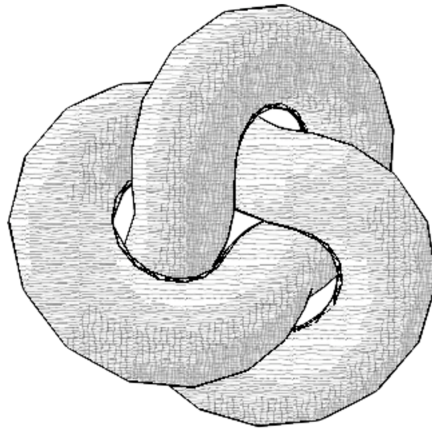
Non-Photorealism in Interactive Rendering Systems

Jorick van der Hoeven

A Project presented for the degree of

BA in Computer Science



Computing Laboratory
University of Oxford
England
May 2008

Non-Photorealism in Interactive Rendering Systems

Jorick van der Hoeven

Submitted for the degree of BA in Computer Science
May 2008

Abstract

This project discusses interactive non-photorealistic rendering techniques. It is split up into two sections – outlining merhods and shading methods. Three outlining methods were implemented – stencilling, front-face culling and ink-sketching. Stencilling uses the stencil buffer to create a mask to draw the outline. Front-face culling uses edge localisation methods to draw the silhouette as well as the outline. Ink-sketching builds on front-face culling to make the edges look like they have been sketched with an ink pen. The second section discusses three shading methods – cellshading, 'simple' crosshatching and 'fine' crosshatching. The cellshading method uses 1D textures to make the shading of the model discrete. 'Simple' crosshatching uses textures to shade shade the polygons. 'Fine' crosshatching moves beyond current research and refines the 'simple' crosshatching to make it more accurate on models with low polygon counts. All of the work presented in this paper is designed to work in real-time with speeds ranging from 24 to 60 frames per second.

Keywords: Non-photorealistic, Outline, Silhouette, Cellshading, Crosshatching, Pencil-shading.

Acknowledgements

I would like to thank my supervisor, Dr Joe Pitt-Francis, for all the help and support he has given me throughout this project. I would also like to thank Ricklef Wohlers and Jessica van der Meer for putting up with my constant questions about style issues and correct grammar. Finally, I would like to thank my family for putting me in an environment which allowed me to perform such interesting work.

Contents

Abstract **ii**

Acknowledgements **iii**

Introduction **1**

1 Background **3**

Basic Pipeline 3

Face Rendering 4

What is Non-Photorealistic Rendering 5

2 Planning **8**

The Language 8

The Graphics Library 8

The Platform 9

The Design of the System 9

3 A Quick Run-Through the Program **12**

4 Non-Photorealistic Rendering Techniques **18**

Outlines 18

Stencil Buffer 20

Front-Face Culling 23

A Fast Randomised algorithm for finding outlines 24

Ink Sketch Outlines 26

Cell Shading 30

Lighting 30
Textures 31
Implementation 32
Crosshatch Shading 35
‘Simple’ Crosshatching 35
’Fine’ Crosshatching 39
Conclusions 46

5 Helper Functions for the Program 48
The Magnifying Glass 48
Manipulating the Object using a Trackball 51
Loading Models 52

6 Conclusions 53
Overall Conclusion 53
Learning Outcomes 54
Future Work 54

Bibliography 56

Figure and Algorithm Lists 59

Introduction

Computer Graphics is a fascinating topic and is becoming even more so as computers become more powerful. This project set out to build a game engine for a 3D environment but ended up focusing on non-photorealistic rendering instead. This change of focus was not motivated by any difference in difficulty between the two tasks but rather by the fact that non-photorealistic rendering is a fascinating, nascent, subject uncommon in computer graphics literature. Therefore, this project has become about some of the different methods used in non-photorealistic rendering.

This project's exposition is limited to two types, outlining and shading. The outlining methods discussed will explain how to render images in real-time with the outline and silhouette lines included without greatly affecting the rendering performance. Furthermore, this project moves on from rendering basic outlines and describes how to draw them in an ink sketched way.

The shading methods are the most interesting part of the project and contain a contribution to the field of non-photorealistic rendering. At first cellshading and one of its common implementations is introduced. Crosshatch shading is then explored and two methods are proposed. The 'Simple' Crosshatch shading method – illustrated in [7] although not implemented – and the 'Fine' crosshatch shading method which extends the paper's discussion of the crosshatching method to increase the quality of rendering when dealing with models with low polygon counts.

Several helper functions were also developed to assist in the demonstration of the non-photorealistic rendering techniques explored. The most important helper functions involved loading models, two different functions were created for this, one for loading MD2 model files and one for Wavefront OBJ files. The remaining two helper functions implemented are a magnification system to examine rendering

defects and a trackball system to allow a user to rotate the rendered model freely.

Before delving into the details of how these methods work Chapter 1 presents the background details that one should be aware of before attempting to understand how the methods work. Chapter 2 discusses the different design decisions made during the project and gives a brief overview of the demonstration program's functionality. Chapter 3 provides a run-through of the program and depicts all the different rendering styles this project can perform and demonstrates a help screen used by the program to aid a user in getting the program to display the rendering techniques they desire. Chapter 4 is the meat of the project report and discusses the non-photorealistic rendering methods in detail providing algorithms and comparisons. Chapter 5 quickly showcases the helper functions used by this project. This report is finally concluded in chapter 6 where the results of this project are discussed and possible future work is described.

Chapter 1

Background

This section will explain how the graphics pipeline works and how computers can move from a collection of numbers representing points and vectors in an arbitrary coordinate space to a stunning image representing a three-dimensional scene. It will then discuss the various existing methods for performing these actions and discusses the pros and cons of each. Finally, it concludes with an introduction to non-photorealistic rendering, explaining what it is; how it has evolved; and which aspects of non-photorealistic rendering this project focuses on.

Basic Pipeline

Figure 1.1 details a generalised graphics pipeline used to display computer graphics onto a screen. For the purposes of this project and to keep the explanation simple we will only focus on three-dimensional (3D) rendering. In such a context, an application defines a 3D model in what is known as the 'world coordinate system'. This is a coordinate system used by applications to keep track of where different objects are in the scene and to move them if necessary. An application will pass

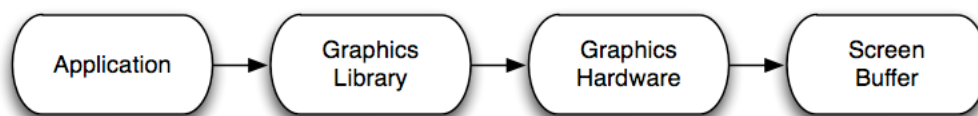


Figure 1.1: A Simple example of the Computer Graphics pipeline

all the objects that it has been instructed to draw and pass them to the graphics library to be processed.

The graphics library, given the coordinates of the objects and the appropriate drawing settings, will perform the translation and projection operations to move the object coordinates from the world coordinate system to the two-dimensional (2D) screen coordinate system. Throughout the process, it will also calculate the lighting and shading of the scene. When this is done the colour information and the line information is passed to the graphics hardware, which in turn will perform the raster operations needed to draw the lines and shapes defined in a continuous coordinate system on the discrete coordinate system that is the screen.

This, of course, has only been a brief summary of how the rendering pipeline works. Empirically, the graphics hardware might take over earlier from the graphics library or the graphics hardware may not exist at all. This will cause changes in performance, but will not affect the general order of operation.

Face Rendering

When determining how to draw an object to the screen there are two main methods, ray tracing and face rendering. Ray tracing works by firing rays out from the virtual camera at the scene. Each point in the scene touched by a ray will modify the color of the pixel the ray represents according to its material properties and then reflect or refract the ray. Traditional ray tracing based systems will allow each ray to be reflected or refracted a certain number of times before stopping the ray and moving onto the next one.

Face rendering will convert the points in the 3D space to a 2D space by using matrix calculations. The colours of the pixels are then calculated by interpolating the colour values between the vertices of the objects. A face renderer – as its name suggests – works primarily with faces. That is to say that the vertices defining an object are collected into polygonal 'faces' which are oriented (having a front and back). These polygons are joined up into a mesh that describes the model to display. To calculate the lighting effects for this mesh each vertex is assigned a

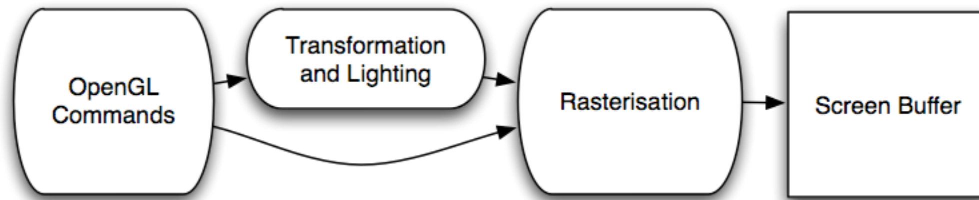


Figure 1.2: The pipeline for a face renderer

corresponding normal vector. The face renderer takes those vertices and projects them onto a two-dimensional plane. Once there, it will use the normals for each of the points to determine what colour the particular vertex should be. When this is done, the face renderer interpolates between the vertices defining the colour of each pixel and sends them to the screen buffer.

Figure 1.2 provides a simple overview of how the OpenGL face renderer processes model data. The OpenGL Commands square represents a collection of vertex data, normal data, lighting parameters, etc. which OpenGL sends to be transformed and lit before it is rasterised and sent to the screen. The shortcut arrow in the model is included for accuracy as the OpenGL Graphics library can also display images in raw format and rasterise them immediately[1].

What is Non-Photorealistic Rendering?

Over the last twenty years computer graphics has evolved tremendously, moving from pixelated images in two dimensions to hyper realistic 3D environments which are almost completely modifiable. This great achievement has been made possible by computer systems which have become increasingly powerful, satisfying the needs of some of the most resource intensive imaging systems. Programs using graphics hardware are now able to make their creations as life-like as possible, increasing the level of detail and mimicking real world lighting effects. Over the last ten to fifteen years, however, some artists have moved away from realist goals towards rendering suggestive images using computers. This gave birth to non-photorealistic rendering.

[1]See [24] for more information on this

Non-photorealistic rendering tries to capture the essence of the subject it is trying to depict in way similar to an artist sketching on a piece of paper, or painting on a canvas. This is a laborious process for computers given that most graphics hardware has been programmed with ultimate realism in mind. To overcome this, non-photorealistic renderings have to perform hacks to try to modify the graphics pipeline to make the graphics hardware work differently than designed.

The first approaches to non-photorealistic rendering converted previously photorealistically rendered images and photographs so that the image appears drawn. To do this, scientists and artists involved in the development of non-photorealistic rendering systems devised very efficient methods to modify colours in pixel based images; weaning out the information about the image that they would like to accentuate. Recently, graphics programmers have started looking at introducing non-photorealistic effects into the computer graphics pipeline and have attempted to make the computers output a non-photorealistic image from the outset, rather than generating an image and then applying their techniques. One of the first examples of this process can be seen in architectural drawings where the architects would loosen the pen plotters' clamp on the pen so that the lines the pen plotter drew from the model would appear sketched and thus appealing more to their clients.

This movement of non-photorealistic rendering techniques to the graphics pipeline has allowed for the drawing of images in real-time[2]. The first major advances in real time non-photorealistic rendering were in the mid 1990s with the release of several 3D games involving comic book characters who had to be shaded and outlined in such a way that they looked like they had been inked by a cartoonist.

Further techniques have been developed; however, the amount of processing power required is often prohibitively expensive, given that most of these techniques involve using randomisation algorithms or expensive remapping techniques. It is therefore important to look into ways to perform such effects whilst using the tools that have been optimised in the graphics hardware to perform non-photorealistic rendering because it would allow interactive 3D environments to be rendered non-

[2]This is a notion where the delay between an object being updated and it being rendered to the screen is so small that it can be considered negligible allowing updates to the model to be instantaneously represented on screen.

photorealistically.

One of the tools which this project examines is texturing. Texturing is a system used in computer graphics to paste images onto objects so that there is no need to define the detailed colour mappings involved in drawing complex objects. Textures for the purpose of this project can essentially be seen as grids of pixel information, commonly referred to as pixmaps – stored in memory. Textures are used to perform various effects in a computationally inexpensive way, as the expensive work has already been done while creating the texture.

Another more recent tool that artists have begun using when programming non-photorealistic renderers are shaders. A shader essentially determines what the colour of a pixel should be given its location regarding the models in a scene and said models' material properties. This allows for accurate definition of how much a certain pixel should be darkened when it is turned away from the light, and how smooth such a transition should be. Shading also allows the programmer to completely ignore standard lighting ideas and create effects using colour values of the image being rendered. An example of this would be to modify the shader to colour the model in relation to its mesh density.

These tools are only a few of the many different functions that are available. This project will focus on the ways to use the two tools – texturing and shading – to implement non-photorealistic rendering techniques and then compare and contrast the different methods that can be used to perform similar effects (checking the methods for speed and space requirements). Throughout, there will also be a discussion of the unimplemented methods, detailing why they weren't implemented, and their advantages or disadvantages.

Chapter 2

Planning

This chapter discusses the design decisions made during this project and justifies the chosen language and platform. Additionally, it describes some of the difficulties encountered during the project and how they were overcome.

The Language

There was a choice between several different implementation languages. The two main ones, Java and C++, were singled out for their object oriented capabilities. C++ was eventually chosen because the main reference books ([12, 28, 29][1]) were written in C++. Another reason for choosing C++ over Java is that the Direct-X and OpenGL Libraries are both written in C and it is therefore easy to link with them from C++, in contrast to Java where the JOGL bindings would need to be used to access OpenGL and one would not be able to use Direct-X at all due to the fact that there are currently no bindings for DirectX in Java.

The Graphics Library

There were three possible choices for the graphics library, OpenGL, Direct-X and Java3D. OpenGL was eventually chosen because Direct-X is not available on Mac OS X and Java3D does not offer the functionality necessary for the purposes of

[1] These books were chosen witht the original project in mind.

this project. Another factor was that the project supervisor, Dr Joe Pitt-Francis, is familiar with OpenGL and would therefore be able to provide implementation assistance if needed.

The Platform

The machine used for the programming of this project is running Mac OS X. This placed limitations on the platforms available to present the renderer's output. The first platform attempted in this project was the game engine included in the book "Ultimate 3D Game Engine Design & Architecture" *[28].* The game engine, however, was unsuitable for the project as the windowing system was difficult to understand and elements of the engine kept breaking due to compatibility issues (the engine was built for computers running Mac OS X 10.2 on PowerPC architecture whereas the project was written on Mac OS X 10.5 on Intel architecture).

The game engine was therefore abandoned and a choice existed between four alternatives, Cocoa, Carbon, GLUT and SDL. Cocoa and Carbon were too complicated for the purposes of the project and would have required unnecessary work. GLUT and SDL were much better suited as they were simpler and allowed for easy event and window management. SDL was chosen over GLUT because it allowed for more flexibility when dealing with windows and user input than GLUT does.

Another platform used in this project is ImageIO, a Mac OS X specific library used to load and decode images so that they could be used as textures in OpenGL.

The Design of the System

The system created in this project is a demonstration of a few real time non-photorealistic rendering techniques. The classes used for the rendering use OpenGL and SDL to take objects and draw them non-photorealistically to the screen. Figure 2.1 shows that the program will allow the user to see several outlining and shading techniques and compare them to a standard photorealistic rendering. This project will discuss the methods highlighted in the flow chart as well as the advantages and

disadvantages of the methods which this project features.

The chart highlights where the rendering pipeline fits into the system. The methods detailed here will be performed right after the model transformation, changing the shading and determining the outlines before allowing OpenGL to finish the rendering.

The entire project has been written in such a way that each of these functions can be turned on or off individually, allowing one to see how much of an effect each function has on the rendering speed. Some of the different rendering methods cannot be drawn simultaneously so they have been separated out. This is the case for the outline and the shading styles.

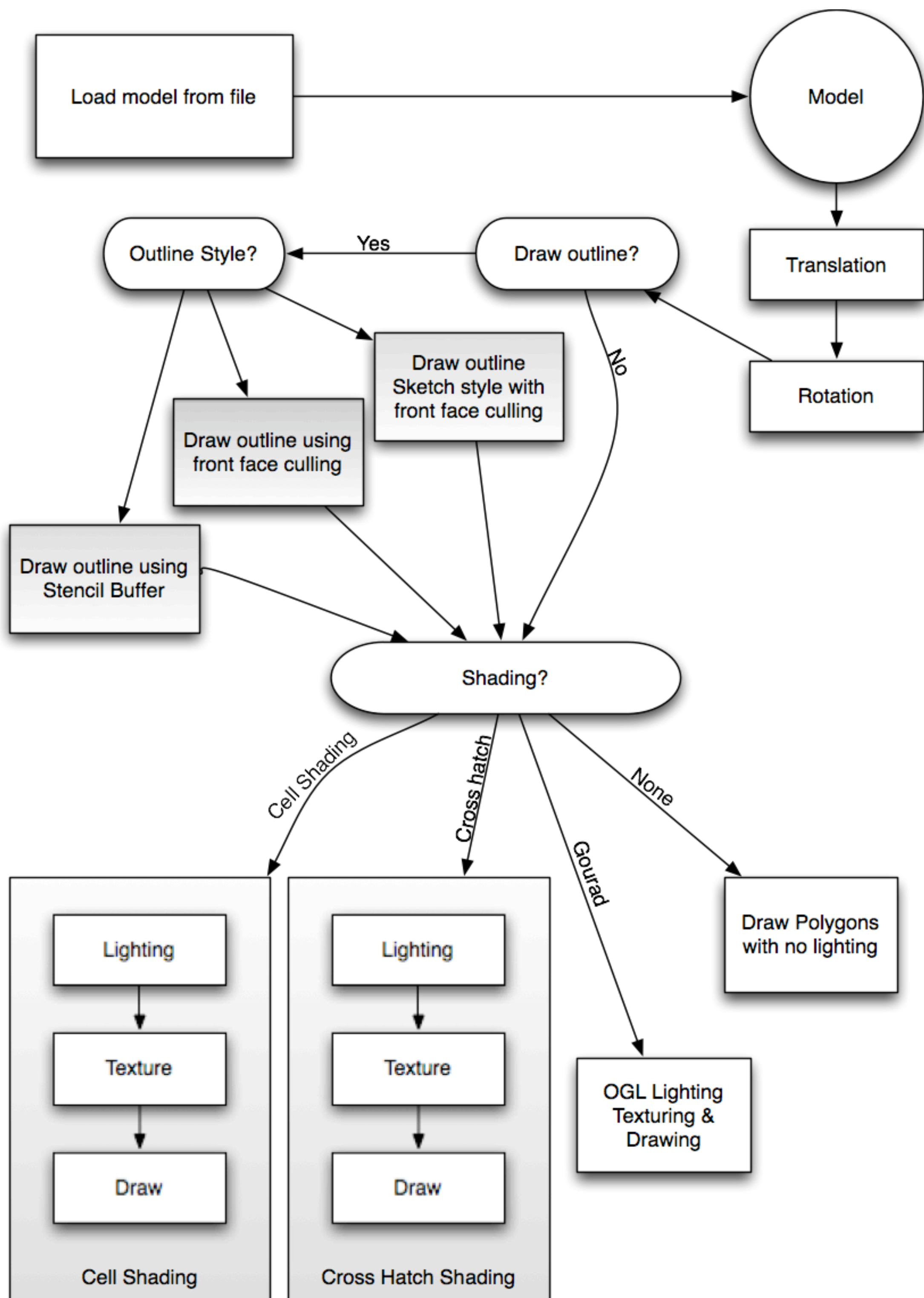


Figure 2.1: This is the schema for the way the system renders the models non-photorealistically and the different options available.

Chapter 3

A Quick Run-Through the Program

This project's program works in four steps:

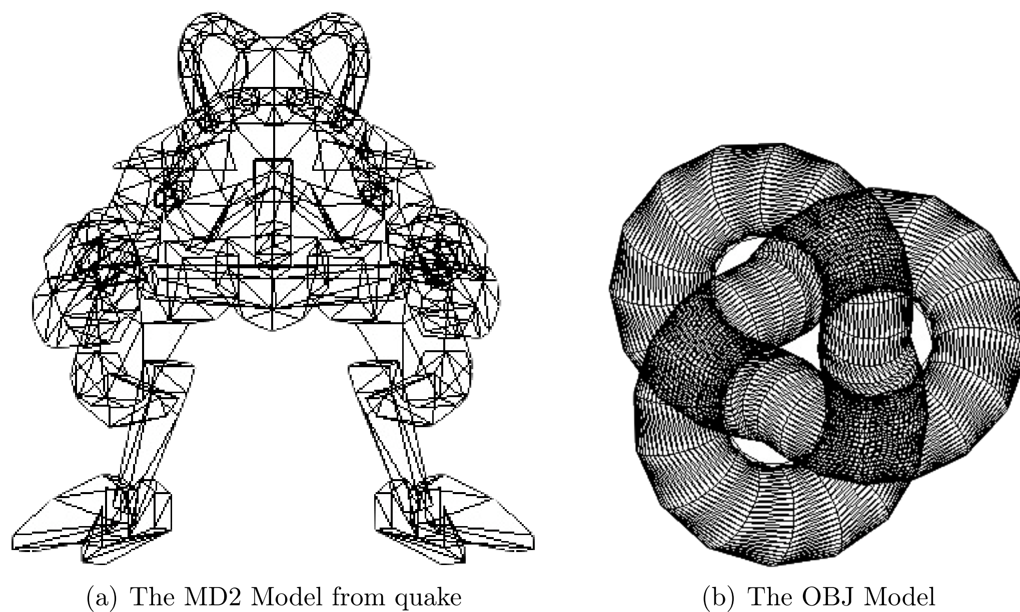
1. Choose the model file to load (currently models in the MD2 and OBJ file formats are supported).

2. Read the model file and parse it into a model object in memory.

3. Choose a rendering style.

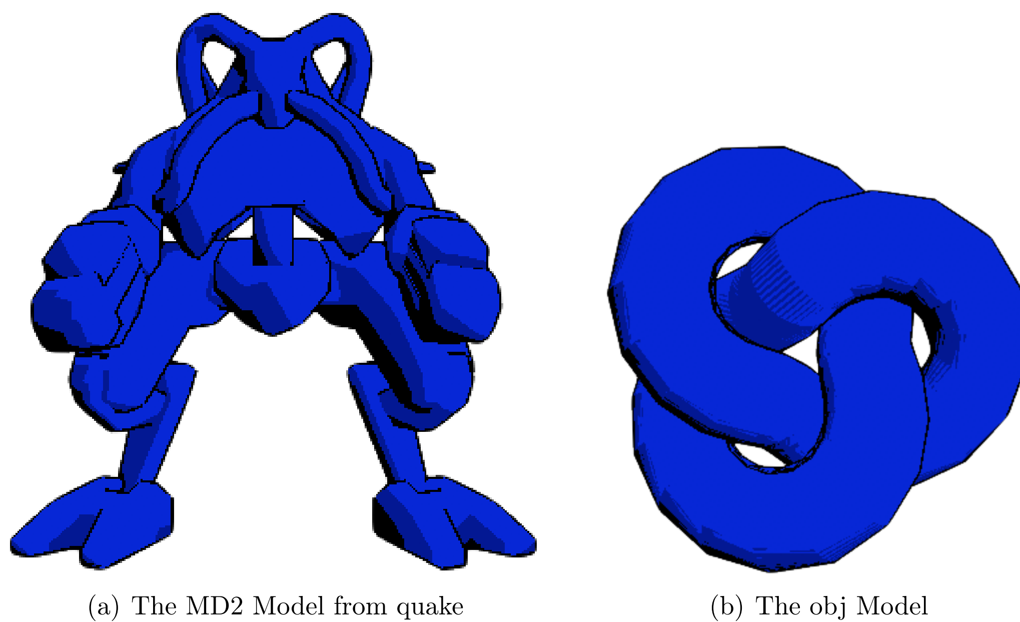
4. Perform the lighting and shading calculations and make the relevant OpenGL calls to render the model.

User interaction is only needed for the first and the third steps although the program is loaded with enough defaults to be able to display an image upon starting the program. The figures below show some of the rendering styles possible on the two default models.



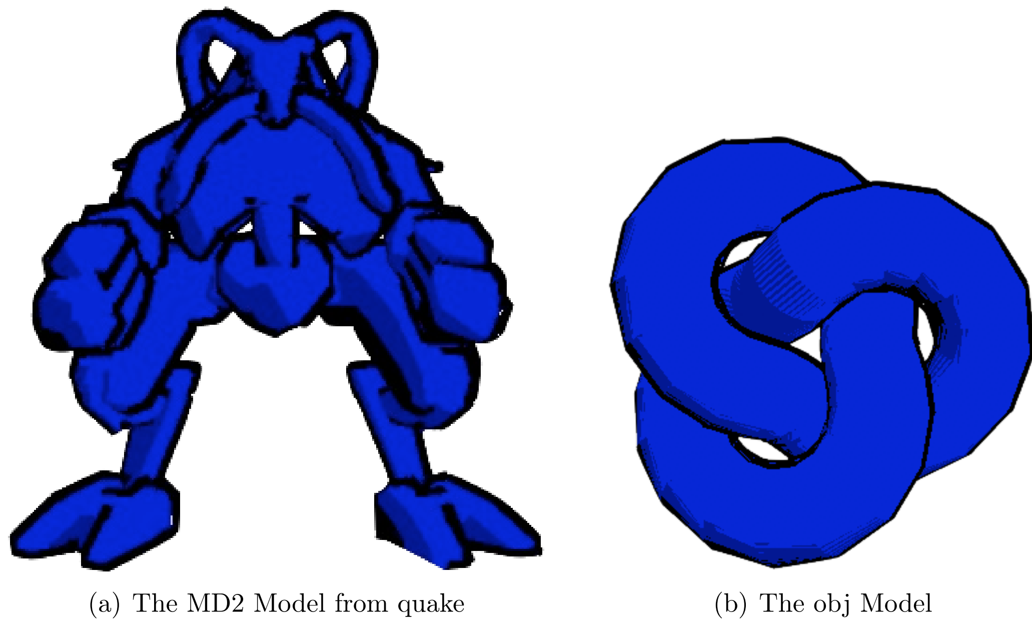
(a) The MD2 Model from quake (b) The OBJ Model

Figure 3.1: The initial loading of the model in wireframe.



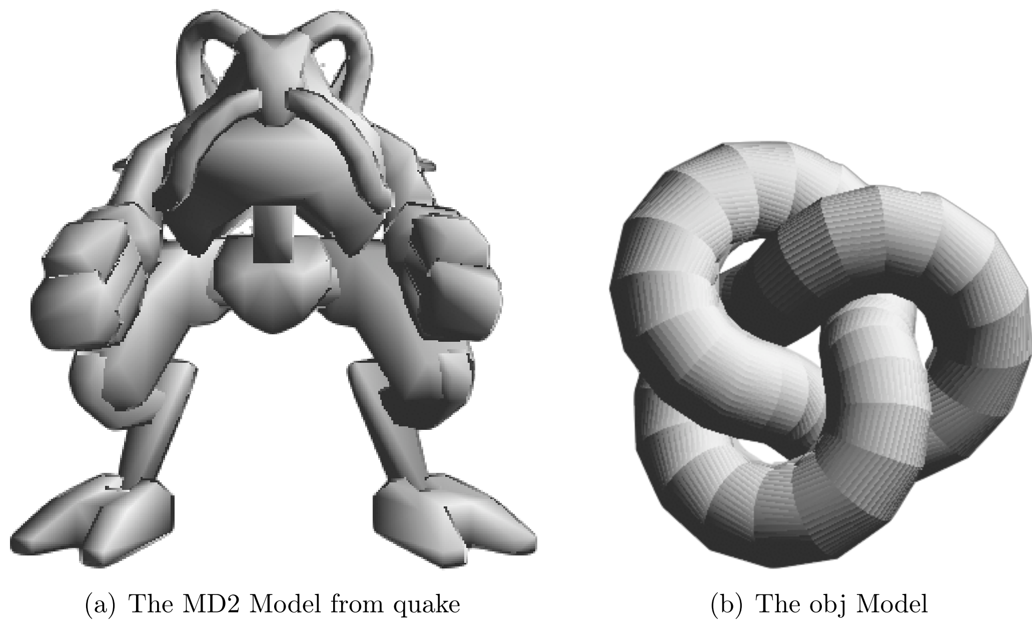
(a) The MD2 Model from quake (b) The obj Model

Figure 3.2: The cellshaded models.



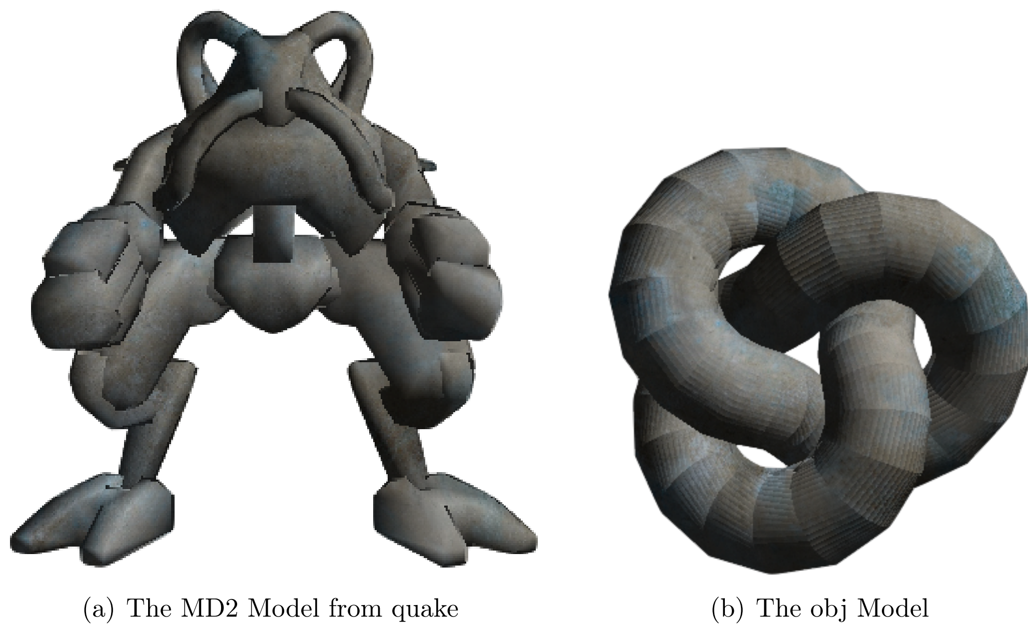
(a) The MD2 Model from quake (b) The obj Model

Figure 3.3: The cellshaded models with ink sketched outlines.



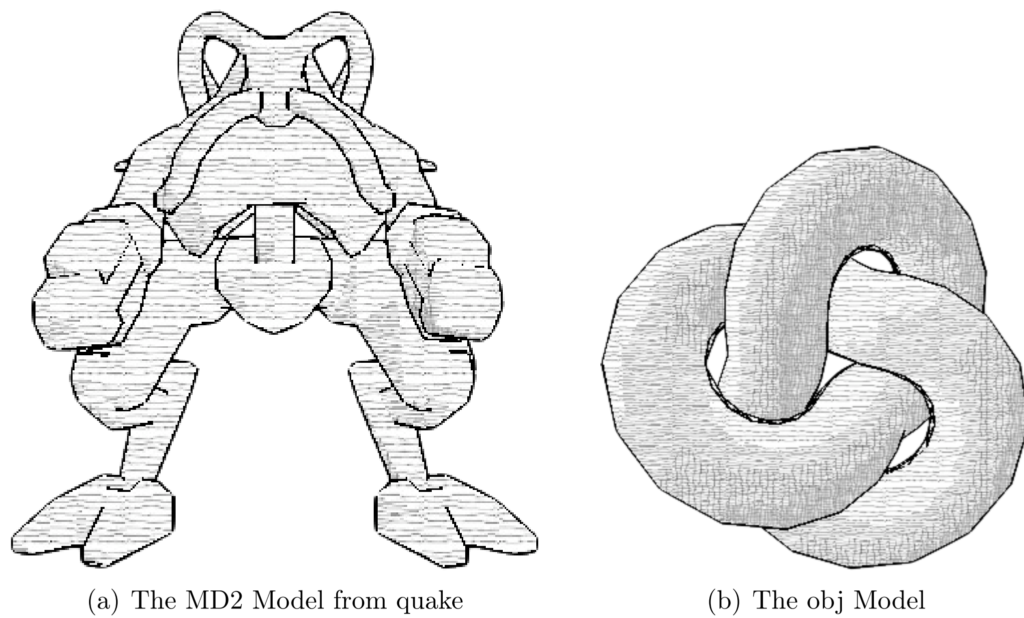
(a) The MD2 Model from quake (b) The obj Model

Figure 3.4: The models rendered with traditional photorealistic gourad shading.



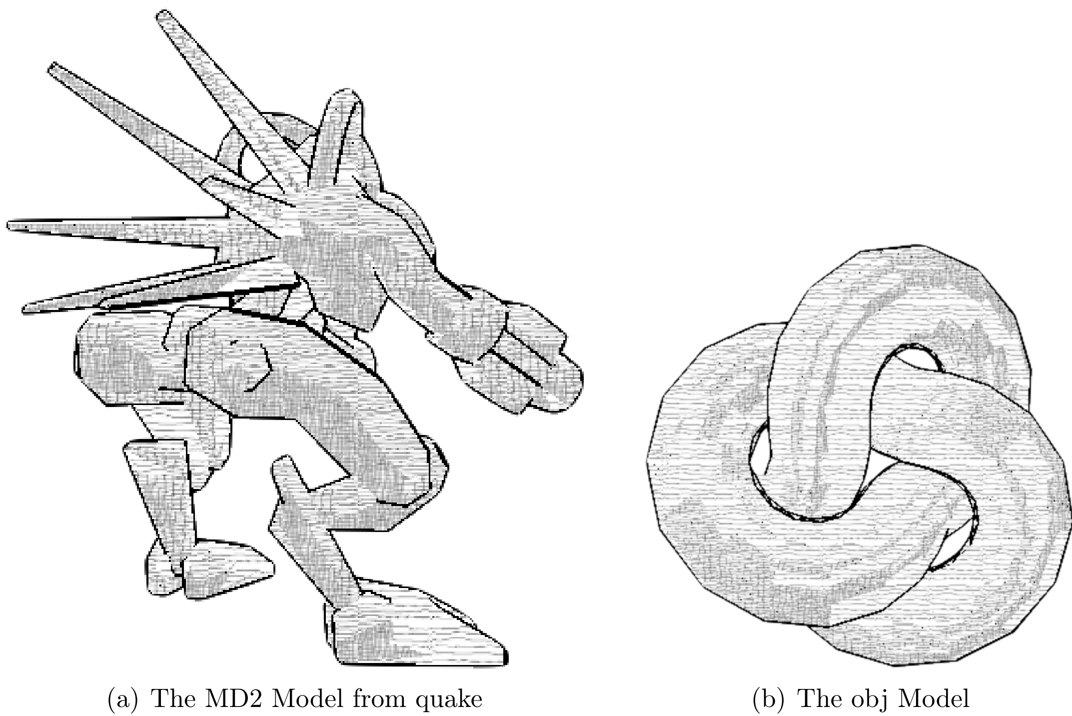
(a) The MD2 Model from quake (b) The obj Model

Figure 3.5: The Models rendered with gourad shading and a metallic texture.



(a) The MD2 Model from quake (b) The obj Model

Figure 3.6: The models rendered with crosshatch textures for shading, ‘simple’.



(a) The MD2 Model from quake (b) The obj Model

Figure 3.7: The models rendered with crosshatch textures for shading, 'fine'.

Figure 3.8: The Help Screen

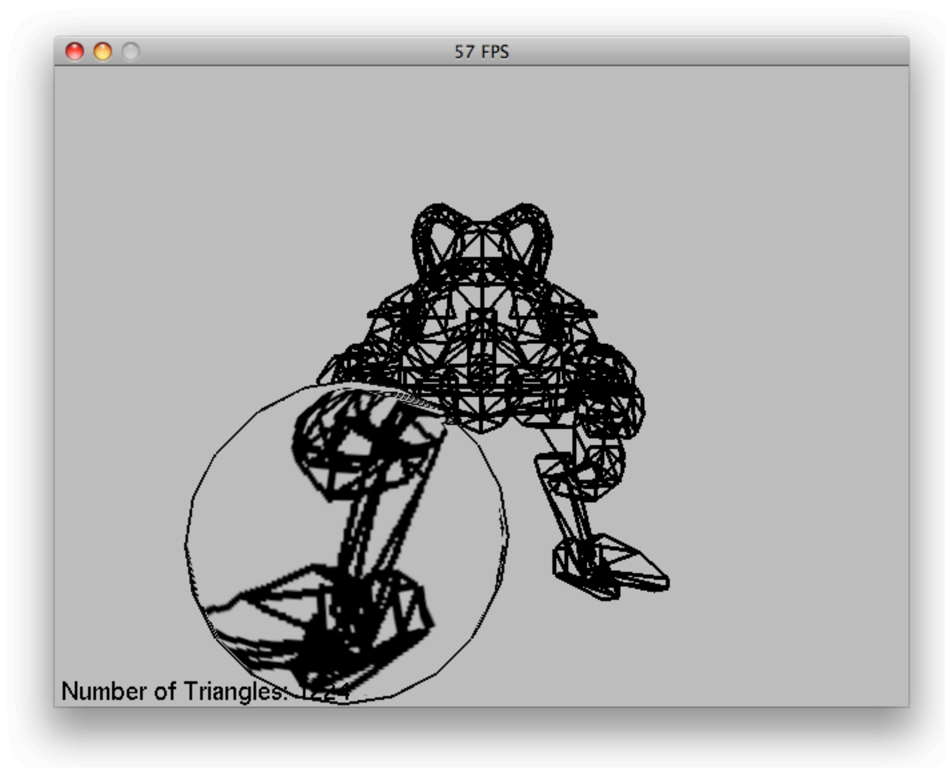


Figure 3.9: The Magnifying Glass

Chapter 4

Non-Photorealistic Rendering Techniques

This chapter discusses the different non-photorealistic methods implemented in this project. It will elaborate upon the advantages and disadvantages of each method and their alternative implementations.

Outlines

When drawing pictures, artists will simplify the image that they see to put it onto paper. One of the main techniques that an artist will use is to detect the edges of the subject being drawn and draw lines that will act as outlines for his drawing. The artist will also include lines for areas where wants to depict folds. These outlines can be divided into two different styles of outlines, boundary lines and inner lines. The boundary line is the continuous line that surrounds the object being drawn and, as its name suggests, marks the boundary between the object and the rest of the scene. The inner lines mark places where the model folds over itself (as in Figure 4.1).

When attempting to mimic the way an artist draws an object both these lines need to be calculated. The way to do this is to detect the edges of the model. There are then two types of edges which need to be detected in the scene. The edges that mark separation between the object and the environment and the edges that mark the presence of a fold in the object, which correspond to boundary lines and inner

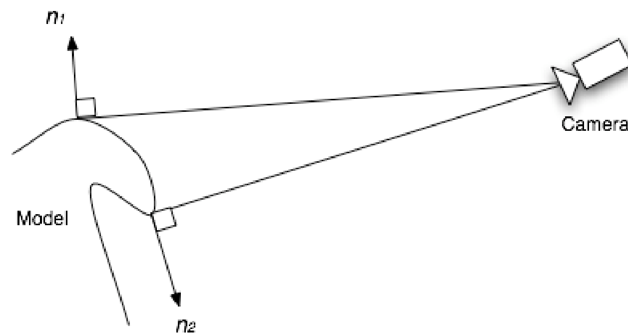


Figure 4.1: Edge detection in Model space

lines respectively.

Edges are defined as the place where the surface of the model goes from an orientation where it is facing the viewer, to where it is no longer facing the viewer. Hence, in edge detection, it is essential to find the places where the model's surface normal is perpendicular to the viewing direction. To find these places a vector from the camera to the point in question is created and checked for perpendicularity to the normal vector of the model's surface. Figure 4.1 shows that this method of detecting the edges encompasses both boundary lines and inner lines. To determine whether the normal is perpendicular to the camera vector the following calculation is performed:

$$n \cdot v = 0 \tag{4.1}$$

Where n represents the normal of the model at any given point and v represents the direction from the camera to that point. If this dot product is 0 then the angle between the two vectors is 90º, marking a point where the model's surface becomes parallel to the camera view vector which thus defines an edge. Once the edges of the model have been found, lines can be added portraying the desired outlines.

This project implements two different methods for outlines, one which uses the discussion above – front-face culling – and another – stencil buffering.

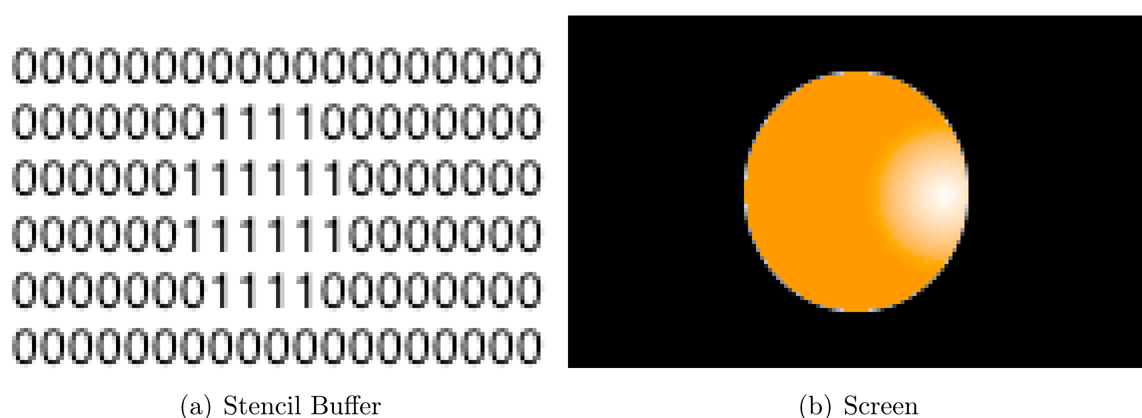
Stencil Buffer

One way of drawing the boundary of an object without performing any edge detection is to create a mask of the shape of the object and trace around it. To do this in OpenGL stencil buffering is used. The stencil buffer is an area of memory used by the OpenGL libraries to determine whether a pixel should be drawn or not. To use the stencil buffer to draw an outline, the following steps are used:

1. Activate the stencil buffer.
2. Set the stencil buffer's attributes so that it will write 1's to the stencil buffer when a draw command is sent.
3. Draw the model.
4. Change the stencil buffer's attributes to make the stencil test check the stencil buffer before drawing.
5. Draw the model to the screen in wireframe.

The important processes of the stencil buffering method take place in steps 2 and 4 where the stencil buffer attributes are set. The stencil buffer is cleared, setting all the bits in it to 0. Creating a view port sized bitmap with all 0's in it.

In step two, a function is defined that, every time a pixel is drawn to the screen places a 1 in the stencil buffer at the same coordinates as the pixel that was drawn to the screen. This allows for simultaneous definition of a mask and rendering a model (Figure 4.2).

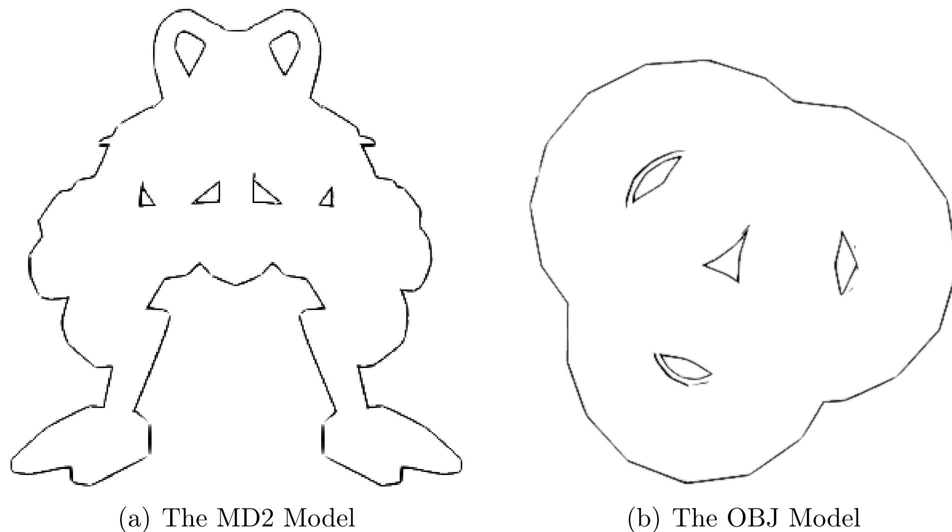


(a) Stencil Buffer (b) Screen

Figure 4.2: The states of the buffer and the screen after step 3

In step 4 the function is modified in such a way that it checks to see if the position of the desired pixel is 1 in the stencil buffer. If it is, the drawing of the pixel is blocked. If it isn't, the pixel is drawn to the screen.

With the new stencilling function setup the model is drawn again, this time in wireframe and slightly larger than the original model. The effect of the stencil function causes only the parts of the wireframe model that are drawn outside the original model to be drawn creating a smooth outline around the object.

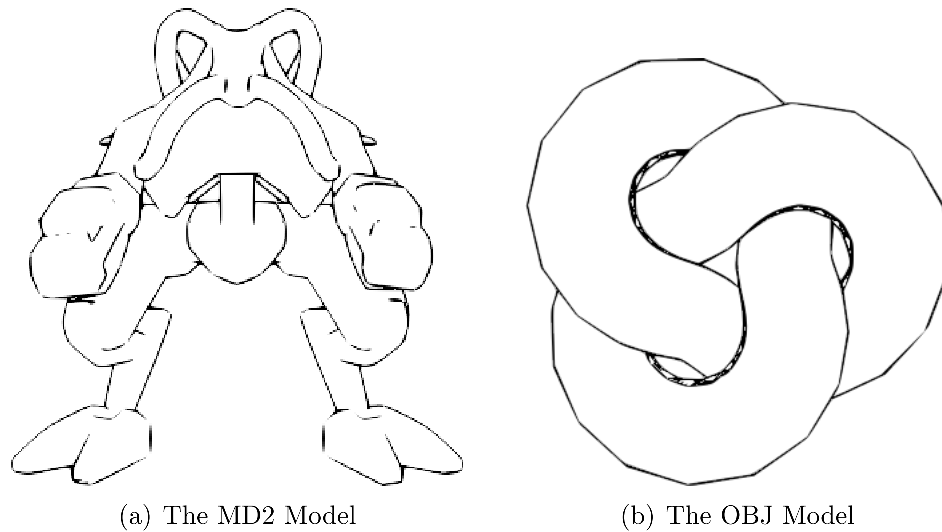


(a) The MD2 Model (b) The OBJ Model

Figure 4.3: The output of the stencil buffer outlining code

Algorithm 1 The OpenGL code for rendering an outline using the Stencil Buffer

```
1  //1
2  /*Set the Value to which the Stencil Buffer should be
      cleared*/
3  glClearStencil(0);
4  /*Clear the Stencil Buffer*/
5  glClear(GL_STENCIL_BUFFER_BIT);
6  /*Enable the use of the stencil function*/
7  glEnable(GL_STENCIL_TEST);
8
9  //2
10 /*Set the stencil function to always return true no
      matter what*/
11 glStencilFunc(GL_ALWAYS, 0x1, 0x1);
12 /*Tell the renderer what to do if the stencil function
      passes in this case we tell it to replace the 0 value
      */
13 glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
14 /*Set the polygon mode so that the model is fully
      rendered*/
15 glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
16
17 //3
18 /*Draw the Model*/
19 RenderTheModel();
20
21 //4
22 /*Change the stencil function to pass only when the
      stencil buffer is not equal to 1*/
23 glStencilFunc(GL_NOTEQUAL, 0x1, 0x1);
24 /*Set the line width of the wireframe rendering allowing
      us to decide how much the wireframe should spill out
      past the boundary of the model*/
25 glLineWidth(params.outlineWidth);
26 /*set the rendering mode to wireframe*/
27 glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
28 /*set the color to black*/
29 glColor3f(0.0f,0.0f,0.0f);
30
31 RenderTheModel();
```



(a) The MD2 Model (b) The OBJ Model

Figure 4.4: The Front Face Culling technique for drawing inner and boundary lines

Front-Face Culling

Whilst drawing the boundary outlines using the stencil buffer is quick it unfortunately fails completely to find and draw any of the inner lines. As previously discussed, any outline that must be drawn, whether it be inner or boundary, can be found by finding all the points where $n \cdot v = 0$. Since polygonal models are used in this project the theory can be simplified by saying that every edge between a front facing polygon to a back facing polygon should be drawn – as those are the edges at which the camera view vector is perpendicular to the surface normal.

OpenGL has a command to help. glCullFace allows the programmer to cancel the drawing of polygons which are facing a certain direction. This takes the calculation – determining whether a polygon is front or back facing – and performs it directly on the display hardware. To use this method optimally culling faces with drawing the edges connecting front and back facing polygons must be combined with this method.

Building from the previous implementation the renderings of the model are overlapped to draw the edges. The model is rendered once in wireframe with glCullFace set to GL_FRONT, only drawing the outline of the back-facing polygons and then the model is rendered a second time in whichever rendering style we prefer with glCullFace set to GL_BACK, hence only rendering the front facing polygons. These two overlapping rendering work in a very similar way to stencil buffering except that

now the wireframe can spill out from behind the model for the inner lines as well.

It is essential that the depth buffer in OpenGL be enabled to ensure that the wireframe drawn in the first rendering is obscured by the final rendering in such a way that only the edges between front and back facing triangles can spill out from behind the final rendering.

These processes yield the following algorithm:

1. Draw the model in wireframe, but only the faces that are facing backwards.

2. Render the front facing polygons normally on top of the wireframe allowing the depth testing to obscure the edges that are fully back facing leaving only the front facing polygons and the lines at the edges where a front facing polygon meets a back facing one.

The overhead of drawing the model twice in this algorithm is usually acceptable, however, once models start having high polygon counts it is more beneficial to search for the edges manually.

A Fast Randomised Algorithm for Finding Outlines

Unfortunately this method was not utilised in this project. However it still remains an interesting technique at as it solves some of the problems arising with the front-face culling method. For example, the algorithm, rather drawing the model twice which would be prohibitively expensive when dealing with highly detailed models, performs a search through the edges in the model to find the edges that correspond to the inner and boundary outlines [22]. Before searching through the edges, the algorithm first sorts all the edges according to their dihedral angles[1]. Each edge is then assigned a probability value which decreases as the dihedral angle increases. This is done because the larger the dihedral angle is, the less likely the edge in question will be an edge between a front and a back facing polygon.

Now that the edges have been sorted the algorithm will randomly, giving appropriate weight to more probable angles, look for edges that mark the boundary

[1] The dihedral angle is the angle between the two faces that constitute the edge.

Algorithm 2 The OpenGL code for rendering boudary outlines and inner outlines using front-face culling

```
1  //1
2  /*Set the rendering mode to wireframe*/
3  glPolygonMode(GL_BACK, GL_LINE);
4
5  /*Make the wireframe lines the desired width*/
6  glLineWidth(params.outlineWidth);
7
8  /*Set the face culling system to cull forward facing
      polygons*/
9  glCullFace(GL_FRONT);
10
11 /*Set the depth function to keep objects that are in
      front or on the same level as other objects.*/
12 glDepthFunc(GL_LEQUAL);
13
14 /*Set the color of the outline*/
15 glColor3fv(&params.outlineColor[0]);
16
17 DrawModel();
18
19 //2
20 /*Set the face culling system to cull backward facing
      polygons*/
21 glCullFace(GL_BACK);
22
23 /*Set the depth function to keep objects that are
      strictly in front the objects drawn. */
24 glDepthFunc(GL_LESS);
25
26 /*Set the rendering mode to filling*/
27 glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
28
29 DrawModel();
```

between a front and a back facing polygon. When such an edge is found, all the adjacent edges are checked to see if they are inner or boundary edges. The algorithm proceeds accordingly until the entire inner or boundary outline has been drawn[2] – after which it begins searching randomly again. It has been calculated by [22] that the algorithm maintains a constant probability of finding the boundary lines if $O(\sqrt{n})$ edges are checked, where n is the number of edges in the model.

This process does not have to be repeated entirely for every screen update because when the model does not move the inner and boundary edges will not change. For small transformations the old inner and boundary edges can be used as seeds for the search algorithm as there is a large probability that the edges are the same, or close to the old ones.

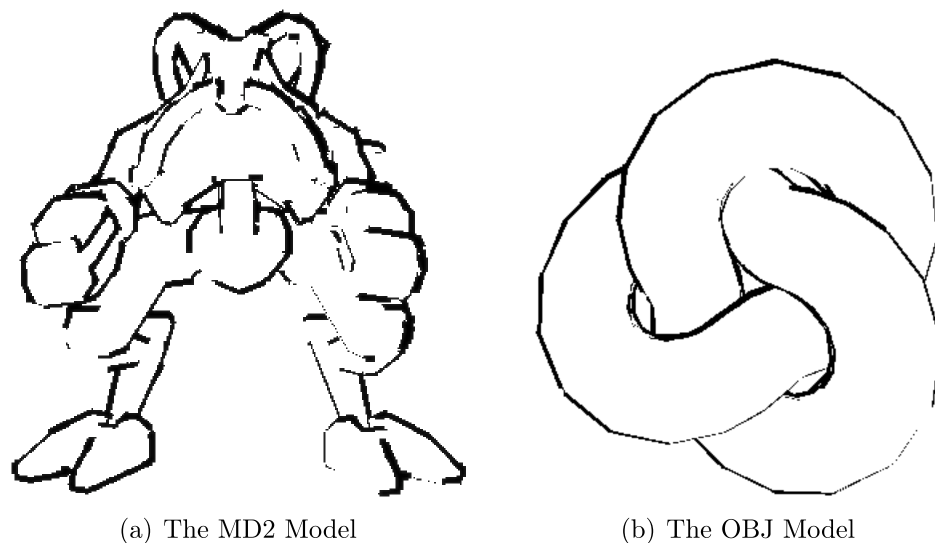
This method manages to deal very well with finding boundary outlines and will, in practice, find the entire boundary. The problem, however, is that it has difficulty with the inner lines – tending to be quite short they will often be missed by the random search algorithm. It does, however, solve the problem of rendering the model twice and could be made to have a reasonable degree of accuracy as missing a few inner lines is not problematic for most non-photorealistic rendering purposes.

Ink Sketch Outlines

Artists however, don't always draw every line with the same width or in exactly the right direction. To mimic this randomness needs to be added to the way the edges are drawn. The method implemented in this project provides a way of viewing the object with the randomised outlines in such a way that the model appears to be sketched quickly by an artist using an ink pen. This leads to an effect where some of the lines are quite thick whereas others appear so thin as to be almost non-existent. In addition, there will not always be one visible line for each defined edge as has been the case previously. Every line in the ink sketch will be present multiple times to make it appear that the artist needed a few attempts before finding exactly the right direction for the edge they desired.

To implement the ink sketch outlines in this project the concepts used for the

[2]We are assuming he model meshes are interconnected and non-overlapping here



(a) The MD2 Model (b) The OBJ Model

Figure 4.5: The ink sketch outlines

front-face culling outlining method are built upon. In fact, the method is almost completely identical except that instead of drawing straight lines between the vertices; three lines are drawn, connecting imaginary points that are slightly off centre from the original point (Figure 4.6).

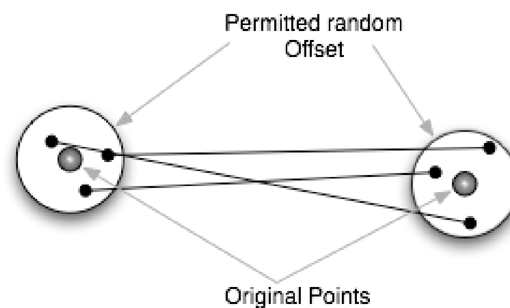


Figure 4.6: Illustration of how the ink sketch outliner would draw a line

To achieve this offset three arrays need to be created (one for each x, y and z) with as many cells as there are triangles in the model. Each cell in the arrays is then filled with a random variable between 0 and 1. When rendering the wireframe model, instead of drawing lines between the vertices, lines between the vertices offset in the x, y and z directions by the random variables stored in the three arrays are drawn. The line drawing is performed twice more, each time swapping the arrays assigned to the x, y and z parameters of the vertices. This method yields the following the following algorithm:

1. Calculate the offsets needed at initialisation

2. Draw the model in wireframe with the offsets, drawing each triangle three times with different offsets.

3. Render the front facing polygons normally on top of the wireframe allowing the depth testing to obscure the edges that are fully back facing leaving only the front facing polygons and the lines at the edges where a front facing polygon meets a back facing one.

Algorithm 3 The ink sketch algorithm

```
1  /*We initialise the arrays with random values, we adjust
      the size of the random values to the range of the
      vertices in the model*/
2  initialiseArraysWithRandomValues(array1,array2,array3);
3
4  //1
5  /*Set the rendering mode to wireframe*/
6  glPolygonMode(GL_BACK, GL_LINE);
7
8  /*Make the wireframe lines the desired width*/
9  glLineWidth(params.outlineWidth);
10
11 /*Set the face culling system to cull forward facing
      polygons*/
12 glCullFace(GL_FRONT);
13
14 /*Set the depth function to keep objects that are in
      front or on the same level as other objects.*/
15 glDepthFunc(GL_LEQUAL);
16
17 /*Set the color of the outline*/
18 glColor3fv(&params.outlineColor[0]);
19
20 /*Draw the model three times with the offsets for the x,y
       and z values taken from a different array each time*/
21 DrawModelWithOffset(array1, array2, array3);
22
23 DrawModelWithOffset(array2, array3, array1);
24
25 DrawModelWithOffset(array3, array1, array2);
26
27 //2
28 /*Set the face culling system to cull backward facing
      polygons*/
29 glCullFace(GL_BACK);
30
31 /*Set the depth function to keep objects that are
      strictly in front the objects drawn. */
32 glDepthFunc(GL_LESS);
33
34 //3
35 /*Set the rendering mode to filling*/
36 glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
37
38 DrawModel();
```

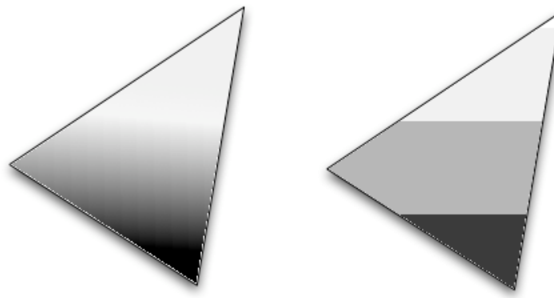


Figure 4.7: The difference between the continuous and discreet shading of a triangle.

Cell Shading

Cell shading is a method shades a model in a three-dimensional environment in such a way that the model appears to be inked by a cartoonist. Rather than allowing a shadow to move from light to dark in a continuous fashion, cell shading simplifies the shadow making it go from light to dark in a certain number of discreet steps (Figure 4.7). The number of steps is determined by the artist – but usually doesn't number beyond three or four. To make the renderer mimic this the calculation of lighting equations must be understood, so that it is possible to determine how much light is falling on a point on the surface of the model.

Lighting

When shading objects the light intensity at every point on the surface of the object must be calculated. Because this project uses polygonal models this idea can be simplified and only the incidence angle of the light ray at each of the polygon vertices needs to be calculated and then the lighting angle between the vertices can be interpolated to get the exact shading value for each point drawn.

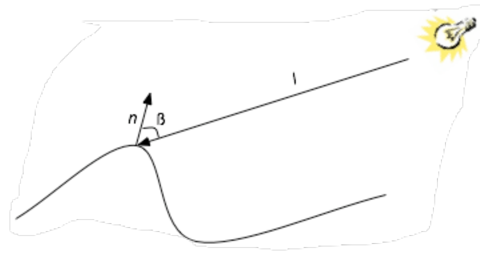


Figure 4.8: Lighting angles: n is the normal to a point on the model surface, l is the vector determining the direction from the light source to the point and β is the angle of incidence of the light to the point.

To calculate the angle of incidence of the light to a vertex the dot product of the normal at that vertex and the direction of the light to the point is performed. This is done because of the properties of the dot product operation:

$$n \cdot l = |n||l| \cos \beta$$

Given the dot product of the two vectors it becomes easy to classify the several different vertices according to the size of cos β. Where the value for cos β would usually be mapped onto a continuous function to determine exactly how much each vertex should be shaded and how much the interpolated values should be shaded, in cell shading cos β is mapped to a discreet function that will determine which shadow density to darken the pixel by (see Figure 4.10).

Textures

Texturing is very important to this project's implementation of the cell shader. A texture is a one, two or three dimensional array of colour data, luminance data or colour and alpha data[3]. Essentially, it is a store of information on the video memory that is used to modify the colours of the model that is being rendered.

When rendering polygons in a face renderer texture coordinates are applied to each of the vertices to tell the renderer which parts of the texture to add to the polygon being drawn. Texture coordinates (s, t) between 0 and 1 will denote a point somewhere on the texture. Texture coordinates greater than one depend on

[3] Alpha data is usually used for transparency

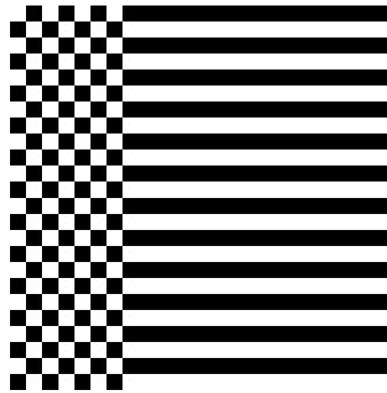


Figure 4.9: The difference between a texture being clamped and repeated. This checkerboard has been clamped in the x axis and repeated in the y axis. (Image taken from [12])

the settings of the texture. Either the texture is repeated across the surface of the polygon or else the texture data is clamped to the vertex which had the texture coordinate 0, 0 set, or the closest one to that (see Figure). A one dimensional texture is a single array of data and acts in exactly the same way as a two dimensional texture[4].

Implementation

The cell shading in this project utilised a method that creates a light map to determine which level of shading a particular point has. To use a light map in conjunction with textures a one-dimensional texture is created with values in it corresponding to the different luminosity levels in use for the cell shading effect.

The light map set up as a one-dimensional texture; the angle between the normal and the light direction at every vertex is calculated using the equation mentioned above. If it is ensured that both vectors involved in the dot product are normalised then the result of the equation will be cos β (which will never be greater than one). Using this property the result of the dot product can be used to determine the amount by which the vertex should be darkened. If cos β is less than zero the angle is greater than 90 degrees and is thus back-facing. Therefore, a lower bound is added to the result of the dot product forcing all negative values to be zero, which in the lighting map equates to a black shadow [18].

[4]See [12] for more information on textures.

As previously discussed the texture coordinates range from zero to one so the result of the dot product calculation can be used as the texture coordinates for our light map. Values closer to one will be facing the light more and will be more illuminated whereas values closer to zero will be darker (Figure 4.10). Now that the vertices of the triangle have been classified interpolation is allowed to handle the rest of the triangle causing the shadow levels to be repeated across the triangle.

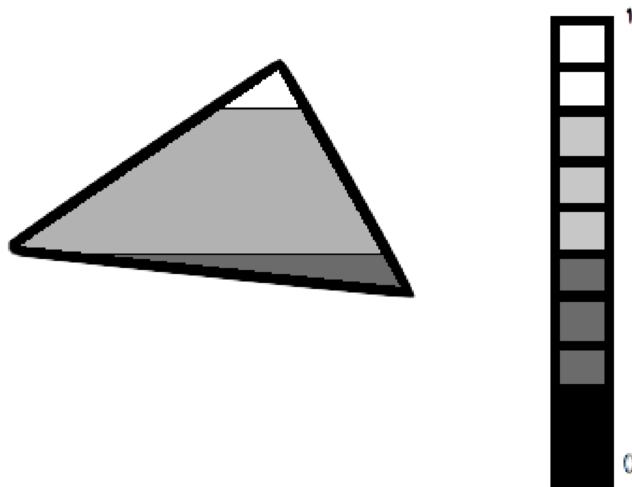


Figure 4.10: Cell shaded triangle and its corresponding 1D texture

To implement this in OpenGL the following steps were followed:

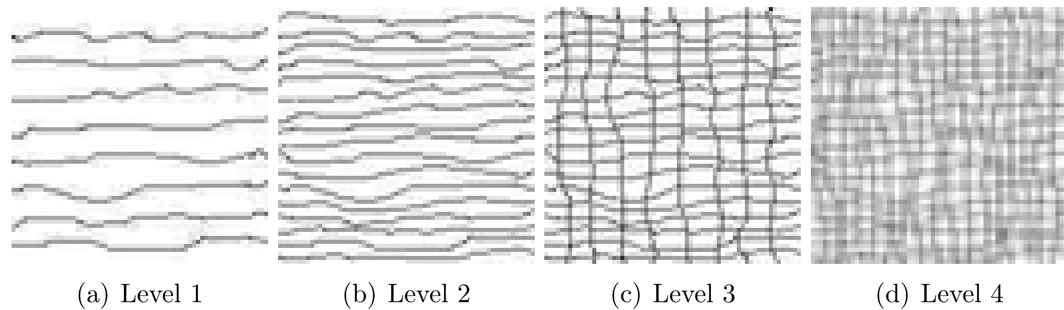
1. Set up a 1D texture and fill it with lighting values from 0 to 1. (0 being black and 1 being the color at full brightness)

2. Set the texture parameters so that the texture is repeated across the surface of the triangles. The texture parameters were also set in such a way that the texture would be blended with the color of the triangle – meaning that the method is not restricted to grey scale images.

3. Traverse all the triangles in the model, calculating the texture coordinates for each vertex and then draw the model.

Algorithm 4 The OpenGL implementation of the Cell Shading algorithm

```
1  /*generate a texture for cell shad lookup and put it in
      params*/
2  glGenTextures(1, &params.shaderTexture[0]);
3
4  /*bind the created texture to a 1D texture*/
5  glBindTexture(GL_TEXTURE_1D, params.shaderTexture[0]);
6
7  /*set the parameters of how the texture should be
      rendered if the texture doesn't fit the space that
      needs to be textured*/
8  glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
      GL_NEAREST);
9  glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
      GL_NEAREST);
10
11 /*apply the pixel data from the shader file onto the 1
      dimensional texture */
12 glTexImage1D(GL_TEXTURE_1D, 0,
13              GL_RGB, 32, 0,
14              GL_RGB, GL_FLOAT, shaderData);
15
16 /*Disable OpenGL's lighting calculations we don't want
      them getting in the way*/
17 glDisable(GL_LIGHTING);
18
19 /*enable 1 dimensional texturing*/
20 glEnable(GL_TEXTURE_1D);
21 /*Set the texture parameters to blend into the colours
      underneath*/
22 glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_BLEND);
23
24 /*Bind the texture in the params to a 1D texture*/
25 glBindTexture(GL_TEXTURE_1D, params.shaderTexture[0]);
26
27 While(not all Vertices Drawn){
28   lightDirection = Vertexi - lightPosition;
29   lightPostition.Normalise();
30   TmpShade = TmpNormal.DotProduct(params.lightAngle);
31   if(TmpShade < 0.0f)
32       TmpShade = 0.0f;
33   /*choose the point in the array determining the right
        color for the cell shading*/
34   glTexCoord1f(TmpShade);
35   glVertex4fv(Vertexi);
36 }
```



(a) Level 1 (b) Level 2 (c) Level 3 (d) Level 4

Figure 4.11: The different levels of crosshatching used by the project

Crosshatch Shading

The crosshatch shading methods implemented in this project will draw and shade the model given do it in a way resembling an artist drawing and shading an image with a pencil using crosshatching. The drawing aspect of the method has already been discussed as it is simply outlining and edge detection. It is the shading that is the interesting aspect of the method. To achieve a good pencil shading effect a way to have crosshatching to define the shadows on the model needs to be found.

This project uses textures to crosshatch the models. Allowing only a few textures with the different styles of crosshatching to be loaded and applied to the model. The way these textures are applied is similar to the cell shading example earlier. The aim is to have several different levels of darkness that are applied to the model to achieving a similar effect to cellshading (Figure 4.11).

To layer the levels of crosshatching in such a way as to shade a model the levels of the different areas of the model that need to be shaded must determined. This project examines two different ways of doing this, 'simple' crosshatching and 'fine' crosshatching.

'Simple' Crosshatching

In 'simple' crosshatching the model is shaded on a per triangle basis applying the appropriate level of crosshatching to each individual triangle. To determine which texture should be associated to any given triangle a check similar to the one used for cellshading is performed. In cellshading the cosine of the angle between the vector indicating the direction of the light source and the normal vector of that vertex was

calculated and then mapped onto a one dimensional texture to retrieve the shading value for that vertex. This allowed interpolation to perform the rest of the shading in the triangles. The problem with crosshatching is that interpolation cannot be used to determine the shading value for the rest of the triangle given that a texture is being applied to the triangle to shade it. Therefore, the most appropriate shading level to assign to the entire triangle must be found.

To classify a triangle the point at the centre (barycentre) of the triangle is used as a vertex for the lighting equation to get the value needed to classify the triangle and apply the appropriate texture to it. The following steps are performed to find out what the normal value at the centre of the triangle is:

Given points p1, p2 and p3 about a triangle the barycentre B calculatd by:

$$B = \begin{pmatrix} (p_1.x + p_2.x + p_3.x)/3 \\ (p_1.y + p_2.y + p_3.z)/3 \\ (p_1.z + p_2.z + p_3.z)/3 \end{pmatrix}$$

Given normals n_1, n_2 and n_3 on the three points of a triangle to get the normal of the barycentre N is calculated by:

$$N = \begin{pmatrix} n_1.x + n_2.x + n_3.x \\ n_1.y + n_2.y + n_3.y \\ n_1.z + n_2.z + n_3.z \end{pmatrix}$$

Having calculated the barycentre of the triangle and the normal at the barycentre, it is possible to calculate the category of the triangle by classifying the result of the following equation:

$$N \cdot (LightPosition - B) \tag{4.2}$$

The result of equation 4.2 is fed into a lookup table to determine the correct classification of the triangle. The triangle classified, the appropriate texture is applied to it.

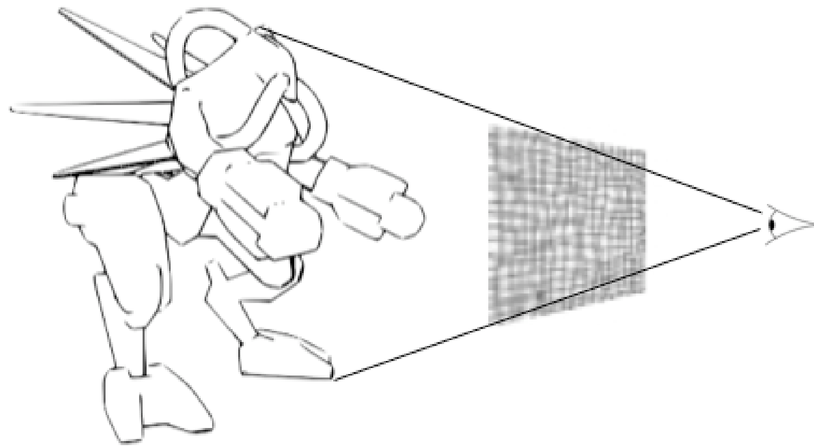


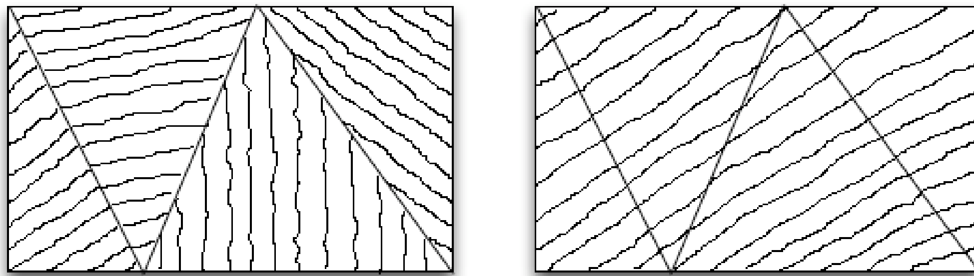
Figure 4.12: How the crosshatching is applied to the model

Texture Coordinate Generation

Having classified the triangles properly, a problem still remains: the texture coordinates need to be calculated in such a way that it appears that the artist crosshatched the entire shaded region rather than just crosshatching each triangle individually (see Figure 4.14). The idea is to project the texture onto the model in such a way that the shaded area is seen through a crosshatching screen making the entire region contiguous (Figure 4.12) . To do this the texture coordinates for each of the vertices need to be calculated in the same way that vertex coordinates are mapped onto the screen. Therefore, to calculate the texture coordinates the vertex coordinates need to be mapped onto the vertex's screen coordinates and then transformed into the square texture coordinates.

There is a function in OpenGL, which performs automatic texture coordinate generation called glTexGen which can take the parameter GL_EYE_LINEAR computing the texture coordinates in this way for us. Unfortunately this function does not deal very well with mapping small textures to larger models. It would have been possible to create bigger textures but that would have been a waste of space considering that crosshatching is a regular pattern and a small texture is sufficient. Therefore, to keep the textures small and have them display properly on the object this project needed to have its own texture generation function.

To create this function it was necessary to look at the transformation pipeline



(a) Triangles with badly generated texture coordinates

(b) Triangles with the texture coordinates generated properly

Figure 4.14: The difference between triangles with the texture coordinates badly applied and well applied

that maps a vertex defined in object space to a point on the screen. This pipeline can be seen in Figure 4.13. The projection performed by this pipeline can be done by calling the OpenGL utility function, gluProject. This function takes the two matrices, Model View and Projection as attributes as well as the OpenGL viewport (which defines the size of the window). The output of this function is in screen coordinates meaning that they need to be processed further. The point returned by gluProject is therefore taken and transformed it into texture coordinates by dividing the x value by a factor of the width of the window and the y value by a factor of the height of the screen[5].

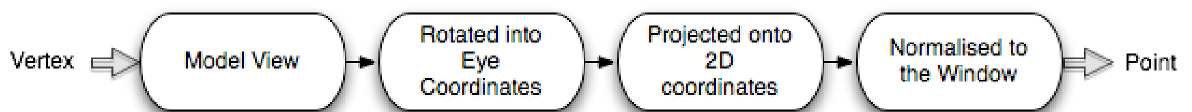
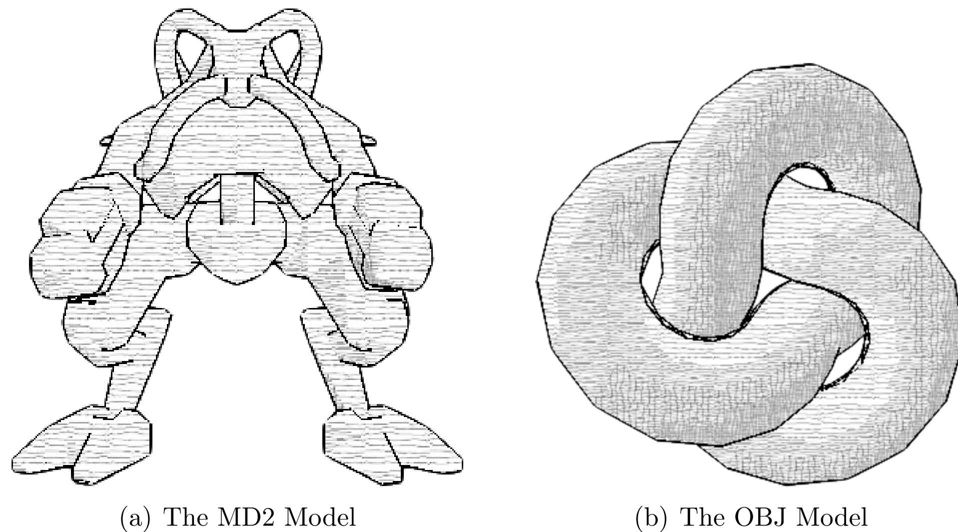


Figure 4.13: The Transformation Pipeline

[5] The size of the factor by which we divide the point by depends on the size of our texture.



(a) The MD2 Model (b) The OBJ Model

Figure 4.15: The result of performing 'Simple' crosshatching on the models

As seen in Figure 4.15, the simple crosshatching method works well on detailed models which have a high polygon count (Figure 4.15b). However, it produces a less polished effect on models where there is a low polygon count (Figure 4.15a) and the triangles start to become significant in size (compared to the model's size). This happens because in the latter case interpolation is very important and thus the approach to give each polygon its own shading level is too rough. This needs to be refined and a method similar to interpolation must performed so that one polygon can accommodate multiple shading levels. This is what is done in fine crosshatching.

'Fine' Crosshatching

Following the discussion about 'simple' crosshatching it is imperative to find a way to refine how crosshatch levels are assigned to different areas of the model. This essentially means that a way to interpolate the textures across the polygons must be found. However, as previously discussed, interpolation is not really possible with textures. The way to do this is to increase the number of polygons available to shade so that an effect similar to the one observed on highly detailed models when we applying 'simple' crosshatching can be obtained. This of course means subdividing the polygons to make it easier to approximate the shadow line (Figure 4.16). One could do this by splitting each triangle larger than a certain size into four smaller triangles by adding vertices at the midpoints of the triangles edges. This however

Algorithm 5 Generating the Texture coordinates using gluProject

```
1  /*Set up some temporary matrices to get the projection
      and model transformations*/
2  GLdouble tmpModelMat[16];
3  GLdouble tmpProjMat[16];
4  GLint    tmpViewVal[4];
5  glGetDoublev(GL_MODELVIEW_MATRIX, tmpModelMat);
6  glGetDoublev(GL_PROJECTION_MATRIX, tmpProjMat);
7  glGetIntegerv(GL_VIEWPORT, tmpViewVal);
8
9  /*set up some temporary floats for the values we're going
       to need to store for texture coordinates*/
10 GLdouble t11;
11 GLdouble t12;
12 GLdouble t13;
13
14 /*This command projects the triangle's coordinate points
      onto the window's coordinate system*/
15 gluProject(p1.GetX(), p1.GetY(), p1.GetZ(),
16            tmpModelMat, tmpProjMat, tmpViewVal,
17            &t11, &t12,&t13 );
18
19 /*seen as the window isn't square we need to remedy the
      stretching a little and so we divide the window
      coordinates by a factor of the total width and length.
      */
20 t11 = t11/factorOfScreenWidth;
21 t12 = t12/factorOfScreenHeight;
```

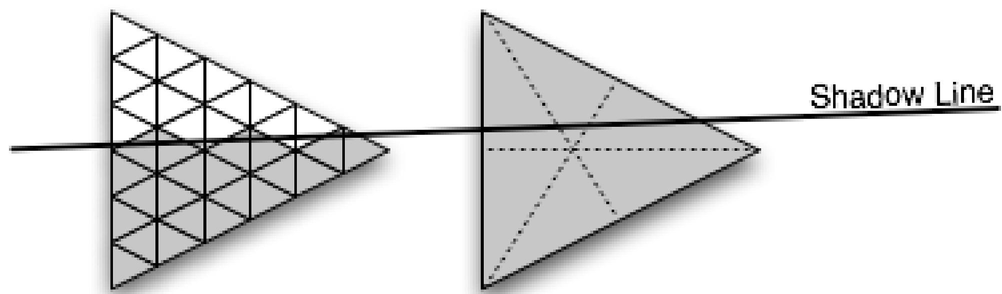


Figure 4.16: The effect of a shadow line on small triangles compared to one large triangle

would be very wasteful. Imagine that a large section of the model that resides in one shadow region every triangle in this region would be unnecessarily subdivided. A better way of approaching the problem would be to perform 'smart' subdivision.

'Smart' subdivision of triangles

In the simple subdivision method sketched out above one would simply take all the edges of the triangle split them in half and then, using the edge midpoints as new vertices, draw the four triangles that this would create – repeating the process on all four triangles if they were not small enough. What 'smart' subdivision seeks to do is; determine whether a triangle needs to be divided, check to find out where the shadow line crosses the edges of the triangle and then divide the triangle according to the intersection points found.

Checking to see which triangles need to be divided

To determine whether a triangle needs to be subdivided one need to determine when a shadowline passes through it. To tell whether a shadow line passes through a triangle the vertices of the triangle need to be checked to find their position in relation to the shadow line and determine if one of the vertices is on the other side of the shadow line. The shadow line however, is an imaginary construct marking the place where a point moves from one shading level[6] into another. Therefore

[6]Areas of a similiar shading level will be refferred to as shadow region from here on.

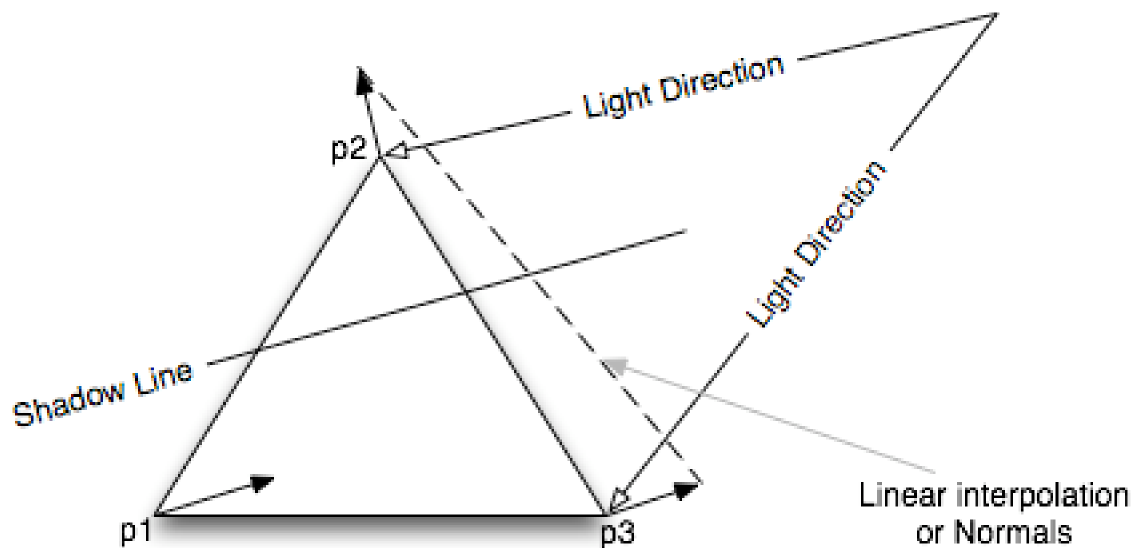


Figure 4.17: Linear interpolation of the normals to find out the point at which the shadow line crosses an edge

the lighting equations on each of the vertices need to be performed to determine which shadow region it is in. If all the vertices are in the same shadow region then the triangle need not be subdivided. If however, one of the vertices is in a different shadow region the triangle must be divided. The triangle will also need to be divided if all three vertices are in different shadow regions but that is a special case and will be handled separately.

Dividing the triangles with one vertex in a different shadow region

Given a triangle with vertices p_1, p_2 and p_3 such that p_2 is in a different shadow region to p_1 and p_3. To be able to find out where the shadow line[7] crosses the edges the normals must be interpolated linearly along the edges and so that the lighting calculations can be performed to find the point where the shadow level changes(Figure 4.17). The naive method to do this would be to walk along the edge and test the lighting equation at each step and then check to see if the threshold has been crossed. This however, is time consuming and prone to error. A better way of doing this is to calculate the value directly. Fortunately as the normals are

[7] If there are two shadow lines crossing the triangle (i.e. p_2 is two shadow levels away from p_1 and p_3) we construct a shadow line between the two.

interpolated linearly this can be done easily using linear algebra.

Given the Lighting equation:

$$\hat{n} \cdot \hat{l} = |\hat{n}||\hat{l}| \cos\theta = \cos\theta \quad \because |\hat{n}| = 1 \,\&\, |\hat{l}| = 1 \tag{4.3}$$

Any normal on the edge between the two vertices will be a weighted average of the two normals (from here on it is assumed that all the vectors are normalised):

$$n_i = \lambda_2 n_2 + (1 - \lambda_2) n_3 \; where \, 0 \leq \lambda \leq 1 \tag{4.4}$$

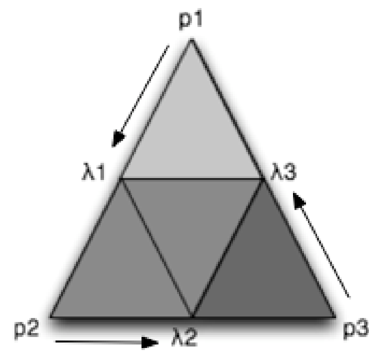
Now that the normal at any point on the line can be calculated, the lighting equation can be performed for any point on the line given the two normals at the end points of the line – assuming that the light is sufficiently far away so that the light direction l is constant:

$$\begin{aligned} \cos\theta &= n_i \cdot l \\ \cos\theta &= (\lambda_2 n_2 + (1 - \lambda_2) n_3) \cdot l \\ \cos\theta &= \lambda_2 (n_2 \cdot l) + (1 - \lambda_2)(n_3 \cdot l) \end{aligned} \tag{4.5}$$

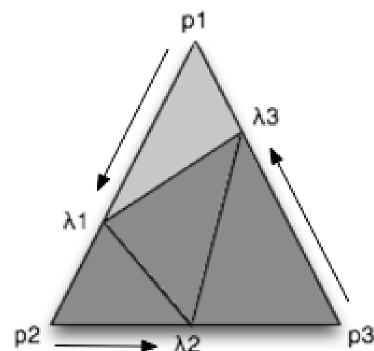
The point of intersection with the shadow line is at the place where value of $\cos\theta$ moves from one shadow level into another and thus crosses a threshold τ. To calculate the offset λ_2 to the intersection point – given $\tau = \cos\theta$ – the equation 4.5 needs to be arranged to:

$$\frac{\tau - n_3 \cdot l}{(n_2 \cdot l) - (n_3 \cdot l)} = \lambda_2 \quad where \, \tau = \cos\theta \tag{4.6}$$

This equation gives λ_2 which is an offset of the vertex p_2. To find out which point corresponds to which offset lambda see figure 4.18. Given the offsets λ_1, λ_2 and λ_3 and the points p_1, p_2 and p_3 the points at which to split the triangle can be calculated. It is important to note that threshold$_\lambda$ values between the two points in the same shadow regions need not be calculated because the three triangles on the same side of the shadow line will all be assigned the same texture and can thus simply be set to 0.5.



(a) When all three vertices are in different shadow regions



(b) When only one vertex is in a different shadow region and the shadow line passes through λ_1 and λ_3

Figure 4.18: The splitting of a triangle and the how each split triangle is assigned a texture

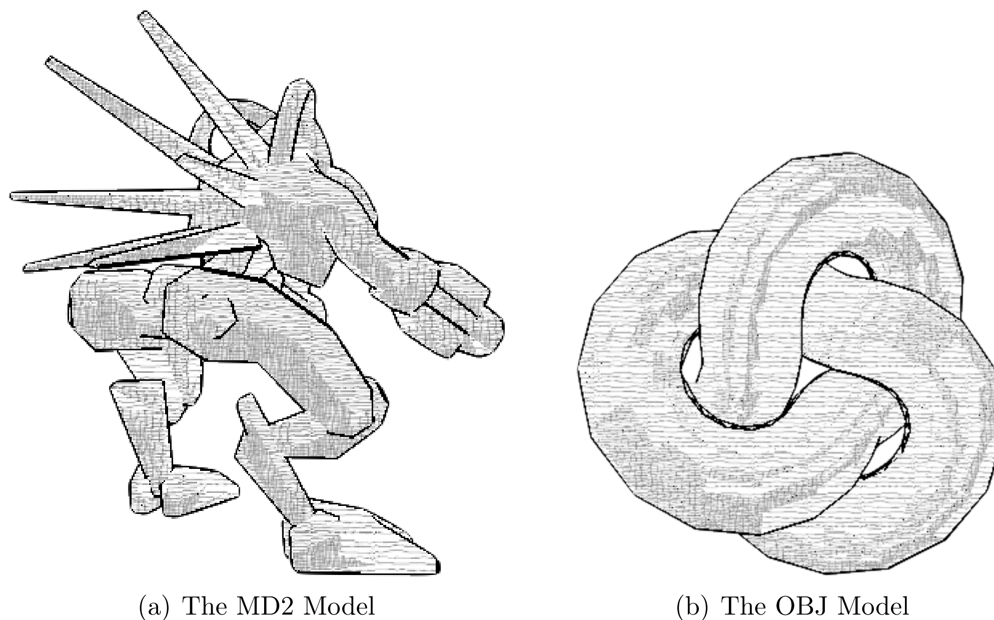
Dividing the triangles with all three vertices in different shadow regions.

Given a situation where all three vertices are in different shadow regions the algorithm in this project will give up and perform a naive split of the triangles. That is to say that the three offset values are set to 0.5 and the three outer triangles defined by this split are assigned the appropriate textures for the shadow regions which their respective original points were in. The middle triangle is randomly assigned to have the same texture as the texture for the triangle containing point p_2 (Figure 4.18a).

Figure 4.19 shows that the fine crosshatching method works better than the 'simple' on models with few polygons. With highly detailed models such as the OBJ model we seem to run into problems because our algorithm subdivides triangles that don't necessarily need to be subdivided. A reasonable addition to 'fine' crosshatching would be to perform a check whether the triangle is big enough compares to the model's overall size to be worth dividing. If it is not, then we can simply allow the triangle to be shaded using the 'simple' crosshatching method.

Algorithm 6 Checking to see if a triangle needs to be divided.

```
1  bool checkTriangle(
2          Vector3D nor1, Vector3D nor2, Vector3D nor3,
3          Vector3D pos1, Vector3D pos2, Vector3D pos3){
4
5   for(all three vertices posi){
6     lightVector = posi - lightPos;
7     lightVector.Normalise;
8     angle = nori.DotProduct(lightVector);
9     posiType = Classify(angle);
10  }
11
12  /*if all the angles are in the same range return false
       as we don't need to split*/
13  if(all posiTypes equal) return false
14
15  /*check to see if only all normals are of different type
       .*/
16  if(all posiTypes distinct){
17   lambda.i = 0.5; //where i = 1..3
18   return true;
19  }
20  else{
21  /*check to see if just one vertex is in a different
       shadow region*/
22   if(posiType different from posjType=poskType){
23    /*Find out how it differs to give appropriate
         threshold to the mu calculation function. If the
         types are more than one shadow region apart we set
         the threshold to a value between the lower and
         upper bounds of the two thresholds
         p1-----p2--|----------|t----------|---p3*/
24     threshhold = getThreshold(posiType, posjType);
25     /*Calculate the first offset with equation 6*/
26     mu = calcMu(threshhold, nor3, nor2, pos3, pos2,
          lightPos);
27     lambda.i = mu;
28     /*Calculate the second offset with equation 6*/
29     threshhold = getThreshold(angle1Type, angle3Type,
          rangelimit);     mu = calcMu(threshhold, nor3,
          nor1, pos3, pos1, lightPos);         lambda.j = mu
          ;
30     /*Set the third*/
31     lambda.k = 0.5;
32     return true;
33   }
34  }
35
36 }
```



(a) The MD2 Model (b) The OBJ Model

Figure 4.19: The effect of 'Fine' Crosshatching

Conclusions

In this chapter three different ways of outlining a model and three different ways of shading a model were discussed. Whilst these methods are interesting from a theoretical perspective their actual performance is also important. These techniques have been tested on a Macbook Core 2 Duo 2.0Ghz with the Intel GMA 950 integrated graphics card. The aims were twofold: one, to test how quickly the models can be rendered, two, to test the degree of performance impact the different techniques have on the base rendering[8]. The models used for testing were the MD2 and OBJ models shown throughout the report. The MD2 model contains 1224 triangles and the OBJ model 5000.

Table 4.1 shows that the results agree with intuition. Where more calculations had to be performed the frame rate dropped in comparison to the base rendering. When using the Stencil buffer the main performance impact came from having to render the model twice as well as updating the stencil buffer. Front-face culling had a much smaller performance impact, this is due to the way OpenGL is optimised, as the face culling was probably performed on the graphics hardware in contrast to

[8]The base rendering is simply the rendering of the model in its simplest form, that is to say without any lighting or outlining, just drawing the triangles in a certain color.

Rendering effect	FPS at 1224 triangles	FPS at 5000 triangles
Base Rendering	60	49
Outlining with the Stencil Buffer	54	41
Outlining using Front-Face Culling	57	43
Gourad Shading	59	48
Cell Shading	57	46
'Simple' Crosshatch Shading	50	36
'Fine' Crosshatch Shading	24	12

Table 4.1: The speed in frames per second (FPS) of the different rendering styles on two different models

the stencil buffer which was maintained and updated by the CPU. The Cell Shading performed very well despite the fact that the lighting calculations were performed in software rather than on the graphics hardware. The two crosshatch shading examples fared less well. This is probably due to the switching of textures needed to apply to the triangles and also performing the texture coordinate generation. The 'fine' crosshatching method also had the added burden of calculating the points whether and where the triangle would need to be split and generating more triangles (up to 4 times as many).

Finally, the rendering techniques performed well given the graphics hardware available. All the animations appeared smooth except for perhaps the 'fine' crosshatching example on the 5000 triangle model. With a longer time frame for optimising the code, it should be possible to significantly reduce the number of instructions necessary to perform the 'fine' crosshatching and equalise its performance with those of other rendering styles.

Chapter 5

Helper Functions for the Program

When developing and implementing the non-photorealistic rendering techniques from the previous chapter these helper functions were implemented to assist with the analysis of the successful rendering techniques.

The Magnifying Glass

This project's development necessitated a way to examine the output rendered by the presented techniques in detail. A magnifying glass was therefore included in the program to allow for a 1.5x magnification of the rendered model.

The magnifying glass method allows a portion of the rendered screen to be enlarged and the enlargement area to be dynamically moved around the screen. Texturing was used to perform this task as seen that it was the image rendered to the screen that needed to be examined and not another, larger, rendering of the model in a small area.

Whenever magnification is turned on the following actions were performed:

1. Capture the mouse position.

2. Look into the screen buffer on video memory and copy a 128x128 pixel square centred on the mouse position into a texture prepared for it.

3. Render a sphere centred at the mouse position in the model world in front of the model but still visible.

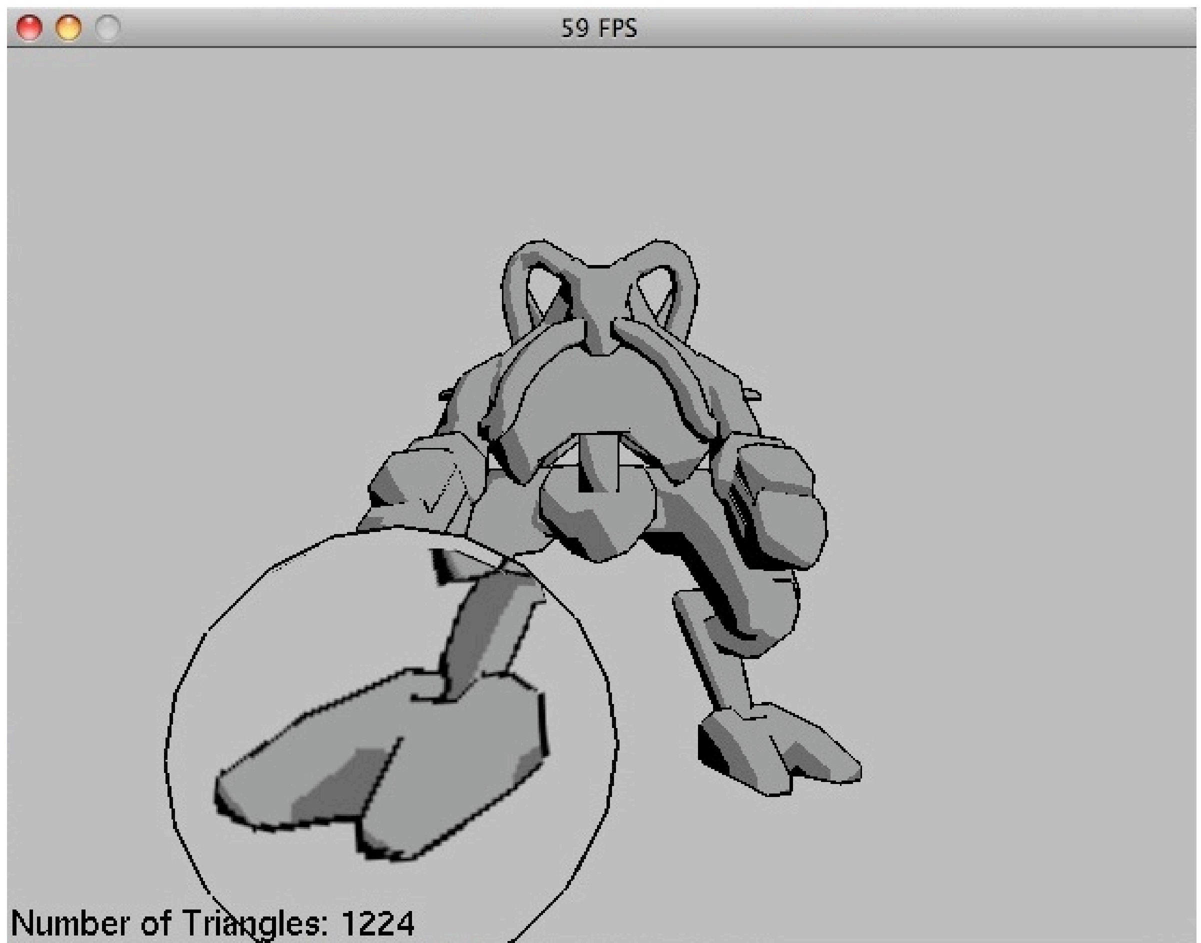


Figure 5.1: Magnifying effect

4. This sphere is then textured by the 128x128 pixel square captured earlier. Using glTexGen with the GL_SPHERE_MAP attribute.

This method works extremely well (Figure 5.1). However, it is important to note that this method relies on the order of the drawing pipeline for its effect. It must execute after the scene has been drawn entirely to the screen buffer and before the screen buffer is flushed to the screen. If this order is changed then the method will break. Computationally, it is very inexpensive; the most expensive step being the generation of the texture coordinates for the sphere. Copying the pixels to a texture involves only a bit copy operation from one part of memory to another which is extremely fast given that the source and the destination will be located very close to each other (either both on video memory or both in the system).

Algorithm 7 The Magnifying Glass

```
1  //1
2  /*Get the mouse position*/
3  GetMousePos(&mouseX,mouseY);
4
5  //2
6  /*Load up the texture that we will be using*/
7  glBindTexture(GL_TEXTURE_2D, params.lens[0]);
8  /*Copy the 128x128 square into the loaded texture*/
9  glCopyTexImage2D(GL_TEXTURE_2D,0,
10                  GL_RGB,mouseX,
11                  winHeight-mouseY,
12                  128,128,0);
13 /*Enable texturing*/
14 glEnable(GL_TEXTURE_2D);
15 /*Set the texture attributes for the newly 'created'
      texture*/
16 glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_REPLACE);
17 glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
18 glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
19 glEnable(GL_TEXTURE_GEN_T);
20 glEnable(GL_TEXTURE_GEN_S);
21
22
23 /*Get the model view Projection and viewport values to be
       able to place the sphere properly*/
24 glGetDoublev(GL_MODELVIEW_MATRIX, ModelMatrix);
25 glGetDoublev(GL_PROJECTION_MATRIX, ProjMatrix);
26 glGetIntegerv(GL_VIEWPORT, ViewPortValues);
27
28 /*Get the sphere's x and y positions given the mouse
      positions and a theoretical z value*/
29 gluUnProject(mouseX,mouseY,0.9,
30              ModelMatrix,
31              ProjMatrix,
32              ViewPortValues,
33              &x,&y,&z);
34
35 //3
36 /*Place the sphere centered at the mouse coordinates in
      the model world in front of the model*/
37 glTranslatef(x,-y,-1.5f);
38
39 //4
40 gluSphere(sphere, 0.3, 20,10);
```

Manipulating the Object using a Trackball

To be able to move the object around and manipulate it with the mouse a trackball system was implemented. A trackball system simulates a sphere around the model. When the screen is clicked the trackball system will find the point on the simulated sphere that has been clicked and rotate the model according to the mouse movements on this sphere (Figure 5.2). Given the position (x, y) on the screen the system will project this onto the virtual sphere to give coordinates (x, y, z) on the surface of the sphere. This will allow the user to 'grab' the environment with the mouse and move it around to see the rendered model from different orientations.

When the mouse is moved its projected position (x', y', z') is recorded and the model is rotated about the rotation axis, given by the cross-product of the coordinate vectors (x, y, z) and (x', y', z'), and the angle θ.

$$\cos\theta = \frac{x_1 \cdot x_2}{|x_1||x_2|} \quad where \;\; x_1 = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \;\&\; x_2 = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}$$

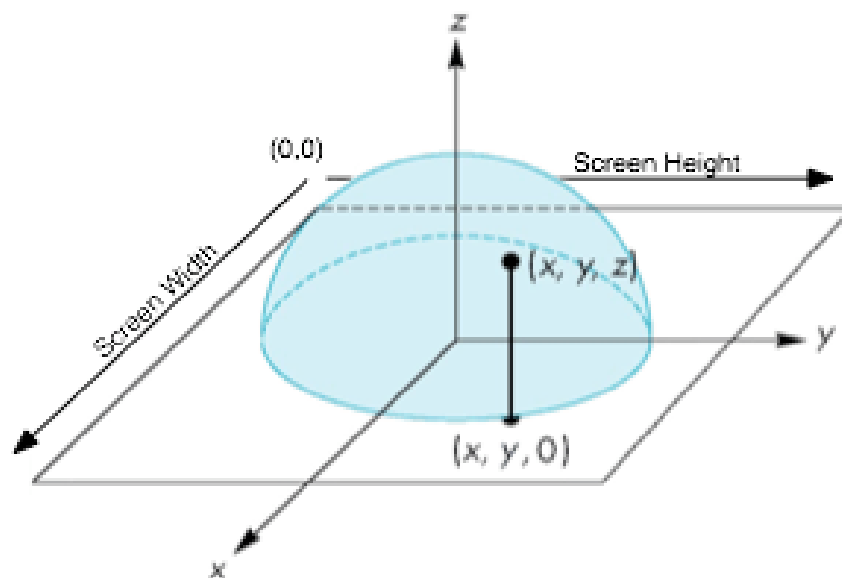


Figure 5.2: How a trackball system works where the Mouse is clicked at $(x,\ y,\ 0)$ and this is linked to the coordinate $(x,\ y,\ z)$.

Loading Models

To demonstrate different rendering methods and compare their performance it was essential for this project to include some way of loading models that could be displayed using these rendering methods. It was decided that this project would implement two different model loading systems, one for models in id software's MD2 format and one for models in the Wavefront OBJ format. These two model formats were chosen mainly because they are simple enough to parse easily and yet complex enough to be able to hold interesting models that would help in the testing of the rendering methods.

Due to the different formats in which the models were stored, it was decided to select the OBJ format for storing models meaning that when loading the MD2 models it would be imperative to convert the model into an OBJ format so that the program could use it. The OBJ model file on the other hand was parsed and then directly fed into the model object in the system.

Chapter 6

Conclusions

Overall Conclusion

This project has presented and analysed a number of the existing real-time non-photorealistic rendering techniques. Research for this project yielded that, while real-time non-photorealistic rendering is a nascent field, it is certainly one that is growing at a tremendous rate. It was therefore decided to only examine a few outlining and shading techniques and develop one of the methods past established research.

The first thing examined in this project was the theory behind finding edges. Three different methods of drawing edges were then presented – stencilling, front-face culling and ink sketching. Stencilling is a method which only draws the boundary outlines and uses masks instead of the edge finding theory presented. Front-face culling moves on from stencilling taking its masking method and integrating it with the edge detection theory outlined at the outset. Ink sketch builds further on front-face culling and draws the outlines in a more non-photorealistic way, drawing the edges several times with their endpoints offset by a small random amount.

The next section of the project presented cellshading and 'simple' crosshatching, proposing an improvement to the latter called 'fine' crosshatching. The cellshading method used 1D textures and interpolation to make the shading of the model discrete. 'Simple' crosshatching took this discrete view of the shading model and used textures to represent the different shading levels. 'Fine' crosshatching took this

idea further and increased the quality of the shading on models with a low polygon count.

Learning Outcomes

This project has been a learning experience teaching me about time management and the pitfalls one can encounter during research. Determining a road map was probably the hardest task of the project. It took a huge amount of time to determine the direction of the project. Changing the direction of my project in middle of it all did not help but it did allow me to focus on an incredibly interesting field that I have now become passionate about. Once the direction was chosen the work went smoothly except for the fact that the platform had to be changed due to compatibility issues. Also, because this is such an interesting field, it was quite difficult to choose which methods to focus on and which ones to leave aside for another day. In all, I have greatly enjoyed working on this project and I am looking forward to seeing the new techniques that will be developed in real time non-photorealistic rendering.

Future Work

Given more time to delve further into this project (and more space for the write up) it would have been interesting to extend the proposed improvements to the crosshatch shading model to make it more efficient and more accurate when dealing with detailed models[1].

One improvement would be to speed up the rendering process by using triangle strips. This would mean that there wouldn't be as many switches in the texture selection thus improving the speed of the process significantly. It would also greatly reduce the amount of vertices that need to be passed to the graphics hardware. This could be implemented by performing a certain amount of pre-processing at each frame and grouping adjacent triangles in the same shadow region together in such a way that it would be possible to send sets of triangles to the graphics hardware,

[1] Models with a high polygon count.

without having to treat each triangle individually when drawing and texturing.

A second improvement would be to find a way to ensure that detailed models are not adversely affected by the proposed method. This could be implemented by a system that would use some a heuristic to measure the size (in relation to the model) and the shape of triangles (on the screen[2]) and judge whether a split should be attempted. This would also provide a viable solution to the problem where the three vertices of the triangle are in different shadow regions as it would allow repeated splitting of triangles with a robust stopping case.

[2]This would not necessarily mean that the triangles would need to be projected to screen coordinates. We could determine approximate shapes by performing the calculation in equation 4.1 on each of the triangle vertices and using the values to calculate the distortion.

Bibliography

[1] OpenGL tutorials - http://nehe.gamedev.net.

[2] http://www.gamedev.net/community/forums/topic.asp?topic_id=476852, January 2008.

[3] Non-photorealistic Quake - http://www.cs.wisc.edu/graphics/gallery/nprquake/sketchynpr. January 2008.

[4] OpenGL and SDL - http://osdl.sourcefourge.net/main/documentation/rendering/sdl-opengl.html, April 2008.

[5] Wavefront and Java3D .obj format - http://www.eg-models.de/formats/format_obj.html, March 2008.

[6] Ed Angel. Virtual trackball. Lecture.

[7] Adam Lake Carl Marshall Mark Harris Marc Blackstein. Stylized rendering techniques for scalable real-time 3d animation. Technical report, Intel Architecture Labs, 2000.

[8] Jonathan Cohen. A non-photorealistic lighting model for automatic technical illustration. Lecture.

[9] Calvin College. Geometric objects and transformations. Lecture.

[10] Apple Corporation. OpenGL Programming Guide for Mac OS X - http://developer.apple.com/documentation/graphicsimaging/conceptual/opengl-macprogguide/index.html, January 2008.

[11] Kyunghyun Yoon Daeuk Kang, Donghwan Kim. A study on the real-time toon rendering for 3D geometry model. Master's thesis, Graduate School of Advanced Imaging Science, Multimedia and Film, 2001.

[12] Dave Shreiner Mason Woo Jackie Neider Tom Davis. *OpenGL Programming Guide: Fourth Edition*. Addison Wesley, 2004.

[13] Philippe Decaudin. Cartoon-looking rendering of 3D-scenes. Technical report, INRIA, June 1996.

[14] David Luebke Derek Cornish, Andrea Rowan. View-dependent particles for interactive non-photorealistic rendering. Technical report, Proceedings of Graphics Interface, June 2001.

[15] Gooch. *Non-Photorealistic Rendering*. AK Peters, Ltd, 2001.

[16] Aaron Hertzmann. Introduction to 3D non-photorealistic rendering: Silhouettes and outlines. Technical report, New York University, 2000.

[17] Seungyong Lee Hyunjun Lee, Sungtae Kwon. Real-time pencil rendering. Technical report, Pohang Institute of Science and Technology, 2006.

[18] Vishvananda Ishaya. Real-time cartoon rendering with Direct-X 8 hardware - http://www.gamedev.net/reference/programming/features/cartoon/, April 2008.

[19] David B. Horvath Jesse Liberty. *Sams Teach Yourself C++*. Sams, 4th edition, 2005.

[20] Jeff Kershner. Object-oriented scene management.

[21] Jeff Lander. Shades of disney: Opaquing a 3D world. *Game Developper*, March 2000.

[22] Michael Kowalski Lee Markosian. Real-time nonphotorealistic rendering. Technical report, Brown University, 1997.

[23] Harry Plantinga. Computer graphics. Lecture.

[24] Nicholas Haemel Richard S. Wright, Benjamin Lipchak. *OpenGL SuperBible: Fourth Edition*. Addison Wesley, 2007.

[25] J.D. Sullivan Robert P. Kuehne. *OpenGL Programming on Mac OS X*. Addison Wesley, 2008.

[26] Nicolas Roussel. Placage de texture avancé – Advanced texture placement. Lecture.

[27] Thomas Strothotte Stephan Schlectweg. *Non-Photorealistic Computer Graphics*. Morgan Kaufmann, 2002.

[28] Allen Sherrod. *Ultimate 3D Game Engine Design and Architecture*. Charles River Media, 2007.

[29] Oliver Düvel Stefan Zerbst. *3D Game Engine Programming*. Thomson Course Technology, 2004.

[30] James D. Foley Andries van Dam Steven K. Feiner John F. Hughes. *Computer Graphics: Practices and Principles*. Addison Wesley, 1990.

[31] Sung-Eui Yoon. Interacting with a 3D world. Lecture.

List of Figures

1.1 A Simple example of the Computer Graphics pipeline 3
1.2 The pipeline for a face renderer 5

2.1 This is the schema for the way the system renders the models non-photorealistically and the different options available. 11

3.1 The initial loading of the model in wireframe. 13
3.2 The cellshaded models. 13
3.3 The cellshaded models with ink sketched outlines. 14
3.4 The models rendered with traditional photorealistic gourad shading. . 14
3.5 The Models rendered with gourad shading and a metallic texture. . . 15
3.6 The models rendered with crosshatch textures for shading, 'simple'. . 15
3.7 The models rendered with crosshatch textures for shading, 'fine'. . . . 16
3.8 The Help Screen 16
3.9 The Magnifying Glass 17

4.1 Edge detection in Model space 19
4.2 The states of the buffer and the screen after step 3 20
4.3 The output of the stencil buffer outlining code 21
4.4 The Front Face Culling technique for drawing inner and boundary lines 23
4.5 The ink sketch outlines 27
4.6 Illustration of how the ink sketch outliner would draw a line 27
4.7 The difference between the continuous and discreet shading of a triangle. 30

4.8 Lighting angles: n is the normal to a point on the model surface, l is the vector determining the direction from the light source to the point and β is the angle of incidence of the light to the point. 31

4.9 The difference between a texture being clamped and repeated. This checkerboard has been clamped in the x axis and repeated in the y axis. (Image taken from [12]) 32

4.10 Cell shaded triangle and its corresponding 1D texture 33

4.11 The different levels of crosshatching used by the project 35

4.12 How the crosshatching is applied to the model 37

4.14 The difference between triangles with the texture coordinates badly applied and well applied 38

4.13 The Transformation Pipeline 38

4.15 The result of performing ‘Simple’ crosshatching on the models 39

4.16 The effect of a shadow line on small triangles compared to one large triangle 41

4.17 Linear interpolation of the normals to find out the point at which the shadow line crosses an edge 42

4.18 The splitting of a triangle and the how each split triangle is assigned a texture 44

4.19 The effect of ’Fine’ Crosshatching 46

5.1 Magnifying effect 49

5.2 How a trackball system works where the Mouse is clicked at $(x, y, 0)$ and this is linked to the coordinate (x, y, z). 51

List of Algorithms

1 The OpenGL code for rendering an outline using the Stencil Buffer . 22

2 The OpenGL code for rendering boudary outlines and inner outlines using front-face culling 25

3 The ink sketch algorithm 29

4 The OpenGL implementation of the Cell Shading algorithm 34

5 Generating the Texture coordinates using gluProject 40

6 Checking to see if a triangle needs to be divided. 45

7 The Magnifying Glass 50