Vit Huang

Real-time Non-Photorealistic Rendering in the Style of Pixel Art

Computer Science Tripos Part II

Churchill College

May 15, 2014

Proforma

Name:	**Vit Huang**
College:	**Churchill College**
Project Title:	**Real-time Non-Photorealistic Rendering in the Style of Pixel Art**
Examination:	**Computer Science Tripos Part II**
Word Count:	**10690**
Project Originator:	Vit Huang
Supervisor:	Dr A. Benton

Original Aims of the Project

To design and implement a system to render 3D models in a way that emulates the non-photorealistic style of pixel art, including common features unique to the art form. The system should be capable of rendering a complex scene in real time as the scene animates.

Work Completed

A system that renders a textured 3D model file in a style emulating pixel art, in the form of a web page using WebGL for graphics hardware acceleration. The system is capable of rendering a scene, containing tens of thousands of polygons and at a resolution of 1024×1024 pixels, 60 times a second.

Special Difficulties

None.

Declaration

I, Vit Huang of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1

Introduction

1.1 Pixel Art

As computer graphics technology has evolved over the years, so have the art styles used in applications such as user interfaces and computer and video games. Early technology only allowed for the use of simple black-and-white *bitmaps* or *vector graphics*, all performed on the main processor. Bitmaps (also known as *raster images*) consist of a 2D array, where each array element maps to one screen pixel; vector graphics are described by a mathematical function, and rendered either using a specialised device such as an oscilloscope, or by converting the vector image into a bitmap (a process called *rasterisation*).

The limitations of display technology combined with the cost of rendering meant that artistic concepts had to be communicated using very few colours in very few pixels, leading to a distinctive blocky style – the first form of *pixel art*.

In order to combine the advantages of 3D computer graphics with the aesthetic style of 2D pixel art, I have made use of modern graphics technology to produce a GLSL shader program that renders a 3D model using techniques and limitations commonly used in pixel art. In order to reap full benefits from 3D techniques, the program runs in real-time, capable of rendering a complex scene with a movable camera at a smooth frame rate. This is impossible to achieve with hand-drawn pixel art, and allows developers wishing to use the pixel art style more flexibility than was previously possible.

The first usage of pixel art was for black-and-white bitmap icons in early graphical user interfaces, starting with Engelbart's 'Mother of All Demos' in 1968 [5]. This type of pixel art usually depicted mundane items that might be present on a physical desktop, such as stacks of papers. However, this changed with the rise of video games, such as Space Invaders (pictured in Figure 1.1). Very early games did not feature pixel art – games such

Figure 1.1: An example of early pixel art, from *Space Invaders* (1978) by Tomohiro Nishikado

as *Tennis for Two* (1958; William Higinbotham) and *Spacewar* (1961; Steve Russell) were played on oscilloscope screens, with graphics traced out by modifying the electron beam's path. However, as the industry grew, developers started to improve the graphics presented to the user in order to gain an edge over their competitors. Consumer video game hardware hooked into pixel-based televisions rather than oscilloscopes, so pixel art similar to that in Figure 1.1 quickly fell into widespread use. As technology and processing power steadily progressed, artists were free to create more and more detailed art.

The use of pixel art in games peaked in the mid-1990s, with games consoles such as Nintendo's SNES (Super Nintendo Entertainment System) using hardware optimised to display many detailed pixel art *sprites* simultaneously. However, pixel art was still defined by the same limitations that spawned it. Despite the vastly more powerful hardware, there were still limitations on the size of both the sprite itself and the *palette* its colours were drawn from; a sprite is a 2D image that forms part of a scene, such as an object or character, while a palette is a vector of colours that are used in a sprite. Sprites were stored as a 2D array of integers, where each element was an index into the sprite's palette. The size of the array allocated per sprite put a limit on the size, and the size per integer index limited the number of colours that could be in the palette. For example, the SNES had an output resolution of 256x224 pixels, with the most commonly-used operating mode of the graphics chip using 4 bits per pixel and thus supporting 16 colours per sprite (including the transparent background colour).

Due to these limitations, pixel art is identified by several features common across all eras. Each image has a relatively small size (the pixel art figures presented are enlarged using nearest neighbour magnification for clarity) and draws the colour of each pixel exclusively from a fixed palette. This leads to quirks usually not seen in other art styles, such as visible 'banding' as a result of quantising a smooth gradient to a small palette (as seen on Link's hair in the rightmost sprite in Figure 1.2), or the use of a single shade of colour for parts of the subject that might actually be different colours in the original concept (compare the colours of Link's hair, shield, and shoes in the leftmost sprite).

is cheaper to produce than 3D assets for many applications (for example, cases where a character is always seen from the same angle).

1.2 2D versus 3D

Meanwhile, 3D graphics technology has also improved; modern hardware now implements 'programmable shaders', giving the programmer a very high degree of control over the rendering process by letting them modify the behaviour of parts of the rendering process. This is usually done using a C-based shading language such as Microsoft's *High-Level Shader Language* (HLSL) or OpenGL's *OpenGL Shading Language* (GLSL). There are two main points where shader programs are executed – per-vertex, and per-pixel (also called per-fragment). This allows programmers to depart from the 'plastic' look previously associated with computer graphics, and produce a range of effects ranging from photo-realistic surfaces to cartoon-like shading.

Even disregarding aesthetic appeal, 3D art has several advantages over 2D art (including pixel art). The most obvious is that a 3D model can be viewed from any angle; 2D art must be drawn separately for each of those angles. 3D models are easier to animate than 2D art – a 3D model can be attached to a deformable skeleton, which is interpolated between key frames of the animation; 2D art must again be drawn separately for each frame of animation (as a side effect, this results in 2D animation usually using a lower frame rate to lower the artist effort required, while 3D animation can be displayed at any frame rate). Finally, 3D assets scale much better than bitmapped 2D art; a 3D model or 2D vector graphic rendered at a high resolution looks just as crisp (barring low-resolution surface textures) as the same model rendered at a low resolution, while bitmaps become either blurry or blocky depending on the scaling algorithm used.

1.3 Related Work

Pixel art is a niche topic, so there is little existing academic work dealing with the art style; however, there exists related work within the broader field of non-photorealistic rendering.

Several elements of the pixel art style, such as a limited palette and outlining, are also present in cartoon art. A method of rendering 3D models as cartoon art is described by Decaudin [4]; the implementation of outlining shown there formed the basis for my own.

Gerstner [7] gives a technique for converting an arbitrary bitmap into a pixel art image by dividing the image into irregular groups of similarly-coloured pixels; this provides an

alternative method of rendering 3D models as pixel art, namely rendering the 3D model realistically to a bitmap, then using Gerstner's technique to convert it into pixel art. However, this method would not meet the goal of real-time rendering of complex scenes; Gerstner's technique produces results in "generally less than a minute" on a modern processor, and the approach cannot be parallelised to make it run on the graphics processing unit.

Inglis [9] describes a technique for rasterising vector line art at low resolutions, similarly to pixel art. This could be used to improve the rendering of outlines, but once again the algorithm cannot be parallelised and therefore cannot be run in real time.

page that can be displayed in the commonly-used Firefox and Chrome web browsers. These browsers were chosen as they together account for 80% of browser usage [15].

One of WebGL's missing features is the ability to use tessellation and geometry shaders. I originally proposed that I implement curve smoothing using shaders; however, this would require the use of one of the missing shader types, as the shader types WebGL supports do not have the ability to account for other vertices that lie on the curve.

Finally, my proposal's success criteria omitted the factor of the rendering resolution, stating only the number of polygons in the model to be rendered. My implementation spends most of its rendering time in fragment shading full-screen polygons (e.g. edge detection on a depth map), so the largest factor in rendering time is resolution rather than polygon count.

Chapter 3

Implementation

3.1 Overview

The project is divided into two parts – the shader code, written in GLSL, and the support code, written in JavaScript. The GLSL itself is divided into vertex shaders and fragment shaders.

GLSL Shaders

A complete shader program consists of a *vertex shader* and a *fragment shader*. The vertex shader is executed once per vertex, even if that vertex is part of multiple polygons; its purpose is to transform vertex coordinates from model space to screen space.

The fragment shader is executed once per *fragment*. A fragment is a screen pixel contained within a polygon being rendered. A single pixel might contain multiple fragments. For example, if two triangles overlap, the pixels in the overlapping region will contain a fragment for each polygon. The z-buffer technique is used in order to avoid evaluating the fragment shader multiple times – the z-buffer is a memory buffer that stores the depth of each pixel relative to the screen as it is rendered, allowing obscured fragments (those with a depth value greater than that present in the depth buffer) to be discarded immediately.

The inputs to the fragment shader are the interpolated outputs from the vertex shader and the fragment coordinates from the previous stages of the OpenGL ES pipeline (see Figure 3.1). The shader program uses these values to compute the RGB colour to be drawn to the screen, by performing texture lookups and lighting calculations. This colour is the primary output of the fragment shader, and the only output I required.

The advantage of performing these calculations using GLSL shader programs is that GLSL

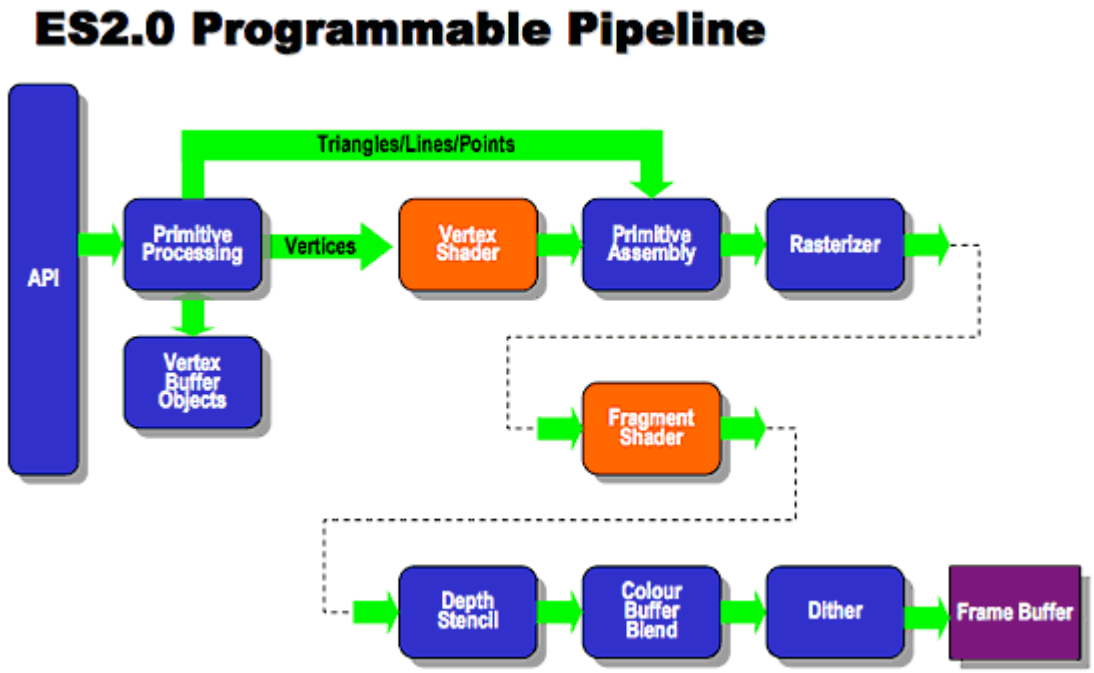

Figure 3.1: The OpenGL ES pipeline; image by Khronos Group (2007).

is executed on the *graphics processing unit* (GPU). Modern GPUs are massively parallel single-instruction multiple-data processors. As one of the goals of my project was to make the program run in real time, making use of this parallelism was very important in meeting that goal.

JavaScript code

GLSL is executed by having the CPU issue rendering commands to the GPU using OpenGL ES. OpenGL ES works as a state machine; the API largely consists of functions that modify the current state. An example of OpenGL ES code is given in Appendix A.

OpenGL ES objects such as textures, vertex buffers, and compiled shaders are stored in GPU memory; the CPU is given an integer ID when an object is allocated. These IDs are used by the JavaScript code when calling OpenGL ES functions rather than using the objects themselves.

Vertex data is supplied to the GPU by encapsulating it in a *vertex buffer object*, a data structure that exists only in GPU memory. The `glBufferData()` function is used to copy data from the CPU to the GPU. Storing data in GPU memory removes the need to resend

every byte of rendering data to the GPU every time it needs to be rendered, as desktop OpenGL's now deprecated 'immediate mode' does.

The GPU vendor includes a GLSL compiler in the hardware driver to abstract architectural differences away from the programmer; the JavaScript passes the GLSL source to the driver for compilation and linking.

After a GLSL program has been compiled and linked, the `glGetUniformLocation` function gets a reference to a *uniform* variable in the shader. Uniform variables can be set by the CPU, and as such are used to pass arguments to a shader, by using the `glUniform` family of functions. Uniforms can be any GLSL type, including matrices and texture samplers; for example, I used a matrix uniform to pass the sRGB $\rightarrow$ CIEXYZ colour space conversion matrix to the colour difference shader.

Vertex buffers are drawn using the `glDrawArrays()` function. OpenGL ES is a state machine, so calling `glUseProgram(program)` at any point before the invocation of `glDrawArrays()` sets the OpenGL ES state to render the vertex buffer using `program`.

As GLSL does not have access to CPU memory, all resource loading must be performed in JavaScript. Fortunately, as JavaScript was designed as a language for use on web pages, functions such as loading images to use as textures were built in to the language, meaning I did not require an additional library to perform these tasks. Chrome blocks the access of local files as a security precaution; to avoid this, I set up a local HTTP server to serve the project files rather than opening the HTML file directly in Chrome.

OpenGL commands are non-blocking; after the JavaScript program issues the commands, it immediately resumes execution. OpenGL ES does not provide a mechanism to read the output of a shader program back into the CPU; hence, any code that is expected to produce output that is consistent across the execution of the entire program must be written in JavaScript.

A WebGL canvas is *anti-aliased* by default. The purpose of anti-aliasing is to reduce visual artefacts caused by rasterisation; as screen pixels are square, a rasterised diagonal line actually appears as a series of steps. WebGL reduces this by blending pixels on the line with the background as in Figure 3.2.

As the project deals with individual pixels, anti-aliasing is actually a hindrance; for example, anti-aliased polygon edges do not have a well-defined single-pixel outline. Although WebGL provides an flag to remove anti-aliasing when setting up a context, I found that using this flag failed to work properly on some browsers. The solution was to render to a texture rather than directly to the JavaScript canvas; rendering to a texture in WebGL does not support anti-aliasing. Rendering to a texture had the additional benefit of allowing the pixel art scene to be scaled before rendering to the screen. By using nearest

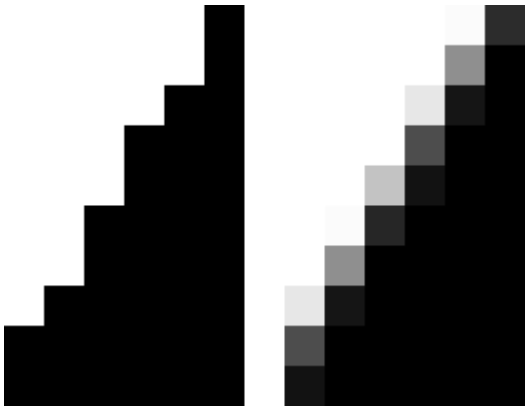

Figure 3.2: An aliased edge is shown on the left, with the same edge but anti-aliased on the right. The anti-aliased image introduces colours that were not directly produced by the shader, which was an issue when attempting to emulate the strict palettes used in pixel art.

neighbour texture magnification, the rendered texture can be displayed at a larger size without blurring the original pixels.

3.2 Colour Quantisation

One of the primary features of pixel art is that the colours used in the image are drawn from a limited palette. I decided to implement this part of the project first, since it is largely independent of the other parts of the project and would give a visible result early on. The only other part of the project that colour quantisation depended on was palette selection, so in order to be able to implement this section first I chose to load the palette from an image file.

An OpenGL texture is created in the GPU's memory to store the palette. My original plan was to use a 1-dimensional texture for this purpose; however, WebGL only supports 2-dimensional textures, so I used a 1-pixel tall 2D texture as a substitute for a 1D texture. This was implemented by using 2D texture lookups of the form `texture2D(texture, vec2(x, 0))` instead of `texture1D(texture, x)` lookup function calls in my shader, providing identical functionality.

I used JavaScript's built-in image loading function in order to guarantee that the image is loaded properly when displayed on a web page. JavaScript provides an `onload` callback on the Image object that is called when the image file is actually in memory; as it may take time to load an image file, I separated the code that copies the texture data to the GPU into a handler function that is linked to the `onload` callback:

Listing 3.1: Texture loaded callback

```
function handleLoadedTexture(model) {
        ...
        model.palette = gl.createTexture();
        gl.bindTexture(gl.TEXTURE_2D, model.palette);
        gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, palette.length / 4, 1, 0,
            gl.RGBA, gl.UNSIGNED_BYTE, new Uint8Array(palette));
        loadedPalette = true;
}
```

JavaScript treats uninitialised object properties as 0. The palette property is only set when the handler is called, meaning that until that happens, the renderer uses the texture with ID 0 for the palette. 0 is OpenGL's null texture ID, producing the same result as rendering the model with texturing disabled. Therefore, the renderer does not produce any unwanted artefacts that might be caused by attempting to use an unloaded texture.

The actual quantisation is performed in a fragment shader. In order to find the most representative palette colour for a given colour, I used the approach of computing the *colour difference* between it and each colour in the palette in GLSL, and outputting the colour from the palette with the smallest difference.

$$\Delta E_{sRGB} = \sqrt{R^2 + G^2 + B^2} \tag{3.1}$$

The simplest colour difference equation is distance in the *standard RGB* (sRGB) colour space, given by Equation 3.1. sRGB is a common colour space used for computer displays; assuming a display is calibrated for sRGB, OpenGL's RGB colour values map directly to sRGB. A downside of sRGB is that it lacks perceptual uniformity – sRGB colour difference does not correspond very well to the difference perceived by the eye, meaning this approach can cause a colour to be quantised to a suboptimal palette colour. This colour difference equation can be implemented using GLSL's `distance` function, as the fragment colour is a simple vector of floats.

After testing that my rendering code functioned with sRGB colour difference, I added code to convert to the CIELAB colour space, which was designed to be more perceptually uniform than sRGB. CIELAB and sRGB are both defined in terms of the CIEXYZ colour space; hence, to transform sRGB to CIELAB, we go via CIEXYZ:

$$C_{\text{linear}} = \begin{cases} \frac{C_{\text{srgb}}}{12.92}, & C_{\text{srgb}} \leq 0.04045 \\ \left(\frac{C_{\text{srgb}}+a}{1+a}\right)^{2.4}, & C_{\text{srgb}} > 0.04045 \end{cases} \tag{3.2}$$

wireframes in desktop OpenGL.

At the time of writing, most WebGL implementations do not yet fully support the `WEBGL_draw_buffers` extension that allows rendering to multiple textures in the same pass; for compatibility reasons, I split the normal, depth, and colour shaders into separate passes. In the future, when the extension is widely supported, this could be optimised by combining the shaders into a single shader and rendering the output of each to a different texture.

The precision of fragments normals is independent on the scene. However, the precision of depth values depends on the positions of the near and far clipping planes in OpenGL ES; as the planes move further apart, the relative depth between two fragments becomes smaller. Depth values are transformed to the range 0.0–1.0 in the vertex shader; therefore, the algorithm needs to be sensitive to smaller depth differences when the clipping planes are further apart.

The normal and depth values are stored in textures. WebGL only supports 8 bits per channel; this is fine for the normal texture (I stored the X, Y, and Z components in the red, green, and blue channels), but inadequate for depth. To get around this, I split the depth value across all four channels, making use of GLSL's vector operators; the resulting colour is effectively a 32-bit fixed-point real.

Listing 3.3: Packing and unpacking floating-point numbers

```
vec4 pack(float x) {
        const vec4 shifts = vec4(256.0 * 256.0 * 256.0, 256.0 * 256.0, 256.0,
            1.0);
        const vec4 mask = vec4(0.0, 1.0 / 256.0, 1.0 / 256.0, 1.0 / 256.0);
        vec4 pack = fract(x * shifts);
        pack -= pack.xxyz * mask;
        return pack;
}

float unpack(vec4 pack) {
        const vec4 shifts = vec4(1.0 / (256.0 * 256.0 * 256.0), 1.0 / (256.0 *
            256.0), 1.0 / 256.0, 1.0);
        return dot(pack, shifts);
}
```

The normal and depth textures are passed into an edge-detection shader. I implemented a Canny edge detector [2] with the Sobel filter, using yet another pass to reduce the outline width to one pixel. However, I found that the width reduction incorrectly determined

the positions of outline pixels where two outlined edges passed close to each other; this was because Canny's algorithm uses only the depth/normal gradient to determine the outline position, placing the outline at the maximum gradient. The true outline lies on the boundaries between pixel. Canny's algorithm does not distinguish between pixels inside and outside the polygon, meaning the outline positioning along the outer edge of the model was inconsistent between animation frames, leading to unpleasant flickering.

Instead, I used a simpler method to determine outline locations. I approximated the gradient by computing the depth difference and normal angle between each pixel and its four immediate neighbours. If either of these were above a threshold value, then I tentatively marked the pixel as being on the outline. If the pixel was tentatively on the outline, I then compared the pixel's depth value to the neighbour in the direction of the greatest gradient. The pixel would be on the outline only if it was closer to the camera than its neighbour. This ensured that outline positions were consistent and unambiguous, eliminating flickering during animation. It also ensures no outline pixels lie outside the model being rendered, which will become important when shading the outlines.

The main advantage of using Canny's algorithm is that it is resistant to noise in the input due to the Gaussian filter step. However, as my input was noise-free computer-generated depth/normal maps, this step made the algorithm less sensitive to fine detail, which the Gaussian filter removes.

Shading Outlines

Colouring outlines using a single colour such as black fulfils the purpose of making a pixel art sprite stand out above the background. Colouring the outlines differently adds definition to parts of the subject where different colours touch; for example, in Figure 3.11, the areas where Whimsicott's green horns overlap its orange eyes are a featureless blob of brown when outlines only use one colour, but show definition when the horns are outlined in green and the eyes in brown and black.

As my outlining algorithm ensured all outline pixels lay within the model, I could sample the Phong shaded render to find the colour beneath the outline. I obtained an initial value for the outline colour by multiplying the RGB components of that colour by a constant smaller than 1, and quantised the initial value to the colours in the palette.

This has the possibility of producing a colour with a luminance equal to or higher than the surrounding non-outlined pixels, which would be unsuitable for outlining. If this is indeed the case, I repeatedly reduced the RGB components of the initial colour by 10% to decrease the luminance, terminating when either the quantised colour had a lower luminance than the surrounding pixels in the Phong-shaded model, or after 10 iterations. This hard limit

the quantisation mapping is expensive; having to recompute this every frame would make it no longer real-time. However, it could be used in an application where real-time rendering is not needed, such as producing first drafts of pixel art for human artists to use as a base.

The problem of detail loss caused by the small size, on the other hand, has a viable solution – enlarge parts of the model that are too small. My original plan was to do this in a tessellation or geometry shader, as those allow me to use information about entire polygons to compute the polygon's size. However, WebGL does not support these types of shader, so I turned to the vertex shader. A vertex shader does not have access to information on the polygons it may be a part of. However, a vertex shader can easily approximate scaling part of a model by translating the vertex along its normal. This effectively increases or decreases the thickness of that part of the model; this is a good target for detail preservation, as part of the model becoming too thin can lead to it not being rendered.

The algorithm requires each vertex be assigned a *model-space size* S_M and a *minimum screen-space size* S_{Smin}. Approximating the part of the model as a sphere with normals pointing outwards from the centre, we find the model-space centre of the sphere C_M by projecting the vertex normal backwards by S_M. The vertex shader then computes the screen-space radius R_S of a circle with radius S_M centred at the vertex's original position. If this is smaller than S_{Smin}, the vertex is translated $(S_{Smin} - R_S)/2$ pixels along the projection of the normal parallel to the screen. This translates the vertex to the edge of the screen-space circle with radius S_{Smin} centred on the point that is the screen-space projection of the model-space approximation sphere.

S_M for a vertex can be computed by approximating the model as a set of spheres, one per vertex of the model, with normals equal to those of the vertices. This exactly satisfies the condition for the earlier approximation to be exact.

I did not consider finding an algorithm to efficiently compute S_M to lie within the scope of the project; as it is a part of the model data, it is computed beforehand and does not affect the execution time of my program. The values of S_M used were approximated by eye.

Chapter 4

Evaluation

Figure 4.1: The ‘Stanford dragon’.

My first success criterion was to render the ‘Stanford dragon’ model shown in Figure 4.1 at a rate of 60 frames per second. I found that a hurdle to this approach was that the model contains over 500,000 vertices; however, WebGL only supports 16-bit vertex indices and therefore only 65,536 vertices per vertex buffer. To avoid this, I chose to use a version of the model with fewer vertices; as pixel art is rarely drawn at very high resolutions, the drop in vertex count is not noticeable in practical situations.

I measured the average frame rate in Firefox and Chrome for four different resolutions, for both a basic Phong shader and my pixel art shader. The results are shown in Figure 4.3.

The maximum frame rate recorded was 60FPS. This is because both browsers I tested with update the WebGL canvas once per screen refresh; as my screen refresh rate is 60Hz, the

Bibliography

[1] H Arthur. clusterfck: Javascript agglomerative hierarchical clustering. `http://web.archive.org/web/20131018090425/http://harthur.github.io/clusterfck/`, 2013.

[2] J Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, June 1986.

[3] CIE. Improvement to industrial colour-difference evaluation. Technical Report 142-2001, CIE, 2001.

[4] Philippe Decaudin. Cartoon looking rendering of 3D scenes. Research Report 2919, INRIA, June 1996.

[5] Douglas C. Engelbart and William K. English. A research center for augmenting human intellect. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 395–410, New York, NY, USA, 1968. ACM.

[6] Robert W. Floyd and Louis Steinberg. An Adaptive Algorithm for Spatial Greyscale. *Proceedings of the Society for Information Display*, 17(2):75–77, 1976.

[7] Timothy Gerstner, Doug DeCarlo, Marc Alexa, Adam Finkelstein, Yotam Gingold, and Andrew Nealen. Pixelated image abstraction. In *NPAR 2012, Proceedings of the 10th International Symposium on Non-photorealistic Animation and Rendering*, June 2012.

[8] Paul Heckbert. Color image quantization for frame buffer display. *SIGGRAPH Comput. Graph.*, 16(3):297–307, July 1982.

[9] Tiffany C. Inglis and Craig S. Kaplan. Pixelating vector line art. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*, NPAR '12, pages 21–28, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association.

[10] T Lottes. Fxaa. White paper, NVIDIA, 2009.

[11] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif., 1967. University of California Press.

[12] R. McDonald and K J Smith. Cie94-a new colour-difference formula*. *Journal of the Society of Dyers and Colourists*, 111(12):376–379, 1995.

[13] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[14] I. Sobel and G. Feldman. A 3x3 Isotropic Gradient Operator for Image Processing. Never published but presented at a talk at the Stanford Artificial Project, 1968.

[15] w3schools.com. Browser statistics. `http://web.archive.org/web/20140507123639/http://www.w3schools.com/browsers/browsers_stats.asp`, 2014.

[16] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, August 1978.

Appendix A

OpenGL ES Example

Listing A.1: Building a texture containing the model outline

```
function renderOutline(model) {
        // render to a framebuffer rather than to the screen
        gl.bindFramebuffer(gl.FRAMEBUFFER, screenFramebuffer);

        // render normals to a texture
        gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
            gl.TEXTURE_2D, normalTexture, 0);
        gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
        drawModel(normalShader, model);

        // render depth to a texture
        gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
            gl.TEXTURE_2D, depthTexture, 0);
        gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
        gl.useProgram(depthShader);
        drawModel(depthShader, model);

        // set up outline shader uniforms
        gl.useProgram(outlineShader);
        gl.activeTexture(gl.TEXTURE0);
        gl.bindTexture(gl.TEXTURE_2D, depthTexture);
        gl.uniform1i(outlineShader.depthTexture, 0);
        gl.activeTexture(gl.TEXTURE1);
        gl.bindTexture(gl.TEXTURE_2D, normalTexture);
        gl.uniform1i(outlineShader.normalTexture, 1);
```

```
        gl.uniform2i(outlineShader.textureSize, screenBufferWidth(),
            screenBufferHeight());

        //render outlines to a texture
        gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
            gl.TEXTURE_2D, edgeTexture, 0);
        gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
        drawRectangle(outlineShader);

        // unbind framebuffer and render to the screen again
        gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
            gl.TEXTURE_2D, screenTexture, 0);
        gl.bindFramebuffer(gl.FRAMEBUFFER, null);
}
```

Appendix B

CIEDE2000 Colour Difference

The CIEDE2000 difference ΔE^*_{00} between two colours in the CIELAB colour space (L^*_1, a^*_1, b^*_1) and (L^*_2, a^*_2, b^*_2) is given by the following equation:

$$\Delta E_{00} = \sqrt{\left(\frac{\Delta L'}{k_L S_L}\right)^2 + \left(\frac{\Delta C'}{k_C S_C}\right)^2 + \left(\frac{\Delta H'}{k_H S_H}\right)^2 + R_T \frac{\Delta C'}{k_C S_C}\frac{\Delta H'}{k_H S_H}} \tag{B.1}$$

The terms used in the equation are given by the following equations (all angles are in degrees):

$$\Delta L' = L^*_2 - L^*_1 \tag{B.2}$$

$$\bar{L} = \frac{L^*_1 + L^*_2}{2} \quad \bar{C} = \frac{C^*_1 + C^*_2}{2} \tag{B.3}$$

$$a'_1 = a^*_1 + \frac{a^*_1}{2}\left(1 - \sqrt{\frac{\bar{C}^7}{\bar{C}^7 + 25^7}}\right) \tag{B.4}$$

$$a'_2 = a^*_2 + \frac{a^*_2}{2}\left(1 - \sqrt{\frac{\bar{C}^7}{\bar{C}^7 + 25^7}}\right) \tag{B.5}$$

$$C'_1 = \sqrt{a'^2_1 + b^{*2}_1} \tag{B.6}$$

$$C'_2 = \sqrt{a'^2_2 + b^{*2}_2} \tag{B.7}$$

$$\bar{C}' = \frac{C'_1 + C'_2}{2} \tag{B.8}$$

$$\Delta C' = C'_2 - C'_1 \tag{B.9}$$

$$h'_1 = \text{atan2}(b^*_1, a'_1) \mod 360^\circ \tag{B.10}$$

$$h'_2 = \text{atan2}(b^*_2, a'_2) \mod 360^\circ \tag{B.11}$$

$$\Delta h' = \begin{cases} h'_2 - h'_1 & |h'_1 - h'_2| \leq 180^\circ \\ h'_2 - h'_1 + 360^\circ & |h'_1 - h'_2| > 180^\circ, h'_2 \leq h'_1 \\ h'_2 - h'_1 - 360^\circ & |h'_1 - h'_2| > 180^\circ, h'_2 > h'_1 \end{cases} \tag{B.12}$$

$$\Delta H' = 2\sqrt{C'_1 C'_2}\sin(\Delta h'/2) \tag{B.13}$$

$$\bar{H}' = \begin{cases} (h'_1 + h'_2 + 360^\circ)/2 & |h'_1 - h'_2| > 180^\circ \\ (h'_1 + h'_2)/2 & |h'_1 - h'_2| \leq 180^\circ \end{cases} \tag{B.14}$$

$$\begin{aligned} T = 1 &- 0.17\cos(\bar{H}' - 30^\circ) \\ &+ 0.24\cos(2\bar{H}') \\ &+ 0.32\cos(3\bar{H}' + 6^\circ) \\ &- 0.20\cos(4\bar{H}' - 63^\circ) \end{aligned} \tag{B.15}$$

$$S_L = 1 + \frac{0.015\left(\bar{L} - 50\right)^2}{\sqrt{20 + \left(\bar{L} - 50\right)^2}} \tag{B.16}$$

$$S_C = 1 + 0.045\bar{C}' \tag{B.17}$$

$$S_H = 1 + 0.015\bar{C}'T \tag{B.18}$$

$$R_T = -2\sqrt{\frac{\bar{C}'^7}{\bar{C}'^7 + 25^7}}\sin\left[60^\circ \cdot \exp\left(-\left[\frac{\bar{H}' - 275^\circ}{25^\circ}\right]^2\right)\right] \tag{B.19}$$

Appendix C

Constructing the CIEDE2000 Cache

```
#define PI 3.14159265358979323846264338327 9
precision highp float;
uniform sampler2D palette;
uniform mat3 rgbConversion;
uniform int paletteSize;
// the precision of the cache
const int bits = 8;

float linearise(float c) {
	if (c <= 0.04045) {
		return (c / 12.92);
	} else {
		return pow((c + 0.055) / 1.055, 2.4);
	}
}

float labf(float c) {
	if (c > 0.008856) {
		return pow(c, 1.0/3.0);
	} else {
		return 7.787 * c + 0.1379;
	}
}

float dsin(float degrees) {
	return sin(degrees / 180.0 * PI);
}
```

```
float dcos(float degrees) {
	return cos(degrees / 180.0 * PI);
}

float datan(float y, float x) {
	return atan(y, x) / PI * 180.0;
}

float ciede2000(vec3 lab1, vec3 lab2) {
	float cavg = (length(lab1.yz) + length(lab2.yz)) / 2.0;
	float cavg7 = cavg * cavg * cavg * cavg * cavg * cavg * cavg;
	float g = 0.5 * (1.0 - sqrt(cavg7 / (cavg7 + 6103515625.0)));
	float ap1 = (1.0 + g) * lab1.y;
	float ap2 = (1.0 + g) * lab2.y;
	float cp1 = sqrt(ap1 * ap1 + lab1.z * lab1.z);
	float cp2 = sqrt(ap2 * ap2 + lab2.z * lab2.z);
	float h1 = 0.0;
	if (lab1.z != 0.0 || ap1 != 0.0) {
		h1 = datan(lab1.z, ap1);
		if (h1 < 0.0) {
			h1 += 360.0;
		}
	}
	float h2 = 0.0;
	if (lab2.z != 0.0 || ap2 != 0.0) {
		h2 = datan(lab2.z, ap2);
		if (h2 < 0.0) {
			h2 += 360.0;
		}
	}
	float dl = lab2.x - lab1.x;
	float dc = cp2 - cp1;
	float dh = 0.0;
	if (cp1 != 0.0 && cp2 != 0.0) {
		dh = h2 - h1;
		if (dh < -180.0) {
			dh += 360.0;
		} else if (dh > 180.0) {
			dh -= 360.0;
		}
```

```
        }
        float dch = 2.0 * sqrt(cp1 * cp2) * dsin(dh / 2.0);
        float lavg = (lab1.x + lab2.x) / 2.0;
        float cpavg = (cp1 + cp2) / 2.0;
        float havg = h1 + h2;
        if (cp1 != 0.0 && cp2 != 0.0) {
                if (abs(h1 - h2) < 180.0) {
                        havg = (h1 + h2) / 2.0;
                } else {
                        if (h1 + h2 < 360.0) {
                                havg = (h1 + h2 + 360.0) / 2.0;
                        } else {
                                havg = (h1 + h2 - 360.0) / 2.0;
                        }
                }
        }
        float t = 1.0 - 0.17 * dcos(havg - 30.0) + 0.24 * dcos(2.0 * havg) +
           0.32 * dcos(3.0 * havg + 6.0) - 0.2 * dcos(4.0 * havg - 63.0);
        float sqterm = ((havg - 275.0) / 25.0);
        float dtheta = 30.0 * exp(-(sqterm * sqterm));
        float cpavg7 = cpavg * cpavg * cpavg * cpavg * cpavg * cpavg * cpavg;
        float rc = 2.0 * sqrt(cpavg7 / (cpavg7 + 6103515625.0));
        float lavgp = (lavg - 50.0) * (lavg - 50.0);
        float sl = 1.0 + (0.015 * lavgp) / sqrt(20.0 + lavgp);
        float sc = 1.0 + 0.045 * cpavg;
        float sh = 1.0 + 0.015 * cpavg * t;
        float rt = -dsin(2.0 * dtheta) * rc;
        float de = sqrt((dl / sl) * (dl / sl) + (dc / sc) * (dc / sc) + (dch /
           sh) * (dch / sh) + rt * (dc / sc) * (dch / sh));
        return de;
}

vec3 srgbToCiexyz(vec3 srgb) {
        vec3 linearRgb = vec3(linearise(srgb.r), linearise(srgb.g),
           linearise(srgb.b));
        return linearRgb * rgbConversion;
}

vec3 srgbToCielab(vec3 srgb) {
        vec3 ciexyz = srgbToCiexyz(srgb);
        // adjust for sRGB white point before calling labf
```

```
        float fX = labf(ciexyz.x / 0.95047);
        float fY = labf(ciexyz.y);
        float fZ = labf(ciexyz.z / 1.08883);
        vec3 cielab = vec3(max(116.0 * fY - 16.0, 0.0), 500.0 * (fX - fY),
            200.0 * (fY - fZ));
        return cielab;
}

vec4 getClosestColour(vec4 colour) {
        float bestDist = -1.0;
        vec4 bestColour;
        vec4 paletteColour;
        // GLSL does not support comparing loop count to a variable, so just
            loop to a safe number and break when necessary
        for (int i = 0; i < 64; i++) {
                paletteColour = texture2D(palette, vec2(float(i) /
                    float(paletteSize), 0));
                float dist = ciede2000(srgbToCielab(colour.rgb),
                    srgbToCielab(paletteColour.rgb));
                if (dist < bestDist || bestDist < 0.0) {
                        bestColour = paletteColour;
                        bestDist = dist;
                }
                if (i >= paletteSize) {
                        break;
                }
        }
        return bestColour;
}

void main(void) {
        float size = pow(2.0, float(bits));
        float sqrtSize = sqrt(size);
        // since we're building a cache texture, the colour to quantise is a
            function of the fragment coordinates
        float r = mod(gl_FragCoord.x, size) / (size - 1.0);
        float g = mod(gl_FragCoord.y, size) / (size - 1.0);
        float b = (floor(gl_FragCoord.x / size) / (sqrtSize - 1.0) +
            floor(gl_FragCoord.y / size)) / (sqrtSize - 1.0);
        gl_FragColor = getClosestColour(vec4(r, g, b, 1.0));
}
```

Project Proposal

to how it affects the surrounding area. At first, I will only consider geometry when looking for details; as an extension, I will try to take texture data into consideration too.

To evaluate the project, I will use two methods. Firstly, I will use an objective method to measure the real-time aspect. I will run the renderer on a number of devices with modern graphics hardware, and collect data on frame render time for scenes of varying complexity, both using standard Phong shading and my pixel art shader.

As the majority of the project concerns aesthetics, I will also use human experiments. I will ask subjects to compare images rendered with the project with images rendered in other ways. Some methods of evaluation I could use are:

- In order to test the visual appearance of my line smoothing algorithm, I will render the same model with and without line smoothing turned on. I will then ask test subjects to say which they prefer. I will make sure the images are shown in a random order.
- Similarly, to test the appearance of dithering, I will render the same model with and without dithering enabled and ask test subjects to say which they prefer. Again, I will show the images in a random order.
- To test detail preservation, I will create several models with easily recognisable detail, such as a texture that has text on it. Then, for each subject, I will render each model once at one of a number of different sizes and with detail preservation enabled or disabled; the number of models used must be small enough such that across all test subjects, each model is represented several times at each size and with detail preservation both enabled and disabled. I will ask the test subjects to identify (e.g. read) the details if possible. Collecting the results, I can measure how effective the algorithm is.
- To test the overall capability of the project, I will display a character model rendered by my project next to pixel art of the same character hand-drawn by several different artists, and ask the test subjects to rank them in order of aesthetics.

Success Criterion

The final deliverables of the project will be:

- A renderer application that allows the user to load a model file and change render settings

- An algorithm that selects a colour palette of a given size to use based on the materials of a model
- Highlights and shadowing in the pixel art style, including the use of dithering
- Pseudo-antialiased and smoothed outlining, as demonstrated in figure 6
- Detail preservation when a model is rendered at small sizes

The criteria for the success of the project will be:

- The renderer must be able to render the Stanford dragon standard test model at a frame rate of at least 60 frames per second on both my desktop and laptop graphics processors (Nvidia GeForce GTX 560Ti and 765M respectively)

Figure 7: A render of the Stanford dragon test model, consisting of several hundred thousand polygons.

- The human experiment comparing rendered images with and without dithered shading should result in the dithered images being preferred in at least 70% of the trials
- The human experiment comparing rendered images with and without line smoothing should result in the line smoothed images being preferred in at least 70% of the trials
- The human experiment measuring the effectiveness of detail preservation should result in the detail preserved images being recognisable at smaller sizes than the non-detail preserved images in at least 70% of the trials
- The human experiment comparing my project's output with hand-drawn art should result in the rendered images being preferred over at least one hand-drawn image in at least 50% of the trials

Timetable and Milestones

Weeks 1 and 2 (26th October to 6th November)

Refresh my knowledge of OpenGL and GLSL by skimming the OpenGL and GLSL reference manuals. Research similar shading algorithms such as cel shading. Decide on a language to use and a file type to load 3D models from.

Weeks 3 and 4 (7th November to 20th November)

Plan and implement a simple renderer, using a Phong shader that will be replaced over the course of the project. Implement model loading.

Weeks 5 and 6 (21st November to 4th December)

Implement loading of palettes from a text file. Add basic palette-limited shading, and start working on implementing dithering.

Weeks 7 and 8 (5th December to 18th December)

Finish implementing dithering, and implement basic pixel outline rendering.

Weeks 9 and 10 (19th December to 1st January)

Finish any unfinished sections from Michaelmas Term. Formally plan out human experiments, and find other pixel artists to make the hand-drawn art required for comparison.

Milestone: Basic pixel art rendering with dithering.

Weeks 11 and 12 (2nd January to 15th January)

Start implementing outline smoothing geometry shader.

Weeks 13 and 14 (16th January to 29th January)

Finish outline smoothing geometry shader, start implementing fragment shader.

Weeks 15 and 16 (30th January to 12th February)

Finish outline smoothing fragment shader, implement automated palette selection.

Milestone: Pixel art rendering with smoothed outlines and automatically selected palettes.

Weeks 17 and 18 (13th February to 26th February)

Start work on algorithm to preserve details across scaling and animation.

Weeks 19 and 20 (27th February to 12th March)

Complete detail preservation algorithm. Find volunteers for human experiments.

Weeks 21 and 22 (13th March to 26th March)

Finish any unfinished sections from Lent Term. Carry out human testing.

Milestone: Complete pixel art rendering and perform human testing.

Weeks 23 and 24 (27th March to 9th April)

Plan out dissertation, start writing.

Weeks 25 and 26 (10th April to 23rd April)

Continue writing dissertation, finish a first draft. Send draft to my supervisor for review and proofreading.

Milestone: First draft of dissertation sent to supervisor.

Weeks 27 and 28 (24th April to 7th May)

Get feedback from supervisor by the middle of the fortnight, and incorporate any necessary changes. As the deadline is the 16th, there is over a week of leeway in case something goes wrong.

Milestone: Submit dissertation.

Resources Required

The project will be implemented using the graphics library OpenGL and GLSL (the OpenGL Shading Language). As such, the project requires the availability of graphics hardware that supports these.

I am also intending to use my own laptop for the project (2.4GHz Core i7, GeForce GTX765M, 8GB RAM, 1TB solid state hybrid drive). As backup, I will use my desktop PC (3.4GHz Core i5, GeForce 560Ti, 8GB RAM, 1TB hard drive). Both computers have the graphics hardware required to implement the project.

As my main backup strategy, I will use the Git version control system, using the online service GitHub to store my files. In addition, I will back up any modified files daily on a USB drive and on my DropBox account.