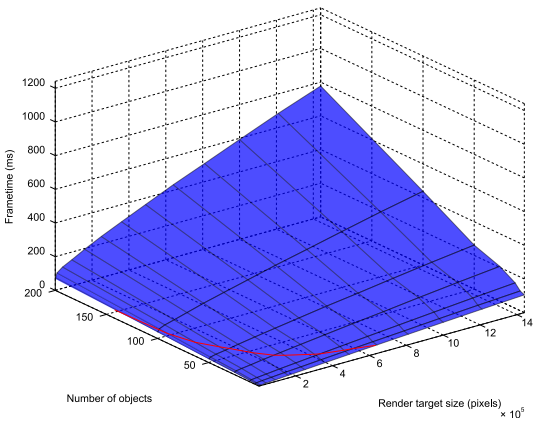


4.7: Performance counters for the *Percentage closer filtering* technique.

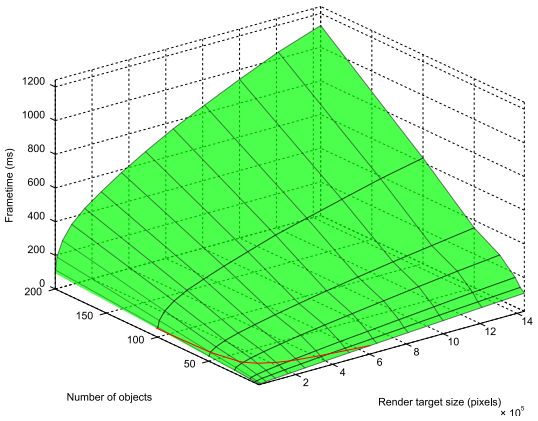
Figure 4.7 shows the GPU counters for the 3x3 bilinear filtered percentage closer filtering technique. The time spent in the shaders and the time the ROP is waiting for the shaders is roughly the same as with the shadow mapping technique, but a lot more time is spent in the texture unit. This is because the 3x3 bilinear filtering percentage closer filtering technique uses 9 texture lookups instead of 1. Still, most of the time is spent in the shaders, and the ROP has to wait for them. This makes the shaders the bottleneck for this technique

Percentage-closer soft shadows

The percentage-closer soft shadows technique uses percentage closer filtering with a variable kernel size to simulate penumbra regions in the shadows. For every rendered pixel, many pixels of the light map have to be examined to find blocking objects, and after that many pixels are used for the percentage closer filtering filtering. This can only be achieved using a very large pixel shader.



(a) 3 Light sources, low poly.

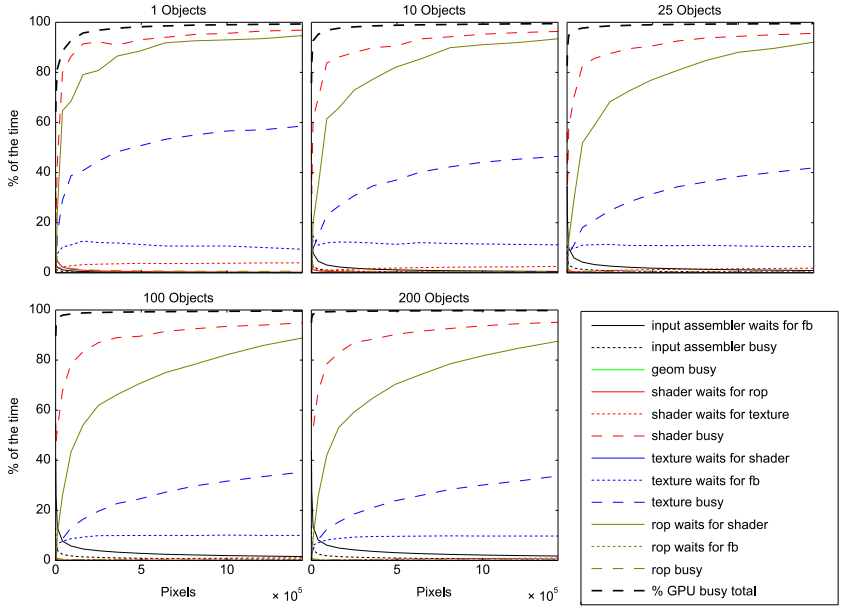


(b) 3 Light sources, high poly.

4.8: Frame times for the *percentage-closer soft shadows* technique.

Figure 4.8a shows that the Percentage-closer soft shadows technique does not perform well when the number of objects increases. Even with the low poly objects, real-time results are only achieved at a render target size of 25x25 with 150 or less objects or 1 object with a render target size of 600x600 or less. In all other cases, too many pixels have to be rendered and the GPU cannot execute the pixel shaders fast enough, because the they have too many instructions.

Figure 4.8b shows that with the high poly objects, the real-time limits do not differ very much (render target size of 25x25 with 100 objects or 1 object with a render target size of 600x600). In these situations the number of pixels that has to be rendered is relatively small. When the render target size increases, more pixels are needed, and the GPU does not have any shader processing power left.



4.9: Performance counters for the *Percentage-closer soft shadows* technique.

The percentage-closer soft shadows technique uses the same shadow maps as the percentage closer filtering and standard shadow mapping techniques. This means that the input assembler has to wait for the generation of the light maps. This waiting will take the same amount of time, because all actions that are performed are the same.

Figure 4.9 shows that the relative amount of time that is spent waiting by the input assembler is much lower than the time needed by the other two techniques. The shader and texture unit use a more time (approximately 4 times at much at low resolutions to 10 times as much for the highest resolution).

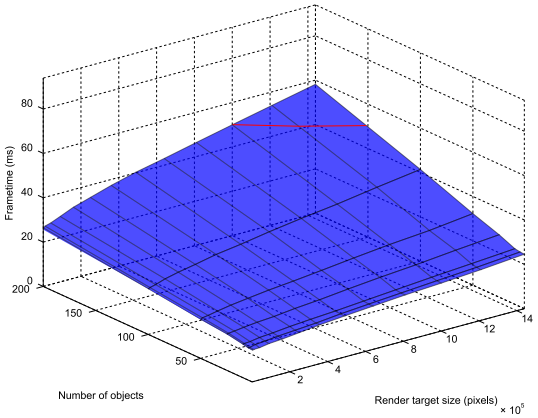
Because the shader is big and uses a lot of time, the ROP is constantly waiting for it. The shader just cannot produce the pixel color values on time. The texture unit is also used extensively, this is because of the high number of texture lookups that are needed to do the blocker search and filtering.

As mentioned before, this technique is an extension to the percentage closer filtering technique. It uses even bigger shaders, and these are the bottleneck.

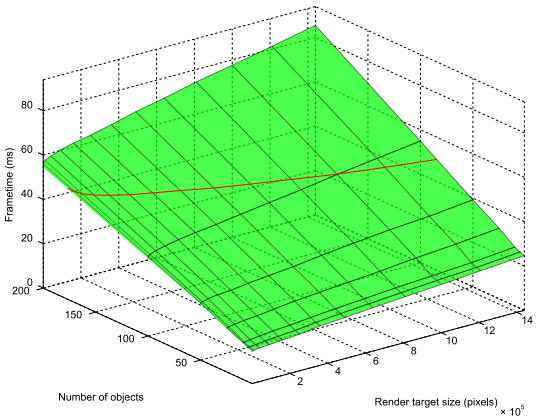
Variance shadow mapping

Variance shadow mapping uses a different approach to filtering than the former three techniques. Figure 4.10a shows the frame times for the low poly objects. This technique filters the light map using an extra pass. Since the light map is always the same size, this filtering takes a constant amount of time. This is why the frame time for 1 object with the smallest render target (25x25) already takes 14ms. Since the shader that uses the light maps is relatively small, the graph is not as steep as the one for the previous technique. The limits for real-time results lie at a render target size of 1000x1000 with 200 objects or 150 objects with a render target size of 1200x1200.

For the high poly objects, figure 4.10b shows that the technique technique can produce real-time results for 150 objects at a render target resolution of 25x25 to 80 objects at a resolution of 1200x1200.

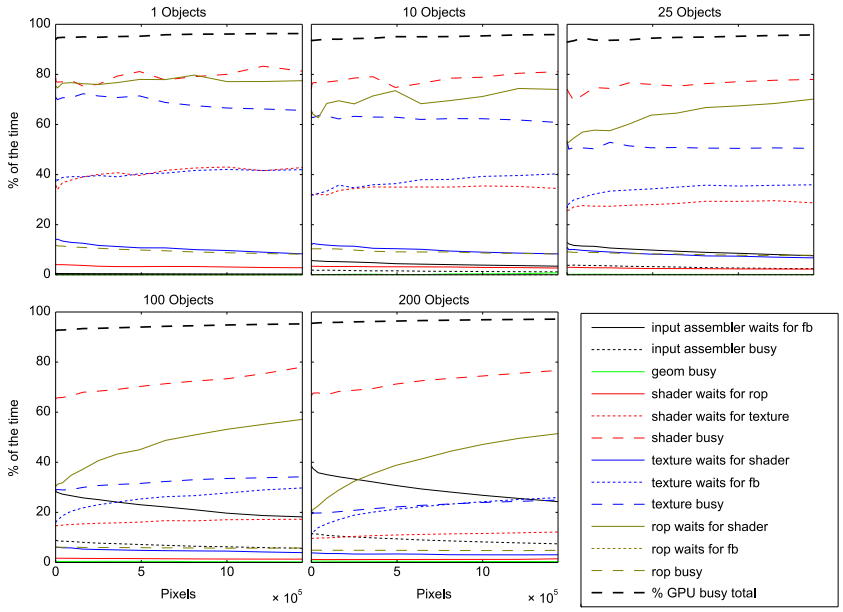


(a) 3 Light sources, low poly.



(b) 3 Light sources, high poly.

4.10: Frame times for the *variance shadow mapping* technique.



4.11: Performance counters for the *Variance shadow mapping* technique.

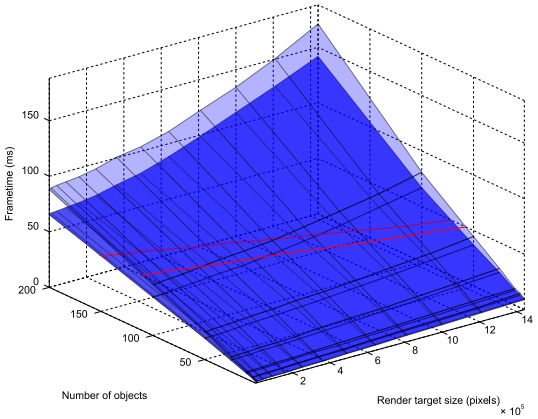
The performance graphs for the variance shadow mapping technique look different than the graphs for the other image based techniques. Figure 4.11 shows that the amount of time spent in the shaders and texture unit does not change much with the increase of the render target resolution, especially at low object counts. This is because the filtering of the light maps takes the longest time.

This filtering is nothing more than doing texture lookups and combining them. This is why the time spent in the shaders is only a little higher than the time spent in the texture unit. For increasing object counts, the graphs start to behave like those of the percentage closer filtering technique; in these cases the filtering of the light maps is no longer the part where most time is spent.

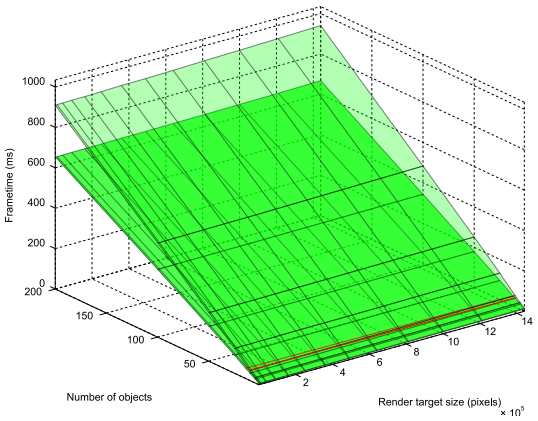
Again, the shaders are the bottleneck for this technique. The ROP uses a lot of time waiting for them.

Shadow volumes

The shadow volumes technique is the first geometry based technique that was tested. Figure 4.12a shows the frame times for this technique using the low poly objects. Two variants of shadow volumes were used: Z-pass and Z-fail shadow volumes. The lower blue surface are the frame times for Z-pass, the higher blue surface represents the frame times for the Z-fail technique.



(a) 3 Light sources, low poly.



(b) 3 Light sources, high poly.

4.12: Frame times for the *shadow volumes* technique.

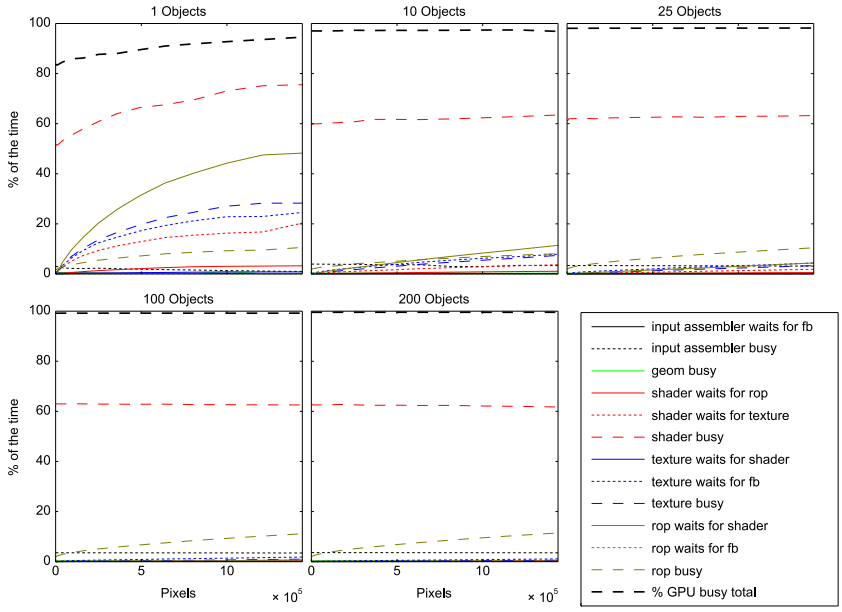
The Z-pass shadow volumes technique only renders the shadow volumes that are closer to the camera than the scene geometry. The Z-pass technique only renders the shadow volumes behind the scene geometry. In most cases a larger area of the polygons is behind the scene geometry, so more pixels have to be rendered. This increases frame times for the Z-fail technique.

The maximum number of objects that can be rendered using the Z-pass technique lies at 150 for a render target resolution of 25x25 pixels and at 75 for a render target resolution of 1200x1200 pixels. The Z-fail takes a little more time; it can produce real-time results for object counts up to 110 at a render target resolution of 25x25 pixels and up to 60 for a render target resolution of 1200x1200 pixels.

Figure 4.12b shows the results for the high poly objects. Note that the scale of this graph is different. The shadow volumes techniques are geometry based. This means that the geometry shader creates additional polygons for every polygon that lies at a silhouette edge. High poly objects have more polygons on the silhouette edges, so more extra polygons are created. These polygons use up a lot of fill rate, making the frame times increase

significantly.

Because of this fill rate increase, the maximum number of high poly objects that can be rendered lies around 8 for the Z-fail technique and around 10 for the Z-pass technique.

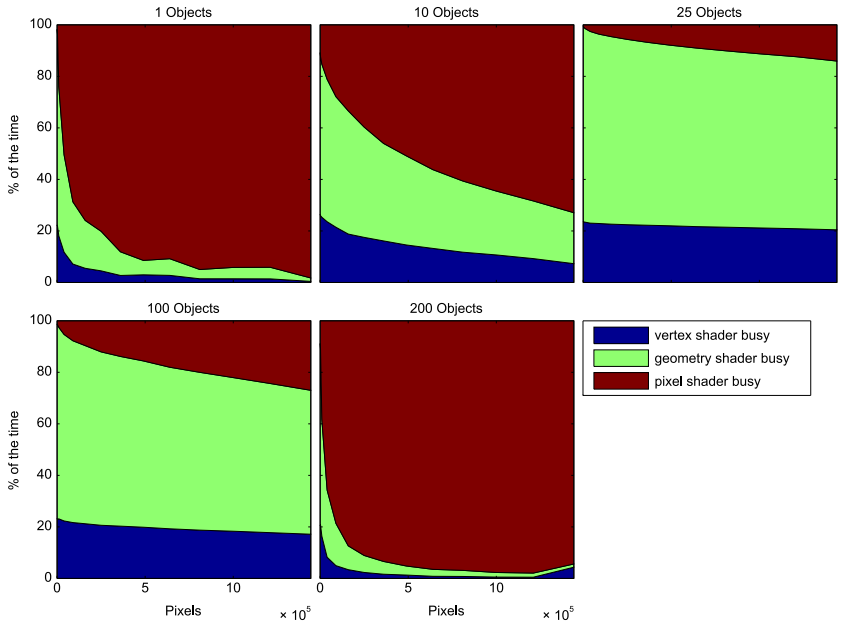


4.13: Performance counters for the *Z-pass shadow volumes* technique.

The performance counter graph for these techniques looks different from the image based techniques. Figure 4.13 shows that for one object, the ROP is still waiting for the shader, but when more objects are added to the scene, only the shaders take up a significant percentage of the time. The performance graphs of the Z-fail shadow volumes technique were not presented, because they are roughly the same.

For the image based techniques, the pixel shader was responsible for the time spent in the shaders, but the shadow volumes volumes use the newly available geometry shader. The shadow volumes technique uses a very simple pixel shader that only returns a constant color. This means that most of the time should be spent in the geometry shader.

To see if this is true, the time spent in the different shader types was analyzed.



4.14: Time spent in the different shader types.

Figure 4.14 shows the time that was spent in each of the shader types. The shadow volumes technique exists of multiple passes. There is one shadow volumes pass for every light source, using only a one-instruction pixel shader. Next to these passes, the scene is also rendered using phong lighting. With one object, most of the time is spent in the pixel shader of this pass. With 10 objects, the influence of the geometry shader is visible. With 25 objects most of the time is spent in the geometry shader.

This is one of the problems using the geometry shader. Although it takes some processing away from the CPU, when it is used heavily, no GPU processing power is left for the pixel shaders.

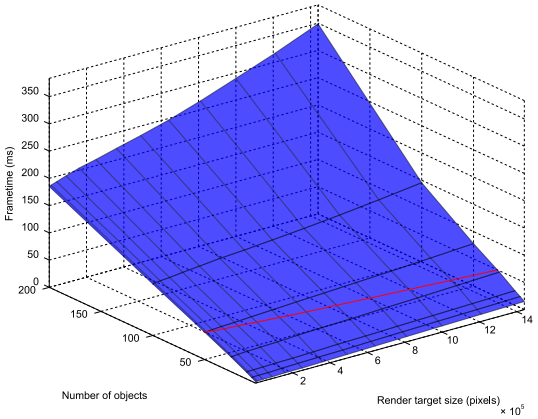
At 100 objects the time spent in the pixel shaders increases. This is because the used fill rate is getting so high, that the pixels can no longer be processed in parallel. The GPU has run out of pixel shader capacity. With 200 objects this effect is clearly visible. Most of the available GPU time goes to the pixel shaders. The fill rate is the bottleneck for this technique.

Soft Shadow volumes

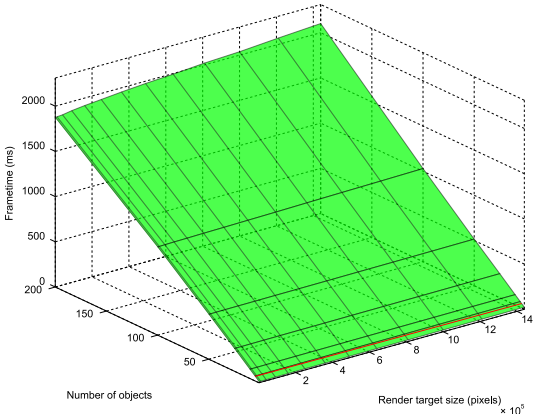
Soft shadow volumes is an extension to the shadow volumes technique. It uses penumbra wedges to create soft shadows. These penumbra wedges are generated in the geometry shader. Where shadow volumes create 2 polygons per silhouette edge, soft shadow volumes create 6. This means that the technique has even more problems with fill rate.

Figure 4.15a shows that the technique produces higher frame times than the Z-pass and Z-fail shadow volumes techniques. For low poly objects, the technique can achieve real-time performance with 50 objects at a render target resolution of 25x25 and with 25 objects at a render target resolution of 1200x1200.

Figure 4.15b shows the results for the high poly objects. Note that this graph has a different scale. In this situation, the number of objects the technique can render while still producing real-time results is aproximately 4.



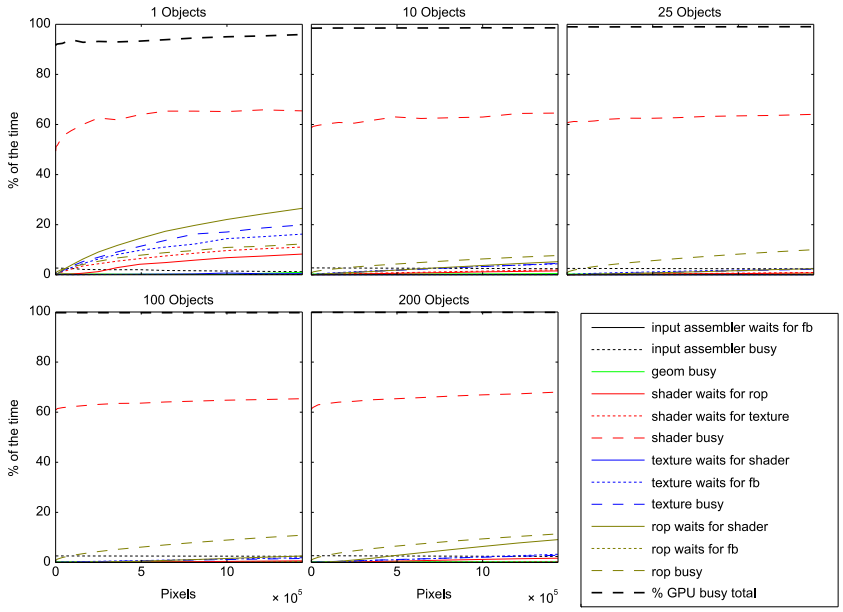
(a) 3 Light sources, low poly.



(b) 3 Light sources, high poly.

4.15: Frame times for the *Soft shadow volumes* technique.

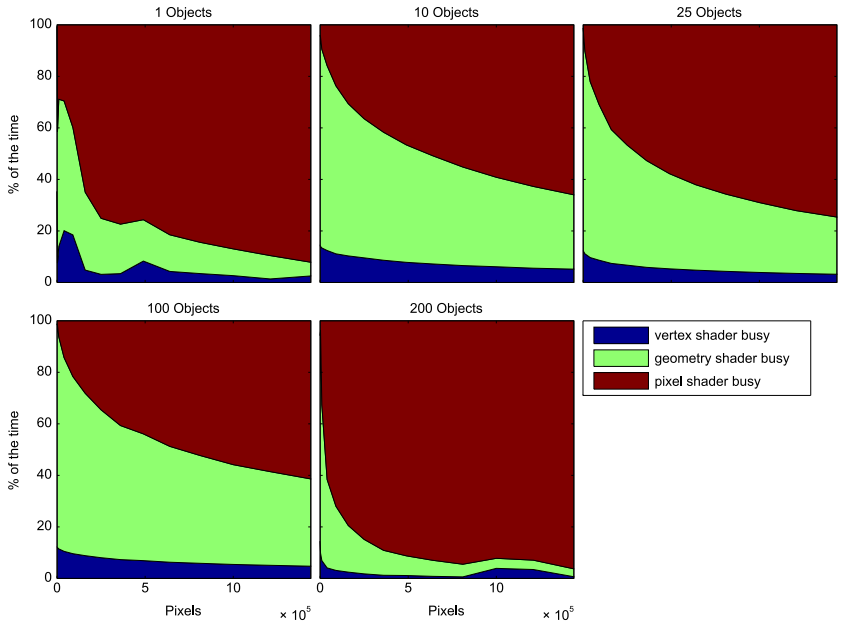
The soft shadow volumes technique takes a lot more time than the other shadow volumes techniques, because it not only renders the shadow volumes, it also uses a pixel shader to calculate the amount of light that reaches every pixel. This calculation takes up a lot of pixel shader instructions.



4.16: Performance counters for the *Soft shadow volumes* technique.

The performance graphs of the soft shadow volumes technique, shown in figure 4.16 resembles the performance graph for shadow volumes. This means that this technique also suffers from the lack of available fill rate.

As mentioned before, the soft shadow volumes technique uses a bigger pixel shader than Z-pass shadow volumes. To see if this has an impact on the amount of time spent in the different shaders, these were plotted in the following figure.



4.17: Time spent in the different shader types.

Figure 4.17 shows that the geometry shader uses less relative time than with the shadow volumes technique. This is purely because the pixel shader needs this time to execute.

The same effect as with shadow volumes is visible here: with 200 objects, the GPU uses too much fill rate, and parallel processing of all the pixels is no longer possible. For this technique the fill rate is also the bottleneck.

4.2.2 Shadow realism

The results of the shadow realism tests will be described in the following paragraphs.

Image Quality

Table 4.2 shows the differences between the images rendered using the shadow techniques and the reference image.

Scene 4 suffers from resolution problems when using the image based techniques. This causes an increase in the RMS score of the image based techniques. For cases where the light source is close to the scene (scenes 1, 2, 3 and 5), the percentage-closer soft shadows technique scores good. The shadow volumes and soft shadow volumes techniques score good for all the images.

The **Avg** column shows the average scores of the techniques. Using this column, the technique with the highest shadow quality is shadow volumes, followed by the soft shadow volumes technique. All image-based techniques have higher scores because of resolution problems in some scenes. Variance shadow mapping is the technique that produces the lowest shadow quality using the RMS as measurement.

According to the results of this test, images rendered with the shadow volumes and soft shadow volumes techniques are the closest to the reference image. These techniques should be used when conformance to the lighting model is important. Percentage-closer soft shadows can also be used when enough light map resolution is available througout the scene.

	Scene						
Technique	**1**	**2**	**3**	**4**	**5**	**6**	**Avg**
Standard shadow mapping	7.739	7.790	4.371	13.728	4.282	8.342	7.709
Percentage closer filtering (3x3)	7.370	6.327	3.610	14.711	3.933	8.145	7.349
PCF (bilinear interpolated)	7.517	7.078	3.658	12.169	4.118	8.246	7.131
PCF (bilinear interpolated) (3x3)	7.303	6.132	3.477	14.507	3.884	8.125	7.238
Percentage closer soft shadows	5.400*	6.982	3.797	21.356	1.862*	8.195	7.932
Variance shadow mapping	7.370	8.433	4.667	17.481	4.007	7.644	8.267
Shadow Volumes	6.954	6.192	3.860	1.587*	4.807	2.662*	4.344
Soft Shadow Volumes	5.698	5.557*	1.888*	4.276	4.395	4.394	4.368

*Closest to reference image

Table 4.2: Root mean square distances of the Y component in YUV space.

4.2.3 Web survey

To measure how people perceive the realism of the generated shadows, a web survey has been created. People had to rank the images generated by the shadow techniques from 1 (most realistic) to 9 (least realistic). They also had to fill in how hard it was to make the ranking; in other words: how much difference between the pictures is perceived. These values are used as weights to obtain the average rankings.

To see if the shadow techniques were ranked significantly different, the techniques were ordered by average ranking. A paired samples T Test was used to find out if the average rankings differed significantly. To keep the significance level at 0.05 a Bonferroni correction was applied to the significance tests.

A total number of 61 people entered in the web survey. The results of the web survey will be described in this section.

For every scene a table is given with the techniques in order of their average ranks. The techniques were divided into subsets of technique that do not differ significantly in ranking. In the tables, the following abreviations were used for the techniques:

- **SSM**, the standard shadow mapping technique.
- **PCF3x3**, percentage closer filtering filtering with a 3x3 filter kernel.
- **PCFb**, bilinear percentage closer filtering filtering.
- **PCFb3x3**, bilinear percentage closer filtering filtering with a 3x3 filter kernel.
- **PCSS**, percentage-closer soft shadows.
- **VSM**, variance shadow mapping.
- **SV**, shadow volumes.
- **SSV**, soft shadow volumes.
- **REF**, the reference image.

Table 4.3 shows the average rankings for the Car scene. No significant difference was found between the soft shadow volumes, variance shadow mapping, bilinear 3x3 percentage closer filtering, percentage-closer soft shadows and 3x3 percentage closer filtering.

Technique	subset for α=0.05		
	1	2	3
SSV	4.09		
VSM	4.16		
PCFb3x3	4.53	4.53	
PCSS	4.74	4.74	
PCF3x3	4.74	4.74	
PCFb		5.02	5.02
REF		5.30	5.30
SV			5.62
SSM			5.79

Table 4.3: Average rankings for the Car scene (1)

The reference image does not score very well; this indicates that conforming to the lighting model does not always make the image look realistic. A possible reason for this is that with static images it can be hard to see where the light source is located. When using the techniques in a dynamic environment the viewer can see the scene from different angles so the position of the light becomes clear.

Shadow volumes and standard shadow mapping scores the lowest. This is probably because they produce shadows with hard borders while the other images all have soft shadows borders.

Table 4.4 shows the average rankings for the Boxes scene. The four images of the techniques that score best are the reference image, variance shadow mapping and 3x3 bilinear percentage closer filtering. Next is the soft shadow volumes technique.

Technique	subset for α=0.05			
	1	2	3	4
REF	3.49			
VSM	4.05	4.05		
PCFb3x3	4.11	4.11		
SSV		4.75	4.75	
PCF3x3			4.91	
PCSS			4.96	
PCFb			5.34	
SV				6.41
SSM				6.72

Table 4.4: Average rankings for the Boxes scene (2)

The 3x3 percentage closer filtering, percentage-closer soft shadows and bilinear filtered percentage closer filtering techniques are in the third subset. This is probably because the image required such a high bias value that caused the shadows to be detached from the shadow casting objects.

The shadow volumes and standard shadow mapping technique score the worst. This can also be caused by the hard shadow borders they produce; these techniques do not support penumbras.

Table 4.5 shows the average rankings for the Disco Box scene. For this scene soft shadow volumes, variance shadow mapping and the reference image have the best rankings. They are perceived significantly more realistic than the percentage closer filtering based techniques and the shadow volumes.

	subset for α=0.05				
Technique	1	2	3	4	5
SSV	3.35				
REF	3.43				
VSM	3.74				
PCFb3x3		4.79			
PCSS		5.05			
PCF3x3			6.21		
SV			6.27	6.27	
PCFb				6.65	
SSM					7.35

Table 4.5: Average rankings for the Disco Box scene (3)

For the Disco box scene standard shadow mapping and binilear filtered percentage closer filtering is are ranked as the least realistic images. This is probably because with these techniques resolution artifacts are visible at the shadow borders.

The results of the Island scene are shown in table 4.6. The variance shadow mapping and bilinear 3x3 percentage closer filtering are the techniques that are valued as most realistic. These techniques both produce heavy filtered results, making the shadows very blurry. The soft shadow volumes, shadow volumes and reference images all have shadows with very hard shadow borders. The reason for this is the light source, which is situated at great distance from the scene geometry.

	subset for α=0.05				
Technique	1	2	3	4	5
VSM	3.35				
PCFb3x3	3.44				
SSV		4.89			
PCSS		5.18	5.18		
REF		5.26	5.26		
SV			5.83	5.83	
PCF3x3				6.08	
PCFb				6.08	
SSM					7.18

Table 4.6: Average rankings for the Island scene (4)

The choice for the blurry shadows could also come from the fact that the shadows are cast onto a surface with a water texture. Shadows that are cast on water are harder to see.

Since the distance of the light source to the scene is quite big, the image based techniques all suffer from resolution problems. The 3x3 percentage closer filtering, bilinear percentage closer filtering and standard shadow mapping do not use enough filtering to hide these resolutions. This is probably why they are valued as the least realistic images.

	subset for α=0.05		
Technique	1	2	3
SSV	4.22		
PCSS	4.23		
REF	4.28		
PCFb3x3	4.86	4.86	
VSM	5.00	5.00	
PCF3x3	5.11	5.11	
PCFb		5.41	
SSM			6.51
SV			6.84

Table 4.7: Average rankings for the Warehouse inside scene (5)

Table 4.7 shows the rankings for the Warehouse inside scene. For this scene, the stan-

dard shadow mapping and shadow volumes techniques do not perform well. For this scene soft shadow volumes probably look more realistic because the techniques with the highest rankings create the shadows with the softest borders.

The average rankings for the Warehouse outside scene are shown in table 4.8. For this scene the reference image, soft shadow volumes and percentage-closer soft shadows techniques score best. These are the techniques that produce the softest shadow borders.

	subset for α=0.05		
Technique	1	2	3
REF	3.95		
SSV	4.39	4.39	
PCSS	4.87	4.87	
PCFb3x3		4.96	4.96
VSM		5.19	5.19
PCFb		5.25	5.25
SV			5.25
SSM			5.36
PCF3x3			5.53

Table 4.8: Average rankings for the Warehouse outside scene (6)

The techniques that do not score well in the ranking are standard shadow mapping, shadow volumes and the percentage closer filtering based techniques with static filter size. In this scene light source and the camera are located close to each other so light map resolution is not a problem. This makes the percentage closer filtering based techniques produce relatively hard shadows.

Soft shadow borders are perceived as the more realistic for this scene too, so these techniques are not ranked as the most realistic.

	subset for α=0.05						
Technique	1	2	3	4	5	6	7
VSM	4.11						
SSV	4.34	4.34					
REF	4.37	4.37					
PCFb3x3		4.40					
PCSS			4.85				
PCF3x3				5.47			
PCFb					5.73		
SV						6.12	
SSM							6.60

Table 4.9: Average rankings for all scenes

Table 4.9 shows the combined rankings for all scenes. According to the tests, for all scenes combined the variance shadow mapping, soft shadow volumes and reference techniques produce the most realistic images. Because this table uses all results of all surveys, the significance of the differences between the ratios is higher.

The variance shadow mapping technique probably scores well because the shadows it produces in low resolution situations is not very dark. Because of the high level of filtering, small objects will create lighter shadows in cases of low light map resolutions.

Using the results of table 4.9 and the results of individual scenes it can be stated that soft shadow borders produce a higher level of perceived realism, even in the cases where the shadow borders should be hard according to the lighting model. Techniques that produce soft shadow borders like variance shadow mapping, percentage-closer soft shadows and soft shadow volumes should be used to make the scenes look more real.

4.2.4 Memory usage

This section describes the memory usage of the different techniques. The memory usage depends on the texture sizes that were used for light maps. For every tested technique that uses light maps, textures of 512x512 pixels were used. For techniques that required rendering to a texture that was later combined with the on screen image, textures were used that had the same size as the render target on screen.

In the following paragraphs the techniques are ordered from lowest to highest memory usage.

The *Z-pass* and *Z-fail shadow volumes* techniques do not use light maps. They do however need adjacency information for the geometry shaders to function properly. This means that twice as many indices for the index buffers are necessary. These indices are 2 bytes in size. When a scene uses more than 131072 indices this technique uses more memory than the shadow mapping based techniques with one light.

The *standard shadow mapping*, *variance shadow mapping*, *percentage closer filtering* and *percentage-closer soft shadows* techniques use a light map for every light source in the scene. As mentioned before, the resolution of this light map is 512x512 pixels. Every pixel stores a floating point number of 4 bytes so the total memory consumed by this technique is 1048576 bytes per light source.

The *Soft shadow volumes* technique uses the extra index information for the geometry shader. It also uses two extra buffers that have the same size as the render target. The first buffer is a floating point texture that is used to render the penumbra wedges to. The second buffer is a color texture that will contain the lit scene. Both textures use 4 bytes per pixel. The total extra memory required for this technique is the render target size in pixels times 8 bytes. For the largest render target size (1200x1200) this technique consumes the most memory, unless the number of light sources in the scene is higher than 10.

Chapter 5

Conclusion

In the following paragraphs the results of the tests of chapter 4 will be briefly mentioned.

5.1 Technique performance

According to the performance test, the best performing technique is standard shadow mapping. Standard shadow mapping produces the lowest frame times and is capable of real-time performance in most cases. Only for 140 or more high poly objects at resolutions of 500x500 this technique no longer produces real-time results. After shadow mapping, percentage closer filtering and variance shadow mapping are the techniques that perform best because they have the lowest frame times.

For percentage closer filtering the filter kernel size is important. Bigger filter kernels take longer to render. This increases frame times and reduces the amount of objects that can be rendered in real-time.

Techniques that only produce real-time results with low object counts or low poly objects are percentage-closer soft shadows and shadow volumes. This is because The time needed for the geometry shaders, and the GPU cannot provide the fill rate that is needed to produce real-time results.

Soft shadow volumes gives the worst performance; with high poly objects only a few objects can be rendered while keeping real-time performance.

5.2 Image quality

The techniques that provide the best image quality according to the used light model are shadow volumes and soft shadow volumes.

When light map resolution is not an issue percentage-closer soft shadows creates good results. Percentage closer filtering performs not as good as percentage-closer soft shadows but it still creates better results than standard shadow mapping.

Variance shadow mapping shadows do not conform to the used light model at all. They produce the worst image quality in this test.

5.3 Web survey

According to the people that participated in the web survey Variance shadow mapping, soft shadow volumes and percentage-closer soft shadows produce the most realistic shadows. Therefore the conclusion can be made that shadows with soft borders are perceived as more real.

Shadow techniques with hard shadow borders, like standard shadow mapping and shadow volumes, are perceived as the techniques that produce the least realistic results.

5.4 Memory usage

The technique that uses the least memory is shadow volumes, followed by the techniques that use light maps. These include shadow mapping, percentage closer filtering, percentage-closer soft shadows and variance shadow mapping.

When using up to 10 light sources techniques that use shadow maps use less memory than soft shadow volumes. In this case the use of techniques that use light maps can be recommended.

In situations where very low amounts of memory is available, shadow volumes is the best choice for creating shadows. In all other cases the image based techniques can be used.

5.5 Shadows in simulations

Shadows used in simulations should be able to perform in real-time and look as realistic as possible. Also, the memory usage should be kept as low as possible.

To select the best technique for realistic shadows in this research, user perception of realism was chosen in favor of conformance to the light model.

The techniques with the best real-time performance are standard shadow mapping, percentage closer filtering and variance shadow mapping. The techniques that are perceived as the most real looking are percentage-closer soft shadows, soft shadow volumes and variance shadow mapping.

These techniques all use the same amount of memory, except for soft shadow volumes, which uses more.

The goal of this research was to answer the following question:

> *Which existing shadow techniques, when adapted for using the capabilities of modern hardware, produce the best results in the areas of performance and shadow quality and how can these techniques be implemented using the Renderer2 API?*

By using the results of the tests the first part of this question, *which existing shadow techniques, when adapted for using the capabilities of modern hardware, produce the best results in the areas of performance and shadow quality* can be answered as follows:

In cases where the amount of available memory is very low, shadow volumes is the most suitable technique. The performance of this technique is at best when the number of objects is below 100 and the objects do not have many polygons (around 100).

In cases where enough memory is available for light maps, variance shadow mapping is the most suitable technique; it creates realistic shadows. The technique can create shadows in real-time for up to 90 high poly objects using render targets of 1200x1200. More objects are supported when they consist out of less polygons.

When conformance to the light model is important soft shadow volumes or percentage-closer soft shadows need to be used. Soft shadow volumes is the best choice for larger render targets because percentage-closer soft shadows does not support a large render target size in real-time. The number of objects that can be used with this technique should be below 25 and their polygon count should be around 100. For more or larger objects this technique does not produce real-time results.

The second part of the research question *how can these techniques be implemented using the Renderer2 API?* is answered in chapter 3:

A Renderer2 driver has been created for Direct3D 10 and a shader library was implemented that countains the shaders for the different shadow techniques.

Chapter 6

Discussion

6.1 Implementation

During this research a lot of time was spent implementing the Renderer2 driver, the application and the shadow techniques. This left little time to compare the shadow techniques to each other.

Another possible approach was to implement less shadow techniques. This would have left more time for the testing. The choice was made to implement all techniques that are used in real-time situations at this time. This gives a more complete representation of the shadow techniques to choose from.

6.2 Measurement results

To measure the performance of the techniques the Nvidia Performance kit SDK was used [NVP07]. This SDK provides access to the GPU counters using an instrumented driver and a programming interface. The instrumented driver decreases performance to a maximum of 6%. Since this driver was used for every technique the performance may be compared to each other, but the results may be lower than they could be when using the performance drivers that are normally used.

The SDK comes with a user manual that explains the available performance. Unfortunately, this user manual is outdated. The GPU that was used for the experiments provided a lot more counters than the user manual stated. Also, some of the counters were named differently. A different way of interfacing with the GPU counters could have helped in speeding up the testing process.

The results that are shown in the graphs of section 4.2.1 were obtained by sampling the counters 2 times for each test scene. To get the values

of these counters, frames had to be rendered up to 179 times because the instrumented driver does not refresh the counters every frame.

The counters were only sampled 2 times, because the total rendering time of all techniques in all situations took a long time. This is why the graphs in sections 4.2.1 show some noise. If there would have been more time, the test could have been repeated several times and the results could have been averaged or filtered in order to decrease the noise.

6.3 Web survey

61 People participated in the web survey. This number of people was enough to obtain signficant results between the shadow technique rankings. However, the siginficance level could be raised if more people participated in the survey.

One problem of the survey was that people had to rank the techniques from 1 (most realistic) to 9 (least realistic). This was explained in the text of the survey on every page.

A better choice would have been to rank the techniques from 9 (most realistic) to 1 (least realistic). This is more intuitive: people tend to associate a 9 with a good situation and a 1 with a bad one. 6 Of the people that filled in the survey indicated that they filled in the survey the wrong way. They used 9 for the most realistic scene and 1 for the least realistic one. The results of these people have been inverted to prevent systematic measurement errors.

However, not all people were contacted after filling in the survey, so more people could have made this mistake.

A paired T-test with bonferroni correction was used to analyze the rankings of the different techniques. This is a rather generic test, so the rankings were compared rather conservatively. There are other tests to search for significant differences between the techniques, but they do not suit the situation of the web survey.

The first approach to analyze the results by using a one-way ANOVA test. This test requires that every subject only ranks 1 situation. In the case of the web survey everyone had to rank 9 situations, so this test cannot be used.

Another approach was using a repeated measures ANOVA. This test works the same as the ANOVA but for multiple situations. An important condition of the repeated measures ANOVA is that the different rankings appear in the same order. For the web survey this was not the case. People need to look at all 9 images before ranking them.

6.4 Techniques

The techniques used to evaluate shadow technique performance in chapter 4 were compared to each other for performance. Only one implementation for each technique was used to do the testing; different scene configurations were used, the techniques were used in their original form.

The performance of the individual techniques can also be changed by changing properties of these techniques. For example, for the techniques that use a light map, the light map size can be changed. For variance shadow mapping the amount of light map filtering can be decreased to increase performance. For percentage-closer soft shadows, the area in which the blocker search is performed can be decreased to obtain an increase in speed in exchange of shadow quality.

For the test scenes, every object in the scene was rendered every frame and for every light map. Sending only the objects that are visible to the GPU could cause a hugh performance increase. In most 3D simulations, not every object can cast dynamic shadows. Only calculating the shadows for the dynamic objects can also cause a performance gain.

Appendix A

Web survey

The following pages contain the web survey that was created to get an idea of what people perceive as realistic shadows. This survey was filled in by 63 people. Some of these people had experience with 3D graphics, but most of them did not.

The survey checks if the participant has filled in all questions before the participant can navigate to the next level. The shadow techniques were presented in a random order.

Vergelijking schaduwtechnieken (1/6)

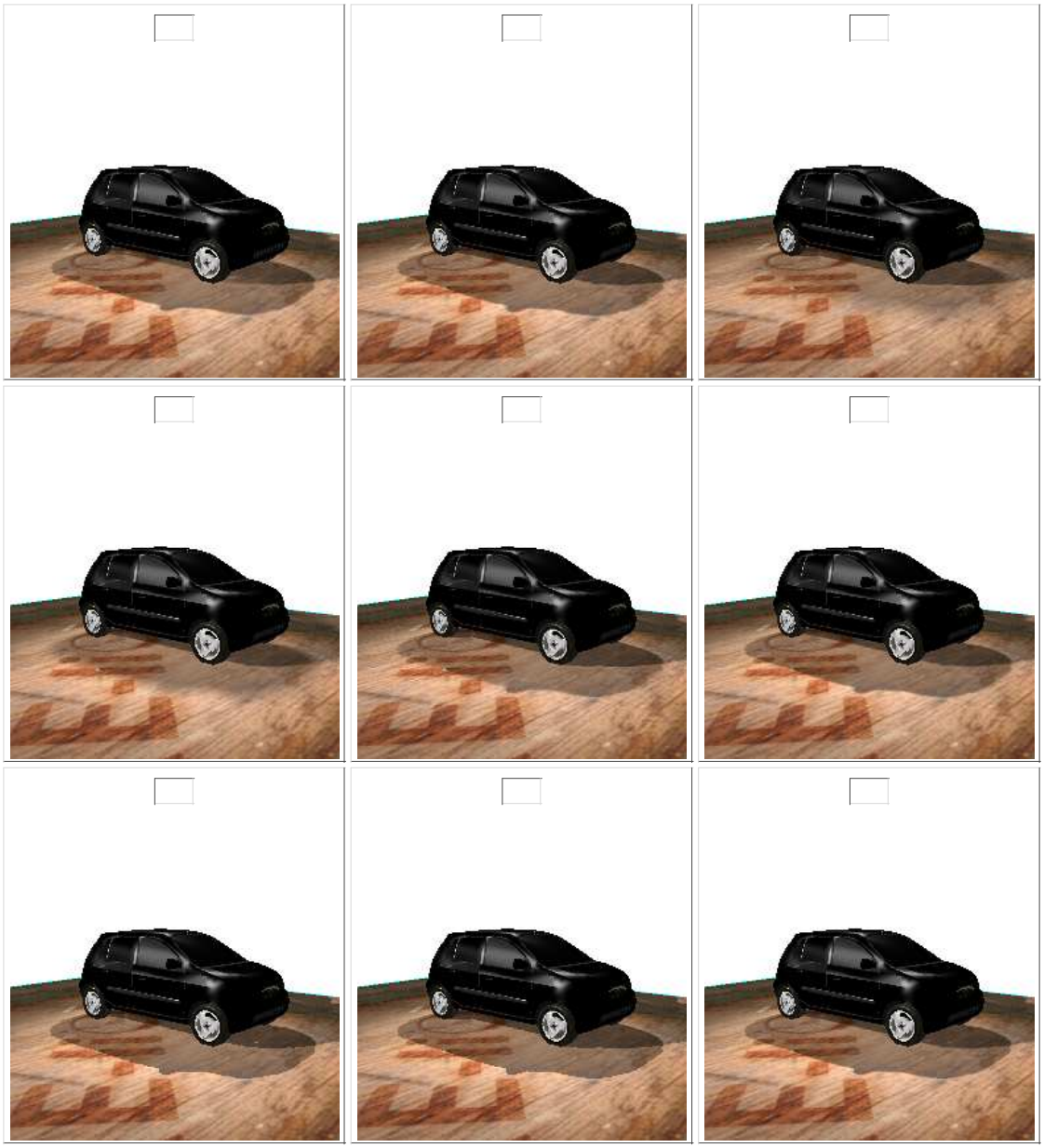
Hieronder staan 9 afbeeldingen die gebruik maken van verschillende schaduwtechnieken. Plaats in het hokje boven iedere afbeelding een 1 (meest realistisch) tot een 9 (minst realistisch). Als je geen verschil ziet tussen twee afbeeldingen mag je ze hetzelfde cijfer geven.

Druk na het invullen op Verder om verder te gaan.

Je kunt een plaatje groter maken door er op te klikken.

Wil je hieronder nog even invullen hoe moeilijk het is verschil tussen de technieken te zien?

Moeilijk verschil te zien. ○ ○ ○ ○ ○ Makkelijk verschil te zien.



Verder

Vergelijking schaduwtechnieken (2/6)

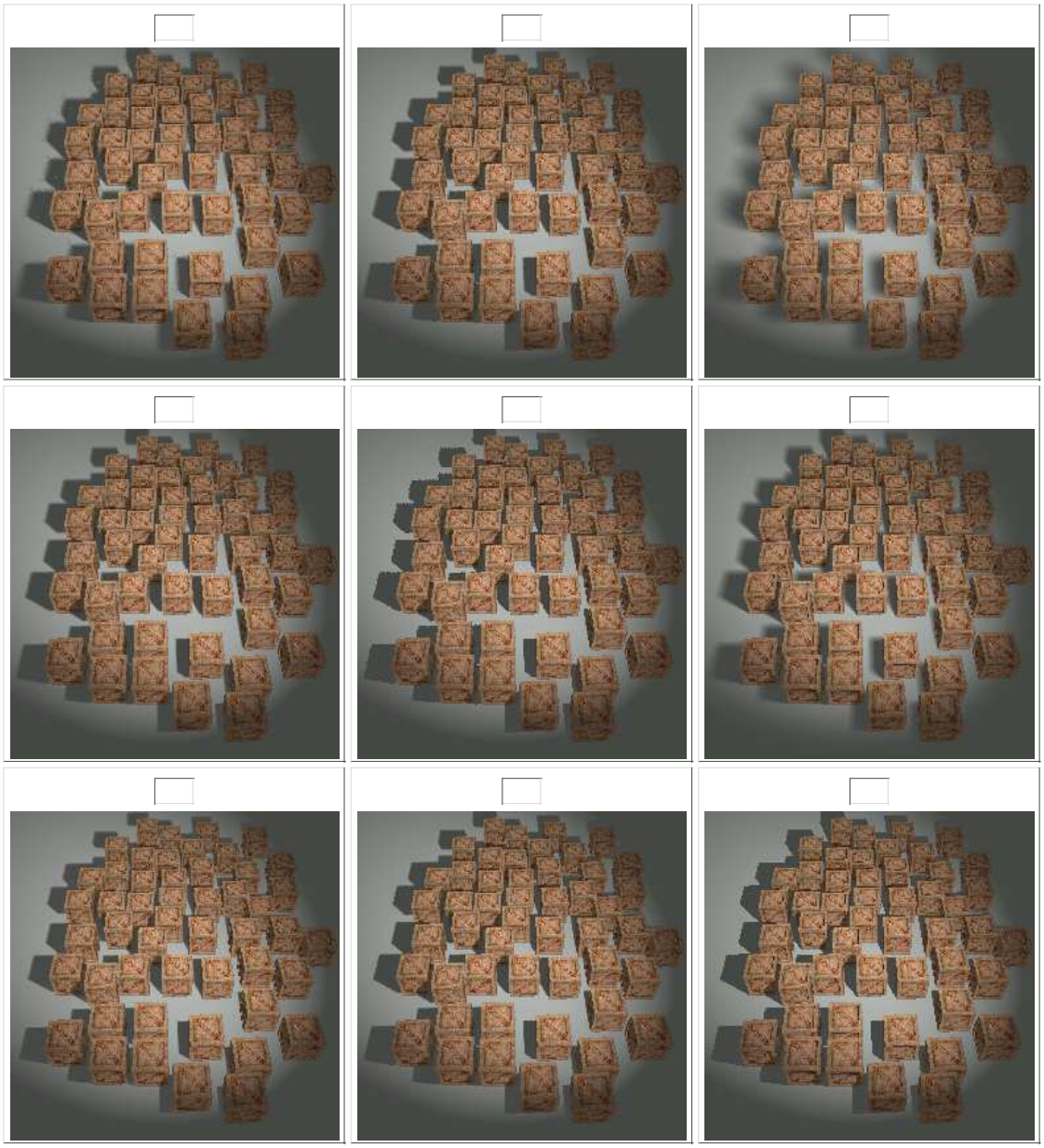
Hieronder staan 9 afbeeldingen die gebruik maken van verschillende schaduwtechnieken. Plaats in het hokje boven iedere afbeelding een 1 (meest realistisch) tot een 9 (minst realistisch). Als je geen verschil ziet tussen twee afbeeldingen mag je ze hetzelfde cijfer geven.

Druk na het invullen op Verder om verder te gaan.

Je kunt een plaatje groter maken door er op te klikken.

Wil je hieronder nog even invullen hoe moeilijk het is verschil tussen de technieken te zien?

Moeilijk verschil te zien. ○ ○ ○ ○ ○ Makkelijk verschil te zien.



Verder

Vergelijking schaduwtechnieken (3/6)

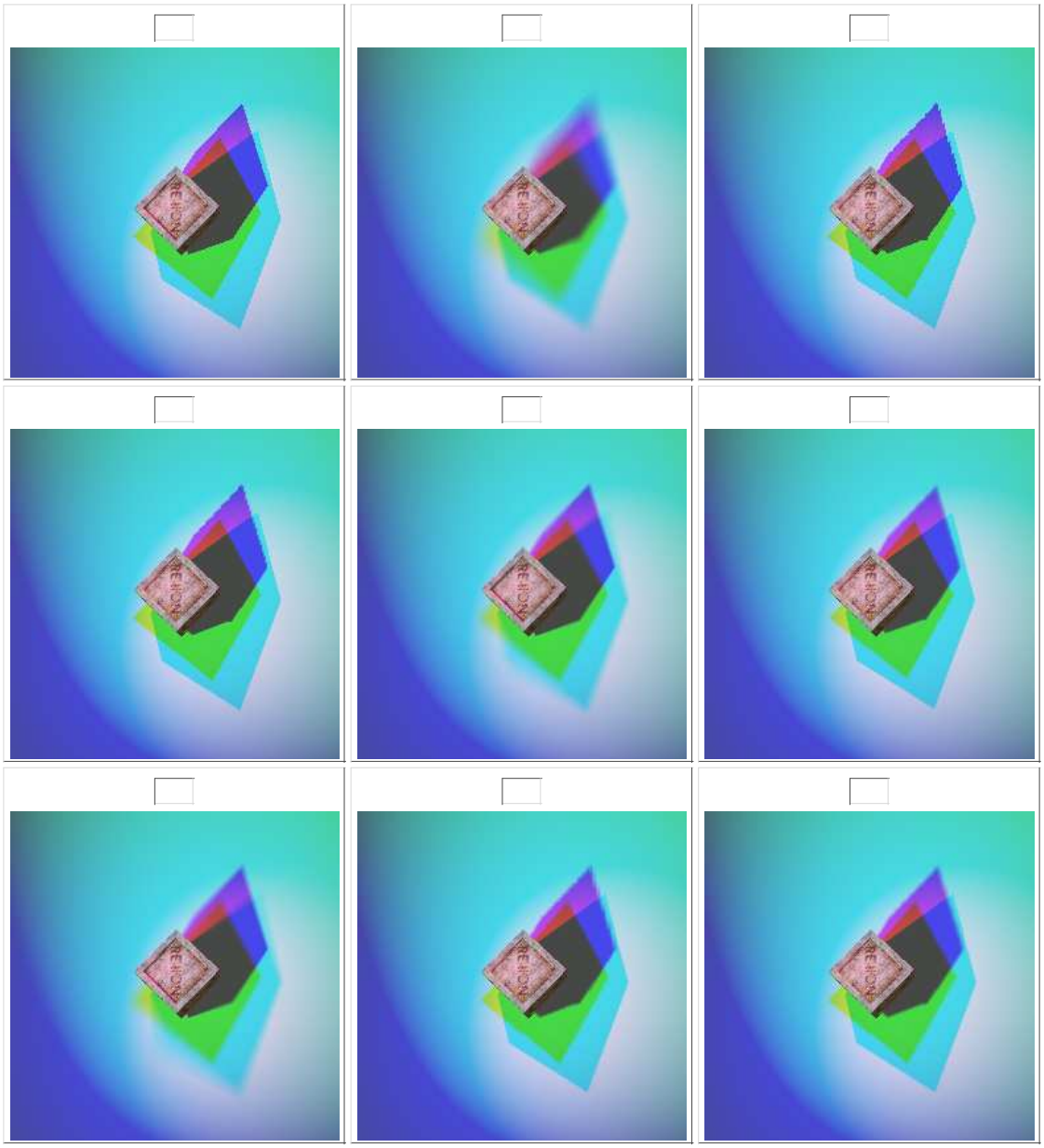
Hieronder staan 9 afbeeldingen die gebruik maken van verschillende schaduwtechnieken. Plaats in het hokje boven iedere afbeelding een 1 (meest realistisch) tot een 9 (minst realistisch). Als je geen verschil ziet tussen twee afbeeldingen mag je ze hetzelfde cijfer geven.

Druk na het invullen op Verder om verder te gaan.

Je kunt een plaatje groter maken door er op te klikken.

Wil je hieronder nog even invullen hoe moeilijk het is verschil tussen de technieken te zien?

Moeilijk verschil te zien. ○ ○ ○ ○ ○ Makkelijk verschil te zien.



Verder

Vergelijking schaduwtechnieken (4/6)

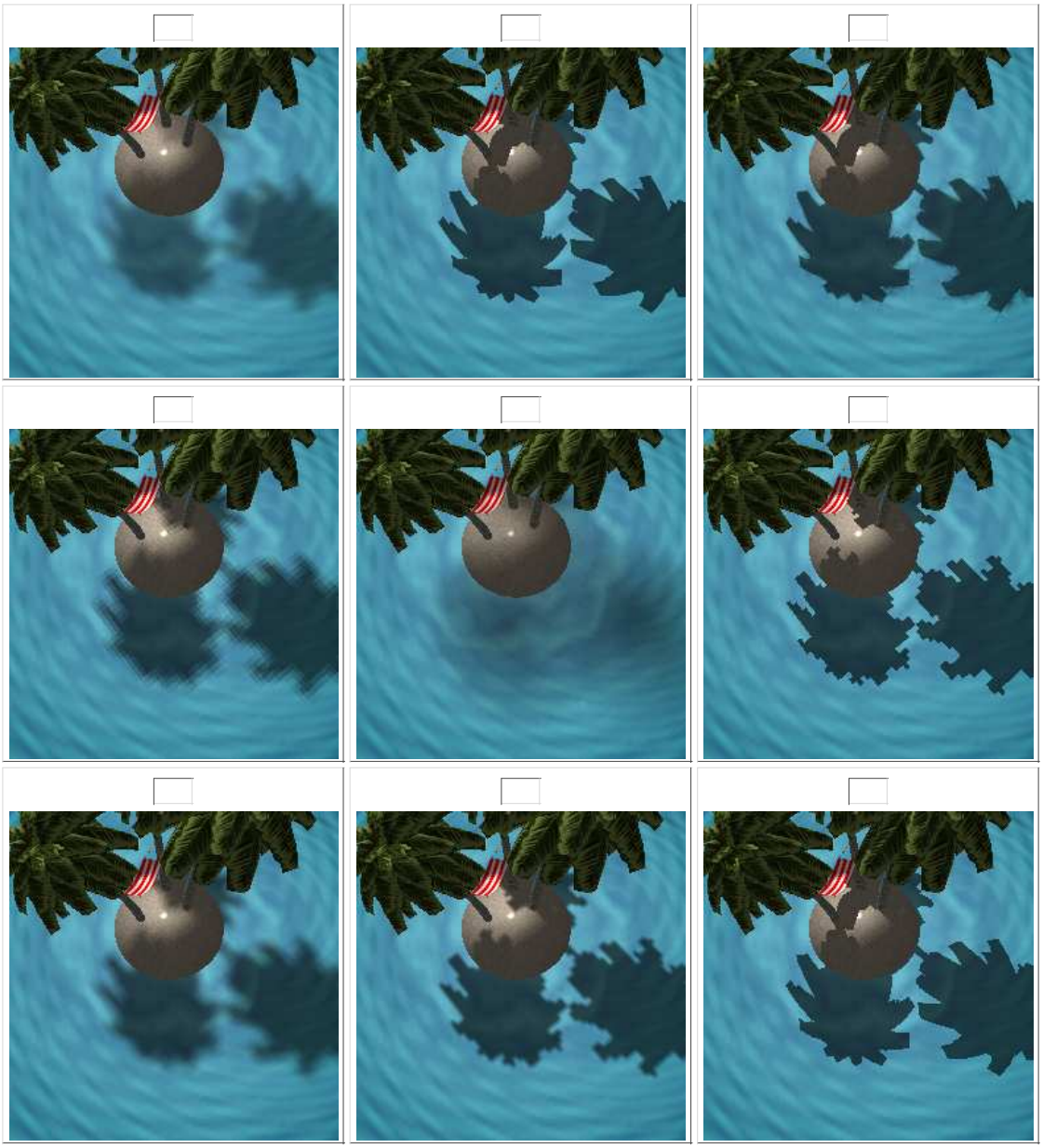
Hieronder staan 9 afbeeldingen die gebruik maken van verschillende schaduwtechnieken. Plaats in het hokje boven iedere afbeelding een 1 (meest realistisch) tot een 9 (minst realistisch). Als je geen verschil ziet tussen twee afbeeldingen mag je ze hetzelfde cijfer geven.

Druk na het invullen op Verder om verder te gaan.

Je kunt een plaatje groter maken door er op te klikken.

Wil je hieronder nog even invullen hoe moeilijk het is verschil tussen de technieken te zien?

Moeilijk verschil te zien. ○ ○ ○ ○ ○ Makkelijk verschil te zien.



Verder

Vergelijking schaduwtechnieken (5/6)

Hieronder staan 9 afbeeldingen die gebruik maken van verschillende schaduwtechnieken. Plaats in het hokje boven iedere afbeelding een 1 (meest realistisch) tot een 9 (minst realistisch). Als je geen verschil ziet tussen twee afbeeldingen mag je ze hetzelfde cijfer geven.

Druk na het invullen op Verder om verder te gaan.

Je kunt een plaatje groter maken door er op te klikken.

Wil je hieronder nog even invullen hoe moeilijk het is verschil tussen de technieken te zien?

Moeilijk verschil te zien. ○ ○ ○ ○ ○ Makkelijk verschil te zien.

Verder

Vergelijking schaduwtechnieken (6/6)

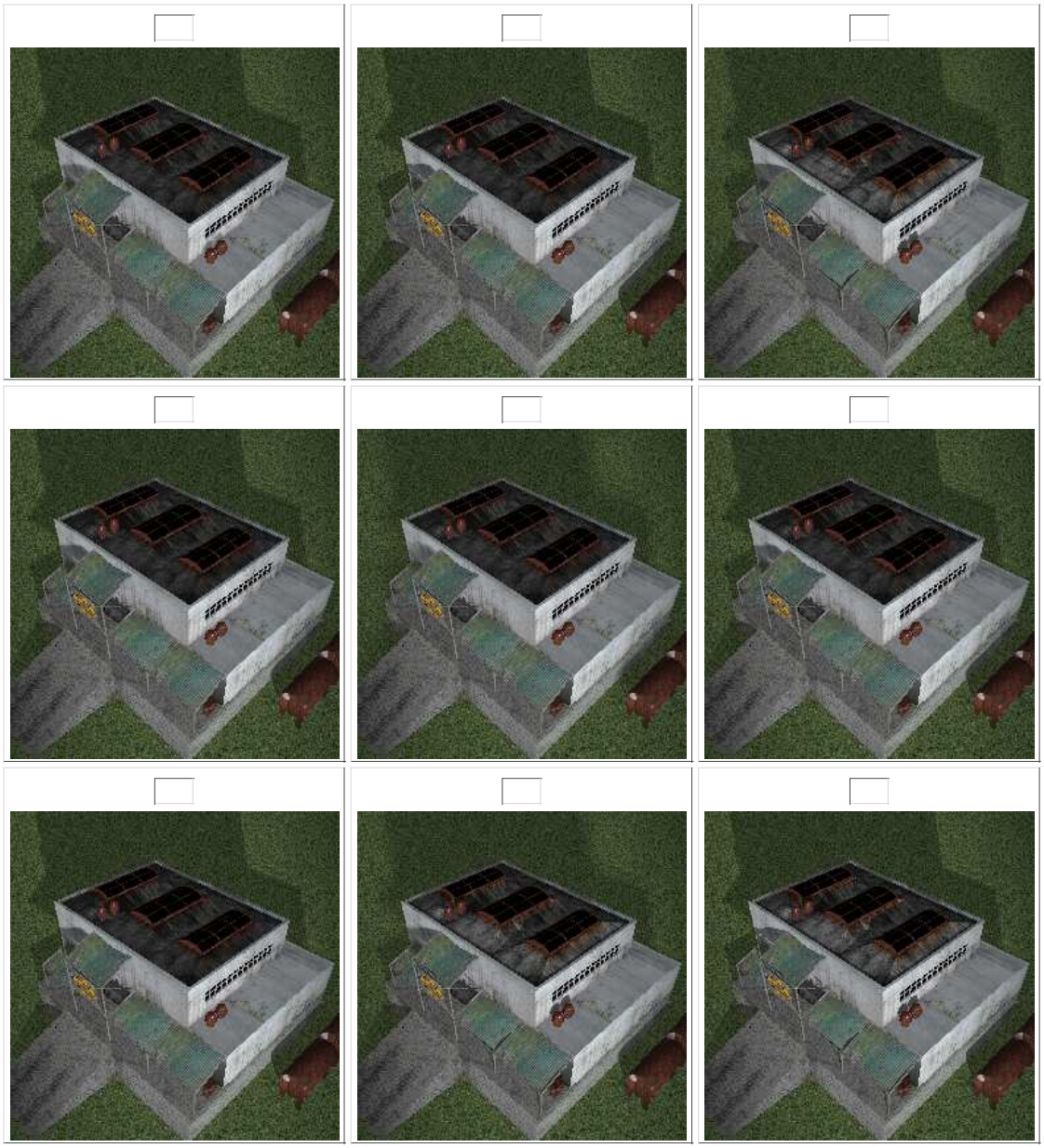
Hieronder staan 9 afbeeldingen die gebruik maken van verschillende schaduwtechnieken. Plaats in het hokje boven iedere afbeelding een 1 (meest realistisch) tot een 9 (minst realistisch). Als je geen verschil ziet tussen twee afbeeldingen mag je ze hetzelfde cijfer geven.

Druk na het invullen op Verder om verder te gaan.

Je kunt een plaatje groter maken door er op te klikken.

Wil je hieronder nog even invullen hoe moeilijk het is verschil tussen de technieken te zien?

Moeilijk verschil te zien. ○ ○ ○ ○ ○ Makkelijk verschil te zien.



Verder

Glossary

API	Application Programming Interface, a collection of function definitions that allows an application to communicate with another application, a library or an operating system., 12, 14, 35–37, 40–43, 46–49, 51, 66, 98
Direct3D 10	A 3D API for Microsoft Windows Vista. Direct3D 10 has dropped backwards compatability. Only hardware that completely supports every aspect of Direct3D 10, can use it., 14, 35–38, 40–51, 57–59, 63, 66, 98
Direct3D 9	A 3D API for Microsoft Windows XP. Direct3D 9 is the last version of Direct3D that is backwards compatible., 14, 36–38, 40–42, 44–51
Directional light	Light source with parallel rays, 18
DirectX	A collection of API's for handling tasks related to multimedia on Windows, 39, 46
Filter kernel	The area used to filter a point, 22
GPU	Graphical Processing Unit, a dedicated graphics rendering device., 2, 11–15, 37, 44, 49, 59, 68, 72–75, 77–80, 86, 89, 96, 99, 101
Multi-pass technique	The geometry in the scene will be rendered multiple times (in multiple passes), and the results of each pass will be used in one of the subsequent passes, 19
OpenGL	A platform independant 3D API specification., 14, 35–37, 41–44, 47

point light	Light source that only illuminates a scene in all directions, 17, 18, 32, 52
Self shadows	Shadow that is cast by an object onto itself, 26
Spot light	Light source that has a direction and a cone angle; only objects inside it's cone are illuminated, 18, 52, 61

Index

References

[AMA02] Tomas Akenine-Möller and Ulf Assarsson. Approximate soft shadows on arbitrary surfaces using penumbra wedges. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pp. 297–306. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2002. ISBN 1-58113-534-3.

[BAS02] Stefan Brabec, Thomas Annen and Hans-Peter Seidel. Practical Shadow Mapping. *journal of graphics tools*, 7(4):pp. 9–18, 2002.

[Bli88] J. Blinn. Me and My (Fake) Shadow. *IEEE Comput. Graph. Appl.*, 8(1):pp. 82–86, 1988. ISSN 0272-1716.

[Car00] John Carmack. unpublished corresondence, 2000.

[Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. *Computer Graphics (Proceedings of SIGGRAPH '77)*, pp. 242–248, 1977.

[Dir07] Microsoft DirectX 10 site. `http://www.gamesforwindows.com/en-US/AboutGFW/Pages/DirectX10.aspx`, 2007. Last checked on November 28, 2007.

[DL06] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pp. 161–165. ACM Press, New York, NY, USA, 2006. ISBN 1-59593-295-X.

[DXC07] Direct3D 9 to Direct3D 10 Considerations (Direct3D 10). `http://msdn2.microsoft.com/EN-US/library/bb205073.aspx`, 2007. Last checked on November 28, 2007.

[Fer05] Randima Fernando. Percentage-closer soft shadows. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, p. 35. ACM Press, New York, NY, USA, 2005.

[FFBG01] Randima Fernando, Sebastian Fernandez, Kavita Bala and Donald P. Greenberg. Adaptive shadow maps. In *SIGGRAPH '01:*

Proceedings of the 28th annual conference on Computer graphics and interactive techniques, pp. 387–390. ACM Press, New York, NY, USA, 2001. ISBN 1-58113-374-X.

[For07] Tom Forsyth. Extremely practical shadows, presentation at GDC 2006. `http://home.comcast.net/~tom_forsyth/papers/GDC2006_Forsyth_Tom_ExtremelyPracticalShadows.ppt.zip`, 2007. Last checked on November 28, 2007.

[Geo07] Shader Stages (Direct3D 10) – Geometry-Shader Stage. `http://msdn2.microsoft.com/en-us/library/bb205146.aspx#Geometry_Shader_Stage`, 2007. Last checked on November 28, 2007.

[GLE07] `GL_EXT_geometry_shader4` extension. `http://www.opengl.org/registry/specs/EXT/geometry_shader4.txt`, 2007. Last checked on November 28, 2007.

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

[HHLH05] Samuel Hornus, Jared Hoberock, Sylvain Lefebvre and John C. Hart. ZP+: correct Z-pass stencil shadows. In *ACM Symposium on Interactive 3D Graphics and Games*. ACM, ACM Press, April 2005.

[HLHS03] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch and François Sillion. A survey of Real-Time Soft Shadows Algorithms. *Computer Graphics Forum*, 22(4):pp. 753–774, dec 2003. State-of-the-Art Reviews.

[LSK+05] Aaron Lefohn, Shubhabrata Sengupta, Joe M. Kniss, Robert Strzodka and John D. Owens. Dynamic Adaptive Shadow Maps on Graphics Hardware. In *ACM SIGGRAPH 2005 Conference Abstracts and Applications*. August 2005.

[LWGM04] Brandon Lloyd, Jeremy Wendt, Naga Govindaraju and Dinesh Manocha. CC shadow volumes. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Sketches*, p. 146. ACM Press, New York, NY, USA, 2004. ISBN 1-59593-896-2.

[NVP07] NVPerfSDK. `http://developer.nvidia.com/object/nvperfsdk_home.html`, 2007. Last checked on November 28, 2007.

[Ope07] OpenGL - The Industry Standard for High Performance Graphics. `http://www.opengl.org`, 2007. Last checked on November 28, 2007.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):pp. 311–317, 1975. ISSN 0001-0782.

[RSC87] William T. Reeves, David H. Salesin and Robert L. Cook. Rendering Antialiased Shadows with Depth Maps. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):pp. 283–291, July 1987.

[SD02] Marc Stamminger and George Drettakis. Perspective shadow maps. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pp. 557–562. ACM Press, 2002. ISBN 1-58113-521-1.

[Wil78] Lance Williams. Casting Curved Shadows on Curved Surfaces. *Computer Graphics (SIGGRAPH '78 Proceedings)*, 12(3):pp. 270–274, August 1978.

[WPF90] Andrew Woo, Pierre Poulin and Alain Fournier. A Survey of Shadow Algorithms. *IEEE Comput. Graph. Appl.*, 10(6):pp. 13–32, 1990. ISSN 0272-1716.

[WSP04] Michael Wimmer, Daniel Scherzer and Werner Purgathofer. Light Space Perspective Shadow Maps. In *Proceedings of Eurographics Symposium on Rendering 2004*. 2004.