2004:165 CIV

EXAMENSARBETE

L

Performance and Visual Enhancement Techniques for Real-time 3D Graphics

Lars Karlström   Daniel Löfmark

MASTER OF SCIENCE PROGRAMME
Department of Computer Science and Electrical Engineering
Division of Media Technology

2004:165 CIV • ISSN: 1402 - 1617 • ISRN: LTU - EX - - 04/165 - - SE

Abstract

Various techniques can be used to improve real-time 3D graphics. Two general parts are present, visual appeal and performance. Different techniques are used to improve these two elements of computer graphics, some of which are discussed in-depth in this thesis. Shadow volumes with stencil buffers is one of the leading techniques used today when it comes to realistic shadows for real-time 3D applications. This approach to shadowing is one of the primary topics of this thesis. The second topic focused upon is octrees using portals, which is an approach to improve performance in real-time 3D rendering. Bump mapping is also discussed. Some other 3D concepts are also touched upon, mainly to highlight the advantages of the previously mentioned techniques and also to illustrate why they were chosen instead of other available options. The goal of this thesis is to improve upon a 3D graphics engine to create a more visually pleasing world. Both visual appeal and performance enhancing elements are used to create a dynamically lit environment.

Table of Contents

1 Reading this Thesis 4
2 Glossary of Terms 5
3 Introduction 8
4 Theory 10
4.1 Basic 3D Graphics Concepts 10
4.1.1 Camera 10
4.1.2 Clipping 10
4.1.3 Culling 10
4.1.4 Object Level Culling 11
4.1.5 View Screen Culling 11
4.1.6 Z-buffer 11
4.1.7 Stencil Buffer 12
4.1.8 Hardware Accelerated Graphics 12
4.1.9 Tools Used For This Thesis 12
4.2 3D Graphics Concepts for this Thesis 12
4.2.1 Shadows 12
4.2.2 Planar Projected Shadows 13
4.2.3 Shadow Mapping 13
4.2.4 Shadow Volumes with Stencil Buffer 16
4.2.5 Shadow Mapping versus Shadow Volumes 19
4.2.6 Counting Shadow Volume Entrances and Exits 19
4.2.7 Depth Pass versus Depth Fail 21
4.2.8 Clipping Problems 22
4.2.9 Combining Depth Pass and Depth Fail 22
4.2.10 Octree 23
4.2.11 Portals 24
4.2.12 Bump Mapping 25
5 Implementation 27
5.1 Shadow Volumes 27
5.1.1 Shadow Silhouettes 27
5.1.2 Shadow Volumes 29
5.1.3 Rendering to the Stencil Buffer 30
5.1.4 Depth Pass 31
5.1.5 Depth Fail 31
5.1.6 Analyzing the Depth Pass and Depth Fail Algorithms 31
5.2 Octrees with Portals 32
5.3 Bump Mapping 34
6 Discussion 37
7 Evaluation 38
8 Conclusion 39
9 Further Work 40
10 References 41

1 Reading this Thesis

It is recommended to read through the glossary of terms and basic 3D concepts before starting on the rest of the thesis for people unfamiliar with 3D graphics. Some terms are used in a non-standard way for the English language and reading through these parts of the report can help alleviate confusion. A certain degree of programming, mathematical and general computer skill is assumed from the readers of this thesis and lacking them may make certain parts very hard to understand.

2 Glossary of Terms

These are the general uses of the following words in the text. There are parts where they have other meanings, however at such places it is specifically noted.

Array - a programming term which describes a list of elements where each element can be accessed through an index.

Camera - the eye through which a 3D world is viewed. It does in most cases consist of a couple of matrices that describes what parts of the 3D world are visible.

Clipping – a technique that makes sure that only what is visible on the screen is actually drawn. It accomplishes its task by cutting polygons which are only partly visible.

Cost - used in the text to refer to the computational cost of an operation, rather than a monetary cost.

Culling – a technique for improving the rendering speed. Culling throws away data that is not visible through the camera. This technique can operate on objects and polygons, often both.

Depth Buffer - synonym for Z-buffer, a buffer used to ensure objects being drawn in the correct order. It is accurate at per-fragment level.

Depth Fail - a technique used to count shadow volume enters and exits. It is slower, but more robust than depth pass.

Depth Pass - a technique used to count shadow volume enters and exits. It is faster, but less robust than depth fail.

Direct3D – a graphics API developed by Microsoft. It is a part of DirectX.

Expensive - used in the text to refer to how computationally expensive an operation is, not the monetary expense.

Face - a segment of the surface of an object, which is always a triangle for the purposes of this text. Used as a synonym for polygon in this text.

Fragment – basically a pixel with additional information such as a depth value. For simplicity consider a pixel a fragment until it is drawn onto the screen at which time it becomes an actual pixel.

Mesh - a collection of vertices and faces which describes an object or part of an object.

Object - 1. a 3D object containing all information necessary to be rendered to the screen. This use of the word is for graphical purposes.
2. a programming term describing a collection of variables and functions.

Octree - a structure dividing the world into multiple parts.

OpenGL – a graphics API originally developed by Silicon Graphics. It is a state machine and the OpenGL functions allow the states to be changed. It is the graphics API used in this thesis.

Polygon - a segment of the surface of an object. Used as a synonym for face in this text.

Portal - a link between two divisions of a 3D world. Used as a link between octrees for the purposes of this text.

Ray-Tracing – a technique where images is produced by tracing rays of light from the camera to the objects and then to the lights that shines on that objects at that point. This approach is currently non real-time for producing images and animations. It can produce photo realistic images. G. S. Owen [1] has described this technique in more depth.

Real-time - a term used to describe something which can not be done in advance as opposed to pre-made. A real-time system performs its tasks as they occur, which means that attributes can be changed during run-time and the system will respond appropriately. When spoken of in this thesis a real-time system is a system which is able to perform its task at an acceptable frame rate. An acceptable frame rate for this purpose means that the animation appears fluent and not jerky.

Self shadowing - self shadowing means that if for example a model of a man holds his arm towards the light to shield his eyes, the arm casts a shadow on his face.

Shadow Volume - a volume dividing the world into shadowed and lit parts.

Shadow Silhouette - the border between shadowed and lit parts of an 3D object.

Stencil Buffer - a buffer which is used to decide how to render the final image. It is essentially a bit mask.

Texture – an image used to represent a surface.

Vertex - a point in 3D space which is defined by three or four coordinates. These are the X, Y, Z, W coordinates, where X, Y, Z are coordinates relative to the coordinate systems origin. The W coordinate is assumed to be 1 for 3 element points. It can be anywhere between 0 and 1 for four coordinate points.

Z-buffer – a buffer that can store and compare depth values. It is in most cases used to make sure that the pixels in the rendered image are the pixels that are closest to the camera.

3 Introduction

In today's rapidly expanding computer graphics industry consumers have come to expect more and more from the various products. The game industry is perhaps the best example of this development. Only a few years ago games had a much simpler look than they do today and it is probably safe to assume that this progress will continue. Presently, leading real-time 3D engines produce graphics which is admittedly not of picture quality but at least very visually pleasing. Modern ray-trace renderers produce graphics of picture quality which are hard and sometimes impossible to distinguish from real pictures, at least if used by a skilled graphics artist. Games have not quite reached this state yet since they need real-time graphics which ray-tracing is not. It is not improbable that ray-tracing will be done in real-time in the future at which time there would be a graphical revolution in games. At present however game designers do not have this luxury. Luckily games today don't need to look realistic, just good. Although the gaming industry is the primary focus for the techniques as they described in this thesis, they are by no means limited to this industry. Octrees for example can be used anywhere large amounts of data needs to be organized into subsets which use a tree structure. The only constraint is that the data needs to be representable in three dimensions. This would include a wide variety of fields. Shadows could be used in any kind of visualization, not just games. Such fields might be anything from medical applications to military simulations. Bump mapping has fields of use similar to those of shadows.

The goal of this thesis is to improve upon a 3D graphics engine to create a more visually pleasing world. The graphics engine to receive this overhaul is AgentFX, which is a 3D graphics engine written in Java by Agency9. As of the start of this project AgentFX is capable of all the basic functionality expected of a modern 3D graphics engine. This includes polygon rendering, texturing, collision detection and so on. What is truly lacking is dynamic lighting. In the real world dynamic lighting plays an important part of how we perceive our surroundings. Thus any 3D engine seeking to mimic real world conditions must include this element. In order to achieve our goal several techniques must be used. One of the problems which must be considered is the polygon count. The earlier polygons or entire objects can be removed from the rendering pipeline the better it is from a performance perspective. The reason an octree is used is that it allows for a lower visible polygon count and a way to divide the world into an easily traversed structure. This in turn allows for higher quality graphics at acceptable frame-rates. In current 3D graphics lighting is done through shading, sadly this is insufficient to create a realistic and dynamic world. The reason for this is that an entire aspect of real world lighting is missing, namely shadows. Shadows are a large part of lighting in the real world. Though shading approximates it nicely in some situations a scene simply does not look real unless the objects themselves cast shadows to interact with the world around them. Since shadow volumes are the most realistic approach to shadows currently available it is the approach chosen for this thesis. Again this introduces more polygons in the world to keep track of where the shadows are which makes the octree even more necessary. Finally bump mapping is used to introduce a more natural look to surfaces within the world.

Bump mapping produces dents, bumps and other imperfections to a surface which makes the surface appear more realistic. Together with shadows, shading and octrees a much more realistic view of the world is produced than with mere texturing. The focus is not on creating highly optimized implementations. Rather the focus is on creating implementations which do their job properly. Finally they should be relatively easy to extend and optimize.

4 Theory

Modern games push the boundaries for how realistic a real-time computer representation can become. In order to accomplish this, the real world is broken down into a series of elements and represented as mathematical formulas and models. As new advances in computer hardware are made previous cutting edge graphics technology becomes standard and new areas are explored to enhance the virtual experience. In some cases new techniques are applied to make existing effects seem more realistic, this is the case with shadows which will be discussed more in depth later on. There is however another aspect which needs to be addressed in order to obtain maximum realism from existing hardware, performance enhancing techniques. There are various ways to increase performance most of which try to minimize the number of calculations necessary by deciding which data needs to be processed and which doesn't. One such technique is called an octree, this will be described in depth in its own section later on.

4.1 Basic 3D Graphics Concepts

The following basic concepts are important to understand in order to fully grasp the rest of the text. These concepts are not usually done in software for the purposes of real-time graphics anymore. This is because leading 3D APIs such as OpenGL and Direct3D do them automatically, and they are done faster in hardware.

4.1.1 Camera

The camera is the eye through which a 3D world is viewed. The camera has its own coordinate system since instead of rotation and translating the camera to fit the world the opposite is done. This may seem strange but it is in fact advantageous to do so. A camera contains a frustum which is a volume consisting of six planes, this defines what the camera can see. It also contains a matrix for transforming coordinates into camera space and sometimes a matrix for projecting from camera space onto the screen.

4.1.2 Clipping

Clipping is a concept which one must grasp in order to do 3D graphics and indeed most computer graphics. When an image is drawn to the screen all coordinates drawn must fall within the screens area. Obviously coordinates that are outside this area cannot be drawn. To make sure that unwanted coordinates are not drawn is the domain of clipping. The basic idea is to draw everything that is visible and throw away everything that isn't. In modern graphics this is usually done automatically by graphics hardware or a graphics API such as OpenGL or Direct3D.

4.1.3 Culling

Culling is another concept which must be understood in order to create efficient 3D

graphics. The basic idea with culling is the same as with clipping, which is to draw only what needs to be drawn. Culling operates on a different level than clipping however. Culling is usually done in two different levels, object level culling and view screen culling, both described below.

4.1.4 Object Level Culling

Object level culling is usually done in 3D space. What this means is that before anything is rendered each object is checked against the view frustum, if it is at least partially visible it is kept, otherwise it is discarded as far as the current rendering cycle is concerned.

4.1.5 View Screen Culling

The other type of culling is view screen culling which operates on polygon level, this can be done in either 3D or in 2D. If done in 3D it works in the same manner as object culling except that individual polygons are tested against the frustum instead of entire objects. Note that view screen culling is only done on objects which have passed the previous object level culling (assuming that object level culling is present). 2D culling takes place right before 2D clipping. Since the culling is done on projected polygons more calculations have to be done before the data can be discarded which is a drawback. 2D culling is however simpler to implement and faster than 3D culling which is an advantage. When culling is mentioned in this text it is safe to assume that it is 3D culling unless otherwise noted.

4.1.6 Z-buffer

In order to guarantee that objects are drawn in the correct order they need to be sorted in some fashion. There are various ways to do this which are more or less secure. The leading way to do this in real-time is a Z-buffer (also called a depth buffer). A Z-buffer is a buffer which has as many elements as the viewing surface (usually a screen) has pixels. If the fragment to be rendered is closer than the fragment already occupying the Z-buffer, at the appropriate position, it is drawn, otherwise it is discarded. The Z-buffer does not store color values, instead it stores depth values. Whenever a fragment is to be drawn the appropriate location in the Z-buffer is first checked.

Usually it is desirable to only allow fragments closer to the screen than the current fragment in the Z-buffer to be drawn. When the Z-buffer or depth buffer is referred to in this thesis it is assumed that it is used to guarantee that the closest fragment to the viewer is drawn. This is not necessarily always what the Z-buffer is used for as it can be used for anything concerning depth, however for the purposes of this thesis it is unless otherwise noted. The buffer guarantees correct order of each pixel and has no special cases in which it does not work as is the case of some other algorithms.

4.1.7 Stencil Buffer

The stencil buffer is basically a mask, which usually has as many elements as the Z-buffer. It can be used to determine which parts of the screen are to be rendered to for example. Stencil buffers have a low bit depth which means that they cannot contain very large values. A technique called stencil wrapping can be used to minimize this issue. C. Everitt and M. J. Kilgard [2] describe the general use of the technique.

4.1.8 Hardware Accelerated Graphics

In order to access the features on graphics cards it is usually desirable to do so through an API. There are two leading graphics APIs on the market today, OpenGL and Direct3D. OpenGL was originally developed by Silicon Graphics and Direct3D by Microsoft. It is desirable to use the features on the graphics card to achieve better performance from graphics intensive applications such as games. The reason for this is that the GPU, the processor on the graphics card, is faster than the CPU, the main processor, at making 3D calculations. The GPU and indeed the entire graphics card, is specialized at its task. The CPU on the other hand is a general processor which can perform any task, but excels at none. In a game one also wishes to use the CPU for everything that the GPU cannot handle. This in essence means that one wishes to unload as much graphics calculations as possible onto the GPU. OpenGL and Direct3D are APIs which allow the programmer to use the graphics acceleration of 3D cards in a standardized fashion.

4.1.9 Tools Used For This Thesis

All code for this thesis have been written in Java, OpenGL and Cg. Java has been used at the request of Agency9 since this is the language the rest of their 3D engine, AgentFX, is written in. As a side note we have found Java to perform above our expectations when used properly. OpenGL was chosen for the same reason and the OpenGL interface used is JOGL which stands for Java bindings for OpenGL. The shaders written for this project are all written in Cg which stands for C for graphics and is developed by Nvidia. The Eclipse IDE has been used to write all code except the Cg shaders.

4.2 3D Graphics Concepts for this Thesis

4.2.1 Shadows

When we perceive the real world there are many aspects which we take for granted. One such aspect is the presence of shadows. If a person were to look at an image of a scene with the shadows removed something would not seem right even though it might not be apparent exactly what. Without shadows a scene would appear static and lifeless. It would also be very hard to determine spatial relationships between objects if there were no shadows in a scene.

Image 1: Un-shadowed Scene. **Image 2:** Shadowed Scene.

Shown above are two images, identical except in one respect. *Image 1* contains no shadows and *image 2* does.

4.2.2 Planar Projected Shadows

Planar projected shadows are a very simple and cost effective way of rendering shadows. The drawbacks are that it has a very limited number of situations in which it can be used with a pleasing result. As the name suggests it projects the shadow casting object onto a plane. This creates a shadow where the object is projected. Because of this the object does not cast shadows on it self and can only cast shadows onto planar surfaces. This method can be extended to cast shadows on multiple planes, for example if an object casts shadows on both the floor and a wall, but at greater computational cost. The reason this method of casting shadows is insufficient and not worth further investigation in this thesis is that objects aren't self shadowing. As the goal of this thesis is to produce as realistic shadows as possible, the technique used needs to produce self shadowing objects.

4.2.3 Shadow Mapping

Shadow mapping is a technique for shadowing which a shadow map in the form of a texture is used to determine shadows in a scene. The main advantage with this approach is the speed at which it can be accomplished. It does however have some major drawbacks, especially if used as a general solution. Shadow mapping cannot handle directional lights. Omni lights are also a problem but can be handled with multiple projections. This solution is best suited for spot lights. Another drawback is that the shadow might appear coarse and pixelated. This can be partially overcome with higher resolution maps and some other techniques but is nevertheless a problem. Basically shadow mapping first renders the scene with a camera at the lights position, looking at the scene and then with a normal camera. The depth values of the first rendering are checked as the second rendering proceeds. Each fragment in the second rendering consults the depth buffer of the first rendering to determine if it is shadowed or lit. A more detailed description of the technique follows below.

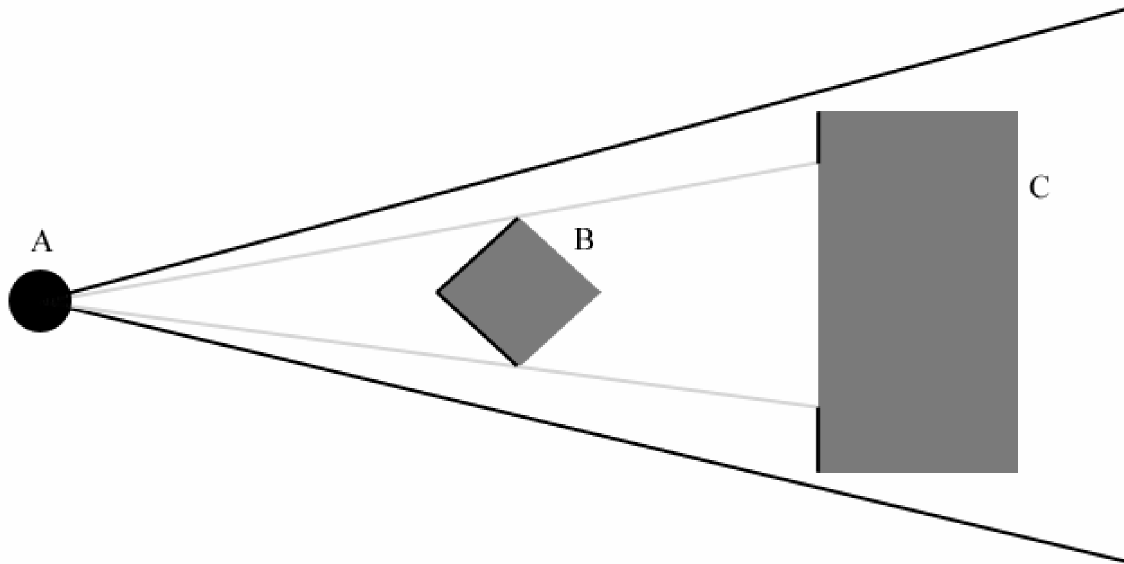

Figure 1: Building the shadow map

First render the scene to the depth buffer from the lights point of view as shown in *figure 1*. A is the light and B and C objects which are rendered to a camera placed at A. The black lines forming a triangle represent the frustum of this camera. The black lines on B and C are the parts which will actually be rendered to the depth map. This map contains the closest fragments to the light and is used for comparison in the second pass. They describe which parts of the scene are actually lit. In the second pass, render the scene from the eyes point of view.

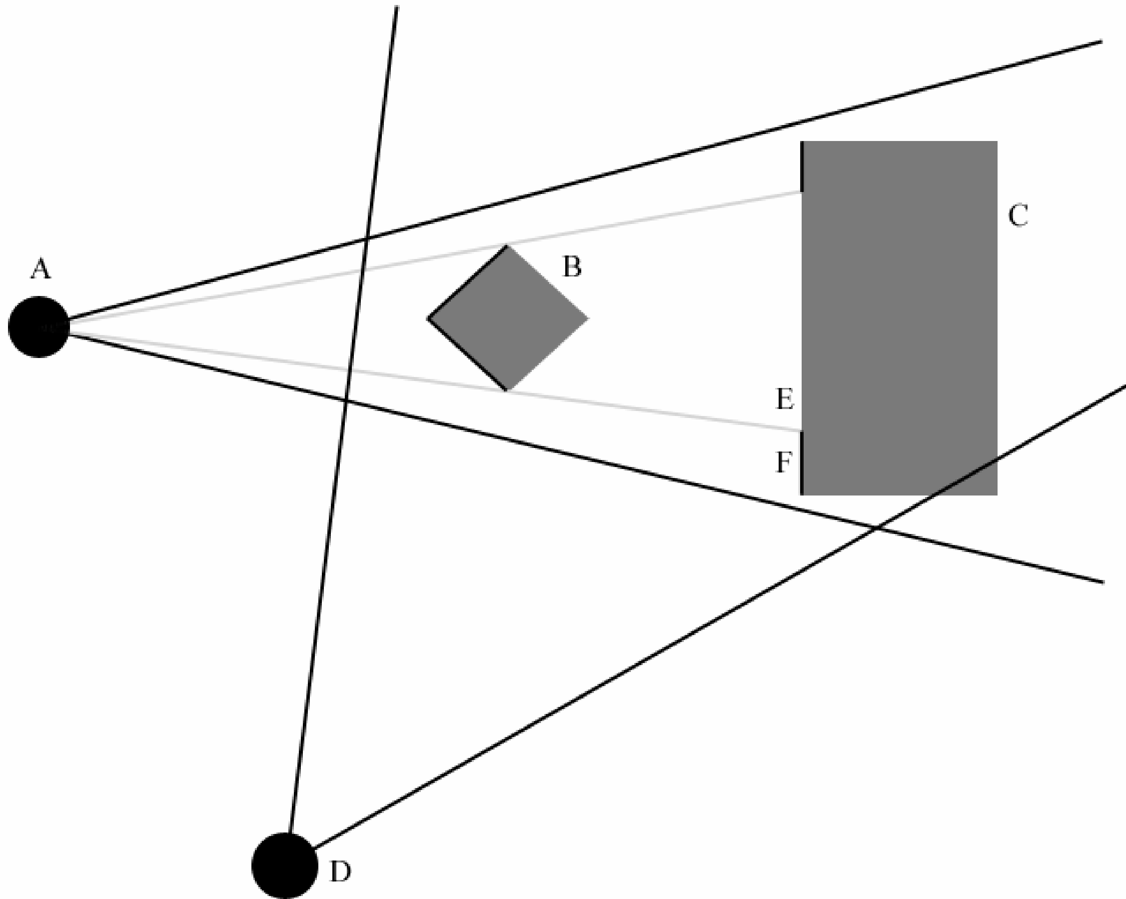

Figure 2: Rendering using a shadow map

Parts A, B and C in *figure 2* are the same as in *figure 1*. Here D is the eye and E and F are shadowed and lit parts of object C respectively. For each rasterized fragment determine the fragments XYZ position relative to the light. This light's position should be the same as used in the previous pass. Compare the depth value in the depth map at the appropriate XY position (The position where the fragment would be if it had been rendered to the depth buffer), called Z_1 from now on, to the fragments Z position relative to the light, called Z_2 from now on. If Z_2 is greater than Z_1 then something must be closer to the light than the current fragment, in other words the fragment is shadowed. If they are approximately equal, which is defined as being within a certain error tolerance, then the fragment is lit. An error value is needed because floating point calculations on a computer are subject to rounding errors. The following algorithm describes the process:

1. Place the camera at the light's position
2. Render the scene to the depth buffer
3. Copy the depth buffer to a texture
4. Place the camera at the eye's position
5. Render the scene checking Z_2 against Z_1 for each fragment

This is a general algorithm and as such could be implemented in many ways. Steps 1-3

need to be repeated for each light in the scene. In step 5 the fragment needs to be checked against each shadow map for multiple lights. Shadow casting, when done properly, produce quite decent results even for complex scenes. Objects are self shadowing when using this technique. M. J. Kilgard [3] explains the basics of shadow mapping further.

4.2.4 Shadow Volumes with Stencil Buffer

The most realistic approach and model for shadowing is shadow volumes. Real shadows have volume and so does real lights. Therefore this solution is the one which most closely matches real world's conditions. The main advantages of this method are that it can handle any light source and objects are self shadowing. Shadow volumes at acceptable frame rates have not really been possible until now. Advancements in CPU speed and added functionality on graphics cards such as a stencil buffer have made the approach more viable. Other refinements are depth clamping, double sided stencil testing, a programmable graphics pipeline, and some other techniques. These all serve to increase performance of shadow volumes. Stencil buffer shadow volumes are accurate at pixel level, or sub-pixel level if multisampling is available. This is because shadow determination is performed in screen space, with the help of a stencil buffer. Just about any PC bought today has a graphics card that supports a stencil buffer. The main disadvantage of the shadow volume approach is that it is slower and more calculation intensive than other approaches. Stencil buffer shadow volume algorithms are inherently multiple pass algorithms. They are so since one pass is required for rendering the scene with ambient lighting, and one pass to render it for each shadow casting light. This means that more than one rendering cycle is required to compute a single frame.

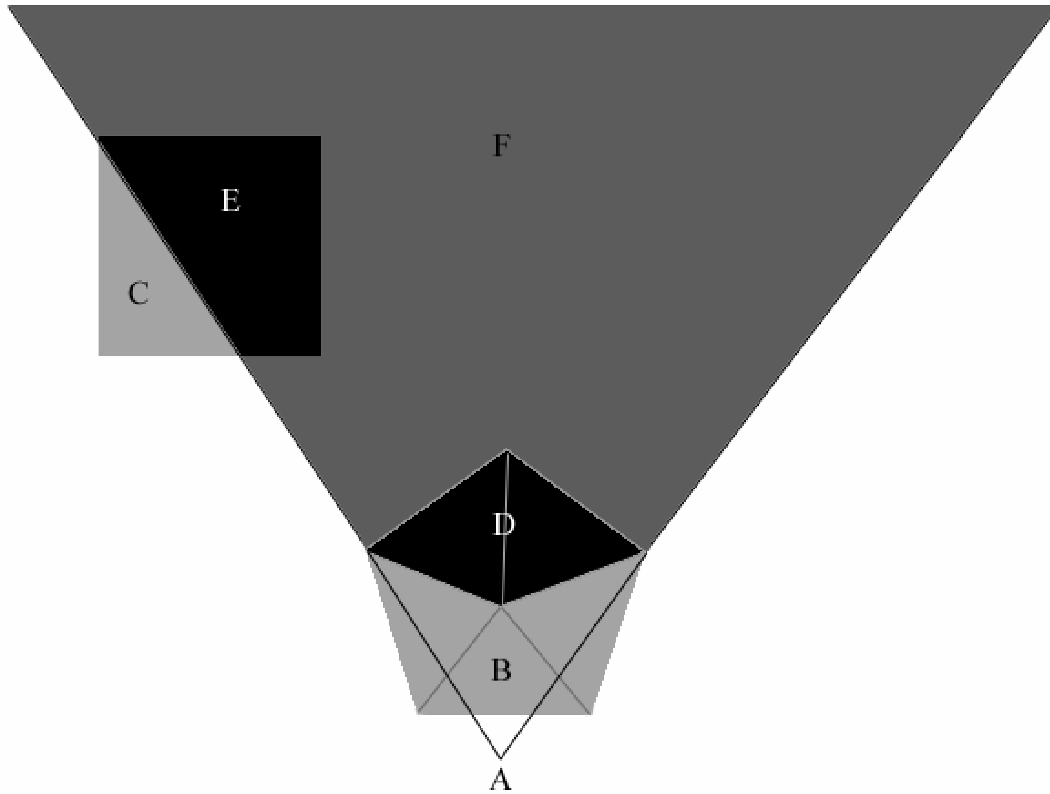

Figure 3: Shadow volume cast from an occluder.

Figure 3 shows how an occluder casts a shadow on the surrounding area. Lit parts of objects are shown in light gray and named B and C. Shadowed parts of objects are shown in black and named D and E. The dark gray part, named F, is the shadow volume itself, and the point at A is the light source. The border between B and D is the shadow silhouette and the polygons comprising D would be used to generate the caps for depth fail. Note that F itself would never be drawn onto the final scene, it would be present in the stencil buffer to determine which fragments are shadowed. F also exists in world space until the time of rendering where it is used to check for camera presence within the shadow volume. The square in *figure 3*, comprised of C and E, called Object 2 from now on, is the shadow receiver, it is shadowed but doesn't cast the shadow itself. Only occluders cast shadows, and each object casts its own shadow meaning that in this case B and D comprises the occluder, called Object 1 from now on, and Object 2 does not. Object 2 will however be the occluder when its shadow volume is calculated. This may seem confusing but just remember the following rules. Every object casts its own shadow. Every object is the occluder for its own shadow volume. Every object shadowed by the shadow volume of another object is called a shadow receiver. Finally every light which causes objects to cast shadows is called a shadow casting light.

A shadow silhouette is the boundary between lit and shadowed polygons. The silhouette is calculated by examining the edges of the faces composing the mesh. This is then extruded to form a shadow volume, which is a mathematical representation of how the object shadows the environment for a specific light. This is done by computing two new

vertices based on the vertices for the current edge and the vector between the light and these vertices. This results in one or two shadow faces depending on the type of polygon used as a primitive, triangles or quads respectively. Calculating silhouette edges can be expensive, especially with high resolution models, multiple shadow casting lights, a large number of polygons in the scene, or any combination of the three. The approach can also consume huge amounts of fill rate. Another disadvantage is that stencil buffer shadow volumes have hard edges because they model ideal light sources (the lights lack volume). They also require polygonal models which must be closed and contain no non-planar polygons unless these are specifically handled. The model can be described with the following algorithm:

1. Render scene with ambient and non-shadow casting lights
2. Silhouette calculation
3. Shadow volume extrusion
4. Render shadow volumes to stencil buffer
5. Render scene with the current shadow casting light enabled
6. Repeat step 2 - 5 for all shadow casting lights in the scene

This is a simplified model, however it serves well to grasp the general concepts involved, a more detailed and design specific model can be found in the Implementation chapter. J. Tsiombikas [4] describes the process in detail. The described method leaves two holes at the front and back sides of the volume however. Depending of the technique used, depth pass or depth fail, these volumes may need to be capped. Capping a volume means that the volume becomes completely closed.

A stencil buffer is essentially a mask, this makes it a good tool for use with shadow computations. Usually when one talks about rendering one refers to the process of producing an image on the screen. This assumption is not necessarily true however as one could render the image to a different medium, one which is not displayed for example. This is precisely what is done with the stencil buffer. The stencil buffer is never rendered onto the screen. It is merely a tool to decide which parts of the scene will be rendered with the shadow casting light later on. In the final step the scene is rendered with the current shadow casting light enabled and ambient and non-shadow casting lights disabled. It is in this stage that the stencil buffer is used. The stencil buffer decides which parts of the scene are rendered with this light. The result is then blended with the previous rendering cycle and this becomes the image which is displayed on the screen after all lights have been processed.

Obviously, in the real world all lights cast shadows. In a computer model however this may not always be desirable for various reasons. One of these is performance since casting shadows costs more than abstaining from it. Another issue is the impression of the scene and what concept the designer is trying to get across. An example which fits quite nicely is if the character being played is chased down a corridor by a "baddie" of some kind. In this case one might wish to place a light at the end of the corridor so that the shadow of the "baddie" engulfs the player and makes the scene seem more frightening. In this case the presence of additional shadow casting lights would reduce the experience

since the shadow of the chaser would become less apparent.

4.2.5 Shadow Mapping versus Shadow Volumes

	Shadow Mapping	Shadow Volumes
Pros:		
Self Shadowing Objects	Yes	Yes
Casts Shadows on curved surfaces	Yes	Yes
Fast hardware only implementation possible	Yes	No
Can handle all types of light sources	No	Yes
Model tied closely to real world conditions	No	Yes
Cons:		
Multi-pass algorithm	Yes	Yes
Frustum dependent	Yes	No

Table 1: Shadow mapping versus shadow volumes

Although shadow mapping is faster than shadow volumes partly because the algorithm can be done completely in hardware we have elected shadow volumes for this thesis. This was done for several reasons. The shadow mapping technique is limited to certain light types because it relies on creating a view frustum, and a depth map based on that frustum. This can as far as we know not be worked around and we wanted a technique which is general enough to handle all situations. Secondly, although shadow mapping has greater hardware support on current graphics cards we feel that this won't be the case in the future as newly added functionality on graphics hardware are great for shadow volumes. The main problem with shadow volumes on hardware is loosing high level object knowledge. In other words the programmable graphics hardware is only "aware" of a single vertex or fragment at a time. This makes hardware only implementations of shadow volumes on current hardware computationally expensive. In order to make a fast hardware only implementation of shadow volumes one basic requirement must be satisfied. The graphics card needs to be able to calculate the silhouette (preferably in a single pass for all lights). In order to accomplish this, an awareness of object geometry must exist within the hardware in some fashion. Future hardware may be able to create fast shadow volumes. The shadow volume model is the closest to the real world conditions of the current leading real-time techniques. Therefore we feel that this is the approach which will eventually be the standard in the gaming industry. Having said that, whenever shadows are discussed later in this thesis it can be assumed that the subject is shadow volumes unless otherwise stated.

4.2.6 Counting Shadow Volume Entrances and Exits

To determine if a given pixel is shadowed or lit entrances and exits through shadow volumes need to be counted. When shadow volumes are rendered to the stencil buffer they are done so by the use of counters. For depth pass the following happens when the depth test passes. A depth test is a test which checks the Z-value of a fragment against the Z-buffer and then makes a decision based on a function. The function can be “less” for

example which means that every fragment which passes the depth test has a Z-value less than the Z-value at the appropriate Z-buffer position. Consequently, for this example all fragments which have a Z-value greater than the value at the appropriate Z-buffer position fails the test. When front facing polygons are rendered the values at the positions in the stencil buffer where the area is projected are increased. When back facing polygons are rendered the values at the positions are decreased. The end result after all volumes have been rendered is that the values in the stencil buffer are zero if the position is lit and greater than zero if it is shadowed. Depth fail increments and decrements the stencil buffer when the depth test fails. For depth fail values are increased for back facing polygons and decreased for front facing polygons. This is because depth fail counts from infinity and to the view plane.

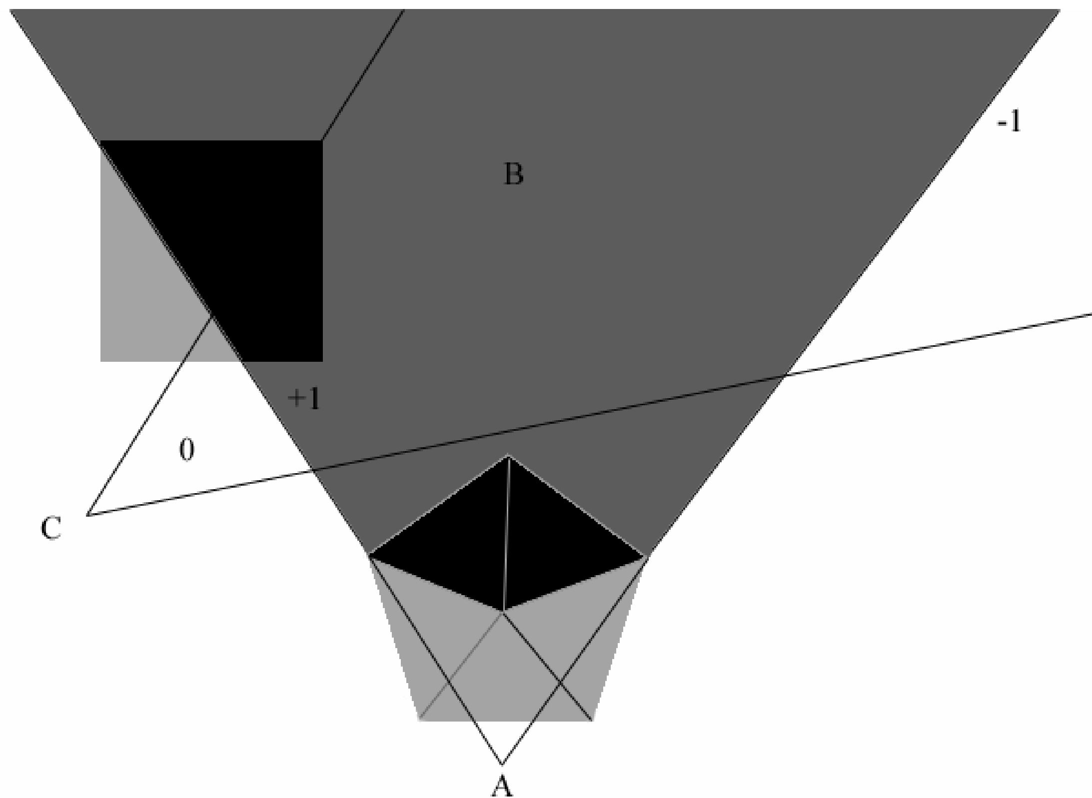

Figure 4: Counting shadow volume entrances and exits for depth pass

Figure 4 shows how the values of the stencil buffer are calculated for the depth pass algorithm. Imagine *figure 4* being a top down view of a 3D world. Starting at C (the camera) all front facing shadow faces are rendered. In this case, this is the face located at +1. The +1 means that the values contained in all positions in the stencil buffer where the face is rendered to are increased by one. Next all back facing polygons are rendered. In this case this is the polygon at -1. The -1 means that the values contained in all positions in the stencil buffer where the face is rendered to are decreased by one. Keep in mind that the polygons of the shadow volume are only rendered if they pass the depth test. Also keep in mind that the polygons comprising the shadow volume are never rendered to the

depth buffer or the frame buffer. This means that they do not affect how other objects are rendered aside from shadowing them.

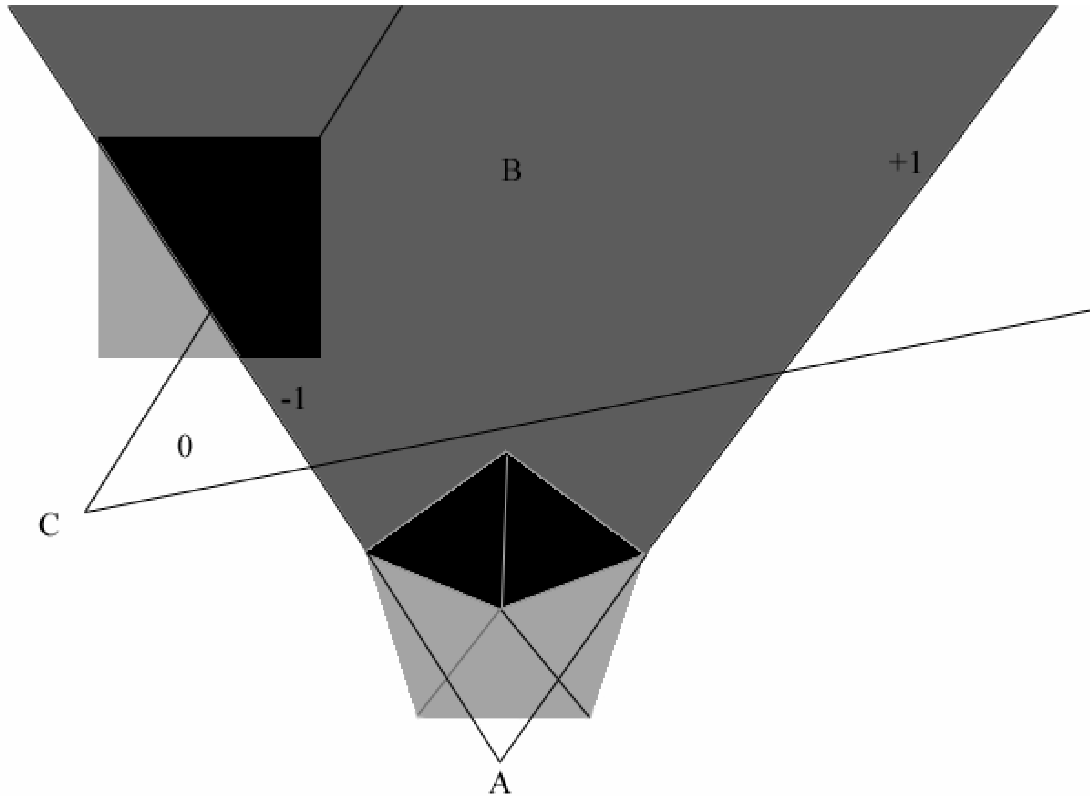

Figure 5: Counting shadow volume entrances and exits for depth fail

Figure 5 shows how the values of the stencil buffer are calculated for the depth fail algorithm. Imagine *figure 5* being a top down view of a 3D world. Starting behind +1 (actually at infinity) and going towards C (the camera) all back facing shadow faces are rendered. In this case this is the face locates at +1. The +1 means that the values contained in all positions in the stencil buffer where the face is rendered to are increased by one. Next all front facing polygons are rendered. In this case this is the polygon at -1. The -1 means that the values contained in all positions in the stencil buffer where the face is rendered to are decreased by one. Keep in mind that the polygons of the shadow volume are only rendered if they fail the depth test. Also keep in mind that the polygons comprising the shadow volume are never rendered to the depth buffer or the frame buffer. This means that they do not affect how other objects are rendered aside from shadowing them. Depth fail is less intuitive than depth pass. Try to remember that when using depth fail the polygons are rendered starting at infinity. With this in mind the reason for rendering them when the depth test fails will hopefully become a bit clearer.

4.2.7 Depth Pass versus Depth Fail

Depth pass and depth fail although similar have different advantages and disadvantages.

They can however be combined and used together to take advantage of the strengths of each.

	Depth Pass	Depth Fail
Pros:		
Fast	Yes	No
Completly robust	No	Yes
Clipping problems solvable	No	Yes
Cons:		
Shadow volumes need to be capped	No	Yes

Table 2: Depth pass versus depth fail

4.2.8 Clipping Problems

Near and far plane clipping can cause serious problems when rendering shadow volumes for depth pass and depth fail respectively. This is due to the fact that the shadow count can fail, this in turn can cause lit areas to register as shadowed and vice versa. Shadow volumes divide the world into shadowed regions and everything else. Near and far plane clipping can slice open these well defined volumes causing errors. These errors manifest as points which are supposed to be shadowed become lit and vice versa. It is possible, but not practical, to construct a completely robust solution based solely on depth pass. The near plane clipping problem can manifest itself in several ways when attempting to cap volumes at the near plane, the approach is at best a fragile solution. The far plane clipping problem can be solved quite easily. This is done by representing the shadow volumes as infinite volumes. To do this the projection matrix needs to be replaced with a special infinite projection matrix. To calculate the infinite projection matrix, P_{inf}, the standard projection matrix P is used. Simply calculate Lim P as Far Plane approaches Infinity. The result is a matrix which doesn't clip at the far plane. C. Everitt and M. J. Kilgard [2] address infinite shadow volumes in detail. This however is not without disadvantages. For instance the precision at infinity is terrible. This is not as bad as it sounds though since it is only important to guarantee that the volumes are not clipped in this case. This approach has another effect as well though. All points used must be defined as (X, Y, Z, W) instead of the usual (X, Y, Z). In fact this is already the case when making 3D calculations in non-infinite space but the W element is assumed to be 1. To make calculations with infinite points the W elements needs to be set to 0. When shadow volumes are talked about later in this thesis they are assumed to be infinite unless otherwise noted.

4.2.9 Combining Depth Pass and Depth Fail

Once the far plane clipping problems has been solved depth fail is completely robust, this means that it can be used in any situation and still give a correct result. Although this is the case it is not advisable to do so, the reason being that depth pass is faster. The reason is that depth fail volumes need to be capped which is somewhat expensive. The solution is to only use depth fail in the instances when depth pass fails and this only happens when the volume is clipped by the near plane. This in turn can only happen when the near plane

is within the shadow volume. By using bounding volumes to detect near plane presence within a shadow volume the instances where depth fail is necessary can be detected. When the two are combined correctly the result is a solution that is as robust as a pure depth fail solution, with some of the added speed of depth pass.

4.2.10 Octree

Octrees are a way to represent world data in order to minimize calculations needed. Octrees have one main advantage over BSP trees, which is that indoor and outdoor scenes can both be handled with equal ease. M. Feldman [5] explains the basics of BSP trees. As games become more and more complex consumers are no longer satisfied with worlds that have a wholly indoors feel. Therefore the BSP is no longer sufficient in all scenarios and an alternative must be devised. This alternative is the octree. The octree can as mentioned earlier handle both indoors and outdoors scenes. To further increase performance and minimize polygon counts portals are used. When used together portals and octrees allow for seamless transitions between indoor and outdoor environments allowing games to take on a whole new scope of realism. They are also used to minimize lighting and shadow volume calculations meaning a more dynamic and visually pleasing world can be produced at acceptable frame rates. Starting with the entire world as a node the world is then split into eight parts, assuming the end conditions for splitting have not been met, each of which forms a new node. Only nodes containing world data are kept. The sizes of the bounding boxes which represent the nodes in the world are all of equal size, for the same level in the octree. The process repeats for a certain number of steps determined either by levels in the tree or desired max number of polygons in each end node or leaf. After the calculations are done a tree has been formed which is well suited for determining what needs to be drawn and what doesn't. Simple visibility detection with the bounding boxes representing the nodes in the game world accomplishes this. Due to the nature of a tree structure if a node isn't visible then none of its children are either. This means that using this structure one can determine visibility for a relatively low cost. The tree can also be used for other calculations than visibility such as collision detection and similar problems. The main advantage with an octree compared to other world splitting solutions is that it can be used with both indoors and outdoors environments. The octree is not as effective for indoors environments as a BSP tree, however some steps can be taken to improve performance for such conditions as is described in the portals section. Here is a simple, general, and recursive algorithm which can be used to generate an octree:

1. Read world data into root node
2. Create bounding box for node
3. Split node into eight child nodes
4. If child node contains data keep node, otherwise throw it away
5. Repeat steps 2 - 4 for each child node until one of the end conditions are met

This is only the algorithm to create an octree, another algorithm is used for visibility determination and collision detection. Once an octree has been created it is usually not changed because doing so would mean recalculating the entire octree. This means that

only static objects which make up the world are included in the tree. Objects which are moveable are not included in the octree. To calculate if a node is visible or not the following algorithm can be used, starting at the root node perform the following steps:

1. Check if node is a leaf and within viewing frustum, if so draw it, otherwise step 2
2. If it is visible repeat step 1 for each child node, otherwise skip node

After the octree testing has been done standard 3D culling and clipping techniques are applied to the data which passed the test. The data which failed the test is merely discarded as far as the current rendering cycle is concerned.

4.2.11 Portals

Portals are basically links. They link together different parts of the 3D world. For example consider a room with a single window connecting it to the outside world. The window could be represented as a portal or link. It follows that in order for anything outside the room to be visible the portal (window) must be visible. Therefore instead of testing if each individual object outside the room is visible only the portal must be tested. If the portal is visible, a new frustum is created which is used to determine which objects are visible through the portal. The specifics of this are described in detail in the Implementation chapter. Generally it can be said however that the new frustum to be created should be made as small as possible, preferably the size of the visible part of the portal. The finished rendering is then applied to the area which the portal occupies. The general process can be described in the following steps:

1. Render normally
2. If portal is visible create new frustum
3. Repeat steps 1-2 for each portal visible

This can be done with any number of portals, even with portals being visible thorough other portals.

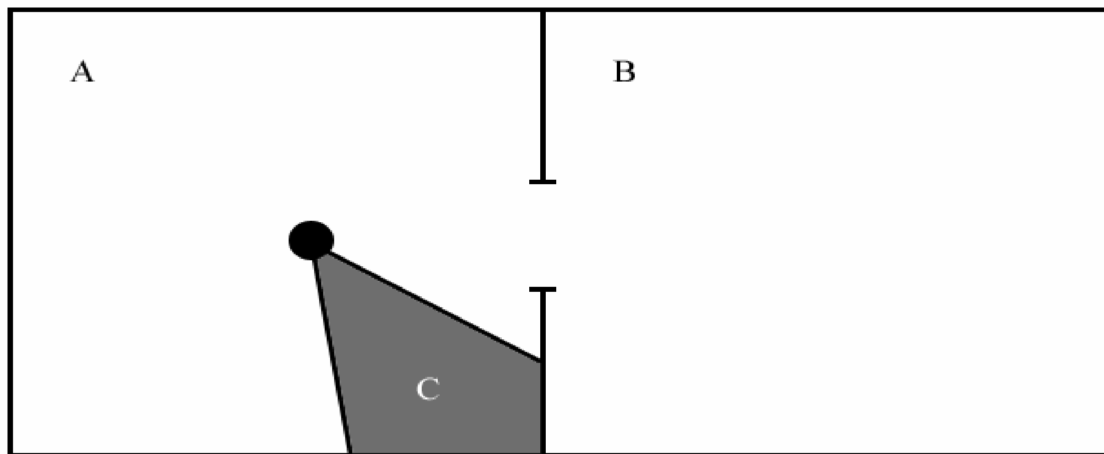

Figure 6: Portal outside view frustum

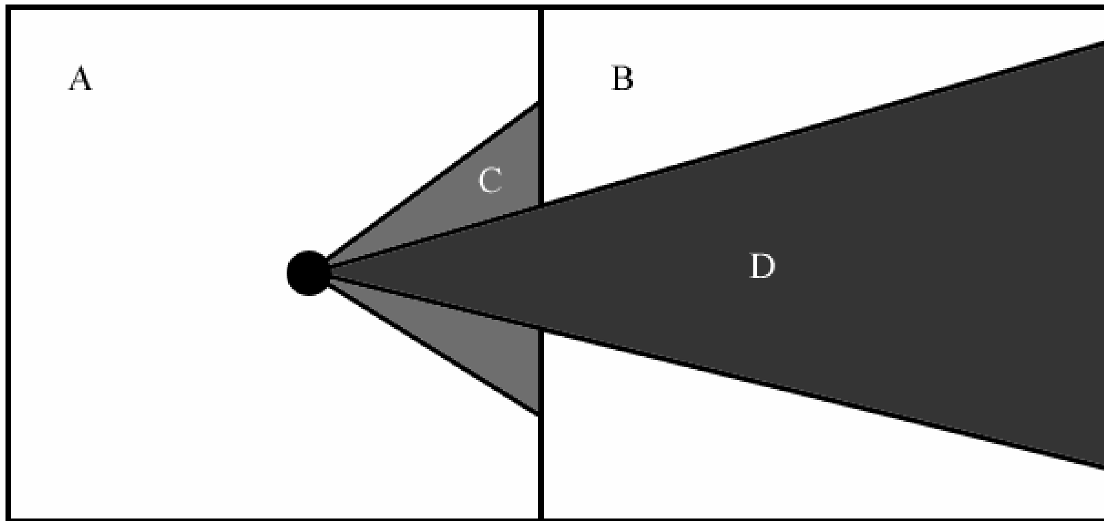

Figure 7: Portal inside view frustum

Figures 6 and *7* illustrate the creation of a new view frustum through a portal. A and B are two separate octrees linked together with a portal. The black circle is the camera and frustum C is the view frustum of this camera. In *figure 4* the portal is not visible inside the view frustum C. This means that no new frustum needs to be created as nothing in octree B could possibly be visible. In *figure 7* the portal is completely visible and a new frustum D is created. D is used to check for visible leaves within octree B. The frustum D is not used to render the image. On the contrary, the same camera is used to render the visible leaves within octree as is used to render octree A. This in essence means that the same matrices are used to transform vertices to screen space for both C and D.

4.2.12 Bump Mapping

Bump mapping is a technique used to increase realism by making surfaces appear rugged or bumped. It can be used to lower polygon counts by using the bump map to describe dents in the surface instead of additional polygons. Bump mapping is accomplished by modifying the normal at the current pixel, how the normal should be modified is described in a bump map. A bump map is in fact a normal map, in the same manner as texture maps contain color values a bump map contains surface normals. These influence how a surface is lit, by varying light intensity, which is why the surface appears to be "bumpy". The end result is a surface which appears more genuine. On modern graphics cards, GeForce 3 and up, bump mapping can be done by using register combiners. On GeForce FX bump mapping can also be implemented by writing vertex and fragment shaders.

In order to calculate normals for bump mapping, all vectors used in the calculations need to be in the same coordinate space. This is a problem because the surface normals of an object are in object space and the normals in the bump map are in texture space. This can be solved by transforming the normals so they all lie in the same coordinate system or

space. The surface normal and the light vector can be transformed into texture space by multiplying them with a matrix. The rows of the matrix consist of the face tangent, the face binormal and the face normal. This technique is further explained by P. Taylor [6] After this transformation it is possible to calculate a bumped normal. Add the interpolated and transformed normal to the appropriate normal from the bump map. To calculate light intensity at the current fragment, compute the dot product between the transformed light vector and the normal from the addition.

5 Implementation

The following describes solutions to shadowing, octrees with portals, and bump mapping. All of which are implementation specific.

5.1 Shadow Volumes

Before any code was written different ways to implement shadows was examined. After considering the pros and cons of the different methods it was decided that shadow volumes were the way to go. The main reason for this decision is that shadow volumes are more general than other solutions. Another factor is that graphics cards continue to improve and therefore the speed limitations of the solution are likely to be less severe in the future. Shadow volumes also provide the best image quality of the solutions examined.

5.1.1 Shadow Silhouettes

Shadow silhouettes are the basis for creating a shadow volume. A shadow silhouette is the border between lit and shadowed polygons. Finding it is not as difficult as it may seem, finding it fast however is a different story. This implementation uses an algorithm which finds the correct shadow silhouette however there are other algorithms which can be used to find an approximation. Which to use depends on if it is speed or precision that is sought after. The implementation used is a direct adaptation of the algorithm described by J. Tsiombikas [4]. The algorithm works as follows:

1. Compute the inverse transformation matrix for the object
2. Transform the light into object space
3. Compute direction between polygon and light (light direction)
4. Create an empty edge list
5. Test if polygon is lit
6. If polygon is lit proceed, otherwise repeat step 3 for next polygon
7. Examine edge and if edge is already in list of edges, remove edge and discard current edge, otherwise add edge to list of silhouette edges
8. Repeat 7 for every edge in the polygon
9. Repeat 3 - 8 for each face in the object

When this algorithm is complete the silhouette edge list has been found. Here the algorithm is represented as pseudo code:

```
// transform the light to the coordinate system of the object
LightPosition = LightPosition * Inverse(ObjectWorldMatrix);
for (every polygon) {
	IncidentLightDir = AveragePolyPosition - LightPosition;
	//if the polygon faces away from the light source....
```

```
        if (DotProduct(IncidentLightDir, PolygonNormal) >= 0.0) {
                for (every edge of the polygon) {
                        if (the edge is already in the contour edge list) {
                        // then it can't be a contour edge since it is
                        // referenced by two triangles that are facing
                        // away from the light
                        remove the existing edge from the contour list;
                        }
                        else {
                                add the edge to the contour list;
                        }
                }
        }
}
```

The code above has been taken from [4] and altered slightly.

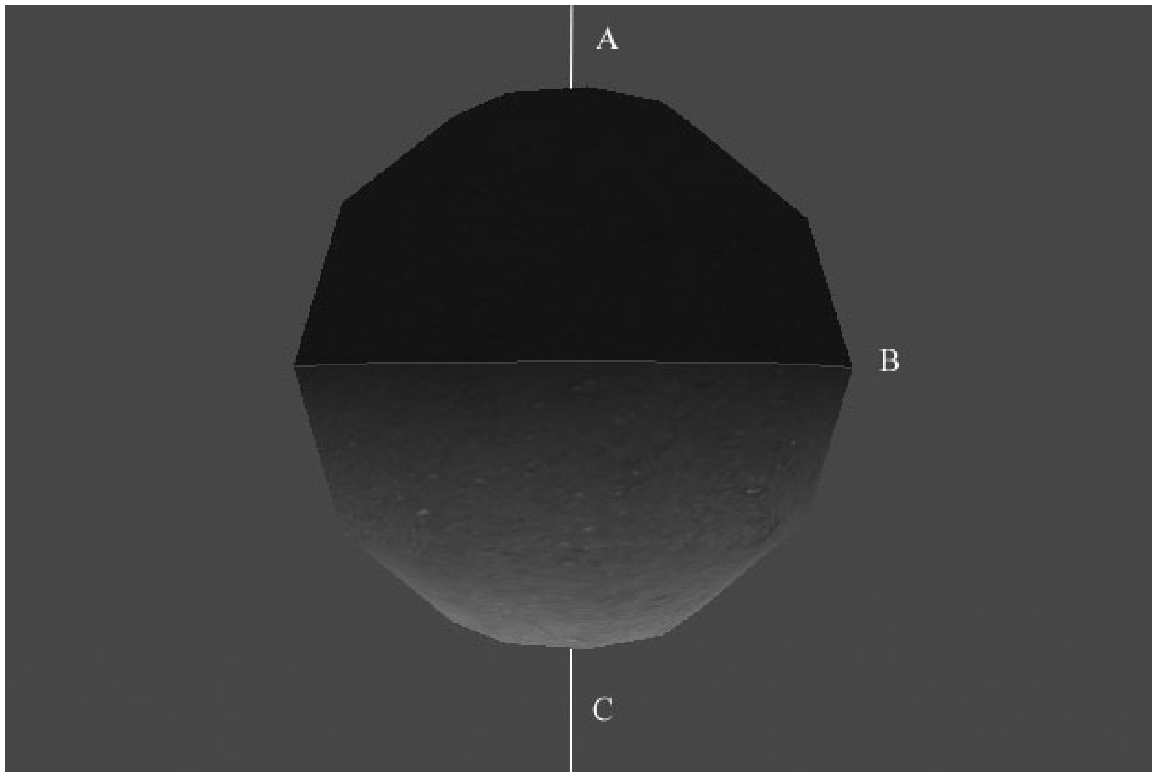

Image 3: A shadowed object

Image 3 shows the lit and shadowed parts of a 3D object, C and A respectively. It also shows the shadow silhouette, B in *image 3*. Rotating the camera 180 degrees around the object's y-axis would result in a similar image as this is a sphere. The shadow silhouette extends all the way around the object which is hard to illustrate with a 2D projection. C is a vector from the light source to the center of the object. A is the central axis of the shadow volume.

5.1.2 Shadow Volumes

The shadow volume is created from the shadow silhouette, in contrast to a shadow silhouette a shadow volume is a collection of faces rather than edges. The shadow volumes used are infinite which basically means that the projection matrix is replaced with a special infinite projection matrix. The vertices also need to contain four values instead of three since W no longer can be assumed to be 1. It will in fact be 0 for points at infinity. But these are the only two values which it can assume. To create a shadow volume the following algorithm is used:

1. Set an extrude distance (multiplier)
2. Compute vector between edge vertex and light (light direction)
3. Add light direction multiplied by the extrude distance to vertex to create a new vertex.
4. Repeat steps 2 - 3 for each vertex in edge
5. Create two triangles from the 4 vertices which are oriented counter clockwise
6. Add vertices to the vertex list of the mesh
7. Add faces to the face list of the mesh
8. For each edge in list repeat 2 - 7

After this algorithm is complete the shadow volume has been created in the form of a mesh. It does however have two holes, one at the front and one in the back. This is not a problem for the depth pass approach, depth fail volumes however need to be capped since they need to be completely sealed or the algorithm will fail. The following algorithm is used to cap shadow volumes.

1. Check if face is shadowed.
2. If shadowed add face to shadow mesh
3. Compute light direction each vertex
4. Add light direction multiplied by extrude distance to each vertex in face.
5. Create a new face using the calculated values
6. Set the W coordinates of the new face to 0
7. Add the face to the shadow mesh
8. Repeat step 2 - 7 for each polygon in the object

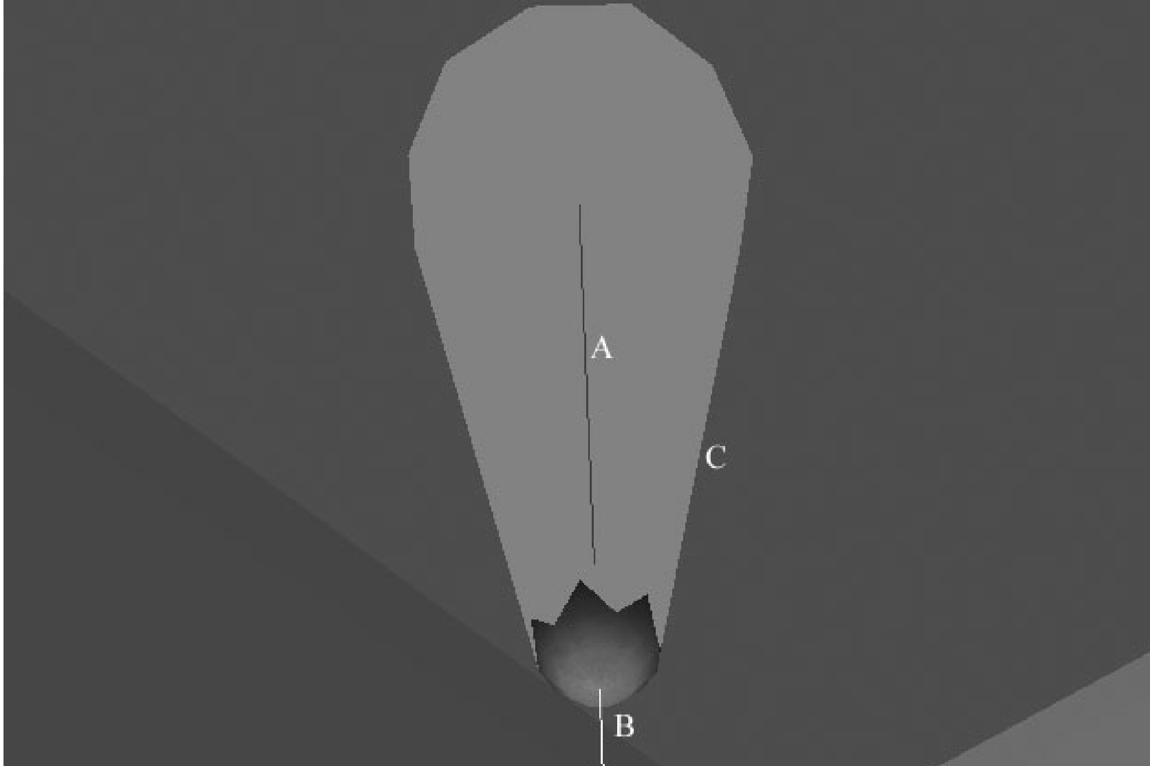

Image 4: Shadow volume extrusion

Image 4 shows a shadow volume as it appears if it is actually rendered to the screen. Keep in mind that this is never done other than for demonstration purposes. The light gray area surrounding A in *image 4* is the rendered shadow volume. The area occupied by the rendered shadow volume is the area in the stencil buffer where the values would be greater than 0. The area outside of A would have a value of 0. B is the vector from the light source to the center of the object (occluder). The vector next to A is the central axis of the shadow volume.

5.1.3 Rendering to the Stencil Buffer

Once the shadow volumes have been created they are rendered to a stencil buffer to determine which parts of the screen are shadowed. This works slightly differently for depth pass and depth fail since depth pass renders from the view point to infinity and depth fail from infinity to the view point. The basic procedure is quite simple, the itty gritty details however can be quite complex. Basically what happens is that instead of rendering the volumes to the screen or color buffer they are rendered to the stencil buffer. The two processes are very similar except that rather than rendering a value directly a counter is used to increment and decrement individual positions in the stencil buffer. Current stencil buffers are also of a lower bit depth than corresponding color buffers meaning that they cannot hold as large numbers. To illustrate the exact process used in this implementation this section is split into a depth pass and a depth fail section.

5.1.4 Depth Pass

Depth pass volumes as mentioned earlier need not be capped. This is because the technique counts entrances and exits through shadow volumes from the view plane to infinity and the point is considered shadowed if values are greater than zero. This holds true even with uncapped volumes. The following algorithm is the one used in the implementation of this thesis:

1. Clear the Z-buffer and stencil buffer, also make sure Z-buffering is enabled and stencil testing is disabled.
2. Render the scene only with ambient light and non-shadow casting lights.
The important part here is that the Z-buffer is updated so that a shadow volume polygon later will not be rendered if it is behind geometry.
3. Disable writing to the Z-buffer.
4. Draw the front faces of the shadow volume, if they are actually drawn (pass depth test) increment the stencil buffer.
5. Draw the back faces of the shadow volume, if they are actually drawn (pass depth test) decrement the stencil buffer.
6. Enable stencil testing, clear Z-buffer, enable writing to the Z-buffer, and enable current shadow casting light.
7. Render scene where the stencil is zero (lit).

5.1.5 Depth Fail

Depth fail volumes need to be capped. This is because depth fail renders from infinity to the view plane and the points are considered shadowed if the values are greater than zero. This doesn't hold for uncapped volumes since the counter can become negative when it passes through a hole in the volume. The following is the algorithm used for depth fail:

1. Clear the Z-buffer and the stencil buffer, also make sure Z-buffering is enabled and stencil testing is disabled.
2. Render the scene only with ambient light and non-shadow casting lights.
The important part here is that the Z-buffer is updated so that a shadow volume polygon later will not be rendered if it is behind geometry.
3. Disable writing to the Z-buffer.
4. Draw the back faces of the shadow volume if they fail depth test, increment the stencil buffer.
5. Draw the front faces of the shadow volume if the fail depth test, decrement the stencil buffer.
6. Enable stencil testing, clear Z-buffer, enable writing to the Z-buffer, and enable current shadow casting light.
7. Render scene where the stencil is zero (lit).

5.1.6 Analyzing the Depth Pass and Depth Fail Algorithms

Steps 1 to 3 are done once per rendering cycle, they are in essence setup steps which

create a blueprint to work on. Steps 4 to 5 are done for each shadow volume for the current shadow casting light. A single light can have multiple shadow volumes because it can cast shadows from multiple objects. Steps 6 and 7 are done for each shadow casting light in the scene. They must be done separately for each light, since the lights usually don't shine on exactly the same parts of the final image. The last step in the depth pass and depth fail algorithms is where the actual color buffer is updated to show where the shadows are cast. After this has been done for each shadow casting light the rendering cycle is complete and a new one can begin.

5.2 Octrees with Portals

As mentioned in the Theory chapter an octree contains nodes and leaves. Each leaf contains one face list for each texture that exists within this leaf. These lists contain the actual faces in the leaf. The list of vertices are contained within the octree itself, that is individual leaves do not contain such a list. The faces in the leaves merely contain references to the vertex list in the octree. This is done in the same manner as in meshes where the faces don't have actual vertices but references to such. The lists are saved in so called native IO buffer which makes transferring it to the graphics card through OpenGL much faster. OpenGL uses 4 arrays, a vertex array, a face array, a normal array, and a texture coordinate array. When the octree is to be rendered the corresponding arrays are sent to the graphics card through the OpenGL API. Note that only the faces to be rendered are actually sent to the graphics card which means only visible faces. Visible faces in this context are faces which lie within visible leaves, this does not mean that they could not later be removed by processes such as back face culling or other clipping or culling steps. This approach however minimizes the amount transformations on vertices the graphics card needs to perform, thus boosting performance. As mentioned earlier this is one of the main purposes of an octree. The following algorithm is used to create the octree:

1. Read vertices and faces from file
2. Create bounding cube (bounding box with all sides of equal size)
3. Create the root node
4. Count polygons in node
5. If end conditions not met proceed, otherwise create leaf
6. Create 8 sub-cubes
7. Count polygons in each cube
8. If cube contains polygons proceed, otherwise discard cube
9. If end condition not met create node and repeat step 6 - 9, otherwise create leaf

The algorithm is recursive, all children are created for one branch until the end condition is met and a leaf is formed before the next branch is constructed. When all branches have been constructed the algorithm is complete and an octree representing the world has been made. Steps 1 - 5 are setup steps which occur only once per octree. Steps 6 - 9 are the recursive bit of the algorithm, they occur for each branch until all branches have been created. This is a general octree which can be used for a number of things, including collision detection and rendering.

Multiple octrees are linked together through portals. Portals always exist in pairs, one in each octree that are to be linked. The portal contains a reference to the octree it links to and its corresponding portal in that octree. Indirectly this means that a portal knows the position it links to in the linked octree. Portals are contained in their own list in the octree. This may seem confusing, however octree in this context refers to this actual implementation of the octree object in Java, not the octree structure as described in the theory chapter. It is done in this fashion for two different reasons. Firstly portals may be so large that they occupy multiple leaves which would cause issues when placing them in the octree. In such a case they would have to be split and placed into all the leaves separately. The end result would be that multiple frustums would have to be created if more than one part of the original portal would be visible. This is not desirable since this would mean rendering the same octree several times which is costly. Secondly it is desirable to render the octrees in reverse order, which means starting at the octree linked by the innermost portal in the chain. Portals can be placed either by hand or by an algorithm which chooses good places to split the world into multiple octrees. This implementation uses the first approach, this because creating such an algorithm is beyond the scope of this thesis and placing them by hand offers a larger degree of control. It does however put more strain on the level designers. That being said it is now time to examine how rendering an actual octree with portals works in practice. The following algorithm describes the procedure:

1. Start in the octree where the camera is located
2. Check if any portals are visible
3. If visible create new frustum and proceed. Otherwise skip to step 5
4. Repeat steps 2 - 3 until no portals are visible
5. Start at the root node of the current octree (The last octree to be rendered)
6. Check if the bounding box is visible
7. If visible proceed, otherwise discard octree from rendering cycle
8. Render the visible leaves (recursively) and return to previous octree
9. Repeat steps 5 - 8 until all octrees have been rendered

When the leaves are rendered the faces are first placed in a render list. This in turn contains as many lists as there are textures in the leaves. These lists are then rendered sequentially. It is done in this fashion because swapping textures is expensive and is to be avoided as much as possible.

```
// activeoctree is the octree where the camera is
activeoctree.draw();

void draw(){
        for(all portals in octree){
                if(portal is visible in frustum){
                        create new frustum;
                        //draw octree that portal links to with newly created frustum
                        portal.linkedOctree.draw();
```

```
            }
        }
        //recursively find all visible leaves in octree
        build renderlist();
        render renderlist();
}
```

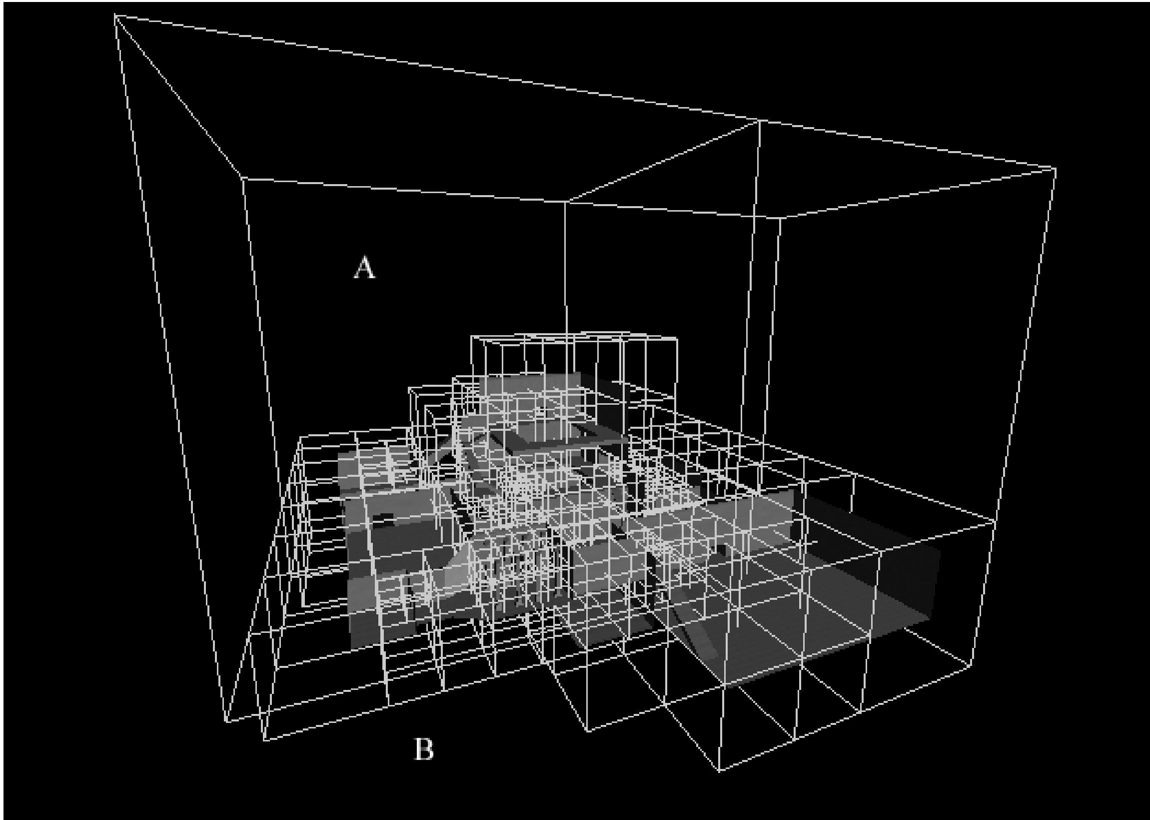

Image 5: An octree with bounding boxes

Image 5 shows how an octree divides a world into nodes and leaves. In fact only the bounding boxes of the leaves are drawn. This octree uses a maximum number of polygons and a maximum number of levels as its end conditions. As shown this means that the sizes of the bounding boxes the leaves can vary greatly. A in *image 5* is located in one such leaf and the possible size difference between leaves is well illustrated in this example. B is located in an area where there are no polygons and thus there are no leaves. This shows that the octree adapts at least partially to the world it represents. Removing boxes in this fashion minimizes the time it takes to traverse the tree and produce the final image.

5.3 Bump Mapping

In order to do correct bump mapping every vertex has to have its own TBN matrix. The rows in the matrix consist of the tangent, binormal and normal. These three rows are

calculated per face according to P. Taylor [6] in the following algorithm.

```
Vector1 = Vertex3 – Vertex2;
Vector2 = Vertex1 – Vertex2;
DeltaU1 = Vertex3.u - Vertex2.u;
DeltaU2 = Vertex1.u - Vertex2.u;
Normal = normalize(DeltaU2*Vector1-DeltaU1*Vector2);
Tangent = normalize(crossproduct(Normal,Vertex.N));
Binormal = normalize(crossproduct(Tangent,Normal));
```

The u in the pseudo-code is the u part of the vertex's texture coordinate, and N is the vertex's normal. These three vectors, Normal, Tangent, and Binormal, are then sent to the vertex shader whenever bump mapping is to be calculated. The following happens in the vertex shader. The vectors are combined into a TBN matrix and used to transform the vertex normal and the light vector into texture space. The transformed and projected position of the vertex is calculated. The output from the vertex shader is then interpolated between the vertices to become the input to the fragment shader where the final bump mapping calculations are performed. The following happens in the fragment shader. The color at the appropriate texture coordinate is fetched, as well as the proper normal from the bump map. The final normal is then calculated by adding the interpolated and transformed normal received from the vertex shader to the fetched normal from the bump map. The resulting normal is then normalized. Finally the color of the fragment is calculated by computing the dot product between the transformed light vector and the final normal and multiplying the result by the color. The whole process can be described in the following algorithm:

1. Calculate the normal, tangent and binormal
2. Send them to the vertex shader
3. Produce a TBN matrix
4. Transform light vector and normal to texture space
5. Transform and project vertex position
6. All vertex shader outputs are automatically interpolated and send to the fragment shader
7. Fetch appropriate texture color and bump map normal
8. Calculate new final normal
9. Calculate final color and write to the frame buffer

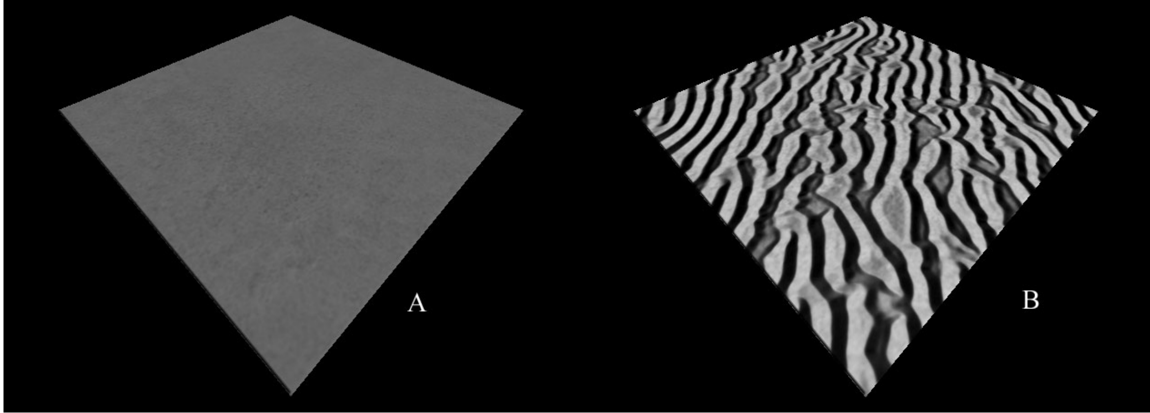

Image 6: Plain surface **Image 7:** Bump mapped surface

Images 6 and *7* above show the effect of bump mapping. A and B are surfaces both textured with the same texture. The only difference is that B is bump mapped to give a rippled impression.

6 Discussion

During this project a few problems have been encountered, some of which have been solved successfully, and some which haven't.

The most significant problem encountered with respect to shadow volumes was Z-buffer "fighting". This occurs when two fragments end up close enough that floating point errors determine which one actually is visible. In practice this causes the visible fragment to shift from one to the other depending on the positioning and rotation of the camera. In the end this problem was solved by introducing a bias value in the vertex shader. This may not be the best way to solve the issue, however several approaches were implemented and tested and among those this proved to be both the fastest and most robust. Basically a bias value is introduced after projection in the vertex shader when handling the shadow volume. This causes subsequent tests such as depth testing to register the projected vertex and all the fragments associated with the polygon to lie at a more desirable position. One way to look at it is to view the bias value as a VIP card, the card entitles the bearer to skip ahead in line a certain number of positions and if no one is in front of the bearer after the card has been used the bearer gets to pass. All other issues encountered pertaining to shadow volumes were resolved quickly and are therefore not worth discussing.

With respect to the octree implementation the biggest problem encountered which hasn't been solved due to time constraints is creating new optimal view frustums. This is done correctly as long as a portal and the current frustum do not intersect. This occurs when a portal is only partly visible and this issue has not been resolved by creating an optimal frustum. Instead the entire portal is used to create the new frustum which means that more objects are passed down the rendering pipeline than necessary. The issue does not affect octree correctness however and the implementation works nicely in all other respects. Performance wise there are also some other gains to be had from various optimizations. These include triangle clipping to ensure polygons lying in a single leaf only, the before mentioned frustum creation would also speed things up.

The bump mapping implementation in this thesis does not work completely correctly for objects composed of faces which lie in different planes. An error occurs which causes the faces of the mesh to appear in different shades. This is akin to what one would expect to happen if the face normal were used to make bump mapping calculations instead of vertex normals. However since vertex normals are used it is strange that this problem occurs. It is more likely that the problem originates elsewhere and we cannot be completely certain that it is within our code. This is due to the fact that the Cg support in JOGL isn't fully developed yet.

7 Evaluation

The performance is still not optimal concerning shadow volumes for several reasons. The biggest reason is that shadow volume extrusion is still done in software which puts unnecessary strain on the main CPU. Polygon rendering for shadow volumes is not done in an optimal way which would be either triangle strips or triangle fans. This problem would be solved with a proper vertex extrusion in the vertex shader though. As for appearance and robustness the shadow volume algorithm used in this thesis works nicely. Shadow volumes can be entered without causing artifacts or other unwanted effects. In other words the shadow volumes are not up to commercial standards regarding performance at this date. They do however work correctly and robustly and further optimizations would in all probability make them run smoothly on GeForce4 and better graphics cards.

The octree implementation performs quite well. The fill rate isn’t optimal due to the fact that triangles are drawn more than once if it lies within multiple leaves. This could be solved with triangle clipping. The before mentioned optimal frustum would also boost performance. On issues not pertaining to performance the octree implementation works nicely as well. It produces the desired results and is robust. In other words it satisfies the goal for this thesis but with room for further optimizations if necessary.

8 Conclusion

All goals for this thesis have been met except complete bump mapping correctness. The shadow volume and octree implementations work correctly and perform above the demands set down by the goal. To restate the goal, the goal was to produce a shadow volume, octree, and bump mapping implementation which works and is easily extendable. That means that performance takes a back seat to correctness. Correctness in this case is not the strict mathematical definition but rather a measure of robustness and that the implementations produce the desired result.

9 Further Work

There are many aspects which could be improved which were left alone because of time constraints. The performance of the shadow volume implementation could be optimized in several ways. Depth clamping, double sided stencil testing, hardware vertex extrusion, and view frustum culling with respect to light and object placement. M. McGuire, J. F. Hughes, K. T. Egan, M. J. Kilgard, and C. Everitt [7] describe several ways to speed things up. Appearance wise a soft shadow extension would greatly increase the realism. For octrees a polygon clipper and optimal frustum creation could be implemented. Coordinate system transforms could also be implemented. This is needed to remove the restriction of multiple octrees needing to use the same coordinate system when linked together by portals.

10 References

1. G. S. Owen, “Ray Tracing”, 1999/07/20, http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtrace0.htm, 2004/01/12
2. C. Everitt, M. J. Kilgard, “Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering”, 2002/03/12, http://developer.nvidia.com/attach/5625, 2004/01/12
3. M. J. Kilgard , “Shadow Mapping with Today’s OpenGL Hardware”, 2000/03/10, http://developer.nvidia.com/attach/1838, 2004/01/12
4. J. Tsiombikas, “Volume Shadows Tutorial” , http://www.gamedev.net/reference/articles/article2036.asp, 2004/01/12
5. M. Feldman, “Introduction to Binary Space Partition Trees”, 1997, http://www.geocities.com/SiliconValley/2151/bsp.html, 2004/01/12
6. P. Taylor, “Per-Pixel Lighting”, 2001/11/13, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndrive/html/directx11192001.asp, 2004/01/12
7. M. McGuire, J. F. Hughes, K. T. Egan, M. J. Kilgard, C. Everitt, “Fast, Practical and Robust Shadows”, 2003/11/06, http://developer.nvidia.com/attach/5900, 2004/01/12