A Highly Parallelized Approach to Silhouette Edge Detection for Shadow Volumes in
Three Dimensional Triangular Meshes

A thesis presented to
the faculty of
the Russ College of Engineering and Technology of Ohio University

In partial fulfillment
of the requirements for the degree
Master of Science

Chad L. Mourning
November 2008

© 2008 Chad L. Mourning. All Rights Reserved.
This thesis titled

A Highly Parallelized Approach to Silhouette Edge Detection for Shadow Volumes in

Three Dimensional Triangular Meshes

by

CHAD L. MOURNING

has been approved for

the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

__________________________________________

David M. Chelberg

Associate Professor of Electrical Engineering and Computer Science

__________________________________________

Dennis Irwin

Dean, Russ College of Engineering and Technology
ABSTRACT

MOURNING, CHAD L, M.S., November 2008, Computer Science

A Highly Parallelized Approach to Silhouette Edge Detection for Shadow Volumes in Three Dimensional Triangular Meshes (111 pp.)

Director of Thesis: David M. Chelberg

This thesis is intended to explore and investigate the possibility and usefulness of a completely hardware based silhouette edge detection mechanism for use in shadow voluming. Several implementations were developed, and rigorous testing was performed to determine if this goal was attainable. The implementations were integrated into an existing three-dimensional game engine to show that they were applicable in a practical setting. It was determined that GPU driven silhouette edge identification is indeed viable and, in fact, preferable for large scenes. This work culminated with an implementation of silhouette detection using CUDA, a fairly young GPGPU technology, performed entirely on the graphics card.

Approved: _____________________________________________________________

David M. Chelberg
Associate Professor of Electrical Engineering and Computer Science
ACKNOWLEDGMENTS

I would like to thank Dr. David Chelberg for being my graduate advisor over the past two years.

I would like to thank Dr. David Chelberg, Dr. Teresa Franklin, and Dr. Chang Liu for securing funding for me over the past two years.

I would like to thank Scott Nykl and Mitch Leitch for helping me develop the STEAMiE Educational Game Engine over the past two years.

I would like to thank Ohio University for allowing me to stay here for the past six years.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>LIST OF EQUATIONS</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>CHAPTER 1: INTRODUCTION</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>CHAPTER 2: LITERATURE REVIEW</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>CHAPTER 3: TECHNICAL FUNDAMENTALS</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Section 1: Rotation Matrices</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Section 2: Quaternions</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Section 3: GLSL</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Section 4: Frame Buffer Objects</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>CHAPTER 4: THE IMPLEMENTATION ENVIRONMENT</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>Section 1: Introduction</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>Section 2: The Basics</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>Section 3: Practical Implications</td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>CHAPTER 5: EXPERIMENTAL DESIGN</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>Section 1: Introduction to the Design Process</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>Section 2: Data Structures and Implementation Interfaces</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>Section 3: Support for Soft Shadowing</td>
<td></td>
<td>43</td>
</tr>
<tr>
<td>Section 4: Silhouette Edge Detection and Surface Visibility with Respect to a Light Source</td>
<td></td>
<td>44</td>
</tr>
<tr>
<td>Introduction</td>
<td></td>
<td>44</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>History of CPU Speed vs. GPU Speed</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Shadows as an indication of spatial relationships</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>Shadows as an indication of receiver geometry</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>Shadows as an indication of hidden geometry</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>An example scene with shadow volumes</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>The same scene without visible volumes</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>The Lower Caps of Shadow Volumes</td>
<td>19</td>
</tr>
<tr>
<td>8</td>
<td>Hard and soft shadow generation</td>
<td>21</td>
</tr>
<tr>
<td>9</td>
<td>Hard shadowing vs. soft shadowing</td>
<td>21</td>
</tr>
<tr>
<td>10</td>
<td>Humanoid casting a hard shadow</td>
<td>22</td>
</tr>
<tr>
<td>11</td>
<td>Humanoid casting a soft shadow</td>
<td>23</td>
</tr>
<tr>
<td>12</td>
<td>A soft shadow blurred by a light source</td>
<td>24</td>
</tr>
<tr>
<td>13</td>
<td>GPU vs. CPU example 1</td>
<td>27</td>
</tr>
<tr>
<td>14</td>
<td>GPU vs. CPU example 2</td>
<td>28</td>
</tr>
<tr>
<td>15</td>
<td>GPU vs. CPU example 3</td>
<td>29</td>
</tr>
<tr>
<td>16</td>
<td>A scene containing two faces and two light sources</td>
<td>47</td>
</tr>
<tr>
<td>17</td>
<td>The same scene with the light sources transformed as described above</td>
<td>47</td>
</tr>
<tr>
<td>18</td>
<td>The original scene</td>
<td>48</td>
</tr>
<tr>
<td>19</td>
<td>Face 2 rotated 45 degrees clockwise</td>
<td>49</td>
</tr>
<tr>
<td>20</td>
<td>Face 2 translated</td>
<td>50</td>
</tr>
</tbody>
</table>
Figure 21: Face 3 rotated 45 degrees clockwise .............................................................51
Figure 22: Face 3 translated............................................................................................52
Figure 23: Face 4 rotated 45 degrees counterclockwise .................................................52
Figure 24: Face 4 translated............................................................................................53
Figure 25: Face 5 rotated 45 degrees clockwise .............................................................53
Figure 26: Instructional diagram for encoding matrices into a texture .......................60
Figure 27: The encoded visibility matrices of a cube .....................................................61
Figure 28: The encoded visibility matrices of a human torso .......................................62
Figure 29: The classical approach on CPUs of varying speeds .....................................73
Figure 30: Results for the software implementation on the CPU .................................75
Figure 31: Results for the hardware approach using fragment shaders .......................76
Figure 32: Results of the CUDA GPGPU implementation .........................................77
Figure 33: Close up view of the results for the CUDA implementation ....................78
LIST OF EQUATIONS

Equation 1 .......................................................................................................................81
Equation 2 .......................................................................................................................81
Equation 3 .......................................................................................................................84
Equation 4 .......................................................................................................................84
Equation 5 .......................................................................................................................84
Equation 6 .......................................................................................................................91
Equation 7 .......................................................................................................................91
Equation 8 .......................................................................................................................92
Equation 9 .......................................................................................................................92
Equation 10 .....................................................................................................................93
Equation 11 .....................................................................................................................93
Equation 12 .....................................................................................................................93
Equation 13 .....................................................................................................................94
Equation 14 .....................................................................................................................95
Equation 15 .....................................................................................................................95
CHAPTER 1: INTRODUCTION

There exists a need in modern computer generated scenes for interactions between lights and objects. The fundamental interaction described in this thesis is shadowing; shadows give depth to a scene and visual cues about objects’ spatial relationships. The two general classes of shadowing techniques used today are shadow maps [2] and shadow volumes [1]. A brief description of both techniques will be given, but the majority of the work described in this thesis focuses on shadow volumes. A necessary step in the creation of a shadow volume is the identification of the silhouette edge set of the shadow casting object, the occluder. The purpose of this thesis is to describe a new technique useful in facilitating the creation of shadow volumes by increasing the speed of identifying the silhouette edge set for triangular meshes. This thesis also compares the efficiency between several different implementations of silhouette edge detection algorithms.

Some fundamental understanding of certain topics will be required to understand the material and will be discussed in the Technical Fundamentals section, Chapter 3. The topics covered there will include: rotation matrices, quaternions, the OpenGL shading language, and OpenGL frame buffer objects. There will be some discussion of the environment into which my shadowing implementation is incorporated. The implementation discussed is incorporated into a fully developed 3D game engine for more rigorous and versatile testing. The 3D game engine used for this implementation is the STEAMiE Educational Game Engine [20].
Additionally, due to the per-face and per-object nature of silhouette edge detection, the advantages of modern graphic hardware when it comes to parallelization are also discussed. Over the past several years GPUs have developed significantly faster than CPUs when used for optimal applications. Optimal applications can be defined as those that have a large data set and are easily solved with highly parallelized solutions. A brief history of this is given in Figure 1 [5]. In particular, NVIDIA’s CUDA (Compute Unified Device Architecture) is a library that allows general purpose applications to run on the graphics card. The CUDA software package was alpha released in September 2006, and version 1.0 was released the following July. CUDA is ideal for applications that can perform in parallel the same operations on a large set of data. As described later, silhouette detection is exactly such an application.

Figure 1. History of CPU Speed vs. GPU Speed [5]
CHAPTER 2: LITERATURE REVIEW

A survey paper by Hasenfratz, et. al. [3], in 2003 gave some wonderful examples, of the importance of shadows in a scene, which will be referenced frequently in this chapter. According to the survey, Hubona, et. al. [16], pointed out that without shadows the spatial relationships between objects may not be able to be determined. For example, given a random viewing angle, it may not necessarily be easy to determine the relative position of an object with respect to a surface on the other side of him with respect to the viewpoint. Figure 2 gives a great example with the same occluder, at the same viewing angle with the receiver at an unknown distance away. The left hand image has no shadows, and there is no way of telling the spatial relationship between the occluder and the receiver. The three remaining images all contain shadows, and show how the interaction would differ with receivers at different distances from the occluder.
Figure 2. Shadow as an indication of spatial relationships [3].

Mamassian et. al. [17], gives another great example of the usefulness of shadows; they can lead to determining the geometry of the receiver. In Figure 3, the left image containing no shadows has very little information about the geometry of the receiver. In the right hand image of Figure 3, the shadows reveal the hilly geometry of the receiver.
Figure 3. Shadows as an indication of receiver geometry [3].

Another example given by Mamassian, et. al. [17] is that shadows can reveal hidden geometry of the occluder. For instance, if the occluder is viewed from an angle such that it is obscuring some of its own geometry, which in fact is always the case, the position of the light source might cast a shadow that reveals this hidden part of the occluder’s geometry. Figure 4 shows a great example of such a phenomenon. In each of the three images, examine the far hand of the model; the contents of the hand are hidden from the viewing angle, but the shadow reveals the differences.

Figure 4. Shadows as an indication of hidden geometry [3].
Shadowing in computer graphics is not a new concept at all. In 1977, Franklin Crow [1] devised a shadowing technique that is now commonly known as the shadow volume technique. The principal behind this technique is that the world can be divided into two sections: those in shadow, and those not in shadow. If a surface lies within the shadow volume, then that surface has a shadow cast upon it and should be treated accordingly. All surfaces outside the shadow volume are treated like normal. The shadow volume can be created in a two step process. The ‘walls’ of a shadow volume can be created by extruding the silhouette edges of an occluder from a light source to infinity. To create a closed volume of space, caps must be added to the top and bottom of the volume. Figure 5 is an example of a scene where shadow volumes have been made visible. Figure 6 is the same scene without the volumes, but with the trapped surfaces shadowed.

Figure 5. An Example Scene with Shadow Volumes.
Shadow volume generation became much easier with the advent of hardware accelerated stencil buffering [6]. A stencil buffer is a tool used to limit which portions of the frame buffer are rendered to. First a stencil buffer is created, and then when subsequent pixels are drawn into the frame buffer, only the stenciled portions remain. OpenGL supports rules to increment/decrement the stencil buffer per pixel when it fails/passes the depth test. This enables the hardware accelerated identification of which pixels lie on a surface inside the shadow volume in two renders of the shadow volume.

Crow’s [1] original algorithm can be easily performed by incrementing the stencil buffer when the front facing shadow volume walls are in front of the surface, and decrementing when the back facing shadow volume walls are in front of the surface. From the camera’s perspective, it enters the shadow through the front facing walls and leaves through the back facing walls. Therefore, if the front wall is in front of a surface
and the back wall is behind the surface, the surface pixel is added to the stencil to be acted upon later. This algorithm has two obvious vulnerabilities. First, if the camera lies within the shadow itself, then the front facing walls will not be incremented because they are behind the near clipping plane. This means that the shadow will ‘disappear’ if the camera is inside the shadow. Second, if the back facing walls are beyond the clipping plane, then the stencil is never decremented and there will appear to be a shadow cast on the clipping plane.

Crow’s method is frequently referred to as the z-pass method, because the stencil is operated upon if the depth test succeeds. A variation was discovered independently by Bilodeau and Songy [8] as well as by Carmack [9], that is commonly called the z-fail method or Carmack’s Reverse. In contrast to the z-pass method, the z-fail method increments the stencil buffer when the back facing walls are behind a surface and decrements when the front walls are behind the surface. This, again, results in a stencil where surfaces “trapped” between the walls of the volume are marked. The cases where the z-pass method fails both work here. The first, when the camera lies inside in the shadow, still works because the back facing walls, which are still visible, generate an increment. The second case, when the back facing walls are beyond the clipping plane, is also moot, because if the back facing walls are beyond the plane, then the stencil is never incremented.

There are some complications with using the z-fail method, most notably, capping the shadow volumes becomes even more important. If an uncapped shadow volume is looked through along the axis of extrusion, there is no back facing polygon to increment
the stencil so a hole will be seen in the generated shadow. A simple capping procedure does exist; the face’s cast shadows must be projected to either end of the shadow volume. The problem comes when either cap is cut off by the near and far clipping planes. This problem was first noticed by Diefenbach [10] in 1996, and his solution worked in many cases but failed in many others. An alternative solution presented by several authors suggests finding the intersection between the clipping planes and the shadow volume and create the cap there. Doing this solution properly can be highly computationally expensive, and approximation tend to leave harsh artifacts, caused by pixel leaks, that makes this solution impractical in real-time applications. Another solution was presented by Everitt and Kilgard [15] in 2002. Their solution makes use of homogeneous coordinates and an ‘infinite’ clipping plane to solve the clipping plane problem. This solution works well in specialized environments, but it must be noted that infinite clipping planes lead to some loss in depth buffer precision, which can lead to artifacts. Additionally, loss of depth buffer precision in a practical implementation can affect simple, but common, operations such as selection. In summary, the caps of the shadow, without regard for clipping planes, can be created by projecting back facing polygons onto the far end of the volume and front facing polygons onto the near end of the volume, with respect to the occluder, but this may or may not work in any particular situation. Figure 7 is an example of the lower caps of shadow volumes; the upper caps are contained inside the objects.
The actual number of shadow volume faces that need to be rendered will obviously alter depending on scene and the position and viewing angle of the camera. A 2003 paper by McGuire [14] claims that for arbitrary objects the number of silhouette edges will approach $f^{8/10}$, where $f$ is the number of faces in the model. This means that if the developer is attempting to shadow volume an object, the developer should be prepared to render an extra $2f^{8/10}$ quadrilaterals. For strictly convex objects the number of silhouette edges is lower, approaching $f^{1/2}$.

One way to reduce the number of shadow volume faces that are rendered is to use some sort of frustrum, oct-tree, or BSP tree culling. Batagelo and Júnior [19] present
one modified BSP approach specialized for dealing with shadow volumes. There are some complications that can occur when choosing to mix shadow casting object with non-shadow casting objects that will be discussed in chapter 5, section 3.

Another topic in shadow generation is the difference between hard shadows and soft shadows [3]. Hard shadows are shadows where portions of a surface are either entirely in shadow or entirely out of shadow. On the other hand, soft shadows are shadows where portions of a surface are partially shadowed. The portion of a shadow that is entirely in shadow is called the umbra whereas the portion that is partially shadowed is called the penumbra. Umbra occurs when all of the light from a source is blocked, whereas penumbra occurs when only some of the light from a source is obscured. Lights modeled as point sources always generate hard shadows, but lights modeled with volume/area may generate a combination of umbra and penumbra. My implementation does support soft shadowing for volumetric lights and will be discussed in more detail later.

The survey paper by Hasenfratz, et. al. [3], has a figure showing an example of the geometrical representations of hard shadowing and soft shadowing. In figure 8, the left hand image is an example of a hard shadow generated by a light source approximated as a point. It can be seen that the shadow is hard consisting entirely of umbra. The two right hand examples show a light source with area and the umbra and penumbra it generates.
Figure 8. Hard and soft shadow generation [3].

Figure 9 is an illustration of the scene rendered using both hard and soft shadowing. It can be quickly seen that the results are strikingly different.

Figure 9. Hard shadowing vs. soft shadowing [3].

The authors of the same survey paper [3] seemed to feel that soft shadows were superior in all cases where they could be used, but depending on the specifications of the scene, hard shadowing may be considered more pleasant looking that soft shadowing. Sometimes the blur effect can be so great that almost no shadow is perceived at all. This happens when light sources are too much larger than the objects that are occluding them.
This blur effect also depends on the distance between the light source and the object. Figure 10 shows an example, from the STEAMiE Educational Game Engine, of a hard shadow being cast from a humanoid model by a light source approximated as a point source.

Assarsson, et. al. [18], presented that a simple way of modeling soft shadowing is by super-sampling the light source. Figure 11 has the same distance between the
humanoid and the light sources, but it is modeled with a volumetric spherical light source simulating soft shadowing with 32 samples.

Figure 11. Humanoid casting a soft shadow.

Figure 12 shows the same scene with a much larger volumetric light source. The blurring effect is quite noticeable. There are some artifacts in this example because the close proximity of a large light source not only increases the blur effect but also makes the lack of sampling precision more noticeable. This soft shadow is approximated using 128 samples. It should be noticed that even with a higher sampling rate, this image has
many more noticeable artifacts. This shows that the same approach may yield very different results in slightly different situations.

Figure 12. A soft shadow blurred by a light source.

Another area that should be discussed is the intricacies of multiple shadow casting objects in the same scene. The same survey paper by Hasenfratz, et. al. [3, p. 756], succinctly said, “the shadow of the union of the objects can be larger than the union of the shadows of the objects.” The implementation presented in this thesis does not attempt to reconcile this effect when casting shadows of multiple objects. While each shadow itself is physically exact for the given model, the entirety of the scene may not necessarily be correct. In general, this effect is not very noticeable.
Shadow volumes are an occluder-based approach to shadow generation. As the complexity of the occluder’s geometry grows so does the complexity of the computation. This allows for the generation of pixel-level physically accurate shadows with respect to the occluder’s model. An alternative approach to shadow generation is called *shadow mapping* [2]. Shadow mapping is a receiver-based approach to shadow generation. A shadow map is created by rendering the scene from the perspective of the light source, storing the resulting depth buffer as a shadow map. This gives the depth information needed to know what the nearest surface to the light is, and any surface further than the nearest will be in shadow.

A hardware implementation of shadow volumes is highly desirable because in most applications the CPU becomes the bottleneck because its resources are fully utilized before those of the graphics card. There have been several previous attempts at creating hardware only shadow volumes. McCool [11] previously devised a hybrid hardware only shadow volume-shadow map approach in 2000; while it has some benefits, it suffers from the same sampling artifacts that all shadow map techniques suffer from, due to limited depth map precision. Brabec and Seidel [4] created a hardware implementation of shadow volumes in 2003 using a combination of textures, fragment shaders, and frame buffer objects. Their approach required that the geometry of the object be encoded into a texture; this thesis will do something similar for one of its implementations. More details will be given later in chapter 5.2.2, Encoding Data in a Texture. Only the fragment shader has access to texture information, so in these situations the bulk of the work must be done there.
A set of power point slides published by Pilipaviciute and Robinson in 2005 gave some excellent examples of the parallel processing power of modern GPUs [13]. They implemented several easily parallelized problems on GPUs using Cg shaders. Cg is a NVIDIA alternative to the GLSL shaders that this thesis’s implementation uses that will be described later in chapter 3, section 2, GLSL. GLSL is a standardized cross-platform shading implementation, so it was chosen over Cg because of Cg’s proprietary nature.

The next three figures show a comparison between running a selection of problems on four different platforms. The blue line represents the GPU of a Mac Powerbook G4 with an ATI Mobility Radeon 9700 which contains 4 pixel pipelines and 2 vertex pipelines with 128MB VRAM. The red line represents the CPU of the same Mac Powerbook G4 with a 1.5GHz processor and 1GB 333MHz DDR SDRAM. The yellow line represents the same problem running on the CPU of a Linux machine running at 2.4GHz with 1GB RAM. The green line represents the same problem running on the CPU of a Windows XP machine running at 2.6GHz with 1 GB RAM. Figure 13 shows the performance results for the four machines while attempting to find the maximum element in an array of size $n$. The CPU implementation is a standard array traversal performing in $O(n)$. The GPU implementation is a parallelized approach running in $O(\log_4 n)$. A texture containing each value is passed in, and in each pass every fragment compares four of the values, resulting in a texture $\frac{1}{4}$ the size, until only one element remains.
Figure 13. GPU vs. CPU example 1 [13].

Upon examination it will be noticed that there is some overhead for running the GPU implementation, so it is not ideal for small datasets. This will be the case for all of the GPU implementations presented in this thesis, if for no other reason than the sending and retrieving of the data to and from the graphics card. The larger the dataset the less this overhead affects the overall performance.

Figure 14 shows the results of a classic data sorting problem. Again, the GPU starts slow because of the overhead for setting up the problem and overtakes the CPUs when the dataset becomes large enough.
Figure 14. GPU vs. CPU example 2 [13].

A third example given is matrix multiplication. These results tie in well to this thesis’s implementation to surface visibility with respect to a light source which relies heavily on matrix multiplication. Figure 15 shows how much processing can be done easily on a GPU using matrix multiplications. The GPU in question can easily get one million matrix multiplications done per millisecond.
Information on ongoing work into general purpose computation done on graphics hardware can be found at [http://www.gpgpu.org](http://www.gpgpu.org) [7]. Some nice tutorials are posted on this site, most of which use Cg shaders. The majority of the work in this thesis will use either GLSL shaders or CUDA.

In September 2006, NVIDIA alpha released their CUDA (Compute Unified Device Architecture) [5]. CUDA version 1.0 was released in July of 2007, so it is still a fairly young technology. CUDA is intended to be an extension to C that allows for general purpose programming on the graphics card. Due to the highly parallel nature of the surface visibility problem the parallel processing power of the graphics card can be exploited to increase the speed of silhouette edge identification. Additionally, the GPU can simply be used as a co-processor to help load balance the computation of the
silhouette edge detection. The most successful implementation in this thesis will use CUDA to compute the silhouette edge set of some scenes.

Even though it is unrelated to the implementations inside this thesis, it should also be noted there are several shadowing implementations using ray tracing. As always, ray tracing produces high quality results, but it is still not suitable for real-time rendering. Laine et. al. [12] has attempted to improve upon this by using a single shadow casting ray and reconstructing the shadow visibility function rather than using hundreds of shadow casting rays. This technique results in an order of magnitude performance speed up. This is not to say that it can perform at real-time speeds. In 2004, when this technique was presented scenes were rending at a rate of one frame per two and a half minutes. Frame rates of less than 10 per second are considered non-interactive and frame rates of less than 30 are undesirable.
CHAPTER 3: TECHNICAL FUNDAMENTALS

Section 1: Rotation Matrices

Some fundamental knowledge of rotation matrices is needed in order to understand my visibility solution using accumulated 4x4 matrices. The basic premise is that there exists a 4x4 matrix that can represent a set of translation, rotation, and scales for a set of points around a local origin. A more thorough tutorial is given in Appendix A, section 1.

Section 2: Quaternions

Quaternions are a useful way to find arbitrary rotation matrices between two three-dimensional vectors. The usefulness of this will be explained later, and a more thorough tutorial is given in Appendix A, section 2.

Section 3: GLSL

The OpenGL Shading Language, commonly known as GLSL, was standardized September 7th, 2004 with the release of OpenGL 2.0 [22]. This was OpenGL’s standardized language for high-level programmable shaders. Programmable shaders are used to overload the fixed functionality of the graphics pipeline. These will prove useful in two of the approaches outlined later in this thesis. A more thorough tutorial is given in Appendix A, section 3.

Section 4: Frame Buffer Objects

Frame buffer objects (FBOs) are a standardized OpenGL way to support rendering to off-screen buffers. Instead of rendering to the screen, all the output is written to one or many pre-allocated color buffers that are very flexible. Depth buffers
for each color buffer are optional. When rendering purely two dimensional scenes into a
color buffer a depth buffer is completely unnecessary and can be left out to save on
memory. Fragment shaders have the ability to write different values to multiple color
buffers at the same time. The color buffers must all be same size, but they do not
necessarily have to be the same resolution as the viewing window. For example, a
program running in an 800x600 viewing window can have operations that render to two
separate 400x400 frame buffer objects, one with a depth buffer and one without,
simultaneously, in different ways. FBOs will be used in solutions where data needs to be
gathered from the output of general purpose computation on the graphics hardware.
CHAPTER 4: THE IMPLEMENTATION ENVIRONMENT

Section 1: Introduction

The shadow volume implementation presented in this thesis is fully integrated into the STEAMiE Educational Game Engine [20]. As such, there is some overhead that will affect the results beyond the execution of the shadowing system itself. A stand alone implementation of the shadowing system may perform faster than one integrated into an engine of this size; therefore, all results produced should be compared as relative speeds in relation to the different silhouette edge detection methods rather than by themselves. For instance, a scene measured at 80 frames per second (fps) using the classical approach in this implementation environment may get 90 fps in another implementation environment. The fact that the CUDA approach gets 85 fps as compared to the classical approach’s 80 fps is what is most important.

Section 2: The Basics

This is a practical implementation of a shadowing system incorporated into an existing environment, so some information about that environment should be given. Firstly, the output generated by the environment is a constantly updating scene. Everything in the scene is an object that has an associated model. There are some restrictions of what is considered a valid model. The valid model restrictions are as follows:

- A valid model is strictly a triangular mesh. This means that all faces are triangles – three points, three edges.
• A valid model is a set of surfaces often with only one element. These surfaces do not have to be closed.

• All edges may be shared by at most two faces. However, vertices may be shared by an arbitrary number of edges.

The shadowing implementation in this thesis is only guaranteed to work with valid models, because it was designed with the aforementioned restrictions in mind. Some of the restrictions could be removed and the implementation would function properly with only minor modifications, others however would make for a much more difficult solution.

The impact of the first restriction on the implementation is only for simplicity. No matter how many edges a face has, if each edge only connects two faces, then the surface visibility solution will be the same. The surface visibility algorithms can be simplified if we ensure that all faces have three edges and therefore up to three neighbors.

The second and third restrictions are necessary to prevent degenerate conditions where more than two faces share the same edge. In this situation, the surface visibility problem becomes much more complicated, and this thesis’s solution will not work as implemented.

Section 3: Practical Implications

A few comments should be made about some of the difficulties of working in a practical implementation of standard shadow voluming. Firstly, recall the idea behind shadow volumes; that all surfaces trapped inside the shadow volume should be considered shadowed. An interesting effect can happen when a developer starts to create
scenes that mix shadow casting and non-shadow casting objects. If a non-shadow casting object has a shadow cast upon it, both the near and far surface, with respect to the shadow casting object, will be shadowed. This can look quite odd, but the algorithm is doing exactly what it is supposed to do. Even though the shadow casting object is on the far side of the non-shadow casting object, with respect to the camera, a shadow is still being cast on the camera’s side. This phenomenon does not exist in reality, because in all cases if the non-shadowing object were actually a shadow casting object, then the shadow being cast would be hidden in the larger shadow cast by the formerly non-shadow casting object.
CHAPTER 5: EXPERIMENTAL DESIGN

Section 1: Introduction to the Design Process

This chapter will discuss the implementation details of the shadowing system and several different implementations of the solution to surface visibility with respect to a light source. This section will begin with a description of the data structures and interfaces used in this implementation. That will be followed with a brief description on capping techniques and related issues. The majority of this section will deal with silhouette edge detection and surface visibility with respect to a light source. This will include an explanation of the classical solution to surface visibility with respect to a light source and six different implementations of my solution to surface visibility with respect to a light source. Some time will also be spent discussing how this implementation can easily be extended to support soft shadowing.

Section 2: Data Structures and Implementation Interfaces

Each shadow caster has an array of a structure further known as a shadow neighborhood, corresponding to each of the faces of the shadow caster’s model. The implementation in this thesis is written specifically for meshes consisting only of triangles and the structure reflects that. Because of this, each shadow neighborhood contains the indices into the list of the model’s faces corresponding to its three neighbors. Each shadow neighborhood also contains the indices into the list of the model’s vertices corresponding to its three vertices. Every face of the models in this implementation has consistent winding, in a clockwise fashion; therefore, neighbors can be consistently labeled. The first neighbor is the other face in the model that also contains the faces first
and second vertices. The second neighbor is the face that also contains the second and third vertices, and the third neighbor is the face that also contains the third and first vertices. Each shadow neighborhood also contains the plane equation of the face, in the form $ax + by + cz + d$ where $[a,b,c]$ is the normal of the plane, and $d$ is the distance from the origin to the nearest point on the plane. This distance can be found by taking the dot product of any point on the plane with its normal. The three vertices of the face are all on this plane, so any of them can be used. Each shadow neighborhood also contains storage for a 4x4 float matrix that will be used as the accumulated visibility matrix in my visibility solution presented below. All these values can be pre-computed or computed at load time. The last value, and the only one that changes during runtime is a simple boolean used to keep track of which faces are currently visible. Although various approaches for determining the silhouette edge set will be discussed in the later sections, to explain the shadow volume algorithms it is sufficient to know that the list of silhouette edges will be received as the input and it does not matter how they were determined. In this implementation, this list of silhouette edges is equivalent to all of the members in the array of shadow neighborhood that has its visibility flagged as true. The actual silhouette edges are determined by finding all edges shared between flagged faces and non-flagged faces. Code samples for the shadow neighborhood and the SteamiePlane structure, used to store the plane equation, can be found in Appendix B, sections 1 and 2, respectively.

Because the z-pass and z-fail methods perform operations that require knowledge of whether a wall of the volume is front facing or back facing, winding direction information must be standardized. Fortunately, the faces of the models of this thesis’s
implementation always have consistent winding, in a clockwise fashion. Therefore, if the
walls of the volume are consistently created by extruding the edges into quadrilaterals in
the order presented, from either the strictly front facing or the strictly back facing faces,
without switching back and forth, the desired consistent winding order will be achieved.

The next object this thesis will describe is the shadow casting light. Scenes
created will commonly have lights – some shadow casting, some not. The shadow
casting lights of my implementation have a few additional features. Most importantly
they have the ability to facilitate the casting of shadows of objects. This feature
necessitates the need for them to keep track of which objects they cause to cast shadows.
This means that in this implementation shadow casting is a binary relationship; it requires
both a shadow casting object and a shadow casting light for a shadow to be cast. The
way the shadow casting actually takes place is by telling all the objects in every shadow
casting light’s list to update the stencil buffer, and then the light itself draws the shadow.
We are using a stencil buffer, so the shadow is actually just a single quadrilateral drawn
over the entire screen that only shows up in the stenciled area. OpenGL state has to be
set to use the stencil buffer and then restored to its original state afterwards, so the light
handles this as well.

In order for the correct shadow to be cast for each light, a four step process is
done:

- the visibility for each object is recalculated,
- the OpenGL state is modified to support stencil modification,
• the stencil is modified by each object in the light’s list of shadow casting objects, and
• the shadow quadrilateral is drawn over the stencil and the OpenGL state is restored.

A code sample of how this is accomplished is included in Appendix B, section 3.

The various possible ways of updating the visibility of the object will be discussed later, but suffice it to say that this function flags the faces in the model that are visible to the light. No matter how that is done, the castShadow() function will act on that information.

The beginShadowSetup() function prepares the stencil by modifying the OpenGL state by performing the following steps:

• Clear the stencil buffer bit,
• Push the current attributes onto the attribute stack,
• Enable culling of back faces so the shadow volume algorithms presented in chapter 2 will work properly,
• Disable drawing into the color buffer, and
• Enable stencil testing.

A code sample showing how this is actually accomplished can be found in Appendix B, section 4.

Once the stencil has been updated by each object’s castShadow() function, the actual shadow is drawn, and the OpenGL state is restored with endShadowSetup() by performing the following steps:
- Re-enable drawing into the color buffers,
- Draw a quadrilateral over the entire frame buffer, for this implementation a black shadow with 50% maximum transparency is used, and
- Pop the attribute stack to restore the OpenGL state.

A code sample showing how this is accomplished can be found in Appendix B, section 5.

It should be pointed out that the ‘numSamples’ in the aforementioned code sample will be discussed in more detail in the chapter 5.3, Support for Soft Shadowing. The default value, the value used in the presented case, is 1. This means that the entire shadow will have a transparency of 50%.

The next structure this thesis will describe is the shadow caster interface that objects in the world will implement in order to cast shadows. A shadow casting light keeps a list of all of the objects the user wishes to cast shadows in their scene. Once per scene render cycle, after all of the objects have been rendered into the frame buffer, the light will request that every object in its list update their visibility information and modify the stencil buffer according to their current visibility information. As such, the shadow caster interface implements several shadow related functions, most importantly, the two used above: castShadow() and updateVisibility(). These are the two remaining steps from the shadow generation process described earlier. The castShadow() function is responsible for updating the stencil buffer for the object based on its visibility information. The visibility information is updated in updateVisibility() through one of a
variety of methods described later, such as, the classical method or one of the many
implementations of the second approach described in this thesis.

In this implementation, each object can independently choose to use either the z-
pass or z-fail method of stencil manipulation. Also, this implementation does not bother
capping z-pass volumes because caps would only be needed in situations where the z-
pass method would already fail. In order for correct stencil manipulation to occur, the
quadrilaterals generated by the extrusion of the edges must be transformed through the
display matrix of the shadow casting object. Additionally, the position of the light source
relative to the object must be transformed through the inverse of the display matrix of the
object. It should be noted that the transformed light position is previously calculated in
the objects update visibility function. The extruded edges from this new light position
through the edges of the object will be in the correct world coordinates. Below is the
description of how the stencil manipulation takes place:

A summary of the steps taken in castShadow() are given below:

- Store the current model view matrix,
- Set the current model view matrix to that of the one associated with the
  model of the object we are shadowing,
- Perform either the z-pass or z-fail algorithm,
- Restore the model view matrix, and
- Recursively do this for all child objects.

A code sample showing how this is actually accomplished can be found in
Appendix B, section 6.
The actual stencil manipulation is done inside of the Zfail() and Zpass(). As covered in chapter 2, the z-fail algorithm is a two pass algorithm:

- Increment the stencil when back faces fail the depth test, and
- Decrement the stencil buffer when front faces fail the depth test.

A code sample for these steps is given in Appendix B, section 7.

Similarly, the z-pass algorithm is done by a similar two step process:

- Increment the stencil for front faces that pass the depth test, and
- Decrement the stencil for back faces that pass the depth test.

A code sample for these steps is given in Appendix B, section 8.

The algorithm to actually render the shadow volume in a z-pass situation is fairly straightforward. For every visible face of the model, and for each of its non-visible (or non-existent) neighbors, draw the extruded quadrilateral. The quadrilateral is extruded through the edge shared between those faces through the some distance. Optimally, this distance is infinity, but in some situations, such as trying to prevent a shadow from being cast on the clipping plane, a lesser value may be desirable.

A code sample for the process described above is given in Appendix B, section 9.

The algorithm to render the shadow volume in the z-fail approach is exactly the same, except some steps must be taken in order to properly cap the volume. The capping method used in this thesis’s implementation is not the most robust, but it is fairly simple. The upper cap is a triangle fan with its center at the center of the object, and edges on all the shadow casting edges. The lower cap is a triangle fan with its center on the axis of extrusion at some chosen length, and edges on the projections of all the shadow casting
edges along the axis of extrusion. This capping method works in all situations where the path from the shadow casting edges to the center of the object is entirely within the object. It should be noted that the render order of the triangles of the caps does matter to maintain winding order.

Another capping mechanism used in this implementation involves creating the top cap by using the front facing faces of the object and creating the lower cap by using the extruded back facing faces of the object. This alternative is more computationally and fill-rate expensive than the first option, but it works for all valid models.

A code sample for this approach is given in Appendix B, section 10.

Section 3: Support for Soft Shadowing

This thesis’s shadow volume implementation does include support soft shadowing. Soft shadowing in this thesis’s shadow volume implementation is accomplished using super-sampling of the light source. For instance, if the light source is a volumetric sphere, then it could be approximated with a number of points around its surface and at its center. The same shadowing techniques are applied once per sample, and a separate quadrilateral is drawn for each sample. Wherever the quadrilaterals overlap, their contributions are added so each quadrilateral is lightened. For instance, assuming a maximum opaqueness of 50%, then each sample has an alpha value of .5 divided by the number of samples. Each quadrilateral in a 10 sample situation would have a contribution of 5%, or in a 25 sample situation, each quadrilateral would have a contribution of 2%.
Section 4: Silhouette Edge Detection and Surface Visibility with Respect to a Light Source

Introduction

This section describes the approaches to surface visibility with respect to a light source that are implemented for this thesis. When the actual call to update the visibility information of an object is made, one of the many approaches described here will be performed, but before that, the transformed position of the light must be calculated. For the shadow volume to be calculated properly, either the object must be transformed into the coordinate system of the light source, or the light source must be transformed into the coordinate system of the object. The former would be incredibly expensive to do in software because it requires every vertex of the object to be multiplied through the object’s display matrix. It could be done in hardware much faster, but as the size of the objects grow the number of vertices that need to be transformed increases as well. To transform a light source it takes the same amount of computation no matter how large the object is. This is in contrast to most of the implementations presented in this thesis, where they are designed to handle larger scenes better than smaller scenes. The computation to transform a light source to an object’s coordinate system is simple. One must transform the position of the light source through the inverse of the display matrix of the object. The inverse of the display matrix, since it is guaranteed to be orthogonal and of unit length per column, is its transpose. With the little computation of finding the transpose of the matrix and a software multiply of the position through it the properly transformed light position is generated.
The Classical Approach

An edge is a silhouette edge candidate if it is shared between a face that is visible (front facing) to a light source and a face that is not visible (back facing). If the signed distance from the plane of the face to the light source is greater than the displacement of the plane then that face is visible to the light, otherwise it is not. Therefore it is sufficient to do a single dot product and a comparison to determine the visibility per face of the object. Assuming a standard dot product of three-space vectors takes five floating point operations (flops), barring the use of any specialized SIMD (single instruction multiple data) instructions, then if a shadow casting object has 300,000 polygons it is necessary to do 1.5 megaflops worth of computation.

This thesis’s implementation did include an implementation of the classical approach. The classical approach worked successfully and is very simple. The other approaches are more complicated, but can be faster due to the parallelization opportunities afforded by modern graphics hardware and the highly parallelizable nature of this problem. The implementation below is straight forward. It consists of a loop that performs a dot product between the transformed position of the light and the plane equation of the face. If this value is greater than the displacement then the light source is marked as visible, otherwise it is marked as not visible.

A code sample of this approach is given in Appendix B, section 11.
A New Approach

The Description and a Few Examples

Due to the use of display matrices in hardware accelerated computer graphics, dedicated graphics cards are equipped with a hardware 4x4 matrix multiplier. Therefore, I have designed an implementation of a solution that can be done exclusively with 4x4 matrix multiplications. This process is based on the idea that if a light is visible to an edge, then if that point is transformed through the inverse of the relative transformation matrix of an adjacent face and the light is still visible to the original face, then it is visible to the second face.

An example of this process is given in figures 16 and 17. It can be seen that the scene consists of two faces (1 and 2) intersecting at a point (5), as well as two light sources (3 and 4) colored for convenience. Figure 16 shows the actual scene, with the red and blue lights clearly visible to face 1. It can be seen that while the red light is visible to face 2, the blue light is not. Because faces 1 and 2 meet at point 5, taking the angle between the normals of the two faces, here 45 degrees, gives us the necessary rotation to transform the plane extension of face 2 into the plane extension of face 1. This means that if face 1 is rotated by 45 degrees it will lie in the same plane as face 2, and if face 2 is rotated by -45 degrees it will lie in the same plane as face 1. If the light sources are rotated -45 degrees with respect to the point where the two faces meet (point 5), then the original idea can be applied. Figure 17 shows what happens to the light sources when rotated by -45 degrees with respect to point 5. Now the blue light source is below face 1.
and the red light source remains about face 1. From this it can be inferred that the blue light source is behind face 2 and the red light is in front of face 2.

*Figure 16.* A scene containing two faces and two light sources.

*Figure 17.* The same scene with the light sources transformed as described above.
This process can be applied to larger chains by including translations from one vertex to the next. Figures 18 through 25 will show a longer transformation chain containing five faces, some reference lines, and a single light source. In the original scene, the light source is clearly visible to faces 1, 2, 3, and 5, going from left to right, and clearly not visible to face 4. Figure 18 shows the original scene. The blue reference line is the scene represents the normal of the right hand face at the intersection of the two faces. In situations where the two faces lie in the same plane, it can also be used to distinguish the end point of the faces.

Figure 18. The original scene.
The planes of the first and second faces share a point and the second face is a rotation about that point by 45 degrees clockwise. If we rotate everything in the scene except for the first face by 45 degrees counterclockwise, with respect to the point where the first and second faces meet, this will bring the second face into the same plane as the first face, and all remaining faces maintain the same relative transformation, with respect to the second face. Figure 19 shows the scene under these conditions. It should be noticed that the light source is still visible to the first face, which, returning to the original idea, would imply that it is visible to the second face, which, in the original scene, it clearly is.

*Figure 19. Face 2 rotated 45 degrees counterclockwise.*

The next step is to translate the second face so that its right hand end point is where the first face’s right hand end point is. Alternatively, it can be thought that the second face’s right hand end point must be translated to where its left hand end point is. All of the tested faces start to stack on top of each other, which is exactly what the
relative transformation process is supposed to do. The reference line will be oriented according to the normal of the third face at the point where the second and third faces meet. Figure 20 shows the scene under these conditions. Notice how the light source moves. As the light source gets further and further from the point of rotation, the arc length of the rotation grows and the light source will move further. All of the rotations in this scene will be 45 degrees, but the distance traversed by the light source will grow with each face translation.

Figure 20. Face 2 translated.

The second and third faces again share a point, and the relative transformation from the second to the third face is 45 degrees counterclockwise. If this transformation is inverted and applied to all of the original scene to right of the second face, then the third face will fall in line with the second (and first) faces. This situation is shown in figure 21. It should be noted that the light source is clearly visible to the original face, therefore the original premise would imply that it is visible to the third face, which in the original scene it clearly was.
Figure 21. Face 3 rotated 45 degrees clockwise.

The same process of translations and rotations can continue for the remaining faces. Figure 22 shows the scene after the third face is translated; figure 23 shows the scene after the subsequent rotation to bring the fourth face into alignment. Here it should be noticed that the light source is no longer visible to the first face. According to the original premise, this would imply that the light source is not visible to the fourth face, and in the original scene the light source was not visible to the fourth face.
Figure 22. Face 3 translated.

Figure 23. Face 4 rotated 45 degrees counterclockwise.

Figure 24 shows the scene after the fourth face is translated, and Figure 25 shows the scene after the fifth face is brought into alignment. It can be seen in Figure 25 that the light source is once again visible to the first face, implying that the light source is visible to the fifth face, which it is.
Figure 24. Face 4 translated.

Figure 25. Face 5 rotated 45 degrees clockwise.

This process can be extended to three dimensions by accumulating a 4x4 matrix for each face’s transformation chain. If we can identify the necessary accumulated 4x4 matrix to determine the visibility of a face relative to a specified key face then visibility can be determined through examination of the output position. If the signed distance of the output is positive then it is visible, otherwise it is not. Unfortunately, determining the signed distance itself takes a dot product operation, so at first glance this would appear
unfruitful. However, if the chosen key face is aligned with a primary axis then only one element of the output matrix would have to be examined. If the chosen key face is not axis aligned then an extra matrix multiplication can be performed to align it. The end result means that rather than a dot product, one matrix multiplication may be substituted if the key face is axis aligned, or two matrix multiplications if it is not.

Once a key face has been chosen, the process of determining the accumulated matrix for each face of the model can begin. Every transformation from one face to the next requires one translation matrix and one rotation matrix. If a model is imagined as a graph with faces as nodes, edges as edges, and the necessary transformation matrix as the weights of the edges between two faces, then a slightly modified depth first search can be used to find the necessary transformation matrix for each face relative to the selected key face. One face must be chosen as the key face for this model, and a 4x4 storage matrix should be kept for easy computation of the necessary accumulated transformation matrix per face.

Start with the selected key face as the starting node and the identity matrix as the storage matrix. Proceed with the depth first search as normal and multiply the storage matrix by the matrix stored as the edge weight from the current face to the new face when the search goes deeper. This matrix will be the necessary accumulated transformation matrix for the new face, and the storage matrix for the rest of the search with respect to this node. Obviously, when the search goes shallower to a previous node, the storage matrix is reset to whatever the now saved necessary accumulated transformation matrix is. The second restriction for valid models stated that a model is set of surfaces. This
means that a single model may contain several different surfaces. Returning to our graph analogy, models with multiple disconnected surfaces can be seen as disjoint graphs. In these situations a depth first search will fail to find all the faces of the model and therefore some faces will not have the necessary accumulated transformation matrix with respect to the selected key face. A slight modification must be made in these situations to account for this. In these situations, this implementation determines one key face for every disconnected surface in the model and performs a depth first search for each key face. Every disconnected surface in the model will be referred to as a \textit{key face network}. Every face in a key face network has a accumulated transformation matrix that is relative to that key face network’s key face.

A key face for each network can be easily identified. This thesis’s implementation starts by assuming that each model contains a single key face network and selects the model’s first face to be that key face network’s key face. During the depth first search, all nodes are already being marked as visited or not visited. This information is retained even after a depth first search is deemed finished by returning to the original node after performing searches on all three of its exit paths. At this point the visitation information for the model is examined, and if any node has not been visited then a new key face network is started with that node as its key face. A new depth first search is started at that node. This same process continues until all faces in the model have been visited.

A recursive implementation for transformation matrix identification will now be given; however, it should be noted that ideally objects will contain exactly one key face
network and therefore the theoretical maximum depth of the search is the number of faces in the model. Some models may contain hundreds of thousands of faces, and some environments may have stack issues under recursion of this depth. A user level stack implementation is recommended for use in scenes with continuous models with high polygon counts.

The implementation to find the transformation matrices is separated into three levels. The outermost level controls initialization of the visitation information and whether additional key face networks are needed. The outermost level calls the second level with parameters of the key face for the new key face network and the visitation information.

A code example for the outermost level is given in Appendix B, section 12.

The second level is responsible for assigning the accumulated transformation matrix for the key face and starting the depth first search for the new key face network. The second level is also responsible for saving and loading the current transformation matrix of the search.

A code sample for the middle level is given in Appendix B, section 13.

The last level is given the following: the face that the search came from, the face the search is going to, which of the three points of the first face the search is coming from, and which of the three points in the first face that is shared with the second face that the search is going to. Realistically, either of the two points on the shared edge between the two faces can be used for the last parameter, but in this implementation the first point is used for the first neighbor, the second point for the second neighbor, and the
third point for the third neighbor winding counterclockwise as stored earlier. The function itself does several things. It starts by checking the visitation information and modifying it properly, and then it adds the current face to current key face network. The next step is to create the necessary translation matrix to go from the incoming face’s first point to its second points. This is a fairly simple process of just finding the vector distance between the two points and encoding it into a translation matrix. The next step is to find the necessary rotation matrix to transform the first face so that it lies in the same plane as the second face and invert it. This brings up a point that should be noticed; not all 4x4 matrices have inverses. Thankfully, all orthogonal 4x4 matrices do, and if all of the rows and columns are of unit magnitude then the inverse is equal to the transpose. An orthogonal 4x4 matrix is a matrix in which the dot product of any column with any other column is zero. This matrix is determined using the quaternion approach outlined in chapter 3.2. Now that the inverses of the translation and rotation matrices have been identified, the actual matrix accumulation for the face can begin. The proper accumulation order with $R$ as the rotation matrix from the first face to the second face, $T$ as the translation matrix from the first face to the second, $X$ as the matrix passed in during the search, and $A$ as the accumulated matrix for this face, is $A = R^{-1}T^{-1}X$.

A code sample for the innermost level is given in Appendix B, section 14.

Using this process the necessary accumulated transformation matrix for every face in a model with respect to the key face of their key face network can be determined. Using this information, the surface visibility with respect to a light source can be
determined using only 4x4 matrix multiplications and examination of one element of the result. The next sections will discuss several implementations of this approach.

Encoding Data in a Texture

The fragment shader weighted hardware accelerated approach requires that the data be encoded into a texture so that it can be easily accessed inside of the fragment shader. First, the basic format of the textures should be explained. Texture can basically be thought of as multi-dimensional arrays. Two dimensional textures are used in this implementation. There are two different styles this implementation will embrace: one involves the creation of a texture of height four and base width equal to the number of faces in the key face network, and the other has height sixteen and base width equal to one fourth the number of faces in the model. The term base width is used because some OpenGL implementations only natively supports textures of dimension equal to a power of two. If a texture of a non-power of two dimensionality is attempted to be used, then the texture will either be stretched or compressed to fit in a power of two, effectively corrupting the data. This implementation predetermines the next power of two larger than the number of faces in the model and uses that for the texture width. The extra allocated space will never be touched, but is necessary to prevent data distortion. This implementation will make use of floating point textures, as opposed to the commonly used unsigned byte textures. This implementation uses an RGBA texture; this way one row can be encoded as a single pixel and an entire 4x4 matrix as four pixels. Pixels are stored vertically in groups of four, and each face is stored horizontally in the texture. Consistently storing the four pixels horizontally and each face vertically would have
worked equally as well. For now, the four height approach will be discussed; once a full explanation has been given the sixteen height approach will be shown as a modification. Since the necessary accumulated transformation matrices have already been determined they can now be easily encoded into the two dimensional array. First, handles must be generated for every key face network; then, for every key face network, their information must be encoded into their texture. Once the texture is created it must be loaded onto the graphics card. Once the texture is on the graphics card it may be deleted from the host.

Figure 26 gives a further explanation of how the texture encoding process works. The first element of any row corresponds to the red contribution to the pixel, the second to the green contribution, the third to the blue contribution, and the fourth to the alpha contribution. For simplicity’s sake, the output image ignores alpha blending. Each row is encoded as a pixel in the output texture. Because all values in this example are 1 or 0, all pixels are purely red, green, or blue. It should also be noticed that while the matrix coordinates start in the upper left corner, texture coordinates start in the lower left corner. This means that the final image will need to be flipped vertically to match the actual displayed outputs in the figures x and y below.
A code sample for encoding the matrices into a texture can be found in Appendix B, section 15.

Below is an example of what data encoded into a texture looks like. Figure 27 shows the encoded visibility matrices of a cube. A cube has six faces split into two triangles each. This yields a base width of 12 which must be expanded to 16 for power of two alignment. If examined closely, 12 peaks can be identified horizontally and 4 peaks can be identified vertically. In this example the uppermost vertical peak of every column is black, but the even spacing of rows and column indicates a fourth row should be present. There is no information between the peaks, this is simply interpolation because the texture was rendered onto a frame buffer greater than 4x16 pixels.
It should be noted in this example that all the data peaks appear pure red, green, or blue. This is because all the faces of the cube are at right angles. Figure 28 contains the next example, the torso of the human seen chapter 2. It contains a base width of 82 extended to 128 pixels. The torso is a rounder than the cube so its encoded visibility matrices produce data that has a wider range of colors.
Figure 28. The encoded visibility matrices of the human torso.

A Software Implementation of This Approach

As a proof of concept, a software implementation of this approach was created. It performs exactly as the approach was described, but is performed in software so it is significantly slower than then the classical approach or any hardware approach should be. As stated before, the classical approach requires five floating point operations per face, whereas the naïve approach to 4x4 matrix multiplication takes seven floating point operations per element, with sixteen elements, for a total of 112 floating point operations. Recall that two matrix multiplications are needed for key face networks whose key face is
not aligned with a primary axis. In this implementation it is assumed that all key faces are not axis aligned and all faces will therefore require two matrix multiplications. In this implementation the necessary rotation between the key face and the z-axis in the positive direction is found and used for the additional matrix multiplication. An extension could be added to the algorithm to detect if the normal of the key face is axis aligned, which axis it is aligned with, and then indicate which element of the result should be inspected to determine visibility. This means that the actual computation of this approach is 224 floating point operations per face and the constant time it takes to compute the additional matrix to put the model into alignment. The constant time it takes to compute the additional matrix to put the model into alignment takes approximately 43 floating point operations. This implementation was tested and worked successfully. Timing information for this implementation will be given in chapter 6, section 2.

First, the transformed position of the light source must be encoded into a translation matrix. Then, for every key face network the rotation matrix for that key face network’s key to the positive z-axis is determined. The necessary accumulated transformation visibility matrices have already been determined by this point, so all that is needed now to multiply the matrices together. The correct order for the multiplication of these matrices with $X$ as the resultant matrix, $L$ as the encoded transformed position of the light source, $F$ as the predetermined necessary accumulated transformation visibility matrix, and $R$ as the necessary rotation matrix to bring the key face network’s key face into alignment with the positive z-axis, is $X = LFR$. At this point the result is ready for inspection. Since the key face network’s key face was aligned with the positive z-axis, if
the 14th element is non-negative, then the face is visible with respect to the light source, otherwise it is not.

A code sample for the software implementation of this approach is given in Appendix B, section 16.

*Hardware approach using OpenGL Functionality*

Another approach was to use the OpenGL built in matrix manipulation functions to do the matrix multiplications and retrieve and examine the results. This implementation works, similar to the software implementation, but it has a couple of issues. The idea is that, because OpenGL uses pre-multiplication of matrices, if the key face network rotation matrix is loaded, then pre-multiplied by the per face accumulated transformation matrix, and then pre-multiplied by the encoded translation matrix of the transformed light position, then the 14th element of the result should be all that is needed to determine face visibility with respect to a light source. This is only a slight modification of the software implementation described above.

A code sample for this approach is given in Appendix B, section 17.

This implementation suffers from low bandwidth to the graphics card. The variants of the glGet function are notoriously slow. In the test environment each glGet took approximately 3ns. With each face requiring one glGet per face, this was unacceptable. It will be shown later that this is about the time it takes to compute 300 faces using the classical approach.
Failed Attempt at using the OpenGL Feedback Buffer for This Approach

The OpenGL feedback buffer is an OpenGL supported way to write data into a buffer instead of rendering to the screen. When OpenGL is in feedback mode, instead of render or selection mode, coordinates and attributes of vertices that would have been rendered are written to the buffer instead.

This approach requires that a position be transformed through the aggregation of two 4x4 matrices. The fixed functionality pipeline transforms a vertex through the aggregation of the projection and model view matrices. To obtain the effect this approach requires, the key face rotation matrix can be loaded into the projection matrix and the per face accumulated transformation matrix into the model view matrix. If the transformed light position is passed in as the vertex to be transformed by the pipeline, then the output result could be inspected to determine whether that face is visible or not with respect to the light source.

The algorithm for this approach is very simple. The key face rotation matrix only needs to be loaded into the projection matrix once per key face network. Then the transformed light source position is just repeatedly rendered once per face, with that face’s accumulated transformation matrix being loaded in as the model view matrix. Because the matrix needed to be updated per vertex, the vertices were “rendered” as GL_POINTS.

Unfortunately, there are a couple drawbacks to this approach. Firstly, recall the description of feedback mode [25], it was stated that “coordinates and attributes of vertices that would have been rendered are written to the buffer.” This method does not
guarantee that the output vertices will be inside the viewing frustum, thereby affording
the possibly that some faces may be missed. This means that this implementation will
not work in all situations, and therefore was not further pursued. Memory bandwidth of
sending information to the graphics card is rather limited so attempting to send the 4x4
matrices between each face proved to be very slow as well. The memory bandwidth
issue is the downfall of many of the attempts at hardware approaches. Many
implementations of OpenGL emulate feedback mode in software and the results probably
wouldn’t have been much faster than the software approach described above let alone the
classical approach.

**Hardware Approach Using GLSL Vertex Shader**

The next approach is designed to use GLSL Vertex Shaders to handle the bulk of
the computation. The idea behind this approach was, again, similar to the feedback
approach, to use the model view and projection matrices to store the key face rotation
matrix and the per face transformation matrix and to transform the incoming previously
transformed light position through it. To ensure that the result was actually rendered this
time, the vertex position passed in was to be the actual screen position and would not be
transformed at all. An additional input value the vertex shader has access to is that
vertex’s color attribute. The transformed light position is passed in through the glColor(),
and then it is transformed inside the shader. This implementation can then work,
similarly to the feedback buffer, by loading in the key face rotation matrix once, loading
the new per face rotation matrix for every face, and then repeatedly ‘rendering’ the
transformed light position using GL_POINTS. The position is then multiplied through
the matrices in the shader and written out to a frame buffer object. Assuming the system supports signed floating point frame buffer objects, this implementation will work. Unfortunately, it is slower than other implementations because resending the different per face transformation matrices every face is a relatively slow operation. The fragment shader for this approach is basically fixed functionality. The GLSL vertex shader for this approach is given in Appendix B, section 18, and the GLSL fragment shader for this approach is given in Appendix B, section 19.

It is very simple, but as mentioned previously, unfortunately, it is also slow.

**Hardware Approach Using GLSL Fragment Shader**

The next approach is one of the much more successful approaches. This approach conforms to many of the normal ways of doing things for general purpose GPU computing. The bulk of the computation is done inside of the fragment shader, and the data is accessed through texture previously sent to the graphics card. Again, output is limited to the screen or other frame buffer objects, but that can be read back once computation is complete.

The per face accumulated necessary visibility transformation matrices have already been encoded into a texture for each key face network and sent to the graphics card. A frame buffer object of width equal to exactly that of the texture has also already been created. Only a one dimensional frame buffer texture is needed in this situation, and the rendering view port is set to match the dimensions of the frame buffer object. The model view and projection matrices are set to identity for simplicities sake; this way a quadrilateral can be rendered over the entire frame buffer from (-1,-1) to (1,1). The
texture coordinates go from (0,1) horizontally across the quad. The quadrilateral will be rendered into a frame buffer whose pixel width is exactly that of the texture, so each fragment’s interpolated texture x-coordinate will align exactly with a column in the texture. This means they will have uncorrupted access to one matrix worth of data in each fragment. Once the fragment shader has done its job, the result is read out of the frame buffer using a glReadPixels() call. In this situation, the transformed light position is stored as a uniform in the shader. It only has to be sent once, so this is only a minor amount of overhead. In this implementation, the key face network’s rotation matrix is stored inside the texture matrix. Because the model view, projection, and color matrices are not used for any other purpose, they could have also been used to store this matrix.

The vertex shader is simple since all of the work is done in the fragment shader. All the vertex shader does for this approach is pass on the texture coordinates and vertex positions untransformed. A code example for this is given in Appendix B, section 20.

The fragment shader is more complicated. It starts by grabbing the data from the texture. Because the four channel approach is being used, the actual data peaks are at the y-coordinates 1/8, 3/8, 5/8, and 7/8. With this information, the two matrices can be aggregated and the light position can be transformed through it. Because only the z-component of the result matters, a dot product is performed between the transformed light position and z-column of the aggregated matrix. The result is then written out as one channel of the output color. There are three channels left, so we could extend this shader to work on a sixteen height texture with four matrices encoded per column. The same amount of computation is done with either approach, but in the latter approach it is spread
over fewer threads. This defeats the purpose of creating a highly parallelized implementation. If all of the pixel pipelines are already consumed then this variation will be done in roughly equal time. The sixteen height variation does produce less output, which means a shorter time retrieving the information. For very large scenes, the sixteen height approach is recommended. A code sample for the fragment shader is given in Appendix B, section 21.

The current approach caps before it can reach maximum efficiency. OpenGL FBOs have a resolution cap, which on the test machine was between 9,000 and 10,000 in the x direction. Theoretically, if all of OpenGLs memory could be used for an FBO that only has 1 color channel, then a card with 512MB of dedicated memory should be able to support a 1D frame buffer with a resolution of 128,000,000. Results for testable values and projections for higher face count results are given in chapter 5, section 3, and chapter 6, respectively. Additionally, testing was not done to see if perhaps it was a dimensional limit, rather than a memory restriction. If this were the case, a 9000 x 9000 FBO could hold 81,000,000 results which should be satisfactory. The algorithm would only have to undergo a minor modification to ‘wrap’ along multiple dimensions of the FBO. Recall that shaders can write to multiple color buffers in a FBO, so this too, could possibly be used to augment the size of usable models.

A CUDA Implementation

NVIDIA’s CUDA is a useful C extension to enable developers to write general purpose software to run on the graphics card. Over one hundred pages of documentation can be found on the CUDA development website. This thesis’s CUDA implementation is
divided into four basic steps. The first is a pre-processing step involving allocating the proper space on the graphics cards for all the CUDA operations. The CUDA implementation in this thesis is an implementation of the classical solution. To determine the visibility of one face needs no information about the other faces, so this problem is ideally suited to be parallelized. The primary implementation tested in this paper requires three floating point values for the light position, four floating point values for each face of the model for plane equations, and one floating point value for each face of the model to write the output to. The plane equations of each face are sent to the graphics card during this step and never need to be updated. The remaining three steps are done during each update visibility phase. These three steps include sending the most recent light position, dispatching a call to run the CUDA kernel, and then retrieving the results from the graphics card.

Each thread of execution of the kernel tests the visibility of exactly one face and stores the result to be recalled by the host afterwards. The number of threads is equal to the number of faces of the model. Below is the CUDA kernel used to determine the visibility of all faces. In this example, the float array A contains the plane equations of every face, the float array B is the output array, and the three element float array C contains the transformed position of the light source. The threadIdx.x is a built in CUDA variable that identifies the coordinates in the thread batch. In this example, a one dimensional thread batch is being used so only the x-coordinate is necessary.

```c
__global__ void updateVisibilityKernelOld(float* A, float* B, float* C)
{
}
```
There are a couple variations on this approach that should probably be discussed. Theoretically, there should be a speed up if memory coalescing is taken advantage of so that multiple threads do not have to access the same memory. In this situation, that would be the light source. If the light source were replicated once per face, then each thread could access its own copy. The memory coalesced version would look something like the following:

```c
__global__ void updateVisibilityKernal(float4* A, float* B, float4* C)
{
    float4 neighbourhood = A[threadIdx.x];
    float4 light = C[threadIdx.x];

    B[threadIdx.x] = neighbourhood.x * light.x + neighbourhood.y * light.y + neighbourhood.z * light.z - neighbourhood.w;
}
```

In this version, exactly one element, corresponding exactly to the thread index is accessed. No substantial gains were achieved using this approach, further research may be in order. Unfortunately, for this variation, additional time is needed to replicate copies of the light position and to send the additional information to the graphics card.

CUDA puts a cap of 512 threads per block, because this will make it easier to run on different graphics cards. CUDA supports a grid of blocks, so if the card is capable, then it will run them concurrently. If it is not, then it will run them consecutively. Below is the modified kernel to support models with larger than 512 faces.

```c
__global__ void updateVisibilityKernalOld(float* A, float* B, float* C)
{
    int loc = (blockIdx.x * 512 + threadIdx.x);
    int loc4 = loc * 4;
}
```

Another variation involves the asynchronous nature of using the GPU as a co-processor. CUDA uses Brook Streams [21] to facilitate flow of control. All three of the
stages of the CUDA approach that are done during the update visibility stage can be done asynchronously. This allows for easy load balancing between the GPU and CPU; if the user desired to do a percentage of the faces on the GPU and the remainder on the CPU concurrently that is entirely possible. Unfortunately, asynchronous memory copies can only be done on page-locked memory. For large scenes, where this technique is most suitable, the user may find consumption of this much page-locked memory undesirable.

All in all, the CUDA approach to silhouette edge detection ran significantly faster than the classical approach on the CPU in large scenes. Results will be given in Chapter 6, section 4.
CHAPTER 6: RESULTS

Section 1: The Classical Approach

Figure 29 will present the results of finding the silhouette edge of scenes of various sizes using the classical approach. This should be considered the baseline results. These results were produced running on an Intel Core 2 Duo running at 2.49GHz in high power mode and at 1.25GHz in low power mode. The blue line represents the 2.49 GHz results, whereas the pink line represents the 1.25 GHz results. The additional yellow line is the theoretical maximum on the 2.49 GHz processor assuming that every floating operation takes four cycles.

Figure 29. The classical approach on CPUs of varying speeds.
Section 2: A Second Approach, a Software Implementation

Figure 30 will present the results of finding the silhouetted edge of scenes of various sizes using the alternative approach presented in this thesis implemented through software. This implementation was intended as a proof of concept and therefore the results are not intended to be very impressive; it is just included for completeness. The tests were performed on the same machine and this time the yellow line represents the theoretical maximum at 2.49 GHz using the assumption of 224 floating point operations per face and 43 floating point operations per key face network as described in chapter 5, section 3.3.3.3. The blue line represents the results for the software implementation, whereas the pink line is the results for the classical approach on models of the same size, given for comparison purposes.
Figure 30. Results for the software implementation on the CPU.

Section 3: A Second Approach, a Hardware Implementation Using Fragment Shaders

Figure 31 will present the results of fragment shader approach to the alternative solution presented in this thesis. Similarly to the last approach, this approach should be considered to be superseded by the CUDA implementation in all cases where CUDA is available. Currently CUDA is only supported on NVIDIA cards, so, for instance, on ATI cards the shader approach could be used. ATI also has their own proprietary GPGPU SDK called “Closer To Metal,” however its use was not tested for this thesis. The testing environment for the graphics hardware accelerated approaches used a NVIDIA GeForce 8600M GT.
Figure 31. Results for hardware approach using fragment shaders.

Section 4: A Second Approach, a Hardware Implementation Using CUDA

Figure 32 gives the results for the CUDA implementation to determining visibility with respect to a light source. The dark blue line is the theoretical maximum for my implementation on a NVIDIA GeForce 8600 GT. How this was calculated will be explained in chapter 7, section 4. The yellow line is the classical approach running on the CPU at 2.49 GHz, and the light blue line is the classical approach running on the CPU at 1.25 GHz. The pink line is the results running the CUDA implementation.
Figure 32. Results of CUDA GPGPU implementation.

Because Figure 32 is a little cluttered on the low end, Figure 33 is the same implementation, but only for results on examples with up to 128,000 faces.
Figure 33. Close up view of the results for the CUDA implementation.
CHAPTER 7: CONCLUSIONS AND FUTURE WORK

Section 1: Introduction

This chapter is intended to discuss the results presented in the previous chapter, my interpretations, projections about untested cases, and reasonings about irregularities. This chapter will also present some ideas for future work and research related to the work done in this thesis. For convenience, this chapter has been divided into separate sections dealing with each of the implementations given in the previous chapter, excluding the software implementation of the alternative approach presented in this thesis. Additionally, a summary section will be given at the end of this chapter.

Section 2: The Classical Approach

As mentioned previously, the theoretically maximums used in figures above were based on the assumption that a floating point operation takes four cycles. It takes five floating point operations to determine the visibility of a face. On a 2.5GHz processor, at 100% consumption, the visibility of 125 million faces could be determined in one second. Obviously, there is some system overhead to consider, and some memory access time that may not have been considered in the optimal computation calculations. The computational times of the classical approach compared to the theoretical maximum seem reasonable.

As expected, the time to determine the visibility of a number of faces with respect to a light source grows linearly with the number of faces in this approach. Similarly, as expected, the processor running at 1.25GHz took roughly twice as long to compute the same task as the processor running at 2.49GHz. As mentioned previously, these numbers
should be considered the baseline results for this sort of problem and seem like they would be hard to beat.

Section 3: An Alternative Approach Using Fragment Shaders

This implementation was a little disappointing. Given the results in Figure 12, it might be assumed that this approach would have easily dominated the performance of the CPU classical approach implementation. Examining Figures 12 and 31, there is clearly some overhead in doing this sort of procedure. However, in Figure 31 there seems to be no gain to make up for the overhead. Also in contrast to figure 12, this implementation only seems to be getting around 160,000 matrix multiplication per millisecond, as opposed to over 1,000,000.

If the performance of Figure 12 could be achieved, this approach would surpass the classical approach in scenes of around 100,000 faces. However, because of the prospects of CUDA, optimizations to this approach were not researched, and the remaining effort was applied to the CUDA implementation. It should be pointed out that Figure 12 was from a three year old document and modern graphics hardware has developed at such a pace that this could probably be done even faster.

Assuming one knows the number of matrix multiplications per second a specific graphics card can do, then the tipping point where the fragment shader approach becomes more efficient than the classical approach on a single threaded implementation is fairly simple to find.
To determine whether it is more efficient to use the fragment shader approach or the classical approach, compare the values generated by equations 11 and 12. Whichever of these two values is lower is the more efficient approach.

In Equation 1, \( F \) is the number of shadow casting faces in the scene, \( f \) is the frequency of the CPU and \( c \) is the number of cycles the CPU takes to do a floating point operation, commonly 4.

\[
5cF / f \quad (1)
\]

In Equation 2, \( F \) is still the number of shadow casting faces in the scene, \( O \) is the overhead for setting up the GPU implementation, and \( m \) is the number of matrix multiplications per second that the GPU can handle.

\[
O + 2F / m \quad (2)
\]

If using the alternative approach presented in this thesis could be matched with a fragment shader implementation that equaled or bettered the results presented in Figure 12, then that would be a valuable cross-platform hardware implementation of this solution.

Section 4: An Alternative Approach Using CUDA

As opposed to the fragment shader approach, the CUDA approach was very successful. This implementation was done in a test environment with a NVIDIA
GeForce 8600 graphics card containing 32 stream processing units (SPU) running at 1.18 GHz each. The CUDA programming guide states that floating point additions and multiplications take 4 cycles to complete. This would mean that with thread saturation 9.44 billion floating point operations can be done per second. This would interpret to all the floating point computation for 1.888 billion faces visibility determination being done in one second. The maximum theoretical limit lines in Figures 32 and 33 are based on these calculations.

The CUDA implementation does significantly under perform from this theoretical lower bound and there a couple reasons for this. There is the overhead of sending and receiving. The sending overhead is constant for the main implementation, because exactly three floats are sent every time. The retrieving overhead grows with the number of faces linearly. Upon examination of Figures 32 and 33, it can be seen that the CUDA implementation is actually diverging from the theoretical limit rather than approaching it. Another reason for this is because memory access time is not taken into account for this theoretical limit, and this implementation does have a rather low mathematical operation to memory access ratio.

Examining Figures 32 and 33, it can be seen that the CUDA implementation presented in this thesis overtakes the efficiency of the classical approach running at 2.5GHz at around 82,000 faces. On a 1.25GHz processor, the CUDA implementation will overtake it at around 25,000 faces. One of the machines that the STEAMiE Educational Game Engine [20] has been tested on runs at 800 MHz. On that machine
this approach would surpass the classical approach run on the CPU at around 20,000 faces.

These results were generated in an environment using an NVIDIA GeForce 8600. The NVIDIA GeForce 8800 has over four times the processing power of the 8600 [23]. As compared to the 8600’s 32 SPUs running at 1.18GHz, the 8800 has 128 SPUs running at 1.35 GHz. The 8800 also has twice as many texture access units and three times as many rasterization operator units as the 8600. For additional perspective, the newest NVIDIA card, the GeForce 9800 GTX OCX [24], has 128 SPUs running at 1.89 GHz each.

The examples above had an approximate overhead of 430ns to send the lighting information and dispatch the CUDA kernel. Using this, an equation can be established to approximate the tipping point for the number of shadow casting faces in a scene before CUDA becomes useful. This is an approximation, because as seen in the results above, neither the classical approach nor the CUDA approach the theoretical limit for the processing power of their environments. To obtain a good estimate of which technique is appropriate compare the values produced by equations 13, 14, and 15. Equation 13 represents the theoretical maximum for the classical approach run on a single core (as it is done in this implementation). Equation 14 represents the theoretical maximum for the classical approach run on a multi-core environment. Equation 15 represents the theoretically maximum number of computed of faces using CUDA, plus the offset mentioned above.
In equation 3, \( ns \) represents the theoretical time in nanoseconds to compute \( F \) faces. \( f \) is the frequency of the CPU, and \( c \) is the number of cycles the CPU takes to perform a floating point operation, commonly 4.

\[
ns = \frac{5cF}{f} \tag{3}
\]

Equation 4 is the same as equation 13 except that the classical approach is being computed in parallel threads on \( t \) separate processors, and \( f \) is now the speed of each processor.

\[
ns = \frac{5cF}{ft} \tag{4}
\]

Equation 5 is slightly more complicated. In this equation \( c,F,f \), are the same as above. \( s \) is the number of SPUs on the graphics; \( b \) is the memory bandwidth from the graphics card to the host. On cards with a higher bandwidth from the host to the graphics card the constant overhead may, in fact, be lower.

\[
ns = 430 + \frac{5cF}{fs} + \frac{32f}{b} \tag{5}
\]

It is important to note that there is very little slowdown of the CUDA implementation by running it on the same graphics hardware but a slower CPU. There is no slowdown at all in the CUDA kernel itself, but there is some in the transfer of the light
position to the graphics card, the results back from the card, and dispatching the kernel. Because of this, techniques that utilize the power of modern graphics hardware can easily be put onto machines with low system resources by simply upgrading their graphics cards.

A few areas of the CUDA implementation could still use some investigation. Recall the discussion about memory coalescing; this is necessary for scene aggregation so that the entire CUDA batch can be processed all at once. One possibility is passing a series of light positions and number of necessary repeats, and dispatching a kernel to copy that information into shared memory on the graphics card. This minimizes the amount of information that has to be created and sent from the host, but does tie up more GPU time operating on the first kernel.

The load balancing aspect of using the GPU as a co-processor could also use further research. The asynchronous tests performed during investigations for this thesis were inconclusive.

This CUDA implementation only makes use of one graphics card. A more robust implementation that auto-detects and optimizes for multiple graphics card will also be useful.

There are a few other optimizations that can be made to the CUDA implementation that do not preclude memory coalescing but will incur fewer memory reads. CUDA can read in 128-bits at a time, which means that 1 float4 can be read in one call. The CUDA routine mentioned earlier could be altered if the structures were set up so that four planes were read in from separate arrays and all computed with respect to the
same light position. This reduces the overall memory reads by $3/8$, but may or may not give a significant speed increase. With large models, all parallelization capability is likely saturated so increasing the workload per thread while reducing the total number of threads is satisfactory.

The largest aspect that can be improved upon is elimination of the retrieval time. If this could somehow be eliminated this would be a significant speedup. This would entail writing the output to some place on the graphics card that could be accessed during the render portion of the shadow voluming algorithm. CUDA does support some OpenGL interoperability and there are a couple of ways that this might work.

First, if the system was designed to render every edge’s extruded quad, then perhaps some sort of vertex attribute array could be used to determine which quadrilaterals should be rendered and which ones should not inside of the vertex shader. This would allow CUDA to write to the same vertex attribute array on the graphics card and not retrieve the information every update.

Alternatively, if the output information was encoded properly, it could be stored as a texture on the graphics card and could be used to determine which quadrilaterals should have all of their fragments discarded. This too would eliminate the necessity of retrieving the information from the graphics card.

If either of these approaches took less time than the retrieval portion of the samples above, then doing so would be beneficial. There may also be other approaches that would work equally as well or better.
Section 5: Summary

As demonstrated above, hardware solutions to silhouette identification are not only viable, but preferable in sufficiently large scenes. Coming full circle, back to the results listed in Figure 1, it should be obvious that as time progresses and development continues, the scene sizes at which GPU based silhouette identification surpasses CPU based silhouette identification will continue to decrease.

The new, alternative approach to silhouette edge detection presented in thesis, as well as, the proof-of-concept CUDA implementation of silhouette edge identification should contribute significant justification that purely GPU driven silhouette edge detection techniques should be pursued.

Additionally, the results garnered by this thesis’s comparisons of various silhouette edge identification techniques should contribute satisfactory justification that the approaches presented in this thesis are viable alternatives to current standard practice.
REFERENCES


www.cs.utsa.edu/~qitian/seminar/Spring05/03_04_05/VectorComputingonGPUs.pdf, as of August 10, 2008.


APPENDIX A: TECHNICAL FUNDAMENTALS AND TUTORIALS

Section 1: Rotation Matrices

First, it should be noted that OpenGL stores its display matrices in column major order rather than row major order (standard C++ style) as a sixteen element float array. OpenGL extensions do support what they call “transpose matrices” which are row major matrices, but for compatibilities sake this implementation will use the standard column major type. For continuity’s sake with the presented implementation, indices into a sixteen element array will be referred to in column major order, as seen below in equation 6.

\[
\begin{pmatrix}
0 & 4 & 8 & 12 \\
1 & 5 & 9 & 13 \\
2 & 6 & 10 & 14 \\
3 & 7 & 11 & 15
\end{pmatrix}
\]

Using homogeneous coordinates a point can be represented as \([x,y,z,1]^T\).

Obviously, this can be post-multiplied with a 4x4 matrix, with the result \([x’,y’,z’,1]^T\), another point. The simplest case is using the 4x4 identity matrix; with an identity matrix, the input is equivalent to the output, as seen in equation 7.

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z \\
1
\end{pmatrix}
= 
\begin{pmatrix}
x \\
y \\
z \\
1
\end{pmatrix}
\]
The next example will be a translation. If a point is to be translated by a given vector \([x, y, z]\), then set the 12\(^{th}\) element to \(x\), 13\(^{th}\) to \(y\), and 14\(^{th}\) to \(z\). Equation 8 is an example matrix for the operation of translating a point \([a, b, c, 1]^T\) by the vector \([x, y, z]\).

\[
\begin{bmatrix}
1 & 0 & 0 & x \\
0 & 1 & 0 & y \\
0 & 0 & 1 & z \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
a \\
b \\
c \\
1
\end{bmatrix} =
\begin{bmatrix}
a + x \\
b + y \\
c + z \\
1
\end{bmatrix}
\] (8)

The next example will be scaling. Assuming that a point’s position is its distance from a local origin then a collection of points around the same local origin can be scaled by multiplying this distance by some constant factor. For instance, a set of points representing a cube, could be used to represent any axis-aligned rectangular prism through the use of the proper 4x4 scaling matrix. Equation 9 is an example of the matrix used to scale the point \([a, b, c, 1]^T\) by two along the x-axis, three along the y-axis, and one half along the z-axis is given next.

\[
\begin{bmatrix}
2 & 0 & 0 & 0 \\
0 & 3 & 0 & 0 \\
0 & 0 & .5 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
a \\
b \\
c \\
1
\end{bmatrix} =
\begin{bmatrix}
2a \\
3b \\
.5c \\
1
\end{bmatrix}
\] (9)

The next several examples are about rotations. Rotations about the relative x, y, or z-axis, while more complicated than translation or scaling, are still a relatively simple procedure. A rotation about the x-axis by \(a\) degrees (equation 10), about the y-axis by \(b\) degrees (equation 11), and about the z-axis by \(c\) degrees (equation 12) is given next.
Rotations about non-primitive axes are also possible using 4x4 rotation matrices; however, they are not as easy to calculate as the others. These will be discussed in more detail in the next section where this thesis will show how to generate the necessary rotation matrix using quaternions.

Another reason why 4x4 display matrices work so well is that they can be easily accumulated. A 4x4 matrix multiplied by another 4x4 matrix yields a 4x4 matrix, so operations can be easily accumulated into one 4x4 matrix by multiplying all of the operation’s matrices together. However, recall that matrix multiplication is not commutative. Therefore, it is important to note whether subsequent operations are pre-multiplied or post-multiplied. OpenGL uses pre-multiplication for its matrix accumulation operations, so the operation below (equation 8) would yield the following accumulated matrix.

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & \cos(a) & \sin(a) & 0 \\
0 & -\sin(a) & \cos(a) & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} \tag{10}
\]

\[
\begin{pmatrix}
\cos(b) & 0 & -\sin(b) & 0 \\
0 & 1 & 0 & 0 \\
\sin(b) & 0 & \cos(b) & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} \tag{11}
\]

\[
\begin{pmatrix}
\cos(c) & \sin(c) & 0 & 0 \\
-\sin(c) & \cos(c) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} \tag{12}
\]
The ability to accumulate matrices in this way leads to my solution to surface visibility with respect to a light source. Details on this will be discussed further in chapter 5.3.2, A New Approach.

Section 2: Quaternions

Without giving an exhaustive definition of what quaternions are and how they work, suffice it to say that they are useful as a way of representing three dimensional rotations. Quaternions are usually represented as a four dimensional vector such as \([a, b, c, d]\). A description will be given of how to determine the quaternion representation of a rotation from source vector \([x, y, z]\) to destination vector \([a, b, c]\), followed by a code sample of this thesis’s implementation.

First, the vector normal to the plane that both the source and destination lie in must be determined. This is accomplished by taking the cross-product of the source and the destination. The normalized unit vector in the direction of this cross-product will be referred to as \([v_x, v_y, v_z]\). Then the angle between the source and the destination in that plane must also be found. This is accomplished by taking the dot-product of the source and the destination. The quaternion will use the opposite of half the value of the

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & -1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
2 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & .5 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
1 & 0 & 0 & 1.5 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 4 \\
0 & 0 & 0 & 1
\end{bmatrix} = \begin{bmatrix}
2 & 0 & 0 & 3.0 \\
0 & 0 & 0.5 & 2.0 \\
0 & -1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

(13)
arccosine of that angle frequently, so this will be referred to as $\theta$. With all this information the quaternion representation would be equation 9.

$$[\cos(\theta), vx \cdot \sin(\theta), vy \cdot \sin(\theta), vz \cdot \sin(\theta)]$$

The conversion of a quaternion to a 4x4 rotation matrix will follow; however, note that this is a purely rotational matrix so the translation column and perspective row are identity. For a quaternion $[a,b,c,d]$ the related rotation matrix is equation 10.

$$\begin{pmatrix}
    a^2 + b^2 - c^2 - d^2 & 2ad + 2bc & 2bd - 2ac & 0 \\
    2bc - 2ad & a^2 - b^2 + e^2 - d^2 & 2ab + 2cd & 0 \\
    2ac + 2bd & 2cd - 2ab & a^2 - b^2 - e^2 + d^2 & 0 \\
    0 & 0 & 0 & 1
\end{pmatrix}$$

This is used to determine the necessary rotation matrix to align an object key-face with a primary axis. The importance of this will be discussed later in chapter 5.3.2, A New Approach.

Section 3: GLSL

Programmable shaders are a way to overload the fixed functionality of the graphics pipeline to perform various functions. This new ability allows the massive hardware capabilities of the graphical unit to be more easily exploited. There are however a few restrictions on what can be done inside of the shaders, as well as, how input and output have to be handled. There are a few aspects that should probably be explained to better understand some of the discussions of a couple of the implementations of the solution to surface visibility with respect to a light source.
In GLSL, shaders are divided into two different categories: vertex shaders and fragment shaders. The purpose of these two different types of shaders and what information each of the shaders has access to will be discussed in the next two paragraphs. The idea behind the vertex-fragment paradigm is that vertex shaders will produce outputs that will be linearly interpolated between each vertex for every fragment lying between. These outputs are referred to as *varying* outputs.

Vertex shaders have access to each of the *attribute* variables set in the OpenGL state per vertex, as well as any additional attributes specified in the shader itself, and *uniform* variables specified in the shader as well. A vertex shader’s most important task is to modify the position of vertices in the scene. Fixed functionality multiplies the given vertex position through the accumulated model view and projection matrices to determine the actual screen coordinates of the vertex. By using a vertex shader, this functionality is overloaded and can be altered to do many different behaviors. For instance, the new functionality could translate every vertex given by two units in the x direction. Techniques that change the position of vertices in the vertex shader are often referred to as displacement shading. Another form of displacement shading could be to randomly perturb each vertex of an object, so if a cube is passed in, something slightly more jagged could come out. Other times, the vertex information could simply be not transformed at all. This mimics the appearance of identity model view and projection matrices, so that the actual information stored in the matrices can be used for other purposes.

In addition to controlling the position per vertex, the vertex shader produces as an output the color of that position that will be linearly interpolated to all adjacent vertices.
This means that the vertex shader, in fixed functionality, is also responsible for doing lighting calculations that affect the color of its face. One commonly used shader approach is to move this responsibility to the fragment shader so that lighting can be done on a per-pixel basis for smoother results.

Another interpolated output of the vertex shader is the texture coordinates. These will be used heavily in this thesis’s implementations of a solution for surface visibility with respect to a light source.

Fragment shaders control operations on a per-pixel level. They have access to their interpolated values of the varying outputs of their face’s vertices, as well as access to all the uniform variables specified in the shader, including textures. Given the interpolated texture coordinates, the information inside of a given texture can be accessed. This is typically used for color modification, but any data can actually be encoded in a texture to be used. This is another aspect heavily used in some of the implementations of this thesis’s solution to surface visibility with respect to a light source. Unfortunately, the only possible output from fragment shaders is the colored pixel that is drawn into the frame buffer. Shaders are meant to render things and that is all they can do. This can be troublesome at times, because the desired data may not be well suited for rendering. However, in the next section we will discuss how to render to off screen buffers that can be read. This allows any data to be generated, stored, and recovered, but is not as clean, easy, or efficient as a more general purpose solution would be. Applications where general purpose data is desirable may be better suited to use CUDA.
class ShadowNeighbourhood
{
    public:
    /**
     * Default constructor. Assigns dispMatrix to NULL.
     */
    ShadowNeighbourhood();
    /** The plane equation of this neighbourhod object */
    SteamiePlane plane;
    /**
     * The three neighbours of this neighbourhod object.
     */
    Neighbours[0] is the member of the neighbourhod that shares the
    edge created by points[0] and points[1].

    Neighbours[1] is the member of the neighbourhod that shares the
    edge created by points[1] and points[2].

    Neighbours[2] is the member of the neighbourhod that shares the
    edge created by points[2] and points[0].
    */
    int neighbours[3];
    /**
     * The three points of this face. Winds in the CCW direction.
     */
    int points[3];
    /**
     * The display matrix need to align this face with the keyface of
     * its shadow network.
     */
    float* dispMatrix;
    /**
     * The only member that changes post initialization.
     */
    bool visible;
};
Section 2: The SteamiePlane Class

class SteamiePlane
{
  public:
  /**
   Default constructor
   */
  SteamiePlane() {}  
  /**
   Constructor taking each element
   */
  SteamiePlane(float x, float y, float z, float d);
  /**
   Default destructor
   */
  ~SteamiePlane() {}
  /**
   Returns a Vector that is the normal of the plane
   */
  Vector getNormal() { return Vector(x, y, z); }
  /**
   Returns the displacement of the plane
   */
  float getDisplacement() { return d; }
  private:
  float x;
  float y;
  float z;
  float d;
};

Section 3: Casting Shadows

void WOLightShadowCasting::castShadows()
{
  Vector temp = this->getPosition();

  //update visibility for all objects
  for(size_t I = 0; I < this->shadowList.size(); i++)
    shadowList[i]->updateVisibility(temp);

  //adjust OpenGL state
  beginShadowSetup();
  for(size_t I = 0; I < this->shadowList.size(); i++)
    shadowList[i]->castShadow();
  //restore OpenGL state, and draw shadow quad
  endShadowSetup();
}
Section 4: Begin Shadow Setup

```cpp
void WOLightShadowCasting::beginShadowSetup()
{
    glCullFace(GL_BACK);
    glPushAttrib(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_ENABLE_BIT | GL_POLYGON_BIT | GL_STENCIL_BUFFER_BIT);
    glClearStencil(0);
    glClear(GL_STENCIL_BUFFER_BIT);
    glEnable(GL_CULL_FACE);
    glDepthMask(GL_FALSE); // Turn Off Writing To The Depth-Buffer
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_STENCIL_TEST); // Turn On Stencil Buffer Testing
    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE); /* Don’t Draw Into The Colour Buffer */
}
```

Section 5: End Shadow Setup

```cpp
void WOLightShadowCasting::endShadowSetup(unsigned int numSamples)
{
    glFrontFace(GL_CCW);
    glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE); /* Enable Rendering To Colour Buffer For All Component*/
    glColor4f(0, 0, 0, .5f / numSamples);
    glStencilFunc(GL_NOTEQUAL, 0, 0xFFFFFFFFL);
    glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();
    glDisable(GL_CULL_FACE);
    glBegin(GL_QUADS);
    glVertex3f(-1, -1, 0);
    glVertex3f(-1, 1, 0);
    glVertex3f(1, 1, 0);
    glVertex3f(1, -1, 0);
    glEnd();
    glPopMatrix();
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
    glPopAttrib();
}
```
Section 6: Casting Shadows

```cpp
void IfaceShadowCaster::castShadow()
{
    float f[16];
    getMyModel()->getDisplayMatrix(f);

    glPushMatrix();//preserve the state
    glLoadIdentity();
    getMyModel()->getCurrentPosition().x,
    getMyModel()->getCurrentPosition().y,
    getMyModel()->getCurrentPosition().z)//translate
    glMultMatrixf(f);//rotate

    if(zfail)
        Zfail(transformedPosition);
    else
        Zpass(transformedPosition);

    glPopMatrix();//restore state

    for(size_t I = 0; I < shadowChildren.size(); i++)
        shadowChildren[i]->castShadow();
}
```

Section 7: Z-Fail

```cpp
void IfaceShadowCaster::Zfail(Vector& pos)
{
    glFrontFace(GL_CW);//back faces
    glStencilOp( GL_KEEP, GL_INCR, GL_KEEP );//increment on fail
    shadowFail(pos);

    glFrontFace(GL_CCW);//front faces
    glStencilOp( GL_KEEP, GL_DECR, GL_KEEP );//decrement on fail
    shadowFail(pos);
}
```

Section 8: Z Pass

```cpp
void IfaceShadowCaster::Zpass(Vector& pos)
{
    glFrontFace( GL_CCW );//front faces
    glStencilOp( GL_KEEP, GL_KEEP, GL_INCR );//increment on pass
    shadowPass(pos);

    glFrontFace( GL_CW );//back faces
    glStencilOp( GL_KEEP, GL_KEEP, GL_DECR );//decrement on pass
    shadowPass(pos);
}
```
void IfaceShadowCaster::shadowPass(Steamie::Vector &pos)
{
  for(size_t j = 0; j < getNeighbourhood()->size(); j++)
  {
    if(getNeighbourhood()->at(j).visible)
    {
      for(int k = 0; k < 3; k++)
      {
          /* if this face has no neighbour on that edge, or that neighbour
             is not visible */
          {
            Vector p1 = *getMyModel()->getCompositeVertexList()->at( getNeighbourhood()->at(j).points[k] );
            Vector p2 = *getMyModel()->getCompositeVertexList()->at( getNeighbourhood()->at(j).points[(k+1)%3] );
            Vector p3 = (p1 - pos) * SHADOW_EXTRUSION;
            Vector p4 = (p2 - pos) * SHADOW_EXTRUSION;

            glBegin(GL_TRIANGLE_STRIP);
            glVertex3f(p1.x, p1.y, p1.z);
            glVertex3f(p3.x, p3.y, p3.z);
            glVertex3f(p2.x, p2.y, p2.z);
            glVertex3f(p4.x, p4.y, p4.z);
            glEnd();
          }
      }
    }
  }
}
void IfaceShadowCaster::shadowFail(Vector& pos) {
    std::vector<Vector> capTop;
    std::vector<Vector> capBottom;

    Vector centerBottom = (getMyModel()->getFixedPoint() - pos);
    centerBottom.normalize();
    centerBottom = centerBottom * SHADOW_EXTRUSION;
    Vector centerTop = getMyModel()->getFixedPoint();

    for (size_t j = 0; j < getNeighbourhood()->size(); j++)
    {
        if (getNeighbourhood()->at(j).visible)
        {
            for (int k = 0; k < 3; k++)
            {
                if (getNeighbourhood()->at(j).neighbours[k] == -1 ||
                    !getNeighbourhood()->at(j).neighbours[k]).visible)
                {
                    Vector p1 = *getMyModel()->getCompositeVertexList()->at(getNeighbourhood()->at(j).points[k]);
                    Vector p2 = *getMyModel()->getCompositeVertexList()->at(getNeighbourhood()->at(j).points[(k+1)%3]);

                    Vector p3 = p1 - pos;
                    p3.normalize();
                    p3 = p3 * SHADOW_EXTRUSION;
                    p3 = p3 + p1;

                    Vector p4 = p2 - pos;
                    p4.normalize();
                    p4 = p4 * SHADOW_EXTRUSION;
                    p4 = p4 + p2;

                    capTop.push_back(p1);
                    capTop.push_back(p2);

                    capBottom.push_back(p3);
                    capBottom.push_back(p4);

                    glBegin(GL_TRIANGLE_STRIP);
                    glVertex3f(p1.x, p1.y, p1.z);
                    glVertex3f(p3.x, p3.y, p3.z);
                    glVertex3f(p2.x, p2.y, p2.z);
                    glVertex3f(p4.x, p4.y, p4.z);
                    glEnd();
                }
            }
        }
    }
}
Section 11: The Classical Approach

```cpp
void IfaceShadowCaster::updateVisibilityTheirWay()
{
    for(size_t j = 0; j < this->neighbourhood.size(); j++)
    {
        float d = transformedPosition.dotProduct(this-
            >neighbourhood[j].plane.getNormal());
        if(d > this->neighbourhood[j].plane.getDisplacement())
            this->neighbourhood[j].visible = true;
        else
            this->neighbourhood[j].visible = false;
    }
}
```
void IfaceShadowCaster::findTransformationMatrices()
{
    if(neighbourhood.size() == 0)
        return; // for empty models skip this step

    bool* b = new bool[neighbourhood.size()];
    for(size_t I = 0; I < neighbourhood.size(); i++)
        b[i] = false; // initialize visitation information

    int first = 0;
    int counter = 0;
    do
    {
        counter = 0;
        findTransformationMatrices(first, b);
        for(size_t I = 0; I < neighbourhood.size(); i++)
        {
            if(!b[i])
            {
                ++counter;
                first = (int) I;
                break;
            }
        }
    } while(counter > 0);
}
void IfaceShadowCaster::findTransformationMatrices(int x, bool* b)
{
    this->keyFaces.push_back(x);
    keyFaceNetworks.push_back(std::vector<int>());
    keyFaceNetworks[keyFaces.size() - 1].push_back(x);
    b[x] = true;

    float* ptr = new float[16];
    for(int I = 0; I < 16; i++)
        ptr[i] = 0; // initialize

    neighbourhood[x].dispMatrix = ptr; // assign identity to key face
    s.push(ptr);

    for(int I = 0; I < 3; i++)
    {
        if(neighbourhood[x].neighbours[i] != -1)
            findTransformationMatrices(x,neighbourhood[x].neighbours[i],0,I,b);
    }
    s.pop();
}
void IfaceShadowCaster::findTransformationMatrices(int x, int y, int f, int s, bool* b)
{
    if(b[y]) // if already visited
        return; // break
    b[y] = true; // keep from infinitely looping
    keyFaceNetworks[keyFaces.size() - 1].push_back(y); // add to network

    // find translation from last place (t)
    float t[16];
    for(int I = 0; I < 16; i++)
        t[i] = 0; // initialize
    /* get the vector translation vector between the entrance and exit points */
    Vector v = *this->getMyModel()->getCompositeVertexList()->at(neighbourhood[x].points[f]) - *this->getMyModel()->getCompositeVertexList()->at(neighbourhood[y].points[s]);
    t[3] = -v.x;
    t[7] = -v.y;
    t[11] = -v.z;

    // find rotation from last place I
    float r[16];
    determineRotationMatrix(neighbourhood[x].plane.getNormal(),
                            neighbourhood[y].plane.getNormal(), r);
    calculateInverseOfDisplayMatrix(r, r);
    float* first = new float[16];
    float* old = this->s.top();
    for(int I = 0; I < 16; i++)
        first[i] = old[i];

    multiply4x4Matrix(t, first); // t = t * first
    multiply4x4Matrix(r, t); // r = r * t

    for(int I = 0; I < 16; i++)
        first[i] = r[i];
    neighbourhood[y].dispMatrix = first;

    this->s.push(first);
    for(int I = 0; I < 3; i++) // map all neighbours
    {
        if(neighbourhood[y].neighbours[i] != -1)
            findTransformationMatrices(y, neighbourhood[y].neighbours[i], s, I, b);
    }
    this->s.pop(); // pop stack back
    return;
}
void IFaceShadowCaster::createTexture()
{
    if(this->neighbourhood.size() == 0)
        return;//if empty model skip

    /* allocate room for enough handles for the number of key face networks */
    textureHandle = new GLuint[keyFaceNetworks.size()];
    //genreate handles for the number of key face networks
    glGenTextures((GLuint) keyFaceNetworks.size(), textureHandle);

    glEnable(GL_TEXTURE_2D);

    //for every key face network
    for(size_t t = 0; t < keyFaceNetworks.size(); t++)
    {
        //determine width from base width
        unsigned int size = nextPowerOf2((unsigned int) this->keyFaceNetworks[t].size());
        //allocate room for 16 floats per pixel-width
        GLfloat* texture = new GLfloat[size * 16];

        for(int k = 0; k < 4; k++)//4 pixels high
            for(int j = 0; j < 4; j++)//4 channels per pixel
            {
                //if in the real information section
                if(l < this->keyFaceNetworks[t].size())
                    texture[k * 4 * size + l * 4 + j] = this->neighbourhood[keyFaceNetworks[t][l]].dispMatrix[k * 4 + j];
                else//in the padding section
                    texture[k * 4 * size + l * 4 + j] = 0.0f;
            }

        //OpenGL setup
        glBindTexture(GL_TEXTURE_2D, textureHandle[t]);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F_ARB, size, 4, 0, GL_RGBA, GL_FLOAT, texture);
        glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
        delete[] texture;}//loaded to graphics memory, no longer needed
}
Section 16: A Software Implementation

```cpp
void IFaceShadowCaster::updateVisibilityMyWaySoftware()
{
    Vector test;
    float r[16];
    float t[16];
    float u[16];

    // for every key face network
    for(size_t j = 0; j < keyFaces.size(); j++)
    {
        /* determine necessary rotation matrix to align the model with the
         positive z-axis */
        determineRotationMatrix(this->getNeighbourhood() -> at( 
            keyFaces[j]).plane.getNormal(), Vector(0,0,1), r);

        // for every face in the key face network
        for(size_t i = 0; i < keyFaceNetworks[j].size(); i++)
        {
            // encode transformed light position into translation matrix
            for(int k = 0; k < 16; k++)
                u[k] = 0;
            u[12] = transformedPosition.x;
            u[13] = transformedPosition.y;
            u[14] = transformedPosition.z;

            for(int k = 0; k < 16; k++)
                t[k] = this->getNeighbourhood() -> at( 
                    keyFaceNetworks[j][i]).dispMatrix[k];

            multiply4x4Matrix(t, r);
            multiply4x4Matrix(u, t);
            if(u[14] > MIN_DISTANCE)
                this->getNeighbourhood() -> at(keyFaceNetworks[j][i]).visible =
                    true;
            else
                this->getNeighbourhood() -> at(keyFaceNetworks[j][i]).visible =
                    false;
        }
    }
}
```
Section 17: The First Hardware Attempt

void IfaceShadowCaster::updateVisibilityMyWayHardware1()
{
    float s[16];
    float r[16];
    float u[16];
    for(int I = 0; I < 16; i++)
    {
        u[i] = 0;
        u[12] = transformedPosition.x;
        u[13] = transformedPosition.y;
        u[14] = transformedPosition.z;
    }
    for(size_t j = 0; j < keyFaces.size(); j++)
    {
        determineRotationMatrix(this->getNeighbourhood()->at(keyFaces[j]).plane.getNormal(), Vector(0,0,1), r);
        for(size_t I = 0; I < keyFaceNetworks[j].size(); i++)
        {
            glLoadMatrixfI;
            glMultMatrixf(this->getNeighbourhood()->at(keyFaceNetworks[j][i]).dispMatrix);
            glMultMatrixf(u);
            glGetFloatv(GL_MODELVIEW_MATRIX, s);
            if(s[14] > MIN_DISTANCE)
            {
                this->getNeighbourhood()->at(keyFaceNetworks[j][i]).visible = true;
            }
            else
            {
                this->getNeighbourhood()->at(keyFaceNetworks[j][i]).visible = false;
            }
        }
    }
}

Section 18: The Vertex Shader for the Vertex Shader Weighted Approach

void main(void)
{
    glFrontColor = glModelViewProjectionMatrix * glColor;
    gl_Position = gl_Vertex;
}

Section 19: The Fragment Shader for the Vertex Shader Weighted Approach

void main(void)
{
    gl_FragColor = gl_Color;
}
Section 20: The Vertex Shader for the Fragment Shader Weighted Approach

void main(void)
{
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_Position = gl_Vertex;
}

Section 21: The Fragment Shader for the Fragment Shader Weighted Approach

uniform sampler2D tex;
uniform vec3 lightPos;

void main(void)
{
    mat4 tempMatrix;
    mat4 faceMatrix;

    faceMatrix[0] = texture2D(tex, vec2(gl_TexCoord[0].s,.125));
    faceMatrix[1] = texture2D(tex, vec2(gl_TexCoord[0].s,.375));
    faceMatrix[2] = texture2D(tex, vec2(gl_TexCoord[0].s,.625));
    faceMatrix[3] = texture2D(tex, vec2(gl_TexCoord[0].s,.875));

    tempMatrix = gl_TextureMatrix[0] * faceMatrix;
    float color;
    color = dot( vec4(tempMatrix[0][2] ,tempMatrix[1][2] ,tempMatrix[2][2],
                  tempMatrix[3][2]),  vec4(lightPos, 1));

    gl_FragColor = vec4(0,0,color,1);
}