A comparison of Shadow Algorithms

Exam Project at IMM•DTU

Written by Joen Sindholt, s991368

Supervisors: Niels Jørgen Christensen
Bent Dalgaard Larsen

IMM•DTU

Written at:
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Lyngby, Denmark

May 30, 2005

A comparison of Shadow Algorithms

by

Joen Sindholt

Written in partial fulfillment of the requirements for the degree of

"MASTER OF SCIENCE IN ENGINEERING"

at the institute of

INFORMATICS AND MATHEMATICAL MODELLING
TECHNICAL UNIVERSITY OF DENMARK, LYNGBY, DENMARK
MAY 2005.

....

Author: Joen Sindholt, s991368

Supervisors:

Niels Jørgen Christensen
Bent Dalgaard Larsen

Technical University of Denmark, May 30, 2005
IMM-Thesis-2005-37

Abstract

Because shadows are very important in computer generated images, a lot of different algorithms for shadow generation have been proposed. However two algorithms namely Shadow Maps[WILL78] and Shadow Volumes[CROW77] are the mostly used for real-time purposes, and many of the current shadow algorithms are built upon these.

While many game-engines choose a single algorithm for shadow generation we look into the possibility of using multiple algorithms, choosing the best based on the current scene at hand.

The idea is, that the choice of algorithm might be done automatically at runtime or at least be specified as a parameter at design time.

To do the above, we implement and analyze four different shadow algorithms which generates hard shadows, and look into the strengths and weaknesses of the four. We will also look into how to choose the appropriate algorithm depending on scene properties.

An application is created allowing to choose among the different algorithms during runtime, enabling easy overview of the differences.

It is found, that none of the four algorithms are perfect with regard to all aspects of shadow generation, as they focus on different problems.

It is also found, that objectively measuring how good a specific algorithm is, seems to be an almost impossible task, and it is concluded that choosing algorithm at design time seems a better approach for choosing the right algorithm.

The four algorithm examined in this thesis are the Shadow Map algorithm introduced by L.Williams [WILL78], the Shadow Volumes algorithm introduced by Frank Crow [CROW77], Perspective Shadow Maps by Marc Stamminger and George Drettakis[SD02] and a hybrid algorithm using ideas of both Shadow Maps and Shadow Volumes by Eric Chan and Frédo Durand [CD04].

Resumé

Fordi skygger er meget vigtige i computer-genererede billeder, er mange forskellige algoritmer til skygge-beregninger foreslået. Trods det, er de to algoritmer Shadow Maps[WILL78] og Shadow Volumes[CROW77] de mest brugte i real-tids applikationer, og mange af de nuværende algoritmer bygger på disse to.

Mens mange game-engines vælger at bruge en enkelt algoritme til skyggeberegninger, kigger vi på muligheden for at bruge flere algoritmer, og vælge den bedste til den nuværende scene.

Ideen er, at valget af algoritme måske kan tages automatisk under kørsel eller i det mindste blive specificeret som en parameter når scenen designes.

For at gøre det ovenstende, implementerer og analyserer vi fire forskellige algoritmer som alle genererer hårde skygger, og vi ser på styrker og svagheder ved de fire. Vi ser også på, hvordan man kan vælge den bedste algoritme afhængigt af scenens indhold.

En applikation laves, i hvilken det er muligt at vælge imellem de forskellige algoritmer under kørsel, hvilket gør det muligt at overskue forskellene.

Det findes, at ingen af de fire algoritmer er perfekte med hensyn til alle områder af det at generere skygger, da de fokuserer på forskellige problemer.

Det findes også, at det at lave en objektiv måling af hvor god en specific algoritme er, er en næsten umulig opgave, og det konkluderes at det at vælge algoritmen under design fasen synes en bedre fremgangsmåde når den rette algoritme skal vælges.

De fire algoritmer der undersøges i denne rapport er: Shadow Map introduceret af L. Williams [WILL78], Shadow Volumes introduceret af Frank Crow [CROW77], Perspective Shadow Maps af Marc Stamminger og George Drattakis[SD02] og en hybrid algoritme, som bruger ideer fra både Shadow Maps og Shadow Volumes, af Eric Chan and Frédo Durand [CD04].

Preface

This thesis is written in partial fulfillment of the requirements for the degree of "Master of Science in Engineering" at the Institute of Informatics and Mathematical Modelling, Technical University of Denmark,

My background for doing the project is an highly increasing interest in computer graphics having attended the courses "02561 Computer Graphics" and "02565 Computer Animation and Modelling" at DTU.

Prior to the project I had limited knowledge of the OpenGL API and the use of CG programs.

I would like to thank my supervisors Niels Jørgen Christensen and Bent Dalgaard Larsen for always being willing to discuss the different aspects of the project.

I would also like to state that the project has given me a great chance to see what is involved in making a larger 3D application using OpenGL, and I have learned a great deal within the area during the work with project.

Contents

1 Introduction **1**

1.1 Hard and Soft Shadows 2
1.2 Real-Time Shadows 4
1.3 Related Work 5

2 Shadow Maps **7**

2.1 Theory 7
2.1.1 Transforming Depth Values 7
2.1.2 Aliasing 8
2.1.3 Biasing 9
2.1.4 Spot Lights 9
2.2 Implementation 10
2.2.1 Initialize Textures 11
2.2.2 Virtual Camera 11
2.2.3 Draw Scene Using Virtual Camera 12
2.2.4 Saving Depth Values 12
2.2.5 Rendering Using Shadow Map 13
2.2.6 Comparing Depth Values 13
2.2.7 Rendering Final Image 14
2.2.8 Biasing 14
2.2.9 Filtering 15
2.2.10 Rendering to Texture 17

3 Shadow Volumes **21**

3.1 Theory 21
3.2 Implementation 23
3.2.1 Adjacent List 24
3.2.2 Initializing Depth Values 24
3.2.3 Drawing Shadow Volumes 24
3.2.4 Rendering Using Stencil Buffer 26
3.2.5 ZFail vs ZPass 26
3.2.6 Two-sided stencil buffer 28
3.2.7 Comparison to Shadow Maps 28

4 Perspective Shadow Maps **31**

4.1 Theory 31
4.2 Implementation 34
4.2.1 Transforming The Scene 35
4.2.2 The Light 'Camera' 35
4.2.3 Calculating Texture Coordinates 36
4.2.4 Results so far 37
4.2.5 Adjusting Real Camera 38

4.2.6 Cube Clipping . . . 39

5 Chan's Hybrid Algorithm **43**
5.1 Theory . . . 43
5.2 Implementation . . . 44
5.2.1 Create Shadow Map . . . 44
5.2.2 Identifying Silhouette Pixels . . . 45
5.2.3 Creating Computation Mask . . . 46
5.2.4 Rendering Shadow Volumes . . . 47
5.2.5 Final Shading . . . 48

6 The Application **53**

7 Results **57**
7.1 Timing . . . 57
7.2 Quality and Robustness . . . 60
7.2.1 Shadow Maps . . . 60
7.2.2 Perspective Shadow Maps . . . 61
7.2.3 SV and Chan's . . . 62
7.3 Generality . . . 62
7.4 Overview . . . 63

8 Discussion **67**
8.1 Comparison . . . 67
8.2 Using Multiple Algorithms . . . 69

9 Conclusion **71**

A Images **72**
A.1 Test Scenes . . . 72
A.1.1 Shadow Maps . . . 72
A.1.2 Perspective Shadow Maps . . . 73
A.1.3 Shadow Volumes . . . 74
A.1.4 Chan's hybrid . . . 75
A.2 Shadow Map Quality . . . 76
A.3 Perspective Shadow Map Quality . . . 78
A.4 Shadow Volumes and Chan's Hybrid Quality . . . 80
A.5 Generality . . . 81

B Keyboard Shortcuts **82**
B.1 Basic Shortcuts . . . 82
B.2 Shadow Maps . . . 82
B.3 Perspective Shadow Maps . . . 82
B.4 Shadow Volumes . . . 83
B.5 Chan's . . . 83

C Code **84**

C.1 Shadow Maps (Using PBuffer) 84
C.2 Perspective Shadow Maps 92
C.3 Shadow Volumes (Z-Fail) 106
C.4 Chan's Hybrid 113
C.5 Shadow Maps (Using Copy To Texture) 132
C.6 Shadow Volumes (Z-Pass) 140
C.7 Other Classes 146
C.7.1 Main.cpp 146
C.7.2 RenderAlgorithm 154

List of Tables

1 Strengths and weaknesses for the four algorithms under the categories: *Speed*, *Quality*, *Robustness* and *Generality* 64
2 Subjective grades diagram using the four algorithms when rendering the five test-scenes. Under each category the algorithms have been graded from 1-5 based on the subjective opinions of the author . . 65

List of Figures

1 Spatial Relationship: In the first image it is hard to determine whether the ball is on the ground or floating in the air. This is easily determined in the two next images due to the shadows cast by the ball . . . 1
2 Blocker, Receiver, light source . . . 3
3 Hard Shadows: Points on the receiver can either see the light source or not, resulting in hard shadows with sharp edges. . . . 3
4 Soft shadows: Some of the points on the receiver is neither fully lit or fully in shadow resulting in a penumbra area . . . 4
5 Shadow map example. The darker the pixel the closer it is . . . 7
6 Rendering pipeline. To get to the light's clip space we must go from the camera's eye space, through world space, the light's eye space and finally into the lights clip space . 8
7 Aliasing: If the area seen through one pixel in the shadow map is seen through multiple pixels from the camera, aliasing occurs. That is if the depth value of one pixel in the shadow map is used to shade multiple pixels in the final image. . . . 9
8 Depth buffer precision: Pixels can be falsely classified as being in shadow because of limited precision. In this figure two points (black and yellow) are to be classified for shadowing. The black point will be wrongly shadowed even though both points should be lit. . . . 10
9 Near and far plane settings: In (a) the optimal near and far plane settings have been applied. It is seen that the contrast of the shadow map is large. Objects being close to the light source are nearly black vice versa. With the near and far plane not being optimized (b), the contrast of the shadow map degrades resulting in less accuracy . . 12
10 The effect of biasing. When no offset is used (a) the results is many falsely shaded areas. When the offset is too high (b) the shadow leaves the object making it appear to be floating. In (c) the offset is just right . . . 15
11 PCF illustrated. Taking the nearest value results in a point being either shadowed or not. Using Percentage Closer Filtering allows a point to be partially shadowed, giving anti-aliased shadow edges . . . 16
12 Using Percentage Closer Filtering. (a) No filtering. (b) 2×2 PCF . . . 16
13 Shadow Map Timing. As seen drawing the scene from the light source and copying the depth buffer to a texture occupies approximately 45% of the total rendering time . . 17
14 Timing when using RenderTexture(PBuffer) . . . 19
15 Shadow regions . . . 21
16 Determining whether or not a point is in shadow. Two rays A and B are followed into the scene. The counter values show whether or not the point hit is in shadow 22
17 Finding silhouette edges. If the angle between the normal and a vector towards the light is less than 90 degrees for both triangles, the edge is *not* a silhouette edge and vice versa . . . 22
18 The generated shadow volume shown as the red line . . . 23
19 Shadow Volumes and Stencil buffer . . . 25
20 Z-Pass (a) and Z-Fail (b). In (a) the resulting stencil value is dependant on the placement of the camera. In (b) the placement does not change the result. . . . 27

21 Z-Pass (a) and Z-Fail (b). The resulting shadowing in (a) is wrong due to the camera being inside a shadow volume. In (b) the resulting shadow is not affected by the camera placement 27
22 Two-Sided Stencil Testing: There is noticeable performance increase when using two-sided stencil testing instead of one-sided. Rendered scene can be seen in appendix A.1.3e. 29
23 Comparison of Shadow Maps and Shadow Volumes 29
24 The scene seen before and after perspective projection. Objects close to the camera are larger than objects far from the camera in post perspective space. This means objects close to the camera occupies a larger area in the shadow map 31
25 A point light in front of the viewer will remain a point light in front of the viewer(left). A point light behind the viewer will be transformed behind the infinity-plane and will be inverted(center), and a point light on the plane through the view point will become a directional light(right). 32
26 The object on the left will cast shadow into the scene. However this object will not be present in the shadow map 33
27 Object behind camera transformed behind infinity plane 34
28 Using transformed scene. In (a) the scene is rendered from the camera, in (b) from the light source and in (c) the transformed scene is rendered from the transformed light source. Objects close to the camera becomes larger than objects far from the camera after transformation 37
29 Comparing Perspective Shadow Maps to ordinary Shadow Maps. When using Perspective Shadow Maps (a) the shadows close to the camera are better due to the objects being larger in the shadow map. As the distance increases objects will be smaller in the shadow map and the quality of the shadow degrades 38
30 Points of interest calculated in original article 39
31 Points of interest calculated in this thesis 39
32 Using Cube-Clipping: When the light source (shown in yellow) is far away from the camera frustum (a and b) there is not much difference in the original approach (a) and using cube-clipping (b). However, as the light source gets nearer to the camera frustum (c and d) the difference in shadow quality in the original approach (c) and using cube-clipping (d) is very noticeable. 41
33 Overview of Chan's Hybrid algorithm. Using Shadow Maps (a) silhouette pixels are classified (b). The Silhouette pixels are shown in white. The scene is then rendered (c) using shadow volumes only at these silhouette pixels and using shadow map elsewhere 43
34 Illustrating silhouette contra nonsilhouette pixels 44
35 CG vertex program for finding silhouette pixels 46
36 CG fragment program for finding silhouette pixels 46
37 Classifying silhouette pixels. Pixels that lie in the proximity of a shadow boundary are classified as silhouette pixels and are colored white. Other pixels are colored black . . 46
38 CG fragment program creating computation mask 47
39 Comparing shadow volumes pixels. Shadow Volumes cover most of the screen when using the ordinary Shadow Volumes algorithm(b). Using Chan's hybrid (c) only a small amount of pixels is processed when rendering shadow volumes 48
40 CG vertex program for shadowmap lookup 49

41 CG fragment program for shadowmap lookup 49
42 Using ALPHA and INTENSITY as return values. When using ALPHA(a) some points are falsely shaded due to failed alpha test. Using INTENSITY(b) results in artifacts at shadow boundary 50
43 The final rendering using Chan's Hybrid algorithm with a mapsize of 512x512. The edges of the shadow area are perfect just as when using the Shadow Volumes algorithm. Also acne as result of limited precision and resolution issues are avoided, since shadow volumes are used to render pixels that would normally have acne when using the Shadow Map algorithm. 51
44 UML Class Diagram 53
45 A screen-shot of the program. Using the menu different algorithms and scenes can be chosen during runtime 55
46 The scenes used for analysis 58
47 Timing for the five test scenes 59
48 Test scenes rendered using Shadow Maps 72
49 Test scenes rendered using Perspective Shadow Maps 73
50 Test scenes rendered using Shadow Volumes 74
51 Test scenes rendered using Chan's hybrid 75
52 Quality as function of shadow map size using the smallest possible cutoff angle enclosing the Water Tower 76
53 Quality as function of shadow map size using a larger cutoff angle 77
54 Quality of PSM's. 78
55 A case in which PSM's quality is much worse than SM' quality 79
56 Using too low shadow map size can lead to small or thin objects not to be classified as silhouette pixels, and therefore be wrongly shadowed 80
57 SM's and PSM's are not concerned with geometry, as SV's and Chan's. 81

1 Introduction

Shadows are a key factor in computer generated images. Not only do they add realism to the images, furthermore without shadows, spatial relationship in a scene can be very hard to determine. An example is shown in figure 1. In the first image there is no shadows, making it very hard to determine whether the ball is on the ground or if it is floating above the ground. This is, on the other hand, easily determined in the following images, where shadows are shown. Here the ball is first on the ground secondly floating.

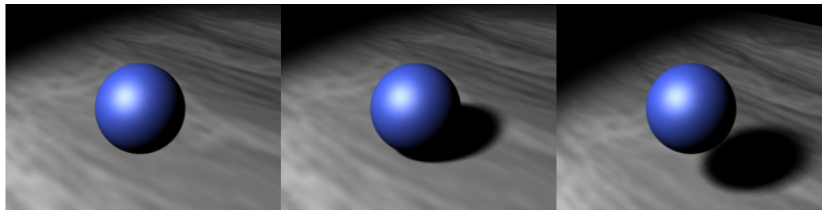


Figure 1: Spatial Relationship: In the first image it is hard to determine whether the ball is on the ground or floating in the air. This is easily determined in the two next images due to the shadows cast by the ball

In current game engines, usually a single algorithm is chosen for shadow calculations. This choice, of a single algorithm, can limit the game engine either in speed or in shadow quality. Designers therefore often face the challenge of choosing a single algorithm, thereby having to compromise with regard to the different aspects of shadow generation.

The goal of this project is to analyze the possibilities of using multiple shadow algorithms within one game engine. That is, instead of confining the game engine to use only one algorithm for shadow calculations, this thesis will try to look at the possibility of using multiple algorithms, by changing the choice of algorithm based on the current scene. This might be done dynamically during runtime or at least be specified as a parameter during level design.

In the thesis we will look at four different shadow algorithms, two designed for speed, and two design for quality. We will look into the strengths and weaknesses of the four algorithms and try to determine which algorithms are best for certain cases. Having done this we will look at the possibility of choosing the algorithm best suited for the current scene. The four algorithms are:

Shadow Maps Shadow Maps was first introduced by L.Williams [WILL78], and is a widely used algorithm. The algorithm excels in speed, having no dependency on the number of objects in the scene. The problem is aliasing resulting in shadows with jagged edges, which can be minimized but usually not completely removed.

Shadow Volumes Introduced by Frank Crow [CROW77], Shadow Volumes creates perfect hard shadows. The drawback is a large fillrate consumption and

object number dependency, resulting in low speeds in high scenes with high complexity shadows.

Perspective Shadow Maps Built upon the idea of Shadow Maps, Marc Stamminger and George Drettakis[SD02] has proposed this algorithm minimizing the problem of jagged edges, while still keeping the high speed of Shadow Maps.

Hybrid by Chan Recently Eric Chan and Frédo Durand [CD04] proposed this algorithm, that combines the Shadow Map and Shadow Volume algorithms. Using Shadow Maps this hybrid algorithm minimizes the fillrate consumption of the original Shadow Volumes algorithm, generating near-perfect shadows minimizing the speed penalty of Shadow Volumes.

During the thesis the algorithms will be explained, implementations will be shown and analysis of the algorithms using different cases will be carried out hopefully giving a picture of the strengths and weaknesses of the algorithms, reminding the above goals for the perfect algorithm. The implementations will be combined in a single application allowing the user to quickly change between the four algorithms at runtime. This application is used for comparing the algorithms and looking into the possibility of dynamically switching algorithms.

Many papers have been written on the subject of strengths and weaknesses of shadow algorithms and some have done comparisons of different algorithms([S92], [WPF90]). These comparisons though, have either focussed on different implementations of the same basic algorithm or tried to cover all aspects of shadow generation within one paper, none of them looking into the possibility of using multiple algorithms. In this thesis we will look into both algorithms optimized for speed and algorithms optimized for quality.

1.1 Hard and Soft Shadows

So how is shadows computed in computer generated images? Well, basically each point in the scene must be shaded according to the amount of light it receives. A point that receives no light, will of course be fully shadowed. This will happen if something is blocking the way from the light source to the point. The term *blocker* is used for an object blocking light. An object that receives shadow is called a *receiver*. Notice that an object can be both a blocker and a receiver (see figure 2).

But will a point always be fully lit or fully in shadow? As we know, from real life, this is not the case, as it depends on the type of light source. Here are some examples.

In figure 3 a simple scene with a point light source, a single blocker and a receiver is set up. The point light is assumed to be infinitely small. As it can be seen, a point on the receiver can either 'see' the light or it cannot. This will result in images with so-called *hard* shadows. The transition from lit areas to shadowed areas are very sharp and somewhat unrealistic.

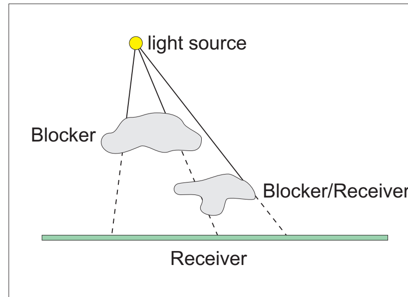


Figure 2: Blocker, Receiver, light source

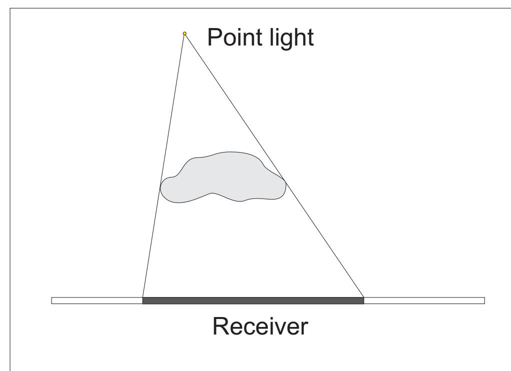


Figure 3: Hard Shadows: Points on the receiver can either see the light source or not, resulting in hard shadows with sharp edges.

In the real world we know that shadows does not look like in the above case. The transition is usually more smooth. But why is that? One of the reasons is that the assumption above, about an infinitely small light source, is rarely the case in the real world. Light sources in the real world cover a given area in space and therefore a point on the reciever might be able to 'see' a part of the light source. This results in so-called *soft* shadows. In figure 4 such an example is shown. As it can be seen, some points are neither fully shadowed nor fully lit. The area covering these points are called the *penumbra*. Areas that are fully shadowed are called *umbra*.

The size of the penumbra is affected by geometric relationships between the light source, the blocker and the receiver. A large light source will result in larger penumbra areas and vice versa. Also the ratio r between the distance from the light source to the blocker d_{lb} and the distance between the blocker and the receiver d_{br} affects the size of the penumbra, which increases as $r = \frac{d_{lb}}{d_{br}}$ decreases.

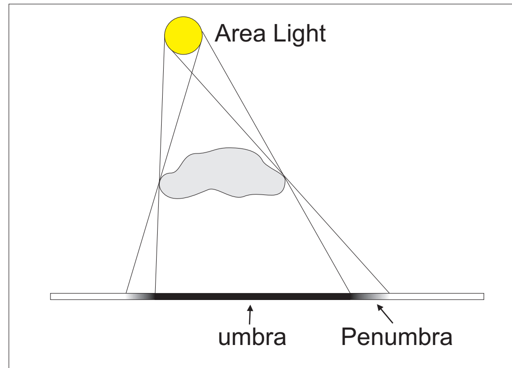


Figure 4: Soft shadows: Some of the points on the receiver is neither fully lit or fully in shadow resulting in a penumbra area

1.2 Real-Time Shadows

As with most things in computer systems, shadows in computer generated images are merely an approximation to the real world. How well this approximation should be, depends highly on the application in which the shadows are to be calculated. Some applications may require the shadows to be as realistic as possible, not being concerned about the time required to calculate the shadows. Other applications wishes to calculate the shadows very fast accepting a less realistic look, while others again, can accept a smaller penalty on speed resulting in a better approximation for the shadows.

So in real-time engines what would be the perfect shadow algorithm? Several goals needs to be reached to built the perfect shadow algorithm.

Speed: The speed of the algorithm should be as high as possible. For a game-engine images should be produced at a rate of minimum 25 frames pr. second, however since multiple effects are used in todays games, the shadow algorithm alone can not take up all of this time. The faster the shadow algorithm the better. The speed of the algorithm should not be affected by movement in the scene. Objects, both blockers, receivers and lights, should be able to move freely without penalty on the speed.

Quality: The shadows should look good. No artifacts should be visible. When dealing with soft shadows the penumbra area should look realistic. Remember though, that regarding quality, the most important thing is not for the shadows to *be* realistic but to *look* realistic.

Robustness: The algorithm should generate 'true' results independent on the location etc. of scene objects. Special settings for each frame should not be necessary.

Generality: The algorithm should support multiple kinds of geometric objects. That is, the algorithm should not limit itself to only work for a given type of objects, for example triangles. Furthermore, the algorithm should work even

for objects shadowing themselves.

Reaching all of the above goals is an almost impossible task in a real-time application, but the above gives a view of which concepts should be taken into account when evaluating a specific shadow algorithm.

1.3 Related Work

Since Williams and Crow in 1978 and 1977 respectively introduced the ideas of Shadow Maps and Shadow Volumes these have been the two most widely used algorithms for creating shadows in dynamic scenes. Most algorithms used today, are actually build upon these two. In 2001, the idea of using multiple shadow maps was introduced [ASM01]. Using a tree of shadow maps instead of just a single one, introduced the ability to sample different regions at different resolutions, thereby minimizing aliasing problems in ordinary shadow maps. However, this algorithm can not be completely mapped to hardware as stated in [DDSM03] and [MT04]. [MT04] even claims the approach is *slow and not suitable for real-time applications.*

Trying to minimize biasing problems, which will be explained later, Weiskopf and Ertl in 2003 came up with the idea of Dual Depth Shadow Maps(DDSM) [DDSM03]. Instead of saving depth values of the nearest polygons in the shadow map, DDSM defines an adaptive bias value calculated using the two nearest polygons.

As will be apparent later, aliasing is the worst problem when using shadow maps. To avoid this multiple article suggest the idea of transforming the scene before creating the shadow map. In 2004 four papers was introduced at the "Eurographics Symposium on Rendering" all exploiting the idea.

Alias Free Shadow Maps(AFSM)[AL04] avoids aliasing by transforming the scene points $p(x, y)$ into light space $p_l(x', y')$ thereby specifying the coordinates used to created the shadow map, instead of using the standard uniform shadow map. This way aliasing is completely removed. The drawback is again that AFSM cannot be done in hardware, increasing the CPU work.

A Lixel For Every Pixel[CG04] also adopts the idea of transforming the scene prior to creating the shadow map. In fact they claim to remove aliasing completely over a *number of chosen planes*, by calculating a perfect transformation matrix solving a small linear system. This approach is as stated by the authors themselves *not optimal in a much more 'free-form' environment, where the important shadow receivers cannot be described in this simple way.*

Light Space Perspective Shadow Maps[WSP04] uses the same idea using a perspective transformation balancing the shadow map texel distribution. The transformation is done using a frustum perpendicular to the light direction, thereby avoiding the change of light source type after transformation that is normally the case when transforming lights.

Trapezoidal Shadow Maps(TSM)[MT04] also uses transformations, but instead

of transforming the scene into light space[1], TSM uses trapezoids to create a transformation matrix that better utilizes the shadow map for the area scene from the eye.

Trying to optimize the speed, when using shadow volumes, Aila and Möller[AM04] divides the framebuffer into 8×8 pixel tiles. By classifying each tile as being either fully lit, fully shadowed or a possible boundary tile, the number of pixels rasterized using shadow volumes are restricted to shadow boundaries. This increases speed by minimizing fill-rate consumption known to be one of the biggest problems using shadow volumes.

In 2004 "CC Shadow Volumes"[LWGM04] was introduced using culling and clamping to minimize the number of pixels in which shadow volumes are drawn. Culling removes shadow volumes that are themselves in shadow, and Clamping restricts shadow volumes to regions containing shadow receivers, thereby minimizing fillrate-consumption.

[1] As done in Perspective Shadow Maps

2 Shadow Maps

2.1 Theory

Shadow mapping was introduced by L. Williams [WILL78]. The algorithm consists of two major steps. First the scene is rendered from the light-source using a virtual camera. This camera is set up at the location of the light-source, pointing in the same direction as the light source. The depth-information of this render is then saved in a so-called shadow map. This gives a gray-scale image in which dark pixels are points close to the light, and light pixels are points far away from the light. An example of such a shadow map is seen in figure 5.

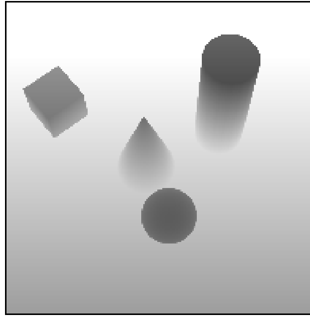


Figure 5: Shadow map example. The darker the pixel the closer it is

In the second pass the scene is rendered from the eye. During the rendering, each point in eye space is transformed into light space and the transformed depth value z_l is compared to the corresponding value in the shadow map z_{sm}. If the value is greater than the shadow map value, there must exist an object closer to the light, and the point must therefore be in shadow. On the other hand, if the value equals the shadow map value the point must be the closest to the light and must therefore be lit. That is

$$z_l = z_{sm} \Rightarrow \text{point is lit}$$
$$z_l > z_{sm} \Rightarrow \text{point is in shadow}$$

It should be noted that theoretically z_l is never less than z_{sm}, since z_{sm} is the distance to the object closest to the light.

2.1.1 Transforming Depth Values

The transformation of points from the camera's eye space to the light's clip space is best described using figure 6. Given the camera view matrix $\mathbf{V}_c$, the virtual light view matrix $\mathbf{V}_l$ and the virtual light projection matrix $\mathbf{P}_l$, the transformation can be described as

$$\mathbf{T} = \mathbf{P}_l \times \mathbf{V}_l \times \mathbf{V}_c^{-1}$$

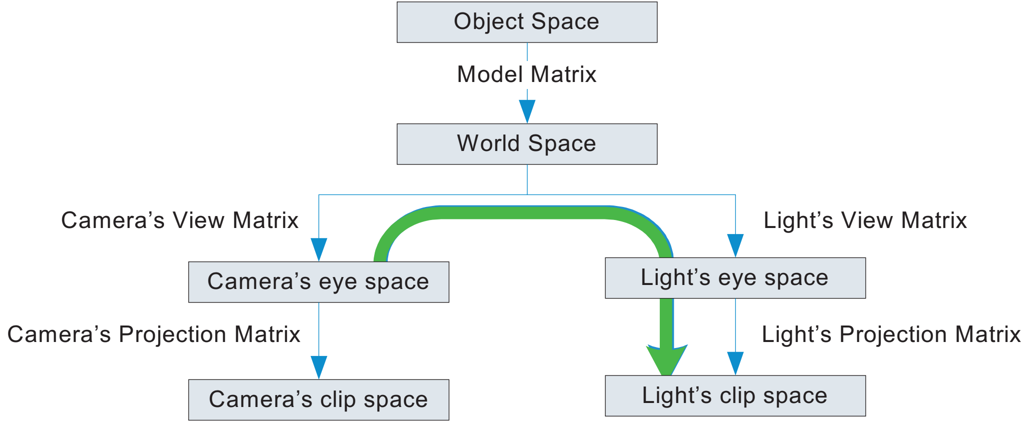


Figure 6: Rendering pipeline. To get to the light's clip space we must go from the camera's eye space, through world space, the light's eye space and finally into the lights clip space

This transformation results in a point in the light's clip space. Since the shadow map is just a 2D projection of this, the only thing left to do, is to transform the range of the $x-$, $y-$ and $z-$components of the point from $[-1, 1]$ to $[0, 1]$. This is done by multiplying the transformation matrix $\mathbf{T}$ with a bias-matrix $\mathbf{B}$:

$$\mathbf{B} = \begin{pmatrix} \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix}$$

The total transformation matrix then becomes:

$$\mathbf{T} = \mathbf{B} \times \mathbf{P}_l \times \mathbf{V}_l \times \mathbf{V}_c^{-1}$$

2.1.2 Aliasing

One of the most serious drawbacks of the algorithm is aliasing. A formulation of the aliasing problem is done in [SD02], and is most easily explained using figure 7. When a bundle of rays, going through a pixel in the shadow map, hits a surface at distance r_s, at an angle α, the resulting area d in the final image can be approximated as:

$$d = d_s \frac{r_s}{r_i} \frac{cos\beta}{cos\alpha} \tag{1}$$

where β is the angle between the light direction and the surface normal and r_i is the distance from the camera to the intersection point.

Aliasing occurs whenever d is larger than the image pixel size d_i. This can appear due to two situations. When the camera is close to the object r_i is small

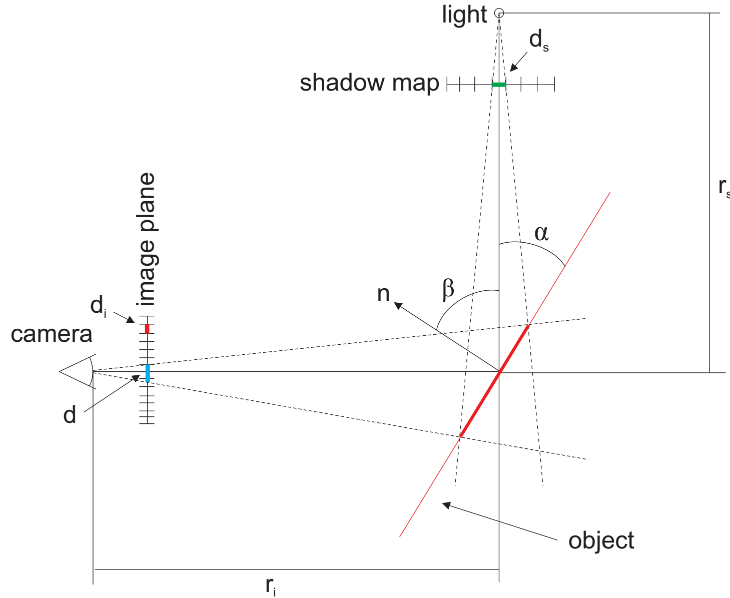


Figure 7: Aliasing: If the area seen through one pixel in the shadow map is seen through multiple pixels from the camera, aliasing occurs. That is if the depth value of one pixel in the shadow map is used to shade multiple pixels in the final image.

and d becomes large. This case is referred to as *perspective aliasing*. Another cause of aliasing arises when α is small. That is, when objects are almost parallel to the light direction. This is referred to as *projective aliasing*. A number of things can be done to minimize these aliasing-problems, and these will be discussed later (Section 2.2.9).

2.1.3 Biasing

Another problem when using shadow mapping is the limitation in resolution and precision which causes problems. In figure 8 such an example is shown. In this figure the transformed depth value, Z_{ct},of two points(black and yellow) on an object scene from the camera, are compared to the depth values, Z_l,stored in the shadow map(green). Both points should be lit, but because of limited resolution the depth values do not match perfectly, thereby resulting in one of the points being classified as being in shadow ($Z_{ct} > Z_l$), and the other being classified as being lit($Z_{ct} \leq Z_l$). The possible error will increase as the slope with regard to the camera viewing direction of the polygon $\delta x/\delta z$ increases, so the points should be offset by a value corresponding to their slope with respect too the light.

2.1.4 Spot Lights

Lastly it should be noted that the simple implementation of shadow maps only supports spot lights. This is due to the fact that a single shadow map cannot

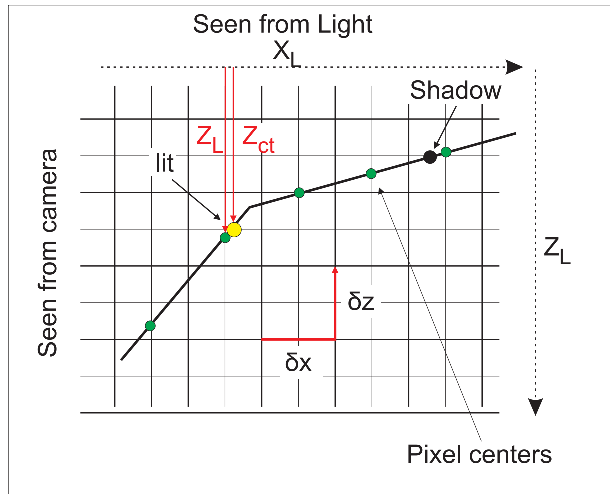


Figure 8: Depth buffer precision: Pixels can be falsely classified as being in shadow because of limited precision. In this figure two points (black and yellow) are to be classified for shadowing. The black point will be wrongly shadowed even though both points should be lit.

encompass the entire hemisphere around itself. Cube mapping could be used to allow point light sources, however this would require up to 6 shadow maps and therefore 6 renderings of the scene, decreasing the speed of the algorithm considerably.

2.2 Implementation

The pseudo-code for the implementation of the Shadow Maps (SM) algorithm can be seen below.

```
init_textures
for each light
    create a virtual camera at lights position
    draw scene using virtual camera
    copy depth buffer to texture
end
for each light
    enable light(i)
    calculate texture coordinates using virtual camera
    enable testing depth against texture value
    draw scene using real camera
end
```

The steps of the algorithm will now be described, looking at the important settings during each of these. For a full view of the implementation see appendices C.5 and C.1.

2.2.1 Initialize Textures

Before starting the rendering, textures to hold the shadow maps are initialized. That is, for each light a texture is generated.

```
GenTextures(lights, shadowmap);
```

2.2.2 Virtual Camera

When creating the virtual camera at the light source we have to specify the properties of this camera. These are:

- Position
- Direction
- Viewing Angle
- Aspect ratio
- Up vector
- Near plane distance
- Far plane distance

The four first properties are clearly determined by the light source. The camera should be positioned at the light source position, pointing in the same direction as the light source. The viewing angle of the camera should be twice the cutoff-angle of the light source ensuring that the shadow map 'sees' the same area as the light source. The aspect ratio of the virtual camera should be one since the spotlight is assumed to light an area described by a cone. The up vector of the camera is actually not important when creating the shadow map, so we only have to assure that its different from the direction-vector of the light source.

Now, the near and far plane has to be specified. As stated earlier the depth buffer has limited precision. The errors occurring because of this should be minimized. This is done by minimizing the distance between the near and the far plane. Therefore, a function has been created that calculates the optimal near and far plane values such that the objects of the scene is only just covered. This will minimize the errors seen when comparing the depth values in later steps. In figure 9 it is shown how the settings of the near and far plane can affect the shadow map.

```
Camera* light_cam = new Camera(light(i)->get_position(),
                               light(i)->get_direction(),
                               Vec3f(0,1,0),
                               light(i)->get_cutoff_angle()*2.0,
                               1,
                               100);
light_cam->calculate_near_and_far();
light_cam->alter_up_vector();
```

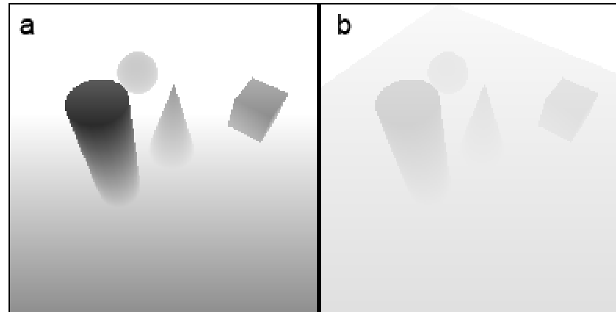


Figure 9: Near and far plane settings: In (a) the optimal near and far plane settings have been applied. It is seen that the contrast of the shadow map is large. Objects being close to the light source are nearly black vice versa. With the near and far plane not being optimized (b), the contrast of the shadow map degrades resulting in less accuracy

2.2.3 Draw Scene Using Virtual Camera

When drawing the scene using the virtual camera the only thing of interest is the depth values. Therefore all effects such as lighting etc. should be disabled to increase speed. Furthermore it is important that the viewport is only the size of the shadowmap.

```
Enable(DEPTH_TEST);
Enable(CULL_FACE);
CullFace(BACK);
ShadeModel(FLAT);
ColorMask(0,0,0,0);
Disable(LIGHTING);
Viewport(0,0,map_size,map_size);
load_projection(light_cam);
load_modelview(light_cam);
draw_scene();
```

2.2.4 Saving Depth Values

Saving the depth values can be done by copying these to a texture. Standard OpenGL does not allow this, but using the OpenGL extension GL_ARB_depth_texture [ADT02] enables us to call glCopyTexImage2D using GL_DEPTH_COMPONENT as internalFormat parameter.

Copying the frame buffer to a texture is not a very efficient way to store the depth value, and later (Section 2.2.10), the use of the so-called *pbuffer* allowing rendering directly to a texture, will be looked into.

```
Enable(TEXTURE_2D);
BindTexture(TEXTURE_2D, shadow_map[i]);
CopyTexImage2D(TEXTURE_2D, 0, DEPTH_COMPONENT, 0, 0,map_size, map_size, 0);
```

2.2.5 Rendering Using Shadow Map

After saving the shadow maps, the scene must be rendered once for each light source blending the results to get the final image. That is, we turn on one light at a time and render the scene using the shadow map for this light

```
ligths->turn_all_off();
light(i)->turn_on();
```

After enabling the lights the texture matrix must be calculated and used to generate the texture coordinates. The texture matrix is calculated as describe in Section 2.1 except for a slight modification. Having calculated the texture matrix needed to transform a point from eye-space into texture coordinates in the shadow map, we need to make OpenGL use this matrix when generating texture coordinates. The transformation can be seen as going from a point (x_e, y_e, z_e, w_e) to a texture coordinate (s, t, r, q). Here the s and t coordinates are the usual texture coordinates while the r value corresponds to the depth value transformed into the lights eye space. r/q is then used in comparing the depth value to the shadow map value. Specifying the transformation in OpenGL is done by specifying transformation functions for each coordinates s, t, r and q. Having calculated the texture matrix we can do this by specifying the four components as the 1'st, 2'nd, 3'rd and 4'th row of the texture matrix.

A little trick when specifying the texture functions is to use EYE_PLANE when calling TexGen. This causes OpenGL to automatically multiply the texture matrix with the inverse of the current modelview matrix. This means that the texture matrix supplied to TexGen should not take the modelview matrix into account. The texture matrix should therefore be calculated as:

$$\mathbf{T} = \mathbf{B} \times \mathbf{P}_l \times \mathbf{V}_l$$

The important thing here is to ensure that the current modelview matrix is the modelview matrix of the camera when calling TexGen.

```
load_modelview(cam);
texture_matrix = BiasMatrix * LightProjectionMatrix * LightViewMatrix;
TexGeni(S, TEXTURE_GEN_MODE, EYE_LINEAR);
TexGenfv(S, EYE_PLANE, texture_matrix(row 0));
TexGenfv(T, EYE_PLANE, texture_matrix(row 1));
TexGenfv(R, EYE_PLANE, texture_matrix(row 2));
TexGenfv(Q, EYE_PLANE, texture_matrix(row 3));
Enable(TEXTURE_GEN_S);
Enable(TEXTURE_GEN_T);
Enable(TEXTURE_GEN_R);
Enable(TEXTURE_GEN_Q);
```

2.2.6 Comparing Depth Values

Using the shadow map to shade the scene can only be done using a second extension to the OpenGL API. This time the extension is ARB_shadow[AS02], enabling us to

do a comparison between the transformed depth value and the shadow map value. This is done by calling

```
BindTexture(TEXTURE_2D, tex_map);
Enable(TEXTURE_2D);
TexParameteri (TEXTURE 2D, TEXTURE MIN FILTER, NEAREST) ;
TexParameteri (TEXTURE 2D, TEXTURE MAG FILTER, NEAREST) ;
TexParameteri(TEXTURE_2D, TEXTURE_COMPARE_MODE_ARB, COMPARE_R_TO_TEXTURE);
TexParameteri(TEXTURE_2D, TEXTURE_COMPARE_FUNC_ARB, LEQUAL);
TexParameteri(TEXTURE_2D, DEPTH_TEXTURE_MODE_ARB, INTENSITY);
```

When rendering the scene, the r/q value described above will then be compared to the value stored in the shadow map. The comparison only passes if the value is less than or equal to the stored value and an intensity value is the result. This will result in lit area's to have intensity one, and shadowed areas to have intensity zero, creating the wanted result.

It should be noticed, that shadows generated using this method will be completely black. If ambient light is wanted, an ALPHA result could be used. However the scene must then be rendered twice. First shadowed areas are drawn using alpha testing and afterwards lit areas are drawn also using alpha testing.

A third possibility is to use fragment shaders when rendering. The shader could then shade pixels differently according to the result of the comparison, resulting in only one pass to be needed.

2.2.7 Rendering Final Image

All that is left now, is to render the image enabling blending to allow multiple light sources

```
Enable(BLEND);
BlendFunc(ONE,ONE);
draw_scene();
```

2.2.8 Biasing

In figure 10(a) the scene as rendered now, can be seen, and the result is not good. This is due to the earlier described problem of the limited precision. As stated earlier we should offset the polygons by an amount proportional to their slope factor with regard to the light source. This can be done using the OpenGL function glPolygonOffset(factor,units). This function offsets the depth value by an amount equal to[2]:

$$offset = factor * DZ + r * units$$

where DZ is a measurement for slope of the polygon and r is the smallest value guaranteed to produce a resolvable offset for the given implementation.

To apply this polygon offset when creating the shadow map we insert the lines in the step of rendering from the light source(Section 2.2.3).

[2]see http://developer.3dlabs.com/GLmanpages/glpolygonoffset.htm

```
Enable(POLYGON_OFFSET_POINT);
Enable(POLYGON_OFFSET_FILL);
PolygonOffset(1.1, 4);
```

The result of this can be seen in figure 10(b and c). If the offset is too low shadowing artifacts will appear. If the offset is too high (b) the shadow will move away from the object giving the impression that it is floating. Finding the correct value is a challenging task which is highly scene dependant. In [KILL02] it is stated that using a *factor* value of 1.1 and a *units* value of 4.0 generally gives good results, which has also shown to be true for the scenes used in this project. In (c) the scene is shown with an offset *factor* of 1.1 and a *units* factor of 4.0.

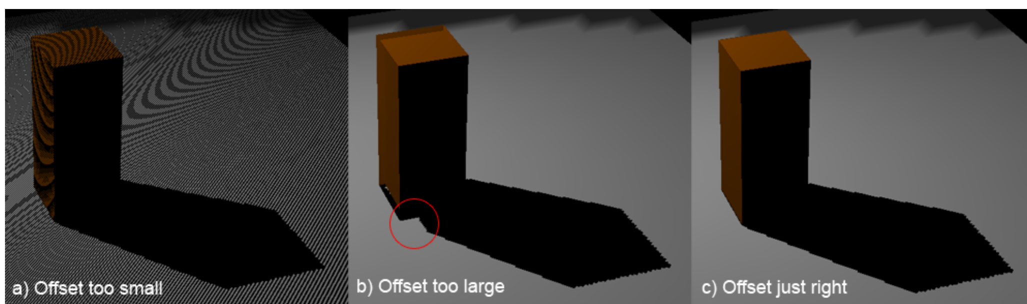


Figure 10: The effect of biasing. When no offset is used (a) the results is many falsely shaded areas. When the offset is too high (b) the shadow leaves the object making it appear to be floating. In (c) the offset is just right

Another technique would be to render only back-facing polygons into the shadow map. This would result in the depth value when seen from the camera clearly being smaller than the corresponding shadow map value, thereby avoiding the biasing problem. This would, however, require the objects to be closed meshes, since false shadowing would otherwise occur, thereby loosing shadow maps ability to calculate shadows independent of geometry.

2.2.9 Filtering

Although biasing errors are now minimized, there is still the problem of aliasing. In figure 12(a) we have focused on a shadow boundary, using a shadow map of size 256x256. The edges of the shadow is very jagged instead of being a straight line. One way to reduce this problem is to use a filtering mechanism to smooth the edge. Instead of just comparing the depth value to the closest shadow map value, it could be compared against a number of the closest values averaging the result. In figure 11 this is illustrated.

The above is known as Percentage Closer Filtering(PCF) and on NVIDIA graphic-cards today, using 4 pixels for PCF, can be done very easily not decreasing the speed of the algorithm. When comparing depth values we simply change some parameters when calling glTexParameter.

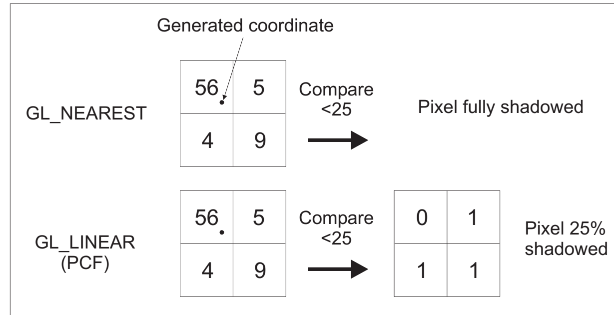


Figure 11: PCF illustrated. Taking the nearest value results in a point being either shadowed or not. Using Percentage Closer Filtering allows a point to be partially shadowed, giving anti-aliased shadow edges

```
TexParameteri(TEXTURE_2D, TEXTURE_MIN_FILTER, NEAREST);
TexParameteri(TEXTURE_2D, TEXTURE_MAG_FILTER, NEAREST);
 .
 changes to
 .
TexParameteri(TEXTURE_2D, TEXTURE_MIN_FILTER, LINEAR);
TexParameteri(TEXTURE_2D, TEXTURE_MAG_FILTER, LINEAR);
```

In figure 12(b) the effect of this change is seen. Although the result are still not perfect it is visually more pleasing, and keeping in mind that there is no cost for doing this, PCF should definitely be used.

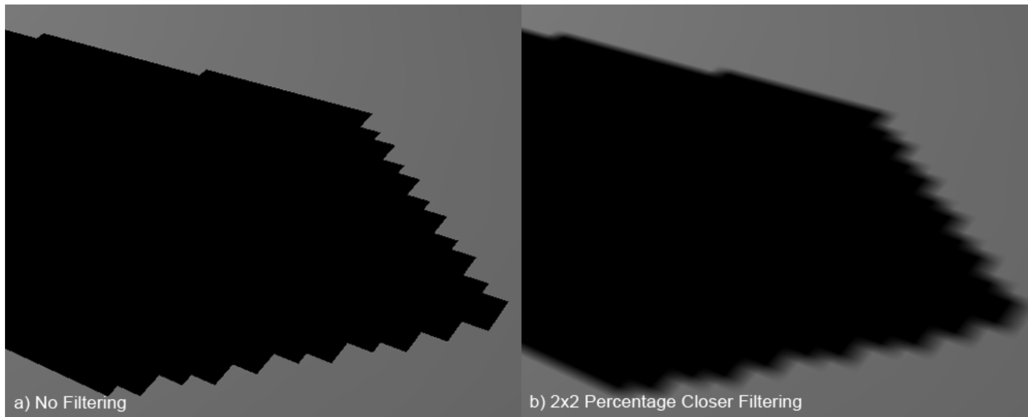


Figure 12: Using Percentage Closer Filtering. (a) No filtering. (b) 2×2 PCF

We could also increase the shadow map resolution, which would also reduce the jaggedness of the shadow boundaries. However this would decrease the speed of the algorithm and as far as the current state of the algorithm, the size can not be larger than the screen size. This however will be treated later (Section 2.2.10). As

an example the scene in figure 10 renders at 260 fps at a resolution of 1280x1024 using a shadow map of size 256x256, whereas the framerate is 200 fps using a map size of 1024x1024.

2.2.10 Rendering to Texture

In search of speed increases, a timing of the seperate stages of the algorithm was carried out for an arbitrary scene using shadow maps of 1024x1024 resolution. This can be seen in figure 13. As seen the two first stages of drawing the scene from the light source, and copying the current buffer to the shadow map, takes up approximately 45% of the total rendering time.

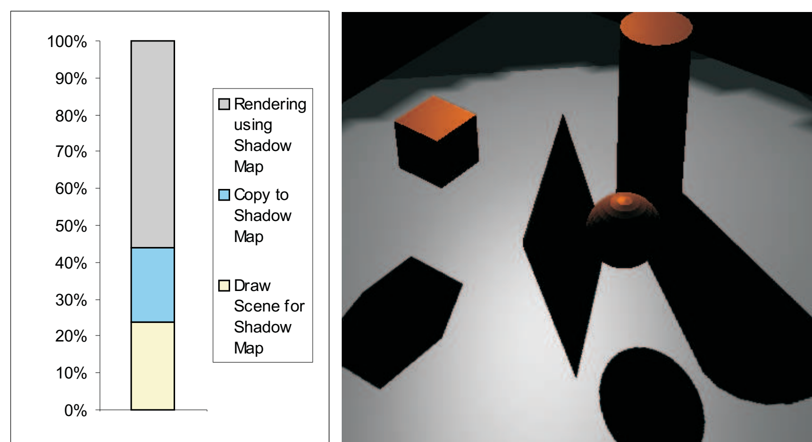


Figure 13: Shadow Map Timing. As seen drawing the scene from the light source and copying the depth buffer to a texture occupies approximately 45% of the total rendering time

Instead of copying to a texture use of a so-called PBuffer allows us to render the scene directly to the texture instead of first rendering the scene and then copying. This PBuffer is used in a package called RenderTexture[3] which will be used here. The RenderTexture class allows us to render to a texture simply by applying the calls beginCapture() and endCapture() around the OpenGL calls of interest. Using the RenderTexture class also eliminates the problem of the limited shadow map size. When using the RenderTexture class the pseudo-code for the algorithm changes to:

```
init RenderTexture
for each light
     beginCapture
     create a virtual camera at lights position
     draw scene using virtual camera
     endCapture
end
for each light
```

[3]http://www.markmark.net/misc/rendertexture.html

```
    enable light(i)
    calculate texture coordinates using virtual camera
    enable testing depth against texture value
    draw scene using real camera
end
```

As seen the change has eliminated the step of copying to a texture.

The initialization of the RenderTexture is done by simply specifying the type and size of the texture to render to. The type is specified in an initialization string and since depth values are the ones of interest the type is set to depthTex2D. We also specify that we are not interested in rgba values and that we want to render directly to a texture.

```
rt[i] = new RenderTexture();
rt[i]->Reset("rgba=0 depthTex2D depth rtt");
rt[i]->Initialize(map_size, map_size);
```

The RenderTexture can now be 'binded' and used as a normal texture in the following steps. Instead of using TEXTURE_2D as texture target when specifying parameters, we simply use rt[i]->getTextureTarget().

The new timing graph can be seen in figure 14. As it is seen the step of creating the shadow map has been decreased to 35% of the total time. This is not quite the performance increase as expected, but by inspecting the calls, it is was found that a major part of the time is used on changing contexts[4]. If a strategy of using only one shadow map for all light sources were adopted, only a single change of context would be necessary when using RenderTexture as opposed to the earlier implementation where a copy to texture had to be done for each light source. As more light sources are used, the increase in performance would therefore be significant, making the use of rendering directly to a texture a much better approach. This is however not done in the current implementation, but could be a point of future work. Even though, the ability to create shadow maps of arbitrary sizes justifies using this approach anyway.

[4]Each render target has a its own context. Here a change from the main window context to the pbuffer context and back is needed

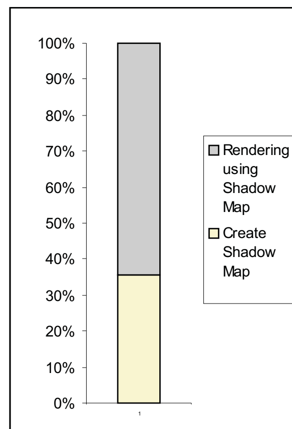
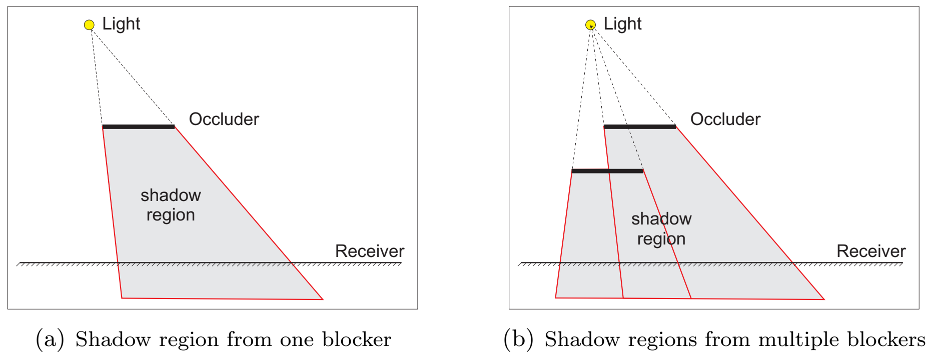


Figure 14: Timing when using RenderTexture(PBuffer)

3 Shadow Volumes

3.1 Theory

The idea of shadow volumes was first introduced by Crow [CROW77]. Each blocker, along with the light-sources in a scene, defines regions of space that are in shadow. The theory is most easily explained using a figure such as figure 15(a). Although the figure is in 2D, the theory works in the same way in 3D. In the figure, a scene is set up with one light-source, one blocker and one receiver. Drawing lines from the edges of the blocker in the opposite direction of the light-source, gives exactly the region that this blocker shadows. As seen in figure 15(b) multiple blockers can be present in the scene, shadowing each other.



(a) Shadow region from one blocker

(b) Shadow regions from multiple blockers

Figure 15: Shadow regions

When drawing the scene from the camera, an imaginary ray is followed through each pixel. Whether the point in space, found at the first intersection, is in shadow or not, is determined by keeping track of how many times shadow volumes are entered and exited. For this, a counter is used. The counter is incremented each time the ray enters a shadow volume and decremented each time the ray exits a shadow volume. If the counter value is equal to zero at the first intersection with an object, the ray has exited as many shadow volumes as it has entered, and therefore the point is not in shadow. On the other hand if the value is greater than zero, the ray has entered more shadow volumes than it has exited and the point must therefore be in shadow. An example is shown in figure 16.

But how is the shadow volume created in 3D space? As stated above the edges of the objects must first be found. Assuming the objects to be closed triangular meshes, this can be done by observing adjacent triangles in the object. If a triangle is facing towards the light and an adjacent triangle is facing away from the light, the edge between the two must be an edge of the object too, as seen from the light. Doing this test for all triangles in the mesh gives a so-called silhouette-edge. That is, if we look at the object from the light, the found silhouette-edge corresponds to the silhouette of the object.

Testing if a triangle is facing towards the light is easily done taking the dot

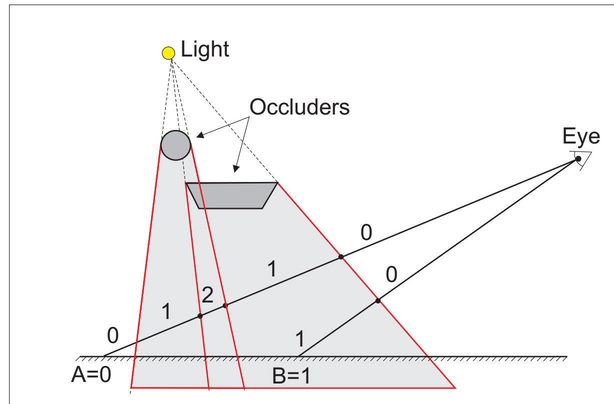


Figure 16: Determining whether or not a point is in shadow. Two rays A and B are followed into the scene. The counter values show whether or not the point hit is in shadow

product of the triangle normal and a unit vector pointing from the triangle center towards the light. If the value of the dot product, $\alpha > 0$, then the angle is less than 90 degrees and the triangle is front-facing. On the other hand, if $\alpha < 0$ the angle is larger than 90 degrees and the triangle is back-facing. This is seen in figure 17.

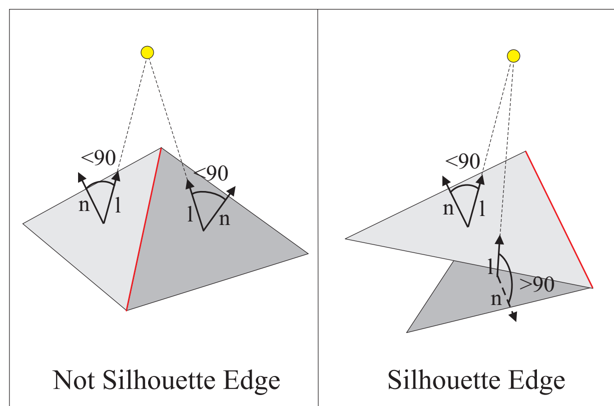


Figure 17: Finding silhouette edges. If the angle between the normal and a vector towards the light is less than 90 degrees for both triangles, the edge is *not* a silhouette edge and vice versa

After determining the edges of the object the shadow volume can now be created as follows: For each silhouette-edge consisting of the vertices A and B, a polygon is created using A and B, and two new vertices created by extruding copies of A and B away from the light. If the position of the light is L, and the

extrusion factor is e, the four vertices becomes

$$\begin{aligned} V_1 &= \langle B_x, B_y, B_z \rangle \\ V_2 &= \langle A_x, A_y, A_z \rangle \\ V_3 &= \langle A_x - L_x, A_y - L_y, A_z - L_z \rangle \cdot e \\ V_4 &= \langle B_x - L_x, B_y - L_y, B_z - L_z \rangle \cdot e \end{aligned} \tag{2}$$

The order of the vertices might seem strange, but is to ensure that the generated polygon faces outward with respect to the shadow volume. The reason for this will be apparent later.

Secondly all front-facing polygons of the object is added to the shadow volume, and lastly the back-facing triangles of the object is also extruded away from the light source and added to the shadow volume. The new vertices are

$$\begin{aligned} V_1 &= \langle A_x - L_x, A_y - L_y, A_z - L_z \rangle \cdot e \\ V_2 &= \langle B_x - L_x, B_y - L_y, B_z - L_z \rangle \cdot e \\ V_3 &= \langle C_x - L_x, C_y - L_y, C_z - L_z \rangle \cdot e \end{aligned} \tag{3}$$

This defines the closed region which the object shadows. The extrusion factor, e, should ideally be equal to infinity since no point behind the object should receive light from the light-source, and this will be dealt with later (Section 3.2.5). In figure 18 the created shadow volume is seen.

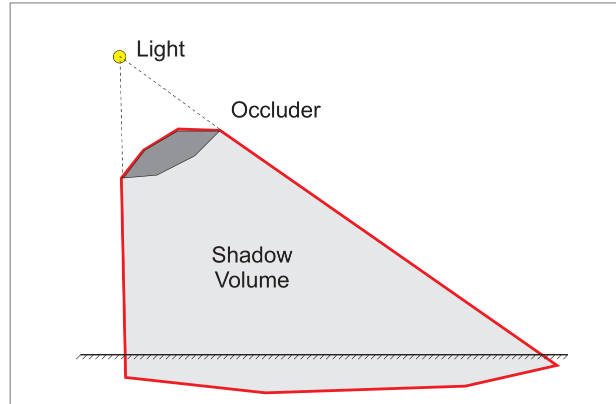


Figure 18: The generated shadow volume shown as the red line

3.2 Implementation

The pseudo code for the implementation of the Shadow Volume algorithm(SV) is seen below

```
create_adjacent_list

disable lights
draw_scene
for each light
    enable stencil buffer
    draw_shadow_volumes
end
enable stencil test
draw_scene
```

The full implementation can be seen in appendices C.6 and C.3

3.2.1 Adjacent List

Before starting the rendering, a list of adjacent triangles is made. This is done to increase the speed during rendering. The list must only be created once, since it will not change depending on the light source or the camera movement. Generating the list of adjacent triangles is done as below, keeping in mind that each triangle has 3 adjacent triangles with two shared vertices each. This can be concluded due to our assumption that models are closed meshes. The end result is a list in which each triangle knows its three adjacent triangles.

```
for each triangle i
    for all other triangles j
        shares two vertices?
            add j to i adjacent list
    end
end
```

3.2.2 Initializing Depth Values

For the later stage of drawing shadow volumes, the depth values of the scene must first be initialized. This can be explained using figure 16. If we look at the depth test as a way of stopping the ray sent from the camera we need to have the depth values of the scene. Thereby, all shadow volume enters and exits behind the objects in the scene can be discarded and will not affect the counting. We initialize the depth values by drawing the scene in shadow. This way we can save the depth values as well as shade the scene were it is in shadow.

```
Clear(BEPTH_BUFFER_BIT);
Enable_ambient_lighting();
Enable(DEPTH_TEST);
draw_scene();
```

3.2.3 Drawing Shadow Volumes

Simulating the rays going from the camera into scene and incrementing/decrementing a counter on shadow volumes enters and exits, can be done by using the so-called stencil buffer. This buffer can act as a counter, which can be incremented

or decremented depending on whether or not we are drawing front-facing or back-facing polygons. Therefore we can use it to define pixels that are in shadow and pixels that are not in shadow. This is done by first drawing front-facing shadow volumes incrementing the stencil buffer every time the depth test passes. That is, when the shadow volumes can be 'seen'. Secondly all back-facing shadow volumes are drawn, now decrementing the stencil buffer when the depth test passes. Thereby the stencil buffer will contain zeros in pixels where the scene is lit and non-zeros where the scene is shadowed. An important thing to keep in mind is that depth writing should be disable while drawing shadow volumes not to overwrite the depth values of the scene. To increase speed, it is important to disable all effects such as lighting and color buffer writes.

```
ColorMask(0,0,0,0);
DepthMask(0);
Disable(LIGHTING);
Enable(CULL_FACE);
ShadeModel(FLAT);

Clear(STENCIL_BIT);
Enable(STENCIL_TEST);
StencilFunc(ALWAYS,0,0);

for each light
    StencilOp(KEEP,KEEP,INCR);
    CullFace(BACK);
    draw_shadow_volumes();

    StencilOp(KEEP,KEEP,DECR);
    CullFace(FRONT);
    draw_shadow_volumes();
end
```

In figure 19 the shadow volumes and the resulting stencil buffer is seen. The stencil buffer is shown as black where the stencil value equals zero and white everywhere else. Now the stencil buffer can be used to determine in which pixels the scene are lit and in which it is not.

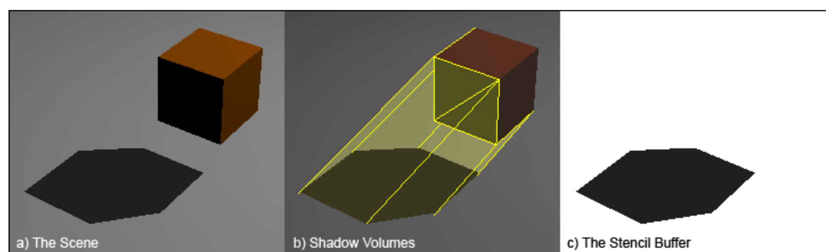


Figure 19: Shadow Volumes and Stencil buffer

3.2.4 Rendering Using Stencil Buffer

Now that we have the stencil buffer values to determine which pixels are lit, we can draw the scene again this time enabling the light sources. We just have to set up stencil testing such that only pixels where the stencil buffer equals zero are updated.

```
DepthMask(1);
DepthFunc(LEQUAL);
ColorMask(1);

StencilOp(KEEP,KEEP,KEEP);
StencilFunc(EQUAL,0,1);

Enable(LIGHTING);
draw_scene();
```

3.2.5 ZFail vs ZPass

Even though the algorithm seems to work now, a problem arises when the camera is position inside a shadow volume. When this happens, the stencil buffer is decremented when exiting the volume, there by setting the value equal to -1[5]. If the ray was to enter another volume later the stencil buffer would be incremented setting the value equal to zero. Remembering that the stencil buffer should be zero at lit areas only, it is seen that this will result in wrongly shadowed areas. The idea is further explained using figure 20(a). Here it is seen how the value of the stencil buffer at two points changes whether or not the camera is placed inside (blue rays) or outside (green rays) the shadow volume.

To correct this problem we adopt what is known as the Z-Fail method described in [EK03]. Instead of *incrementing* the stencil buffer on when entering and *decrementing* when exiting when the depth test passes, the opposite is done. That is, we *decrement* on entering and *increment* on exiting when the depth test fails. The new method can be thought of as following a ray from infinity towards the eye. The method is equivalent in that it produces the same results as the old method, but as seen in figure 20(b) the resulting stencil value is no longer affected by the camera being positioned inside the shadow volume.

In figure 21 an example using the two different methods is shown. In (a) the z-pass method is used resulting in the shadowed areas being wrong. In (b) the z-fail method is used resulting in correct shadows.

The problem above can also be seen as a problem of the shadow volumes being 'sliced' open by the near and far plane of the camera. Doing the reversed stencil test avoids problems with the near plane but the far plane could still cause problems. To prevent this, a special projection matrix is constructed that 'moves' the far clip plane to infinity. That is, a vertex placed at infinity using homogeneous

[5]actually the value is not -1 as the buffer cannot be less than 0. The value can be imitated though, using the extension EXT_stencil_wrap

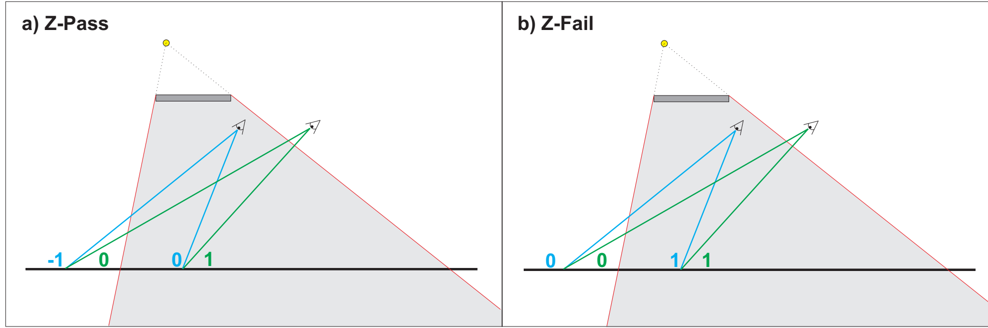


Figure 20: Z-Pass (a) and Z-Fail (b). In (a) the resulting stencil value is dependant on the placement of the camera. In (b) the placement does not change the result.

Figure 21: Z-Pass (a) and Z-Fail (b). The resulting shadowing in (a) is wrong due to the camera being inside a shadow volume. In (b) the resulting shadow is not affected by the camera placement

coordinates is *not* clipped by the far plane. The normal projection matrix P and the new 'infinity' matrix P_{inf} is seen below.

$$P = \begin{bmatrix} \frac{2\times Near}{Right-Left} & 0 & \frac{Right+Left}{Right-Left} & 0 \\ 0 & \frac{2\times Near}{Top-Bottom} & \frac{Top+Buttom}{Top-Bottom} & 0 \\ 0 & 0 & -\frac{Far+Near}{Far-Near} & -\frac{2\times Far\times Near}{Far-Near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$P_{inf} = \begin{bmatrix} \frac{2\times Near}{Right-Left} & 0 & \frac{Right+Left}{Right-Left} & 0 \\ 0 & \frac{2\times Near}{Top-Bottom} & \frac{Top+Buttom}{Top-Bottom} & 0 \\ 0 & 0 & -1 & -2\times Near \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Moving the far plane to infinity makes it possible to extrude the shadow volumes to infinity using homogeneous coordinates. The new vertices of the shadow volume now becomes

$$\begin{aligned} V_1 &= \langle B_x, B_y, B_z \rangle \\ V_2 &= \langle A_x, A_y, A_z \rangle \\ V_3 &= \langle A_x L_w - L_x A_w, A_y L_w - L_y A_w, A_z L_w - L_z A_w, 0 \rangle \\ V_4 &= \langle B_x L_w - L_x B_w, B_y L_w - L_y B_w, B_z L_w - L_z B_w, 0 \rangle \end{aligned} \tag{4}$$

and the extruded back-facing polygons becomes

$$\begin{aligned} V_1 &= \langle A_x L_w - L_x A_w, A_y L_w - L_y A_w, A_z L_w - L_z A_w, 0 \rangle \\ V_2 &= \langle B_x L_w - L_x B_w, B_y L_w - L_y B_w, B_z L_w - L_z B_w, 0 \rangle \\ V_3 &= \langle C_x L_w - L_x C_w, C_y L_w - L_y C_w, C_z L_w - L_z C_w, 0 \rangle \end{aligned} \tag{5}$$

By doing this, we have achieved what was wanted from earlier stages. The shadow volumes are extruded to infinity and still they are not clipped by the far plane. This results in a very robust implementation of the Shadow Volume algorithm.

3.2.6 Two-sided stencil buffer

Trying to increase the speed of the algorithm an attempt was made using two-sided stencil testing using the EXT_stencil_two_side[ASTS02] extension. This allows different stencil operations on front- and back-facing polygons in the same rendering pass. Thereby it is possible to render all shadow polygons in one pass instead of two. As seen in figure 22 some performance increase are obtained using this technique.

3.2.7 Comparison to Shadow Maps

A quick comparison to using the Shadow Map algorithm is shown in figure 23. It is seen that the problems of aliasing in Shadow Mapping is avoided completely when using Shadow Volumes. The biggest problem in using Shadow Volume which will become apparent in later sections being the reduction of speed as the number of polygons increases.

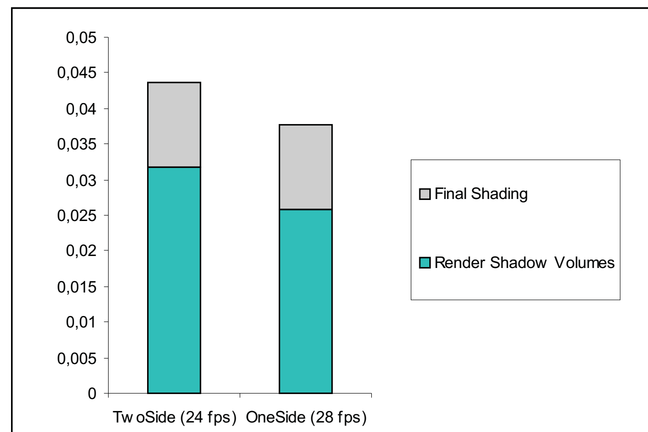


Figure 22: Two-Sided Stencil Testing: There is noticeable performance increase when using two-sided stencil testing instead of one-sided. Rendered scene can be seen in appendix A.1.3e.

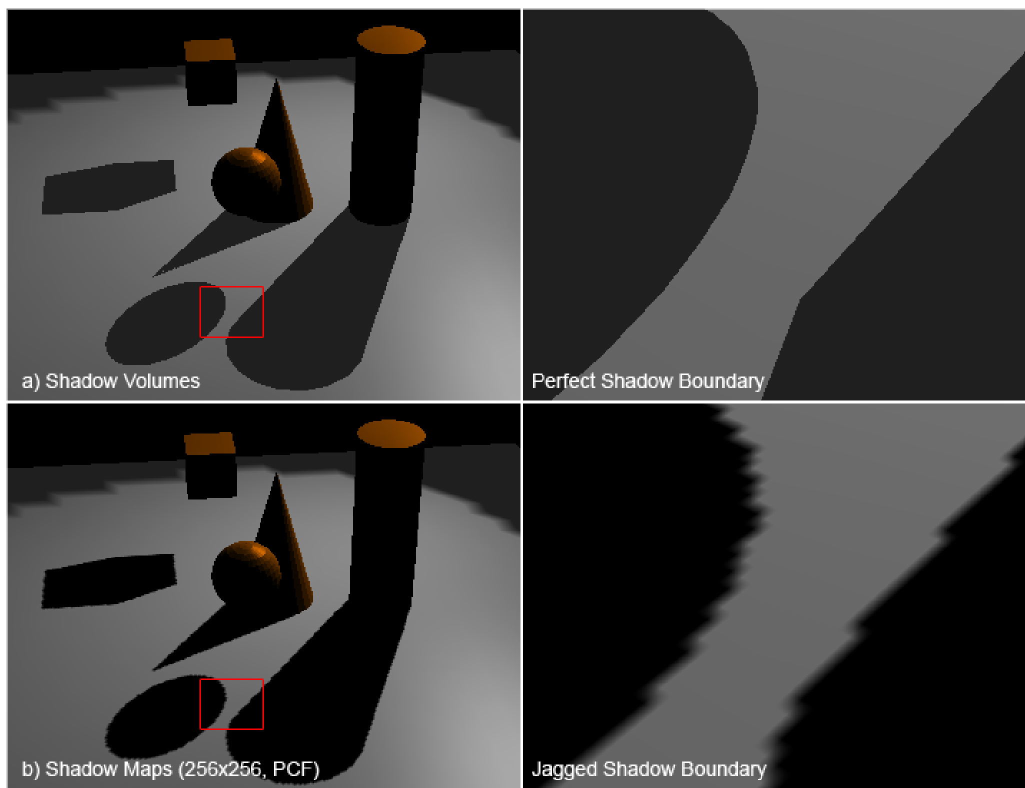


Figure 23: Comparison of Shadow Maps and Shadow Volumes

4 Perspective Shadow Maps

4.1 Theory

As stated earlier, one of the problems with ordinary shadow maps is perspective aliasing. Perspective Shadow Maps is a technique that can decrease and in some cases even remove this aliasing, by increasing the number of pixels used for objects close to the camera. The idea of perspective shadow maps was introduced by Marc Stamminger and George Drettakis [SD02]. In 2004, Simon Kozlov [KOZ04] has looked into many off the problems of the original article.

The idea in perspective shadow maps, is closely related to shadow maps, the only difference being that the shadow map is generated in normalized device coordinates i.e. after perspective transformation. That is, the scene is first transformed into the unit cube using the projection matrix of the camera. The shadow map is then generated by looking at this transformed scene from the light source, and this *perspective shadow map* is then used for determining shadow regions in the final image.

When generating the shadow map in post-perspective space, objects close to the camera will be larger than objects far from the camera. This means that more pixels in the shadow map is used for the objects closer to the camera, and thereby aliasing will be decreased or in some cases removed entirely.

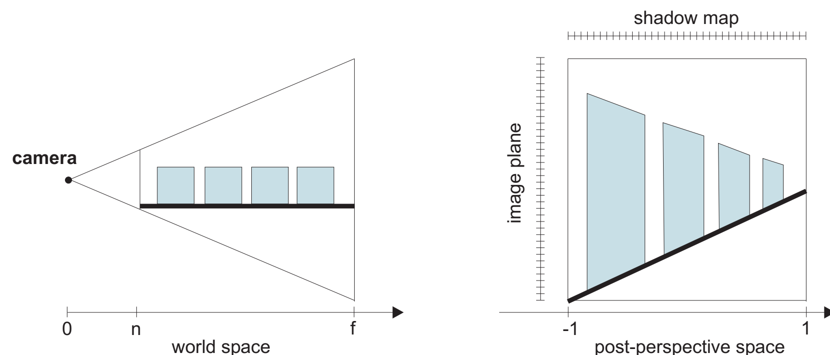


Figure 24: The scene seen before and after perspective projection. Objects close to the camera are larger than objects far from the camera in post perspective space. This means objects close to the camera occupies a larger area in the shadow map

On thing to keep in mind when doing the perspective transformation of the scene, is that light-sources can change both position and more problematic type. That is directional light-sources can become point light-sources and vice versa. In [SD02] 6 cases with different light types and positions are discussed, 3 of them involving directional light-sources and 3 of them involving point light-sources. The cases involving point light-sources is seen in figure 25. A point light in front of the viewer will remain a point light in front of the viewer(left). A point light behind the viewer will be transformed behind the infinity-plane and will be inverted(center),

and a point light on the plane through the view point will become a directional light(right). This is important to notice since it will affect the way the shadow map has to be generated.

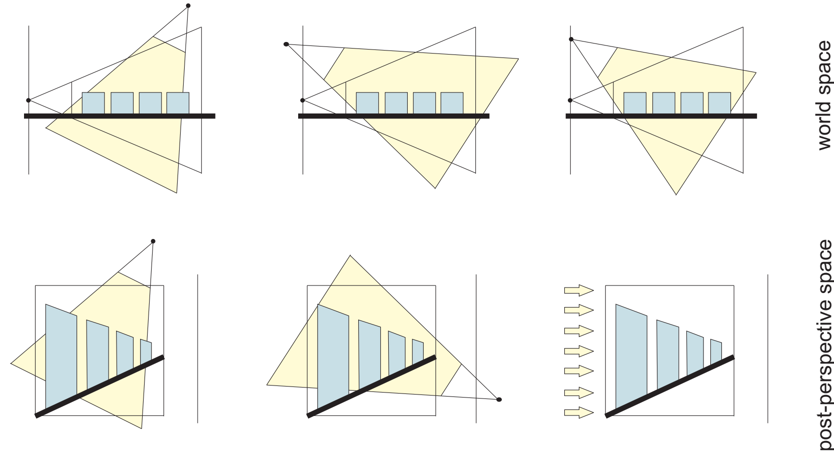


Figure 25: A point light in front of the viewer will remain a point light in front of the viewer(left). A point light behind the viewer will be transformed behind the infinity-plane and will be inverted(center), and a point light on the plane through the view point will become a directional light(right).

To sum up the original algorithm goes through the following steps:

1. Transform the scene using the projection matrix of a virtual camera. The frustum of this camera should cover all objects casting shadow in the final image. Done by doing a virtual shift of the original camera position.

2. Transform the light sources also using the above projection matrix. Notice the possible change in type of the light source

3. Setup a virtual camera at the new light position. The type of camera (orthographic or perspective) is defined by the type of light(directional or point light). The frustum should cover the unit cube, in which all objects, scene from the camera, are.

4. Render the scene to a texture (the shadow map).

5. Render the scene from the real camera, doing depth comparison with the depth map. Texture coordinates are generated using the projection matrix of the virtual camera in step 1 and the view and projection matrix of the transformed light source.

A problem discussed in the original article, is ensuring that all objects that could contribute to the shadows in the final image, are included in the shadow map. As known, all objects lying in the view frustum of the camera, will be

transformed into the unit cube, and will therefore contribute to the shadow map. However objects outside the view frustum could also contribute to the shadow in the final image, so these objects must also be included in the shadow map. An example of this is seen in figure 26.

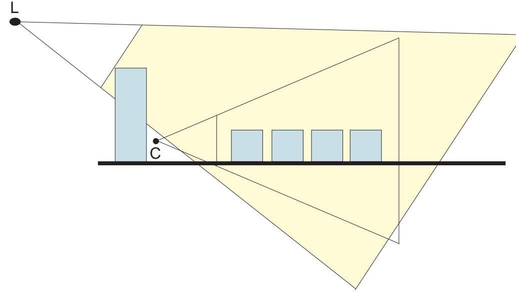


Figure 26: The object on the left will cast shadow into the scene. However this object will not be present in the shadow map

Here an object outside the view frustum casts shadow into the scene. This would not be accounted for in the current implementation. The original article suggests doing a 'virtual' shift of the camera, to ensure that all objects are included in the view frustum. That is, a virtual camera used for the perspective transformation is moved backwards until all shadow casting objects are included in the frustum of the camera. This however means that the shadow map must cover a larger area and therefore higher resolution must be used to obtain the desired results. Kozlov also points out that determining the actual shift-distance is a non-trivial operation that would require analyzing the scene for possible shadow casters. This operation would require CPU calculations which is not wanted.

Instead Kozlov suggests using another projection matrix for the transformed light:

$$\mathbf{P}_{l,b} = \begin{pmatrix} c & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 1 \\ 0 & 0 & \frac{1}{2}a & 0 \end{pmatrix}, \text{ where}$$

$$\begin{aligned} a &= |Z_{near}| = |Z_{far}| \\ c &= \frac{2Z_{near}}{r-l} \\ d &= \frac{2Z_{near}}{t-b} \end{aligned}$$

When an object is placed behind the camera in the scene, this object will be transformed to the other side of the infinity plane (see figure 27) after projection.

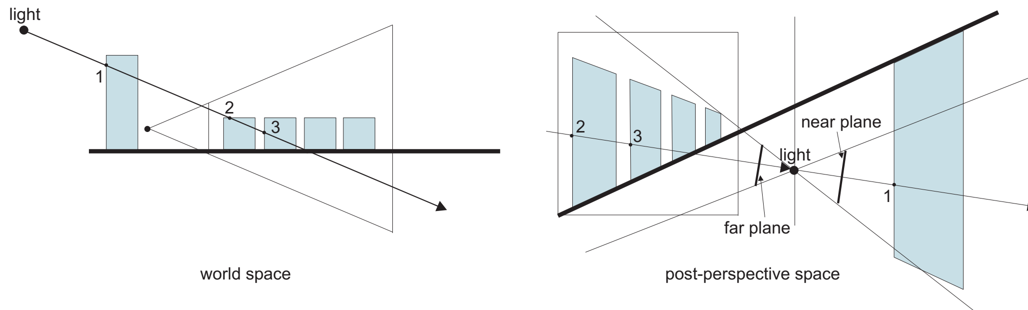


Figure 27: Object behind camera transformed behind infinity plane

In the real scene this object would be intersected first if we traced a ray from the light source, so this should still be the case in post-projective space. The matrix above actually lets us imitated this ray in post-projective space. Setting the near plane to a negative value $-a$, and the far plane to a positive value a, corresponds to tracing a ray, from the transformed light source towards infinity and then from minus infinity towards the light source. This preserves the order of intersections.

4.2 Implementation

As stated, the Perspective Shadow Map(PSM) in theory looks a lot like the Shadow Map algorithm. However there are many things to take into account when implementing PSM. In the following we will show the simple implementation, go over the issues in this implementation and try to solve the issues using methods described in [KOZ04]. It has been chosen, in contrast to the original paper, to focus the implementation on spotlights as opposed to directional lights. This choice is based on keeping the implementations simple and the fact that we deal with spotlights in the Shadow Map algorithm and in Chan's hybrid algorithm. It should of course be noted that all three of the algorithms also work with directional light sources, and comparing the algorithms when using directional light sources should definitely be a focus point in future work.

The pseudo-code for the PSM algorithm is seen below.

```
init_textures();

for each light
    transform scene
    create virtual camera at light source position
    render scene using virtual camera
    copy depth buffer to texture
end
for each light
    enable light(i)
    calculate texture coordinates
    enable depth testing
    draw scene using real camera
```

```
end
```

As some may have noticed, this looks a lot like the Shadow Map algorithm pseudo-code, the only change being the transformation of the scene. However implementation is not as trivial as it might seem, and the single steps will now be explained showing which problems arises during implementation.

4.2.1 Transforming The Scene

Transforming the scene into normalize device coordinates is done by multiplying all vertices by the Modelview- and Projetion-matrix of the current camera. That is we create a matrix $\mathbf{M}_c = \mathbf{P}_c \times \mathbf{MV}_c$, where $\mathbf{P}_c$ is the projection matrix and $\mathbf{MV}_c$ is the modelview matrix of the camera.

```
P_c = load_projection_matrix(cam);
MV_c = load_modelview_matrix(cam);
M_c = P_c * MV_c;
```

Multiplying with this matrix will move all vertices within the viewing frustum to the unit cube as seen in figure 24.

4.2.2 The Light 'Camera'

Since it is intended to transform the objects in the scene the light sources also have to be transformed and thereby the virtual camera used to create the shadow map. As earlier, the camera is defined by the 7 parameters being *position*, *direction*, *viewing angle*, *aspect ratio*, *up vector*, *near plane* and *far plane*. When we transform the scene, the position and direction of the camera are given by the transformed position and direction of the light source. The viewing angle on the other hand is hard to calculate, so this is set manually. In the first implementation the angle is set so that the frustum of the camera entirely covers the unit cube thereby encompassing all objects in the scene. This ensures that all objects receiving shadow in the scene is within the new viewing angle. The aspect ratio of the camera is, as earlier, set to 1 due to the nature of the spotlight.

The most interesting parameters in this stage is actually the near and far distances of the virtual camera. As stated earlier a special transformation matrix should be used to avoid troubles when the light source is positioned behind the camera. This special matrix is actually just the result of setting the near plane behind the camera and the far plane in front of the camera. That is, we set the near plane value to the negated near plane value, and the far plane to the near plane value.

Before doing this the near and far plane needs to be calculated. The most reasonable thing to do here would be to let the near and far plane just encompass the unit cube, as with the viewing angle. However, objects that are not visible in the scene might cast shadow into the scene. Using the suggested near and far planes could result in these objects being clipped and therefore not being represented in the shadow map. This would cause shadow to disappear in the scene. Instead we

calculate the near and far planes so that they encompass a transformed bounding box of the scene. This way objects outside the viewing frustum will still appear in the shadow map even though they will not be transformed into the unit cube.

This is guarantied even though the viewing angle was calculated without concern for these objects. This is explained as follows. For a point to be in shadow a line from the light source towards this point must hit a blocker. Since this line will, even after transformation, always hit the blocker and since the shadow receiving point is transformed into the unit cube, the blocker will always be represented in the shadow map, if it is not clipped by the near or far plane.

Now all parameters are set and the virtual camera should be able to see all objects that the original light source could see.

```
light_cam.position = M_c * light.position;
light_cam.direction = M_c * light.direction;
light_cam.calculate angle(unit cube);
light_cam.aspect_ratio = 1;
light_cam.up = (0,1,0);
light_cam.alter_up();
light_cam.calculate_near_and_far(bbox_all);
if light behind camera
    light_cam.far = light_cam.near;
    light_cam.near = -light_cam.near;
end
```

Having set all the parameters of the virtual camera we can now draw the scene as usual except for multiplying M_c on the current modelview matrix.

```
rt->beginCapture();
Viewport(0,0,map_size,map_size);
load_projection(light_cam);
load_modelview(light_cam);
disable(LIGHTING);
ShadeModel(FLAT);
CullFace(BACK);
MultMatrix(M_c);
draw_scene();
rt->EndCapture();
```

The result is seen in figure 28. In (a) the scene is rendered as seen from the camera. In (b) the scene is seen from the light source. This would be used for shadow map in the Shadow Map algorithm. In (c) the transformed scene is seen from the transformed light source. Notice how the cylinder that is close to the camera is larger than the box that is far away from the camera. This is the view used for the shadow map.

4.2.3 Calculating Texture Coordinates

After having created the shadow map using the transformed scene, we must set up texture coordinate generation taking this transformation into account. As in the Shadow Map algorithm the texture matrix is computed using the projection- and modelview-matrix of the light source. This time however, we use the matrices for

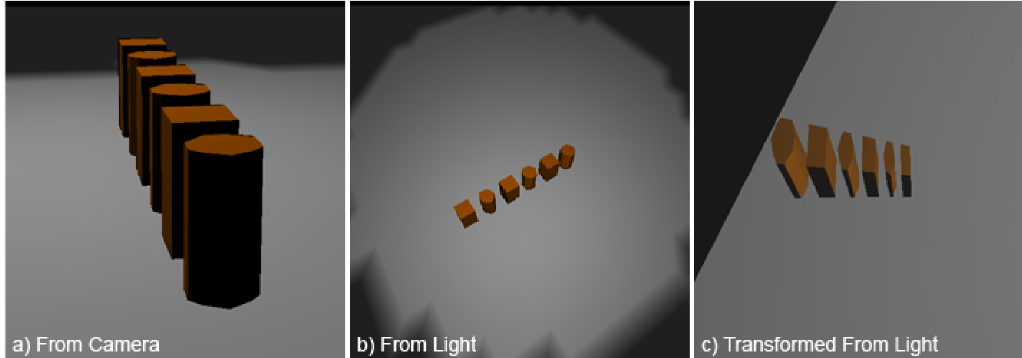


Figure 28: Using transformed scene. In (a) the scene is rendered from the camera, in (b) from the light source and in (c) the transformed scene is rendered from the transformed light source. Objects close to the camera becomes larger than objects far from the camera after transformation

the transformed light source. We also multiply with the transformation matrix to get into light space.

$$\mathbf{T} = \mathbf{B} \times \mathbf{P}_{lt} \times \mathbf{V}_{lt} \times \mathbf{M}_c \times \mathbf{V}_c^{-1}$$

where $\mathbf{B}$ is the bias-matrix from section 2.1.1, $\mathbf{P}_{lt}$ is the projection matrix of the transformed light source, $\mathbf{V}_{lt}$ is the viewing matrix of the transformed light source, $\mathbf{M}_c$ is the transformation matrix described above and $\mathbf{V}_c$ is the viewing matrix of the camera. Using OpenGL's ability to automatically multiply the inverse of the current modelview matrix, by using EYE_LINEAR as TEXTURE_GEN_MODE, the scene can now be lit using the transformed shadow map.

```
texture_matrix = bias * get_projection_matrix(light) * get_modelview_matrix(
    light) * M_c;
load_projection(cam);
load_modelview(cam);
TexGeni(S, TEXTURE_GEN_MODE, EYE_LINEAR);
TexGenfv(S, EYE_PLANE, texture_matrix(row 0));
TexGenfv(T, EYE_PLANE, texture_matrix(row 1));
TexGenfv(R, EYE_PLANE, texture_matrix(row 2));
TexGenfv(Q, EYE_PLANE, texture_matrix(row 3));
Enable(TEXTURE_GEN_S);
Enable(TEXTURE_GEN_T);
Enable(TEXTURE_GEN_R);
Enable(TEXTURE_GEN_Q);
enable_shadow_comparison();
enable(LIGHTING);
draw_scene();
```

4.2.4 Results so far

In figure 29 a scene is rendered using (a) Perspective Shadow Maps and (b) Ordinary Shadow Maps. As it can be seen the result of the transformation is as we

expected. The shadows close to the camera are better in the PSM algorithm and as the distance increases the shadows degrades towards the ones of the ordinary shadow map algorithm.

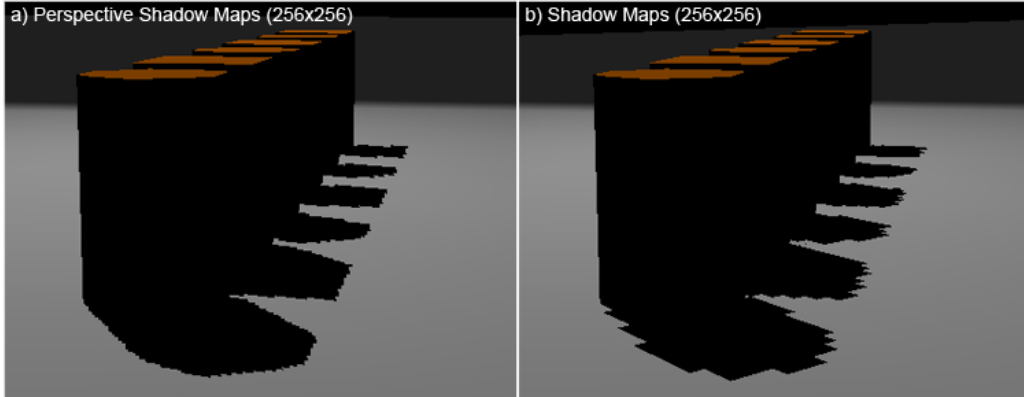


Figure 29: Comparing Perspective Shadow Maps to ordinary Shadow Maps. When using Perspective Shadow Maps (a) the shadows close to the camera are better due to the objects being larger in the shadow map. As the distance increases objects will be smaller in the shadow map and the quality of the shadow degrades

4.2.5 Adjusting Real Camera

Although the image in figure 29 seems to show that our current implementation of PSM's are good, there a multiple cases in which things go wrong.

Up till now the scene has been transformed using the current matrices for the camera. But what happens if the near and far planes of this camera are not taken into account? Well let's look at an example where the near and far planes of the camera are set to some arbitrary values just to ensure we can see everything in the scene. If the scene is transformed using arbitrary near and far plane settings, the actual objects in the scene might occupy a very small area of the unit cube. This would result in a shadow map in which the objects are very small. Therefore, the camera matrices used for the transformation should be the matrices of a new virtual camera in which the near and far planes are set, just to encompass the objects of interest. These objects must be the objects that both the light and the camera can see. That is, we create a bounding box containing the objects that both the light source and the camera can see, and then we set the near and far plane of the new camera, so that they just encompass this bounding box.

```
for each object
    if object in light frustum
        if object in cam frustum
            add to bbox_view
        end
    end
end
```

```
virt_cam->set_near_and_far(bbox_view);
```

Doing this ensures that the quality of the shadows are not dependant on the near and far planes of the camera specified by the user, which could lead to very bad results.

In the original paper the points of interest is calculated using the computational geometry library CGAL(www.cgal.org). With this the area of interest is defined as $H = M \cap S \cap L$ where $M = \text{convex hull}(l \cup V)$ (see figure 30). The area is shown in yellow. Using the example from the article it is seen however, that our simple calculations results in almost the same area. This is seen in figure 31. The reason for not using this library has mostly been due to technical difficulties getting it to function. This is of course not optimal, but due to the time limits regarding the delivery of this thesis abandoning the idea of using the library seemed the only possibility.

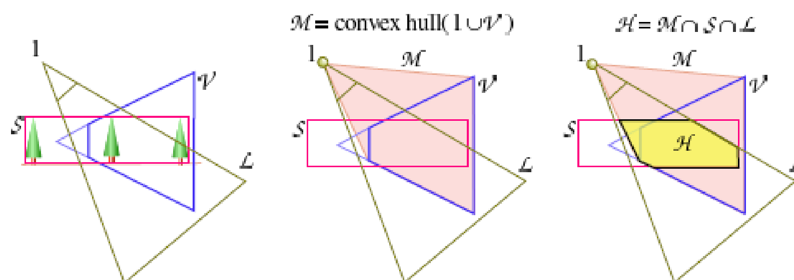


Figure 30: Points of interest calculated in original article

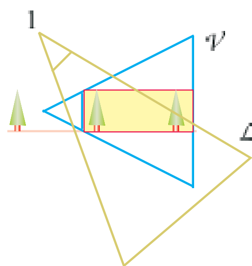


Figure 31: Points of interest calculated in this thesis

4.2.6 Cube Clipping

Another problem arises if the light source is within the camera frustum, that is, the light can be seen from the camera, the transformed light source will itself be

positioned within the unit cube when transformed into normalized device coordinates. Up till now the viewing angle and the near and far planes of the virtual camera has been calculated using the unit cube, but this is not ideal when the camera is itself positioned inside the cube. How should the angle be specified? A very large angle should maybe be used, resulting in very low shadow map quality. And what about the near plane? Should that be negative?

Instead of using the unit cube as the area which the virtual camera should see, the frustum could be calculated using the objects in the scene. To do this we start of by creating a bounding box in world space, which covers all the points/objects that is within the light source frustum. This bounding box must be represented by eight corner points, since the transformation into normalized device coordinates's most likely distort the bounding box. We also create a bounding box that contains points that both the light and the real camera can see. But why do we do this? Well, the viewing angle of the virtual camera, created at the transformed light source position, should be chosen such that it covers only points that can be seen by both the real camera and the light source. This is indeed all points that can receive shadow, and they should therefore be in the shadow map. The near and the far plane on the other hand should be chosen so that all objects that can cast shadow into the scene are covered. A quick classification is to choose all point that the light can see. Doing these calculations ensures that the frustum of our virtual camera is as small as possible resulting in the best possible shadow map.

```
for each mesh
    if mesh.bbox in light frustum
        add to bbox_light
        if mesh.bbox in cam frustum
            add to bbox_view
        end
    end
end
```

For the floor each vertex is tested whether or not to be inside the frustums.

```
for each vertex
    if vertex in light frustum
        add to bbox_light
        if vertex in cam frustum
            add to bbox_view
        end
    end
end
```

The result of using cube-clipping is seen in figure 32. When the light source is far away from the camera frustum (a and b), the transformed light source is far away from the unit cube after transformation, and the quality improvement is very small. However as the light source gets closer to the camera frustum (c and d) it will also get nearer to the unit cube after transformation and the angle of the virtual camera will increase dramatically for the camera to hold the entire unit cube. Using cube-clipping (d) the angle does not have to be that large and the quality of the shadow map and the resulting shadow, is much better.

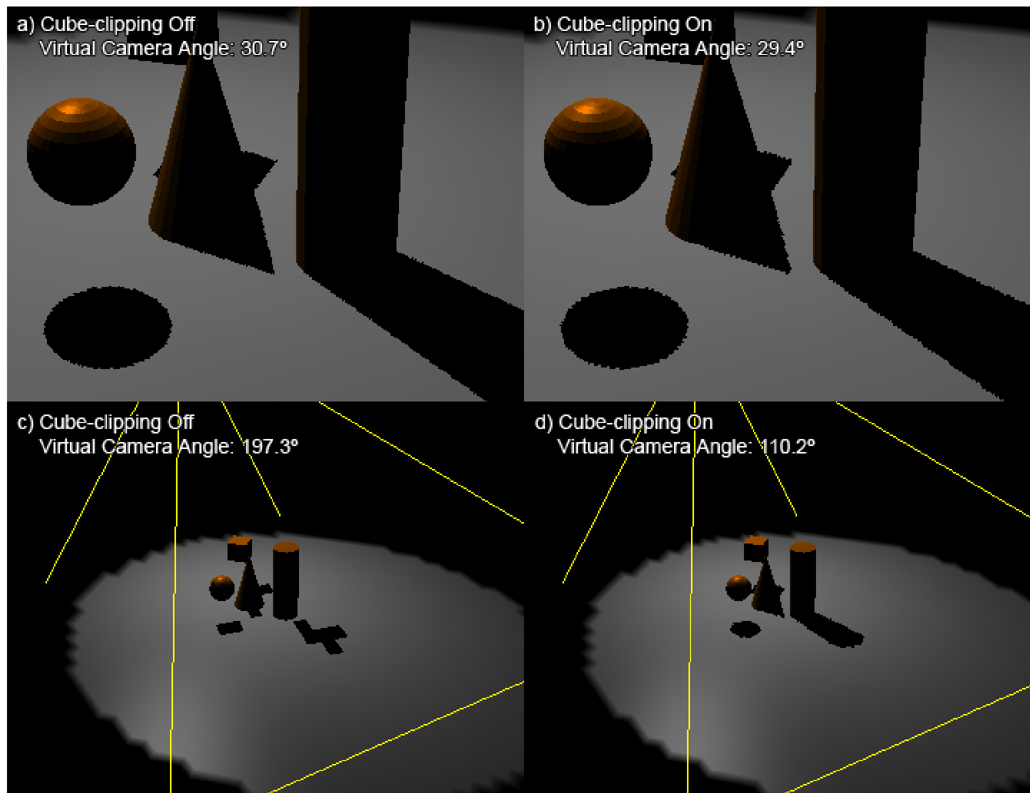


Figure 32: Using Cube-Clipping: When the light source (shown in yellow) is far away from the camera frustum (a and b) there is not much difference in the original approach (a) and using cube-clipping (b). However, as the light source gets nearer to the camera frustum (c and d) the difference in shadow quality in the original approach (c) and using cube-clipping (d) is very noticeable.

5 Chan's Hybrid Algorithm

5.1 Theory

At the *Proceedings of the Eurographics Symposium on Rendering 2004* Eric Chan and Frédo Durand presented what they called "An Efficient Hybrid Shadow Rendering Algorithm" [CD04]. The idea behind this algorithm, is to combine shadow maps and shadow volumes, using the good properties of the two. As it has been shown earlier shadow maps are efficient and flexible but prone to aliasing at shadow edges. Shadow volumes however produce perfect shadow edges but are fairly slow due to high fillrate-consumption. Chan and Durand suggests combining the two algorithms, using shadow volumes to determine shadows only at shadow edges also referred to as shadow silhouette pixels, and shadow maps every where else, thereby trying to achieve a fast algorithm that creates perfect hard shadows.

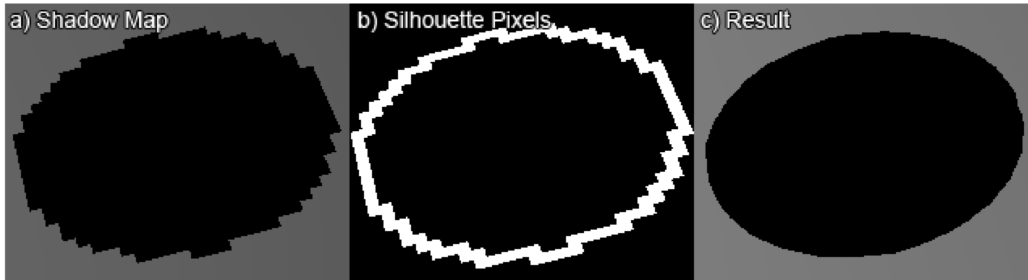


Figure 33: Overview of Chan's Hybrid algorithm. Using Shadow Maps (a) silhouette pixels are classified (b). The Silhouette pixels are shown in white. The scene is then rendered (c) using shadow volumes only at these silhouette pixels and using shadow map elsewhere

The algorithm goes through the following steps. First an ordinary shadow map is created. Since shadow volumes are later used for the edges of the shadows, the resolution of this shadow map can be fairly low. Only demand to the resolution is, that all objects must be present in the map, that is the resolution must be high enough to ensure that no objects are too small to be represented in the map.

Now, shadow silhouette pixels can be identified as follows. The scene is rendered from the camera, transforming all samples to light space, as in the normal shadow map algorithm. The samples are now compared to the four nearest samples in the shadow map, using *percentage closer filtering*. If all four values are equal, that is, the value returned by the comparison are either 0 or 1, all four samples are either in shadow or not, and the pixel is *not* a silhouette pixel. On the other hand, if the four values are not equal, the pixel must be a silhouette pixel. This way all pixels can be classified as being silhouette pixels or not (see figure 34).

Having found all silhouette pixels, the shadow volume algorithm can be used to determine the shadows at the edges. The advantage here is that shadow volumes are only rendered at shadow silhouette pixels, instead of everywhere in the scene.

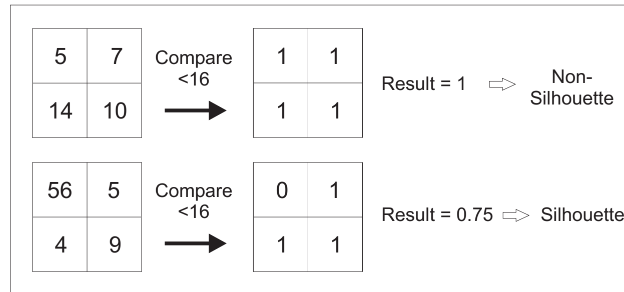


Figure 34: Illustrating silhouette contra nonsilhouette pixels

This minimizes the fillrate-consumption of the shadow volume algorithm. Shadows everywhere else are determined using the ordinary shadow map.

5.2 Implementation

The pseudo-code for the hybrid algorithm(Chan) is shown below. For a full view of the implementation see appendix C.4.

```
// Create shadow map
virtual camera at light position
draw scene using virtual camera
copy depth values to texture

// Identify Silhouette pixels
render scene comparing depth values.
if(silhouette)
    pixel = white
else
    pixel = black
end
copy to texture

// Create Computation mask
draw screen quad
if(non-silhouette)
    depth = 0;
end

// Render
if(depth = 0)
    render using shadow volumes
else
    render using shadow maps
end
```

5.2.1 Create Shadow Map

Creating the shadow map is done exactly as in the Shadow Map implementation using the RenderTexture class. However since we only need the shadow map to be

able to identify silhouette pixels the resolution of the shadow map can be very low.

```
rt->BeginCapture();
Enable(DEPTH_TEST);
Enable(CULL_FACE);
CullFace(BACK);
ShadeModel(FLAT);
ColorMask(0,0,0,0);
Disable(LIGHTING);
Viewport(0,0,map_size,map_size);
load_projection(light_cam);
load_modelview(light_cam);
draw_scene();
rt->EndCapture();
```

5.2.2 Identifying Silhouette Pixels

As stated we need to identify silhouette pixels to determine which pixels to render using shadow volumes and which to render using shadow maps. To do this we make use of the CG[6] programs seen in figure 35 and figure 36. For these to work as intended we have to make sure that PCF is enabled. Doing this allows us to identify silhouette pixels by looking at the value returned when looking in the shadow map. If the value does not equal either one or zero the pixel must in the near proximity of a shadow boundary and is defined as a silhouette pixel. These pixels are colored white, while the rest are colored black.

The scene is then copied to a texture used for creating the so-called computation mask. The result can be seen in figure 37. The boundary of the shadow area is classified as silhouette pixels. Notice how a lot of pixels on the ball are also classified as silhouette pixels. This is due to the earlier mentioned depth buffer problems. However, looking at the figure one can realize that classifying these like this actually is a good thing, since shadow volumes will then be used for rendering the scene where biasing problems would otherwise occur.

```
Clear(COLOR_BUFFER_BIT | DEPTH_BUFFER_BIT);
rt->BindDepth();
TexParameteri(rt->getTextureTarget(), TEXTURE_MIN_FILTER, LINEAR);
TexParameteri(rt->getTextureTarget(), TEXTURE_MAG_FILTER, LINEAR);
load_projection(cam);
load_modelview(cam);
set_cg_parameters();
calculate_texture_matrix();

enable_cg_programs()
disable(LIGHTING);
draw_scene();
disable_cg_programs();

Enable(TEXTURE_RECTANGLE_NV);
BindTexture(TEXTURE_RECTANGLE_NV, silTexture);
CopyTexImage2D(TEXTURE_RECTANGLE_NV, 0, RGB, 0, 0,vpWidth,vpHeight, 0);
```

[6]NVIDIA "C for Graphics", http://developer.nvidia.com/page/cg_main.html

```
void main (in    float4 in_col    : COLOR0,
           out float4 color       : COLOR0,
           out float4 lClip       : TEXCOORD0,
           out float4 pClip       : POSITION,
           out float  diffuse     : TEXCOORD1,
               float4 pObj        : POSITION,
               float3 nObj        : NORMAL,
           uniform float4x4 mvp,
           uniform float4x4 mv,
           uniform float4x4 mvIT,
           uniform float4x4 observerEyeToLightClip,
           uniform float4 lightPosEye)
{
    pClip = mul(mvp, pObj);
    float3 nEye = mul(mvIT, float4(nObj, 0)).xyz;
    float4 pEye = mul(mv, pObj);
    lClip = mul(observerEyeToLightClip, pEye);
    diffuse = dot(normalize(nEye), normalize(lightPosEye.xyz - pEye.xyz));
    color = in_col;
}
```

Figure 35: CG vertex program for finding silhouette pixels

```
void main (out half4 color              : COLOR,
           float4 uvShadowMap           : TEXCOORD0,
           half diffuse                 : TEXCOORD1,
           uniform sampler2D shadowMap  : TEX0)
{
    fixed v = tex2Dproj(shadowMap, uvShadowMap).x;
    color = (v > 0 && v < 1) ? 1 : 0;
}
```

Figure 36: CG fragment program for finding silhouette pixels

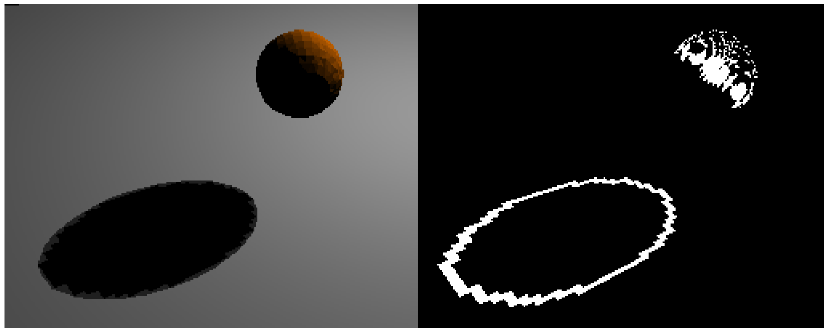


Figure 37: Classifying silhouette pixels. Pixels that lie in the proximity of a shadow boundary are classified as silhouette pixels and are colored white. Other pixels are colored black

5.2.3 Creating Computation Mask

Having classified silhouette pixels in the scene, the next step is to create a computation mask. That is, a device that makes it possible to specify which pixels in the

picture to process and which not to. This should increase the speed by minimizing the pixels in which we process shadow volumes reducing fillrate consumption.

The computation mask is created by drawing a screen aligned quad. Using the fragment program in figure 38 we can set the depth value of all non-silhouette pixels equal to zero, and leave the depth value of all other pixels. Afterwards the extension EXT_depth_bounds_test[EDBT02] is used to discard pixels with depth zero thereby only processing silhouette pixels.

```
Enable(DEPTH_TEST);
DepthFunc(ALWAYS);
set_cg_parameters();
enable_cg_program();
ColorMask(0,0,0,0);
draw_screen_quad();
disable_cg_program();
ColorMask(1,1,1,1);
```

```
void main (out float4 color      : COLOR,
           out float depth       : DEPTH,
           uniform samplerRECT texColorMask,
           float4 wpos           : WPOS)
{

    if ( texRECT(texColorMask, wpos.xy).x > 0){
        discard;
    }else{
        color = float4(1,0,0,1);
        depth = 0.0;
    }
}
```

Figure 38: CG fragment program creating computation mask

The result of this is a depth buffer in which non-silhouette pixels have depth zero and silhouette pixels have the depth value of the scene.

At this point one could wonder why we first create a texture specifying which pixels are silhouette pixels, and the use this texture to set depth values, instead of just setting depth values immediately. This however, is not possible since there in the current implementation is no control of the order in which polygons are drawn. The aim is to set depth value of silhouette pixels to zero. If this is done during the drawing of the scene, some pixels could falsely pass the depth test if they have already been classified as silhouette pixels when drawing objects farther away. Thereby the depth values obtained will not be the correct depth values of the scene and wrong pixels are classified as silhouette pixels.

5.2.4 Rendering Shadow Volumes

The EXT_depth_bounds_test extension adds the ability to discard pixels based on the depth buffer value already saved at that pixels position. This is done by specifying depth bounds within which pixels are processed. Specifying the depth

bounds to $[\epsilon, 1]$, where ϵ is some small constant $\epsilon \approx 0.001$, enables us to process only silhouette pixels, when drawing shadow volumes. The increase in speed however is depending on the hardware's ability to preserve early z-culling[7]. This feature is only available on NVIDIA GeForce 6 series and up, and on ATI Radeon 9700 and up. However ATI does not support the EXT_depth_bounds_test extension.

Being able to discard all but silhouette pixels we can now render shadow volumes as described earlier, but now the time required will be lower, if the program is fillrate limited.

```
Enable(DEPTH_BOUNDS_TEST_EXT);
DepthBoundsEXT(0.001, 1.0);
render_shadow_volumes();
```

In figure 39 the difference when drawing shadow volumes using the Shadow Volumes algorithm and Chan's Hybrid algorithm is shown. In figure 39(b) the shadow volumes, shown in yellow, are covering most of the screen, while in figure 39(c) the pixels where shadow volumes are processed occupies only a small amount of screen-space. In a fill-limited application this will result in increased performance.

A little change has been done to the step of rendering using shadow volumes here. In the original approach the scene was first rendered to obtain the depth values. In this step lighting was disabled and flat shading was used to increase speed. This time, the rendering of the scene can actually also be used in the following step, so this time ambient lighting and smooth shading are enabled.

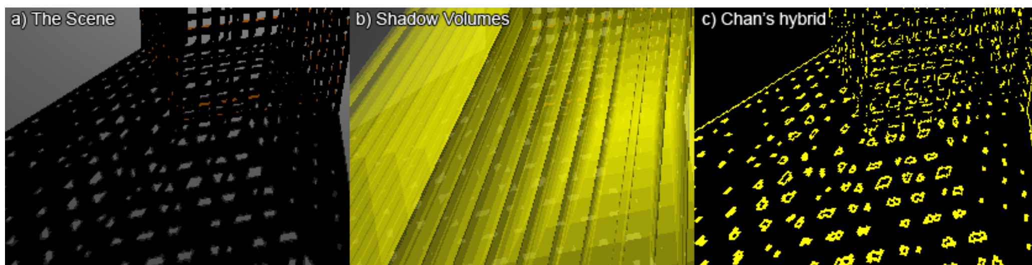


Figure 39: Comparing shadow volumes pixels. Shadow Volumes cover most of the screen when using the ordinary Shadow Volumes algorithm(b). Using Chan's hybrid (c) only a small amount of pixels is processed when rendering shadow volumes

5.2.5 Final Shading

Having determined which of the silhouette pixels are to be lit and which are in shadow, it is time to do the final shading. Realizing that the stencil buffer has zero values at lit silhouette pixels and non-silhouette pixels, we can simply render the scene at the pixels where the stencil buffer is zero. The rest of the pixels where

[7]Early Z-Culling means that a depth test is performed before texturing and shading effects. This allows a pixel to be rejected early in the pipeline not wasting time on shading

shaded in the last step where, as described, the scene was rendered using ambient colors. The final shading is done using the CG programs in figure 40 and figure 41.

```
void main (in   float4 in_col  : COLOR0,
            out float4 color   : COLOR0,
            out float4 lClip   : TEXCOORD0,
            out float4 pClip   : POSITION,
            out float diffuse  : TEXCOORD1,
                float4 pObj    : POSITION,
                float3 nObj    : NORMAL,
            uniform float4x4 mvp,
            uniform float4x4 mv,
            uniform float4x4 mvIT,
            uniform float4x4 observerEyeToLightClip,
            uniform float4 lightPos)
{
    pClip = mul(mvp, pObj);
    float3 nEye = mul(mvIT, float4(nObj, 0)).xyz;
    float4 pEye = mul(mv, pObj);
    lClip = mul(observerEyeToLightClip, pEye);
    diffuse = dot(normalize(nObj), normalize(lightPos.xyz - pObj.xyz));
    color = in_col;
}
```

Figure 40: CG vertex program for shadowmap lookup

```
void main (in   float4 in_color     : COLOR0,
            out float4 color        : COLOR0,
                float4 uvShadowMap  : TEXCOORD0,
                float diffuse       : TEXCOORD1,
            uniform float nlights,
            uniform sampler2D shadowMap)
{
    fixed v = tex2Dproj(shadowMap, uvShadowMap).x;
    color = (v > 0) ? diffuse * in_color / nlights : float4(0.0,0.0,0.0,1.0)
        ;
}
```

Figure 41: CG fragment program for shadowmap lookup

At pixels where the stencil buffer is zero the CG programs use the shadow map to shade the scene. If the shadow map value is greater than zero the pixel is lit otherwise in shadow. The reason for doing the final shading in a CG program instead of using standard OpenGL is found in the way we can use the shadow map in standard OpenGL. Comparing depth values results in INTENSITY, ALPHA or LUMINANCE values returned from the comparison. If INTENSITY is used as done in the Shadow Map algorithm, the jagged edges obtained when doing normal shadow maps will also be present now. Using the ALPHA return value should then seem reasonable. This could allow an alpha test passing only when the comparison is greater than zero. This however, results in situations with wrong shading since fragments that are in shadow could be rejected. When this happens, other points

in the scene that should be hidden can then be rendered. The two cases are shown in figure 42.

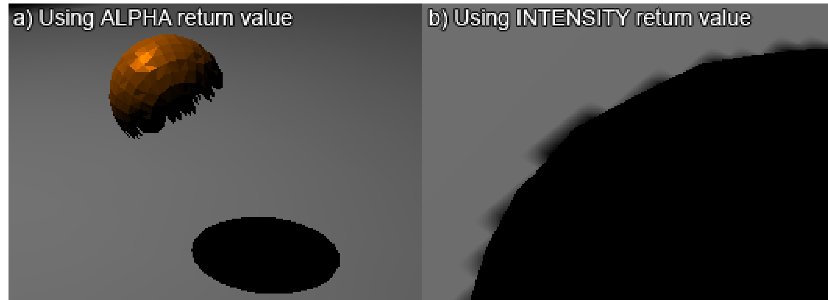


Figure 42: Using ALPHA and INTENSITY as return values. When using ALPHA(a) some points are falsely shaded due to failed alpha test. Using INTENSITY(b) results in artifacts at shadow boundary

To get the correct result the shader does the shadow map lookup and shades the pixel according to the result. If the result is greater than zero, the pixel should be lit, and if it is equal to zero the pixel is shaded black. Doing two different things based on the shadow map lookup and alpha testing would not be able in one pass without a shader program. Off course one could do two passes, using alpha testing, first rendering lit pixels and then shadowed pixels, but this would decrease the speed of the algorithm.

```
load_projection(cam);
load_modelview(cam);

calculate_texture_matrix();
set_cg_parameters();
enable_cg_programs();

glDepthMask(TRUE);
glClear(DEPTH_BUFFER_BIT);
glEnable(DEPTH_TEST);
glDepthFunc(LEQUAL);
glColorMask(1,1,1,1);
glStencilFunc(EQUAL, 0, ~0);
glStencilOp(KEEP,KEEP,KEEP);

glCallList(dlDrawScene);

disable_cg_programs();
```

The final result using a simple shader is seen in figure 43.

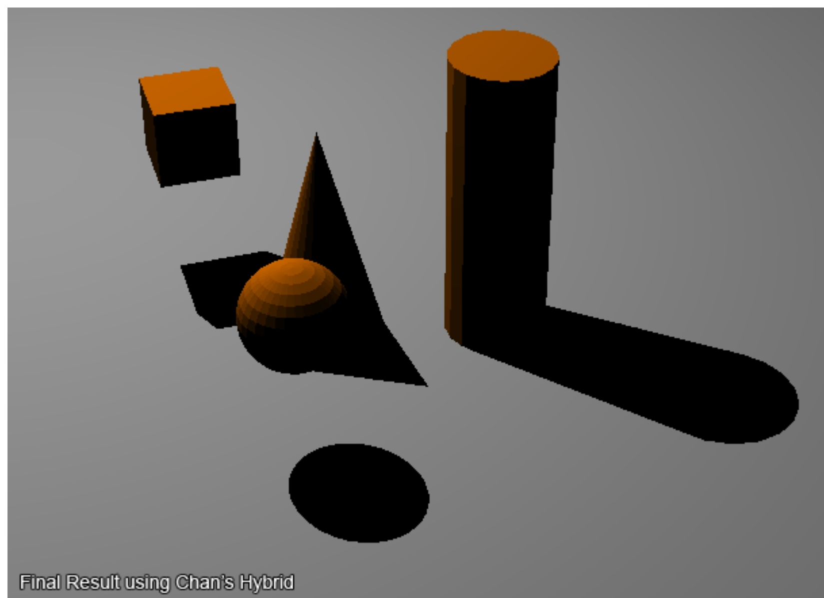


Figure 43: The final rendering using Chan's Hybrid algorithm with a mapsize of 512x512. The edges of the shadow area are perfect just as when using the Shadow Volumes algorithm. Also acne as result of limited precision and resolution issues are avoided, since shadow volumes are used to render pixels that would normally have acne when using the Shadow Map algorithm.

6 The Application

To be able to compare the four algorithms discussed earlier, an application using the four algorithms have been created. While implementing a total game-engine has not been the purpose of this project, the program allows for testing different scenes changing the shadow algorithm used dynamically during runtime. The application is using the OpenGL API and GLUT, the language being C++. In the program some classes from the course ”02561 Computer Graphics” at DTU is used[8].

A UML-diagram showing the structure of the program can be seen in figure 44. In this diagram only the most important classes are shown.

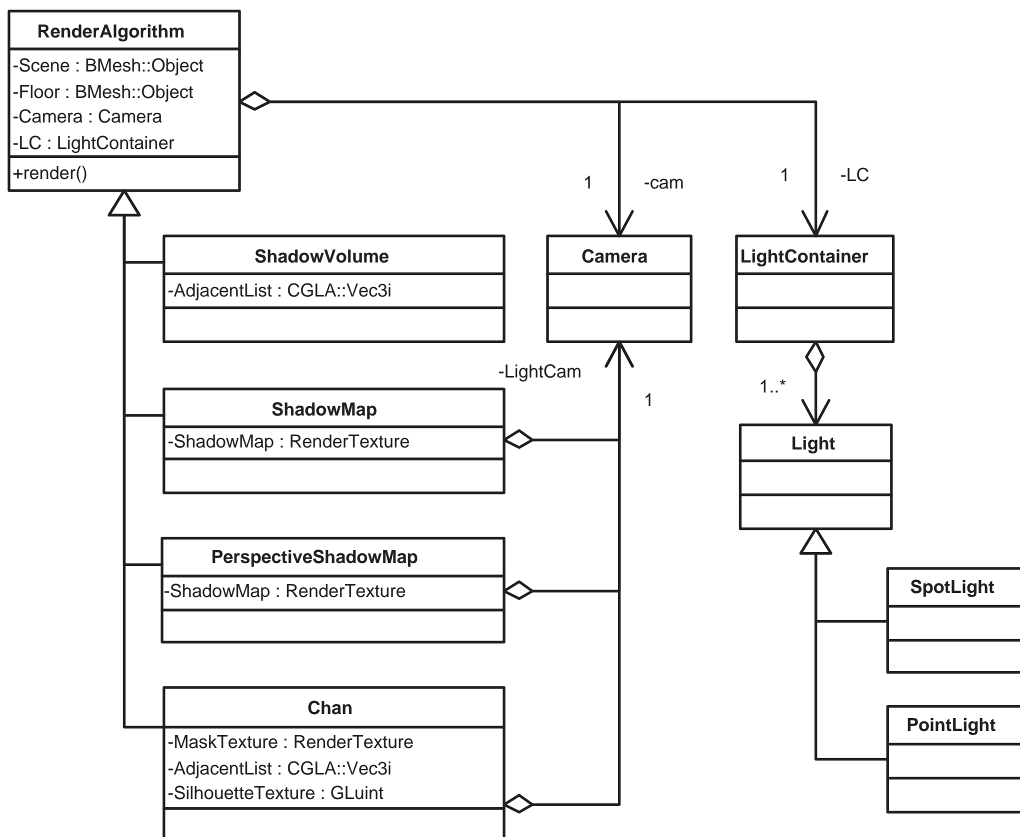


Figure 44: UML Class Diagram

The implementations of the four algorithms are, as seen, all depending on the top RenderAlgorithm class. This class controls properties common for the four algorithms e.g. camera, light sources and scene properties. The class instantiates

[8]BMesh::Object - A class containing the geometry and coloring of the scene
CGLA - A collection of classes containing common vector and matrix functions (*not* CGAL)

one Camera, one LightContainer, the Scene and the Floor(or ground). The floor is instantiated as an object for itself because of the way Perspective Shadow Maps calculates bounding boxes.

The four algorithms also know about eachother. This is to allow algorithms to use functions from other algorithms. This could be the Chan class using a DrawShadowVolumes function in the ShadowVolume class or the PerspectiveShadowMap using a function CreateShadowMap from the ShadowMap class.

The RenderAlgortihm class assumes that a window for rendering has already been created and that OpenGL is used.

The main program file should, to be able to use the RenderAlgorithm class, follow the steps below:

1. Load the scene into an object of type BMesh::Object*
2. Load the floor into an object of type BMesh::Object*
3. Ensure that the faces in the object has colors
4. Create a Camera
5. Add lights to a light container.
6. Initialize a window (Done using GLUT)
7. Instantiate the algorithms wanted.
8. In the main loop call RenderAlgortihm−>render()

Instantiating the RenderAlgortihm class is done with the call:

```
RenderAlgorithm RA = new ShadowMap(BMesh::Object* scene, LightContainer* LC
    , Camera* cam, BMesh::Object* floor);
```

The above is of course if for example the Shadow Map algorithm is to be used. Multiple instances of the RenderAlgortihm class can then be used to switch between different algorithms.

Passing pointers as arguments to the RenderAlgorithm class allows the user to use their own functions for controlling cameras and light sources. Still, basic functions are a part of the class such as rotate_cam() etc.

Switching algorithm is done using a menu in the program. Here the different algorithms can be chosen as well as a predefined set of scenes. A screen-shot of the program is seen in figure 45

With regard to the four algorithms settings such as Shadow Map size etc can also be changed during runtime. This is done using the keyboard. A full list of keyboard shortcuts can be seen in appendix B. Movement of the camera is controlled using the mouse. Pressing the left button, moving the mouse, rotates the camera. Middle button zooms in and out.

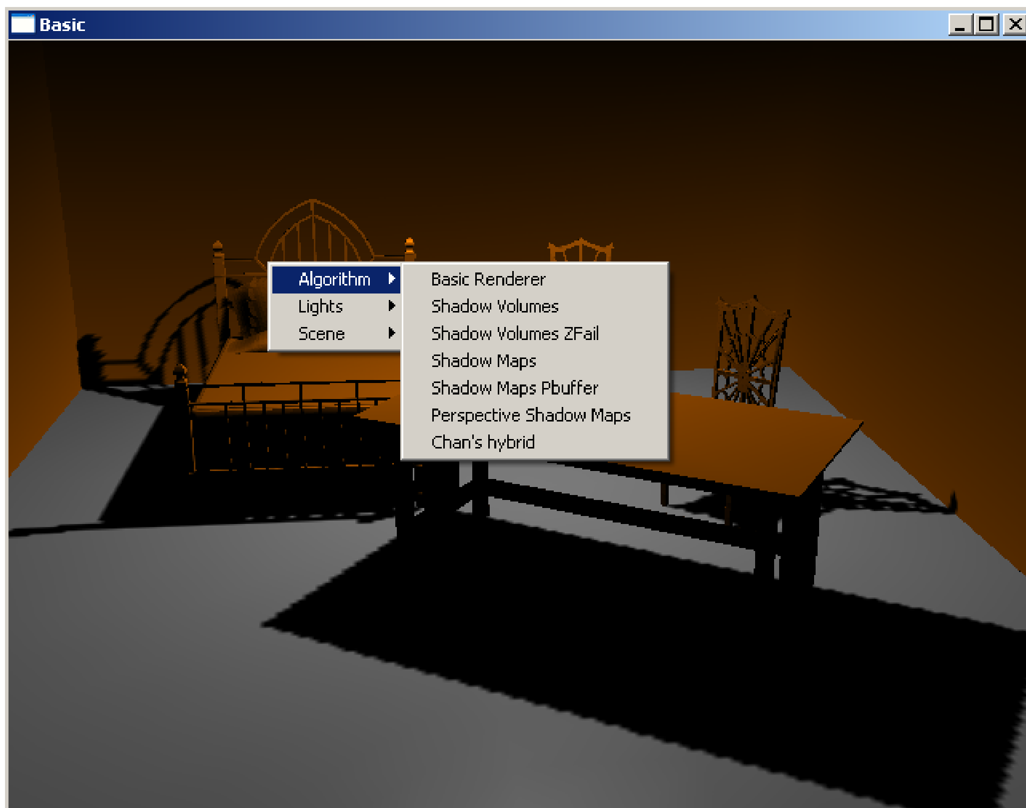


Figure 45: A screen-shot of the program. Using the menu different algorithms and scenes can be chosen during runtime

7 Results

Analyzing the results using the four algorithms is done using the scenes in figure 46[9]. Each of these scenes have properties that should allow the different algorithm to show their potential. Below, the properties of the scenes are listed. For all scenes a floor consisting of 20000 triangles are used.

BoxGrid The BoxGrid Scene(figure 46a) consists of 12000 triangles arranged to form a 3D grid of boxes casting shadow into the scene. This results in a large number of shadow volumes and should therefore show the advantages of using Chan's Hybrid Algorithm in such cases. To maximize the effect of using Chan's algorithm the camera is focuses on the shadow area cast by the boxes. An overview of the scene is shown in the right corner of figure 46a.

Rocks The Rocks Scene (figure 46b) consists of 63000 triangles forming five copies of a *StoneHenge like* object. In the scene three of these objects are scene with the light source positioned on the right side. In this scene Perspective Shadow Maps should increase the quality of the shadows near to the camera.

Street The Street Scene (figure 46c) consists of only 400 triangles arranged to form a number of objects to resemble buildings on either side of a street. With the low number of polygons Shadow Volumes should excel in both quality and speed.

Room The Room Scene (figure 46d) consists of 19000 triangles resembling an indoor scene. With both high detail shadows and large shadowed areas it could give a view of picture quality as well as speed when using indoor scenes.

Water Tower The Water Tower Scene (figure 46e) consists of 12500 triangles resembling a water tower. Again both high detailed shadows and larger shadow areas are present, and could give a view of both quality and speed for outdoor scenes.

All rendering is done on an Intel Pentium 4, 3.4GHz with 1GB RAM, graphics card NVidia GeForce 6600 GT - 128MB RAM.

7.1 Timing

In figure 47 a timing diagram for the five test-scenes using the four algorithms is shown. In the diagram the time used on the different steps in the algorithms is shown. All scenes were rendered at a resolution of 1280x1024 using shadow maps of size 1024x1024 where needed.

Not surprisingly the SM and the PSM algorithms are the fastest for all scenes, except the Street scene in which SV's actually out-perform PSM's because of the

[9] All scenes in figure 46 are rendered using Shadow Volumes. See appendix A.1 for images rendered using the other algorithms

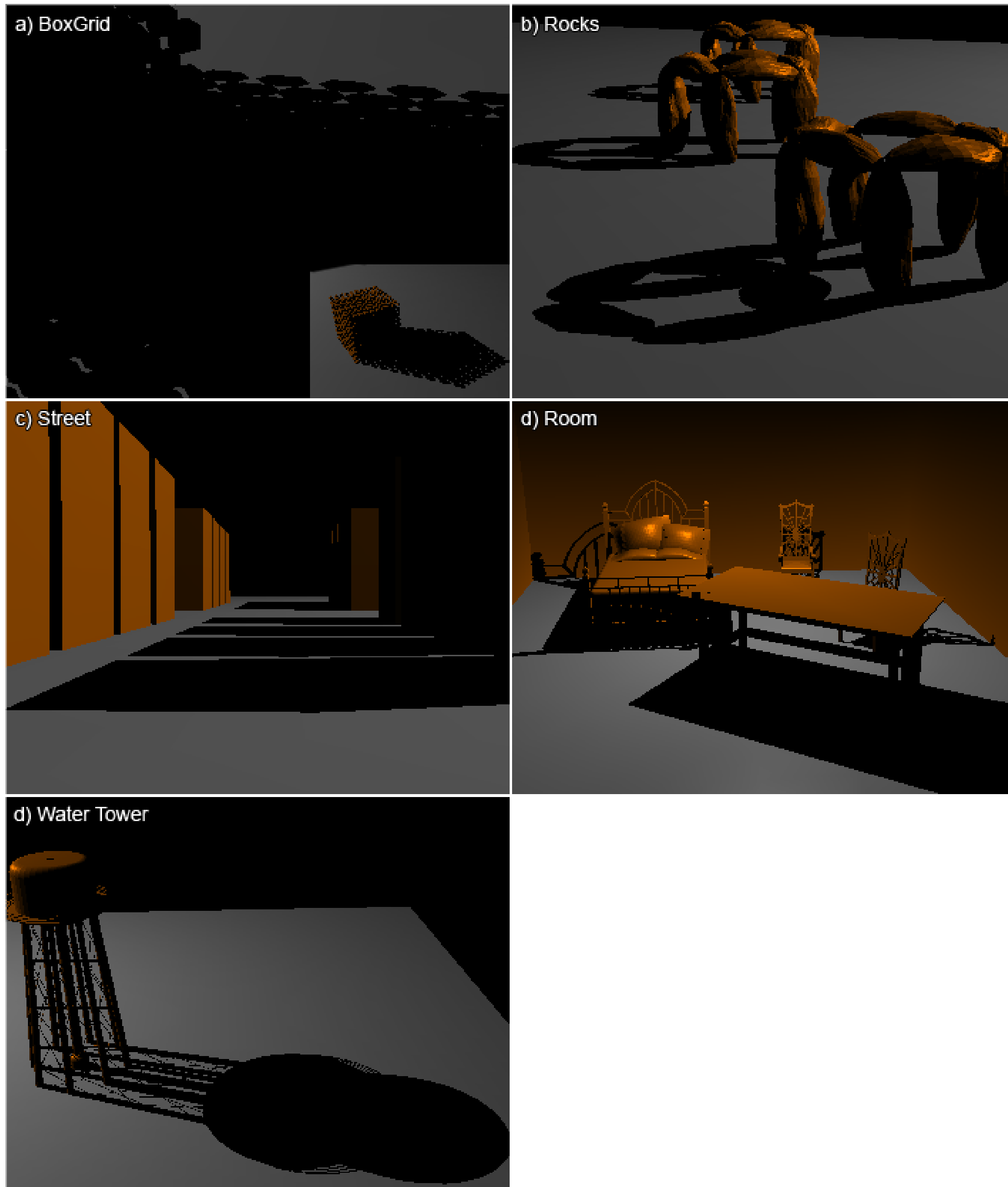


Figure 46: The scenes used for analysis

extremely low number of polygons. This is due to the object-dependencies of both SV's and Chan's. As the number of polygons rise, the amount of time used to calculate and draw the shadow volumes simply increases to an amount that makes these algorithms slower. The higher speed obtained when using SM's and PSM's, as said, does not come for free. The images rendered using these algorithms (see appendix A.1.1 and A.1.2), clearly shows the jagged edges in both of them, even

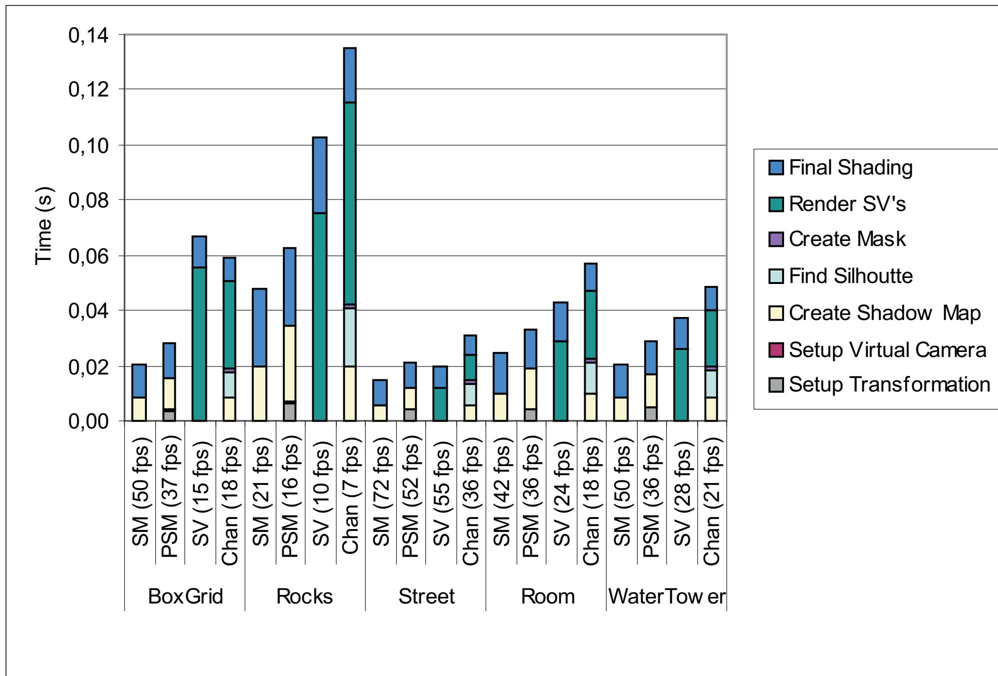


Figure 47: Timing for the five test scenes

though PSM's in some cases minimizes this.

Regarding using SV's and Chan's it should also be noted, that these are dependant on the screen resolution. As this is increased the fill-rate problem also increases making the difference in speed more pronounced. The performance hit when using Chan's algorithm is however not as high as in SV's as Chan's is targeting exactly this problem. Since the SM and PSM algorithms uses the same map size regardless of the screen resolution they are not affected by screen size. All of this is of course without taking into account the increased time used on rendering the scene objects at a high resolution.

Comparing the SM- and the PSM-algorithms shows that the steps of creating the shadow map and rendering using this shadow map is approximately the same. This was to be expected. However when using PSM's the amount of time used to calculate the transformation matrix results in the PSM algorithm to be a little slower in all cases. Whether the speed decrease can be accepted will be discussed later when comparing the quality of the two algorithms.

Comparing SV's and Chan's algorithm for the five test scenes shows that SV's out-perform Chan's algorithm for all but the BoxGrid scene. However in both the BoxGrid and the Water Tower scene, the time used for calculating and rendering shadow volumes is seen to be lower when using Chan's algorithm. This is however not the case in the other scenes, where the rendering time used for drawing shadow

volumes are almost equal, still with the Chan algorithm being a little faster. The penalty using Chan's algorithm is seen to lie in the steps of creating the shadow map and finding silhouette pixels.

This is not an unexpected result. In the BoxGrid scene and the Water Tower scene, complex shadows results in a high overdraw when rendering shadow volumes, leading to high fillrate-consumption. These are exactly the cases in which Chan's algorithm should increase rendering speed. In the current implementation however, creating the shadow map and finding silhouette pixels takes up much of the rendering time, resulting in SV's being faster in most of the test-scenes. This is in agreement with what is stated in [CD04], namely that *...our method is designed to handle dynamic scenes with high shadow complexity, which would ordinarily give rise to many overlapping shadow volumes and quickly saturate the hardware's fillrate*. It should also be noted that comparing the timing diagram in this thesis with the one in [CD04] shows, that the step of finding silhouette pixels could be done faster than in the implementation done here. This could make the Chan algorithm even more interesting.

To sum up it is seen that SM's are clearly the fastest algorithm followed closely by PSM's. SV's is the slowest in all reasonable scenes, and Chan's algorithm lies in between SM's and SV's for scenes with high shadow complexity but is slowest in scenes with low shadow complexity.

7.2 Quality and Robustness

Measuring the quality of the four algorithms is a difficult task as it a question of perception and subjective oppinions. In this section the four algorithms will be investigated one by one using the scenes described above, altering light and camera positions. The pictures can all be seen in appendix A.1. The quality however, can not be discussed without also looking at the speed of the algorithms so there will be a connection to the previous section in this discussion.

7.2.1 Shadow Maps

As stated earlier the quality obtained when using SM's is dependant on both the size of the shadow map, and the cutoff-angle of the light source. As the angle increases the quality decreases since the area covered by the shadow map increases. Increasing the shadow map size is of course a way of minimizing this problem but in real-time applications the size should be as low as possible. In appendix A.2 a series of images is shown rendered using different shadow map sizes and different cutoff-angles. The Water Tower scene is used and the camera is focused on a shadow area of the scene. In figure 52 the cutoff-angle was set as small as possible still enclosing the water tower, while the shadow map size was increase from 256x256 to 2048x2048. As it is clearly seen the jaggedness of the edges are minimized as the map size is increased giving very nice results at a size of 2048x2048. But as stated the quality of the rendered images also depends on the

cutoff-angle of the light source. As the light source cutoff-angle increases shown in figure 53 the quality degrades and in this case even a map of size 2048x2048 cannot remove the jagged edges of the shadow boundary.

7.2.2 Perspective Shadow Maps

A quality measure of the images rendered using PSM's is even harder than for SM's. This is due to the fact that the quality is highly dependant one the locations of both the light source and the camera as discussed earlier. One clear advantage of the PSM algorithm is that the transformed light source used to create the shadow map is focused on areas of interest. This means that the quality will increase as the camera focuses on shadow areas. This however could also be implemented in ordinary SM's and therefore we will only look at the quality differences between SM's and PSM's occurring due to the transformation of the scene.

In appendix A.3 a series of images is shown using the Rocks scene all rendered using a map size of 512x512. The light source have been placed in three different positions: To the right of the viewer, in front of the viewer and behind the viewer. In the images where the light source is to the right of the viewer the effect of using PSM's is clearly seen as the shadow boundary quality of the shadows cast by the nearest rocks are better than the ones when using SM's. This quality increase comes on the expense of the shadow quality far away from the camera. Even though it can be hard to see, the shadow quality here is actually worse than when using SM's. This however could be considered a strength for PSM's as the users normally might focus on objects close to the camera, and not notice the worser quality on far away objects.

When the light source is placed in front of the viewer the quality of SM's and PSM's are seen to be just about equal. This actually happens because of the use of spot light sources. The objects near the camera that becomes large when transformed are far away from the light source and therefore will be transformed back towards their original size in the light projection matrix. One could say that the two transformations cancels each other.

In the last case where the light source is placed behind the camera, the shadows seen when using both SM's and PSM's are quite good. This is as expected since this 'miners lamp' case, is actually a good one for SM's. Here the area seen through a pixel in the camera almost equals the area seen through a pixel in the shadow map, and therefore the shadows are nearly perfect. That is, in this case PSM's cannot contribute to the quality of the shadows.

All in all, it seems that PSM's is a good idea. In some cases the shadow quality is increased and otherwise the quality seems to be as good as SM. This however, in the current implementation, is not always the case. In figure 55 a case where the use of PSM's results in very bad quality is shown. This often happens when the light source is closer to the objects than the camera. The transformation seems to stretch the objects, resulting in the cutoff-angle having to be large to encompass the objects of interest. The result is a shadow map in which objects

are actually smaller than when using ordinary shadow maps. Having this in mind it seems that the robustness of PSM's are quit bad. There are multiple cases in which the quality is worse than when using SM and even using optimizations as cube-clipping, seems not to be enough to make the PSM algorithm really stand out. However the use of spotlights are not the best for PSM's as stated in the original article, in which directional lights are used which gives better results.

7.2.3 SV and Chan's

Doing a quality analysis on images rendered using SV's is trivial since the shadows are always as good as possible. Regarding robustness the use of the z-fail method also ensures that shadows are always calculated correct, even when the camera is placed inside a shadow volume.

Instead we will focus on the quality when using Chan's algorithm for rendering. As stated SV's are used to compute shadows at shadow boundaries and should therefore ensure that Chan's algorithm as SV's should always produce the best possible shadows. However, if the shadow map size is too low, small objects in the scene might not be present in the shadow map and therefore would not be classified as silhouette pixels. The result of this is, that the objects will not cast the appropriate shadow into the scene. An example of this is seen in figure 56 in appendix A.4. Determining the appropriate size of the shadow map is again an almost impossible task, and the programmer using Chan's algorithm might have to determine the size dependant on the scene he is creating.

7.3 Generality

As stated earlier another measurement of how good an algorithm is, is the generality of the algorithm. That is, among other things, the ability to support multiple types of geometry. In this category the SM and the PSM algorithms have a definite advantage. Since they are image space algorithms they are in no way concerned how the geometry is specified. Using SV's the geometry have to be well defined as triangular closed meshes. The same goes for Chan's algorithm which relies on the rendering of shadow volumes too. An example is shown in figure 57 appendix A.5 where a tree consisting of multiple kinds of geometry is rendered. As it is seen SM's and PSM's are not affected by this, whereas SV's and Chan's are not working in this case.

Also the light sources supported are different. Using SV's omni-directional as well as spotlights are supported. This is not the case for the current implementation of SM's and PSM's which only supports spotlights because of the use of a shadow map. Chan's algorithm again is limited by the fact that both SM's and SV's are used for rendering. That is, Chan's algorithm also only supports spotlights.

7.4 Overview

To sum up the results, a table has been made (see Table 7.4) in which the properties of the four algorithms, within the four categories, are shown.

Having these properties in mind, an attempt to compare the algorithms using a subjective grade system has been made. Table 7.4 shows the author's thoughts on the four different aspects of the algorithms, using the five test-scenes. Measuring the quality, robustness, and generality is done on a scale of 1-5 where 5 is best. The points given to the different algorithms are purely the authors opinion of the four algorithms during the work with these. This subjective test is done, because doing an objective test of properties as quality, robustness and generality seems to be an almost impossible task.

Under each category the averages of the grades using the five scenes are shown for each algorithm. For example the average speed points given to SV is 2.6. Under *Generality* only the points 1 and 5 are given, since the points have given based on comparison between the algorithms, in which SV's and Chan's have serious problems compared to SM's and PSM's.

As maybe expected, the average values of the four algorithms under the category speed, shows that SM's and PSM's are in the fast end, while SV's and Chan's are in the low end. The picture regarding quality is reversed. Here SV's and Chan's score high, while SM's and PSM's score low. It is also seen that when it comes to robustness PSM are in the bottom, while the other algorithms are in the top. The table should give an overview of the strengths and weaknesses of the algorithms compared to each other.

	Speed	**Quality**	**Robustness**	**Generality**
SM	- The fastest of the four - Depends only on shadow maps size	- Jagged edges worse the lower the map size and the higher the light cutoff-angle	- Aliasing worst when light shines viewer in the eyes - Biasing factor hard to determine, must compromise	- Handles all kinds of geometry - Handles self shadowing - Handles only spot lights
PSM	- Nearly as fast as SM's - Adds small amount of time to setup transformation	- Better than SM in some cases - worse in others	- Best for outdoor scenes where light is far away from the camera frustum - Multiple situations in which shadows are bad or almost disappears	- Handles all kinds of geometry - Handles self shadowing - Handles only spot lights
SV	- Slow when many polygons - Decreases with screensize due to fill-rate consumption - Decreases with shadow complexity	- Best possible shadows	- Always generates the right results	- Handles only closed meshes - Handles self shadowing - Handles all light source types
Chan	- Faster than SV's when high shadow complexity - Adds time to create shadow map and silhouette mask before using shadow volumes in selected pixels	- Almost as good as SV's - If shadow map is too little, small shadow regions may be missing	- Eliminates most biasing problems although using SM's.	- Handles only closed meshes - Handles self shadowing - Handles only spotlights

Table 1: Strengths and weaknesses for the four algorithms under the categories: *Speed*, *Quality*, *Robustness* and *Generality*

	Scene	Polygons	Shadow Complexity	Speed		Quality		Robustness		Generality
SM	BoxGrid	12000	5	5		2		4		5
	Rocks	63000	2	5		3		5		
	Street	400	2	5	5	2	2.4	5	4.6	
	Room	19000	3	5		4		4		
	Tower	12500	4	5		1		5		
PSM	BoxGrid	12000	5	4		3		2		
	Rocks	63000	2	3		3		2		
	Street	400	2	5	4	2	2.6	3	2.2	
	Room	19000	3	4		2		2		
	Tower	12500	4	4		3		2		
SV	BoxGrid	12000	5	1		5		5		1
	Rocks	63000	2	2		5		5		
	Street	400	2	5	2.8	5	5	5	5	
	Room	19000	3	3		5		5		
	Tower	12500	4	3		5		5		
Chan	BoxGrid	12000	5	2		4		5		
	Rocks	63000	2	1		5		5		
	Street	400	2	5	2.4	5	4.2	5	5	
	Room	19000	3	2		4		5		
	Tower	12500	4	2		3		5		

Table 2: Subjective grades diagram using the four algorithms when rendering the five test-scenes. Under each category the algorithms have been graded from 1-5 based on the subjective opinions of the author

8 Discussion

In the following sections the results obtained will be discussed. We will look into comparing the algorithms, discussing whether or not a definite winner can be chosen or if all of the algorithms are equally interesting. After this it will be discussed whether or not using multiple algorithms could be an idea.

8.1 Comparison

As stated earlier comparing the four algorithms is a challenging tasks, since such a comparison will always be a matter of subjective opinions. Measuring the speed of an algorithm is easily done, but measuring quality, robustness etc. is quite hard.

Doing a comparison of the four algorithms is therefore a matter of looking at the strengths and weaknesses of the algorithms, deciding which aspects are important in the application in which they should be used.

Looking at the Shadow Map algorithm, it is clearly seen that the speed is the main advantage. One could state that this should be reason enough for choosing this algorithm. In the images shown the quality of the images even seems quite good, if the size of the shadow map is not too small, and problems only occur in situations where we have focussed on a shadow boundary.

But what are the problems then? Well, in a complete game-engine texture memory consumption might very well be a problem, meaning that the shadow map size is limited in some way. If we look at the memory consumption of one light source using 24 bit shadow maps of sizes 512×512 and 2048×2048 the result is:

MapSize: 512×512 Memory Usage: $512 \times 512 \times 3 \simeq 790KB$

MapSize: 2048×2048 Memory Usage: $2048 \times 2048 \times 3 \simeq 12.6MB$

That is an increase of a factor 16. And for only one light source.

Another problem is the area that should be covered by the light source. As this increases either the quality of the shadows decreases, larger shadow maps should be used or multiple shadow maps should be used, again decreasing the frame-rate. The need for high resolution shadow maps, also present itself when the light is shining at the viewer. As stated earlier, this case is where aliasing problems are worst. The same applies to Perspective Shadow Maps, whereas Shadow Volumes and Chan's are not concerned with the lights position.

A way of making shadow maps more attractive, would be to use more than four samples for PCF. This could for example be done in a fragment program. Thereby, the edges of the shadows would become even smoother and resemble fake soft shadows more. How this would affect the speed of the algortihm is not tested in the current implementation, but is definitely worth looking into.

Even though we have concentrated our efforts on situations involving spotlights, it should also be noted that Shadow Maps in the current form, only supports spotlights, as opposed to Shadow Volumes, which supports all types of light

sources. Using Shadow Maps for omni-directional point light sources can be done in various ways, but most of these involves creating multiple shadow maps which again decreases the speed.

So, how about using Perspective Shadow Maps? Well, the idea seems good. Modifying the scene prior to creating the shadow map can result in increased shadow map resolution at areas of interest, thereby minimizing the need for larger shadow maps. They also focus the shadow map on areas of interest which could be an advantage if a large area is to be lit, but only a small fraction of this area is seen at a specific time. This idea though, could also be used in the ordinary Shadow Map algorithm increasing the quality here too.

However, it seems that there are not that many cases in which Perspective Shadow Maps increases the quality noticeably, and there are even cases where the quality is worse than that obtained using Shadow Maps.

The biggest problem when using Perspective Shadow Maps seems to be the low degree of robustness. The algorithm is highly dependant on scene geometry and positioning of camera and lights. Much care have to be taken, that none of the cases resulting in bad shadows are possible in the specific application. This makes it difficult to find situations where Perspective Shadow Maps would be the algorithm of choice.

Another thing, is the fact that shadows calculated using Perspective Shadow Maps are affected by camera movement. This results in shadows flickering as the camera moves, giving highly undesirable results. This is not the case in the other three algorithms as they are not taking camera placement into account.

All in all, it seems that with the implementation done in this thesis, there are multiple things that should be changed for the use of Perspective Shadow Maps to be really interesting. Even if the idea is great, there are many difficulties in the implementation and in the idea itself, and the time spent on doing the steps, involving the transformation, does not seem to increase the quality enough for the use of Perspective Shadow Maps to be advised.

This is not to say that we should forget about Perspective Shadow Maps. The idea of transforming the scene is good, but maybe the transformation should be done in another way, such as proposed in [WSP04] or [MT04].

Regarding Shadow Volumes the picture is quite clear. Perfect quality at the expense of speed. As seen the quality obtained when using Shadow Volumes are as good as possible, with regard to hard shadows. However, the speed decreases rapidly as the number of polygons rise, and with the number of polygons in a game scene today using *ordinary* Shadow Volumes seems to be a bad idea.

This however should be seen, mostly to be the case in dynamic scenes. That is, scenes where lights or objects move relatively to each other. If this is not the case, the shadow volumes needs not to be recalculated for every frame, and the speed could be increased considerably. There is however still the problem of fillrate-consumption since the shadow volumes needs to be redrawn as long as the camera is moving.

Also it should be noted that Shadow Volumes handles all types of light sources,

and that the algorithm is robust in all cases. That is, the shadows created are not affected by scene geometry, light or camera position etc. However, the algorithm has a minor weakness in the fact that geometry must be closed meshes. Although using such meshes are often done, situations where other types of geometry is used, is likely to appear.

Chan's algorithm proposes a great idea, that could make the idea of shadow volumes worth paying much more attention too. Realizing the fact that much of the shadow volumes does not affect the final result, there seems to be reason in trying to eliminate the drawing of shadow volumes in these areas.

Chan's suggestion seems to work well in situations with high shadow complexity. As the overall complexity of game scenes today, as well as capabilities of the graphics cards, seems to increase continuously, the difference in speed between the Shadow Maps and the Chan algorithms might decrease, making hybrid algorithms as Chan's and other optimizations to the idea of Shadow Volumes very interesting.

While Chan's algorithms uses the best properties of both Shadow Maps and Shadow Volumes, it also inherits some of the bad properties of them both. That is, Chan's algorithm, as Shadow Maps, only works with spotlights. Even though the map size is not that important in Chan's algorithm it has been shown that at minimal resolution is needed. Also, as with Shadow Volumes, geometry needs to be closed meshes.

So, do we have a definite winner? No, it seems that the choice of algorithm is affected by many questions, such as *Speed or Quality?*, *Indoor or Outdoor scenes?*, *Large areas covered by light source?*, etc. So why not look into the idea of using multiple algorithms, choosing the one that seems best in the specific case.

8.2 Using Multiple Algorithms

As seen from the above, none of the discussed algorithms have all of the properties of the "perfect" algorithm, meaning that none of them are perfect for all situations that can arise in a game. Instead of using just a single algorithm for all shadow calculations, it is therefore suggested to use multiple algorithms.

The optimal goal would be, to let the application choose this continuously at realtime. This however, seems to be a very challenging task since there are many aspects to be considered when choosing the algorithm and many of these are nearly impossible to measure.

Instead it could be an idea to let the choice of algorithm be decided at design time. That is, the programmer should decide, based on the observations in this thesis as well as other papers, which algorithm to use in specific cases. This could mean that the idea of Perspective Shadow Maps is used for large outdoor scenes, while Shadow Maps could be used when light source cover a minimal area in the scene, such as light from a table lamb or such.

Covering all cases that could appear in a game is an almost impossible task, but below some ideas on when to choose the different algorithms are given.

Shadow Maps Use it when the area receiving shadow is small (the light cut-off angle is low), or when the camera will never get near shadow boundaries. If the light is positioned close to the camera aliasing is almost completely avoided and Shadow Maps will create nearly perfect shadows.

Perspective Shadow Maps Use when the light source covers a large area while the camera is viewing only a little part of this. Could be a large outdoor scene with the light source being the sun. Note that it seems that in many cases results are worse shadows than when using Shadow Maps.

Shadow Volumes Use when the number of polygons is low or when perfect shadows are crucial. Might be good if objects and light sources are not moving.

Chan Use in scenes with complex shadows where camera could get close to the shadows. Especially good when Shadow Volumes would cover the entire scene but only cast limited shadow. For example light coming through a complex ceiling.

Besides choosing the desired algorithm, the designer should reflect on the fact, that all of these algorithms could be altered, to make them more suitable for the specific task. That is, if special assumptions about scene such as object movement etc. can be made, alterations to the algorithms could most likely increase their performance with regard to speed, quality etc.

9 Conclusion

During this thesis four different shadow algorithms for generating hard shadows have been implemented and analyzed. Strengths and weaknesses of the algorithms has been looked into, and an full description of how to implement these have been shown.

An application has been created in which the choice of algorithm used, can be change at runtime allowing to compare the four algorithms.

It has been shown that none of the algorithms are perfect in all aspects of generating shadows, like speed, quality etc., overall results showing that the image based algorithms like Shadow Maps and Perspective Shadow Maps generally excels in speed having problems with quality, whereas the object based algorithms like Shadow Volumes and Chan's hybrid algorithm excels in quality at the expense of speed.

It has been shown that many aspects are to be considered if only a single algorithm is to be used in a realtime 3D application. Therefore, the idea of using multiple algorithms depending on the case at hand, has been proposed.

We have discussed the possibility of choosing the appropriate algorithm dynamically during runtime, and seen that this is a very challenging task, since most aspects of shadow generation is not measurable. Furthermore the choice of algorithm has been shown to be highly scene dependant. Instead we have proposed that the choice of algorithm is done at design time, and we have given some suggestions to which algorithm to choose, based on the observations in the thesis.

A Images

A.1 Test Scenes

A.1.1 Shadow Maps

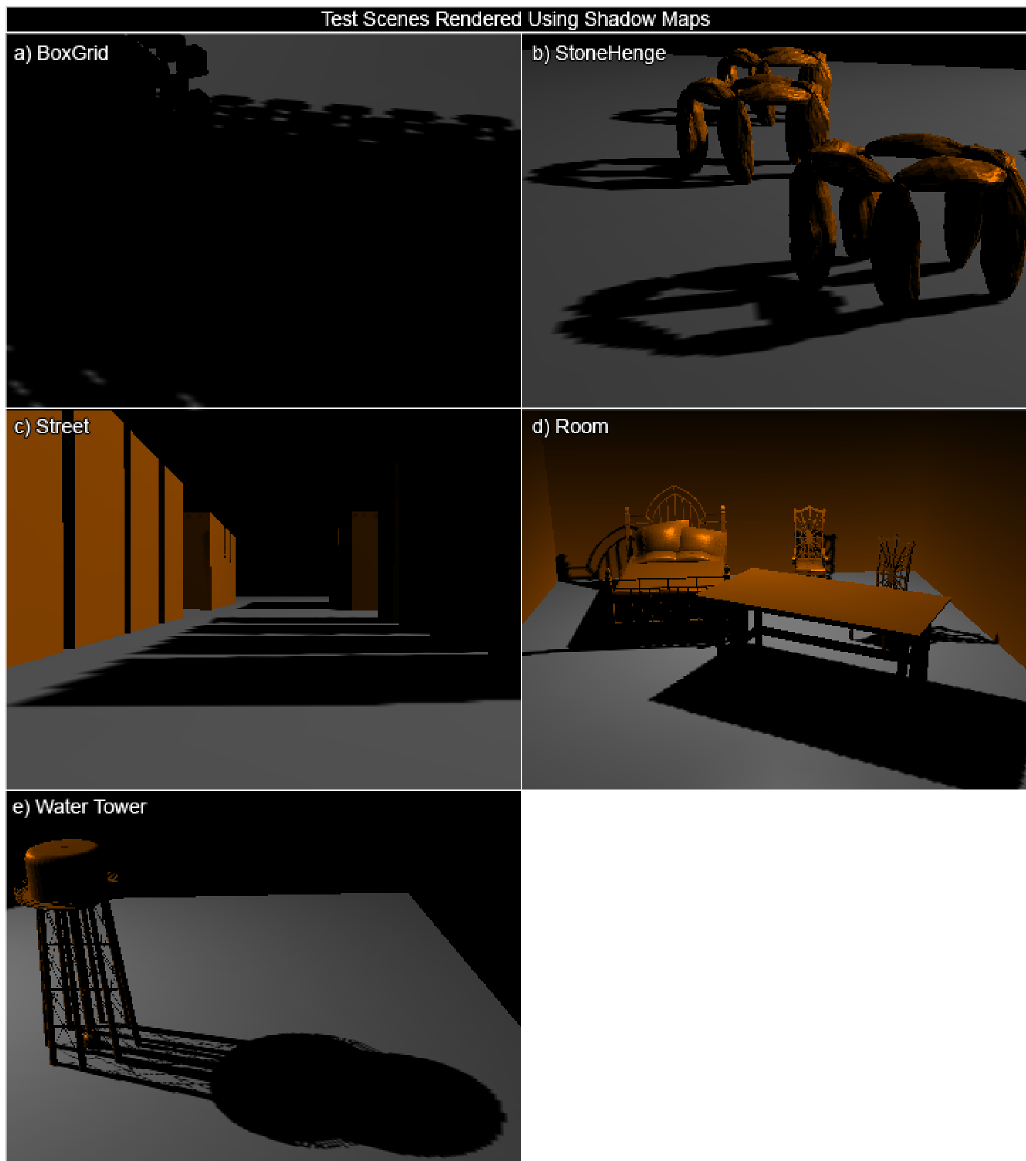


Figure 48: Test scenes rendered using Shadow Maps

A.1.2 Perspective Shadow Maps

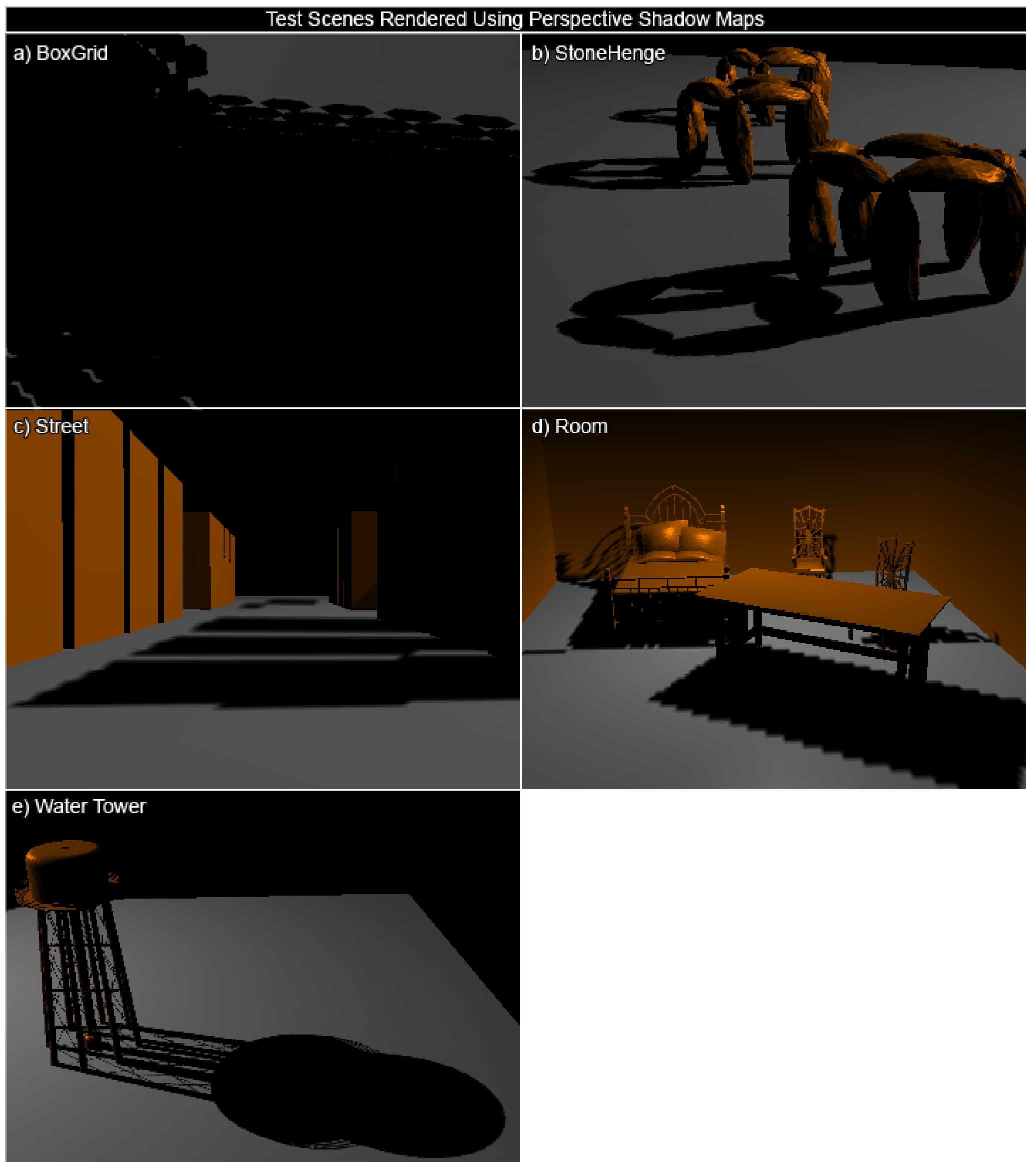


Figure 49: Test scenes rendered using Perspective Shadow Maps

A.1.3 Shadow Volumes

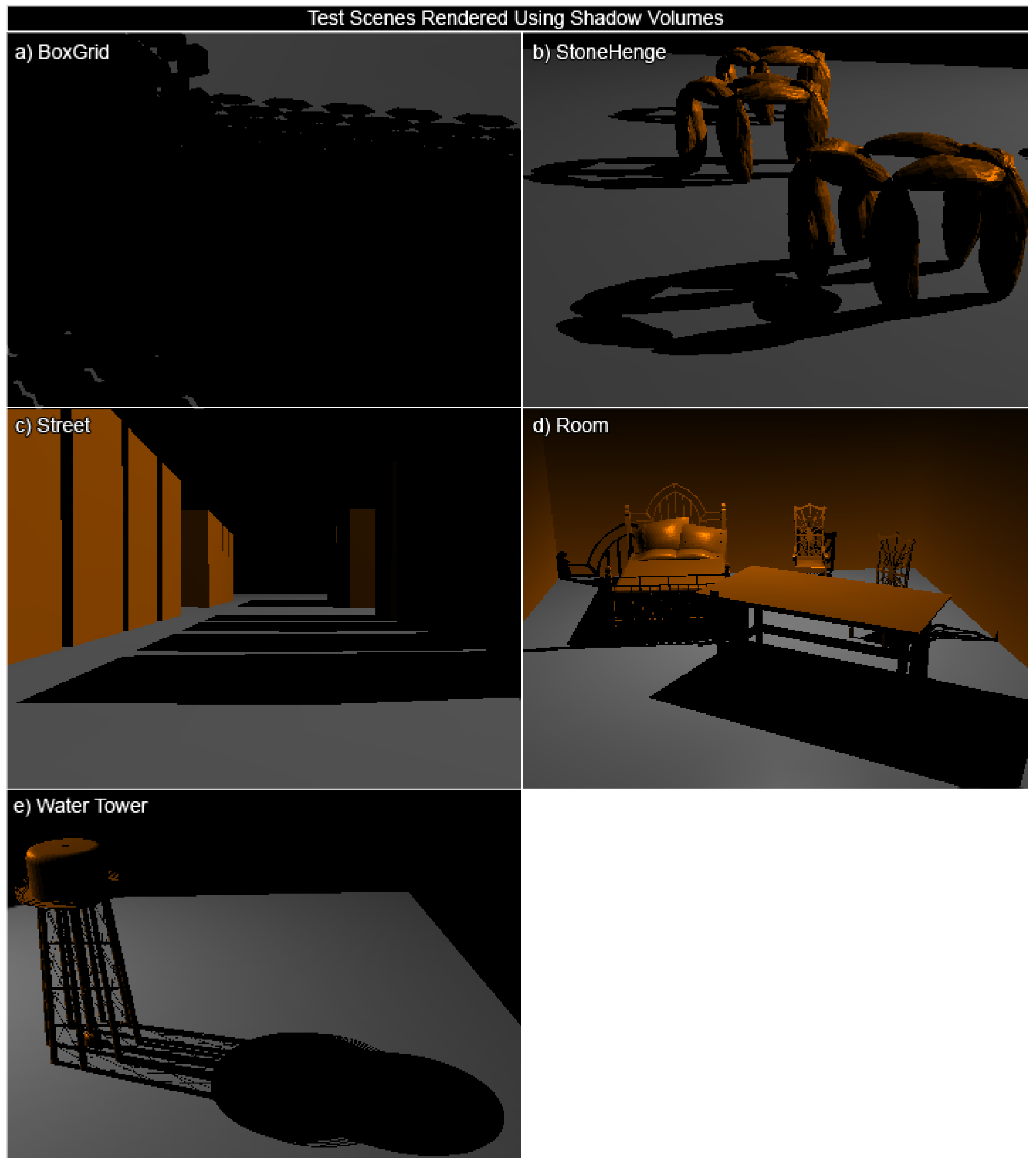


Figure 50: Test scenes rendered using Shadow Volumes

A.1.4 Chan's hybrid

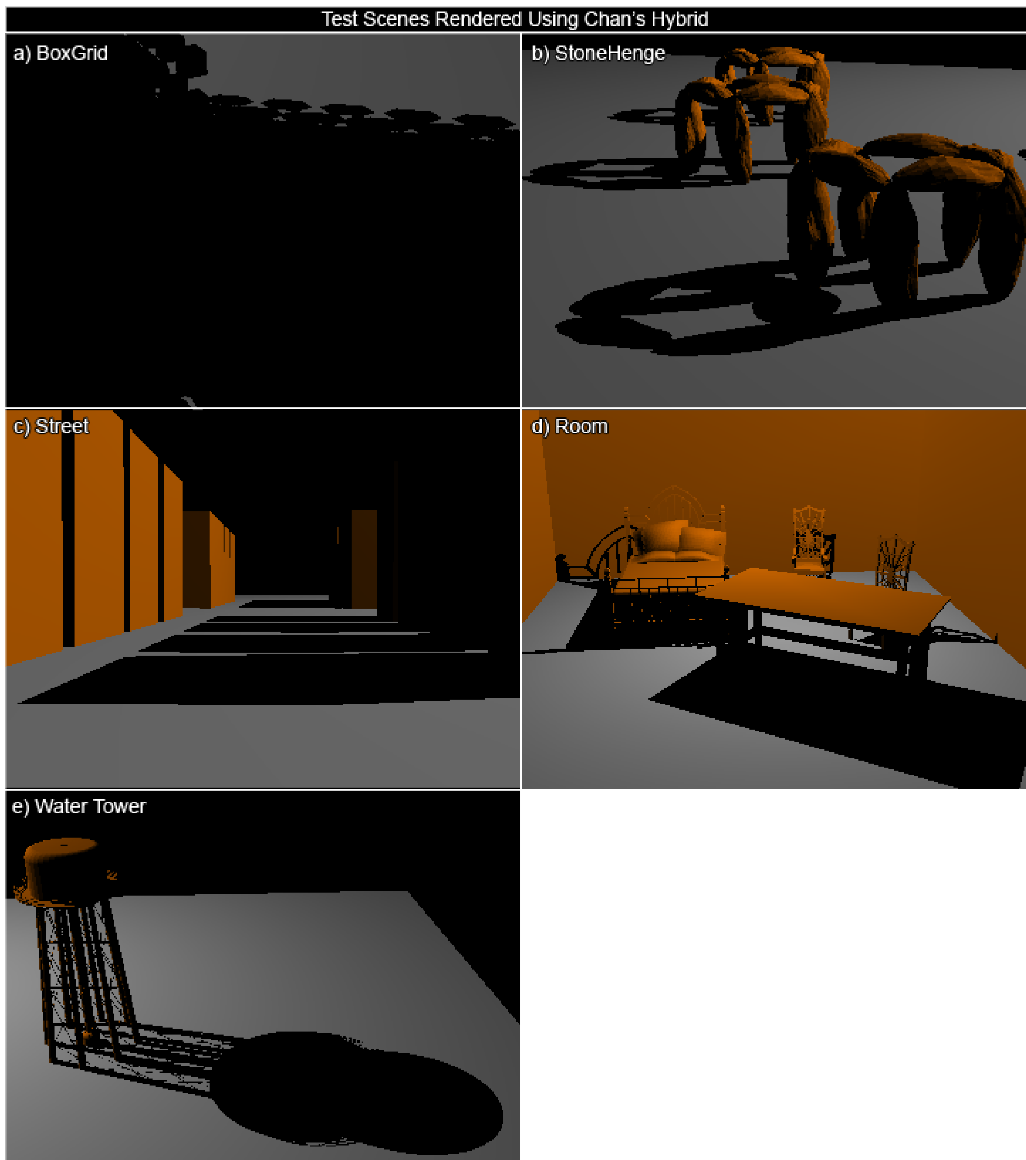


Figure 51: Test scenes rendered using Chan's hybrid

A.2 Shadow Map Quality

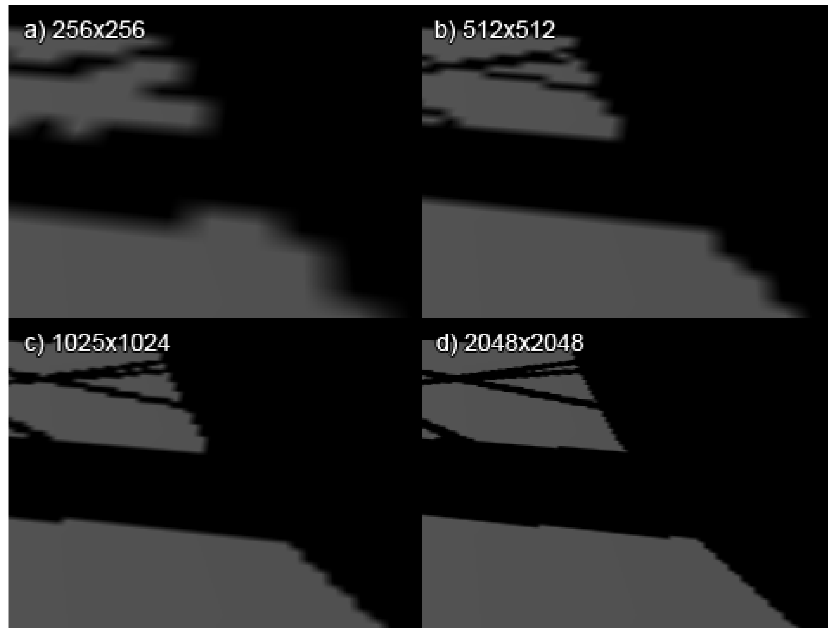


Figure 52: Quality as function of shadow map size using the smallest possible cutoff angle enclosing the Water Tower

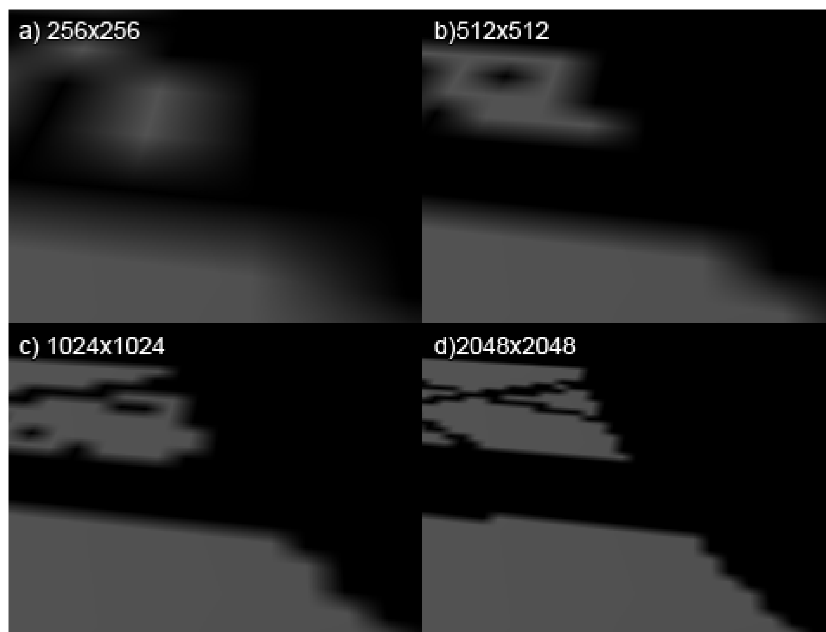


Figure 53: Quality as function of shadow map size using a larger cutoff angle

A.3 Perspective Shadow Map Quality

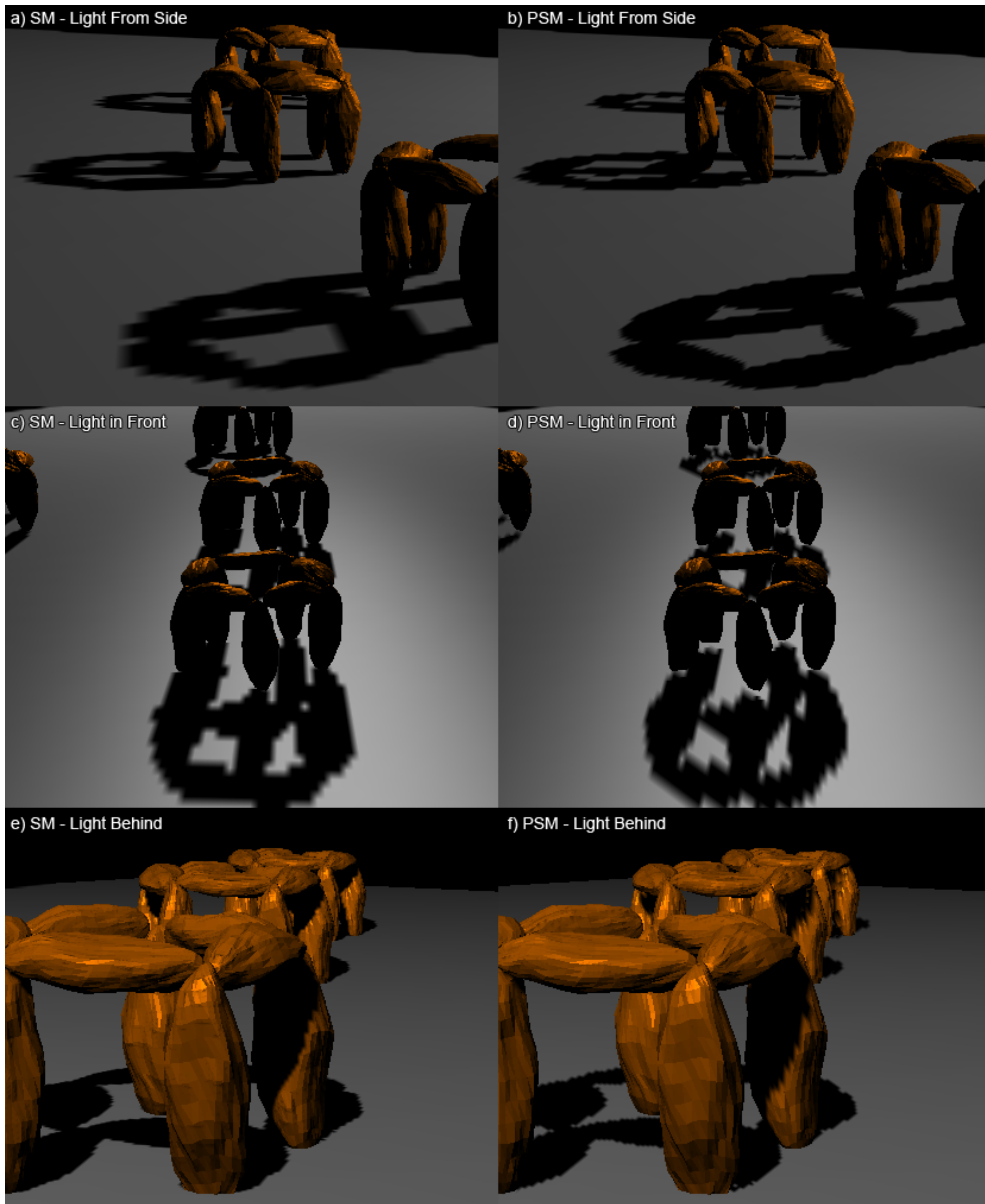


Figure 54: Quality of PSM's.

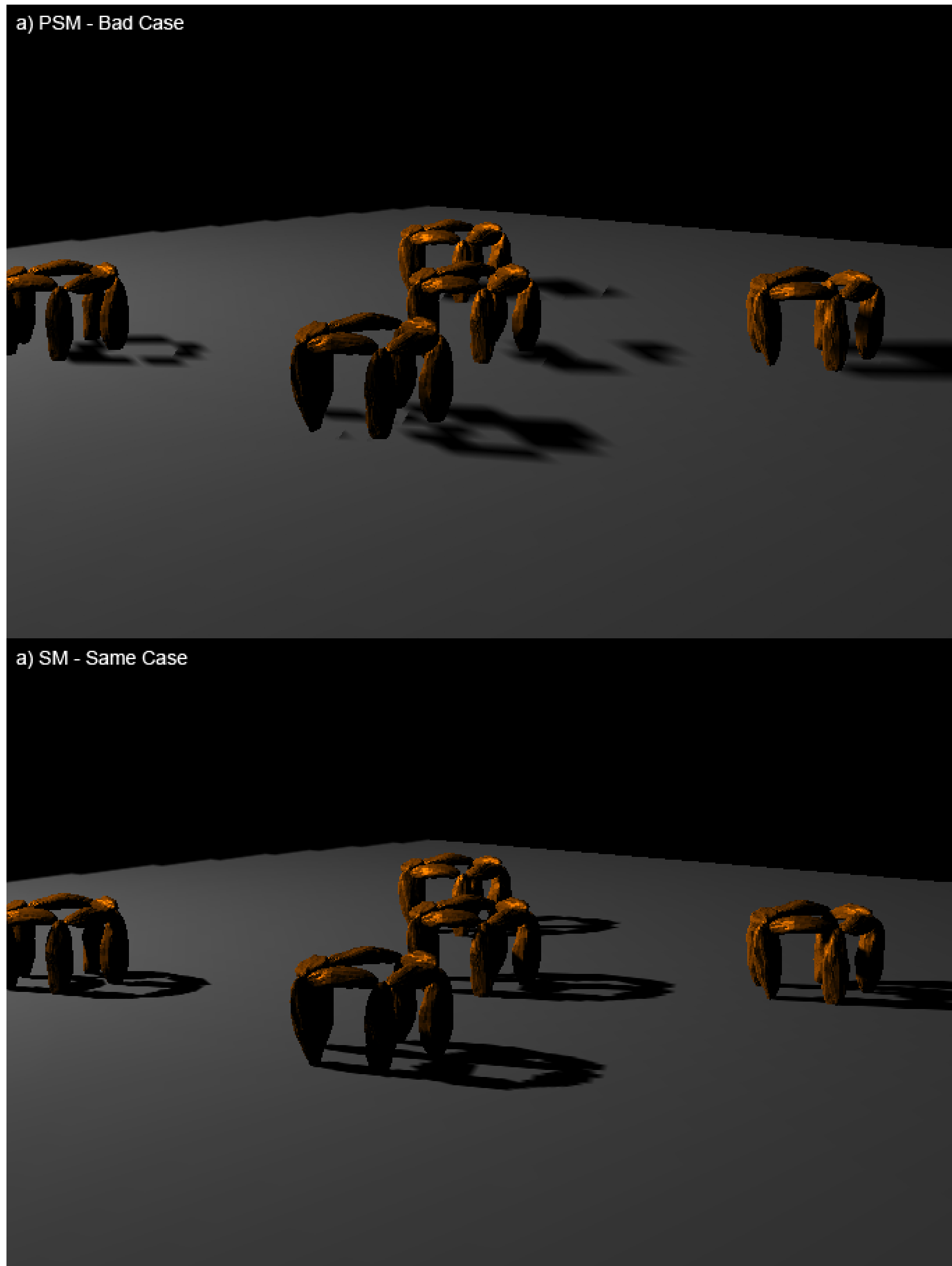


Figure 55: A case in which PSM's quality is much worse than SM' quality

A.4 Shadow Volumes and Chan's Hybrid Quality

Figure 56: Using too low shadow map size can lead to small or thin objects not to be classified as silhouette pixels, and therefore be wrongly shadowed

A.5 Generality

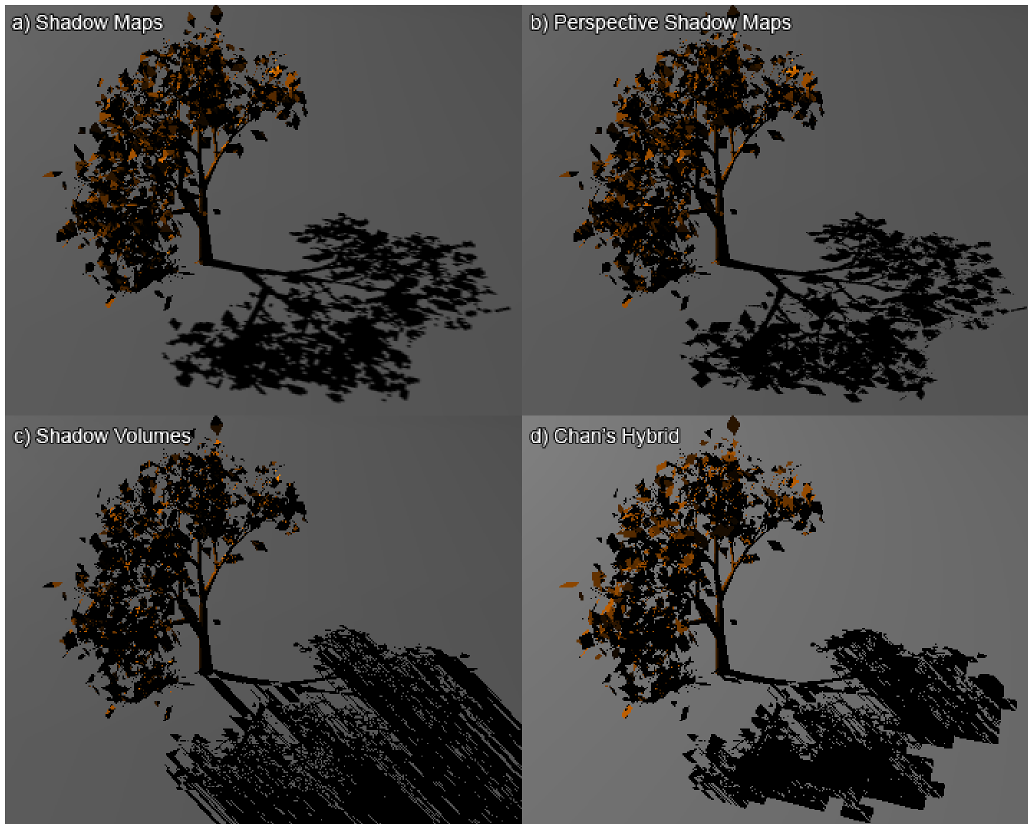


Figure 57: SM's and PSM's are not concerned with geometry, as SV's and Chan's.

B Keyboard Shortcuts

B.1 Basic Shortcuts

```
a   :   Move light source left
s   :   Move light source back
w   :   Move light source forward
d   :   Move light source right
z   :   Move light source up
x   :   Move light source down
r   :   Zoom in
f   :   Zoom out
m   :   Animate light
9   :   Decrease light cut-off angle
0   :   Increase light cut-off angle
P   :   Print number of polygons
```

B.2 Shadow Maps

```
o   :   Sets near and far plane, used when creating shadow map,
        to encompass all objects
l   :   Minimizes light cutoff-angle to only encompass objects
        in the scene
p   :   Toggles use of Percentage Closer Filtering
+   :   Increases size of Shadow Map
-   :   Decreases size of Shadow Map
y   :   Increases Bias factor
h   :   Decreases Bias factor
Y   :   Increases Bias units
H   :   Decreases Bias units
```

B.3 Perspective Shadow Maps

```
o   :   Sets near and far plane, used when creating shadow map,
        to encompass all objects
j   :   Toggles use of camera slideback
p   :   Toggles use of Percentage Closer Filtering
c   :   Toggles use of cube clipping
u   :   Toggles see from light
k   :   Toggles using unit cube or bounding box for
        near and far planes of transformed light camera
l   :   Toggles optimizing near and far plane of camera
        before transformation
+   :   Increases size of Shadow Map
-   :   Decreases size of Shadow Map
```

```
y   :   Increases Bias factor
h   :   Decreases Bias factor
Y   :   Increases Bias units
H   :   Decreases Bias units
```

B.4 Shadow Volumes

```
t   :   Toggles Shadow Volumes
b   :   Toggles using two-sided stencil test
```

B.5 Chan's

```
t   :   Toggles use of Depth Bounds Test (Only for testing purposes)
b   :   Toggles using two-sided stencil test
v   :   Toggles showing silhouette pixels
n   :   Toggles showing shadow volumes
+   :   Increases size of Shadow Map
-   :   Decreases size of Shadow Map
```

C Code

C.1 Shadow Maps (Using PBuffer)

```
/****************************************************************/
/* Filename: ShadowMapPBuf.h                                    */
/****************************************************************/

#ifndef __SHA_MAPPB_H__
#define __SHA_MAPPB_H__

#include "My/RenderAlgorithm.h"
#include "Components/Timer.h"
#include "My/timebox.h"

class ShadowMapPBuf : public RenderAlgorithm
{
protected:
    vector<RenderTexture*> rt;
    Components::Timer timer;
    int map_size;

    //Matrices
    GLfloat lightProjectionMatrixTemp[16], lightViewMatrixTemp[16];
    GLfloat cameraProjectionMatrixTemp[16], cameraViewMatrixTemp[16];
    Mat4x4f biasMatrix;
    Mat4x4f texture_matrix;

    bool show_map;
    bool pcf;

    // For biasing
    float factor, units;

    //For text output
    int biastext;
    int mapsizetext;

#ifdef REC_TIME
    TimeToExcel* exc_timer;
#endif

public:
    ShadowMapPBuf(Object* _obj, LightContainer* _lights, Camera* _cam,
        Object* _floor) : RenderAlgorithm(_obj, _lights, _cam, _floor){
        rt.resize(lights->no_lights());
        map_size = 1024;
        pcf = true;
        init();
        init_shadow_texture();
        show_map = true;
        biasMatrix = Mat4x4f(Vec4f(0.5, 0.0, 0.0, 0.5),
                             Vec4f(0.0, 0.5, 0.0, 0.5),
                             Vec4f(0.0, 0.0, 0.5, 0.5),
                             Vec4f(0.0, 0.0, 0.0, 1.0));

        biastext = displaytext.addText("");
        mapsizetext = displaytext.addText("");
        factor = 1.1;
        units = 4.0;
```

```
        char buf[50];
        sprintf(buf, "Mapsize: %dx%d", map_size, map_size);
        displaytext.updateText(mapsizetext, buf);

#ifdef REC_TIME
        // For Time Reporting
        exc_timer = new TimeToExcel("./output/ShadowMapPBufOutput.txt");
        exc_timer->add_title_item("Create Shadow Map");
        exc_timer->add_title_item("Rendering using Shadow Map");
        exc_timer->write_title_line();
#endif
};

    void render();
    void init();
    void init_shadow_texture();
    void show_shadow_map(int shadow_map);
    void show_misc_info(int window);
    void copy_drawing_buffer_to_texture(GLuint tex_map);
    void prepare_for_texture_render();
    void enable_r_to_texture_comparison(int i);
    void reset_all();
    void restore_states();
    Mat4x4f calculate_texture_matrix(Light* light);
    Mat4x4f get_light_projection_matrix(Light* light);
    Mat4x4f get_light_view_matrix(Light* light);
    void generate_texture_coordinates(Mat4x4f* texture_matrix);
    void process_key(unsigned char key)
    {
        Vec3f p0;
        Vec3f p7;
        switch(key) {
            case 'p':
                pcf = !pcf;
                init_shadow_texture();
                break;
            case 'o':
                obj->get_bbox(p0, p7);
                for(int i=0; i<lights->no_lights(); i++)
                    lights->get_light(i)->calculate_near_far(bbox(p0,p7));
                break;
            case 'l':
                obj->get_bbox(p0, p7);
                for(int i=0; i<lights->no_lights(); i++)
                    lights->get_light(i)->calculate_fustum(bbox(p0,p7));
                break;
            case 'y':
                increase_bias_factor(0.1);
                break;
            case 'h':
                decrease_bias_factor(0.1);
                break;
            case 'Y':
                increase_bias_units(0.1);
                break;
            case 'H':
                decrease_bias_units(0.1);
                break;
            case 43: //Ascii +
                increase_map_size();
                break;
            case 45: //Ascii -
```

```
                decrease_map_size();
                break;
            default:
                RenderAlgorithm::process_key(key);
                break;
        }
    }

    /* set functions */
    void set_map_size(int i){
        map_size = i;
        for(int j=0; j<lights->no_lights(); j++)
            delete rt[j];
        init_shadow_texture();
        char buf[50];
        sprintf(buf, "Mapsize: %dx%d", map_size, map_size);
        displaytext.updateText(mapsizetext, buf);
    }
    void increase_map_size(){
        if(map_size < 2048){
            map_size *= 2;
            for(int j=0; j<lights->no_lights(); j++)
                delete rt[j];
            init_shadow_texture();
            char buf[50];
            sprintf(buf, "Mapsize: %dx%d", map_size, map_size);
            displaytext.updateText(mapsizetext, buf);
        }
    }
    void decrease_map_size(){
        map_size /= 2.0;
        for(int j=0; j<lights->no_lights(); j++)
            delete rt[j];
        init_shadow_texture();
        char buf[50];
        sprintf(buf, "Mapsize: %dx%d", map_size, map_size);
        displaytext.updateText(mapsizetext, buf);
    }

    /* print functions */
    void print_all()
    {
        RenderAlgorithm::print_all();
        cout << "Map Size: " << map_size << endl;
    }

    void increase_bias_factor(float amount=0.1){factor += amount;}
    void decrease_bias_factor(float amount=0.1){factor -= amount;}
    void increase_bias_units(float amount=0.1){units += amount;}
    void decrease_bias_units(float amount=0.1){units -= amount;}
};
#endif

/*****************************************************************/
/* filename: ShadowMapPBuf.cpp                                   */
/*****************************************************************/
#include "My/ShadowMapPBuf.h"

void ShadowMapPBuf::init()
{
    /* Initialize Glew*/
    GLenum err = glewInit();
```

```
    if (GLEW_OK != err)
    {
        fprintf(stderr, "Error loading GLEW");
        system("PAUSE");
        exit(0);
    }
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glShadeModel (GL_SMOOTH);
    /* Enable depth test and stencil test */
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_STENCIL_TEST);
    glDepthFunc(GL_LEQUAL);
    /* Enable lighting */
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    RenderAlgorithm::init();
}

void ShadowMapPBuf::render()
{
#ifdef REC_TIME
    exc_timer->start();
#endif
    /* For all lights */
    /* 1. Render the scene as seen from the light */
    /* Create virtual camera at lights position */
    for(int i=0; i<lights->no_lights(); i++){
        Camera* light_cam = new Camera(lights->get_light(i)->get_position(),
                                       lights->get_light(i)->get_direction()
                                           ,
                                       lights->get_light(i)->get_up(),
                                       lights->get_light(i)->
                                           get_cutoff_angle()*2.0,
                                       lights->get_light(i)->get_near(),
                                       lights->get_light(i)->get_far());
        rt[i]->BeginCapture();
        glEnable(GL_DEPTH_TEST);
        glViewport(0,0,map_size,map_size);
        prepare_for_texture_render();
        load_projection(light_cam);
        load_modelview(light_cam);
        glCallList(dlDrawScene);
        rt[i]->EndCapture();
        restore_states();
        delete(light_cam);
    }
#ifdef REC_TIME
    exc_timer->add_time();
#endif

    /* 2. Render the scene from the camera using shadow maps */
    /*    The depth-value seen from the camera is transformed to light-space
            and compared to the depth-value from the texture map. */
    glViewport(0,0,vpWidth,vpHeight);
    clear_all();

    for(int i=0; i<lights->no_lights(); i++)
    {
        lights->turn_all_off();
        lights->get_light(i)->turn_on();
        load_projection(cam);
        load_modelview(cam);
```

```
        update_lights();

        texture_matrix = calculate_texture_matrix(lights->get_light(i));
        generate_texture_coordinates(&texture_matrix);

        enable_r_to_texture_comparison(i);
        rt[i]->EnableTextureTarget();

        glCallList(dlDrawScene);

        rt[i]->DisableTextureTarget();

#ifdef REC_TIME
        exc_timer->add_time();
        exc_timer->end_line();
#endif
        /* Draw misc info */
#ifdef SHOW_MISC_INFO
        glDisable(GL_BLEND);
        glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
        glTexParameteri(rt[i]->GetTextureTarget(),
            GL_TEXTURE_COMPARE_MODE_ARB, GL_NONE);
        show_shadow_map(i);
        glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
#endif
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_COLOR, GL_ONE);
        glBlendFunc(GL_ONE, GL_ONE);

    }

    glDisable(GL_BLEND);
    reset_all();
    RenderAlgorithm::render();
}

void ShadowMapPBuf::reset_all()
{
    //Disable textures and texgen
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
    glDisable(GL_TEXTURE_GEN_R);
    glDisable(GL_TEXTURE_GEN_Q);
    //Restore other states
    glDisable(GL_LIGHTING);
    glDisable(GL_ALPHA_TEST);
}

void ShadowMapPBuf::enable_r_to_texture_comparison(int i)
{
    //Bind & enable shadow map texture
    rt[i]->BindDepth();
    glEnable(rt[i]->GetTextureTarget());

    //Enable shadow comparison
    glTexParameteri(rt[i]->GetTextureTarget(), GL_TEXTURE_COMPARE_MODE_ARB,
        GL_COMPARE_R_TO_TEXTURE);

    //Shadow comparison should be true (ie not in shadow) if r<=texture
    glTexParameteri(rt[i]->GetTextureTarget(), GL_TEXTURE_COMPARE_FUNC_ARB,
        GL_LEQUAL);
```

```
    //Shadow comparison should generate an INTENSITY result
    glTexParameteri(rt[i]->GetTextureTarget(), GL_DEPTH_TEXTURE_MODE_ARB,
        GL_INTENSITY);
}

void ShadowMapPBuf::generate_texture_coordinates(Mat4x4f* texture_matrix)
{
    //Set up texture coordinate generation.
    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_S, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[0].get()));
    glEnable(GL_TEXTURE_GEN_S);

    glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_T, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[1].get()));
    glEnable(GL_TEXTURE_GEN_T);

    glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_R, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[2].get()));
    glEnable(GL_TEXTURE_GEN_R);

    glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_Q, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[3].get()));
    glEnable(GL_TEXTURE_GEN_Q);
}

Mat4x4f ShadowMapPBuf::calculate_texture_matrix(Light* light)
{
    Mat4x4f light_projection_matrix = get_light_projection_matrix(light);
    Mat4x4f light_view_matrix = get_light_view_matrix(light);

    return biasMatrix * light_projection_matrix * light_view_matrix;
}

Mat4x4f ShadowMapPBuf::get_light_projection_matrix(Light* light)
{
    glMatrixMode (GL_PROJECTION);
    glPushMatrix();
        glLoadIdentity ();
        gluPerspective(light->get_cutoff_angle()*2.0, 1, light->get_near(),
            light->get_far());
        glGetFloatv(GL_TRANSPOSE_PROJECTION_MATRIX,
            lightProjectionMatrixTemp);
        Mat4x4f lightProjectionMatrix(lightProjectionMatrixTemp);
    glPopMatrix();
    return lightProjectionMatrix;
}

Mat4x4f ShadowMapPBuf::get_light_view_matrix(Light* light)
{
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
        glLoadIdentity();
        gluLookAt(light->get_position()[0],
                  light->get_position()[1],
                  light->get_position()[2],
                  light->get_direction()[0],
                  light->get_direction()[1],
                  light->get_direction()[2],
                  light->get_up()[0],
                  light->get_up()[1],
                  light->get_up()[2]);
```

```
        glGetFloatv(GL_TRANSPOSE_MODELVIEW_MATRIX, lightViewMatrixTemp);
        Mat4x4f lightViewMatrix(lightViewMatrixTemp); //convert to Mat4f4
            matrix
    glPopMatrix();
    return lightViewMatrix;
}


void ShadowMapPBuf::prepare_for_texture_render()
{
    glClear(GL_DEPTH_BUFFER_BIT);

    glDepthFunc(GL_LESS);
    //Draw back faces into the shadow map
    glEnable(GL_CULL_FACE);
    glCullFace(GL_BACK);

    //Disable color writes, and use flat shading for speed
    glShadeModel(GL_FLAT);
    glColorMask(0,0,0,0);

    //Disable Lighting
    glDisable(GL_LIGHTING);

    glEnable(GL_POLYGON_OFFSET_POINT );
    glEnable(GL_POLYGON_OFFSET_FILL);
    glPolygonOffset(factor, units);

    char buf[50];
    sprintf(buf, "Bias: ON     Factor: %.1f     Units: %.1f", factor, units)
        ;
    displaytext.updateText(biastext, buf);
}

void ShadowMapPBuf::restore_states()
{
    //restore states
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_CULL_FACE);
    glShadeModel(GL_SMOOTH);
    glColorMask(1, 1, 1, 1);

    glViewport(0, 0, vpWidth, vpHeight);
}

void ShadowMapPBuf::copy_drawing_buffer_to_texture(GLuint tex_map)
{
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, tex_map);
    glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, 0, 0,map_size,
        map_size, 0); //GL_DEPTH_COMPONENT
}

void ShadowMapPBuf::init_shadow_texture()
{
    for(int i=0; i<lights->no_lights(); i++)
    {
        fprintf(stdout, "\nInitialize RenderTexture\n");

        rt[i] = new RenderTexture();
        rt[i]->Reset("rgba=0 depthTex2D depth rtt");
```

```
        rt[i]->Initialize(map_size,map_size);

        if(!rt[i]->IsInitialized())
        {
            fprintf(stderr, "Error initializing RenderTexture");
            system("PAUSE");
            exit(0);
        }

        rt[i]->BindDepth();
        if(pcf){
            glTexParameteri(rt[i]->GetTextureTarget(), GL_TEXTURE_MIN_FILTER
                , GL_LINEAR); //GL_LINEAR
            glTexParameteri(rt[i]->GetTextureTarget(), GL_TEXTURE_MAG_FILTER
                , GL_LINEAR);
        }
        else{
            glTexParameteri(rt[i]->GetTextureTarget(), GL_TEXTURE_MIN_FILTER
                , GL_NEAREST); //GL_LINEAR
            glTexParameteri(rt[i]->GetTextureTarget(), GL_TEXTURE_MAG_FILTER
                , GL_NEAREST);
        }
        fprintf(stdout, "-> RenderTexture initialized OK\n");
        fprintf(stdout, "-> Size: %dx%d\n", map_size, map_size);
        fprintf(stdout, "-> IsDepthTexture %d\n", rt[i]->IsDepthTexture());
        fprintf(stdout, "-> IsTexture %d\n", rt[i]->IsTexture());
        fprintf(stdout, "-> IsFloatTexture %d\n", rt[i]->IsFloatTexture());
        fprintf(stdout, "-> IsDoubleBuffered %d\n", rt[i]->IsDoubleBuffered
            ());
    }
}

void ShadowMapPBuf::show_shadow_map(int shadowmap)
{
    glColor3f(1,1,1);
    rt[shadowmap]->BindDepth();
    rt[shadowmap]->EnableTextureTarget();

    int maxs = rt[shadowmap]->GetMaxS();
    int maxt = rt[shadowmap]->GetMaxT();

    glDisable(GL_LIGHTING);
    glViewport(0+shadowmap*200,0,200,200);

    glDisable(GL_TEXTURE_GEN_Q);
    glDisable(GL_TEXTURE_GEN_R);
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);

    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(-1,1,-1,1);

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();

    glBegin(GL_POLYGON);
        glTexCoord2f(0.0, 0.0);
        glVertex2f(-1,-1);
        glTexCoord2f(0.0, maxt);
```

```
        glVertex2f(-1, 1);
        glTexCoord2f(maxs, maxt);
        glVertex2f(1, 1);
        glTexCoord2f(maxs, 0.0);
        glVertex2f(1, -1);
    glEnd();

    rt[shadowmap]->DisableTextureTarget();

    glMatrixMode( GL_PROJECTION );
    glPopMatrix();

    glMatrixMode( GL_MODELVIEW );
    glPopMatrix();

    glViewport(0,0,600,600);
}

void ShadowMapPBuf::show_misc_info(int window){}
```

C.2 Perspective Shadow Maps

```
/*************************************************************/
/* filename: PerspectiveShadowMapv2.h                        */
/*************************************************************/
#ifndef __PERSHA_MAP_H___
#define __PERSHA_MAP_H___

#include "My/RenderAlgorithm.h"
#include "Components/Timer.h"

class PerspectiveShadowMapCF : public RenderAlgorithm
{
protected:
    vector<RenderTexture*> rt;
    int map_size;
    vector<Camera*> virt_cam;
    Light* trans_light[100];
    vector<bbox> bb_view;
    vector<bbox> bb_all;
    vector<bbox> bb_light_see;

    // For text
    int slidebacktext;
    int cubeclippingtext;
    int lightbehindtext;
    int biastext;
    int mapsizetext;

    // For biasing
    float factor, units;

    //Matrices
    GLfloat lightProjectionMatrixTemp[16], lightViewMatrixTemp[16];
    GLfloat cameraProjectionMatrixTemp[16], cameraViewMatrixTemp[16];
    Mat4x4f biasMatrix;
    vector<Mat4x4f> M_c;
    vector<Mat4x4f> MV_c;
    vector<Mat4x4f> P_c;
    Mat4x4f texture_matrix;

    bool show_map;
```

```
    bool cube_clip;
    bool pcf;
    bool b_slide_back;
    bool bSeeFromLight, bNearFarUCube, bUpdateVNearAndFar;

    Components::Timer timer;

#ifdef REC_TIME
    TimeToExcel* exc_timer;
#endif

public:
    PerspectiveShadowMapCF(Object* _obj, LightContainer* _lights, Camera*
        _cam, Object* _floor) : RenderAlgorithm(_obj, _lights, _cam, _floor)
        {
        map_size = 1024;

        cube_clip = true;
        pcf = true;
        b_slide_back = false;
        bSeeFromLight = false;
        bNearFarUCube = false;       //Standard false
        bUpdateVNearAndFar = true;   //Standard true

        init();
        rt.resize(lights->no_lights());
        virt_cam.resize(lights->no_lights());
        bb_view.resize(lights->no_lights());
        bb_all.resize(lights->no_lights());
        bb_light_see.resize(lights->no_lights());
        M_c.resize(lights->no_lights());
        MV_c.resize(lights->no_lights());
        P_c.resize(lights->no_lights());

        for(int i=0; i<lights->no_lights(); i++)
            init_shadow_texture(i);

        show_map = true;
        biasMatrix = Mat4x4f(Vec4f(0.5, 0.0, 0.0, 0.5),
                             Vec4f(0.0, 0.5, 0.0, 0.5),
                             Vec4f(0.0, 0.0, 0.5, 0.5),
                             Vec4f(0.0, 0.0, 0.0, 1.0));
        slidebacktext = displaytext.addText("");
        cubeclippingtext = displaytext.addText("");
        lightbehindtext = displaytext.addText("");
        biastext = displaytext.addText("");
        mapsizetext = displaytext.addText("");

        factor = 1.7;
        units = 4.0;

        char buf[50];
        sprintf(buf, "Mapsize: %dx%d", map_size, map_size);
        displaytext.updateText(mapsizetext, buf);

#ifdef REC_TIME
        // For Time Reporting
        exc_timer = new TimeToExcel("./output/PSMOutput.txt");
        exc_timer->add_title_item("Calculate Transformation Matrix M_c");
        exc_timer->add_title_item("Setup Virtual Camera");
        exc_timer->add_title_item("Create Shadow Map");
        exc_timer->add_title_item("Final Shading");
```

```
        exc_timer->write_title_line();
#endif
    };

    void render();
    void init();
    void init_shadow_texture(int l);
    void show_shadow_map(GLuint tex_map);
    void show_shadow_map(int i);
    void show_misc_info(int window, int i);
    void copy_drawing_buffer_to_texture(GLuint tex_map);
    void prepare_for_texture_render(int i);
    void enable_r_to_texture_comparison(GLuint tex_map);
    void enable_r_to_texture_comparison(int i);
    void reset_all();
    void restore_states(int i);
    Mat4x4f calculate_texture_matrix(Light* light, int i);
    Mat4x4f get_light_projection_matrix(Light* light);
    Mat4x4f get_light_view_matrix(Light* light);
    Mat4x4f get_cam_projection_matrix(Camera* cam);
    Mat4x4f get_cam_view_matrix(Camera* cam);
    Mat4x4f transpose_matrix(Mat4x4f mat);
    void generate_texture_coordinates(Mat4x4f* texture_matrix);

    void process_key(unsigned char key)
    {
        Vec3f p0;
        Vec3f p7;
        switch(key) {
            case 'o':
                obj->get_bbox(p0, p7);
                for(int i=0; i<lights->no_lights(); i++)
                    lights->get_light(i)->calculate_near_far(bbox(p0,p7));
                break;
            case 'j':
                b_slide_back = !b_slide_back;
                break;
            case 'p':
                pcf = !pcf;
                for(int i=0; i<lights->no_lights(); i++)
                    init_shadow_texture(i);
                break;
            case 'c':
                cube_clip = !cube_clip;
                break;
            case 'g':
                trans_light[0]->print_all();
                break;
            case 43: //Ascii +
                increase_map_size();
                break;
            case 45: //Ascii -
                decrease_map_size();
                break;
            case 'y':
                increase_bias_factor(0.1);
                break;
            case 'h':
                decrease_bias_factor(0.1);
                break;
            case 'Y':
                increase_bias_units(0.1);
```

```
                break;
            case 'H':
                decrease_bias_units(0.1);
                break;
            case 'u':
                bSeeFromLight = !bSeeFromLight;
                break;
            case 'k':
                bNearFarUCube = !bNearFarUCube;
                cout << "Near And Far using Unit Cube: " << bNearFarUCube
                     << endl;
                break;
            case 'l':
                bUpdateVNearAndFar = !bUpdateVNearAndFar;
                break;
            default:
                RenderAlgorithm::process_key(key);
                break;
        }
    }

    /* set functions */
    void set_map_size(int i){
        map_size = i;
        for(int l=0; l<lights->no_lights(); l++){
            delete rt[l];
            init_shadow_texture(l);
        }
        char buf[50];
        sprintf(buf, "Mapsize: %dx%d", map_size, map_size);
        displaytext.updateText(mapsizetext, buf);
    }
    void increase_map_size(){
        if(map_size < 2048){
            map_size *= 2;
            for(int l=0; l<lights->no_lights(); l++){
                delete rt[l];
                init_shadow_texture(l);
            }
            char buf[50];
            sprintf(buf, "Mapsize: %dx%d", map_size, map_size);
            displaytext.updateText(mapsizetext, buf);
        }
    }
    void decrease_map_size(){
        map_size /= 2.0;
        for(int l=0; l<lights->no_lights(); l++){
            delete rt[l];
            init_shadow_texture(l);
        }
        char buf[50];
        sprintf(buf, "Mapsize: %dx%d", map_size, map_size);
        displaytext.updateText(mapsizetext, buf);
    }

    /* print functions */
    void print_all()
    {
        RenderAlgorithm::print_all();
        cout << "Map Size: " << map_size << endl;
    }
```

```
    Camera* get_virt_cam(int i){return virt_cam[i];}
    Light* get_trans_light(int i){return trans_light[i];}
    void increase_bias_factor(float amount=0.1){factor += amount;}
    void decrease_bias_factor(float amount=0.1){factor -= amount;}
    void increase_bias_units(float amount=0.1){units += amount;}
    void decrease_bias_units(float amount=0.1){units -= amount;}
};
#endif

/************************************************************/
/* filename: PerspectiveShadowMapv2.cpp                     */
/************************************************************/
#include "My/PerspectiveShadowMapv2.h"

void PerspectiveShadowMapCF::init()
{
    GLenum err = glewInit();
    if (GLEW_OK != err)
    {
        fprintf(stderr, "Error loading GLEW");
        system("PAUSE");
    }
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glShadeModel (GL_SMOOTH);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective(cam->get_fov(), cam->get_aspect(), cam->get_near(),cam->
        get_far());

    /* Enable depth test and stencil test */
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_STENCIL_TEST);
    //glEnable(GL_NORMALIZE);

    glDepthFunc(GL_LEQUAL);
    glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );

    /* Enable lighting */
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    RenderAlgorithm::init();
}

void PerspectiveShadowMapCF::render()
{
    clear_all();
    bbox bb_temp;
#ifdef REC_TIME
    exc_timer->start();
#endif
    for(int l=0; l<lights->no_lights(); l++){

        lights->get_light(l)->set_matrices();
        cam->set_matrices();
        /* 1. First of all we set up a virtual camera at the camera position
             */
        /* This cameras frustum should contain all objects casting shadow
            into the scene */
        virt_cam[l] = new Camera(cam->get_position(), cam->get_direction(),
            cam->get_up(), cam->get_fov());
        virt_cam[l]->set_matrices();
```

```
bb_view[l].clear();
bb_light_see[l].clear();
bb_all[l] = bbox(obj);
bb_all[l].add_bbox(bbox(floor));

/*********************************************************************/

/* Generating bounding box for use in cube-clipping
                          */
/*********************************************************************/

Vec3f p0, p7;                     // Vectors used to hold the max and
    min values of the bounding box from TriMesh
bool obj_in_frustum = false;      // A boolean used to check whether
    or not any bounding boxes has been added to
                                  // bb_view
for(int i=0; i<obj->no_meshes(); i++) {
    obj->get_mesh(i)->get_bbox(p0,p7);                    // We
        get the bounding box of each mesh
    bbox bb(p0,p7);                                       //
        Create a bbox object
    if(MyCGLA::bbox_in_frustum(&bb, lights->get_light(l))){ // if
        this bounding box is inside the light frustum
        bb_light_see[l].add_bbox(bb);
        if(MyCGLA::bbox_in_frustum_M(&bb, cam)){          // and
            the camera frustum
            bb_view[l].add_bbox(bb);                      // we
                add it to the bb_view bounding box
            obj_in_frustum = true;                        // and
                sets the boolean, indicating there was an object, to
                 true
        }
    }
}
// The floor is taken into account
BMesh::TriMesh* mesh = floor->get_mesh(0);
for(int i=0; i<mesh->no_vertices(); i++){
    Vec3f p = mesh->get_vertex(i);
    if(MyCGLA::point_in_frustum(p,lights->get_light(l))){
        bb_light_see[l].add_point(p);
        if(MyCGLA::point_in_frustum_M(p,virt_cam[l])){
            bb_view[l].add_point(p);
            obj_in_frustum = true;
        }
    }
}

bb_temp = bb_light_see[0];

if(bUpdateVNearAndFar){
    if(bb_view[l].contains_multiple_points()){        // If any
        objects were in the frustum of both the light and the camera
        virt_cam[l]->update_near_far(bb_view[l]);     // We update
            the near and far plane to only just encompas these
    }
}

// We calculate the value of Z_inf = Z_f / (Z_f - Z_n)
float Z_inf = virt_cam[l]->get_far() / (virt_cam[l]->get_far() -
    virt_cam[l]->get_near());
```

```
        if(b_slide_back){
            // If Z_inf is smaller than 1.5 we slide the camera back as done
                in PSM C&F
            if(Z_inf < 1.5){
                virt_cam[l]->slide_back();
                Z_inf = virt_cam[l]->get_far() / (virt_cam[l]->get_far() -
                    virt_cam[l]->get_near());
            }
            char buf[50];
            sprintf(buf, "SlideBack: ON      Z_inf: %.2f", Z_inf);
            displaytext.updateText(slidebacktext, buf);
        }
        else
        {
            char buf[50];
            sprintf(buf, "SlideBack: OFF      Z_inf: %.2f", Z_inf);
            displaytext.updateText(slidebacktext, buf);
        }

        P_c[l] = get_cam_projection_matrix(virt_cam[l]); // The projection
            and modelview matrix of the virtual camera is now
        MV_c[l] = get_cam_view_matrix(virt_cam[l]);     // multiplied to give
            us the transformation matrix for the shadow map
        M_c[l] = P_c[l] * MV_c[l];
    }
#ifdef REC_TIME
    exc_timer->add_time();
#endif

    /* For all lights */
    /* 2. Render the scene as seen from the light */
    /* Multiply with new M_c matrix */
    for(int i=0; i<lights->no_lights(); i++){

        Light* light = lights->get_light(i);

        trans_light[i] = new SpotLight(light);          // A new spotlight
            is created to be used for transformation
        trans_light[i]->set_position(multM4V3(&M_c[i], &trans_light[i]->
            get_position())); // The position is transformed

        // We now has to set the direction and "fov" for the transformed
            light.
        // This can be done by setting it to encompas the unit cube, or even
            better
        // by setting the direction and the "fov" using the transformed
            bb_view bounding box,
        // and the near and far plane using the bounding box of all shadow
            casters aka. obj bounding box

        // Cube clipping. Frustum optimized for transformed bounding box, if
            any
        if(cube_clip){ //obj_in_frustum){
            bb_view[i].transform(M_c[i]);
            bb_all[i].transform(M_c[i]);
            bb_light_see[i].transform(M_c[i]);

            trans_light[i]->set_direction(multM4V3( &M_c[i], &trans_light[i
                ]->get_direction()));
            trans_light[i]->calculate_fustum(bb_view[i]);
            draw_bbox(bb_view[i]);
            show_light_frustum(trans_light[i]);
```

```
            trans_light[i]->calculate_near_far(bb_light_see[i]);

            char buf2[50];
            sprintf(buf2, "CubeClipping: ON      Light_cam fov: %.2f",
                trans_light[i]->get_cutoff_angle() * 2.0);
            displaytext.updateText(cubeclippingtext, buf2);
        }
        else{ //unit cube used for frustum calculation
            bb_light_see[i].transform(M_c[i]);
            trans_light[i]->set_direction(multM4V3( &M_c[i], &trans_light[i
                ]->get_direction()));
            trans_light[i]->calculate_fustum(bbox(Vec3f(1,1,1), Vec3f
                (-1,-1,-1)));
            if(!bNearFarUCube){
                trans_light[i]->calculate_near_far(bb_light_see[i]);}
            else{
                trans_light[i]->calculate_near_far(bbox(Vec3f(1,1,1), Vec3f
                    (-1,-1,-1)));}

            char buf[50];
            sprintf(buf, "CubeClipping: OFF      Light_cam fov: %.2f",
                trans_light[i]->get_cutoff_angle() * 2.0);
            displaytext.updateText(cubeclippingtext, buf);
        }

        if(MyCGLA::light_behind_camera(light, virt_cam[i])){
            float temp_far = trans_light[i]->get_far();
            trans_light[i]->set_far(trans_light[i]->get_near());
            trans_light[i]->set_near(-temp_far);
            char buf[50];
            sprintf(buf, "Light behind camera...");
            displaytext.updateText(lightbehindtext, buf);
        }
        else
        {
            char buf[50];
            sprintf(buf, "Light in front of camera...");
            displaytext.updateText(lightbehindtext, buf);
        }

#ifdef REC_TIME
        exc_timer->add_time();
#endif

#ifdef SHOW_MISC_INFO
        cout << "Virtcam: " << endl;
        cout << "-> Near:   " << trans_light[i]->get_near() << endl;
        cout << "-> Far:    " << trans_light[i]->get_far() << endl;
        cout << "-> Diff    " << trans_light[i]->get_far() - trans_light[i]->
            get_near() << endl;
        cout << "-> Angle: " << trans_light[i]->get_cutoff_angle()*2 << endl
            ;
#endif
        Camera* light_cam = new Camera(trans_light[i]->get_position(),
                                       trans_light[i]->get_direction(),
                                       trans_light[i]->get_up(),
                                       trans_light[i]->get_cutoff_angle()
                                           *2.0,
                                       trans_light[i]->get_near(),
                                       trans_light[i]->get_far());
        rt[i]->BindDepth();
        rt[i]->EnableTextureTarget();
```

```
        rt[i]->BeginCapture();
        glClearColor(1.0, 0.0, 0.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT|
            GL_STENCIL_BUFFER_BIT);
        glEnable(GL_DEPTH_TEST);
        glViewport(0,0,map_size,map_size);
        prepare_for_texture_render(i);
        load_projection(light_cam);
        load_modelview(light_cam);
        update_lights();
        glMultMatrixf(CGLA::transpose(M_c[i]).get());
        glCallList(dlDrawScene);
        rt[i]->EndCapture();
        restore_states(i);
        delete(light_cam);
    }
#ifdef REC_TIME
    exc_timer->add_time();
#endif
    clear_all();
    load_projection(cam);
    load_modelview(cam);
    /* 3. Render the scene from the camera at lit parts */
    /*    The depth-value seen from the camera is transformed to light-space
           and compared to the depth-value from the texture map. */
    for(int i=0; i<lights->no_lights(); i++)
    {
        texture_matrix = calculate_texture_matrix(trans_light[i], i);
        generate_texture_coordinates(&texture_matrix);
        enable_r_to_texture_comparison(i);
        rt[i]->EnableTextureTarget();
        lights->turn_all_off();
        lights->get_light(i)->turn_on();
        update_lights();
        glCallList(dlDrawScene);
        rt[i]->DisableTextureTarget();
        glEnable(GL_BLEND);
        glBlendFunc(GL_ONE, GL_ONE);
    }
    glDisable(GL_BLEND);

#ifdef REC_TIME
    exc_timer->add_time();
    exc_timer->end_line();
#endif
    /* Draw misc info */
#ifdef SHOW_MISC_INFO
    for(int i=0; i<lights->no_lights(); i++)
        show_shadow_map(i);
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    load_projection(cam);
    load_modelview(cam);
    //show_cam_frustum(virt_cam[0]);
    draw_bbox(bb_temp);
#endif

    RenderAlgorithm::render();

    /* reset all */
    reset_all();

    if(bSeeFromLight){
```

```
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        Camera* fromLight = new Camera(lights->get_light(0)->get_position(),
                                       lights->get_light(0)->get_direction()
                                           ,
                                       lights->get_light(0)->get_up(),
                                       lights->get_light(0)->
                                           get_cutoff_angle()*2.0);

        load_projection(fromLight);
        load_modelview(fromLight);
        lights->turn_all_on();
        update_lights();
        glCallList(dlDrawScene);
        delete(fromLight);
    }
}

void PerspectiveShadowMapCF::reset_all()
{
    //Disable textures and texgen
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
    glDisable(GL_TEXTURE_GEN_R);
    glDisable(GL_TEXTURE_GEN_Q);

    //Restore other states
    glDisable(GL_LIGHTING);
    glDisable(GL_ALPHA_TEST);
}

void PerspectiveShadowMapCF::enable_r_to_texture_comparison(GLuint tex_map)
{
    //Bind & enable shadow map texture
    glBindTexture(GL_TEXTURE_2D, tex_map);
    glEnable(GL_TEXTURE_2D);
    //Enable shadow comparison
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE_ARB,
        GL_COMPARE_R_TO_TEXTURE);
    //Shadow comparison should be true (ie not in shadow) if r<=texture
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC_ARB, GL_LEQUAL);
}

void PerspectiveShadowMapCF::enable_r_to_texture_comparison(int i)
{
    //Bind & enable shadow map texture
    rt[i]->BindDepth();
    glEnable(GL_TEXTURE_2D);

    //Enable shadow comparison
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE_ARB,
        GL_COMPARE_R_TO_TEXTURE);

    //Shadow comparison should be true (ie not in shadow) if r<=texture
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC_ARB, GL_LEQUAL);
}

void PerspectiveShadowMapCF::generate_texture_coordinates(Mat4x4f*
    texture_matrix)
{
    //Set up texture coordinate generation.
    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
```

```
    glTexGenfv(GL_S, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[0].get()));
    glEnable(GL_TEXTURE_GEN_S);

    glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_T, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[1].get()));
    glEnable(GL_TEXTURE_GEN_T);

    glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_R, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[2].get()));
    glEnable(GL_TEXTURE_GEN_R);

    glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_Q, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[3].get()));
    glEnable(GL_TEXTURE_GEN_Q);
}

Mat4x4f PerspectiveShadowMapCF::calculate_texture_matrix(Light* light, int i
    )
{
    Mat4x4f light_projection_matrix = get_light_projection_matrix(light);
    Mat4x4f light_view_matrix = get_light_view_matrix(light);

    return biasMatrix * light_projection_matrix * light_view_matrix * M_c[i
        ];
}

Mat4x4f PerspectiveShadowMapCF::get_light_projection_matrix(Light* light)
{
    glMatrixMode (GL_PROJECTION);
    glPushMatrix();
        glLoadIdentity ();
        gluPerspective(light->get_cutoff_angle()*2.0, 1, light->get_near(),
            light->get_far());
        glGetFloatv(GL_TRANSPOSE_PROJECTION_MATRIX,
            lightProjectionMatrixTemp);
        Mat4x4f lightProjectionMatrix(lightProjectionMatrixTemp);
    glPopMatrix();
    return lightProjectionMatrix;
}

Mat4x4f PerspectiveShadowMapCF::get_light_view_matrix(Light* light)
{
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
        glLoadIdentity();
        gluLookAt(light->get_position()[0],
                  light->get_position()[1],
                  light->get_position()[2],
                  light->get_direction()[0],
                  light->get_direction()[1],
                  light->get_direction()[2],
                  light->get_up()[0],
                  light->get_up()[1],
                  light->get_up()[2]);
        glGetFloatv(GL_TRANSPOSE_MODELVIEW_MATRIX, lightViewMatrixTemp);
        Mat4x4f lightViewMatrix(lightViewMatrixTemp); //convert to Mat4f4
            matrix
    glPopMatrix();
    return lightViewMatrix;
}

Mat4x4f PerspectiveShadowMapCF::get_cam_projection_matrix(Camera* _cam)
```

```
{
    glMatrixMode (GL_PROJECTION);
    glPushMatrix();
        glLoadIdentity();
        gluPerspective(_cam->get_fov(), _cam->get_aspect(), _cam->get_near()
            , _cam->get_far());
        glGetFloatv(GL_TRANSPOSE_PROJECTION_MATRIX,
            cameraProjectionMatrixTemp);
        Mat4x4f cameraProjectionMatrix(cameraProjectionMatrixTemp);
    glPopMatrix();
    return cameraProjectionMatrix;
}

Mat4x4f PerspectiveShadowMapCF::get_cam_view_matrix(Camera* _cam)
{
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
        glLoadIdentity();
        gluLookAt(_cam->get_position()[0],
                  _cam->get_position()[1],
                  _cam->get_position()[2],
                  _cam->get_direction()[0],
                  _cam->get_direction()[1],
                  _cam->get_direction()[2],
                  _cam->get_up()[0],
                  _cam->get_up()[1],
                  _cam->get_up()[2]);
        glGetFloatv(GL_TRANSPOSE_MODELVIEW_MATRIX, cameraViewMatrixTemp);
        Mat4x4f cameraViewMatrix(cameraViewMatrixTemp);
    glPopMatrix();
    return cameraViewMatrix;
}


void PerspectiveShadowMapCF::prepare_for_texture_render(int i)
{
    glViewport(0, 0, map_size, map_size);
    glEnable(GL_CULL_FACE);
    if(MyCGLA::light_behind_camera(lights->get_light(i), virt_cam[i]))
        glCullFace(GL_BACK);
    else
        glCullFace(GL_FRONT);

    glEnable(GL_POLYGON_OFFSET_POINT );
    glEnable(GL_POLYGON_OFFSET_FILL);
    glPolygonOffset(factor, units);

    char buf[50];
    sprintf(buf, "Bias: ON      Factor: %.1f      Units: %.1f", factor, units)
        ;
    displaytext.updateText(biastext, buf);
}

void PerspectiveShadowMapCF::restore_states(int i)
{
    //restore states
    rt[i]->DisableTextureTarget();
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_CULL_FACE);
    glShadeModel(GL_SMOOTH);
    glColorMask(1, 1, 1, 1);
```

```
    glViewport(0, 0, vpWidth, vpHeight);
}

void PerspectiveShadowMapCF::copy_drawing_buffer_to_texture(GLuint tex_map)
{
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, tex_map);
    glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, 0, 0, map_size,
        map_size, 0); //GL_DEPTH_COMPONENT
}

void PerspectiveShadowMapCF::init_shadow_texture(int l)
{
    fprintf(stdout, "\nInitialize RenderTexture\n");
    rt[l] = new RenderTexture();
    rt[l]->Reset("depthTex2D");
    rt[l]->Initialize(map_size, map_size);
    if(!rt[l]->IsInitialized())
    {
        fprintf(stderr, "Error initializing RenderTexture");
        system("PAUSE");
        //exit(0);
    }

    rt[l]->BindDepth();
    if(pcf){
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    }
    else{
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    }

    fprintf(stdout, "-> RenderTexture initialized OK\n");
    fprintf(stdout, "-> ID: %d\n", rt[l]->GetDepthTextureID());
    fprintf(stdout, "-> Target: %d\n", rt[l]->GetTextureTarget());
    fprintf(stdout, "-> Size: %dx%d\n", map_size, map_size);
    fprintf(stdout, "-> IsDepthTexture %d\n", rt[l]->IsDepthTexture());
    fprintf(stdout, "-> IsTexture %d\n", rt[l]->IsTexture());
    fprintf(stdout, "-> IsFloatTexture %d\n", rt[l]->IsFloatTexture());
    fprintf(stdout, "-> IsDoubleBuffered %d\n", rt[l]->IsDoubleBuffered());
}

void PerspectiveShadowMapCF::show_shadow_map(GLuint tex_map)
{
    glViewport(0+150*(tex_map-1),0,150,150);

    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(-1,1,-1,1);

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();

    glBindTexture(GL_TEXTURE_2D, tex_map);
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE_ARB, GL_NONE);
    glEnable(GL_TEXTURE_2D);
    glDisable(GL_TEXTURE_GEN_S);
```

```
    glDisable(GL_TEXTURE_GEN_R);
    glDisable(GL_TEXTURE_GEN_T);
    glDisable(GL_TEXTURE_GEN_Q);
    glDisable(GL_LIGHTING);
    glBegin(GL_POLYGON);
        glColor3f(0.0, 1.0, 0);
        glTexCoord2f(0.0, 0.0);
        glVertex2f(-1,-1);
        glTexCoord2f(0.0, 1.0);
        glVertex2f(-1, 1);
        glTexCoord2f(1.0, 1.0);
        glVertex2f(1, 1);
        glTexCoord2f(1.0, 0.0);
        glVertex2f(1, -1);
    glEnd();
    glEnable(GL_LIGHTING);
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glDisable(GL_TEXTURE_2D);

    glMatrixMode( GL_PROJECTION );
    glPopMatrix();

    glMatrixMode( GL_MODELVIEW );
    glPopMatrix();

    glViewport(0,0,vpWidth,vpHeight);

}

void PerspectiveShadowMapCF::show_shadow_map(int i)
{
    glColor3f(0,1,1);
    rt[i]->BindDepth();
    rt[i]->EnableTextureTarget();
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
    glTexParameteri(rt[i]->GetTextureTarget(), GL_TEXTURE_COMPARE_MODE_ARB,
        GL_NONE);

    int maxs = rt[i]->GetMaxS();
    int maxt = rt[i]->GetMaxT();

    glDisable(GL_LIGHTING);
    glViewport(0+i*200,0,200,200);

    glDisable(GL_TEXTURE_GEN_Q);
    glDisable(GL_TEXTURE_GEN_R);
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);

    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(-1,1,-1,1);

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();

    glBegin(GL_POLYGON);
        glTexCoord2f(0.0, 0.0);
        glVertex2f(-1,-1);
        glTexCoord2f(0.0, maxt);
```

```
        glVertex2f(-1, 1);
        glTexCoord2f(maxs, maxt);
        glVertex2f(1, 1);
        glTexCoord2f(maxs, 0.0);
        glVertex2f(1, -1);
    glEnd();

    rt[i]->DisableTextureTarget();

    glMatrixMode( GL_PROJECTION );
    glPopMatrix();

    glMatrixMode( GL_MODELVIEW );
    glPopMatrix();

    glViewport(0,0,vpWidth,vpHeight);
}

void PerspectiveShadowMapCF::show_misc_info(int window, int i)
{
    glutSetWindow(window);
    clear_all();
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);
    glViewport(0,0,500,500);

    Camera* cube_cam = new Camera(Vec3f(-5,5,-5), Vec3f(0,0,0), Vec3f(0,1,0)
        , 45);

    load_projection(cube_cam);
    load_modelview(cube_cam);


    draw_unit_cube();
    show_light_frustum(trans_light[i]);


    glMultMatrixf(CGLA::transpose(M_c[i]).get());

    draw_bbox(bb_view[i]);
    update_lights();
    glCallList(dlDrawScene);
    glutSwapBuffers();
    glutSetWindow(1);
}

Mat4x4f PerspectiveShadowMapCF::transpose_matrix(Mat4x4f mat)
{
    Mat4x4f mat_t;
    for(int i=0; i<4; i++){
        for(int j=0; j<4; j++){
            mat_t[i][j] = mat[j][i];
        }
    }
    return mat_t;
}
```

C.3 Shadow Volumes (Z-Fail)

```
/*****************************************************************/
/* filename: ShadowVolumeZFail.h                                 */
/*****************************************************************/
```

```
#ifndef __SHA_VOLZF_H__
#define __SHA_VOLZF_H__

#include "My/RenderAlgorithm.h"

class ShadowVolumeZFail : public RenderAlgorithm
{
private:
    vector<Vec3i> adjacent;
    bool draw_sv;
    float P_inf[16];
    int shavoltext;
#ifdef REC_TIME
    TimeToExcel* exc_timer;
#endif
    bool bTwoSideStencil;
    Components::Timer SVtimer;
    DisplayText* SVdt;
    int dtSV;

public:
    ShadowVolumeZFail(Object* _obj, LightContainer* _lights, Camera* _cam,
        Object* _floor) : RenderAlgorithm(_obj, _lights, _cam, _floor){

        SVdt = new DisplayText();
        dtSV = SVdt->addText("");

        draw_sv = false;
        shavoltext = displaytext.addText("");
        char buf[50];
        sprintf(buf, "Draw ShadowVolumes(t): OFF");
        displaytext.updateText(shavoltext, buf);

        /* We make a list of adjacent faces */
        init();

        glMatrixMode (GL_PROJECTION);
        glPushMatrix();
        glLoadIdentity();
        gluPerspective(cam->get_fov(), cam->get_aspect(), cam->get_near(),
            cam->get_far());
        glGetFloatv(GL_PROJECTION_MATRIX, P_inf);
        glPopMatrix();
        P_inf[14] = -2.0*cam->get_near();
        P_inf[10] = P_inf[11] = -1.0;

        // For Time Reporting
#ifdef REC_TIME
        exc_timer = new TimeToExcel("./output/ZFailOutput.txt");
        exc_timer->add_title_item("Render Shadow Volumes");
        exc_timer->add_title_item("Final Shading");
        exc_timer->add_title_item("Misc Info");
        exc_timer->write_title_line();
#endif
        bTwoSideStencil = true;
    };

    ShadowVolumeZFail() : RenderAlgorithm() {
        init();
        glMatrixMode (GL_PROJECTION);
        glPushMatrix();
        glLoadIdentity();
```

```
        gluPerspective(cam->get_fov(), cam->get_aspect(), cam->get_near(),
            cam->get_far());
        glGetFloatv(GL_PROJECTION_MATRIX, P_inf);
        glPopMatrix();
        P_inf[14] = -2.0*cam->get_near();
        P_inf[10] = P_inf[11] = -1.0;
    };

    void init();
    void render();
    void make_adjacent_list();
    void draw_shadow_volumes(Light* light);
    bool is_backfacing(Light* light, int mesh_index, int face_index);
    void shadow_volumes_on(){draw_sv = true;}
    void shadow_volumes_off(){draw_sv = false;}
    void shadow_volumes_toggle(){
        draw_sv = !draw_sv;
        cout << "draw_sv : " << draw_sv << endl;
    }
    void process_key(unsigned char key)
    {
        Vec3f p0;
        Vec3f p7;
        switch(key) {
            case 't':
                shadow_volumes_toggle();
                break;
            case 'b':
                bTwoSideStencil = !bTwoSideStencil;
                break;
            default:
                RenderAlgorithm::process_key(key);
                break;
        }
    }

    float* getPinf(){return P_inf;}
};
#endif


/*****************************************************************/
/* filename: ShadowVolumeZFail.cpp                               */
/*****************************************************************/
#include "My/ShadowVolumeZFail.h"

void ShadowVolumeZFail::render()
{
#ifdef REC_TIME
    exc_timer->start();
#endif
    timer.start();
    glViewport(0,0,vpWidth,vpHeight);
    /* Clear window */
    clear_all();

    /* Load special projection matrix */
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    glPushMatrix();
    glLoadIdentity();
    gluPerspective(cam->get_fov(), cam->get_aspect(), cam->get_near(), cam->
        get_far());
```

```
glGetFloatv(GL_PROJECTION_MATRIX, P_inf);
glPopMatrix();
P_inf[14] = -2.0*cam->get_near();
P_inf[10] = P_inf[11] = -1.0;
glMultMatrixf(P_inf);
load_modelview(cam);

/* Render scene with ambient colors*/
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
lights->turn_all_off();
update_lights();
//draw_scene();
glCallList(dlDrawScene);

/* Prepare for shadow volume render */
glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );
glDepthMask( GL_FALSE );
glShadeModel(GL_FLAT);
glDisable(GL_LIGHTING);

if(GL_EXT_stencil_two_side && bTwoSideStencil)
{
    glDisable(GL_CULL_FACE);
    glClear(GL_STENCIL_BUFFER_BIT);
    glEnable(GL_STENCIL_TEST);
    glEnable(GL_STENCIL_TEST_TWO_SIDE_EXT);

    for(int i=0; i<lights->no_lights(); i++){
        glActiveStencilFaceEXT(GL_BACK);
        glStencilOp(GL_KEEP,                    // stencil test fail
                    GL_INCR_WRAP_EXT,           // depth test fail
                    GL_KEEP);                   // depth test pass
        glStencilMask(~0);
        glStencilFunc(GL_ALWAYS, 0, ~0);

        glActiveStencilFaceEXT(GL_FRONT);
        glStencilOp(GL_KEEP,                    // stencil test fail
                    GL_DECR_WRAP_EXT,           // depth test fail
                    GL_KEEP);                   // depth test pass
        glStencilMask(~0);
        glStencilFunc(GL_ALWAYS, 0, ~0);

        draw_shadow_volumes(lights->get_light(i));
    }
    glEnable(GL_CULL_FACE);
    glCullFace(GL_BACK);
}
else{
    glEnable(GL_CULL_FACE);
    glClear(GL_STENCIL_BUFFER_BIT);
    glEnable(GL_STENCIL_TEST);
    glStencilFunc(GL_ALWAYS, 0, ~0);

    for(int i=0; i<lights->no_lights(); i++){
        glCullFace(GL_FRONT);
        glStencilOp(GL_KEEP,GL_INCR,GL_KEEP);
        draw_shadow_volumes(lights->get_light(i));

        glCullFace(GL_BACK);
```

```
            glStencilOp(GL_KEEP,GL_DECR,GL_KEEP);
            draw_shadow_volumes(lights->get_light(i));
        }
    }
#ifdef REC_TIME
    exc_timer->add_time();
#endif
    /* draw lit parts */
    glShadeModel(GL_SMOOTH);
    glDepthMask(GL_TRUE);
    glDepthFunc(GL_LEQUAL);
    glColorMask(1,1,1,1);
    glStencilFunc(GL_EQUAL, 0, ~0);        //Shade where stencil = 0
    glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);

    lights->turn_all_on();
    update_lights();
    glCallList(dlDrawScene);

    glDepthFunc(GL_LESS);
    glDisable(GL_BLEND);
    glDisable(GL_STENCIL_TEST);

    if(draw_sv)
    {
        for(int i=0; i<lights->no_lights(); i++){
            glDisable(GL_LIGHTING);
            glEnable(GL_CULL_FACE);
            glDepthFunc(GL_LEQUAL);

            glEnable(GL_POLYGON_OFFSET_LINE);
            glPolygonOffset(-0.5,0.0);
            glDisable(GL_BLEND);
            glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
            glColor3f(1.0, 1.0, 0.0);

            glActiveStencilFaceEXT(GL_BACK);
            glStencilOp(GL_KEEP,              // stencil test fail
                GL_INCR_WRAP_EXT,             // depth test fail
                GL_KEEP);                     // depth test pass
            glStencilMask(~0);
            glStencilFunc(GL_ALWAYS, 0, ~0);

            glActiveStencilFaceEXT(GL_FRONT);
            glStencilOp(GL_KEEP,              // stencil test fail
                GL_DECR_WRAP_EXT,             // depth test fail
                GL_KEEP);                     // depth test pass
            glStencilMask(~0);
            glStencilFunc(GL_ALWAYS, 0, ~0);

            draw_shadow_volumes(lights->get_light(i));

            glEnable(GL_BLEND);
            glBlendFunc(GL_ONE,GL_ONE);
            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
            glColor3f(0.1, 0.1, 0.0);
        }
        char buf[50];
        sprintf(buf, "Draw ShadowVolumes(t): ON");
        displaytext.updateText(shavoltext, buf);
    }
    else
```

```
    {
        char buf[50];
        sprintf(buf, "Draw ShadowVolumes(t): OFF");
        displaytext.updateText(shavoltext, buf);
    }
#ifdef REC_TIME
    exc_timer->add_time();
#endif
    SVdt->draw();
    RenderAlgorithm::render();
#ifdef REC_TIME
    exc_timer->add_time();
    exc_timer->end_line();
#endif
}

void ShadowVolumeZFail::make_adjacent_list()
{
    cout << "Generating adjacent list" << endl;
    //For each mesh
    for(int mesh=0; mesh<obj->no_meshes(); mesh++)
    {
        TriMesh* cur_mesh = obj->get_mesh(mesh);
        if(!cur_mesh->adjInitialized)
        {
            cur_mesh->adjacents.resize(cur_mesh->no_faces());
            //For each face
            for(int i=0; i<cur_mesh->no_faces(); i++)
            {
                int found_tris = 0;
                for(int j=0; j<cur_mesh->no_faces(); j++)
                {
                    if(i != j){
                        // if face(i) shares two vertices with face(j)
                        int shared_vertices = 0;
                        for(int k=0; k<3; k++)
                        {
                            for(int l=0; l<3; l++)
                            {
                                //if(cur_mesh->get_face(i)[k] == cur_mesh->
                                    get_face(j)[l])
                                //  shared_vertices++;
                                if(cur_mesh->get_face(i)[k] == cur_mesh->
                                    get_face(j)[(l+1)%3] && cur_mesh->
                                    get_face(i)[(k+1)%3] == cur_mesh->
                                    get_face(j)[l])
                                {
                                    (cur_mesh->adjacents[i])[k] = j;
                                }
                            }
                        }
                        if(shared_vertices >= 2){
                            (cur_mesh->adjacents[i])[found_tris] = j;
                            found_tris++;
                        }
                    }
                }
            }
            cur_mesh->adjInitialized = true;
        }
        else{
            cout << " -> Already initialized" << endl;
```

```
        }
    }
    cout << " -> done!" << endl;
}

bool ShadowVolumeZFail::is_backfacing(Light* light, int mesh_index, int
    face_index){

        Vec3f face_avg_pos = (obj->get_mesh(mesh_index)->get_face_vertex(
            face_index,0)
                        + obj->get_mesh(mesh_index)->get_face_vertex(
                            face_index,1)
                        + obj->get_mesh(mesh_index)->get_face_vertex(
                            face_index,2))/3.0;
        Vec3f light_dir = light->get_position() - face_avg_pos;
        if(dot(normalize(light_dir), obj->get_mesh(mesh_index)->
            get_face_normal(face_index)) <= 0)
            return true;
        else

        return false;
    }
void ShadowVolumeZFail::draw_shadow_volumes(Light* light) //, TriMesh* TM,
    vector<Light>* lights
{

    Vec4f L = Vec4f(light->get_position(), 1.0);
    //For all meshes
    for(int k=0; k<obj->no_meshes(); k++)
    {
        // For all triangles
        for(int i=0; i<obj->get_mesh(k)->no_faces(); i++)
        {
            TriMesh* mesh = obj->get_mesh(k);
            // If triangle is backfacing
            if(is_backfacing(light, k, i))
            {
                //for each edge
                for(int j=0; j<3; j++)
                {
                    //if adjacent face is front-facing, edge is silhouette
                        edge
                    //int adj = (mesh->adjacents[i])[j];
                    if(!is_backfacing(light, k, (mesh->adjacents[i])[j]))
                    {
                        // then we extrude and draw the edge
                        Vec4f A(mesh->get_face_vertex(i,j), 1);
                        Vec4f B(mesh->get_face_vertex(i,(j+1)%3), 1);
                        Vec4f AL(A[0]*L[3]-L[0]*A[3],
                                    A[1]*L[3]-L[1]*A[3],
                                    A[2]*L[3]-L[2]*A[3], 0);
                        Vec4f BL(B[0]*L[3]-L[0]*B[3],
                                    B[1]*L[3]-L[1]*B[3],
                                    B[2]*L[3]-L[2]*B[3], 0);

                        glBegin(GL_QUADS);
                            glColor4f(1.0, 1.0, 0.0, 0.2);
                            glVertex4fv(A.get());
                            glVertex4fv(B.get());
                            glVertex4fv(BL.get());
                            glVertex4fv(AL.get());
                        glEnd();
```

```
                    }
                }
                glBegin(GL_TRIANGLES);
                for(int v=0; v<3; v++){
                    glColor4f(1.0, 1.0, 0.0, 0.2);
                    Vec4f V = Vec4f(mesh->get_face_vertex(i,v), 1.0);
                    glVertex4f(V[0]*L[3]-L[0]*V[3],
                               V[1]*L[3]-L[1]*V[3],
                               V[2]*L[3]-L[2]*V[3], 0);
                }
                glEnd();
            }
            else // Front facing triangle
            {
                glBegin(GL_TRIANGLES);
                    for(int v=0; v<3; v++){
                        Vec4f V = Vec4f(mesh->get_face_vertex(i,v), 1.0);
                        glVertex4f(V[0], V[1], V[2], V[3]);
                    }
                glEnd();
            }
        }
    }
}

void ShadowVolumeZFail::init()
{
    RenderAlgorithm::init();
    make_adjacent_list();

    glViewport(0,0,vpWidth,vpHeight);
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glShadeModel (GL_SMOOTH);

    /* Enable depth test and stencil test */
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_STENCIL_TEST);

    /* Enable lighting */
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
}
```

C.4 Chan's Hybrid

```
/*****************************************************************/
/* filename: Chan2.h                                             */
/*****************************************************************/
#ifndef __CHAN_H__
#define __CHAN_H__

#include "My/RenderAlgorithm.h"
#include "My/ShadowVolumeZFail.h"
#include <Cg/cg.h>
#include <Cg/cgGL.h>
#include "My/timebox.h"

class Chan : public RenderAlgorithm
{
private:
    // For Creating shadow map
    vector<RenderTexture*> rt;
```

```
    int MapSize;
    float BiasFactor, BiasUnits;
    Mat4x4f BiasMatrix;

    // Silhoutte texture
    vector<GLuint> silTexture;

    // For creating computation mask
    RenderTexture* maskTexture;

    // For CG
    CGcontext Context, maskContext;
    CGprofile fprofile, vprofile, maskprofile;
    CGprofile maskvprofile, shamapvprofile,shamapfprofile;
    CGprogram fProgram, vProgram, maskprogram;
    CGprogram maskvprogram, shamapvprogram, shamapfprogram;
    CGparameter mvp, mv, mvIT, observerEyeToLightClip, lightPosEye;
    const char* fpDir;
    const char* vpDir;
    const char* maskDir;
    const char* maskvDir;
    const char* shamapvDir;
    const char* shamapfDir;

    //For shadow volume
    float P_inf[16];

    // Booleans Controling which misc info to display
    bool bShowMap;
    bool bPCF;
    bool bDPT;
    bool bTwoSideStencil;
    bool bShowScreenQuad;
    bool bShowShaVol;
    bool bShowShaMap;
    bool bTestIntensity;

    // DisplayText
    int dtMapSize;
    int dtBias;

    Components::Timer SVtimer;
    DisplayText* SVdt;
    int dtSV;

    // For time reporting
#ifdef REC_TIME
    TimeToExcel* exc_timer;

#endif
#ifdef SHOW_TIME_BOX
    TimeBox* tb;
#endif
public:
    Chan(Object* _obj, LightContainer* _lights, Camera* _cam, Object* _floor
        ) : RenderAlgorithm(_obj, _lights, _cam, _floor){
        fprintf(stderr, "Chan Object Created\n");
        rt.resize(lights->no_lights());
        for(int i=0; i<lights->no_lights(); i++){
            rt[i] = new RenderTexture();
        }
```

```
        SVdt = new DisplayText();
        dtSV = SVdt->addText("");

        silTexture.resize(lights->no_lights());
        // For Shadow Map
        MapSize = 512;
        BiasFactor = 1.1;
        BiasUnits = 4.0;
        bShowMap = false;
        bPCF = true; //Must be true
        bDPT = true;
        bTwoSideStencil = true;
        bShowScreenQuad = false;
        bShowShaVol = false;
        bShowShaMap = false;
        bTestIntensity = true; //Use INTESITY result or not --- Not use when
             USE_CG defined

        BiasMatrix = Mat4x4f(Vec4f(0.5, 0.0, 0.0, 0.5),
                             Vec4f(0.0, 0.5, 0.0, 0.5),
                             Vec4f(0.0, 0.0, 0.5, 0.5),
                             Vec4f(0.0, 0.0, 0.0, 1.0));

        // For CG
        fpDir       = "findsil_f.cg";
        vpDir       = "findsil_v.cg";
        maskDir     = "setmask_f.cg";
        maskvDir    = "setmask_v.cg";
        shamapvDir = "shadowmap_v.cg";
        shamapfDir = "shadowmap_f.cg";

        init();

        // Misc info
        dtMapSize = displaytext.addText("");
        dtBias = displaytext.addText("");

        // For shadowVolume
        glMatrixMode (GL_PROJECTION);
        glPushMatrix();
        glLoadIdentity();
        gluPerspective(cam->get_fov(), cam->get_aspect(), cam->get_near(),
            cam->get_far());
        glGetFloatv(GL_PROJECTION_MATRIX, P_inf);
        glPopMatrix();
        P_inf[14] = -2.0*cam->get_near();
        P_inf[10] = P_inf[11] = -1.0;

        RenderAlgorithm::init();

#ifdef REC_TIME
        // For Time Reporting
        exc_timer = new TimeToExcel("./output/ChanOutput.txt");
        exc_timer->add_title_item("Create Shadow Map");
        exc_timer->add_title_item("Find Silhoutte (Copy to Texture)");
        exc_timer->add_title_item("Create Mask");
        exc_timer->add_title_item("Render Shadow Volumes");
        exc_timer->add_title_item("Final Shading");
        exc_timer->add_title_item("Misc Info");
        exc_timer->write_title_line();
#endif
#ifdef SHOW_TIME_BOX
```

```
        tb = new TimeBox();
#endif
        InitCG();
    }
    ~Chan()
    {
        cgDestroyProgram(fProgram);
        cgDestroyProgram(vProgram);
        cgDestroyContext(Context);
    }

    void init(){

        fprintf(stderr, "Initializing Chan Object\n");
        make_adjacent_list();
        InitGlew();
        InitRenderTexture();
        RenderAlgorithm::init();
    };

    void InitGlew();
    void InitRenderTexture();
    void InitCG();
    void CheckCgError();

    void make_adjacent_list();
    bool is_backfacing(Light* light, int mesh_index, int face_index);
    void draw_shadow_volumes(Light* light);
    void initShadowVolume();

    void render();
    void prepare_for_texture_render();
    void show_shadow_map(int i);
    Mat4x4f calculate_texture_matrix(Light* light);
    Mat4x4f get_light_projection_matrix(Light* light);
    Mat4x4f get_light_view_matrix(Light* light);
    void generate_texture_coordinates(Mat4x4f* texture_matrix);
    void enable_r_to_texture_comparison(int i);

    void process_key(unsigned char key){
        switch(key) {
            case 'p':
                bPCF = !bPCF;
                break;
            case 't':
                bDPT = !bDPT;
                fprintf(stdout, "t pressed\n");
                break;
            case 'b':
                bTwoSideStencil = !bTwoSideStencil;
                break;
            case 'v':
                bShowScreenQuad = !bShowScreenQuad;
                break;
            case 'n':
                bShowShaVol = !bShowShaVol;
                break;
            case 'c':
                bShowShaMap = !bShowShaMap;
                break;
            case 'm':
                bTestIntensity = !bTestIntensity;
```

```
                break;
            case 43: //Ascii +
                increase_map_size();
                break;
            case 45: //Ascii -
                decrease_map_size();
                break;

            default:
                RenderAlgorithm::process_key(key);
                break;
        }
    }

    void increase_map_size(){
        if(MapSize < 2048){
            MapSize *= 2;
            for(int j=0; j<lights->no_lights(); j++){
                InitRenderTexture();
            }
            cout <<"Mapsize Increased To: " << MapSize << endl;
        }
    }
    void decrease_map_size(){
        MapSize /= 2.0;
        for(int j=0; j<lights->no_lights(); j++)
            delete rt[j];
        init();
        cout <<"Mapsize Decreased To: " << MapSize << endl;
    }
};
#endif

/******************************************************************/
/* filename: Chan2.cpp                                            */
/******************************************************************/
#include "My/Chan2.h"

void Chan::InitGlew(){
    fprintf(stderr, "\nInitializing Glew\n");
    GLenum err = glewInit();
    if (GLEW_OK != err)
    {
        fprintf(stderr, "Error loading GLEW");
        system("PAUSE");
    }
    else
    {
        fprintf(stderr," -> Glew Initialized Succefully\n");
    }
}

void Chan::InitRenderTexture(){
    fprintf(stderr, "\nInitializing RenderTexture\n");

    for(int i=0; i<lights->no_lights();i++){
        rt[i]->Reset("depthTex2D");
        rt[i]->Initialize(MapSize,MapSize);

        if(!rt[i]->IsInitialized())
        {
            fprintf(stderr, "Error initializing RenderTexture");
```

```
            system("PAUSE");
            exit(0);
        }

        rt[i]->BindDepth();
        if(bPCF){
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
                ; //GL_LINEAR
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
                ;
        }
        else{
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST
                );
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST
                );
        }

        fprintf(stdout, " -> RenderTexture initialized OK\n");
        fprintf(stdout, " -> Size: %dx%d\n", MapSize, MapSize);
        fprintf(stdout, " -> IsDepthTexture %d\n", rt[i]->IsDepthTexture());
        fprintf(stdout, " -> IsTexture %d\n", rt[i]->IsTexture());
        fprintf(stdout, " -> IsFloatTexture %d\n", rt[i]->IsFloatTexture());
        fprintf(stdout, " -> IsDoubleBuffered %d\n", rt[i]->IsDoubleBuffered
            ());
    }

    maskTexture = new RenderTexture();
    maskTexture->Reset("tex2D");
    maskTexture->Initialize(MapSize, MapSize);
    if(!maskTexture->IsInitialized())
    {
        fprintf(stderr, "Error initializing MaskTexture");
        system("PAUSE");
        exit(0);
    }
    maskTexture->Bind();
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); //
        GL_LINEAR
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    for(int i=0; i<lights->no_lights();i++){
        glEnable(GL_TEXTURE_RECTANGLE_NV);
        glViewport(0,0,vpWidth,vpHeight);
        glGenTextures(1, &silTexture[i]);
        glBindTexture(GL_TEXTURE_RECTANGLE_NV, silTexture[i]);
        glTexParameteri(GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_MIN_FILTER,
            GL_NEAREST);
        glTexParameteri(GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_MAG_FILTER,
            GL_NEAREST);
        glTexParameteri(GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_WRAP_S, GL_CLAMP
            );
        glTexParameteri(GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_WRAP_T, GL_CLAMP
            );
    }

    fprintf(stdout, " -> MaskTexture initialized OK\n");
    fprintf(stdout, " -> TextureID: %d\n", maskTexture->GetTextureID());
    fprintf(stdout, " -> Size: %dx%d\n", MapSize, MapSize);
    fprintf(stdout, " -> IsDepthTexture %d\n", maskTexture->IsDepthTexture()
        );
    fprintf(stdout, " -> IsTexture %d\n", maskTexture->IsTexture());
```

```
    fprintf(stdout, " -> IsFloatTexture %d\n", maskTexture->IsFloatTexture()
        );
    fprintf(stdout, " -> IsDoubleBuffered %d\n", maskTexture->
        IsDoubleBuffered());

}
void Chan::InitCG(){
    fprintf(stderr, "\nInitializing CG...\n");
    if(cgGLIsProfileSupported(CG_PROFILE_FP30)){
        fprofile = CG_PROFILE_FP30;
        maskprofile = CG_PROFILE_FP30;
        shamapfprofile = CG_PROFILE_FP30;
        fprintf(stderr, " -> Profile: FP30\n");
    }
    if (cgGLIsProfileSupported(CG_PROFILE_VP30)){
        vprofile = CG_PROFILE_VP30;
        maskvprofile = CG_PROFILE_VP30;
        shamapvprofile = CG_PROFILE_VP30;
        fprintf(stderr, " -> Profile: VP30\n");
    }
    else
    {
        fprintf(stderr, "Video card does not support vertex programs,
            exiting...\n");
        exit(-1);
    }

    /* Test cgContext creation */
    Context = cgCreateContext();
    CheckCgError();

    fProgram = cgCreateProgramFromFile(Context, CG_SOURCE, fpDir, fprofile,
        NULL, NULL);
    vProgram = cgCreateProgramFromFile(Context, CG_SOURCE, vpDir, vprofile,
        NULL, NULL);
    maskprogram = cgCreateProgramFromFile(Context, CG_SOURCE, maskDir,
        maskprofile, NULL, NULL);
    maskvprogram = cgCreateProgramFromFile(Context, CG_SOURCE, maskvDir,
        maskvprofile, NULL, NULL);
    shamapvprogram = cgCreateProgramFromFile(Context, CG_SOURCE, shamapvDir
        , shamapvprofile, NULL, NULL);
    shamapfprogram = cgCreateProgramFromFile(Context, CG_SOURCE, shamapfDir
        , shamapfprofile, NULL, NULL);

    fprintf(stderr, " -> fProgram: ");
    fprintf(stderr, fpDir);
    fprintf(stderr, "\n");
    fprintf(stderr, " -> vProgram: ");
    fprintf(stderr, vpDir);
    fprintf(stderr, "\n");
    fprintf(stderr, " -> maskProgram: ");
    fprintf(stderr, maskDir);
    fprintf(stderr, "\n");
    fprintf(stderr, " -> maskvProgram: ");
    fprintf(stderr, maskvDir);
    fprintf(stderr, "\n");
    fprintf(stderr, " -> shamapvProgram: ");
    fprintf(stderr, maskvDir);
    fprintf(stderr, "\n");
    fprintf(stderr, " -> shamapfProgram: ");
    fprintf(stderr, maskvDir);
    fprintf(stderr, "\n");
```

```
    if(fProgram != NULL)
    {
        cgGLLoadProgram(fProgram);
        CheckCgError();
    }
    if(vProgram != NULL)
    {
        cgGLLoadProgram(vProgram);
        CheckCgError();
    }
    if(maskprogram != NULL)
    {
        cgGLLoadProgram(maskprogram);
        CheckCgError();
    }
    if(maskvprogram != NULL)
    {
        cgGLLoadProgram(maskvprogram);
        CheckCgError();
    }
    if(shamapvprogram != NULL)
    {
        cgGLLoadProgram(shamapvprogram);
        CheckCgError();
    }
    if(shamapfprogram != NULL)
    {
        cgGLLoadProgram(shamapfprogram);
        CheckCgError();
    }

    if (!cgIsProgramCompiled(fProgram))
        cgCompileProgram(fProgram);
    if (!cgIsProgramCompiled(vProgram))
        cgCompileProgram(vProgram);
    if (!cgIsProgramCompiled(maskprogram))
        cgCompileProgram(maskprogram);
    if (!cgIsProgramCompiled(maskvprogram))
        cgCompileProgram(maskvprogram);
    if (!cgIsProgramCompiled(shamapvprogram))
        cgCompileProgram(shamapvprogram);
    if (!cgIsProgramCompiled(shamapfprogram))
        cgCompileProgram(shamapfprogram);

    // Get Parameters
    mvp = cgGetNamedParameter(vProgram, "mvp");
    mv = cgGetNamedParameter(vProgram, "mv");
    mvIT = cgGetNamedParameter(vProgram, "mvIT");
    observerEyeToLightClip = cgGetNamedParameter(vProgram, "
        observerEyeToLightClip");
    lightPosEye = cgGetNamedParameter(vProgram, "lightPosEye");

    fprintf(stderr, " -> CG Initialized Succesfully\n");
};

void Chan::CheckCgError()
{
    CGerror err = cgGetError();

    if (err != CG_NO_ERROR)
    {
```

```
        printf("CG error: %s\n", cgGetErrorString(err));
        exit(1);
    }
}

void Chan::render(){

#ifdef REC_TIME
    exc_timer->start();
#endif
    timer.start();

    // Step 1 : Render To Shadow Map from Light
    // For all lights
    for(int i=0; i<lights->no_lights(); i++){

        Camera* LightCam = new Camera(lights->get_light(i)->get_position(),
                                    lights->get_light(i)->get_direction(),
                                    lights->get_light(i)->get_up(),
                                    lights->get_light(i)->get_cutoff_angle()
                                        *2.0,
                                    lights->get_light(i)->get_near(),
                                    lights->get_light(i)->get_far());

        glViewport(0,0,MapSize, MapSize);
        rt[i]->BeginCapture();
                glViewport(0,0,MapSize, MapSize);
                prepare_for_texture_render();
                load_projection(LightCam);
                load_modelview(LightCam);
                glCallList(dlDrawScene);
                glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE_ARB,
                    GL_COMPARE_R_TO_TEXTURE);
        rt[i]->EndCapture();
    }
    glViewport(0,0,vpWidth, vpHeight);
#ifdef REC_TIME
    exc_timer->add_time();

#endif
#ifdef SHOW_TIME_BOX
    glFinish();
    tb->set_value(0, timer.get_secs(), "Create Shadow Map");
    timer.start();
#endif

    /***************************************************************************/

    /* Step 2 : Identify Shadow Silhouette pixel using the shadow map.
                */
    /*          vertex and fragment program used to create an image of the
        scene */
    /*          in which silhouette pixels are white and non-silhouette
        black     */
    /***************************************************************************/

    for(int i=0; i<lights->no_lights(); i++){

        glClear(GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT |
            GL_DEPTH_BUFFER_BIT);

        rt[i]->BindDepth();
```

```
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        load_projection(cam);
        load_modelview(cam);
        cgGLSetTextureParameter(cgGetNamedParameter(fProgram, "shadowMap"),
            rt[i]->GetDepthTextureID());

        cgGLSetStateMatrixParameter(mvp,
                                    CG_GL_MODELVIEW_PROJECTION_MATRIX,
                                    CG_GL_MATRIX_IDENTITY);
        cgGLSetStateMatrixParameter(mv,
                                    CG_GL_MODELVIEW_MATRIX,
                                    CG_GL_MATRIX_IDENTITY);
        cgGLSetStateMatrixParameter(mvIT,
                                    CG_GL_MODELVIEW_MATRIX,
                                    CG_GL_MATRIX_INVERSE_TRANSPOSE);

        Mat4x4f ObsToEye = BiasMatrix
                     * get_light_projection_matrix(lights->get_light(i))
                     * get_light_view_matrix(lights->get_light(i))
                     * CGLA::invert(get_cam_modelview_matrix(cam));
        cgGLSetMatrixParameterfr(observerEyeToLightClip, ObsToEye.get());
        Vec3f lightPosEyeM = multM4V3(&CGLA::invert(get_cam_modelview_matrix
            (cam)),&lights->get_light(i)->get_position());;
        cgGLSetParameter3fv(lightPosEye, lightPosEyeM.get());

        cgGLEnableProfile(vprofile);
        cgGLEnableProfile(fprofile);
        cgGLBindProgram(vProgram);
        cgGLBindProgram(fProgram);

        load_projection(cam);
        load_modelview(cam);
        glColor3f(1,0,0);
        glDisable(GL_LIGHTING);
        glCallList(dlDrawScene);

        cgGLDisableProfile(fprofile);
        cgGLDisableProfile(vprofile);

        glEnable(GL_TEXTURE_RECTANGLE_NV);
        glBindTexture(GL_TEXTURE_RECTANGLE_NV, silTexture[i]);
        glCopyTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_RGB, 0, 0,vpWidth,
            vpHeight, 0);
    }
#ifdef REC_TIME
    exc_timer->add_time();
#endif
#ifdef SHOW_TIME_BOX
    glFinish();
    tb->set_value(1, timer.get_secs(), "Identify Silhouette pixels copy to
        tex");
    timer.start();
#endif

    if(!bShowShaMap){
    /*****************************************************************/
    /* Step 3 : We setup a screen aligned quad and use fragment program */
    /*          maskProgram to set depth of pixels.                     */
    /*          Pixels with color 0 aka. non-silhouette pixels          */
    /*          are given depth  0. Other pixels are discarded          */
```

```
/*****************************************************************/
glClear(GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_ALWAYS);

    cgGLSetTextureParameter(cgGetNamedParameter(maskprogram, "
        texColorMask"), silTexture[0]);
    cgGLEnableProfile(maskprofile);
    CheckCgError();
    cgGLBindProgram(maskprogram);
    CheckCgError();
    cgGLEnableTextureParameter(cgGetNamedParameter(maskprogram, "
        texColorMask"));

    if(!bShowScreenQuad)
        glColorMask(0,0,0,0);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1,1,-1,1,-10,100);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,1,0,0,0,0,1,0);

    glEnable(GL_DEPTH_TEST); //For use later DONT DELETE
    glBegin(GL_QUADS);
        glVertex3f(-1,-1,0);
        glVertex3f( 1,-1,0);
        glVertex3f( 1, 1,0);
        glVertex3f(-1, 1,0);
    glEnd();

    glColorMask(1,1,1,1);
    cgGLDisableTextureParameter(cgGetNamedParameter(maskprogram, "
        texColorMask"));
    cgGLDisableProfile(maskprofile);
    CheckCgError();

glDisable(GL_TEXTURE_RECTANGLE_NV);
if(!bShowScreenQuad){


/*********************************************************************/

/* Step 5 : Rendering the silhouette pixels using shadow volumes
            */
/*********************************************************************/

if(bDPT){
    glEnable(GL_DEPTH_BOUNDS_TEST_EXT);
    glDepthBoundsEXT(0.001, 1.0);

}

#ifdef REC_TIME
    exc_timer->add_time();
#endif
#ifdef SHOW_TIME_BOX
    glFinish();
    tb->set_value(2, timer.get_secs(), "Create Mask");
#endif
```

```
timer.start();
glClear(GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

glDepthMask(1);

glMatrixMode (GL_PROJECTION);
glLoadIdentity();
glPushMatrix();
glLoadIdentity();
gluPerspective(cam->get_fov(), cam->get_aspect(), cam->get_near(), cam->
    get_far());
glGetFloatv(GL_PROJECTION_MATRIX, P_inf);
glPopMatrix();
P_inf[14] = -2.0*cam->get_near();
P_inf[10] = P_inf[11] = -1.0;

glMultMatrixf(P_inf);

load_modelview(cam);

/* Render scene with ambient colors*/
glDepthMask(1);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LEQUAL);
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
lights->turn_all_off();
update_lights();
glCallList(dlDrawScene);
glDepthFunc(GL_LESS);

/* Prepare for shadow volume render */
if(!bShowShaVol)
    glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );
glDepthMask( GL_FALSE );
glShadeModel(GL_FLAT);
glDisable(GL_LIGHTING);

glDisable(GL_CULL_FACE);
glClear(GL_STENCIL_BUFFER_BIT);
glEnable(GL_STENCIL_TEST);

for(int i=0; i<lights->no_lights(); i++){
    if(GL_EXT_stencil_two_side && bTwoSideStencil){
        glEnable(GL_STENCIL_TEST_TWO_SIDE_EXT);

        glActiveStencilFaceEXT(GL_BACK);
        glStencilOp(GL_KEEP,                    // stencil test fail
            GL_INCR_WRAP_EXT,                   // depth test fail
            GL_KEEP);                           // depth test pass
        glStencilMask(~0);
        glStencilFunc(GL_ALWAYS, 0, ~0);

        glActiveStencilFaceEXT(GL_FRONT);
        glStencilOp(GL_KEEP,                     // stencil test fail
            GL_DECR_WRAP_EXT,           // depth test fail
            GL_KEEP);                   // depth test pass
        glStencilMask(~0);
        glStencilFunc(GL_ALWAYS, 0, ~0);
        draw_shadow_volumes(lights->get_light(i));
    }
    else{
```

```
            glEnable(GL_CULL_FACE);
            glClear(GL_STENCIL_BUFFER_BIT);
            glEnable(GL_STENCIL_TEST);
            glStencilFunc(GL_ALWAYS, 0, ~0);

            glCullFace(GL_FRONT);
            glStencilOp(GL_KEEP,GL_INCR,GL_KEEP);
            draw_shadow_volumes(lights->get_light(i));

            glCullFace(GL_BACK);
            glStencilOp(GL_KEEP,GL_DECR,GL_KEEP);
            draw_shadow_volumes(lights->get_light(i));

        }
    }

    glEnable(GL_CULL_FACE);
    glCullFace(GL_BACK);

    glStencilFunc(GL_EQUAL, 0, ~0);
    glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);
    if(!bShowShaVol)
    {
#ifdef REC_TIME
    exc_timer->add_time();
#endif
#ifdef SHOW_TIME_BOX
    glFinish();
    tb->set_value(3, timer.get_secs(), "Render Shadow Volumes");
    timer.start();
#endif
    glDisable(GL_DEPTH_BOUNDS_TEST_EXT);
#ifdef USE_CG
        load_projection(cam);
        load_modelview(cam);

        cgGLSetTextureParameter(cgGetNamedParameter(shamapfprogram, "
            shadowMap"), rt[0]->GetDepthTextureID());
        cgGLSetParameter1f(cgGetNamedParameter(shamapfprogram, "nlights"), (
            float)lights->no_lights());

        cgGLSetStateMatrixParameter(cgGetNamedParameter(shamapvprogram, "mvp
            "),
            CG_GL_MODELVIEW_PROJECTION_MATRIX,
            CG_GL_MATRIX_IDENTITY);
        cgGLSetStateMatrixParameter(cgGetNamedParameter(shamapvprogram, "mv
            "),
            CG_GL_MODELVIEW_MATRIX,
            CG_GL_MATRIX_IDENTITY);
        cgGLSetStateMatrixParameter(cgGetNamedParameter(shamapvprogram, "
            mvIT"),
            CG_GL_MODELVIEW_MATRIX,
            CG_GL_MATRIX_INVERSE_TRANSPOSE);

        Mat4x4f ObsToEye = BiasMatrix
            * get_light_projection_matrix(lights->get_light(0))
            * get_light_view_matrix(lights->get_light(0))
            * CGLA::invert(get_cam_modelview_matrix(cam));

        cgGLSetMatrixParameterfr(cgGetNamedParameter(shamapvprogram, "
            observerEyeToLightClip"), ObsToEye.get());
        cgGLSetParameter3fv(cgGetNamedParameter(shamapvprogram, "lightPos")
```

```
            , lights->get_light(0)->get_position().get());

        cgGLEnableProfile(shamapvprofile);
        cgGLEnableProfile(shamapfprofile);

        cgGLBindProgram(shamapvprogram);
        cgGLBindProgram(shamapfprogram);

        cgGLEnableTextureParameter(cgGetNamedParameter(shamapfprogram, "
            shadowMap"));

        /* draw lit parts aka where stencilbuffer == 0 */
        glDepthMask(GL_TRUE);
        glClear(GL_DEPTH_BUFFER_BIT);
        glShadeModel(GL_SMOOTH);
        glEnable(GL_DEPTH_TEST);
        glDepthFunc(GL_LEQUAL);
        glColorMask(1,1,1,1);
        glStencilFunc(GL_EQUAL, 0, ~0);
        glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);

        load_projection(cam);
        load_modelview(cam);
        glCallList(dlDrawScene);

        cgGLDisableProfile(shamapfprofile);
        cgGLDisableProfile(shamapvprofile);
#else
    /* draw lit parts aka where stencilbuffer == 0 */
    glDepthMask(GL_TRUE);
    glClear(GL_DEPTH_BUFFER_BIT);
    glShadeModel(GL_SMOOTH);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glColorMask(1,1,1,1);
    glStencilFunc(GL_EQUAL, 0, ~0);
    glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);

    rt[0]->BindDepth();
    rt[0]->EnableTextureTarget();
    lights->turn_all_on();
    Mat4x4f texture_matrix = BiasMatrix *
                             get_light_projection_matrix(lights->get_light
                                 (0)) *
                             get_light_view_matrix(lights->get_light(0));
    generate_texture_coordinates(&texture_matrix);
    rt[0]->BindDepth();
    glEnable(rt[0]->GetTextureTarget());

    //Enable shadow comparison
    glTexParameteri(rt[0]->GetTextureTarget(), GL_TEXTURE_COMPARE_MODE_ARB,
        GL_COMPARE_R_TO_TEXTURE);

    //Shadow comparison should be true (ie not in shadow) if r<=texture
    glTexParameteri(rt[0]->GetTextureTarget(), GL_TEXTURE_COMPARE_FUNC_ARB,
        GL_LEQUAL);

    //Shadow comparison should generate an INTENSITY result
    glTexParameteri(rt[0]->GetTextureTarget(), GL_DEPTH_TEXTURE_MODE_ARB,
        GL_ALPHA);
    if(bTestIntensity)
        glTexParameteri(rt[0]->GetTextureTarget(), GL_DEPTH_TEXTURE_MODE_ARB
```

```
            , GL_INTENSITY);

    //Set alpha test to discard false comparisons
    if(!bTestIntensity){
        glAlphaFunc(GL_GREATER, 0.0f);
        glEnable(GL_ALPHA_TEST);
    }

    load_projection(cam);
    load_modelview(cam);
    lights->turn_all_on();
    update_lights();
    glCallList(dlDrawScene);

    rt[0]->DisableTextureTarget();

    glTexParameteri(rt[0]->GetTextureTarget(), GL_DEPTH_TEXTURE_MODE_ARB,
        GL_INTENSITY);
    glDisable(GL_ALPHA_TEST);
    glDisable(GL_STENCIL_TEST);
#endif
    glDisable(GL_BLEND);
    glDepthFunc(GL_LESS);
    glDisable(GL_BLEND);
    glDisable(GL_STENCIL_TEST);
#ifdef REC_TIME
    exc_timer->add_time();
#endif
#ifdef SHOW_TIME_BOX
    glFinish();
    tb->set_value(4, timer.get_secs(), "Render lit parts");
    timer.start();
#endif

    RenderAlgorithm::render();
    }     //End of !bShowShaVol
    }     //End of !bShowSqreenQuad
    }     //End of !bShowShaMap

    SVdt->draw();

#ifdef REC_TIME
    exc_timer->add_time();
    exc_timer->end_line();
#endif
}

void Chan::prepare_for_texture_render()
{
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);
    glDepthMask(1);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
        GL_STENCIL_BUFFER_BIT);
    glDisable(GL_LIGHTING);
    //Draw front faces into the shadow map
    glEnable(GL_CULL_FACE);
    glCullFace(GL_BACK);
    //Disable color writes, and use flat shading for speed
    glShadeModel(GL_FLAT);
    glColorMask(0,0,0,0);
    //Enable Biasing
```

```
    glEnable(GL_POLYGON_OFFSET_POINT );
    glEnable(GL_POLYGON_OFFSET_FILL);
    glPolygonOffset(BiasFactor, BiasUnits);

    char buf[50];
    sprintf(buf, "Bias: ON      Factor: %.1f       Units: %.1f", BiasFactor,
        BiasUnits);
    displaytext.updateText(dtBias, buf);
}

void Chan::show_shadow_map(int i)
{

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    rt[i]->DisableTextureTarget();
    glDisable(GL_TEXTURE_RECTANGLE_NV);
    glDisable(GL_TEXTURE_2D);

    rt[i]->BindDepth();
    rt[i]->EnableTextureTarget();

    int maxs = rt[i]->GetMaxS();
    int maxt = rt[i]->GetMaxT();

    glDisable(GL_LIGHTING);
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

    glViewport(0+i*200,0,200,200);

    glDisable(GL_TEXTURE_GEN_Q);
    glDisable(GL_TEXTURE_GEN_R);
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);

    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    glOrtho(-1,1,-1,1,-10,10);

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();

    glColor3f(1,0,1);

    glBegin(GL_POLYGON);
        glTexCoord2f(0.0, 0.0);
        glVertex3f(-1,-1, 0);
        glTexCoord2f (0.0, maxt);
        glVertex3f( 1,-1, 0);
        glTexCoord2f(maxs, maxt);
        glVertex3f( 1, 1, 0);
        glTexCoord2f(maxs, 0.0);
        glVertex3f(-1, 1, 0);
    glEnd();

    rt[i]->DisableTextureTarget();

    glMatrixMode( GL_PROJECTION );
    glPopMatrix();
```

```
    glMatrixMode( GL_MODELVIEW );
    glPopMatrix();

    glViewport(0,0,vpWidth, vpHeight);
}
Mat4x4f Chan::calculate_texture_matrix(Light* light)
{
    Mat4x4f light_projection_matrix = get_light_projection_matrix(light);
    Mat4x4f light_view_matrix = get_light_view_matrix(light);

    return light_projection_matrix * light_view_matrix; //BiasMatrix maybe
        multiplied
}

Mat4x4f Chan::get_light_projection_matrix(Light* light)
{
    glMatrixMode (GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity ();
    gluPerspective(light->get_cutoff_angle()*2.0, 1, light->get_near(),light
        ->get_far());
    float lightProjectionMatrixTemp[16];
    glGetFloatv(GL_TRANSPOSE_PROJECTION_MATRIX, lightProjectionMatrixTemp);
    Mat4x4f lightProjectionMatrix(lightProjectionMatrixTemp);
    glPopMatrix();
    return lightProjectionMatrix;
}

Mat4x4f Chan::get_light_view_matrix(Light* light)
{
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();
    gluLookAt(light->get_position()[0],
        light->get_position()[1],
        light->get_position()[2],
        light->get_direction()[0],
        light->get_direction()[1],
        light->get_direction()[2],
        light->get_up()[0],
        light->get_up()[1],
        light->get_up()[2]);
    float lightViewMatrixTemp[16];
    glGetFloatv(GL_TRANSPOSE_MODELVIEW_MATRIX, lightViewMatrixTemp);
    Mat4x4f lightViewMatrix(lightViewMatrixTemp); //convert to Mat4f4 matrix
    glPopMatrix();
    return lightViewMatrix;
}

void Chan::make_adjacent_list()
{
    cout << "Generating adjacent list (if not initialized)" << endl;

    //For each mesh
    for(int mesh=0; mesh<obj->no_meshes(); mesh++)
    {
        TriMesh* cur_mesh = obj->get_mesh(mesh);
        if(!cur_mesh->adjInitialized)
        {
            cur_mesh->adjacents.resize(cur_mesh->no_faces());
            //For each face
            for(int i=0; i<cur_mesh->no_faces(); i++)
```

```
            {
                int found_tris = 0;
                for(int j=0; j<cur_mesh->no_faces(); j++)
                {
                    if(i != j){
                        // if face(i) shares two vertices with face(j)
                        int shared_vertices = 0;
                        for(int k=0; k<3; k++)
                        {
                            for(int l=0; l<3; l++)
                            {
                                if(cur_mesh->get_face(i)[k] == cur_mesh->
                                    get_face(j)[(l+1)%3] && cur_mesh->
                                    get_face(i)[(k+1)%3] == cur_mesh->
                                    get_face(j)[l])
                                {
                                    (cur_mesh->adjacents[i])[k] = j;
                                }
                            }
                        }
                        if(shared_vertices >= 2){
                            (cur_mesh->adjacents[i])[found_tris] = j;
                            found_tris++;
                        }
                    }
                }
            }
            cur_mesh->adjInitialized = true;
        }
        else
        {
            cout << " -> Already initialized" << endl;
        }

    }
    cout << " -> done!" << endl;
}

bool Chan::is_backfacing(Light* light, int mesh_index, int face_index){

    Vec3f face_avg_pos = (obj->get_mesh(mesh_index)->get_face_vertex(
        face_index,0)
        + obj->get_mesh(mesh_index)->get_face_vertex(face_index,1)
        + obj->get_mesh(mesh_index)->get_face_vertex(face_index,2))/3.0;

    Vec3f light_dir = light->get_position() - face_avg_pos;

    if(dot(normalize(light_dir), obj->get_mesh(mesh_index)->get_face_normal(
        face_index)) <= 0)
        return true;
    else

        return false;
}
void Chan::draw_shadow_volumes(Light* light) //, TriMesh* TM, vector<Light
    >* lights
{
    Vec4f L = Vec4f(light->get_position(), 1.0);
    //For all meshes
    for(int k=0; k<obj->no_meshes(); k++)
    {
        // For all triangles
```

```
        for(int i=0; i<obj->get_mesh(k)->no_faces(); i++)
        {
            TriMesh* mesh = obj->get_mesh(k);
            // If triangle is backfacing
            if(is_backfacing(light, k, i))
            {
                //for each edge
                for(int j=0; j<3; j++)
                {
                    //if adjacent face is front-facing, edge is silhouette
                        edge
                    //int adj = (mesh->adjacents[i])[j];
                    if(!is_backfacing(light, k, (mesh->adjacents[i])[j]))
                    {
                        // then we extrude and draw the edge
                        Vec4f A(mesh->get_face_vertex(i,j), 1);
                        Vec4f B(mesh->get_face_vertex(i,(j+1)%3), 1);
                        Vec4f AL(A[0]*L[3]-L[0]*A[3],
                            A[1]*L[3]-L[1]*A[3],
                            A[2]*L[3]-L[2]*A[3], 0);
                        Vec4f BL(B[0]*L[3]-L[0]*B[3],
                            B[1]*L[3]-L[1]*B[3],
                            B[2]*L[3]-L[2]*B[3], 0);
                        //glNormal3f(0.0, 1.0, 0.0);

                            glBegin(GL_QUADS);
                                glColor4f(1.0, 1.0, 0.0, 0.2);
                                glVertex4fv(A.get());
                                glVertex4fv(B.get());
                                glVertex4fv(BL.get());
                                glVertex4fv(AL.get());
                            glEnd();

                    }
                }

                glBegin(GL_TRIANGLES);
                for(int v=0; v<3; v++){
                    glColor4f(1.0, 1.0, 0.0, 0.2);
                    Vec4f V = Vec4f(mesh->get_face_vertex(i,v), 1.0);
                    glVertex4f(V[0]*L[3]-L[0]*V[3],
                        V[1]*L[3]-L[1]*V[3],
                        V[2]*L[3]-L[2]*V[3], 0);
                }
                glEnd();


            }
            else // Front facing triangle
            {
                glBegin(GL_TRIANGLES);
                for(int v=0; v<3; v++){
                    Vec4f V = Vec4f(mesh->get_face_vertex(i,v), 1.0);
                    glVertex4f(V[0], V[1], V[2], V[3]);
                }
                glEnd();
            }
        }
    }

}
```

```
void Chan::initShadowVolume()
{
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glShadeModel (GL_SMOOTH);

    /* Enable depth test and stencil test */
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_STENCIL_TEST);

    /* Enable lighting */
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
}

void Chan::generate_texture_coordinates(Mat4x4f* texture_matrix)
{
    //Set up texture coordinate generation.
    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_S, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[0].get()));
    glEnable(GL_TEXTURE_GEN_S);

    glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_T, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[1].get()));
    glEnable(GL_TEXTURE_GEN_T);

    glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_R, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[2].get()));
    glEnable(GL_TEXTURE_GEN_R);

    glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_Q, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[3].get()));
    glEnable(GL_TEXTURE_GEN_Q);
}

void Chan::enable_r_to_texture_comparison(int i)
{
    rt[i]->BindDepth();
    glEnable(rt[i]->GetTextureTarget());

    //Enable shadow comparison
    glTexParameteri(rt[i]->GetTextureTarget(), GL_TEXTURE_COMPARE_MODE_ARB,
        GL_COMPARE_R_TO_TEXTURE);

    //Shadow comparison should be true (ie not in shadow) if r<=texture
    glTexParameteri(rt[i]->GetTextureTarget(), GL_TEXTURE_COMPARE_FUNC_ARB,
        GL_LEQUAL);

    //Shadow comparison should generate an INTENSITY result
    glTexParameteri(rt[i]->GetTextureTarget(), GL_DEPTH_TEXTURE_MODE_ARB,
        GL_INTENSITY);
}
```

C.5 Shadow Maps (Using Copy To Texture)

```
/*****************************************************************/
/* filename: ShadowMap.h                                         */
/*****************************************************************/
#ifndef __SHA_MAP_H__
#define __SHA_MAP_H__

#include "My/RenderAlgorithm.h"
#include "Components/Timer.h"
```

```
#include "My/timebox.h"

class ShadowMap : public RenderAlgorithm
{
protected:
    vector<GLuint> shadowMapTexture;
    int map_size;
    Components::Timer timer;

    //Matrices
    GLfloat lightProjectionMatrixTemp[16], lightViewMatrixTemp[16];
    GLfloat cameraProjectionMatrixTemp[16], cameraViewMatrixTemp[16];
    Mat4x4f biasMatrix;

    bool show_map;
    bool bPCF;

    // For biasing
    float factor, units;

    //For test output
    int biastext;
    int mapsizetext;

#ifdef REC_TIME
    TimeToExcel* exc_timer;
#endif
public:
    ShadowMap(Object* _obj, LightContainer* _lights, Camera* _cam, Object*
        _floor) : RenderAlgorithm(_obj, _lights, _cam, _floor){
        init();
        shadowMapTexture.resize(lights->no_lights());
        bPCF = true;
        init_shadow_texture();
        map_size = 1024;
        show_map = true;

        biasMatrix = Mat4x4f(Vec4f(0.5, 0.0, 0.0, 0.5),
                             Vec4f(0.0, 0.5, 0.0, 0.5),
                             Vec4f(0.0, 0.0, 0.5, 0.5),
                             Vec4f(0.0, 0.0, 0.0, 1.0));

        biastext = displaytext.addText("");
        mapsizetext = displaytext.addText("");
        char buf[50];
        sprintf(buf, "Mapsize: %dx%d", map_size, map_size);
        displaytext.updateText(mapsizetext, buf);

        factor = 1.2;
        units = 4.0;

#ifdef REC_TIME
        // For Time Reporting
        exc_timer = new TimeToExcel("./output/ShadowMapOutput.txt");
        exc_timer->add_title_item("Draw Scene for Shadow Map");
        exc_timer->add_title_item("Copy to Shadow Map");
        exc_timer->add_title_item("Rendering using Shadow Map");
        exc_timer->add_title_item("Misc Info");
        exc_timer->write_title_line();
#endif
    };
```

```
void render();
void init();
void init_shadow_texture();
void show_shadow_map(GLuint tex_map, int window);
void show_misc_info(int window);
void copy_drawing_buffer_to_texture(GLuint tex_map);
void prepare_for_texture_render();
void enable_r_to_texture_comparison(GLuint tex_map);
void reset_all();
void restore_states();
Mat4x4f calculate_texture_matrix(Light* light);
Mat4x4f get_light_projection_matrix(Light* light);
Mat4x4f get_light_view_matrix(Light* light);
void generate_texture_coordinates(Mat4x4f* texture_matrix);

void process_key(unsigned char key)
{
    Vec3f p0;
    Vec3f p7;
    switch(key) {
        case 'o':
            obj->get_bbox(p0, p7);
            for(int i=0; i<lights->no_lights(); i++)
                lights->get_light(i)->calculate_near_far(bbox(p0,p7));
            break;
        case 'l':
            obj->get_bbox(p0, p7);
            for(int i=0; i<lights->no_lights(); i++)
                lights->get_light(i)->calculate_fustum(bbox(p0,p7));
            break;
        case 'p':
            bPCF = !bPCF;
            init_shadow_texture();
            break;
        case 43: // Ascii +
            increase_map_size();
            break;
        case 45: // Ascii -
            decrease_map_size();
            break;
        case 'y':
            increase_bias_factor(0.1);
            break;
        case 'h':
            decrease_bias_factor(0.1);
            break;
        case 'Y':
            increase_bias_units(0.1);
            break;
        case 'H':
            decrease_bias_units(0.1);
            break;
        default:
            RenderAlgorithm::process_key(key);
            break;
    }
}

void increase_map_size(){
    if(map_size < 2048){
        map_size *= 2;
        init_shadow_texture();
```

```
            char buf[50];
            sprintf(buf, "Mapsize: %dx%d", map_size, map_size);
            displaytext.updateText(mapsizetext, buf);
        }
    }
    void decrease_map_size(){
        map_size /= 2.0;
        init_shadow_texture();
        char buf[50];
        sprintf(buf, "Mapsize: %dx%d", map_size, map_size);
        displaytext.updateText(mapsizetext, buf);
    }

    /* set functions */
    void set_map_size(int i){
        map_size = i;
        char buf[50];
        sprintf(buf, "Mapsize: %dx%d", map_size, map_size);
        displaytext.updateText(mapsizetext, buf);
    }

    /* print functions */
    void print_all()
    {
        RenderAlgorithm::print_all();
        cout << "Map Size: " << map_size << endl;
    }

    void increase_bias_factor(float amount=0.1){factor += amount;}
    void decrease_bias_factor(float amount=0.1){factor -= amount;}
    void increase_bias_units(float amount=0.1){units += amount;}
    void decrease_bias_units(float amount=0.1){units -= amount;}
};
#endif

/*************************************************************/
/* filename: ShadowMap.cpp                                   */
/*************************************************************/
#include "My/ShadowMap.h"
void ShadowMap::init()
{
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glShadeModel (GL_SMOOTH);

    /* Enable depth test and stencil test */
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_STENCIL_TEST);
    //glEnable(GL_NORMALIZE);
    glDepthFunc(GL_LEQUAL);
    /* Enable lighting */
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );
    RenderAlgorithm::init();
}

void ShadowMap::render()
{
#ifdef REC_TIME
    exc_timer->start();
#endif
    /* For all lights */
```

```
    /* 1. Render the scene as seen from the light */
    /* Create virtual camera at lights position */
    for(int i=0; i<lights->no_lights(); i++){
        Camera* light_cam = new Camera(lights->get_light(i)->get_position(),
            lights->get_light(i)->get_direction(),
            lights->get_light(i)->get_up(),
            lights->get_light(i)->get_cutoff_angle()*2.0,
            lights->get_light(i)->get_near(),
            lights->get_light(i)->get_far());

        prepare_for_texture_render();

        load_projection(light_cam);
        load_modelview(light_cam);

        glCallList(dlDrawScene);
#ifdef REC_TIME
        exc_timer->add_time();
#endif
        copy_drawing_buffer_to_texture(shadowMapTexture[i]);
#ifdef REC_TIME
        exc_timer->add_time();
#endif
        restore_states();
        clear_all();
    }
    /* 2. Render the scene from the camera using the shadowmap*/
    /*    The depth-value seen from the camera is transformed to light-space
            and compared to the depth-value from the shadow map. */
    for(int i=0; i<lights->no_lights(); i++)
    {
        load_projection(cam);
        load_modelview(cam);
        lights->turn_all_off();
        lights->get_light(i)->turn_on();
        update_lights();

        Mat4x4f texture_matrix = calculate_texture_matrix(lights->get_light(
            i));
        generate_texture_coordinates(&texture_matrix);
        enable_r_to_texture_comparison(shadowMapTexture[i]);

        glCallList(dlDrawScene);

        /* Draw misc info */
        glDisable(GL_BLEND);
        //show_shadow_map(shadowMapTexture[i], i);
        glEnable(GL_BLEND);
        glBlendFunc(GL_ONE, GL_ONE);
    }
    glDisable(GL_BLEND);

#ifdef REC_TIME
    exc_timer->add_time();
#endif
    /* reset all */
    reset_all();

    RenderAlgorithm::render();
#ifdef REC_TIME
    exc_timer->add_time();
    exc_timer->end_line();
```

```
#endif
}

void ShadowMap::reset_all()
{
    //Disable textures and texgen
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
    glDisable(GL_TEXTURE_GEN_R);
    glDisable(GL_TEXTURE_GEN_Q);

    //Restore other states
    glDisable(GL_LIGHTING);
    glDisable(GL_ALPHA_TEST);
}

void ShadowMap::enable_r_to_texture_comparison(GLuint tex_map)
{
    //Bind & enable shadow map texture
    glBindTexture(GL_TEXTURE_2D, tex_map);
    glEnable(GL_TEXTURE_2D);

    //Enable shadow comparison
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE_ARB,
        GL_COMPARE_R_TO_TEXTURE);

    //Shadow comparison should be true (ie not in shadow) if r<=texture
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC_ARB, GL_LEQUAL);


    //Shadow comparison should generate an INTENSITY result
    glTexParameteri(GL_TEXTURE_2D, GL_DEPTH_TEXTURE_MODE_ARB, GL_INTENSITY);
}

void ShadowMap::generate_texture_coordinates(Mat4x4f* texture_matrix)
{
    //Set up texture coordinate generation.
    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_S, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[0].get()));
    glEnable(GL_TEXTURE_GEN_S);

    glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_T, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[1].get()));
    glEnable(GL_TEXTURE_GEN_T);

    glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_R, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[2].get()));
    glEnable(GL_TEXTURE_GEN_R);

    glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_Q, GL_EYE_PLANE, (GLfloat*)((*texture_matrix)[3].get()));
    glEnable(GL_TEXTURE_GEN_Q);
}

Mat4x4f ShadowMap::calculate_texture_matrix(Light* light)
{
    Mat4x4f light_projection_matrix = get_light_projection_matrix(light);
    Mat4x4f light_view_matrix = get_light_view_matrix(light);

    return biasMatrix * light_projection_matrix * light_view_matrix;
}
```

```
Mat4x4f ShadowMap::get_light_projection_matrix(Light* light)
{
    glMatrixMode (GL_PROJECTION);
    glPushMatrix();
        glLoadIdentity ();
        gluPerspective(light->get_cutoff_angle()*2.0, 1, light->get_near(),
            light->get_far());
        glGetFloatv(GL_TRANSPOSE_PROJECTION_MATRIX,
            lightProjectionMatrixTemp);
        Mat4x4f lightProjectionMatrix(lightProjectionMatrixTemp);
    glPopMatrix();
    return lightProjectionMatrix;
}

Mat4x4f ShadowMap::get_light_view_matrix(Light* light)
{
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
        glLoadIdentity();
        gluLookAt(light->get_position()[0],
                  light->get_position()[1],
                  light->get_position()[2],
                  light->get_direction()[0],
                  light->get_direction()[1],
                  light->get_direction()[2],
                  light->get_up()[0],
                  light->get_up()[1],
                  light->get_up()[2]);
        glGetFloatv(GL_TRANSPOSE_MODELVIEW_MATRIX, lightViewMatrixTemp);
        Mat4x4f lightViewMatrix(lightViewMatrixTemp); //convert to Mat4f4
            matrix
    glPopMatrix();
    return lightViewMatrix;
}


void ShadowMap::prepare_for_texture_render()
{
    glClear(GL_DEPTH_BUFFER_BIT);

    glViewport(0, 0, map_size, map_size);

    glDepthFunc(GL_LESS);

    //Draw back faces into the shadow map
    glEnable(GL_CULL_FACE);
    glCullFace(GL_BACK);

    //Disable color writes, and use flat shading for speed
    glShadeModel(GL_FLAT);
    glColorMask(0,0,0,0);

    //Disable Lighting
    glDisable(GL_LIGHTING);

    glEnable(GL_POLYGON_OFFSET_POINT );
    glEnable(GL_POLYGON_OFFSET_FILL);

    glPolygonOffset(factor, units);

    char buf[50];
```

```
    sprintf(buf, "Bias: ON      Factor: %.1f      Units: %.1f", factor, units)
        ;
    displaytext.updateText(biastext, buf);
}

void ShadowMap::restore_states()
{
    //restore states
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_CULL_FACE);
    glShadeModel(GL_SMOOTH);
    glColorMask(1, 1, 1, 1);
    glViewport(0, 0, vpWidth, vpHeight);
}

void ShadowMap::copy_drawing_buffer_to_texture(GLuint tex_map)
{
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, tex_map);
    glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, 0, 0,map_size,
        map_size, 0);
}

void ShadowMap::init_shadow_texture()
{
    for(int i=0; i<lights->no_lights(); i++)
    {

        glGenTextures(1, &shadowMapTexture[i]);
        glBindTexture(GL_TEXTURE_2D, shadowMapTexture[i]);
        if(!bPCF){
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST
                ); //GL_LINEAR
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST
                );
        }
        else{
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
                ; //GL_LINEAR
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
                ;
        }
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    }
}

void ShadowMap::show_shadow_map(GLuint tex_map, int _i)
{
    glViewport(0+200*_i,0,200,200);

    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(-1,1,-1,1);

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();

    glBindTexture(GL_TEXTURE_2D, tex_map);
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
```

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE_ARB, GL_NONE);
    glEnable(GL_TEXTURE_2D);
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_R);
    glDisable(GL_TEXTURE_GEN_T);
    glDisable(GL_TEXTURE_GEN_Q);
    glDisable(GL_LIGHTING);
    glBegin(GL_POLYGON);
        glColor3f(0.0, 1.0, 1.0);
        glTexCoord2f(0.0, 0.0);
        glVertex2f(-1,-1);
        glTexCoord2f(0.0, 1.0);
        glVertex2f(-1, 1);
        glTexCoord2f(1.0, 1.0);
        glVertex2f(1, 1);
        glTexCoord2f(1.0, 0.0);
        glVertex2f(1, -1);
    glEnd();
    glEnable(GL_LIGHTING);
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glDisable(GL_TEXTURE_2D);

    glMatrixMode( GL_PROJECTION );
    glPopMatrix();

    glMatrixMode( GL_MODELVIEW );
    glPopMatrix();

    glViewport(0,0,vpWidth,vpHeight);

}

void ShadowMap::show_misc_info(int window){}
```

C.6 Shadow Volumes (Z-Pass)

```
/****************************************************************/
/* filename: ShadowVolume.h                                     */
/****************************************************************/
#ifndef __SHA_VOL_H__
#define __SHA_VOL_H__

#include "My/RenderAlgorithm.h"

class ShadowVolume : public RenderAlgorithm
{
private:
    vector<Vec3i> adjacent;
    bool draw_sv;
    bool bShowStencil;
    int shavoltext;
public:
    ShadowVolume(Object* _obj, LightContainer* _lights, Camera* _cam, Object
        * _floor) : RenderAlgorithm(_obj, _lights, _cam, _floor){

        /* We make a list of adjacent faces */
        init();
        bShowStencil = false;
        draw_sv = false;
        shavoltext = displaytext.addText("");
        char buf[50];
        sprintf(buf, "Draw ShadowVolumes(t): OFF");
```

```
        displaytext.updateText(shavoltext, buf);
    };

    void init();
    void render();

    void make_adjacent_list();
    void draw_shadow_volumes(Light* light);
    bool is_backfacing(Light* light, int mesh_index, int face_index);
    void shadow_volumes_on(){
        draw_sv = true;
    }
    void shadow_volumes_off(){
        draw_sv = false;
    }

    void shadow_volumes_toggle(){
        draw_sv = !draw_sv;
        cout << "draw_sv : " << draw_sv << endl;
    }

    void process_key(unsigned char key)
    {
        Vec3f p0;
        Vec3f p7;
        switch(key) {
            case 't':
                shadow_volumes_toggle();
                break;
            case 'y':
                bShowStencil = !bShowStencil;
                break;
            default:
                RenderAlgorithm::process_key(key);
                break;
        }
    }
};
#endif

/*****************************************************************/
/* filename: ShadowVolume.cpp                                    */
/*****************************************************************/
#include "My/ShadowVolume.h"

void ShadowVolume::render()
{
    /* Clear window */
    clear_all();

    // ****************************************************************************
    // Pass 1: Render the scene in shadow
    // ****************************************************************************

    load_projection(cam);
    load_modelview(cam);
    lights->turn_all_off();
    update_lights();
    glCallList(dlDrawScene);
```

```
// ******************************************************************************************
// Pass 2: Draw shadow volumes
// ******************************************************************************************

glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );
glDepthMask( GL_FALSE );

// Setup stencil-test
glEnable( GL_CULL_FACE );
glClear(GL_STENCIL_BUFFER_BIT);
glEnable( GL_STENCIL_TEST );
glStencilFunc( GL_ALWAYS, 0, 0 );

for(int i=0; i<lights->no_lights(); i++)
{
    //generate front-facing shadow polygons.
    glStencilOp( GL_KEEP, GL_KEEP, GL_INCR );
    glCullFace( GL_BACK );
    draw_shadow_volumes(lights->get_light(i));

    //generate back-facing shadow polygons.
    glStencilOp( GL_KEEP, GL_KEEP, GL_DECR );
    glCullFace( GL_FRONT );
    draw_shadow_volumes(lights->get_light(i));
}

if(bShowStencil){
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColorMask(1,1,1,1);
    glDepthMask(0);

    glEnable(GL_STENCIL_TEST);
    glStencilFunc(GL_EQUAL, 0, 1);
    glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-1,1,-1,1);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,1,0,0,0,0,1,0);


    glColor3f(1,1,1);
    glBegin(GL_QUADS);
        glVertex3f(-1, -1, 0.5);
        glVertex3f(-1,  1, 0.5);
        glVertex3f( 1,  1, 0.5);
        glVertex3f( 1, -1, 0.5);

    glEnd();
}
// When done, set the states back to something more typical.
glDepthMask( GL_TRUE );
glDepthFunc( GL_LEQUAL );
glColorMask( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE );
glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );
glDisable( GL_CULL_FACE );

if(!bShowStencil){
```

```
        // Render the lit part...where stencilbuffer = 0
        glStencilFunc( GL_EQUAL, 0, 0xffff);
        glEnable(GL_LIGHTING);

        /* Set lights to their original values */
        lights->turn_all_on();

        glBlendFunc(GL_SRC_COLOR, GL_ONE);
        load_projection(cam);
        load_modelview(cam);
        update_lights();
        glCallList(dlDrawScene);

        // When done, set the states back to something more typical.
        glDisable( GL_STENCIL_TEST);
        glDisable(GL_BLEND);

        if(draw_sv) //
        {
            for(int i=0; i<lights->no_lights(); i++){
                glDisable(GL_LIGHTING);
                glEnable(GL_CULL_FACE);
                glDepthFunc(GL_LEQUAL);

                glEnable(GL_POLYGON_OFFSET_LINE);
                glPolygonOffset(-0.5,0.0);
                glDisable(GL_BLEND);
                glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
                glColor3f(1.0, 1.0, 0.0);
                glCullFace(GL_FRONT);
                draw_shadow_volumes(lights->get_light(i));

                glCullFace(GL_BACK);
                draw_shadow_volumes(lights->get_light(i));

                glEnable(GL_BLEND);
                glBlendFunc(GL_ONE,GL_ONE);
                glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
                glColor3f(0.1, 0.1, 0.0);
                glCullFace(GL_FRONT);
                draw_shadow_volumes(lights->get_light(i));

                glCullFace(GL_BACK);
                draw_shadow_volumes(lights->get_light(i));
            }
            char buf[50];
            sprintf(buf, "Draw ShadowVolumes(t): ON");
            displaytext.updateText(shavoltext, buf);
        }
        else
        {
            char buf[50];
            sprintf(buf, "Draw ShadowVolumes(t): OFF");
            displaytext.updateText(shavoltext, buf);
        }

        RenderAlgorithm::render();
    } //end of bShowStencil
}

void ShadowVolume::make_adjacent_list()
```

```
{
    cout << "Generating adjacent list" << endl;
    //For each mesh
    for(int mesh=0; mesh<obj->no_meshes(); mesh++)
    {
        TriMesh* cur_mesh = obj->get_mesh(mesh);
        if(!cur_mesh->adjInitialized)
        {
            cur_mesh->adjacents.resize(cur_mesh->no_faces());

            //For each face
            for(int i=0; i<cur_mesh->no_faces(); i++)
            {
                int found_tris = 0;
                for(int j=0; j<cur_mesh->no_faces(); j++)
                {
                    if(i != j){
                        // if face(i) shares two vertices with face(j)
                        int shared_vertices = 0;
                        for(int k=0; k<3; k++)
                        {
                            for(int l=0; l<3; l++)
                            {
                                //if(cur_mesh->get_face(i)[k] == cur_mesh->
                                    get_face(j)[l])
                                //  shared_vertices++;
                                if(cur_mesh->get_face(i)[k] == cur_mesh->
                                    get_face(j)[(l+1)%3] && cur_mesh->
                                    get_face(i)[(k+1)%3] == cur_mesh->
                                    get_face(j)[l])
                                {
                                    (cur_mesh->adjacents[i])[k] = j;
                                }
                            }
                        }
                        if(shared_vertices >= 2){
                            (cur_mesh->adjacents[i])[found_tris] = j;
                            found_tris++;
                        }
                    }
                }
            }
            cur_mesh->adjInitialized = true;
        }
        else{
            cout << " -> Already initialized" << endl;
        }

    }
    cout << " -> done!" << endl;
}


bool ShadowVolume::is_backfacing(Light* light, int mesh_index, int
    face_index){

        Vec3f face_avg_pos = (obj->get_mesh(mesh_index)->get_face_vertex(
            face_index,0)
                        + obj->get_mesh(mesh_index)->get_face_vertex(
                            face_index,1)
                        + obj->get_mesh(mesh_index)->get_face_vertex(
                            face_index,2))/3.0;
```

```
        Vec3f light_dir = light->get_position() - face_avg_pos;

        if(dot(normalize(light_dir), obj->get_mesh(mesh_index)->
            get_face_normal(face_index)) <= 0)
            return true;
        else

        return false;
    }
void ShadowVolume::draw_shadow_volumes(Light* light) //, TriMesh* TM, vector
    <Light>* lights
{

    //For all meshes
    for(int k=0; k<obj->no_meshes(); k++)
    {
        // For all triangles
        for(int i=0; i<obj->get_mesh(k)->no_faces(); i++)
        {
            TriMesh* mesh = obj->get_mesh(k);
            // If triangle is backfacing
            if(is_backfacing(light, k, i))
            {
                //for each edge
                for(int j=0; j<3; j++)
                {
                    //if adjacent face is front-facing, edge is silhouette
                        edge
                    int adj = (mesh->adjacents[i])[j];
                    if(!is_backfacing(light, k, adj))
                    {
                        // then we extrude and draw the edge
                        Vec3f p0 = mesh->get_face_vertex(i,j);
                        Vec3f p1 = p0 + 1000.0 * normalize(p0 - light->
                            get_position());
                        Vec3f p3 = mesh->get_face_vertex(i,(j+1)%3);
                        Vec3f p2 = p3 + 1000.0 * normalize(p3 - light->
                            get_position());

                        glPushMatrix();
                            glShadeModel (GL_FLAT);
                            //glEnable (GL_BLEND);
                            glDisable (GL_LIGHTING);
                            //glBlendFunc (GL_SRC_ALPHA,
                                GL_ONE_MINUS_SRC_ALPHA);
                            //glBlendFunc(GL_ONE,GL_ONE);
                            glNormal3f(0.0, 1.0, 0.0);
                            glBegin(GL_QUADS);

                                glVertex3fv(p3.get());
                                glVertex3fv(p2.get());
                                glVertex3fv(p1.get());
                                glVertex3fv(p0.get());
                            glEnd();

                        glPopMatrix();

                    }
                }
                glPushMatrix();
                glShadeModel (GL_FLAT);
```

```
                //glEnable (GL_BLEND);
                glDisable (GL_LIGHTING);
                //glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
                //glBlendFunc(GL_ONE,GL_ONE);
                glBegin(GL_TRIANGLES);
                    glVertex3fv(obj->get_mesh(k)->get_face_vertex(i,2).get()
                        );
                    glVertex3fv(obj->get_mesh(k)->get_face_vertex(i,1).get()
                        );
                    glVertex3fv(obj->get_mesh(k)->get_face_vertex(i,0).get()
                        );
                glEnd();
                glPopMatrix();

                //glDisable (GL_BLEND);
                glShadeModel (GL_SMOOTH);

            }
        }
    }
}

void ShadowVolume::init()
{
    make_adjacent_list();

    glClearColor (0.0, 0.0, 0.0, 1.0);
    glShadeModel (GL_SMOOTH);

    /* Enable depth test and stencil test */
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_STENCIL_TEST);

    /* Enable lighting */
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    RenderAlgorithm::init();
}
```

C.7 Other Classes

C.7.1 Main.cpp

```
/*****************************************************************/
/* Filename: Main.cpp                                            */
/*****************************************************************/

#include <iostream>
#include <string>
#include <vector>

#include "main.h"
#include "My/RenderAlgorithm.h"
#include "My/ShadowVolume.h"
#include "My/ShadowVolumeZFail.h"
#include "My/ShadowMap.h"
#include "My/ShadowMapPBuf.h"
#include "My/PerspectiveShadowMapv2.h"
#include "My/Chan2.h"
#include "My/BasicRenderer.h"
#include "GL/glut.h"
```

```
#include "My/scene_loader.h"

/* globals */
#ifdef REC_TIME
    int window_width = 1280;
    int window_height = 1024;
#else
    int window_width = 1280;
    int window_height = 1024;
#endif

vector<string> files;
bool second_window = false;

/* for trackball */
float last_x = 0.0;      // last known x-position
float last_y = 0.0;      // last known y-position
enum button_state{LEFT, MIDDLE, RIGHT};
button_state bs = LEFT;

/* GLUT functions */
void init_glut(int argc,char**argv);
void display();
void display_misc();
void reshape (int w, int h);
void animate();
void moveCamera( int x, int y);
void mouse(int button, int state, int x, int y);
void keyboard(unsigned char key, int x, int y);
void keyboard_special_key(int key, int x, int y);
void menu_clicked(int i);
void create_menus();
int main_window, misc_window;

RenderAlgorithm* render_algorithm;
RenderAlgorithm* basic_renderer;
RenderAlgorithm* shadow_volumes;
RenderAlgorithm* shadow_volumes_zfail;
RenderAlgorithm* shadow_volumes_cg;
RenderAlgorithm* shadow_maps;
RenderAlgorithm* shadow_maps_pbuf;
RenderAlgorithm* perspective_shadow_maps;
RenderAlgorithm* chans;

Camera* cam;
Camera* cam1;
Camera* view_cam;
LightContainer* lc;
BMesh::Object* obj;
vector<BMesh::Object*> scenes;

int main(int argc,char**argv){
    for(int i=0; i<6; i++){
        BMesh::Object* obj_temp = new BMesh::Object();
        scenes.push_back(obj_temp);
    }
    MySceneLoader msl;
    msl.load_box_grid(scenes[0]);
    msl.load_Rocks(scenes[1]);
    msl.load_Street(scenes[2]);
    msl.load_room(scenes[3]);
    msl.load_single_mesh(scenes[4], "../resource_files/models/Water.obj");
```

```
obj = scenes[0];

// The floor is added
BMesh::Object* floor = new BMesh::Object();
BMesh::obj_load("../resource_files/models/Floor/floor_500_100.obj", *
    floor);

// Color is added to the vertices of the mesh. All vertices same color
for(int s=0; s<scenes.size(); s++){
    for(int j=0; j<scenes[s]->no_meshes(); j++){
        for(int i=0; i<scenes[s]->get_mesh(j)->no_vertices(); i++){
                scenes[s]->get_mesh(j)->add_cvertex(Vec3f(0.8, 0.4, 0.0)
                    );
        }
    }
}

for(int i=0; i<floor->get_mesh(0)->no_vertices(); i++){
        floor->get_mesh(0)->add_cvertex(Vec3f(0.6,0.6,0.6));
}

/* Add a camera */
cam = new Camera(100,100,100,0,0,0,0,1,0,20.26);
cam->set_far(10000);

/* Add a LightContainer and lights*/
lc = new LightContainer();
Light* light = new SpotLight(Vec3f(-30,100,40), Vec3f(0,0,0), Vec3f
    (0,1,0), Vec4f(0.8, 0.8, 0.8, 1.0), Vec4f(0.8, 0.8, 0.8, 1.0), Vec4f
    (0.0,0.0,0.0,1.0));
lc->add_light(light);

/* Initialize GLUT */
init_glut(argc, argv);
// The OpenGL state is saved so that we can pop it before using
    different algorithms
glPushAttrib(GL_ALL_ATTRIB_BITS);

// The different rendering algorithms are initialized
basic_renderer            = new BasicRenderer(obj, lc, cam, floor);
shadow_volumes_zfail      = new ShadowVolumeZFail(obj, lc, cam, floor)
    ;
shadow_volumes            = new ShadowVolume(obj, lc, cam, floor);
shadow_maps               = new ShadowMap(obj, lc, cam, floor);
shadow_maps_pbuf          = new ShadowMapPBuf(obj, lc, cam, floor);
perspective_shadow_maps   = new PerspectiveShadowMapCF(obj, lc, cam,
    floor);
chans                     = new Chan(obj, lc, cam, floor);

// We set the active algorithm
render_algorithm = shadow_volumes_zfail;

// Stupid way to select scene...but it works
// Scenes are 300 - BoxGrid
//            310 - StoneHenge
//            320 - Street
//            330 - Room
//            340 - WaterTower
menu_clicked(340);

glutMainLoop();
return 0;
```

```
}

void init_glut(int argc,char**argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_STENCIL | GLUT_RGBA);
    glutInitWindowSize (window_width, window_height);
    glutInitWindowPosition (10, 10);
    main_window = glutCreateWindow ("Basic");
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutIdleFunc(animate);
    glutKeyboardFunc(keyboard);
    glutSpecialFunc(keyboard_special_key);
    glutMotionFunc(moveCamera);
    glutMouseFunc(mouse);

    create_menus();

    if(second_window){
        // window for misc info
        glutInitWindowSize (500, 500);
        glutInitWindowPosition (100, 100);
        misc_window = glutCreateWindow ("Misc Info");
        glutDisplayFunc(display_misc);
        glutSetWindow(main_window);
    }
}

void display()
{
    //glutSetWindow(main_window);
    render_algorithm->render();
    glutSwapBuffers();
}

void display_misc()
{
}

void reshape (int w, int h)
{
    glutSetWindow(main_window);
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    cam->set_aspect((float)w/(float)h);
    render_algorithm->set_vp((float)w, (float)h);
    glutPostRedisplay();

    if(second_window){
        glutSetWindow(misc_window);
        glViewport (0, 0, 500, 500);
        glutPostRedisplay();
    }
}

void animate()
{
    render_algorithm->animate_lights();
    glutSetWindow(main_window);
    glutPostRedisplay();
    if(second_window){
        glutSetWindow(misc_window);
```

```
        glutPostRedisplay();
    }
}
void moveCamera( int x, int y)
{
    if(bs == LEFT)
    {
        if(x > last_x)
            cam->rotate_right();
        if(x < last_x)
            cam->rotate_left();
        if(y > last_y)
            cam->rotate_up();
        if(y < last_y)
            cam->rotate_down();
    }
    if(bs == MIDDLE)
    {
        if(y > last_y)
            cam->zoom_out();
        if(y < last_y)
            cam->zoom_in();
    }
    if(bs == RIGHT)
    {

        if(x > last_x)
            cam->rotate_pov_right();

        if(x < last_x)
            cam->rotate_pov_left();
        if(y > last_y)
            cam->rotate_pov_up();
        if(y < last_y)
            cam->rotate_pov_down();
    }
    last_x = x;
    last_y = y;

    glutSetWindow(main_window);
    glutPostRedisplay();

    if(second_window){
        glutSetWindow(misc_window);
        glutPostRedisplay();
    }
}

void mouse(int button, int state, int x, int y)
{
    if(state == GLUT_DOWN && button == GLUT_LEFT_BUTTON)
    {
        bs = LEFT;
        cout << "Left" << endl;
    }
    if(state == GLUT_DOWN && button == GLUT_MIDDLE_BUTTON)
    {
        bs = MIDDLE;
        cout << "Middle" << endl;
    }
    if(state == GLUT_DOWN && button == GLUT_RIGHT_BUTTON)
    {
```

```
        bs = RIGHT;
        cout << "Right" << endl;
    }

}
void keyboard(unsigned char key, int x, int y)
{
    render_algorithm->process_key(key);
    glutPostRedisplay();
}

void keyboard_special_key(int key, int x, int y)
{
    cout << "Special key pressed:" << key << endl;
    if(key == GLUT_KEY_RIGHT)
    {
        cam->rotate_pov_right();
    }
    else if(key == GLUT_KEY_LEFT)
    {
        cam->rotate_pov_left();
    }
    else if(key == GLUT_KEY_UP)
    {
        cam->rotate_pov_down();
    }
    else if(key == GLUT_KEY_DOWN)
    {
        cam->rotate_pov_up();
    }
}

void create_menus()
{
    int alg_menu = glutCreateMenu(menu_clicked);
    glutAddMenuEntry("Basic Renderer", 1);
    glutAddMenuEntry("Shadow Volumes", 2);
    glutAddMenuEntry("Shadow Volumes ZFail", 22);
    glutAddMenuEntry("Shadow Maps", 3);
    glutAddMenuEntry("Shadow Maps Pbuffer", 33);
    glutAddMenuEntry("Perspective Shadow Maps", 4);
    glutAddMenuEntry("Chan's hybrid", 5);

    int sublight_menu[2];
    for(int i=0; i<lc->no_lights(); i++){
        sublight_menu[i] = glutCreateMenu(menu_clicked);
        glutAddMenuEntry("Activate", 210+i);
    }
    int light_menu = glutCreateMenu(menu_clicked);
    for(int i=0; i<lc->no_lights(); i++){
        char strL[100];
        sprintf(strL, "Light %d\0", i);
        int index = 50+i;
        glutAddSubMenu(strL, sublight_menu[i]);
    }

    int model_menu = glutCreateMenu(menu_clicked);
    glutAddMenuEntry("Box Grid", 300);
    glutAddMenuEntry("Stone Henge", 310);
    glutAddMenuEntry("Street", 320);
    glutAddMenuEntry("Room", 330);
    glutAddMenuEntry("WaterTower", 340);
```

```
    int top_menu = glutCreateMenu(menu_clicked);
    glutAddSubMenu("Algorithm", alg_menu);
    glutAddSubMenu("Lights", light_menu);
    glutAddSubMenu("Scene", model_menu);

    glutAttachMenu(GLUT_RIGHT_BUTTON);
}

void menu_clicked(int i)
{
    cout << "Menu clicked:" << i << endl;
    switch(i)
    {
    case 1:
        glPopAttrib();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        render_algorithm = basic_renderer;
        glutSetWindowTitle("Basic Renderer");
        render_algorithm->init();
        break;
    case 2:
        glPopAttrib();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        render_algorithm = shadow_volumes;
        glutSetWindowTitle("Shadow Volumes");
        render_algorithm->init();
        break;
    case 22:
        glPopAttrib();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        render_algorithm = shadow_volumes_zfail;
        glutSetWindowTitle("ZFail Shadow Volumes");
        render_algorithm->init();
        break;
    case 3:
        glPopAttrib();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        render_algorithm = shadow_maps;
        glutSetWindowTitle("Shadow Maps (Copy To Texture)");
        render_algorithm->init();
        break;
    case 33:
        glPopAttrib();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        render_algorithm = shadow_maps_pbuf;
        glutSetWindowTitle("Shadow Maps (Render To Texture)");
        render_algorithm->init();
        break;
    case 4:
        glPopAttrib();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        render_algorithm = perspective_shadow_maps;
        glutSetWindowTitle("Perspective Shadow Maps");
        render_algorithm->init();
        break;
    case 5:
        glPopAttrib();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        render_algorithm = chans;
        glutSetWindowTitle("Chan's Hybrid Algorithm");
        render_algorithm->init();
```

```
        break;
    case 210: //Light 0
        lc->set_active_light(0);
        break;
    case 211: //Light 1
        lc->set_active_light(1);
        break;
    case 220:
        glPopAttrib();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        render_algorithm->init();
        break;
    case 230:

    case 221:
        glPopAttrib();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        lc->delete_light(lc->get_light(0));
        render_algorithm->init();
        break;
    case 300:
        glPopAttrib();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        lc->get_light(0)->set_position(-114,100,122);
        lc->get_light(0)->set_cutoff_angle(27);
        lc->get_light(0)->set_near(170);
        lc->get_light(0)->set_far(210);
        cam->set_position(40.4, 17.9, 11.2);
        cam->set_direction(19.9,0,-1.8);
        cam->set_fov(35);
        render_algorithm->change_scene(scenes[0]);
        break;
    case 310:
        glPopAttrib();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        lc->get_light(0)->set_position(128,174,237);
        lc->get_light(0)->set_cutoff_angle(50);
        lc->get_light(0)->set_near(155);
        lc->get_light(0)->set_far(408);
        cam->set_position(-212.8, 45.6, 103.9);
        cam->set_direction(-41.5,0,24.7);
        cam->set_fov(20.26);
        render_algorithm->change_scene(scenes[1]);
        break;
    case 320:
        glPopAttrib();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        lc->get_light(0)->set_position(197,174,39);
        lc->get_light(0)->set_cutoff_angle(50);
        lc->get_light(0)->set_near(155);
        lc->get_light(0)->set_far(408);
        cam->set_position(1.8, 4.4, 108.7);
        cam->set_direction(-7.8,0,-145.6);
        cam->set_fov(35);
        render_algorithm->change_scene(scenes[2]);
        break;
    case 330:
        glPopAttrib();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        lc->get_light(0)->set_position(0,37.1,0.1);
        lc->get_light(0)->set_cutoff_angle(80);
        lc->get_light(0)->set_near(3);
```

```
        lc->get_light(0)->set_far(40);
        cam->set_position(-10.2, 37.4, 94.4);
        cam->set_direction(1,0,-16);
        cam->set_fov(35);
        render_algorithm->change_scene(scenes[3]);
        break;
    case 340:
        glPopAttrib();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        lc->get_light(0)->set_position(54,204,-139.9);
        lc->get_light(0)->set_cutoff_angle(58);
        lc->get_light(0)->set_near(134);
        lc->get_light(0)->set_far(266);
        cam->set_position(-293, 187, 80);
        cam->set_direction(78.4,0,102.2);
        cam->set_fov(35);
        render_algorithm->change_scene(scenes[4]);
        break;
    default:
        break;
    }
}
```

C.7.2 RenderAlgorithm

```
/*****************************************************************/
/* filename: RenderAlgorithm.h                                   */
/*****************************************************************/

#ifndef __RA_H__
#define __RA_H__

#include <main.h>
#include <iostream>
#include "BMesh/Object.h"
#include "CGLA/Mat4x4f.h"
#include "CGLA/Vec4f.h"
#include <Graphics/DisplayText.h>

#if defined USE_RT
    #include <RenderTexture.h>
#else
    #include <ExtGL/extgl.h>
#endif
#include <GL/glut.h>
#include "LightContainer.h"
#include "Camera.h"
#include "My/bbox.h"
#include "My/bbox2.h"
#include "My/MyCGLA.h"
#include "Components/Timer.h"
#include "My/TimeToExcel.h"

//using namespace GFX;
using namespace std;
using namespace BMesh;

class MyQuad{
private:
    vector<Vec4f> p;
    Vec3f normal;
```

```
public:
    MyQuad(){};

    void add_point(Vec4f _p){
        p.push_back(_p);
    }

    void set_normal(Vec3f _n){
        normal = _n;
    }

    Vec4f get_vertex(int i){
        return p[i];
    }

    Vec3f get_normal(){
        return normal;
    }
};


class RenderAlgorithm{
protected:
    static Object* obj;
    Object* floor;
    Camera* cam;
    Camera* view_cam;
    LightContainer* lights;
    DisplayText displaytext;
    Components::Timer timer;

    // For shadow volumes and chan
    vector<MyQuad> shadow_quads;

    //Viewport parameters
    static float vpWidth, vpHeight, aspect;

    //Display List draw_scene
    static GLuint dlDrawScene;

public:
    RenderAlgorithm(Object* _obj, LightContainer* _lights, Camera* _cam,
        Object* _floor){
        cout << "RenderAlgorithm created" << endl;
        obj = _obj;
        floor = _floor;
        cam = _cam;
        view_cam = _cam;
        lights = _lights;

        float vp[4];
        glGetFloatv(GL_VIEWPORT, vp);
        vpWidth  = vp[2];
        vpHeight = vp[3];
        aspect = vpWidth/vpHeight;

        /* Check that normals are present for all meshes */
        cout << "Checking normals" << endl;
        for(int i=0; i<obj->no_meshes(); i++){
            obj->get_mesh(i)->check_normals();
        }
```

```
        floor->get_mesh(0)->check_normals();

        for(int i=0; i<lights->no_lights(); i++){
            //update_frustum(lights->get_light(i));
        }

        glEnable(GL_DEPTH_TEST);
        displaytext.addFramerate();
    };
    RenderAlgorithm(){};

    virtual void render(){
#ifdef SHOW_MISC_INFO
        glViewport(0,0,vpWidth, vpHeight);
        for(int i=0; i<lights->no_lights(); i++)
            show_light_frustum(lights->get_light(i));
        show_marker();
        draw_axis();
        draw_unit_cube();
        displaytext.draw();
#endif

    };
    virtual void shadow_volumes_toggle(){};
    virtual void show_shadow_map(GLuint tex_map){};
    virtual void show_misc_info(int window){};
    virtual void init(){
        /* Check that normals are present for all meshes */
        cout << "Checking normals" << endl;
        for(int i=0; i<obj->no_meshes(); i++){
            obj->get_mesh(i)->check_normals();
        }

        floor->get_mesh(0)->check_normals();

        glDeleteLists(dlDrawScene,1);
        dlDrawScene = glGenLists(1);
        glNewList(dlDrawScene, GL_COMPILE);
        draw_scene();
        glEndList();
    };

    virtual void process_key(unsigned char key);
    virtual void increase_bias_factor(float amount=0.1){};
    virtual void decrease_bias_factor(float amount=0.1){};
    virtual void increase_bias_units(float amount=0.1){};
    virtual void decrease_bias_units(float amount=0.1){};

    void update_frustum(Light* light);
    void animate_lights(){
        lights->animate_all();
    }

    /* draw functions */
    void draw_scene(Camera* cam);
    void draw_scene();
    void draw_objects();
    //void draw_floor(int w, int h, int s);
    void draw_floor();
    void draw_point(Vec3f* p);
    void load_projection(Camera* _cam);
    Mat4x4f get_cam_projection_matrix(Camera* _cam);
```

```
    Mat4x4f get_cam_modelview_matrix(Camera* _cam);
    void load_modelview(Camera* _cam);
    void update_lights();
    void draw_axis();
    void draw_unit_cube();
    void draw_bbox(bbox bb);
    void draw_bbox(bbox2 bb);
    void show_light_frustum(Light* light);
    void show_cam_frustum(Camera* cam);
    void show_marker();

    /* print functions */
    virtual void print_all(){
        cam->print_all();
        lights->print_all();
        Vec3f p0, p7;
        obj->get_mesh(0)->get_bbox(p0, p7);
        cout << "BBox: " << p0 << " : " << p7 << endl;
    }

    /* set function */
    virtual void set_map_size(int i){};
    void set_vp(float w, float h){
        vpWidth = w;
        vpHeight = h;
    }

    void change_scene(Object* _obj){
        obj = _obj;
        init();
    }

    float get_vpWidth(){return vpWidth;}
    float get_vpHeight(){return vpHeight;}
    void set_aspect(float a){aspect = a;}
    void clear_all();

    /* misc function */
    Vec3f multM4V3(Mat4x4f* M, Vec3f* p){
        Vec4f p4(*p, 1.0);
        p4 = *M * p4;
        return Vec3f(p4[0]/p4[3], p4[1]/p4[3], p4[2]/p4[3]);
    }
};
#endif

/******************************************************************/
/* filename: RenderAlgorithm.cpp                                  */
/******************************************************************/
#include "My/RenderAlgorithm.h"

#define PI 3.14159

float RenderAlgorithm::vpWidth;
float RenderAlgorithm::vpHeight;
float RenderAlgorithm::aspect;
GLuint RenderAlgorithm::dlDrawScene = 1;
Object* RenderAlgorithm::obj;

void RenderAlgorithm::draw_scene(){
    draw_floor();
    draw_objects();
```

```
}

void RenderAlgorithm::draw_objects()
{
    float zero[4] = {0.0,0.0,0.0,1.0};
    // Draw all triangles
    glPushMatrix();
        glEnable(GL_NORMALIZE);
        for(int j=0; j<obj->no_meshes(); j++)
        {
            for(int i=0; i<obj->get_mesh(j)->no_faces(); i++)
            {
                glColor3fv(obj->get_mesh(j)->get_cvertex(obj->get_mesh(j)->
                    get_face(i)[1]).get());
                glBegin(GL_TRIANGLES);

                    glMaterialfv(GL_FRONT, GL_DIFFUSE, obj->get_mesh(j)->
                        get_cvertex(obj->get_mesh(j)->get_face(i)[1]).get())
                        ;
                    glMaterialfv(GL_FRONT, GL_AMBIENT, zero);
                    glMaterialfv(GL_FRONT, GL_SPECULAR, obj->get_mesh(j)->
                        get_cvertex(obj->get_mesh(j)->get_face(i)[1]).get()
                         );
                    glMaterialf(GL_FRONT, GL_SHININESS, 50.0);

                    bool flat_shading = true;

                    if(!flat_shading) //VertexNormals
                        glNormal3fv((obj->get_mesh(j))->get_normal((obj->
                            get_mesh(j))->get_face(i)[0]).get());
                    else //Face Normals
                        glNormal3fv((obj->get_mesh(j))->get_face_normal(i).
                            get());
                    glVertex3fv(((obj->get_mesh(j)->get_face_vertex(i,0)).
                        get()));

                    if(!flat_shading) //VertexNormals
                        glNormal3fv((obj->get_mesh(j))->get_normal((obj->
                            get_mesh(j))->get_face(i)[1]).get());
                    else //Face Normals
                        glNormal3fv((obj->get_mesh(j))->get_face_normal(i).
                            get());
                    glVertex3fv(((obj->get_mesh(j)->get_face_vertex(i,1)).
                        get()));

                    if(!flat_shading) //VertexNormals
                        glNormal3fv((obj->get_mesh(j))->get_normal((obj->
                            get_mesh(j))->get_face(i)[2]).get());
                    else //Face Normals
                        glNormal3fv((obj->get_mesh(j))->get_face_normal(i).
                            get());
                    glVertex3fv(((obj->get_mesh(j)->get_face_vertex(i,2)).
                        get()));
                glEnd();
            }
        }
        glDisable(GL_NORMALIZE);
    glPopMatrix();
};

void RenderAlgorithm::draw_floor(){
```

```
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
        glEnable(GL_NORMALIZE);

        for(int j=0; j<floor->no_meshes(); j++)
        {
            for(int i=0; i<floor->get_mesh(j)->no_faces(); i++)
            {
                glColor3fv(floor->get_mesh(j)->get_cvertex(floor->get_mesh(j
                    )->get_face(i)[1]).get());
                glBegin(GL_TRIANGLES);

                    glMaterialfv(GL_FRONT, GL_DIFFUSE, floor->get_mesh(j)->
                        get_cvertex(floor->get_mesh(j)->get_face(i)[1]).get
                        ());
                    glMaterialfv(GL_FRONT, GL_AMBIENT, floor->get_mesh(j)->
                        get_cvertex(floor->get_mesh(j)->get_face(i)[1]).get
                        ());
                    glMaterialfv(GL_FRONT, GL_SPECULAR, floor->get_mesh(j)->
                        get_cvertex(floor->get_mesh(j)->get_face(i)[1]).get
                        () );
                    glMaterialf(GL_FRONT, GL_SHININESS, 100.0);

                    glTexCoord2f(0,0);
                    glNormal3fv((floor->get_mesh(j))->get_normal((floor->
                        get_mesh(j))->get_face(i)[0]).get());
                    glVertex3fv(((floor->get_mesh(j)->get_face_vertex(i,0)).
                        get()));

                    glTexCoord2f(0,1);
                    glNormal3fv((floor->get_mesh(j))->get_normal((floor->
                        get_mesh(j))->get_face(i)[1]).get());
                    glVertex3fv(((floor->get_mesh(j)->get_face_vertex(i,1)).
                        get()));

                    glTexCoord2f(1,0);
                    glNormal3fv((floor->get_mesh(j))->get_normal((floor->
                        get_mesh(j))->get_face(i)[2]).get());
                    glVertex3fv(((floor->get_mesh(j)->get_face_vertex(i,2)).
                        get()));
                glEnd();
            }
        }
        glDisable(GL_NORMALIZE);
    glPopMatrix();
}
void RenderAlgorithm::draw_point(Vec3f* p)
{
    glPushMatrix();
        glDisable(GL_LIGHTING);
        glColor3f(1,0,0);
        glutSolidSphere(10,10,10);
    glPopMatrix();
}
void RenderAlgorithm::load_projection(Camera* _cam)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(_cam->get_fov(), _cam->get_aspect(), _cam->get_near(),
        _cam->get_far());
};
Mat4x4f RenderAlgorithm::get_cam_projection_matrix(Camera* _cam)
```

```
{
    glMatrixMode (GL_PROJECTION);
    glPushMatrix();
        glLoadIdentity();
        gluPerspective(_cam->get_fov(), _cam->get_aspect(), _cam->get_near()
            , _cam->get_far());
        float cameraProjectionMatrixTemp[16];
        glGetFloatv(GL_TRANSPOSE_PROJECTION_MATRIX,
            cameraProjectionMatrixTemp);
        Mat4x4f cameraProjectionMatrix(cameraProjectionMatrixTemp);
    glPopMatrix();
    return cameraProjectionMatrix;
}
Mat4x4f RenderAlgorithm::get_cam_modelview_matrix(Camera* _cam)
{
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glPushMatrix();
    //Set up the camera
    gluLookAt(_cam->get_position()[0],
        _cam->get_position()[1],
        _cam->get_position()[2],
        _cam->get_direction()[0],
        _cam->get_direction()[1],
        _cam->get_direction()[2],
        _cam->get_up()[0],
        _cam->get_up()[1],
        _cam->get_up()[2]);

    float cameraModelviewMatrixTemp[16];
    glGetFloatv(GL_TRANSPOSE_MODELVIEW_MATRIX, cameraModelviewMatrixTemp);
    Mat4x4f cameraModelviewMatrix(cameraModelviewMatrixTemp);
    glPopMatrix();

    return cameraModelviewMatrix;
}
void RenderAlgorithm::load_modelview(Camera* _cam){
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    //Set up the camera
    gluLookAt(_cam->get_position()[0],
            _cam->get_position()[1],
            _cam->get_position()[2],
            _cam->get_direction()[0],
            _cam->get_direction()[1],
            _cam->get_direction()[2],
            _cam->get_up()[0],
            _cam->get_up()[1],
            _cam->get_up()[2]);
};

void RenderAlgorithm::update_lights()
{
    glEnable(GL_LIGHTING);
    for(int l=0; l<lights->no_lights(); l++)
    {
        Light* light = lights->get_light(l);
        Vec3f spot_dir = normalize(light->get_direction()-light->
            get_position());
        Vec4f pos = Vec4f(light->get_position(), 1.0);

        glEnable(GL_LIGHT0 + l);
```

```
        glLightfv(GL_LIGHT0 + l, GL_SPOT_DIRECTION, spot_dir.get());
        glLightf( GL_LIGHT0 + l, GL_SPOT_CUTOFF, light->get_cutoff_angle());
        glLightf( GL_LIGHT0 + l, GL_SPOT_EXPONENT, 0.0);
        glLightfv(GL_LIGHT0 + l, GL_POSITION, pos.get());
        glLightfv(GL_LIGHT0 + l, GL_DIFFUSE, (light->get_diffuse()/lights->
            no_lights()).get());
        glLightfv(GL_LIGHT0 + l, GL_SPECULAR, (light->get_specular()/lights
            ->no_lights()).get());
        glLightfv(GL_LIGHT0 + l, GL_AMBIENT, (light->get_ambient()/lights->
            no_lights()).get());
        float zero[] = {0.0,0.0,0.0,1};
        glLightModelfv(GL_LIGHT_MODEL_AMBIENT, zero);
    }
};
void RenderAlgorithm::show_light_frustum(Light* light)
{
    glDisable(GL_LIGHTING);
    glColor3f(1,1,0);

    /* Draw sphere at position */
    glPushMatrix();
        glTranslatef(light->get_position()[0], light->get_position()[1],
            light->get_position()[2]);
        glutSolidSphere(0.1, 6,6);
    glPopMatrix();

    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

    Vec3f dir = normalize(light->get_position() - light->get_direction());
    float near_dist = light->get_near();
    float far_dist = light->get_far();
    float x = tan(light->get_cutoff_angle() * (PI/180.0)) * near_dist;
    float x2 = tan(light->get_cutoff_angle() * (PI/180.0)) * far_dist;

    Vec3f v = normalize(cross(light->get_up(),dir));
    Vec3f u = normalize(cross(dir,v));
    Vec3f pos = light->get_position();

    Vec3f p0 = pos - near_dist*dir + x*u + x*v;
    Vec3f p1 = pos - near_dist*dir + x*u - x*v;
    Vec3f p2 = pos - near_dist*dir - x*u - x*v;
    Vec3f p3 = pos - near_dist*dir - x*u + x*v;
    Vec3f p4 = pos - far_dist*dir + x2*u + x2*v;
    Vec3f p5 = pos - far_dist*dir + x2*u - x2*v;
    Vec3f p6 = pos - far_dist*dir - x2*u - x2*v;
    Vec3f p7 = pos - far_dist*dir - x2*u + x2*v;


    /* draw near plane */
    glBegin(GL_POLYGON);
        glVertex3fv(p0.get());
        glVertex3fv(p1.get());
        glVertex3fv(p2.get());
        glVertex3fv(p3.get());
    glEnd();

    /* draw far plane */
    glBegin(GL_POLYGON);
        glVertex3fv(p4.get());
        glVertex3fv(p5.get());
        glVertex3fv(p6.get());
        glVertex3fv(p7.get());
```

```
    glEnd();

    /* draw sides */
    glBegin(GL_POLYGON);
        glVertex3fv(p0.get());
        glVertex3fv(p4.get());
        glVertex3fv(p5.get());
        glVertex3fv(p1.get());
    glEnd();

    glBegin(GL_POLYGON);
        glVertex3fv(p0.get());
        glVertex3fv(p3.get());
        glVertex3fv(p7.get());
        glVertex3fv(p4.get());
    glEnd();

    glBegin(GL_POLYGON);
        glVertex3fv(p1.get());
        glVertex3fv(p5.get());
        glVertex3fv(p6.get());
        glVertex3fv(p2.get());
    glEnd();

    glBegin(GL_POLYGON);
        glVertex3fv(p3.get());
        glVertex3fv(p7.get());
        glVertex3fv(p6.get());
        glVertex3fv(p2.get());
    glEnd();

    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
}

void RenderAlgorithm::show_cam_frustum(Camera* _cam)
{
    glDisable(GL_LIGHTING);

    Vec3f dir = normalize(_cam->get_direction() - _cam->get_position());
    float near_dist = _cam->get_near();
    float far_dist = _cam->get_far();
    float x = tan(_cam->get_fov()/2.0 * (PI/180.0)) * near_dist;
    float x2 = tan(_cam->get_fov()/2.0 * (PI/180.0)) * far_dist;

    Vec3f v = normalize(cross(_cam->get_up(),dir));
    Vec3f u = normalize(cross(dir,v));
    Vec3f pos = _cam->get_position();

    Vec3f p0 = pos + near_dist*dir + x*u + x*v;
    Vec3f p1 = pos + near_dist*dir + x*u - x*v;
    Vec3f p2 = pos + near_dist*dir - x*u - x*v;
    Vec3f p3 = pos + near_dist*dir - x*u + x*v;
    Vec3f p4 = pos + far_dist*dir + x2*u + x2*v;
    Vec3f p5 = pos + far_dist*dir + x2*u - x2*v;
    Vec3f p6 = pos + far_dist*dir - x2*u - x2*v;
    Vec3f p7 = pos + far_dist*dir - x2*u + x2*v;

    /* Color */
    glColor3f(0,1,0);

    /* Draw sphere at position */
    //glMatrixMode(GL_MODELVIEW);
```

```
    glPushMatrix();
        glTranslatef(_cam->get_position()[0], _cam->get_position()[1], _cam
            ->get_position()[2]);
        glutSolidSphere(0.1, 6,6);
    glPopMatrix();

    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    /* draw near plane */
    glBegin(GL_POLYGON);
        glVertex3fv(p0.get());
        glVertex3fv(p1.get());
        glVertex3fv(p2.get());
        glVertex3fv(p3.get());
    glEnd();

    /* draw far plane */
    glBegin(GL_POLYGON);
        glVertex3fv(p4.get());
        glVertex3fv(p5.get());
        glVertex3fv(p6.get());
        glVertex3fv(p7.get());
    glEnd();

    /* draw sides */
    glBegin(GL_POLYGON);
        glVertex3fv(p0.get());
        glVertex3fv(p4.get());
        glVertex3fv(p5.get());
        glVertex3fv(p1.get());
    glEnd();

    glBegin(GL_POLYGON);
        glVertex3fv(p0.get());
        glVertex3fv(p3.get());
        glVertex3fv(p7.get());
        glVertex3fv(p4.get());
    glEnd();

    glBegin(GL_POLYGON);
        glVertex3fv(p1.get());
        glVertex3fv(p5.get());
        glVertex3fv(p6.get());
        glVertex3fv(p2.get());
    glEnd();

    glBegin(GL_POLYGON);
        glVertex3fv(p3.get());
        glVertex3fv(p7.get());
        glVertex3fv(p6.get());
        glVertex3fv(p2.get());
    glEnd();
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
}

void RenderAlgorithm::draw_axis(){
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(cam->get_fov(), cam->get_aspect(), cam->get_near(), cam->
        get_far());

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
```

```
    //Set up the camera
    gluLookAt(cam->get_position()[0],
              cam->get_position()[1],
              cam->get_position()[2],
              cam->get_direction()[0],
              cam->get_direction()[1],
              cam->get_direction()[2],
              cam->get_up()[0],
              cam->get_up()[1],
              cam->get_up()[2]);
    // Axis drawing
    glDisable(GL_LIGHTING);
    glBegin(GL_LINES);
        glColor3f(1.0, 0.0, 0.0);
        glVertex3f(0.0, 0.0, 0.0);
        glVertex3f(10.0, 0.0, 0.0);
        // Ticks
        for(int i=0; i<20; i++){
            glVertex3f((float)i/2.0, -0.1, 0.0);
            glVertex3f((float)i/2.0, 0.1,0.0);
        }
    glEnd();
    glBegin(GL_LINES);
        glColor3f(0.0, 1.0, 0.0);
        glVertex3f(0.0, 0.0, 0.0);
        glVertex3f(0.0, 10.0, 0.0);
        // Ticks
        for(int i=0; i<20; i++){
            glVertex3f(-0.1, (float)i/2.0, 0.0);
            glVertex3f(0.1, (float)i/2.0,0.0);
        }
    glEnd();
    glBegin(GL_LINES);
        glColor3f(0.0, 0.0, 1.0);
        glVertex3f(0.0, 0.0, 0.0);
        glVertex3f(0.0, 0.0, 10.0);
        // Ticks
        for(int i=0; i<20; i++){
            glVertex3f(0, -0.1, (float)i/2.0);
            glVertex3f(0, 0.1, (float)i/2.0);
        }
    glEnd();


}

void RenderAlgorithm::draw_unit_cube(){
    // Cube drawing
    glDisable(GL_LIGHTING);
    glBegin(GL_LINE_STRIP);
        glColor3f(1.0, 1.0, 0.0);
        glVertex3f(1.0, 1.0, 1.0);
        glVertex3f(-1.0, 1.0, 1.0);
        glVertex3f(-1.0, 1.0, -1.0);
        glVertex3f(1.0, 1.0, -1.0);
        glVertex3f(1.0, 1.0, 1.0);
        glVertex3f(1.0, -1.0, 1.0);
        glVertex3f(1.0, -1.0, -1.0);
        glVertex3f(1.0, 1.0, -1.0);
    glEnd();
    glBegin(GL_LINE_STRIP);
        glVertex3f(1.0, -1.0, 1.0);
```

```
        glVertex3f(-1.0, -1.0, 1.0);
        glVertex3f(-1.0, -1.0, -1.0);
        glVertex3f(1.0, -1.0, -1.0);
    glEnd();
    glBegin(GL_LINES);
        glVertex3f(-1.0, -1.0, 1.0);
        glVertex3f(-1.0, 1.0, 1.0);
        glVertex3f(-1.0, -1.0, -1.0);
        glVertex3f(-1.0, 1.0, -1.0);
    glEnd();
}

void RenderAlgorithm::draw_bbox(bbox bb){
    // BBox drawing
    glDisable(GL_LIGHTING);
    glColor3f(0.0, 1.0, 1.0);
    glBegin(GL_LINE_LOOP);
        glVertex3fv(bb.get_bbox_vert(0).get());
        glVertex3fv(bb.get_bbox_vert(1).get());
        glVertex3fv(bb.get_bbox_vert(3).get());
        glVertex3fv(bb.get_bbox_vert(2).get());
    glEnd();
    glBegin(GL_LINE_LOOP);
        glVertex3fv(bb.get_bbox_vert(4).get());
        glVertex3fv(bb.get_bbox_vert(5).get());
        glVertex3fv(bb.get_bbox_vert(7).get());
        glVertex3fv(bb.get_bbox_vert(6).get());
    glEnd();
    glBegin(GL_LINE_LOOP);
        glVertex3fv(bb.get_bbox_vert(2).get());
        glVertex3fv(bb.get_bbox_vert(3).get());
        glVertex3fv(bb.get_bbox_vert(4).get());
        glVertex3fv(bb.get_bbox_vert(5).get());
    glEnd();
    glBegin(GL_LINE_LOOP);
        glVertex3fv(bb.get_bbox_vert(0).get());
        glVertex3fv(bb.get_bbox_vert(1).get());
        glVertex3fv(bb.get_bbox_vert(6).get());
        glVertex3fv(bb.get_bbox_vert(7).get());
    glEnd();
}

void RenderAlgorithm::draw_bbox(bbox2 bb){
    // BBox drawing
    glDisable(GL_LIGHTING);
    glColor3f(0.0, 1.0, 1.0);
    glBegin(GL_LINE_LOOP);
        for(int i=0; i<4; i++){
            glVertex3fv(bb.get_bbox_vert(i).get());
        }
    glEnd();
    glBegin(GL_LINE_LOOP);
        for(int i=4; i<8; i++){
            glVertex3fv(bb.get_bbox_vert(i).get());
        }
    glEnd();
    glBegin(GL_LINE_LOOP);
        glVertex3fv(bb.get_bbox_vert(0).get());
        glVertex3fv(bb.get_bbox_vert(3).get());
        glVertex3fv(bb.get_bbox_vert(5).get());
        glVertex3fv(bb.get_bbox_vert(6).get());
    glEnd();
```

```
    glBegin(GL_LINE_LOOP);
        glVertex3fv(bb.get_bbox_vert(1).get());
        glVertex3fv(bb.get_bbox_vert(2).get());
        glVertex3fv(bb.get_bbox_vert(4).get());
        glVertex3fv(bb.get_bbox_vert(7).get());
    glEnd();
}

void RenderAlgorithm::show_marker()
{
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(-1,1,-1,1);

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();

    glDisable(GL_LIGHTING);

    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_LINES);
        glVertex2f(-0.1, 0.0);
        glVertex2f(0.1, 0.0);
        glVertex2f(0.0, 0.1);
        glVertex2f(0.0, -0.1);
    glEnd();

    glMatrixMode(GL_PROJECTION);
    glPopMatrix();

    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();
}
void RenderAlgorithm::clear_all()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
        GL_STENCIL_BUFFER_BIT);
    glShadeModel(GL_SMOOTH);
}
void RenderAlgorithm::process_key(unsigned char key){
    Vec3f p0;
    Vec3f p7;
    int polygons = 0;
    switch(key) {
        case 'd':
            lights->get_active_light()->move_right();
            //update_frustum(lights->get_light(0));
            break;
        case 'a':
            lights->get_active_light()->move_left();
            //update_frustum(lights->get_light(0));
            break;
        case 'w':
            lights->get_active_light()->move_forward();
            //update_frustum(lights->get_light(0));
            break;
        case 's':
            lights->get_active_light()->move_backward();
            //update_frustum(lights->get_light(0));
            break;
```

```
        case 'z':
            lights->get_active_light()->move_up();
            //update_frustum(lights->get_light(0));
            break;
        case 'x':
            lights->get_active_light()->move_down();
            //update_frustum(lights->get_light(0));
            break;
        case 'r':
            cam->zoom_in();
            break;
        case 'f':
            cam->zoom_out();
            break;
        case 'm':
              lights->get_active_light()->animate_toggle();
            break;
        case '5':
            lights->get_active_light()->decrease_far();
            break;
        case '6':
            lights->get_active_light()->increase_far();
            break;
        case '7':
            lights->get_active_light()->decrease_near();
            break;
        case '8':
            lights->get_active_light()->increase_near();
            break;
        case '9':
            lights->get_active_light()->decrease_cutoff_angle();
            break;
        case '0':
            lights->get_active_light()->increase_cutoff_angle();
            break;
        case 'k':
            print_all();
            break;
        case '1':
            cam->set_near(cam->get_near()+1);
            break;
        case '2':
            cam->set_near(cam->get_near()-1);
            break;
        case 'P':

            for(int i=0; i<obj->no_meshes(); i++){
                polygons += obj->get_mesh(i)->no_faces();
            }
            cout << "Polygons Objects: " << polygons << endl;
            cout << "Polygons Floor:    " << floor->get_mesh(0)->no_faces()
                 << endl;
            break;
        default:
            break;
    }
}
void RenderAlgorithm::update_frustum(Light* light)
{
    Vec3f p0;
    Vec3f p7;
    obj->get_bbox(p0,p7);
```

```
    light->calculate_fustum(&p0, &p7);
}
```

References

[WILL78] L.Williams. "Casting curved shadows on curved surfaces." *In Computer Graphics (SIGGRAPH 78 Proceedings)*, pages 270274, Aug. 1978.

[KILL02] "Shadow Mapping with Todays OpenGL Hardware." SIGGRAPH 2002 Course http://developer.nvidia.com/attach/6770

[CROW77] Frank Crow "Shadows Algorithms for Computers Graphics." Computer Graphics, Vol. 11, No.3, Proceedings of SIGGRAPH 1977, July 1977

[BS03] Stefan Brabec and Hans-Peter Seidel "Shadow Volumes on Programmable Graphics Hardware" Comput. Graph. Forum. Vol 22 n. 3, 2003 p. 433-440

[EK03] Cass W. Everitt and Mark J. Kilgard "Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering" Computing Research Repository. Vol cs.GR/0301002, 2003

[CD04] Eric Chan and Frédo Durand "An Efficient Hybrid Shadow Rendering Algorithm" Proceedings of the Eurographics Symposium on Rendering, 2004 p. 185-195

[SD02] Marc Stamminger and George Drettakis "Perspective Shadow Maps" Proceedings of SIGGRAPH 2002 p. 557-562

[KOZ04] Simon Kozlov "Perspective Shadow Maps: Care and Feeding" GPU Gems, ch. 14, 2004

[ASM01] Randima Fernando, Sebastian Fernandez, Kavita Bala and Donald P. Greenberg "Adaptive Shadow Maps" SIGGRAPH 2001, Computer Graphics Proceedings, 2001 p. 387-390

[DDSM03] Weiskopf, D. and Ertl, T. "Shadow Mapping Based on Dual Depth Layers" Procceedings of Eurographics '03 Short Papers, 2003 p. 53-60

[CG04] Chong, Hamilton and Gortler, Steven J. "A Lixel for Every Pixel" Proceedings of Eurographics Symposium on Rendering, 2004

[AL04] Timo Aila and Samuli Laine "Alias-Free Shadow Maps" Proceedings of Eurographics Symposium on Rendering, 2004 p. 161-166

[WSP04] Michael Wimmer and Daniel Scherzer and Werner Purgathofer "Light Space Perspective Shadow Maps" Proceedings of Eurographics Symposium on Rendering, 2004 p. 143-151

[MT04] Martin, Tobias and Tan, Tiow-Seng "Anti-aliasing and Continuity with Trapezoidal Shadow Maps" Proceedings of the 2nd EG Symposium on Rendering, 2004

[AM04] Timo Aila and Tomas Akenine-Möller "A Hierarchical Shadow Volume Algorithm" Proceedings of Graphics Hardware 2004 p. 15-23

[LWGM04] Lloyd, Brandon and Wendt, J. and Govindaraju, Naga K. and Manocha, Dinesh "CC Shadow Volumes" Proceedings of the 2nd EG Symposium on Rendering, 2004

[SCH03] Pradeep Sen, Mike Cammarano and Pat Hanrahan "Shadow silhouette maps" ACM Trans. Graph. Vol 22, 2003 p. 521-526

[S92] Mel Slater "A comparison of three shadow volume algorithms" The Visual Computer: International Journal of Computer Graphics, Vol. 9, 1992 p. 25-38

[WPF90] Andrew Woo and Pierre Poulin and Alain Fournier "A Survey of Shadow Algorithms" IEEE Comput. Graph. Appl. Vol. 10 Numer 6. 1990 p. 13-32

[ADT02] OpenGL Architechtural Review Board `ARB_depth_texture` OpenGL Extension Specification, 2002 http://oss.sgi.com/projects/ogl-sample/registry/ARB/depth_texture.txt

[AS02] OpenGL Architechtural Review Board `ARB_shadow` OpenGL Extension Specification, 2002 http://oss.sgi.com/projects/ogl-sample/registry/ARB/shadow.txt

[ASTS02] OpenGL Architechtural Review Board `EXT_stencil_two_side` OpenGL Extension Specification, 2002 http://oss.sgi.com/projects/ogl-sample/registry/EXT/stencil_two_side.txt

[EDBT02] OpenGL Architechtural Review Board `EXT_depth_bounds_test` OpenGL Extension Specification, 2003 http://oss.sgi.com/projects/ogl-sample/registry/EXT/depth_bounds_test.txt