A Manga Styled Implementation of Non-photorealistic Rendering

Killian McCabe
B.A. (Mod.) Computer Science
Final Year Project April 2015
Supervisor: Dr. John Dingliana

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

______________________________
Killian McCabe, April 22, 2015
PERMISSION TO LEND

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

Killian McCabe, April 22, 2015
ACKNOWLEDGEMENTS

I would like to thank my supervisor, John Dingliana. Without his guidance I would have been quite lost in my endeavours. And I’m grateful for his help in selecting a research topic that I thoroughly enjoyed working on.

I would also like to thank my family and friends for their continuous support throughout my degree.
In this project we explore an alternative to realistic 3D graphics programming. Realism is not always the best method for artistic expression and can often hinder how ideas and emotions are received.

Japanese manga is concise in how it conveys emotion and action. For this reason we will attempt to procedurally generate real time graphics that replicate the style of Japanese manga and describe motion through similar methods used in manga.

The use of several non-photorealistic techniques such as cel shading, hatching and outlining will be explored on static images. We will also compare the use of image processing and primitive generation to create motion cues applicable to animation.
CONTENTS

i INTRODUCTION AND BACKGROUND 10
1 INTRODUCTION 11
  1.1 Objectives 11
  1.2 Possible Uses 11
  1.3 Report Layout 12
2 BACKGROUND 13
  2.1 Non-Photorealistic Rendering 13
    2.1.1 Cel Shading 13
    2.1.2 Cross Hatching 13
  2.2 Motion Cues 13
  2.3 Graphics Processing Unit 14
  2.4 Graphics Hardware Pipeline 14
    2.4.1 Shader Programmes 14
  2.5 Transformation Pipeline 16
  2.6 Technologies Involved 17
    2.6.1 OpenGL 17
    2.6.2 Blender 17

ii IMPLEMENTATION 19
3 SET UP 20
  3.1 Initializing and Drawing Model Data 20
4 IMPLEMENTATION 22
  4.1 Overview 22
  4.2 Style 22
    4.2.1 Cel shading 22
    4.2.2 Texture Sampling 22
  4.3 Drawing Outlines 26
  4.4 A Note on Rendering Good Edges 30
  4.5 Scene Motion 30
    4.5.1 Background Motion 30
    4.5.2 Camera Motion 30
  4.6 Object Motion 33
    4.6.1 Silhouette Fading 33
    4.6.2 Motion Lines 33
  4.7 Notes on Combining Features 38

iii FURTHER WORK AND CONCLUSION 40
5 FURTHER WORK 41
  5.1 Improvements on Implemented Features 41
    5.1.1 Background Motion Lines 41
    5.1.2 Drawing Outlines 41


<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2 Additional Features</td>
<td>42</td>
</tr>
<tr>
<td>5.3 Improved Scenes</td>
<td>42</td>
</tr>
<tr>
<td>6 CONCLUSION</td>
<td>43</td>
</tr>
<tr>
<td>iv APPENDIX</td>
<td>44</td>
</tr>
<tr>
<td>7 ELECTRONIC SOURCES AND RESOURCES</td>
<td>45</td>
</tr>
<tr>
<td>8 REFERENCES</td>
<td>46</td>
</tr>
<tr>
<td>9 GLOSSARY</td>
<td>48</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1  Hardware pipeline, Anton Gerdelan’s “Anton’s OpenGL 4 Tutorials” 15
Figure 2  3D transformation pipeline, Anton Gerdelan’s “Anton’s OpenGL 4 Tutorials” 18
Figure 3  Texture mapping performed on a cube 20
Figure 4  Cel shading output 23
Figure 5  Phong shading output 24
Figure 6  Output of styles recreated with texture sampling 25
Figure 7  Depth map 27
Figure 8  Depth map edges 27
Figure 9  Normal map 28
Figure 10  Normal map edges 28
Figure 11  Complete output of object outline 29
Figure 12  Sharpening edges in Blender 31
Figure 13  Background motion output 32
Figure 14  Camera motion output 32
Figure 15  Silhouette fading object motion 33
Figure 16  Object motion lines attempt 1 34
Figure 17  Object motion lines attempt 2 35
Figure 18  Object motion lines on bird model with different levels of tesselation, (a) input, (b) low tesselation, (c) higher tesselation 36
Figure 19  Creation of the wing texture in Blender 37
Figure 20  Complete object motion lines output 38
Figure 21  Combining background motion, tone shading and object outline 39
Figure 22  Combining background motion, cel shading and object motion 39
ACRONYMS

**CPU**  Central Processing Unit

**GPU**  Graphics Processing Unit

**NPR**  Non-Photorealistic Rendering

**OpenGL**  Open Graphics Library

**2D**  Two Dimensional

**3D**  Three Dimensional
Part I

INTRODUCTION AND BACKGROUND
INTRODUCTION

In this project we attempt to procedurally generate graphics that reproduce several styles and effects seen in manga.

Manga is a particular art style used in Japanese comics. It is typically drawn in black and white with patterns used to compensate for lack of colours. Hatching and toning are two common patterns. We will recreate these using with non-photorealistic rendering (NPR) techniques.

Manga also has a unique way of capturing motion. Though it is limited by the medium of ink on paper there is a clear sense of movement within the picture. This is because manga artists draw trails of motion on objects that should appear to be moving. This is then emphasised so that both the direction and speed of a moving object are displayed with a strong impression. We will render similar motion cues with image processing techniques and primitive generation.

1.1 OBJECTIVES

With this project we intend to do the following:

1. Create real time 3D graphics on the GPU with OpenGL.

2. Replicate the style and techniques of Japanese Manga.

3. Enhance the users’ understanding of a scene with simple artwork.

4. Render motion cues similar to those used in manga with a method suitable for 3D programs.

The ideal result would be to improve the aesthetic appeal of the scene without losing vital information (e.g. object movement, character faces).

1.2 POSSIBLE USES

This project could be further developed into a useful tool for creating manga. Manga artists usually work with a small team to produce new chapters weekly. It is in the artists favour to automate some aspects of their work. For example, background objects such as cars
and buildings have to be redrawn often in manga. Using 3D modelling software, this inefficiency can be eliminated. The object would be created as a model which is then passed into a specialized rendering program with particular view parameters to produce a finished manga panel. Progress is already being made to simplify the manga creation process (e.g. manga colourization [1]).

Manga is frequently adapted for anime which is a style of animation originating in Japan usually produced as TV series or films. Popular anime are then sometimes adapted into video games. In this project we examine the possibility of creating and using real time manga to help create a video game straight from a manga series. Video game creation is also relevant to this project as NPR has become especially popular in video games recently [2]. For example, the video game XIII used cel shading to create a comic book style game which is similar to what we attempt in this project.

1.3 REPORT LAYOUT

In **BACKGROUND** we introduce some related abstract ideas and the core concepts that this project builds upon. We provide some knowledge about 3D graphics programming that is needed to understand this project report.

Then in **SET UP** we discuss necessary actions in setting up 3D models, textures and shaders for rendering.

**IMPLEMENTATION** is an in-depth walk-through of how we recreated several features of manga.

In **FURTHER WORK** and **CONCLUSION** we measure the success of this project and talk about possible future improvements.

Lastly there is a **GLOSSARY** in which we’ve described some necessary graphical terms used throughout this report.
BACKGROUND

2.1 NON-PHOTOREALISTIC RENDERING

Traditionally realism has been the aim of computer graphics. But a realistically rendered image often has more information than necessary and does not always convey what the user needs or what the artist is trying to communicate. For example, a builder may simply want to know the dimensions of an object in the photo. Or an artist may want to convey the atmosphere of a scene. NPR presents a growing alternative where this is possible. For practical uses, NPR considers rendering objects as blueprints or cross-sections. And for an artists needs, it allows them to draw a scene in the style of paint, pencil sketches or watercolour.

2.1.1 Cel Shading

Cel shading, also called Toon shading, is a very common NPR technique. It has been applied to a large number of video games and is probably the simplest NPR technique to implement. There are plenty of simple tutorials on-line about how to do cel shading [4].

2.1.2 Cross Hatching

Cross hatching displays lighting as a series of brush strokes where the lines become thicker and cross over each other more to display darker areas. This is a common NPR technique that I learned from an on-line tutorial [5]. From this I was also able to implement screen-tone shading.

2.2 MOTION CUES

In realistic renderers motion blur is applied to simulate the effect of a scene moving too fast for a camera’s shutter. Redrawing an object multiple times is inefficient so alternative methods have been devised. One such method is to draw the scene, calculate the motion of each fragment and stretch that fragment’s colour over its vector of motion [6] [7].
But this doesn’t help the viewer understand the motion of each object. Instead we can use motion cues to help convey the actions in a scene by giving visual aids to describe a history of activity. In manga and cartoons, motion is intensified by drawing lines onto an object that stretch towards the direction they’ve moved from. Alternatively the outline of an object is redrawn from previous positions to create the effect of a fast moving object. Recreating these effects has been attempted before in 2D with computer vision [3]. In their case, all the information was coming from video and they were limited by working in only 2D. We can do more with the third dimension.

2.3 GRAPHICS PROCESSING UNIT

The GPU is a specialised piece of hardware that excels in producing images. Their ability to create images is accelerated by the amount of parallelism they can achieve. They are far more efficient at image processing than the CPU because of their highly parallel structure and effectiveness in processing large amounts of data.

2.4 GRAPHICS HARDWARE PIPELINE

There are five programmable stages in the OpenGL 4 pipeline. They are the vertex shader, the two tessellation shaders, the geometry shader and the fragment shader. Their order and function is outlined in Figure 1. Each one is also described in the next subsection. The vertex shader and fragment shader are necessary for objects to appear on screen. The other 3 shaders are generally left alone unless needed to implement advanced features.

2.4.1 Shader Programmes

Shader programmes are used to describe to the GPU how an object should be drawn (i.e. where to place each triangle on screen and how to colour them).

1 - VERTEX SHADER

The first shader in the graphics hardware pipeline, the vertex shader, is used to position model vertices. This handles object movement and animation.

2 - TESSELLATION CONTROL_SHADER (OPTIONAL)

Here triangles are subdivided to allow for higher levels of detail on an object. This stage essentially just controls the number of new polygons to be created. This is often used for generating more detailed terrain close to the camera.
Figure 1: Hardware pipeline, Anton Gerdelan’s “Anton’s OpenGL 4 Tutorials”
3 - TESSELATION EVALUATION SHADER (OPTIONAL)
Here any triangles created in the previous stage are positioned.

4 - GEOMETRY SHADER (OPTIONAL)
This is used to generate new primitives (e.g. points, lines, triangles). Often used in creating and animating grass or fur. After this the built in stages Clipping and Rasterisation occur.

CLIPPING (BUILT-IN, NON-PROGRAMMABLE)
Calculates which triangles appear on screen and removes any from the to-draw list if they are out of frame. This stage is done for efficiency.

RASTERISATION (BUILT-IN, NON-PROGRAMMABLE)
Flattens each 3D polygon against the screen, calculating which pixels each triangle covers and dividing each triangle up into pixel-sized pieces called fragments.

5 - FRAGMENT SHADER
Lastly, the fragment shader, colours each fragment generated from the Rasterisation stage.

2.5 TRANSFORMATION PIPELINE

Model data is already on the GPU when we want to draw it. But very often we’ll want to reposition this model data in the created world (e.g. for object movement). This is done by passing matrices across on the draw call and storing them in uniform variables. A uniform is a GLSL variable used to store values we need when we make the call to draw an object. They act like parameters to the shader programme. These matrices are then used to alter the position of each vertex point on the model. There are commonly 3 matrices used for this: the model matrix, view matrix and perspective matrix. Their effect is outlined in Figure 2.

Each of these 4x4 matrices are passed to the vertex shader where they are applied to a vector of length 4 containing the input vertex position. This multiplication gives the vertex a position in the world relative to the camera’s view. These matrices are essentially used to simulate the view of a camera moving through space.

\[ \text{out\_vertex\_position} = \text{P} \times \text{V} \times \text{M} \times \text{vec4 (vertex\_point, 1.0)}; \]

The model matrix M is used to position each point on the model into a position in the world relative to the world’s origin.

View matrix V is then used to position model points relative to the camera’s view. It repositions each vertex point of the model so that the new origin in the scene is the location of the camera.

Lastly the perspective matrix P alters the shape of the camera’s viewing area. It describes the field of view and depth of what the
camera sees. This works by positioning model vertices relative to the shape of the camera’s view area (e.g. triangles at the farthest point of the camera’s view will be squished together so that more of the scene is viewed from a distance, this replicates the effect of far away things appearing smaller).

The GPU will then cut out any fragments of each triangle that is not in the camera’s view. This is called clipping which is a stage that happens automatically doesn’t need to programmed.

2.6 TECHNOLOGIES INVOLVED

2.6.1 OpenGL

This is the graphics library we selected to implement our project with. It was chosen because of our familiarity with working with it previous to beginning this project.

OpenGL was introduced in 1992. Since then it has become the most widely adopted graphics library in the industry. Its 2D and 3D graphics application programming interface (API) has become an industry standard. The library has been used to produce graphical applications across multiple operating systems. [9]

2.6.2 Blender

Blender is a 3D modelling tool. All models other than the monkey model were created in Blender for this project. The monkey model, called Suzanne, is included in Blender to function as a test model [1]. Models created in Blender can be exported and used in OpenGL to create a 3D virtual scene.
3D Transformation Pipeline

- Create mesh
- Has local origin
- e.g. character mesh

LOCAL SPACE

- Model matrix
- Inverse model matrix

VIEWPORT SPACE

- Frustum → cube
- Visible xyz is -1:1

NORMALISED DEVICE SPACE

- View matrix
- Inverse view matrix

WORLD SPACE

- Meshes placed in world
- World origin

- Camera is origin
- Xyz axes line up to cam. orientation

EYE SPACE

- Origin at cam.

HOMOGENEOUS CLIP SPACE

- W = -2
- Xyz scaled into frustum shape
- Gl_Position =

- Projection matrix
- Inverse projection matrix

Figure 2: 3D transformation pipeline, Anton Gerdelan’s “Anton’s OpenGL 4 Tutorials”
Part II

IMPLEMENTATION
SET UP

Most of this project involves programming GPU shaders. To best understand how the shader code is functioning, it’s necessary to see how the shaders were set up and what data needs to be prepared for a draw call.

3.1 INITIALIZING AND DRAWING MODEL DATA

Before an object is rendered it’s necessary that some data be prepared. Outside of our program the model will first need to be created, painted and exported from a 3D modelling tool. Painting here refers to colouring each polygon of the model. This can also be done in blender and essentially results in an image that holds the colour of a flattened version of the object. See Figure 3 for an example.

There are many file formats to store model data and textures in. For this project we stored our model in the .obj file extension and any accompanying textures were stored in .jpg or .png.

Before we enter the drawing loop it is useful for any data that will be used to already be on the GPU. This is because it is very slow to pass data across the CPU-GPU bus. Only variables that are likely to change often should be passed over on the draw call such as the position of the model and the location of each light source.

Figure 3: Texture mapping performed on a cube
Thus the following must be set up:

1. Load model data from .obj and store data on GPU, keeping a reference to its location.

2. Load textures and store texture data on GPU, keeping a reference to its location.

3. Load shader files, compile a program from shader files and pass to GPU, keeping a reference.

4. Obtain references to each uniform variable used in the shader programme.

For each frame, the following sequence of set up is also needed to draw an object:

1. Bind the reference to whichever shader program is to be used. Here we’re basically telling the GPU “Hey remember that data I gave you earlier? Yeah, I’d like to use that now”.

2. Similarly, activate and bind any textures to be used.

3. Bind model data.

4. Pass uniform variables over to GPU.

5. Call glDrawArrays(). This tells the GPU to run with what we’ve given it and display something on the screen. In this report we commonly refer to this as “the draw call”.

3.1 INITIALIZING AND DRAWING MODEL DATA
IMPLEMENTATION

4.1 OVERVIEW

For this project we separately implemented several features of manga in OpenGL. Here we describe how each feature was implemented and then there will be a note on combining these features.

4.2 STYLE

It is necessary that we are able to mimic the manner in which objects in manga are presented.

4.2.1 Cel shading

Cel shading displays lighting as a series of discrete steps (see Figure 4), as opposed to realistic shading (e.g. Phong shading, see Figure 5), where the contrast from light to dark is smooth. Cel shading has been used in popular games, one of such is Legend of Zelda: The Wind Waker where is was used to create a cartoon-like effect.

4.2.2 Texture Sampling

In cel shading we select a colour to display for each step of light intensity. If for each lighting step, we instead sample a texture, we can copy common styles used in manga (e.g. hatching and half-toning).

We do this by selecting a style and having several images of that texture with increasing levels of shade. These images are all tiled, meaning that when they’re lined up, the transition from one to the other is seamless even across darker versions of the same tile. So for each pixel in the output image, we map that to a pixel in our tiled image. We then calculate the strength of light for that pixel and choose a version of the tile that displays a similar amount of light. Finally we take the colour value in the matching tile pixel and output that to our current pixel. This is done for every pixel in the output window.
Figure 4: Cel shading output
Figure 5: Phong shading output
Figure 6: Output of styles recreated with texture sampling
4.3 Drawing Outlines

In manga the shape of an object is simplified to just an outline. The artist draws lines around the silhouette of the object and on any surface where there should be a perceptible curve. We recreated this by using edge detection, a technique commonly used in vision to discover the boundaries of objects from 2d images.

First we render the scene and colour it by distance from the camera. We position our model points as usual in the vertex shader. And then in the fragment shader we colour each fragment by its distance from the camera (i.e. the further away an area is from the camera the darker its shaded so that an object fragment immediately in front of the camera is bright white and a fragment at the furthest possible viewing distance is the darkest black). This is called a depth map, see Figure 7.

Edge detection is applied to this image to achieve an outline of the silhouette as shown in Figure 8. We used the Sobel Filter, this is a common image processing technique that calculates the difference in colour intensity between a pixel and its surrounding neighbours [11]. A pixel with a large difference on two opposite sides is determined to be an edge and is given a high value in the output image, this will appear as white on screen. Otherwise the pixel will have a low value, appearing black. The Sobel technique was chosen for its speed, other more accurate techniques exist but are slower [10] and thus unsuitable for real-time use.

Next a normal map of the scene is rendered. To produce a normal map each fragment is coloured by the direction that its polygon is facing (e.g. right facing triangles are coloured red, top-facing green and front facing blue). See Figure 9.

Again, edge detection is applied to the image and lines are drawn on surfaces with a sharp curvature, see Figure 10.

Finally, the two edge maps are combined to produce a complete outline of the scene. Inverting this gives a black outline of the object (shown in Figure 11) on a white background. This is similar to how outlines are seen in manga.
4.3 Drawing Outlines

Figure 7: Depth map

Figure 8: Depth map edges
Figure 9: Normal map

Figure 10: Normal map edges
Figure 11: Complete output of object outline
4.4 A NOTE ON RENDERING GOOD EDGES

Not all models are suitable for rendering the outlines of. For example if there is no sharp surface curves there will be no lines on the surface. If you want specific edges on the model to have an outline then how the model is created is quite important. The artists need to be sure that any edges that they want to be outlined are well defined and sharp. This can be done in Blender by marking which edges to sharpen and then applying the EdgeSplit modifier as seen in Figure 12.

4.5 SCENE MOTION

In manga there are two common techniques for dealing with general motion in the scene.

4.5.1 Background Motion

Often in manga the motion of one object is exaggerated (e.g. for a character throwing a fist or some explosion in the scene). This exaggeration is done by replacing the background with motion lines. This was recreated in OpenGL by generating a set amount of lines from random points at the edge of the screen towards a set point in space. In OpenGL the view area is within the x range of -1 to 1, the y range is within -1 and 1 and the z range is between 0 and 1. An edge point is found by taking values that correspond to any point on the perimeter of the z-y plane (i.e. either the x value or the y value needs to be at the maximum or minimum range value). These lines are pushed to the back of the rendering area so that they don’t pass through any objects in the scene (i.e. the z value is set to 1). To colour the line, the end point of the line is set to be white and the start point (at the screen edge) is set to be black. Fragments along the line will interpolate their colour from their proximity to one of the endpoints giving the line a smooth gradient from black to white. Each frame, the lines will be repositioned and redrawn. See the output in Figure 13.

4.5.2 Camera Motion

Another common method of displaying motion in manga is to draw lines over the scene to depict the camera’s movement. This was recreated by picking random points on the edge of the screen and extending a line towards where that screen point was in the previous frame. Lines are coloured the same way as in Background Motion. These lines are however drawn over the scene as opposed to behind everything like before. See Figure 14 for output.
Figure 12: Sharpening edges in Blender
4.5 Scene Motion

Figure 13: Background motion output

Figure 14: Camera motion output
4.6 OBJECT MOTION

In manga, object motion is usually emphasized through one of two methods:

4.6.1 Silhouette Fading

One method for displaying motion is to redraw the silhouette from previous frames. On each draw call we do this by fading all the previous outlines and adding the current outline. We then draw all the outlines. This causes old outlines to become more faded each time the object is drawn displaying a gradient of faded outlines. Eventually each outline fades completely and is no longer drawn. You can see how this was rendered in Figure 15.

4.6.2 Motion Lines

Often in manga, lines are drawn on the object to show motion. We accomplished this in OpenGL by using the geometry shader. This
shader was coded to take vertex points as input and output new lines. The parameters to this shader are two uniform model matrices containing the current position and previous position of each vertex. For each vertex on the model we generate lines extending from the vertex’s current position towards where it was in the previous frame. Because the time between frames is so small each line had to be set to a static length. The result from this didn’t describe well the speed of the object or much of its change in motion (see Figure 16).

To improve this we used a uniform array of model matrices instead of just the current model matrix and the previous one. This array held the model matrices that were used to position the model in the last N frames. The length of the array could be changed to control the amount of motion displayed. This array was designed to be circular so that on each update only the oldest model matrix needed to be replaced with the current model matrix. The result gave a much smoother display of motion (shown in Figure 17).

Next we wanted to have the possibility of drawing more lines from the object. For this we used the tesselation shader to subdivide model triangles on the GPU so that we’d have more points to generate lines
from. We started off with already a lot of lines on the wing so when we added even more this resulted in a messy cluster of lines (Figure 18).

We needed a method of assigning which points on the object motion lines would be generated from. It was decided that the most convenient method for the artist would be to allow them to draw these areas onto the model and we could read this data from texture. This is a separate texture to the one used to paint the object. This could be be done in the artist’s 3D modelling tool of choice. We used Blender to map that only the tips of the wing model should spawn motion lines (Figure 19). We also removed the lines from the body but left them on the tail and feet. For each point on the object we sample the texture colour at the vertex coordinate. If the sampled colour is white we generate a motion line for that vertex otherwise no line is drawn on that vertex. This allows the artist to have total control of where lines are spawned on the object.

The final output can be seen in Figure 20.
Figure 18: Object motion lines on bird model with different levels of tesselation, (a) input, (b) low tesselation, (c) higher tesselation
Figure 19: Creation of the wing texture in Blender
4.7 NOTES ON COMBINING FEATURES

Extra work had to go into combining Drawing Outlines with other features.

This is because edge detection is a screen space technique meaning a whole image is drawn at the front of the view port. This will cover anything that is drawn in the rest of the scene. The main problem here is that displaying an image over the scene causes the whole scene to have the same depth, hence we are losing the information that tells the GPU what objects should be drawn in front or behind other objects. To fix this we need to place depth back into the scene. We do this by taking the image we generated earlier that contained the depth of every fragment (i.e. depth map Figure 7). This is passed into the last stage of rendering the object and each fragment is given back its original depth value.

Also to render an object with both an outline and shading, we have to separately render the object with shading. We then combine the outline image with the shaded image. This is done in the fragment shader. Essentially we tell the shader that for each fragment it should just output the corresponding fragment in the shaded image. But if we find that the fragment had a noticeable outline (e.g. an edge value greater than 0.5) we instead draw a black pixel (for a black outline).

Examples of combined output can be seen in Figure 21 and Figure 22.
4.7 Notes on combining features

Figure 21: Combining background motion, tone shading and object outline

Figure 22: Combining background motion, cel shading and object motion
Part III

FURTHER WORK AND CONCLUSION
In this chapter we discuss what would have been done with more time and extra features not in the exact scope of this project.

5.1 Improvements on Implemented Features

Several features are not ideally implemented could be improved with more effort.

5.1.1 Background Motion Lines

Background motion lines could be made more efficient. At the moment all the lines that are drawn are being created on the CPU and passed over the bus. This has never caused the number of frames drawn per second to slow down to below 60fps. However if there were efficiency issues, a better method would be to just pass a variable containing the focal point to the GPU and to have each of these lines generated in a geometry shader.

5.1.2 Drawing Outlines

The output of drawing outline could be made cleaner by applying more image processing techniques like non-maxima suppression. Currently when we detect an edge we allow several points around the edge to claim that they are the edge. Non-maxima suppression is used to remove any points that are orthogonal to any pixel of a higher edge value so that we are left with a very concise line, one pixel thick, that represents the determined edge. This would also allow the line thickness to be customizable by then applying dilation to the determined edge points with a user specified mask. This was not implemented because adding other features took priority over improving this one and time limitation meant we couldn’t do both.
5.2 ADDITIONAL FEATURES

How speech and actions are conveyed is unique in manga. Different speech bubbles are used to display the emotion of a character in conversation and often the written representation of a sound (e.g. crash, boom, etc.) is added to a scene to emphasise an action. Adding these would enhance the ability of this technology to be used for storytelling.

5.3 IMPROVED SCENES

Models take a considerable amount of time to create. The bird model alone took several hours to create. With more time better models could have been made to better demonstrate the output of each feature and to recreate specific scenes in manga.
CONCLUSION

We succeeded in completing each of our outlined tasks.

We are able to render objects in a manga-like style. An outline of an object can be drawn and within that outline we can shade the object with similar methods of shading to those seen in manga. More importantly this style can be easily customized by swapping out the texture used to shade the object.

We also accomplished recreating similar motion guides to those used in manga. Our motion cues have a resembling character to those that appear in manga, being just as clear but with more precision and informative.

This was all done in a method that would be easy for an artist to work with. For example, to draw background motion, the artist needs to only supply the focal point and lines will be generated to give the effect of motion coming from that given point. Also for drawing object motion lines the artist just has to draw where lines should appear on the object and then select a level of tessellation though, depending on the artist’s needs, tessellation might not even be needed at all.

Lastly we kept the rendering techniques simple and efficient, keeping the project suitable for real-time interactive applications.
Part IV

APPENDIX
ELECTRONIC SOURCES AND RESOURCES

The provided DVD contains a zipped file of the following:

1 - BACKGROUNDMOTIONLINES
   This contains the code used to draw background motion lines.

2 - CAMERAMOTIONLINES
   This contains the code used to draw camera motion lines.
   The camera can be controlled with W/A/S/D for moving forward/left/backwards/right. Camera is turned using the mouse.

3 - SILHOUETTEANDCONTOURS
   This contains the code used to draw outlines.
   The camera can be controlled with W/A/S/D to rotate the model. Press the space bar to display shading.

4 - SILHOUETTEMOTIONBLUR
   This contains the code used to draw silhouette faded motion cues.
   The model can be moved with W/A/S/D for moving up/left/down/right.

5 - OBJECTMOTIONLINES
   This contains the code used to draw object motion lines.

DEMO1
   Demo1 combines tone shading and outline and background motion lines.
   The model can be rotated with W/A/S/D. Press space to enable shading.

DEMO2
   Demo2 combines cel shading and object motion lines.
   The model can be moved with W/A/S/D for moving up/left/down/right.

LIBS
   libs contains the necessary libraries to build each project (GLEW, GLFW, GLM).


10. Shrivakshan, G. T., and C. Chandrasekar. “A comparison of various edge detection techniques used in image processing.” IJCSI
REFERENCES


GLOSSARY

FRAGMENT
This is a segment of the output image, similar to a pixel. Each triangle drawn will produce at least one fragment.

MODEL
A 3D representation of an object stored as an array of triangle vertex positions.

MODEL MATRIX
This is a 4x4 matrix used to position model data into a world setting. It is data passed to the vertex shader to position each vertex of a model into its location in the scene.

NON-PHOTOREALISTIC RENDERING
This is an sub-field of computer graphics which unlike most, doesn’t strive for realism and instead implements stylistic techniques or creates technical images.

VIEW MATRIX
This is a 4x4 matrix used to position model data relative to the camera’s view. It is data passed to the vertex shader to reposition each vertex of a model so that the new origin in the scene is the location of the camera.

PERSPECTIVE MATRIX
This is a 4x4 matrix used alter the shape of the camera’s viewing area. It describes the field of view and depth of what the camera sees. This works by positioning model vertices relative to the shape of the camera’s view area (e.g. triangles at the farthest point of the camera’s view will be squished together so that more of the scene is viewed from a distance, this replicates the effect of far away things appearing smaller).

PRIMITIVE
The term primitives refers to a collection of vertices that can be read by the GPU for rendering (e.g. points, lines, triangles).

SHADER
This is GLSL code used to program the stages of the hardware pipeline.
Glossary

Uniform
A shader variable that holds a value passed over to the GPU from the CPU on a draw call.

Vector
A direction described by x, y and z values. In OpenGL we store this in a vec3 or vec4 which is just an array of 3 or 4 values.

Vertex
A 3D point in space described by x, y and z values. In OpenGL we store this in a vec3 or vec4 which is just an array of 3 or 4 values.

Vec3
An array of length three used to hold vector or vertex data.

Vec4
An array of length four. In OpenGL these are used to represent both vectors and vertices. The fourth value (w) is used to differentiate between which type of data the vec4 hold (w is 1.0 for a vertex and 0.0 for a vector). This allows for some neat tricks when doing matrix multiplication.