Figure 1.2: The evolution of pixel art: several sprites depicting the character 'Link' from *The Legend of Zelda* series (Nintendo, 1986-2005)

In addition, parts of the subject are often outlined with a one-pixel-wide border in order to make them stand out from the background and from each other (notice how much easier it is to resolve the features of the second and third examples of Link, which use this technique, than the first, which does not). Originally this was done with a single colour (usually black, as seen in the middle sprite in Figure 1.2), but as the style evolved, artists started to replace this with darker shades of the base colour of the object, resulting in a softer look that preserved the purpose of the technique (compare the edge of Link's hair in the middle and rightmost sprites).

Finally, smaller features on the subject are often enlarged in order to make them identifiable in the small number of pixels available; in the rightmost example, Link's shoes have been enlarged relative to their physical proportions in order to make his walking animation clearer, and the hair at the sides of his head has been widened to allow for it to be outlined to separate it from his face and the background.

Figure 1.3: Examples of types of dithering (from Wikimedia Commons)

In order to produce the illusion of colours in addition to those already in the palette, a technique named *dithering* was used. The image would contain a pattern of pixels of different colours. The cathode ray tube displays at the time used a shadow mask in order to ensure the three electron beams could only strike and illuminate one colour of subpixel each. However, the beams themselves diverged slightly, so each would slightly illuminate surrounding subpixels of the same colour. This resulted in neighbouring pixels blurring together, causing dithered areas to appear closer to the uniform colour formed by blending

the dithered colours. Although modern displays no longer produce this effect, the high resolutions in use now pack the pixels closely enough that it is difficult for the human eye to distinguish separate pixels – this gives a similar effect, as we perceive a dithered pattern as a blend of the colours used.

Eventually, 3D graphics support in desktop PCs and consoles such as Sony's PlayStation overshadowed the comparatively primitive 2D hardware that pixel art thrived on, and the art form started to fall out of use. However, in the 2000s it found a new niche in applications such as mobile phone apps and handheld game devices, where 3D graphics hardware was impractical due to size and power consumption.

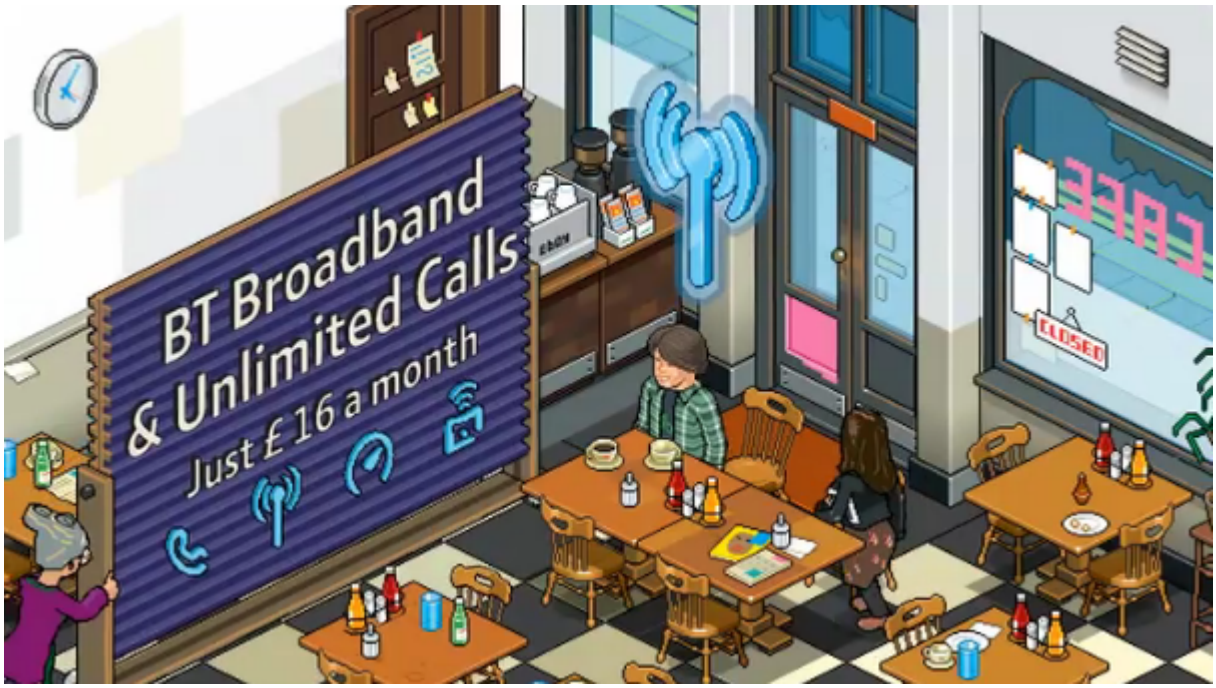

Figure 1.4: An advertisement using pixel art, by Pedro Cyrne (2012)

Pixel art has also continued to be an inspiration in design of user interfaces, with many of its techniques remaining useful for designing icons that must be identifiable at a small size and for embedded devices with low graphical capabilities such as calculators. In addition, several companies such as BT have adopted the use of pixel art in advertising (an example is given in Figure 1.4) in order to appeal to consumers with fond memories of pixel art-based games, rendering everyday scenes in the eye-catching style.

Recently, many independent games developers, and even some larger companies, have returned to pixel art-inspired styles – this is due both to aiming to reproduce the look of what many people consider to be the 'golden age' of video games, and that 2D art

Chapter 2

Preparation

2.1 Art Style

'Pixel art' is an umbrella term for a family of art styles sharing common elements, with examples shown in Figure 2.1. These vary by, for example, their use of outlining – many variations, such as that used in the lower-left image in Figure 2.1, choose to forgo outlining, and must resort to the use of lighting and colour to distinguish parts of the subject.

I chose to focus on a single style, with some freedom in terms of parameters that can be modified. This ensured I would have a single goal to aim for, which benefited me in that I would not need to spend time tweaking the output in order to match a shifting target, and helped keep the scope of the project constant.

In order to achieve this, I required a style with a large number of existing samples, so I would have enough references to work from. It would have to incorporate all of the core features shared among most pixel art styles. Ideally, there should exist 3D models depicting the same subjects shown in the pixel art – this would allow me to make a direct comparison between the original pixel art and my project's rendering of the 3D model in the same style.

These requirements led me to choose to emulate the style used in Game Freak's *Pokémon* series of video games. The series is known for having hundreds of unique designs for its 'Pokémon' characters. It consists of both 2D titles for handheld game systems and 3D titles for home consoles. As such, each character has both pixel art in several styles and high quality 3D models available. In addition, each character has pixel art from two different angles (front and back), usually in the same pose; again, this gives me an additional comparison. Pixel art was used in the games as late as 2012's *Pokémon Black/White Version 2*, so I believe the style used in those games acts as a good representation of the

current state of pixel art.

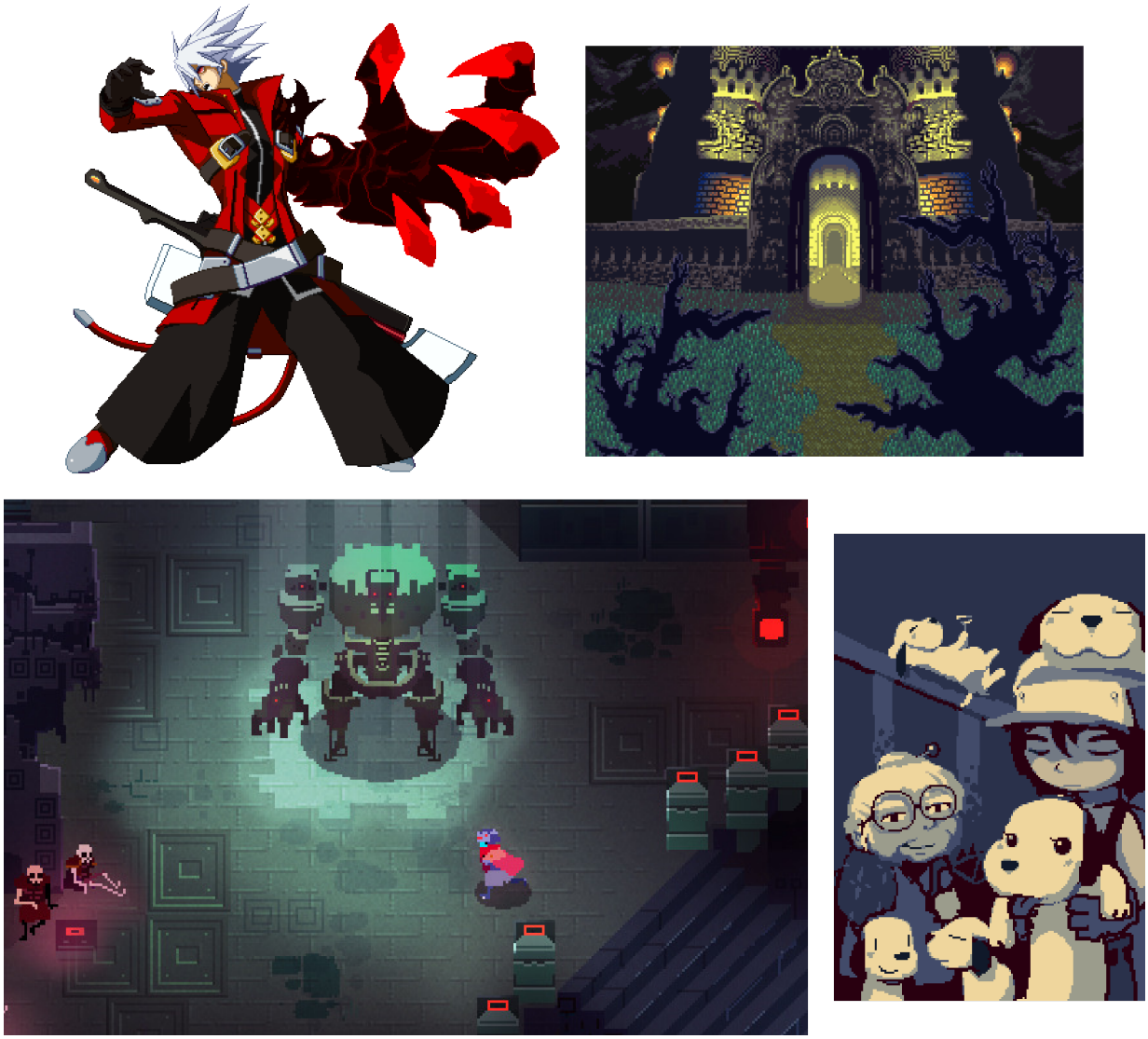

Figure 2.1: Examples of different styles of pixel art in video games. Clockwise from top left: a character animation frame (*BlazBlue: Calamity Trigger*, Arc System Works, 2008), background scenery (*Chrono Trigger*, Square, 1995), a scene illustration (*Cave Story*, Studio Pixel, 2004), and pixel art mixed with newer technology such as alpha blending (*Hyper Light Drifter*, Heart Machine, 2014)

Figure 2.2: Pixel art from *Pokémon Black/White Version 2* (2012) by Game Freak; pixel art team led by Hironobu Yoshida

2.2 Implementation Choices

Before starting to write code, I had to make several decisions about what I would use to implement it, such as the programming languages used.

Firstly, I had to decide on a language to implement the shader component of the project in. The two commonly used shading languages are Microsoft's *High-Level Shader Language* (HLSL) and OpenGL's *OpenGL Shading Language* (GLSL). Both languages have a very similar feature set, with the primary differences between the two detailed below:

- HLSL is a proprietary language developed by Microsoft.
- HLSL is supported only on Microsoft operating systems (Windows and Xbox)
- GLSL is an open standard.
- GLSL, or GLSL-based languages, are supported on all commonly-used operating systems, including smartphones and games consoles (except Xbox).

I chose to use GLSL, as it would enable the shader portion of the project to be ported to a larger number of devices.

Secondly, there was the choice of which language to use to implement the framework of the program, including the code to load in the model and pass the shader program to the GPU. My choices here were Java, C++, and JavaScript (using the WebGL library).

- I already knew Java and C++ from Part IA and IB of my course.
- C++ is the fastest language; however, I expect the performance bottleneck to be in the shader portion, so speed is not a large factor.
- C++ also requires recompilation for each platform, and lacks built-in graphics libraries.
- JavaScript/WebGL is the most portable language, supported by most web browsers without an additional runtime environment download.
- WebGL is based on *OpenGL for Embedded Systems* (OpenGL ES), which has a smaller feature set than desktop OpenGL; however, desktop OpenGL features are unavailable on many devices such as smartphones.

I chose to use WebGL for my implementation due to its portability. Although I would have to learn a new language and lose out on some OpenGL features, JavaScript's similarity to Java would make learning the language fast, and I ended up finding alternatives to the missing features that I required.

2.3 Unit Testing

After I had decided to use WebGL, I looked into unit testing frameworks. I decided to use Google's glsl-unit framework for testing my GLSL code; this was especially suitable for the project as it was designed specifically for WebGL. The framework is an extension of JsTD, an established JavaScript unit testing framework; therefore, it made sense to use JsTD to test my JavaScript code.

I had issues setting up glsl-unit due to a missing configuration file, but as it was based on the well-documented JsTD, I managed to resolve the problem quickly.

2.4 Changes to the Proposal

My decision to use WebGL means that the goal in my proposal to produce an application to display the output became unnecessary. I amended this goal to be to produce a web

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.4124 & 0.3576 & 0.1805 \\ 0.2126 & 0.7152 & 0.0722 \\ 0.0193 & 0.1192 & 0.9502 \end{bmatrix} \begin{bmatrix} R_{\text{linear}} \\ G_{\text{linear}} \\ B_{\text{linear}} \end{bmatrix} \tag{3.3}$$

$$L^\star = 116 f(Y/Y_n) - 16 \tag{3.4}$$

$$a^\star = 500[f(X/X_n) - Y/Y_n] \tag{3.5}$$

$$b^\star = 200[f(Y/Y_n) - Z/Z_n] \tag{3.6}$$

where X_n, Y_n, and Z_n are the CIEXYZ values of the reference *white point* (the point defined as 'white' in the colour space in use). The sRGB colour space uses the D65 white point meant to simulate noon daylight; for this white point we have $X_n = 95.047$, $Y_n = 100.0$, and $X_n = 108.883$.

Both the palette colour and fragment colour vectors were transformed in this way before calculating the colour distance using Equation 3.7. In theory, this should lead to an image that is perceived to be closer to the original image than using RGB difference.

$$\Delta E_{CIELAB} = \sqrt{(L^\star)^2 + (a^\star)^2 + (b^\star)^2} \tag{3.7}$$

However, I found that using distance in CIELAB space led to artefacts where colours were being quantised oddly, as shown in Figure 3.4. After writing many unit tests and cross-checking my colour space transformation code's output against existing implementations, I deduced that this error was due to CIELAB not being perceptually uniform [12]. After some additional research, I decided to use the CIEDE2000 colour difference equation [3]; the full equation is given in Appendix B.

CIEDE2000 was more complex to evaluate than the colour difference formulae I considered previously, causing the rendering time for high-resolution scenes to be too long to be considered real-time. To resolve this problem, I opted to cache the mapping from RGB colour space to the closest colour in the palette using the CIEDE2000 formula. One method of doing so would be to cache the closest colours on the fly by writing them to a texture in the GLSL shader. However, the version of OpenGL ES used in WebGL does not support rendering to multiple render targets on the same pass. To implement this, I would have to use a pass to update the cache, then another pass to actually draw the model (even if the cache was not updated).

As caching closest colours on the fly required an extra pass, I chose to quantise incoming RGB values to 8 bits per RGB component, then precompute the closest palette colour for all 2^{24} RGB triples. The output RGB colour is stored in a texture using the input triple as the texel coordinate. My implementation is given in Appendix C

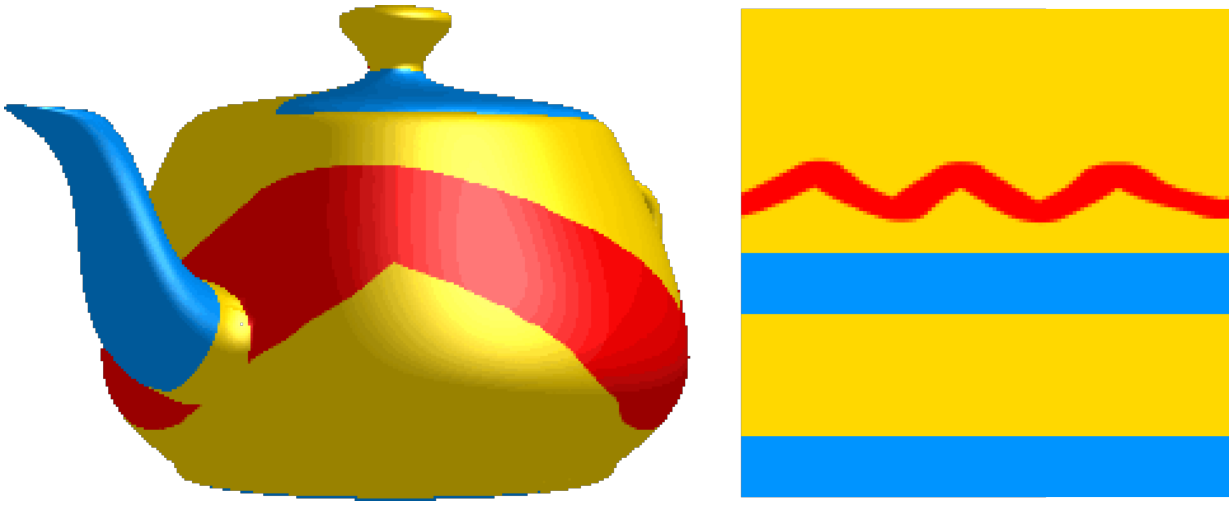

Figure 3.3: A textured 'Utah teapot' model, lit using the Phong algorithm. The texture used is shown to the right.

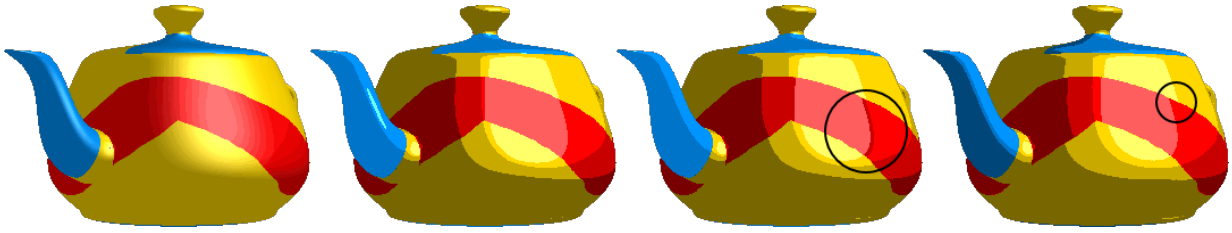

Figure 3.4: From left to right: no quantisation, sRGB distance, CIELAB distance, CIEDE2000. Note the circled artefact on the red stripe on the CIELAB-quantised teapot caused by the CIELAB difference between shades of red not corresponding to perceived difference, and the shading edges not lining up on the CIELAB and CIEDE2000 (circled) teapots due to the palette being designed for sRGB.

The memory size of this cache is 4×2^{24} bytes, or 64MB. Although this is a large memory footprint for a single texture, the GPUs I used to test have 1GB total memory each, and current and future GPUs will have similar, and even larger, amounts. If memory were an issue, then the incoming RGB values could be quantised further; each extra bit of quantisation halves the memory usage of the cache (for example, using 7-bit red and blue components with an 8-bit green component results in a 16MB cache).

While implementing this, I found that WebGL did not support the use of 3D textures. I worked around this by devising a mapping from a triple of 8-bit integers to a pair of 12-bit integers, allowing the 3D data to be stored in and retrieved from a 2D texture.

In order to perform this large number of computations efficiently, I leveraged the GPU by

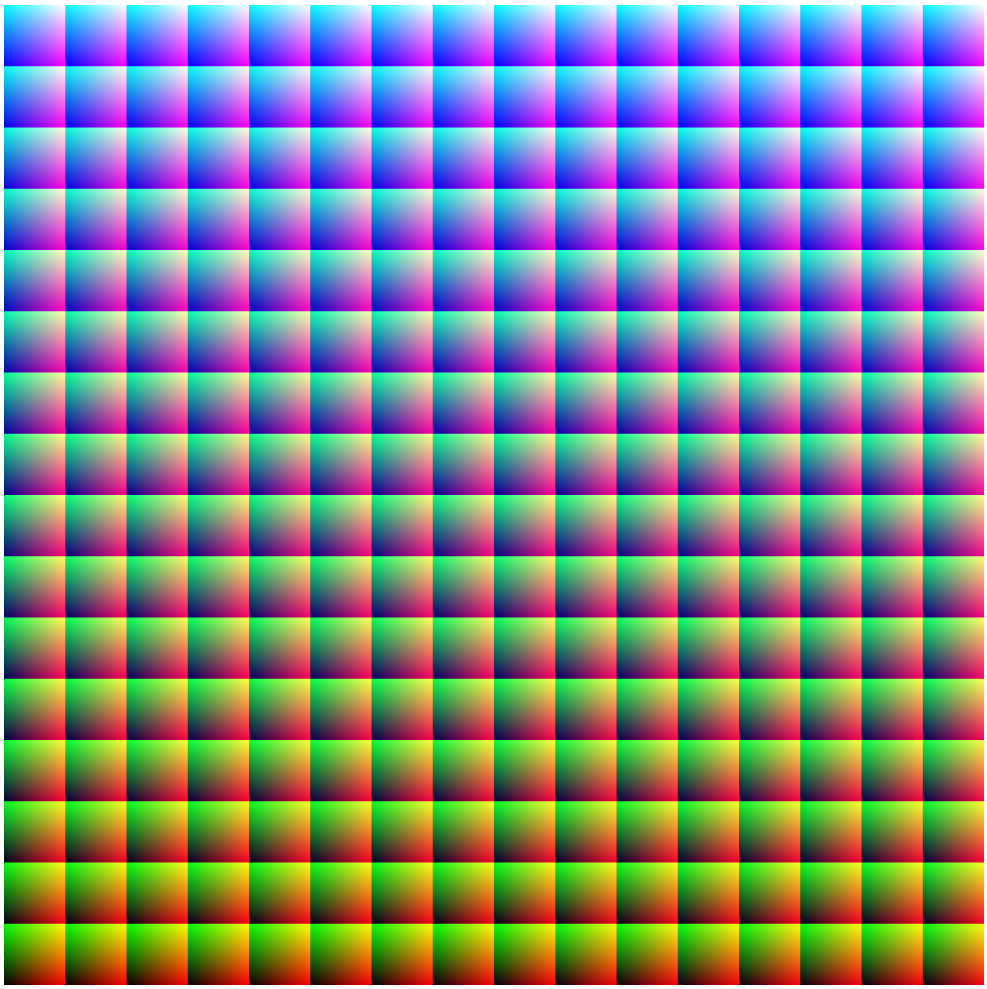

Figure 3.5: A 2D texture representing the 3D RGB colour space. Each of the 256 square tiles is a different slice of the full RGB cube; each 24-bit RGB value appears exactly once.

writing a shader to compute these in parallel. Using GLSL for creating the cache texture meant I could use the same 3D $\rightarrow$ 2D mapping function in the real-time shader and not need to rewrite it, minimising the potential for making errors.

As the palette mapping does not change as long as the palette itself remains constant, the cache can be saved in an image file and included alongside the model. The cache texture has the property that it contains exactly the number of colours in a palette; in addition, as colours with similar RGB values often map to the same palette colour, the cache texture consists of large contiguous blocks of colour. These properties mean that the image can be compressed very efficiently; a 2048×2048 cache texture stored as an uncompressed bitmap is 48MB in size, while the same image stored using the lossless indexed PNG format is less than 150KB in size, less than 0.01 of the uncompressed size. If multiple models are to be

rendered using the same palette, the cache image can be shared between them.

3.3 Palette Selection

Traditionally, the palettes used for pixel art have been hand-picked by the artist. However, as the renderer handles textured models, I decided to explore how these palettes could be selected automatically.

The number of colours in the palette varies depending on the style used. In the past, the number was determined by hardware limitations; often, the hardware would support several different rendering modes, with a trade-off between number of colours in the palette and number of sprites or sprite layers displayed simultaneously. For example, the 16-bit SNES supported four layers of 4-colour sprites, three layers of 16-colour sprites, or a single layer with a 256-colour sprite. Modern graphics hardware does not have these limitations, so palettes used in contemporary pixel art are normally only limited by the number of colours needed to express the subject.

The actual best palette to use is impossible to determine before the model is rendered, as the view and lighting of the model affects which colours are visible and require more or fewer shades in the palette. Choosing a palette when the model is rendered has its own problems. Firstly, it increases the rendering time, which a real-time application aims to avoid. Secondly, if the view or lighting of the model changes, the palette must be recomputed; this can cause visually unpleasant flickering as slightly different shades of each colour are chosen.

I opted to generate a palette by analysing the colours that appear in the texture used for the model. The problem of generating a palette to represent a 3D model is an extension of the problem of generating a palette to represent a 2D image.

One method of achieving this, the median cut algorithm, is presented by Heckbert [8] (see Algorithm 1).

To handle 3D shading, my version of the algorithm has a preprocessing step. This step copies the texture image several times, and multiplies each RGB value by a different constant in order to simulate different lighting levels. The images are combined before being used as the input to the algorithm.

A different preprocessing step stems from the observation that in real-world use cases, some colours are less common in the texture than others; for example, a face texture might be mostly pink, but have a small amount of blue for the eyes.

Median cut has the advantage that, unlike k-means (see Algorithm 2) and similar clustering

Algorithm 1 Median Cut

```
1: initial box := image
2: minimise volume of initial box while keeping all points inside
3: boxes := [initial box]
4: while boxes.length < palette size do
5:     working box := element of boxes with largest side dimension
6:     boxes.remove(working box)
7:     sort working box on largest dimension
8:     split working box into two around median
9:     boxes := boxes :: lower half
10:    boxes := boxes :: upper half
11: end while
12: palette := []
13: for each box in boxes do
14:    palette := palette :: mean(box)
15: end for
16: return palette
```

approaches, it is deterministic. An artist using median cut to generate a palette will get a consistent palette for each input texture.

However, there is one key disadvantage to using median cut – the median cut algorithm attempts to assign a similar number of pixels in the image to each different colour in the palette, and can therefore over-assign palette entries to common colours, and under-assign to rare colours. The leftmost image in Figure 3.6 shows this in effect; median cut has over-assigned entries to the pink body colour, and under-assigned the more uncommon grey and purple.

A naïve solution to this is to remove duplicate colours before running the algorithm in order to ignore colour frequencies; this is not a useful solution, as prioritising more common colours in the model for extra shades is still necessary once the main colours have been identified. A colour that appears in large contiguous areas should be assigned more lighter/darker shades in order to make those areas look more interesting in the final model; conversely, colours that only appear in small, disconnected areas already have visual interest, and therefore benefit less from additional shades. This can be seen in the central image in Figure 3.6; yellow, pink, and purple are each assigned a similar number of palette entries despite pink being much more common than yellow.

I also implemented palette selection using k-means clustering [11]. The k-means algorithm attempts to partition a data set into k clusters, minimising the total sum of squares dif-

Figure 3.6: A model of 'Scolipede'; the two images on the left show two palettes selected using median cut. The leftmost image uses unmodified median cut, while the central image had duplicate colours removed beforehand. Model and texture from *PokéPark 2* (2011; Creatures Inc.); graphic director Hiroaki Ito; the rightmost image shows how the model appears in the original game.

ference between elements in each cluster. I used an existing JavaScript clustering library by Arthur [1] to reduce the amount of code I would need to write.

As the k-means clustering algorithm does not guarantee an optimal solution, and its initial state is selected randomly, the result can vary over different executions, as shown in the left and central images in Figure 3.7. The random nature does suggest an approach to improving the algorithm, namely computing several palettes and choosing the best one.

I estimated the 'goodness' of a palette by, for each colour in the image data (including the additional shades used to compute the palette), computing the minimum CIEDE2000 difference between that and a colour in the palette, and using the squared sum over the entire texture as a 'closeness' value. The smaller the closeness, the better the palette.

The right image in Figure 3.7 shows a typical result of running k-means several times and choosing the best result. Although doing this does not completely remove the chance of a non-representative palette, it does significantly reduce the chance.

Algorithm 2 k-means

$clusters = [[] * k]$
$centroids[k] := random\ subset\ of\ points$
$continue := true$
while *continue* **do**
    $continue := false$
    for each p in *points* **do**
        $cluster = \text{argmin}_{k \in \{0 \ldots k\}} distance(point, centroids[k])$
        if $p \notin clusters[cluster]$ **then**
            move p *to* $clusters[cluster]$
            $continue := true$
        end if
    end for
    for i in 0 to k **do**
        $centroids[k] = \sum_{point \in clusters[k]} point / |clusters[k]|$
    end for
end while
return *centroids*

Figure 3.7: 'Scolipede' with three different palettes selected using k-means. The left image has a total squared difference of 1781.51, while the central image has a total squared difference of 3259.26. The right image uses the palette with the smallest total squared difference from 20 runs of the algorithm.

3.4 Lighting

Intensity stepping

The lighting algorithm is based on Phong [13] lighting; the lighting equation is given in Equation 3.8. Phong interpolation is provided by reassigning input variables to *varying* variables in the vertex shader; the GPU interpolates these per-fragment before passing them to the fragment shader.

$$I = I_a + I_l(k_d(\hat{L} \cdot \hat{N}) + k_s(\hat{R} \cdot \hat{V})^\alpha) \tag{3.8}$$

where I is the Phong lighting intensity, I_a is the ambient lighting intensity, I_l is the light source intensity, k_d is the diffuse reflection coefficient, k_s is the specular reflection coefficient, α is the Phong roughness factor, $\hat{L}$ is the normalised light direction (in world space), $\hat{N}$ is the surface normal, $\hat{R}$ is the reflected light direction, and $\hat{V}$ is the camera direction.

The end result of my lighting algorithm is similar to that produced by 'toon shading' such as that described by Decaudin [4], with discrete lighting levels on curved surfaces rather than the smooth gradients of photorealistic lighting methods.

Toon lighting computes the Phong lighting intensity as usual, then uses a step function to map it to a 'colour ramp'. This can be implemented either by quantising the intensity to the nearest step, and computing the final colour as usual, or by directly assigning colours based on the intensity, such as by performing a texture lookup into a 1D texture containing the quantised gradient.

The disadvantage of the first approach – quantising the intensity to the nearest step – is that it does not guarantee that the colours produced fall within a limited palette. On the other hand, using the Phong intensity to choose colours from a predefined set is inefficient; it requires the implementation of a function mapping texture colour and Phong intensity to a colour in the palette. The two approaches for doing this are:

1. Mapping directly from texture colour and intensity to palette colour using a 2D texture. This is inefficient due to needing to look up the colour in the mapping texture first, and gets unwieldy as the number of colours in the model texture increases.

2. Storing copies of the model texture, one for each intensity step, and rendering the appropriate texture depending on intensity. This has the drawback of multiplying the GPU memory required to store the texture by the number of intensity steps. For example, a 512×512 texture consumes 1MB of graphics memory; four intensity steps means an additional 3MB per model texture.

Alternatively, the model could be shaded using Phong lighting, then the final image quantised to the palette; this approximates toon shading reasonably well, and as it is a post-processing step that is performed independently of rendering, is very flexible. However, it has the disadvantage that, depending on the palette, intensity steps on the rendered image might not line up on colour boundaries, as can be seen in the CIEDE2000-shaded teapot in Figure 3.4 at the boundary between red and yellow.

The disadvantages with the different approaches can be eliminated by combining the various methods. The only drawback to using Phong lighting, then quantising the resulting image, is that the steps do not align between colours. I avoided this by quantising the Phong intensity before multiplying it with the texture colour, as is done in toon shading. This removes any gradients in the image that could cause intensity steps to fail to line up when the image is quantised again to the colours in the palette; the second quantisation is necessary in order to ensure the result uses only a predefined number of colours, as toon shading guarantees no such thing.

Dithering

Dithering is the technique of alternating two colours in a pixel-level pattern in order to give the illusion of a third colour, a blend between the two colours used in the dither pattern. The limited visual acuity of the human eye means if the pattern is small enough, we cannot discern individual pixels, and the eye blurs the pixels together. The use of dithering ties into the limited palettes used in pixel art; it can simulate shades of colour not present in the palette, such as the intermediate shades of blue and orange seen in Figure 3.8.

I implemented dithering on the edges of lighting intensity steps. I used the final fragment coordinate to decide whether a fragment lay on a light or dark pixel in the checkerboard pattern, and modified the fragment's Phong intensity accordingly before quantising.

Listing 3.2: Simple dithering in GLSL

```
phongIntensity += (mod(gl_FragCoord[0] + gl_FragCoord[1], 2.0)) * ditherFactor
    * 2.0 - ditherFactor;
```

Let 'even pixels' be the pixels such that the sum of their x and y coordinates is even, and 'odd pixels' be those such that the sum of their x and y coordinates is odd. The dithering algorithm adds a $ditherFactor$ constant to the intensity of odd pixels, and subtracts $ditherFactor$ from the intensity of the even pixels.

As this is performed before quantising the intensity, the results are only visible at the edges of shading steps, where $ditherFactor$ is larger than the difference between the quantised

Figure 3.8: Pixel art of 'Charizard' from *Pokémon Black 2/White 2*. Note the use of dithering on the wing and tail.

and unquantised Phong intensity. In the middle of a shading step, the quantised and unquantised Phong intensities are further apart, so $ditherFactor$ is not large enough to cause dithering. Figure 3.9 shows the results of different values of $ditherFactor$.

Figure 3.9: 'Scolipede' rendered with three different levels of dithering. From left to right, $ditherFactor = 0.01$, 0.02, and 0.05.

Simply adjusting lighting intensity is a naïve method that can only handle dithering of intermediate shades of the same colour. To extend the technique to handle other cases,

such as dithering a gradient between two different colours, an element of error correction can be used. This algorithm performs error correction instead of just adding/subtracting $ditherFactor$:

Algorithm 3 Error-corrected checkerboard dithering

```
1: paletteColour = findClosestPaletteColour(pixelColour)
2: if even(pixelCoordinates) then
3:     return paletteColour
4: else
5:     oppositeColour = 2 × pixelColour − paletteColour
6:     return findClosestPaletteColour(oppositeColour)
7: end if
```

The algorithm finds the nearest palette colour for all even pixels, while odd pixels attempt to correct the error caused by quantisation. If the error is small (i.e. the palette colour chosen is a good match for the input), the odd pixels will be quantised to the same colour as the even pixels; if the error is large, the odd pixels will be quantised to a different colour, introducing dithering.

Rather than using a simple checkerboard pattern, I considered a version of Floyd-Steinberg dithering [6]. However, serial error correction algorithms such as Floyd-Steinberg are incompatible with GLSL's parallel architecture; Floyd-Steinberg considers each pixel of the image in turn, using the dithered colour of the preceding pixel to decide the colour of the pixel under consideration. GLSL only provides read-only and write-only buffers (textures and render targets); serial error correction requires the use of a read/write buffer.

3.5 Outlines

Detecting Outlines

Characters and objects in video games must be able to be located and identified quickly across a range of different backgrounds. The use of a one-pixel-wide outline ensures that the character remains distinct from the background regardless of the background colour. In addition, with a limited palette available, similarly-coloured parts of the character may be hard to distinguish as the shading cues a viewer would normally use are reduced; outlining within the character's silhouette adds definition to the interior.

There are two commonly-used methods for rendering outlines. The first method is to draw the front-facing polygons (polygons with the outside normal facing towards the camera) of a model normally, but render the back-facing polygons as solid-coloured wireframes, offset

slightly towards the camera. This results in a one-pixel-wide line being drawn on top of any edge where a front-facing and back-facing polygon meet, i.e. the outline of any part of the model that occludes another part of the model. The method works assuming the model is a closed surface.

The second method is to render the depth map of the scene to a texture, then use an edge-detection algorithm such as the Sobel operator [14] on the depth texture to find discontinuities in depth. These indicate an edge that should be outlined.

The advantage of the wireframe method is that by setting the line width to one, the outline can be rendered with single-pixel width immediately. The edge-detection method gives wider outlines that must be reduced to one pixel in width first. In addition, it is possible for the edge-detection method to give spurious edges; if the edge detector sensitivity is too low, it will fail to find some edges, but if it is too high it may find edges where no edges actually exist.

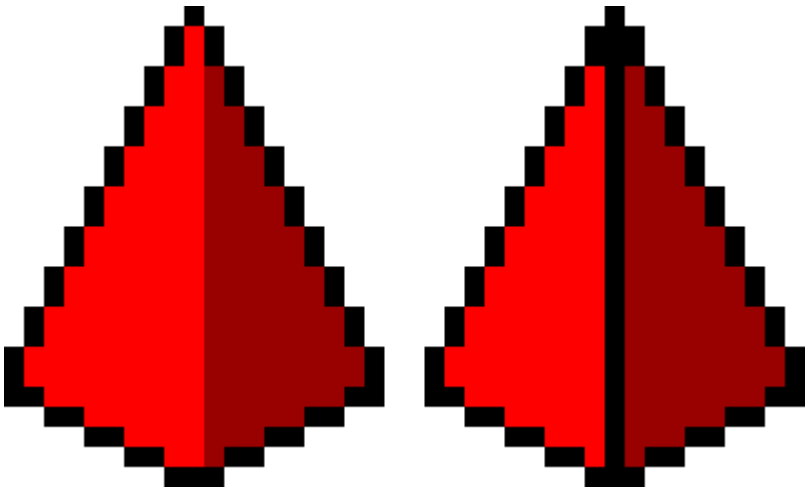

Figure 3.10: The pyramid on the left is only outlined where front faces meet back faces. The pyramid on the right also has an outline along the edge where the front faces meet.

The edge-detection method can also be applied to the scene normals; this allows it to detect outlines where two front faces meet, as shown in Figure 3.10. These outlines should not be applied to every edge between two polygons; if this was used on a curved surface, the polygons used to smooth the curve should not be outlined. By increasing the edge intensity threshold, we can ignore edges between polygons with a small exterior angle, and only outline edges between polygons with a large exterior angle.

I chose to use the edge detection method due to its ability to work with edges between front-facing polygons, and because implementing the wireframe method would require finding a work-around for OpenGL ES's lack of the `glPolygonMode` function used to render

Figure 3.11: Pixel art of 'Whimsicott' from *Pokémon Black 2/White 2*. The leftmost image has all outlines shaded with the same colour, while the central image utilises various outline shading techniques. The rightmost image is a close-up showing anti-aliasing.

ensures the shader cannot get stuck in situations where one of the surrounding pixels quantises to the darkest colour in the palette. The GLSL code for the algorithm is given in Appendix D.

Another shading technique used is to anti-alias the outlines. An example of this is shown in Figure 3.11; lighter pixels are used at the ends of columns of black pixels in order to make the switch between columns smoother.

I implemented this using a variation of *fast approximate anti-aliasing* (FXAA) [10]. First of all, instead of outputting just 'true' or 'false' from the edge detector, it outputs the edge intensity – the maximum depth or normal difference with a neighbouring pixel. This allows me to determine whether a pixel lies in a row or column of outline pixels – in a row, the neighbours to the left and right are also on an edge, and therefore have a higher total intensity than those to the top and bottom, and vice-versa for pixels in a column.

After determining whether a pixel is on a row or column, the algorithm works in both directions along that line of pixels until it reaches the pixels at the end of the outline (their outline intensity is smaller than a threshold value). It then computes the pixel's distance along the line, and the fraction of the line that lies below or to the left of it.

I modelled the intensity along a line of pixels as two constants weighted by a sine curve; the intensity is high in the middle of the line, and smaller towards the edges. I then multiplied the colour of the pixel underneath by the intensity, and ensured that it was darker than the surrounding pixels as before.

Figure 3.12: From left to right: single-colour outlines, colour based on the pixel underneath, outlines made darker than neighbouring pixels, and anti-aliased outlines.

3.6 Detail Preservation

Details in a model can be lost when the model is rendered at a small size and with a limited palette. As these are common features of pixel art, there is a need to avoid this problem by implementing algorithms to preserve details.

Detail loss caused by a limited palette is inevitable. This loss of detail is reduced by the use of outlining and careful palette selection, as described in sections 3.5 and 3.3.

However, when preserving detail across animations, we can go further. Sprite-based video games devices allow palettes to be switched on the fly, or occasionally even part-way through a screen refresh, allowing the system to give the impression of a sprite with more colours than are supported by the hardware. Although this is not an issue with modern graphics hardware, simulating it can make pixel art seem more authentic.

One way of doing this is to generate a palette slightly larger than the palette size; if we want to find a palette P of size p, generate a palette Q of size q. Then, render the model using Q. Calculate the squared distance of each pixel's Phong-shaded colour from its Q-quantised colour, and find the $q - p$ colours such that the sum of squared distances of pixels quantised to those colours is minimal. Remove those colours from Q to obtain P. However, as colours in use may be removed from P, there is the possibility of unpleasant flickering as the palette changes.

Alternatively, rather than starting with a larger palette and selecting a subset, we can start with a smaller palette and add extra colours when needed, only removing colours again when that colour is no longer present in the render. This removes the risk of a colour being removed while still in use, but also requires that the palette mapping be tracked across frames to prevent the reverse problem of new colours being added causing a colour already in use to switch.

I chose not to implement additional palette detail preservation across animation frames in this way both because it is not used in the *Pokémon* pixel art style, and because generating

Figure 4.2: The Stanford dragon rendered at 128×128 pixels, with both Phong shading (left) and pixel art shading (right).

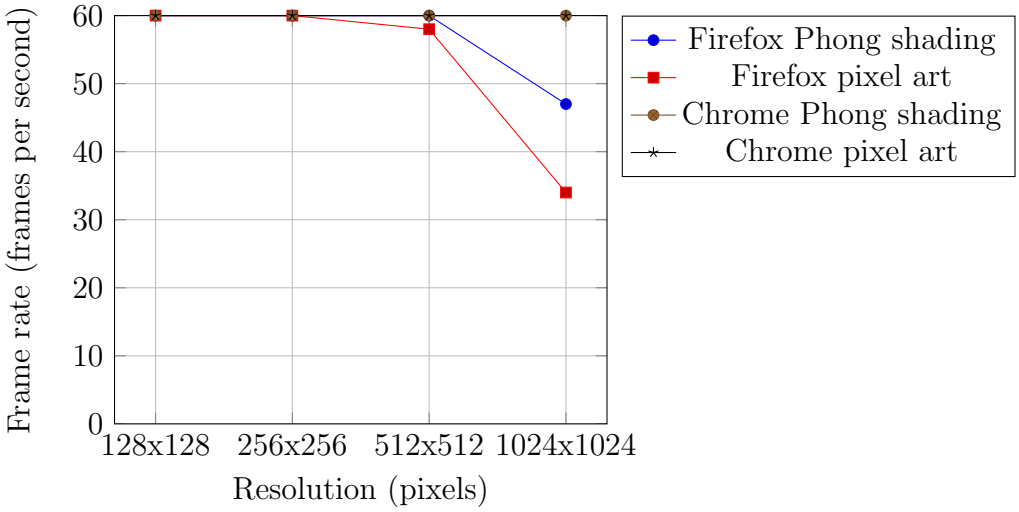

Figure 4.3: Frame rate while rendering a single reduced-polygon Stanford dragon on a NVIDIA GeForce GTX 765M GPU.

render code was called 60 times per second.

This limit proved to be detrimental when testing on Chrome, as its WebGL renderer easily maintained 60FPS at all tested resolutions. This made it impossible to evaluate the performance difference between my pixel art shader and the Phong shader. In order to collect useful results, I simulated an increase in scene complexity and resolution by rendering the model 8 times per frame; this gave the results seen in Figure 4.4.

I repeated this on my desktop, with a NVIDIA GeForce GTX 560Ti GPU, and identical browsers. The results are shown in Figure 4.5.

The total number of polygons rendered was just under 700,000 per frame. Modern video games often display millions, or even tens of millions, of polygons per scene at resolutions of 1920×1080 and above – my system would be inadequate for use in such a situation.

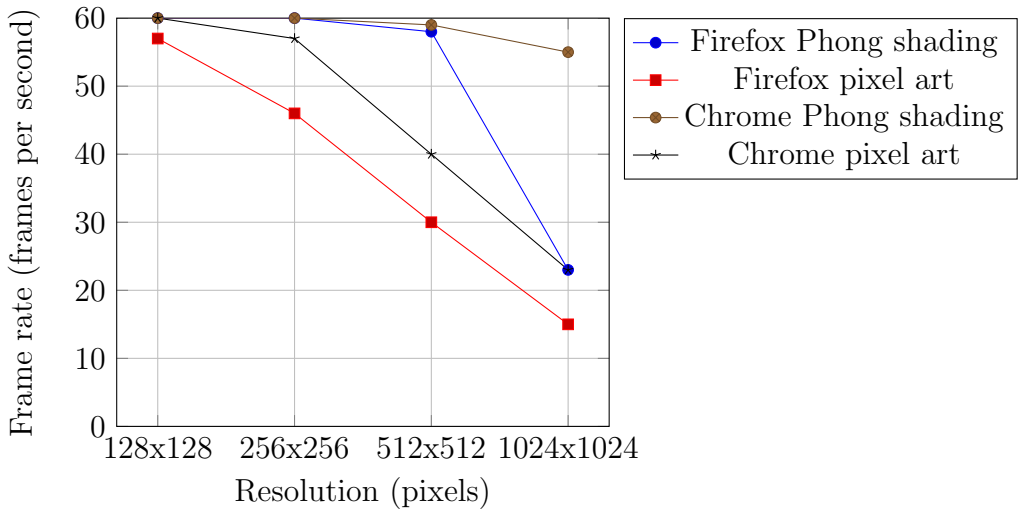

Figure 4.4: Frame rate while rendering 8 copies of the reduced-polygon Stanford dragon on a NVIDIA GeForce GTX 765M GPU.

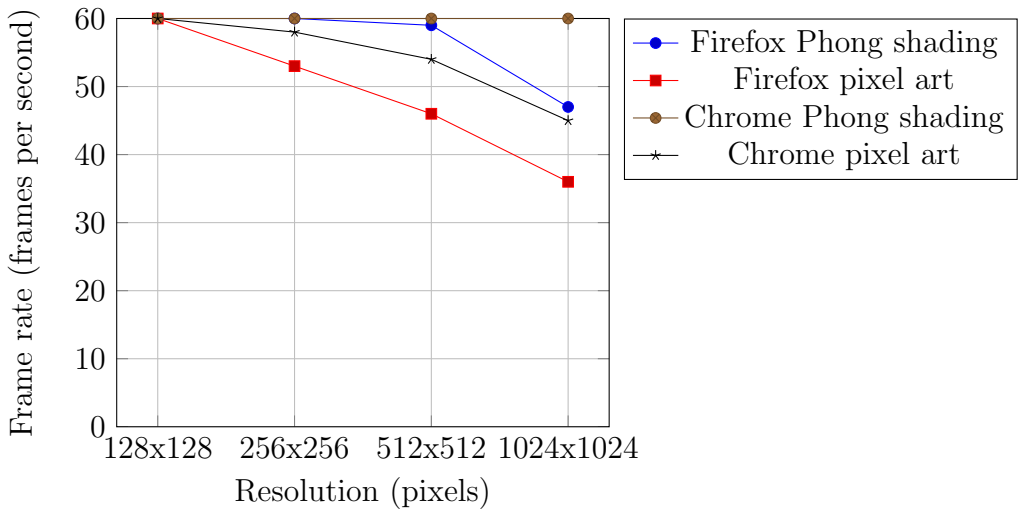

Figure 4.5: Frame rate while rendering 8 copies of the reduced-polygon Stanford dragon on a NVIDIA GeForce GTX 560Ti GPU.

However, due to the detail loss inherent in palette quantisation and the common practice of scaling pixel art up before rendering it in order for the viewer to more easily appreciate the pixel-level detail, such high polygon counts and resolutions would not be required for many use cases of the system.

As the aesthetic aspect of the project was important, I performed some informal tests to roughly gauge the effectiveness of parts of the project.

I found five people who volunteered to test parts of the system. Two of those were familiar with pixel art already through playing video games.

I asked them to compare two images, one which used a particular technique and one which did not, and say which they thought was clearer, and which they preferred. I randomised the order of the images each time to avoid any bias. An example is shown in Figure 4.6.

Figure 4.6: Black outlines on the left compared with coloured outlines on the right.

Overall, I found that in most cases, three or more participants preferred the appearance of models shaded with pixel art techniques rather than those shaded without them. However, the reverse was true for clarity; in most cases, the additional pixel art features such as dithering made the images less clear. For example, three people found a Phong shaded dragon clearer than the pixel art dragon (using the image in Figure 4.2).

I found there was a large difference in opinions between participants who were familiar with pixel art already and those who were not. Those familiar with pixel art preferred the images with additional pixel art features in over 80% of cases, while those unfamiliar with it were more evenly split.

Without formal human testing, I cannot be certain that my project was successful; I did not treat the above informal study as a measure of success, though the results from it were encouraging.

As I chose to emulate the pixel art style used in *Pokémon Black 2/White 2*, I attempted to recreate sprites from those games. The results are shown in Figure 4.7 and Figure 4.8.

I created histograms for the frequency of each colour normalised for the non-transparent area of each sprite for both the rendered and hand-drawn images, and computed the differences between them. These histograms are shown in Figure 4.9 and Figure 4.10.

The differences between the histograms reflect the discrepancies between the computer-generated pixel art and the hand-drawn pixel art; the following paragraphs explain these differences.

The models I used have slightly different proportions and pose than the game sprites; for example, Scolipede's horns and tail are thinner in the model, while Eevee's pose is different.

Figure 4.7: From left to right: Scolipede rendered with an automatically-selected palette, Scolipede rendered with the original pixel art's palette, and Scolipede's original art from *Pokémon Black 2/White 2*.

Figure 4.8: A model of 'Eevee' from *PokéPark 2* rendered using the pixel art shader, and Eevee's original art from *Pokémon Black 2/White 2*.

Assuming the 3D model has the correct proportions, Scolipede's sprite shows how the pixel art form modifies proportions for clarity; my detail preservation algorithm is not flexible enough to imitate this. Interestingly, note how the silhouette of the horns and tail on the 3D model render is proportionally similar to the area within the outline on the hand-drawn

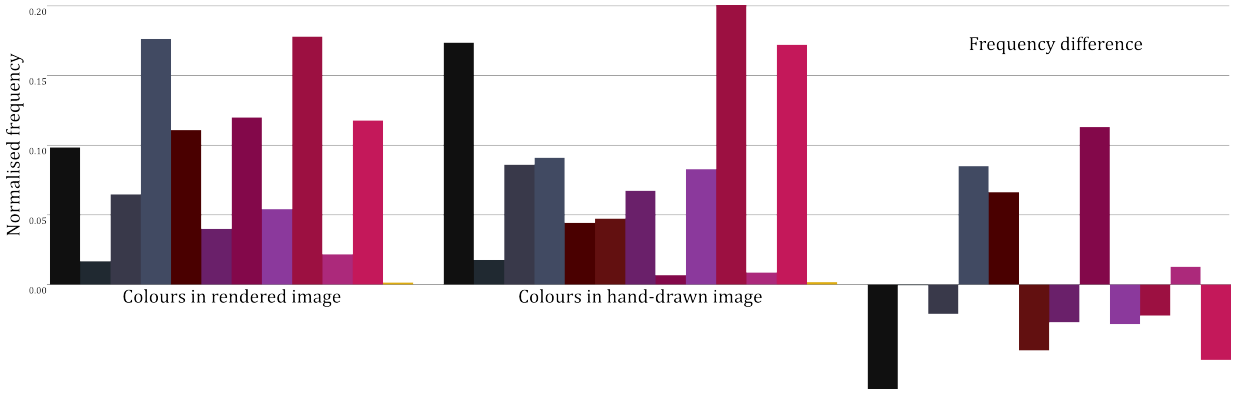

Figure 4.9: Histograms comparing the normalised frequencies of each colour in the rendered image of Scolipede and the hand-drawn art.

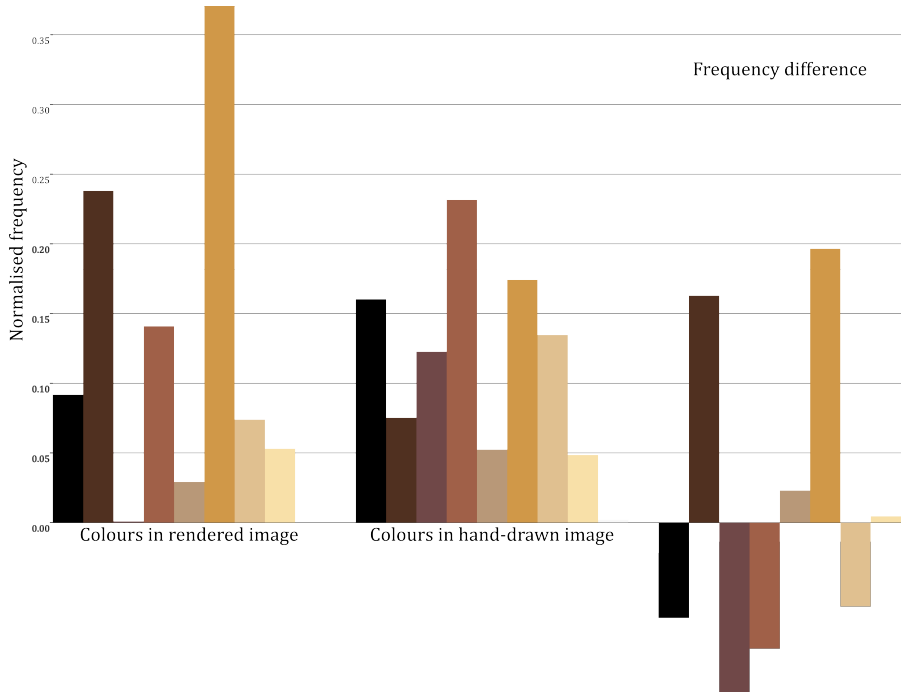

Figure 4.10: Histograms comparing the normalised frequencies of each colour in the rendered image of Eevee and the hand-drawn art.

sprite – rendering the outline on the outside of the model makes the proportions of the horns and tail match the hand-drawn art better, but also makes parts such as the legs look unnaturally bulbous, as in Figure 4.11.

Ignoring discrepancies in the model, there are a few further apparent differences between the rendered and hand-drawn versions. The most noticeable is that the rendered sprites have fewer outlines. For example, Eevee's sprite lacks outlining on its mane and between

Figure 4.11: Scolipede rendered with the outline on the outside of the model.

its toes. In addition, both Scolipede's and Eevee's rendered sprites use about 40% less black in outlines, as can be seen in the histograms in Figure 4.9 and Figure 4.10. This shows that the outlining algorithm is a possible area of improvement; for example, a better algorithm could take into account the texture as well as the depth and normals.

The second is that the rendered sprites use more colours for shading than the hand-drawn sprites; the rendered Scolipede appropriates the blend between red and purple used at the edge of the rings as an additional shade of red, as is evidenced by the spike in the difference histogram in Figure 4.9. This is because the shading algorithm treats all colours in the palette equally in order to approximate the shading as closely as possible, while a human pixel artist reserves certain colours for certain parts of the subject. The latter approach is useful when 'palette swapping', a technique commonly used in video games to create variations of characters, is used (see Figure 4.12); if palette colours are associated with actual colours, palette swaps can be created by modifying the palette without changing the bitmap.

Interestingly, the rendered sprite does not use one of the colours from the hand-drawn sprite, a shade of dark red (seen in Figure 4.9 between the grey and purple bars). This is due to the rigid shading algorithm; none of the Phong shading levels for any colour in the texture were matched to that particular shade of red, which is used as an additional outline colour in the hand-drawn sprite.

Finally, the lighting is different between the rendered and hand-drawn sprites. This is down to two reasons – I did not implement shadowing, so parts of the model that should be in shadow, such as Eevee's tail and back leg, are lit. This is made obvious in Figure 4.10; the colours that are used more commonly in the render than in the hand-drawn art are the lit colours and the lighter outline colours, while the shadowed colours are neglected in the render. Additionally, a human artist can use 'artistic license' when shading, while my system shades rigidly based only on light source position. For example, Scolipede's

Figure 4.12: 'Shiny Scolipede', an example of a palette swap in *Pokémon Black 2/White 2*. Note how the shading on the left, rendered, sprite is different to that in Figure 4.7 while the shading on the right, hand-drawn, sprite is identical.

foreground horn on the hand-drawn sprite is completely illuminated, implying the light source is to the side, in the direction of the viewer; most of the other shading implies the light source should be above it. Although incorrect, illuminating the foreground horn and shading the background horn gives a sense of depth. My shading algorithm does not reproduce this technique.

As with any computer-generated emulation of a hand-drawn art style, the output of the program is not entirely correct. However, in the intended use-case of real-time pixel art rendering, the smooth animation produced hides many of the imperfections. By the art form's nature, errors are generally only a few pixels in size; displaying these in a real-time animation means each error is only visible for a fraction of a second. These two factors mean that most errors are difficult for viewers to discern.

While I was implementing the project, I kept a suite of unit tests up-to-date for both my JavaScript and GLSL functions. This allowed me to ensure critical parts of the system, such as the code to pack floating-point numbers into colours, would work correctly. I made sure to cover edge cases such as 1/256 as well as ordinary use cases.

Chapter 5

Conclusion

The implementation of the project was generally successful. The system's output is a good approximation of hand-drawn pixel art; the project, or at least some of the methods used in the implementation, has the potential to see use by video game developers wishing to add a pixel art look to a game. As mentioned in the introduction, using 3D models rendered as pixel art has several advantages over hand-drawing the art in both visuals and production effort.

The main failure of the project was due to not strictly keeping up with my schedule. Falling behind led to a lack of time for evaluation, which in turn meant I cannot claim to have met my success criteria; in hindsight, I would have cut part of the implementation short in order to have more time for evaluation, potentially returning to complete it afterwards.

Potential extensions to the project that could be performed in the future include porting the code to a library or engine more commonly used in game development; WebGL is very suitable for demonstrating the project and for portability, but is rarely used for video games.

The shading system could also be improved; two things mentioned in Chapter 4 are shadowing and associating particular colours with parts of the model. Shadowing can be implemented using an algorithm such as shadow mapping [16], while colour association could be performed by dividing palette colours into sets, mapping certain texture regions to palette subsets.

Introduction and Description of the Work

As computer graphics technology has advanced over the years, original artistic styles have evolved with it.

Early technology was scanline-based, capable of drawing simple blocks of colour. As the fidelity demanded of graphics increased, specialised hardware that could deal with complex two-dimensional scenes was developed; however, it was still limited in many ways. Two widespread limitations were that higher-speed graphics modes restricted the palette of colours that could be displayed per 2D object ('sprite'), and the output resolution of the hardware was relatively tiny compared to today's technology (for example, 320×240 pixels was commonly used).

Figure 1: An early example of pixel art, from Space Invaders (Taito, 1978)

The combination of these restrictions led to the rise of an art style now known as 'pixel art'. This was characterised by bright colours with distinct levels of shading, dark single-pixel outlines that contrasted against often busy backgrounds, and thanks to the low resolution, the use of single pixel positioning to show small details. As the style evolved, further techniques were invented, such as the use of several shades of the same outline colour to make edges appear smooth, and 'dithering', a pixel-grid checkerboard pattern between two shades of a colour to give the illusion of a third intermediate shade.

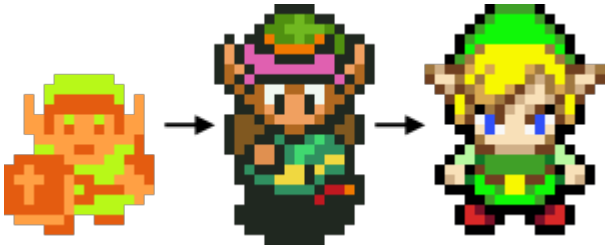

Figure 2: The evolution of pixel art, from The Legend of Zelda series (Nintendo, 1986-2005)

Eventually, 2D graphics hardware was replaced by 3D hardware, and pixel art fell out of general use due to the requirement to painstakingly hand-draw pixel by pixel, for each of

dozens or even hundreds of frames of animation. In terms of artistic effort required, 3D is much easier to work with for complicated scenes and characters. However, with the rise of mobile devices with limited graphics hardware and users looking fondly back at the computer games of their past, pixel art remains a popular art style.

The project intends to combine the ease of 3D animation with the aesthetics of pixel art, by using modern graphics technology to render 3D models in the style of pixel art. As the style is mainly used in computer games, the code will perform this in real time, allowing developers to use the pixel art style dynamically in a way that is impossible with hand-drawn art.

Starting Point

I worked with GLSL, specifically fragment shaders, on a project at the University Computer Lab in Summer 2013, and also have some experience writing OpenGL code. I have drawn pixel art graphics in the past and consider myself to be experienced with the art style.

Figure 3: Examples of pixel art graphics I have drawn

Substance and Structure of the Project

The main aim of the project is to implement OpenGL and GLSL code to render 3D models in a 2D pixel art-like style. The project will allow a 3D scene and corresponding animations to be loaded from a file, then render it to the screen at a smooth framerate. At the beginning of the project, I will choose a programming language and a corresponding OpenGL binding to use. I am considering C++, Java, and WebGL.

I will decide on a file type to use for loading 3D models. There are many file types designed for this purpose; the chosen type must support texturing, different materials, and animation. Ideally, it will be simple to parse so I can implement the parser quickly.

Next, I will need to write a skeleton renderer. At the beginning of the project, this will render the model using the standard Phong shading algorithm. As the project progresses, I will gradually implement custom rendering code instead; this approach will allow me to see the output of my code as I implement it. This means I can spot any bugs I insert quickly, and fix them.

As pixel art uses a limited colour palette, I will need to implement an algorithm to select a palette from the input model and texture data. This will need to generate new colours for areas of the model in shade and directly in the light. As stated above, I will also include the option to specify a palette for the model manually; this will be added first, allowing me to focus on the rendering code. I will return to the problem of selecting a palette later in the project, using the median cut algorithm as a starting point.

The part of the project that applies block shading will be relatively quick to write, as it is similar to the existing 'cel shading' algorithm; it differs mainly in that it needs to select colours from a predefined palette instead of performing a 1D texture lookup. It will need to support ordinary 2D texture mapping. I will also implement dithering in this step, looking at the material properties and surface normals of the model to choose areas where the effect should be used. All of this will be implemented solely using fragment shaders.

Single pixel outlines can also be implemented using a fragment shader; note that outlines occur when a front-facing polygon meets a back-facing polygon, so an outline can be drawn by rendering front-facing polygons as usual, then setting the depth test to allow fragments with an equal depth and rendering back-facing polygons. The front-facing polygon depth data still exists, so only pixels along edges will pass the depth test and be rendered.

Line smoothing will be implemented by extending the above with a vertex shader passing data to a geometry shader. The edges to be outlined must be identified on a separate pass before the model is shaded; then, the geometry shader can subdivide polygons along the edge to ensure the pixels along the edge appear smooth. Another fragment shader will be

Figure 4: An example of cel shading (from Wikimedia Commons)

Figure 5: Examples of types of dithering (from Wikimedia Commons)

used to add further shading to the outline.

Figure 6: An example of pixel art line smoothing. A curve rendered by a painting application is on the left, while a smoothed version is on the right.

The most difficult part will be to implement preservation of detail at small sizes and across frames; for example, if a detail ends up being smaller that a single pixel, it may not be rendered normally. This will be done by identifying details smaller than a threshold size (possibly depending on the type of detail; for example, one-pixel outlining means geometry details will need to be at least three pixels wide in order for the actual material colour to be visible), and using a geometry shader to expand them. Attention will need to be given