Generating Normal Maps from 2D Textures

10th of April 2018

Acknowledgements

I would like to give my thanks to Edinburgh Napier University for providing me with the skills and knowledge required to undertake this project. I would like to thank Dr Kevin Chalmers and Dr Ben Kenwright for teaching and inspiring my interest in computer graphics.

I would also like to thank my supervisor, Prof Ben Paechter, who has guided me throughout this project, as well as Jyoti Bhadwaj who helped me structure this document.

Lastly, to the numerous people who participated in my surveys, I thank you for your willingness to participate, and for your contribution to the project.

Contents

Abstract

This paper outlines the methodology and research undertaken to develop a program that systematically generates normal maps from albedo textures. It explains the algorithm used within the program and compares the program written to a piece of commercial software created for the same purpose. Findings from a user-survey are included to evaluate the usability of the program and the aesthetics of the normal map generated with it. These are considered as reflections for future work.

1. Introduction

1.1 Background

The games industry is a multi-billion-dollar business and is therefore important; but making big-budget games is no trivial thing to do. They require workers from many different disciplines to come together to make a cohesive product. These include artists, sound designers, engineers, actors and directors.

Due to the complexity of making games, these professionals will often use tools to help them in their endeavours. These include games-engines for level designers (people who decide the 3D environment that the player, or end-user, interacts in) and image-editing software for artists who design textures.

The technology behind the tools is discussed further in the section: 3. Technology Background

1.2 Aims and Objectives

1.2.1 Aims

The aim of this project was to produce a program that could be used by games artists to automatically generate normal maps (Blinn, Simulation of Wrinkled Surfaces, 1978) from textures. These normal maps are used in games to improve their realism. The textures they were generated from, were the images that were applied to in-game surfaces. The program was made with a mind to be usable and produce aesthetically pleasing results.

1.2.2 Objectives

- Research existing algorithms
- Research existing solutions to the problem
- Create a program based on the research
- Add desirable features to the program
- Evaluate the effectiveness of the product with the use of a survey

1.3 Scope

The target market for this product was game artists and independent developers. Upon completion, the tool was released to the public free-of-charge through a blog. It is an alternative to paid tools and ones that are more complex.

1.4 Constraints

The program itself was written in C++ and GLSL (The OpenGL Shader Language) as they are relatively low-level and so faster for the computer to execute (DirectX was used in the past, but it was harder to work with). Because it is a C++ program, anyone with the source code can compile it to run it on their machine if they have a graphics card or chip that supports OpenGL; specifically, version 4.1 – this was chosen because it supports the features required by the program.

This program was compiled for Windows machines, and the executable file for it was released publicly, along with the source code. This is because Windows is currently the most popular operating system.

For this project, two different algorithms were adapted to do the calculations to create the normal maps, but they were altered to make sure they worked properly and efficiently.

1.5 Sources of Information & Libraries

Existing programs were observed and analysed, to see what they did well and what they didn't – they were also used as sources of inspiration.

To fully evaluate and demonstrate the product and what it produces, multiple 3D simulated environments were created. To do this, Unreal Engine 4 was used; this is because the author had previous experience with this, more so than other engines like Unity. A new "environment simulator" wasn't created, as it would have been time consuming and unnecessary when tools like Unreal that can do this already exist; and it was not the focus of the project.

A large amount of information about graphics programming has come from online OpenGL tutorials; particularly *Learn OpenGL* (Vries, 2014), from which a single class was taken to compile shader programs into machine code.

To use OpenGL effectively, the GLEW (The OpenGL Extension Wrangler) framework was used, this library gives the programmer a higher-level perspective of the low-level OpenGL functions.

The GLFW (Graphics Library Frame-Work) was also used so that windows could be created and used in a cross-platform setting.

So that images can be manipulated, SOIL (The Simple OpenGL Image Library) was used. This contains functionality that allows the programmer to save images to the file system as well as to RAM (not VRAM) so that they can be operated on. To give an example, it is very tedious to count pixels using the GPU (as it is more adapted to parallel operations than sequential ones such as counting) so they need to be loaded into the CPU.

1.6 Chapter Outlines & Structure

Chapter 1: Introduction of the subject and problem

Chapter 2: Literature Review

Chapter 3: Problem Brief and Analysis incorporating the findings from the literature review to set the stage for the project

Chapter 4: Methodology outlining the steps taken to create the product and evaluate it

Chapter 5: Results containing the findings and analysis of the survey

Chapter 6: Conclusions & Future Work drawn from the analysis and from the project and setting out a proposal as to how the project could be improved in the future

Chapter 7: References

2. Literature Review

2.1 Introduction

This section contains a review of the literature and programs used to glean background information on the generation of maps from textures. The original problem posed was to generate materials from a single texture. This has been developed into generating only a normal-map for that texture.

To expand on the abstract, when developing a game, the game's creators will often use photo-realistic texturing on surfaces to make them look real; not all games do this, but it is a common choice. To achieve such a realistic effect, photos of real world surfaces are taken and applied to the corresponding surfaces in the game; for example, an image of bricks could be tiled on a wall of a house in the game. This is not always the case, often an artist will create a texture instead of taking a photo.

To increase performance the house wall will most likely be flat and made up of 2 polygons (triangles) but in real life, the wall is not flat, so when the brick texture is applied to it, the wall looks like a flat surface with a brick wall-paper on it.

To get around this problem other textures (maps) can be applied to the same surface. These maps store different information about the wall in the form of coloured pixels. For this project normal maps were chosen as the focus, but bump maps were also investigated.

A normal is a type of vector. It is a unit-vector (has a length of 1) and is perpendicular to the surface it lies on. Since the project worked in 3-dimensional geometry, it had 3 components, an X, Y, and Z. In an image its 3 components were stored as an RGB pixel colour values. This allowed each point / pixel in the texture to be facing in a unique direction on a surface. Below is an example of a normal map. It shows the top of a torus, cone, sphere and a truncated square-based pyramid. Normals are used in lighting calculations, a map is shown in Figure 1.

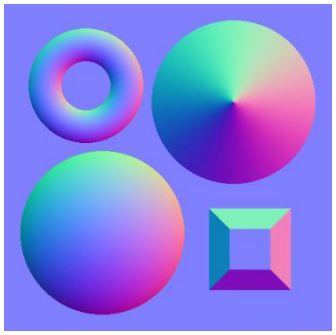

Figure 1: A normal map of various shapes

A bump map is a type of texture that holds information about how high points on the surface of the geometry are. It has the advantage over normal maps in that it appears to change the shape of the surface with little performance impact. The normals do have to be calculated after this step but game engines do this automatically. From a programming perspective, they are also simpler than normal maps to comprehend when generating them – darker areas are depressed, and lighter areas are raised.

This concept can be taken further to do displacement mapping, which does physically change the geometry of an object using a geometry-shader, but this is non-trivial.

Here is an image comparing a normal map and bump map applied to a cylinder with an orange slice albedo texture.

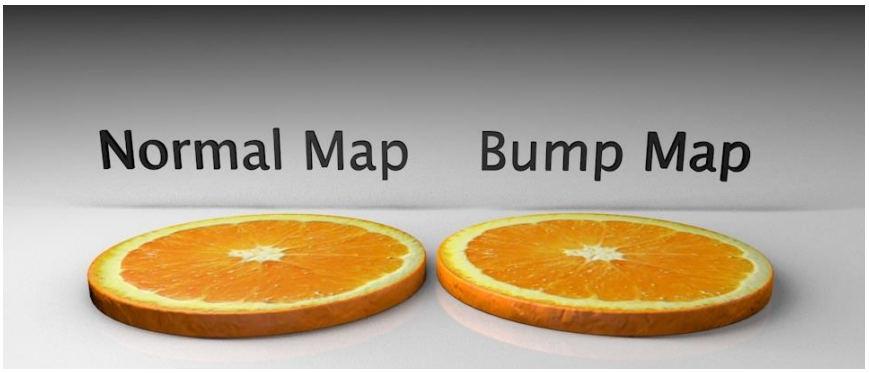

Figure 2: A normal map versus a bump map

Although the difference is subtle, the object with the bump map does appear to be more realistic.

For this project, a program for game artists to use was created. It generated normal maps to use in materials. It aimed to be accessible, so that any artist could use it. So, they could select an image and press a button to generate the texture. Some additional features included are: a visual representation of what will be generated before it is saved, as well as a parameter to tweak the strength of the effect - i.e. how deep / shallow the surface should be.

Most of the papers researched contained information regarding processes to guess the distance of objects from the camera in certain images. The reason they were researched was to get an idea of what might be involved when predicting the depth of a surface texture which is needed to generate various maps.

2.2 3-D Depth Reconstruction from a Single Still Image (Saxena, Chung, & Ng, 2006)

This paper detailed the process of using machine learning in combination with multi-scale Markov Random Fields to predict the depth of a monocular image. It also used sample images to compare with sections of the monocular image to detect edges.

Unfortunately, this approach conflicted with the desired one as their algorithm could only be implemented on a CPU, and this program aimed to run on the GPU to show the real-time preview.

2.3 Real-time Volumetric Surface Reconstruction (Ondrúška, Kohli, & Izadi, 2015)

Even though this paper focussed on mobile technology, when they stated, "Note we only use the internal RGB camera, and all computation is performed on the device"(p. 1.), it was implied that their approach only used RGB information, which could be applied to an approach that used a PC. Unfortunately, it used the 6 degrees of freedom motion tracking to create the maps and geometry and not the RGB image and therefore could not be used on a PC.

To achieve the same result with a similar approach, two or more images of the surface would need to be taken at different angles. The aim, however, was to only need one image.

2.4 Floating Textures (Eisemann, et al., 2008)

In this paper a method to perform texture mapping was presented. It was of interest as it involved using projections, which require multiple images, whilst this technique was considered the preference was to only require one image as an input.

The paper does however provide some interesting knowledge on analysing the image to determine its focus.

2.5 Depth Map Design and Depth-based Effects with a Single Image (Liao, Shen, & Eisemann, 2017)

This solution was very appropriate to the problem - generating a depth map from a single image. It did however require a level of user interaction that was undesirable; in this, the user needed to draw on a layer above the image to help the computer decide what the foreground and background of the image each were.

Their program was written in Java and was targeted to run on a CPU. Unlike the *3-D Depth Reconstruction from a Single Image's* approach, this one could have been translated, in part, to run on the GPU.

2.6 Monocular Depth Map Prediction (Kuznietsov, Stückler, & Leibe, 2016)

This approach used deep learning to predict the depth of points in an image. It also used LiDAR to assist in determining the depth-maps of training images. One takeaway gleaned from this approach was to use something called a ground-truth which can assist in acquiring a sense of perspective in the image.

Initially, it was thought to ask the user to select a pixel in the image by clicking on it so that a base colour could be set. This would have been the deepest point in the image and could have been used to calibrate the depths of the rest of the image. This idea was later adapted to the levelling method presented in 5.3 Changes Made & Re-Testing.

2.7 Correctly and Accurately Combining Normal Maps in 3D Engines (Dutton, 2013)

While this paper is not aimed at calculating depths, it gave a good overview of normal maps, which was useful to gain a better understanding of them and therefore made it easier to write an algorithm to generate them.

2.8 Program Review: Bitmap2Material

2.8.1 Overview

Bitmap2Material is a program aimed at 3D artists to help them create materials from existing textures in much the same way that the project program aimed to. Below is an image of its user-interface.

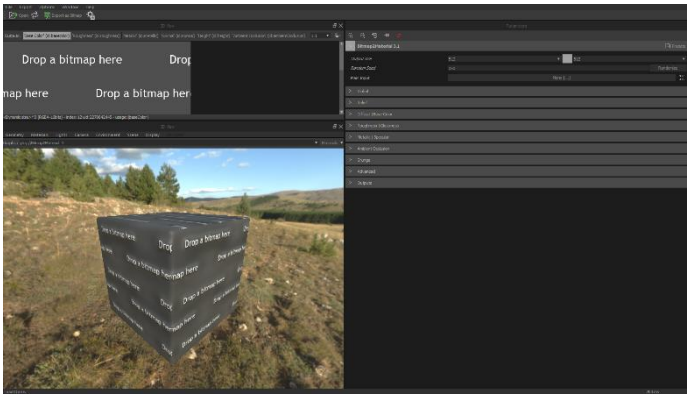

Figure 3: Bitmap2Material

The top left panel shows the original texture. The bottom left panel shows a real-time rendering of the material on a cube with lighting applied to it; the cube can also be rotated in real-time. On the right are all the different properties that can be adjusted.

Once the desired configuration has been set, the user can then press a button and have all the different textures that make up the material generated.

2.8.2 Advantages

The main appeal of this program is that you can see in real time the changes you make on a lit 3D object. The reason for this is to get a visualisation of what the result will look like when it is in the game engine.

There are many options to tweak the properties of the material – such as how metallic it should be, or how rough the surface is – allowing for a lot of control in the process.

2.8.3 Disadvantages

There is so much choice in what can be changed that it can be overwhelming for beginner users, particularly those who haven't had much experience with CG lighting.

The 3D rendering can be excluding to some users if they do not have powerful enough GPUs to render it in real-time.

3. Technology Background

A games engine is a piece of software that provides many features to assist developers in creating a game. It is useful because a lot of games have various features in common and programmers do not want to have to start from scratch or reinvent the wheel when making a game. Some of the most common features of games engines are:

- 3D world / environment editors
- A physics module that decides how objects in the world should interact with each other (move, direction and speed) when they collide
- A sound module that can be used to play sounds at different volumes and timings through each speaker – to give the impression of a source or origin
- A lighting module that determines how objects are lit, and the shadows they cast on each other
- Various other scripting tools that allow the programmer to trigger events when certain conditions are met; this can be as simple as the player character walking forward when a key is pressed, or complex as a non-player character talking when the player is close enough to them

Because there are so many elements in any given game, a lot of computing power is usually required to run them. To make a game run at a reasonable frame-rate[1], and be playable on the widest number of devices as possible (so that the customer base is expanded), certain optimisations or "tricks" are performed (Cohen, Olano, & Manocha, 1998).

A lot of the optimisations are focussed on graphical calculations. Whilst these are not computationally expensive, they are highly numerous since every object in the world needs to be lit or have rules as to how they should be lit (Krishnamurthy & Levoy, 1996).

In the real world, photons (light particles) come from a source, hit physical objects and interact with them. These interactions involve them being absorbed, bouncing off, and having

[1] The number of frames drawn to the screen each second. If this drops too low, the game looks like a flip-book or slideshow and becomes unenjoyable to play. Think of a frame as a picture, a picture of what the game looks like at an instant, like a photo. When these pictures are shown in quick succession, it becomes like a film.

their trajectories bent until they reach one's eyes. This is happening continually. For present day computers, this is too difficult to process in real time. However, some animation studios, like DreamWorks and Pixar have very powerful computers and a lot of time to render scenes using this ray-tracing technique. For games though, some shortcuts must be taken.

Very recently, real-time ray-tracing was demonstrated in the Frostbite games engine (Hillaire, 2018). Due to the cutting-edge nature of this technology traditional approaches are still preferred for their compatibility.

Consider a brick wall. Its colour is determined by our perception of the light it reflects. The wavelength (or frequency) of the light it reflects is determined by what it's made of[2] – which chemical elements; those elements are determined by the quantity of protons, neutrons and electrons that make them up. The protons and neutrons themselves are determined by the kinds of quarks that make them up. It's a lot of information that the computer doesn't need to know about.

To simplify, the wall can be represented as a flat surface. But how is a representation of a wall made? Some geometry is required. What is the simplest (2D) geometric shape? A triangle. A single triangle isn't a great shape for a wall, two triangles can be used instead. Put next to each other they make a rectangle.

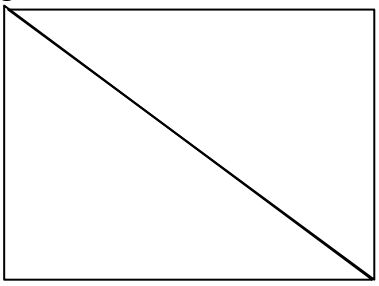

Figure 4: 2 Triangles arranged into a rectangle

[2] Note that metallic elements are more reflective because they have more freely available electrons. Therefore, porcelain looks white and not like a mirror.

It's not a very convincing wall right now. Perhaps if it was given a colour, it would look more like a wall.

Figure 5: A wall with a single colour

This is very easy for the computer to handle (Mikkelsen, 2008), as each pixel is the same colour. This would make a satisfactory wall in a stylised game but it's not very realistic. What is required is a photo of a real wall.

This is where the artists come in. A real wall in the real world might not tessellate[3] very well, so it would be better if one were drawn or painted that would suit the objectives better. These images that the artists create are often referred to as "textures".

Figure 6: A brick wall texture

A texture is a collection of blocks of colours in the form of an array. Each square of colour is called a "Texel", a portmanteau of texture and element. This looks much better, but what happens if a light is shone on it?

Figure 7: A brick wall with lighting

[3] To save memory usage in games, large complex textures are avoided. This means that the same texture is used multiple times, often next to itself. If the left side of the texture fits well with the right side when they are next to each other, as well as the top and bottom, then it is said to tessellate well.

At a glance, this looks okay, but look a little closer, and one can see that the wall is flat. The effect is more apparent on this ball. It is more like a brick wallpaper than actual bricks.

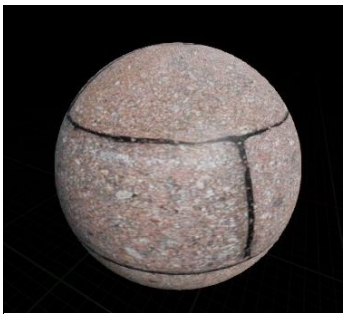

Figure 8: A brick ball with lighting

The geometry of the object could be made more complex, but as mentioned previously, this is inefficient, and is part of the reason textures are being used in the first place (Cignoni, Montani, Rocchini, & Scorpigno, 1998). An algorithm could even be written for the GPU[4] to make it look rougher (more textured), but this feature is only available on modern hardware so is not available to everyone. What is needed is another texture that describes the surface, not its colour. Fortunately, such a thing exists, and it is called a normal map.

What is a map? A map is a 2D representation of an area. In computing, it is much the same.

What is a normal? A normal is a unit vector that is perpendicular to the plane that it lies on. A plane is a 2D surface, in the case of the wall example, two triangles. If it is perpendicular to the plane, that means that it is pointing out from the plane at right angles to both the x and y axes of the plane. A unit vector is a vector of length one, i.e. a directional vector. A vector is an n-dimensional mathematical construct that is just a collection of numbers, in this case it represents the distances in the x, y, and z planes from the origin of the vector (which is a point on the plane's surface).

[4] A GPU or graphics processing unit is a specialist piece of hardware, that is very efficient at doing many simple calculations at the same time. For example, lighting calculations that may need to be performed on a per-pixel basis.

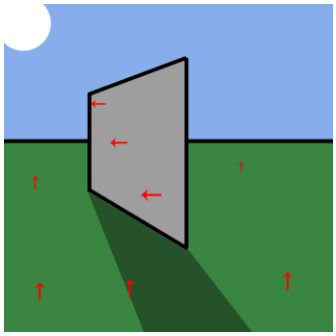

Figure 9: A diagram to show surface normals on planes.

The small red arrows in the diagram represent surface normals on the two flat planes (the ground and the wall). If the surfaces were bumpy, however, then the arrows would be pointing in different directions.

When normal maps were invented, textures had been around for a while, and so had their formats (bitmap, jpeg etc.). Because images contain only 3 channels (Red, Green, and Blue, disregarding opacity) they could be adapted easily to hold values for x, y, and z lengths.

Say the x plane (left and right) value is stored in the red channel, y (up and down) is stored in the green and z (in and out) is stored in the blue. Now, because one can't have a negative colour but can have a negative vector, there needs to be a slight re-mapping of values (except for Z because a surface would never have a normal that pointed into it). So, a value of 0 would point fully left (a negative value), and a value of 1 would point fully right (a positive value) in the red or x channel. Colour components used to have only 256 values, but now floating points are used so that HDR (high dynamic range) technology can be taken advantage of, meaning more values are now available for the normal vectors.

Here is an example of a normal map sourced from the internet.

Figure 10: A normal map for a fabric or leather surface

But what do normals do? Normals represent the direction the surface is facing across all points on it. When light falls on a surface, the surface is brighter in areas that are facing the light, and darker in areas that face away from it (Blinn, Models of Light Reflection for Computer Synthesized Pictures, 1977). The calculations performed to find the brightness and therefore colour of these points, can be performed on the object's geometry. It is difficult to do this in detail, hence the need for normal maps (Heidrich & Seidel, 1999). The reason this is difficult is because it requires a lot more computational power to render geometry using a mesh or model than using a normal map with the GPU (a piece of hardware that is highly specialised in processing images in parallel (Krušna & Denisov, 2013)).

Here is an example. The light is coming from the left, so the normals pointing left are better lit.

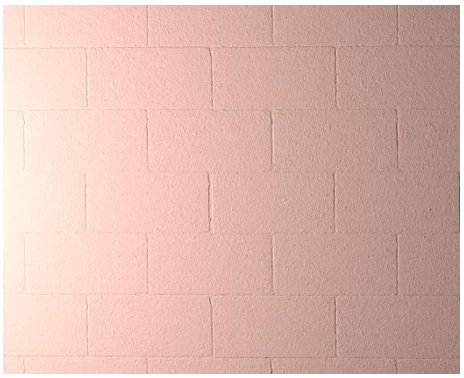

Figure 11: Lit normals on a surface. There is no albedo texture applied to better show the effects of the map.

That is what a normal map is, and how it used. But how are they created? For a short time, artists had to create them themselves, but this process was time consuming, so tools were invented to automate it. This was the goal of the project, to create a tool that could be used by game developers to create normal maps for existing textures, using those existing textures as inputs. Specifically, this program was designed to be simple to use, as well as fast and responsive.

4. Method

4.1 Overview

This methodology has been split into two parts, the 4.2 Development section describes how the program was created along with the algorithms used to generate the normal maps. The 4.5 Survey section refers to how the survey and evaluation process were designed and carried out.

4.2 Development

This program initially started out as a simple OpenGL application, which read in a configuration file that held data on where to find images and shader programs to use as inputs. It made use of GLEW (the OpenGL Extension Wrangler), GLFW (the OpenGL Frame-Work), SOIL (the Simple OpenGL Image Library) and some open-source code from learnopengl.com to interpret the shader files (Vries, 2014). The program loaded the images into memory and compiled the shader it had been given, rendering an output based on its instructions. Crucially, the shader had access to the images that had been loaded.

After some modifications, the program could then be used to save what was being currently rendered as a bitmap on the file system. It was also modified to use a Microsoft Windows style window instead of a command prompt to appear more user-friendly.

This advantage of this design meant that shader files could be interchanged using the same base program, which meant that little had to be altered to implement new algorithms. Some usability improvements were also added.

4.2.1 Algorithm A

The first algorithm that was implemented was created from pseudocode sourced from a computer graphics forum (Bois, 2010). Without a specific structure, it could have been implemented on the CPU or the GPU; the latter was chosen as the program was already structured in a way that made this easier.

The pseudocode for this was as follows:

```
for each texel:
let hd = the difference between it and its horizontal neighbour
let vd = the difference between it and its vertical neighbour
normal.x = sine(arctan(-hd / texelscale))
normal.y = sine(arctan(-vd / texelscale))
normal.z = sqrt(1 – (normal.x)² – (normal.y)²)
output: normal
```

Figure 12: Pseudocode for Algorithm A

The method did have to be modified a little so that it would compile properly, but it is much the same. It is as follows:

```
37  void main()
38  {
39      float texelscale = 1.0;
40
41      vec3 normal   = vec3(0, 0, 0);
42
43      vec4 original = texture2D(ourTexture1, TexCoord);
44      vec4 vertical = texture2D(ourTexture1, vec2(TexCoord.x - INV_WIDTH, TexCoord.y));
45      vec4 hozontal = texture2D(ourTexture1, vec2(TexCoord.x, TexCoord.y - INV_HEIGHT));
46
47      vec4 vDif = original - vertical;
48      vec4 hDif = original - hozontal;
49
50      normal.x = sin(atan(sum(-hDif) / texelscale));
51      normal.y = sin(atan(sum(-vDif) / texelscale));
52      normal.z = sqrt(1.0f - pow(normal.x, 2.0) - pow(normal.y, 2.0));
53
54      color = vec4(normal, 1.0);
55  }
```

Figure 13: Algorithm A. Note that the sum function is not a GLSL keyword, it adds the first 3 components of a 4-component vector and returns that as a floating point number.

Below are two images showing the input and the output of the program, with the original image on the left, and the normal map it produced on the right.

Figure 14: Albedo Bricks

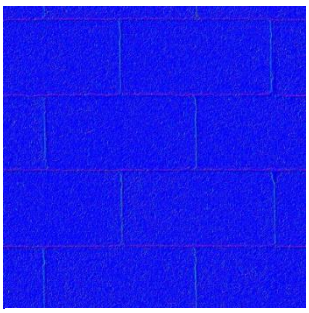

Figure 15: Initial Normal Map

Unfortunately, this was an undesirable outcome, but after analysing the pseudocode, a semantical error was found. For a normal pointing towards the camera, it would have 0% red and green, and 100% blue. This was not the case, it should have had 50% red and green. The code was then altered to add this 50% to those components.

To improve the algorithm, some variables were introduced that could be modified in real time with key presses. These were:

- Sampling Radius – How far away the horizontal and vertical textures should be sampled from
- Texel Scale – The intensity of the normal
- Is-Inverted – Whether or not the normal vectors are inverted

With these changes, a much more suitable result for the bricks texture was obtained:

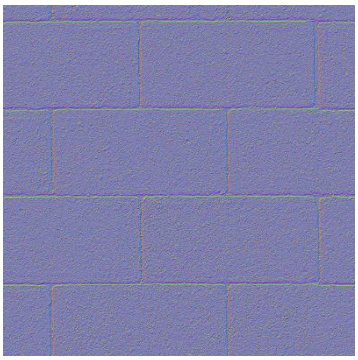

Figure 16: Final Normal Map for Algorithm A

It was deemed to be more suitable because its colours looked closer to those of existing examples of normal maps, and since the colours were closer, its normal values must have been closer and therefore more accurate. For example, blue contains no red or green components, which would suggest that the normal would point to the bottom left corner; whereas purple contains some red and green, suggesting the normal points more towards the camera.

4.2.2 Algorithm B

This algorithm was based on the methods described by Omar Ahmad (Ahmad, 2018) in his article outlining generating normal maps from height maps. He did not provide any

pseudocode for his methods, just a diagram at the end showing how to get the partial derivatives using a visual programming language.

As this method was designed with heightmaps in mind, it was decided it would be best to find the average value from the RGB channels. This is akin to a grey-scale function. The reason for this is that heightmaps are grey-scale images with lighter areas being higher, and darker areas being lower.

The algorithm relies on finding the tangential vectors of normals in the x and y planes, which are found by using the forward, backward, and central differences. So once these are found, the cross product can be performed on them to find a vector perpendicular to the plane they represent. Next the normals need to be normalised (turned into directional vectors), then converted from tangent space to normal space (just the red and green values as the Z component is always positive). The algorithm for this is as follows:

```
67      vec4 Dx = h2 - h1;
68      vec4 Dy = v2 - v1;
69
70      vec3 V1 = vec3(1, 0, avg(Dx));
71      vec3 V2 = vec3(0, 1, avg(Dy));
72
73      vec3 c = cross(V1, V2);
74
75      c = normalize(c);
76
77      normal.r = (c.x + 1) / 2.0;
78      normal.g = (c.y + 1) / 2.0;
79      normal.b = c.z;
80
81      vec3 ts = vec3(texelScale, texelScale, texelScale);
82      normal = pow(normal, ts);
83
84      if(showOriginal == 1)
85      {
86          color = texture2D(ourTexture1, TexCoord);
87      }
88      else
89      {
90          color = vec4(normal, 1.0);
91      }
```

Figure 17: Algorithm B. Dx is partial derivative, obtained from the change in colour in the horizontal plane. Dy is the same for the vertical plane. V1 and V2 are the tangential vectors. Note that avg is not a function of GLSL, it finds the average of a vector.

The output of this algorithm was much more favourable than that of Algorithm A. When testing in Unreal, it produced more visually appealing results.

The output for the same bricks texture as shown before, is below:

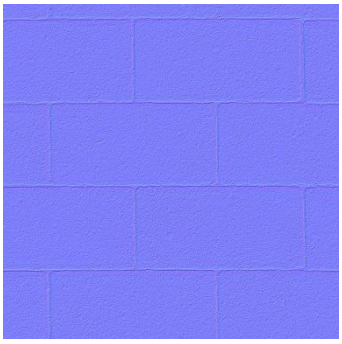

Figure 18: Normal Map produced by Algorithm B

Although the texture may appear "inverted" at a glance, in testing it was found that this is how the texture should appear for the bricks to be raised.

To explain the code for Algorithm B further, an annotated diagram of it is available on the next page.

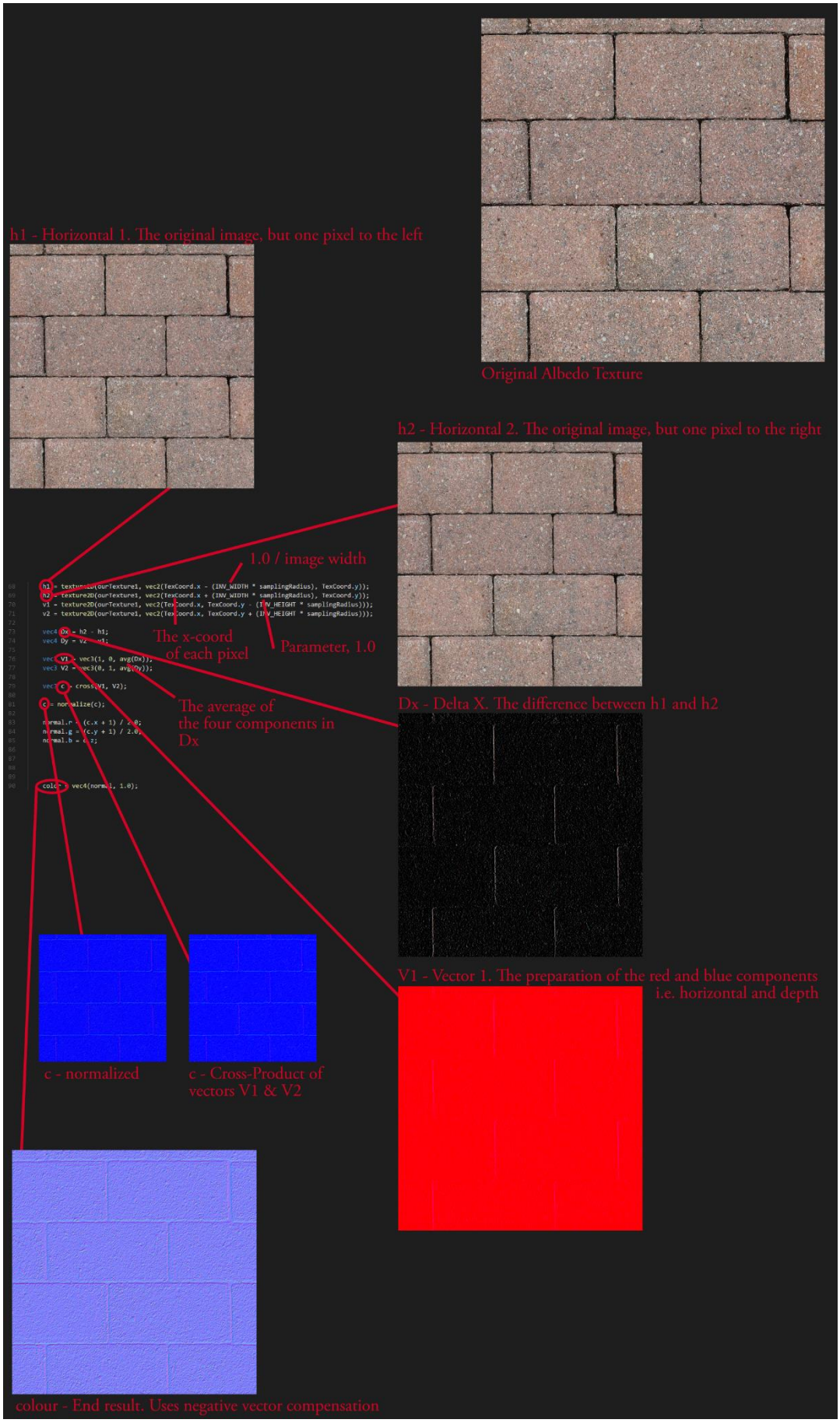

Figure 19: Diagram to show how the code for Algorithm B produces normal maps

4.2.3 U.I. and Features

Various features were added to the program when the need for them arose. They include but are not limited to:

- A way to open other images on the file system (i.e. an open file dialog) while the program is running, whereas previously the configuration needed to be edited and the program restarted
- A way to increase the "intensity" of the normals, this makes the effect appear exaggerated
- A way to increase the sampling radius of the algorithm, this allows for a bigger difference to be seen when the colour-change is more gradual
- A way to save the images. Images are saved as bitmaps in an images folder with "_normal" as their suffix, and the original image name as their prefix. Perhaps in future other file types will be possible
- A way to resize the images to 512x512 pixels specifically. This is useful for consistency and MIP-mapping. Again, in future, the size may be configurable
- A way of showing the original image, this is useful if the image is corrupted and cannot be displayed properly
- A way of inverting the map. This is useful if the lighter portions of an image are in a depressed section of the surface it was taken from. (For example, black tiles with white grouting)
- An info window, this is useful for giving the user information about the image, the values of its intensity for example, whether it is inverted etc. as well as information about the program as it is loading - which is useful for debugging. This was originally a command prompt but that was deemed not user-friendly enough

The last important feature that utilised the CPU was a method for finding the modal (most common) colour in the image. The reason for this was that, originally, in unreal, the original texture (that the normal map was created from) was being applied to the surface in conjunction with the normal map, and it was unclear what part of that image was being

affected by the normal map – so it was decided that a flat colour should be used instead. At first, the average colour was being found, but since this took the average from all 3 colour channels, the colour often appeared washed-out (as each of the R, G, and B components' values were close together).

One way of finding the modal colour is to go through each pixel's value and store it in a tuple (data type used to store two objects as a pair) with its count, but with large images, this would use a very large amount of memory. Instead, a 3-dimensional array was created, this was 8x8x8 (about 2KB of memory). Each coordinate value represented a colour value (x = r etc.) and ranged from 0 to 7. Next, each pixel was mapped to a coordinate value by dividing each of its components by 256 to get a value between 0 and 1, then multiplied by 8 and cast to an integer to truncate it. Then the value at that coordinate was incremented. The array was parsed, its maximum value was found, and the coordinates of this value were returned. The coordinate values were then converted back into RGB colour values and normalised RGB colour values. Note that because of the truncation earlier, a value of 1/16 was added to each component. This made it so that the minimum value was 1/16 and the maximum was 15/16 for each component.

4.3 Obtaining Test Images

To test the program, some sample images needed to be gathered. Initially, some images from the internet were sourced but these couldn't be used due to copyright laws. It was then decided that some new images should be taken – photographs.

The best kinds of images were those taken of "flat" surfaces as those were what they were going to be applied to. It was also better when they were evenly lit and did not have a light shone directly on them, it was for this reason that they were taken on an overcast day. The camera was not positioned close to the surfaces as this creates a fish-eye effect, producing textures that tesselate poorly. Surfaces with a similar colour throughout tended to work better as the algorithm is better suited to brightness changes than hue changes.

Some surfaces with raised sections that were dark and impressed sections that were light, e.g. bathroom tiles, had to use the program's inversion feature.

Before processing, each image had to be cropped to a square so that the resulting normal map would keep the same aspect ratio, and work properly in the engine. The feature to resize

the image to 512x512 pixels was particularly useful for when the image was imported into the engine as this allowed for MIP-mapping[5].

4.4 Unreal Engine

While Unreal may be more complex in certain aspects than Unity for creating games, it is easier to use to create materials. Materials in Unreal Engine are akin to shader programs, but they use a visual programming language. A visual programming language works like a script, it is interpreted, not compiled. It is called a visual language because it shows a visual representation of what the "code" will do, often it looks like a flow-chart.

Below is an example of such a material. There is a preview window on the left. The inputs in the middle are "piped" into the outputs on the right.

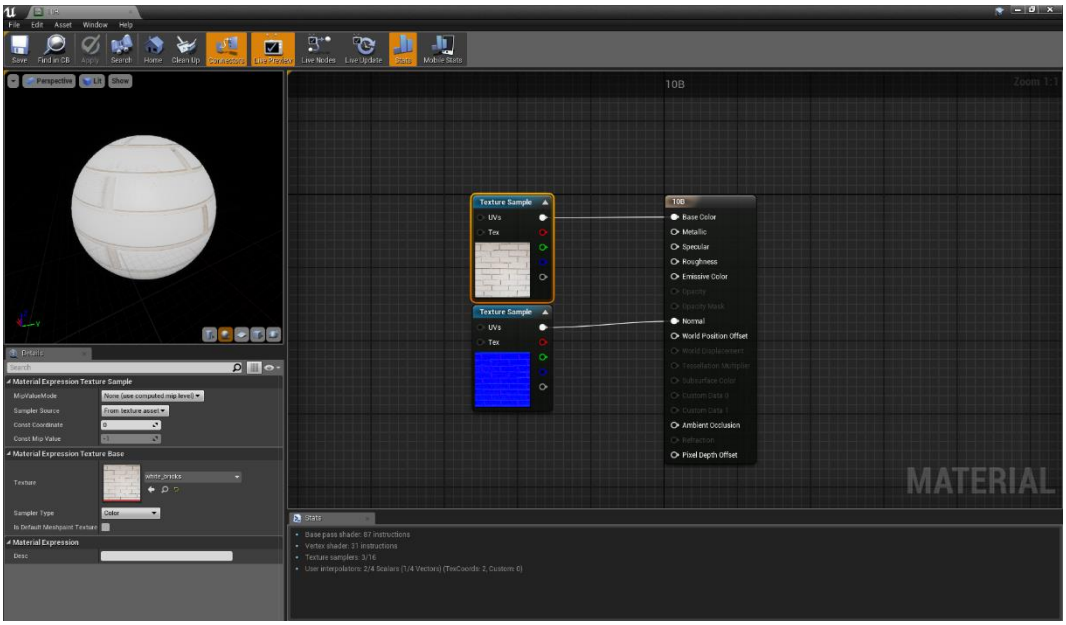

Figure 20: A material in Unreal Engine

These materials can then be applied to volumes – 3D shapes (forms) that allow for tessellation.

In the demonstration environment that was created this method was followed. Once the textures were applied, point lights are added to the scene to show the effect of the normal maps.

[5] MIP maps are smaller versions of the texture generated at or before compile-time. They are used to increase efficiency by being used in levels of detail (a texture being rendered further away does not have to be as big) as well as reducing Moiré patterns (apparent artefacts in tessellating textures). The dimensions of the image must be a power of 2 to generate MIP maps.

In the scene created, the materials used a colour as a base input, not a texture, so that the effects of the normals were easier to see, this would not be the case in a game. This base colour was the modal colour calculated previously.

4.5 Survey

To evaluate the program and measure its effectiveness, a survey was carried out. It was divided into two sections, the first asking participants to identify which images they preferred visually, out of ten pairs of images. In each pair there was a material that used a normal map generated by the project program, and a material created by a control program. They both used the same base images to generate the normals. The order of the pairings was random.

For those participants who had access to Unreal Engine, they were shown the materials in a 3D environment where they could move the camera around in real-time. (Approximately 50% of participants.) The rest were shown static images of the materials. Ideally, they would have been able to see the materials in 3D so that they could have had a better idea of how the light played off the surfaces. Figure 14 shows an example of a pair of images that participants had to choose between.

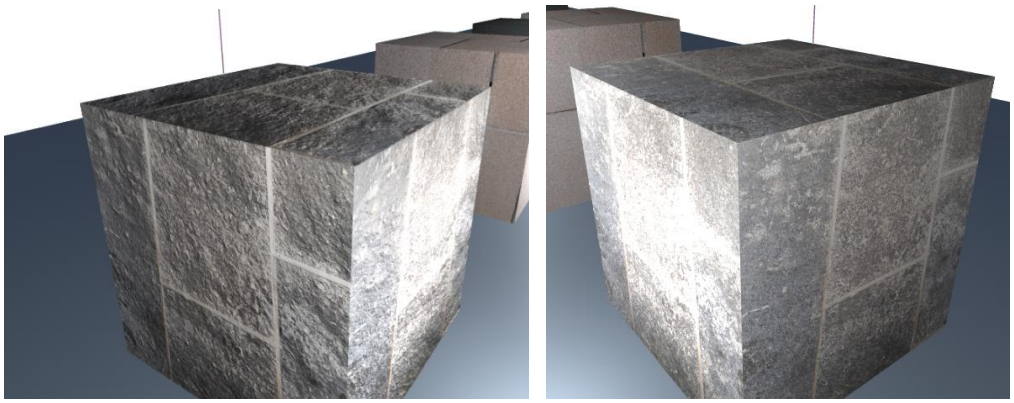

Figure 21: An example of a pair of materials participants had to choose between.

The participants were also asked which image would be most suited for use in a video-game, if their answer was different from the ones they preferred. In the latter part of the first section, they were asked to say why they preferred one over the other, and to give an example; as well as for their additional thoughts.

In the second section of the survey, participants were asked to use the program to complete a set task. The task made use of all the features except for the file selector. At the end, participants were asked if they managed to complete it, and how long they spent doing it.

They were then asked what difficulties they encountered and what they liked about using it. There was also a question with a right or wrong answer to test that the participants did use the program (and did not pretend to). The reason for this was because not all tests were supervised.

Here is an image of what the program looks like when the task has been successfully completed:

Figure 22: The program once the test had been successfully completed

Participants were also asked how they would improve the program if they had the chance, and if they had any other comments.

To make it a fair test, participants were neither assisted in the second half, nor were their opinions influenced in the first half – they were left alone when they were making their decisions and the order of the materials was randomised, however, they were shown how to control the camera in the 3D environment.

5. Results & Analysis

5.1 Evaluation of Images

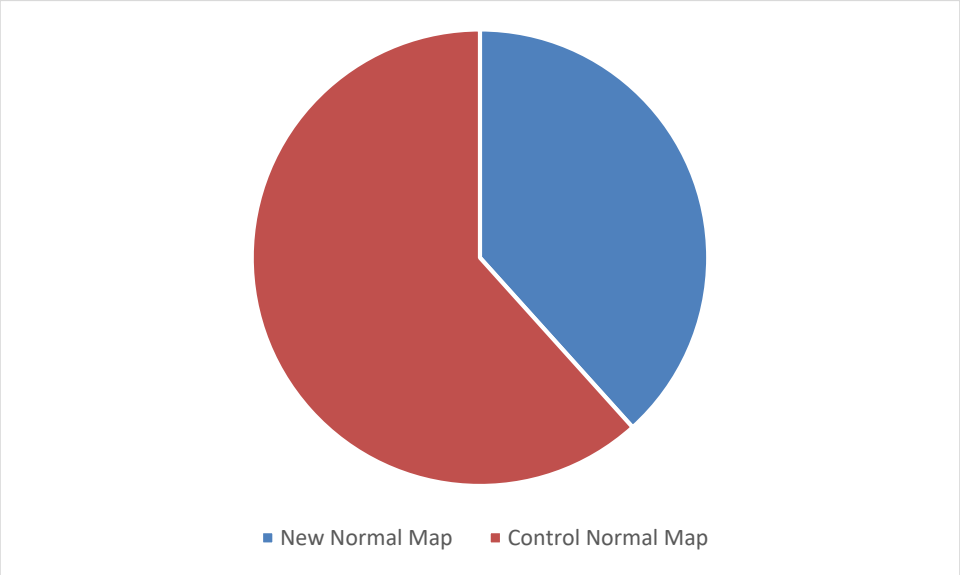

Figure 23: Pie chart to show the overall percentage of the number of times the new normal map was picked over the control

Overall, participants only preferred the new normal maps 38.3% of the time. This, however, is not the full picture, there was a large deviation in participants' preferences, as shown by the following box-plot.

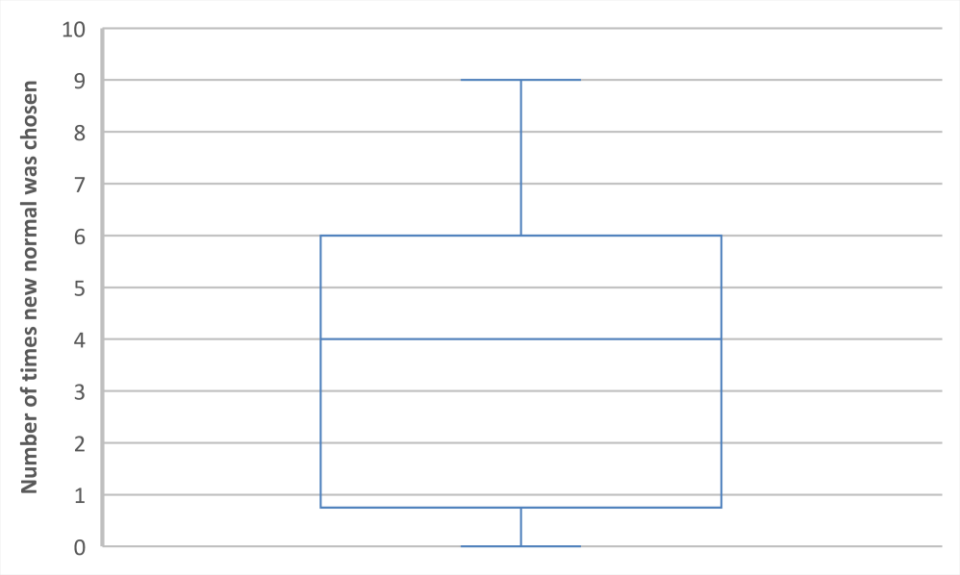

Figure 24: Box plot to show how the distribution of participants' votes

This shows that the maximum number of times the new normal maps were chosen over the control maps was 9, and the minimum was 0. I.e. There was one participant who always preferred the control maps over the new ones.

Since there were only 10 images, these results could be interpreted as having a large margin of error. That, however, is quite reductive. A better conclusion to draw from this is that the test was highly subjective and what one participant liked in the images, another found unappealing.

To get a broader picture, it would have been better if there were more participants.

For certain images, the participants mostly had the same opinion, and it was from these that a new method was devised. The image pairing with the most agreement was number 9. All but one of the participants preferred the control map over the new one.

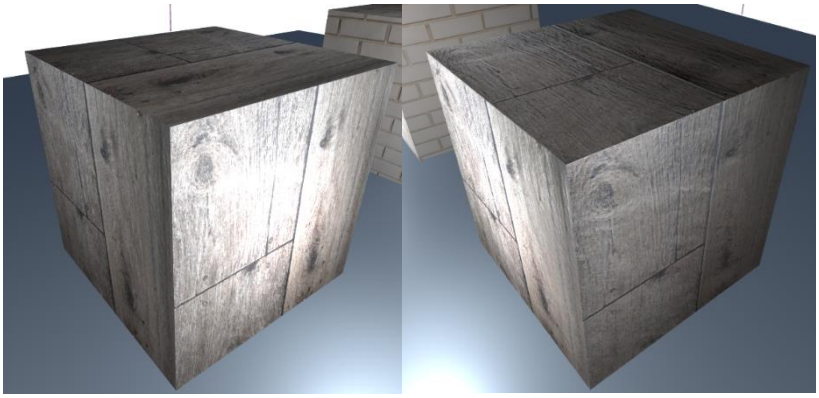

Figure 25: Image pairing 9. On the left is the new map (A) and on the right is the control (B).

This is an image of vinyl flooring made to look like wood. In a game, this would be easier to pass off as wood than in real-life, and this is how most participants saw it – they thought it was intended to be wood and so voted for the one that looked most like wood. In this case, it was the control.

The person who chose to use 9B as their example to justify their choices they said that "The surface appears more textured and interesting." For the rest of the participants, who gave other examples, their reasoning was much the same, preferring depth and exaggeration over what may have been more realistic. This idea is explored further in 5.3 Changes Made & Re-Testing.

5.2 Evaluation of Program

When getting the users to try out the program, they were asked to complete a small task using it. They were then asked some questions about it. The results of this are as follows:

- 100% of participants completed the task

- 100% of participants found the correct modal colour using the UI
- 66% of participants said they spent a minute or less on the task

Participants were also asked to describe in words, what they liked and didn't like about using the program.

The most common theme in their comments about the positive aspects of the program was that it was fast and responsive, they liked the fact that they could see the changes they were making to the image in real-time.

They also liked that they could see the values of the parameters they were changing – and the ability to change them in the first place, although they did not like the floating-point inaccuracies of these values.

Overall, however, the participants had more suggestions on how to improve the program than things they liked about it. Most of the issues stemmed from the UI (user interface). These issues included:

- Having to use keyboard controls / a lack of sliders and input boxes
- Not being able to interact with (highlight / copy / etc.) the text on the information window
- A bug with the information window where resizing it caused all text on it to disappear
- The fact that there were two separate windows
- Not being able to hold down keys to change values faster
- Not being able to change the destination and name of saved images
- Too much information on the information window
- Not being able to use other size pre-sets than 512x512 pixels

Due to the way the program is constructed, only a few of these issues could be addressed easily. Merging the windows into one, and having sliders, for example, were impossibilities, meaning that the program would have to have been remade from the ground up. Manually setting the destination for saved images was also difficult because the Windows-specific code required to do this was obtuse and not always reliable.

In the future, however, it would be nice to address all these issues in a fresh attempt at creating the program, but that would potentially mean marrying C# code with C++ and that may prove difficult. This is because it is difficult to create a U.I. in C++ but not in C#, however, in C# one cannot easily use OpenGL to render the preview using the GPU. There are libraries that exist to assist in this area. Further thoughts on this topic are discussed in 6. Conclusions & Future Work.

5.3 Changes Made & Re-Testing

To create more depth and dynamism in the normals, the original images were altered to be "dynamic" – i.e. they made to have a wide range of colour values. With the white bricks as an example, the minimum colour value was quite high (some shade of white) while the maximum value was also quite high and, therefore, similar with a small numerical difference.

To get the most difference in the image, the minimum value was lowered to black, and the maximum was increased to white. The values in between were linearly interpolated. This was done in *Adobe Photoshop* with a technique called "levelling".

Here is an image showing the levelling tool being used on one of the images.

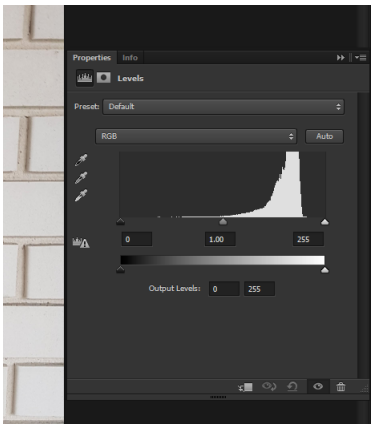

Figure 26: The unmodified levels of the white bricks texture

The x-axis of the graph represented the colours from 0 to 255, the y-axis represented the quantity of those colours within the image. The graph showed most colours in the image were above the half-way mark.

There were 3 sliders at the bottom of the graph, they were moved left and right to edit the colours of the image. The white slider's position represented the "new white" value, the

black's position was the "new black" value, and the grey's position was the "new midpoint" value.

To get a desirable value, the black slider was moved right until it was at a point where the quantity of colours in the image at that point exceeded 2 pixels high on the graph; and the white slider was moved left until the same criterion was met. Below is an image to illustrate.

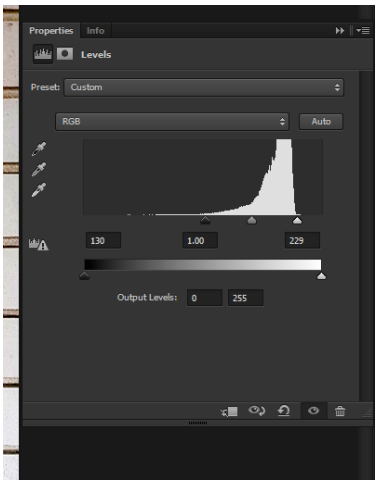

Figure 27: The altered levels of the white bricks texture

Here are the before and after levelling images for white bricks. First is the original, then the resulting normal maps, then their use in Unreal. In each case the image on the left is unlevelled.

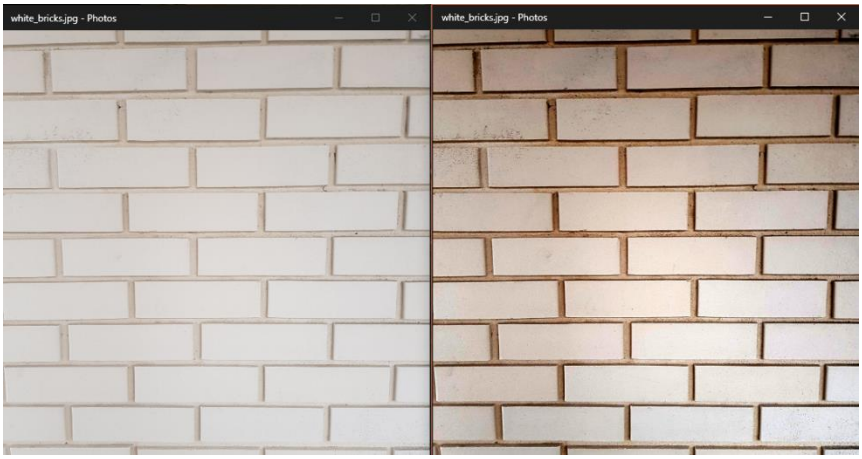

Figure 28: The before and after levelling images of the white bricks texture

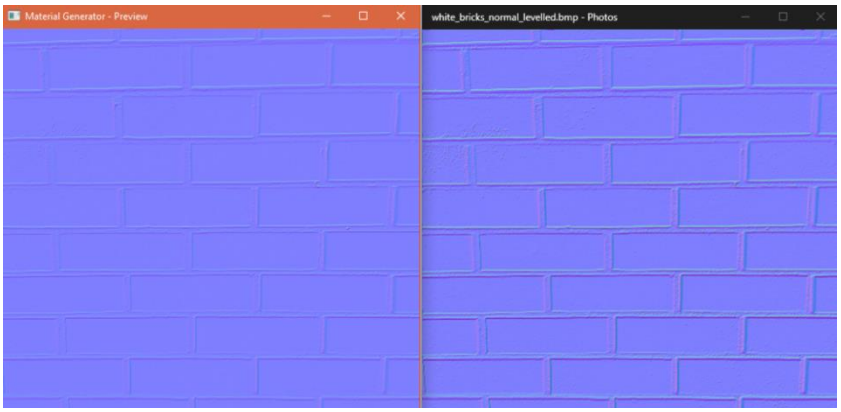

Figure 29: The normal maps produced by the unlevelled and levelled white bricks textures

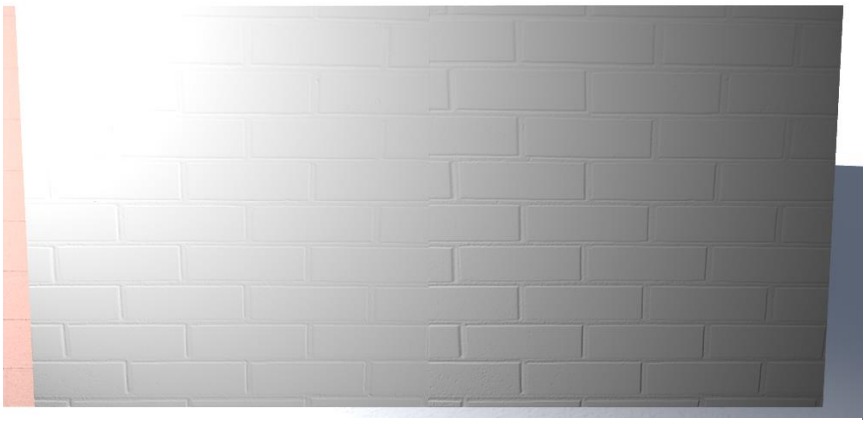

Figure 30: The normal maps applied to a surface in Unreal Engine

Below is an image of what each version of the white bricks material looks like, in engine, with the original texture applied.

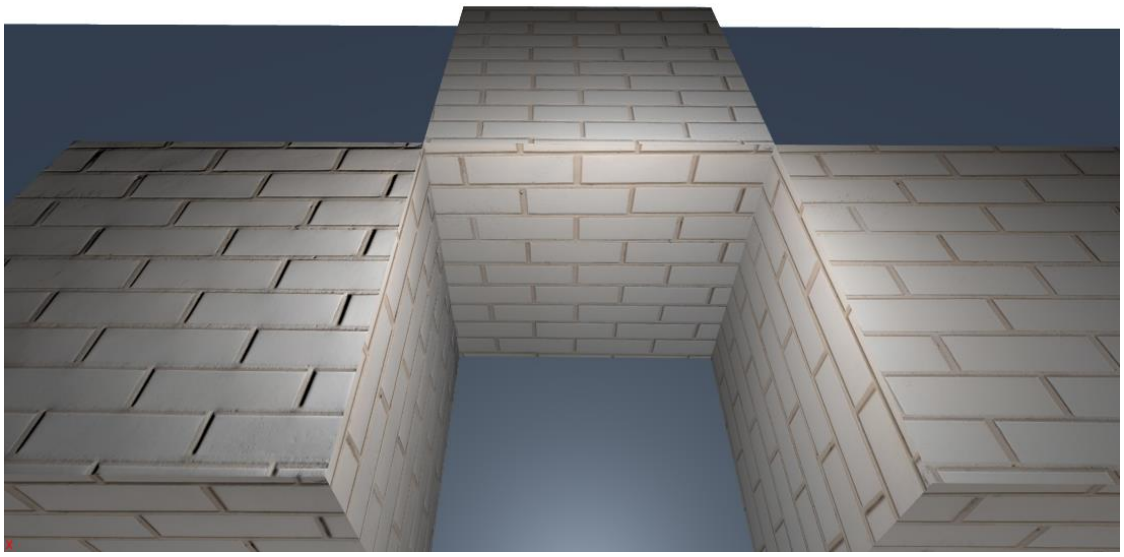

Figure 31: Normal maps applied to objects in Unreal Engine with albedo textures

On the left is the control material, on the right is the material that used the first iteration of the generated normal map, in the middle is the material that uses the normal map generated from the levelled image. Although it is hard to tell the difference between the middle and right images, it is more apparent in the engine itself where one can move the camera.

The first part of the survey was then performed again using the normal maps generated from levelled textures in place of the previously generated ones. It was also re-randomised to produce a fair test.

It was noted that in the first survey there were some aspects that weren't making it as fair a test as it could have been. Because of the positions of the cubes and the lights, they were not being illuminated in the same way, the camera was also not consistent in its positioning and angle for the screenshots (meaning that some sides were more in view than others), this was changed in the second survey.

The presentation cubes were aligned in a single row, each with their own light positioned in the same place relative to its cube. Extra lights were placed at each end of the row, to compensate for the indirect lighting the cubes in the centre were receiving that the ones on the end were not. The screenshots for each cube were then taken at the same position and angle relative to each cube, this was done by using a camera object, positioning it, and then piloting it.

This second test found that participants of this survey had a slight preference for the materials whose normal maps were generated by the application, over the control (51.67%).

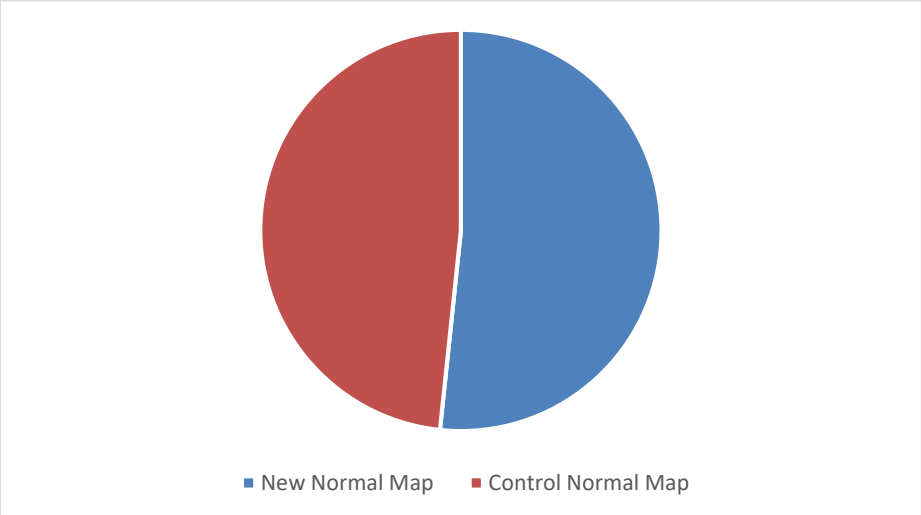

Figure 32: Pie Chart to show the overall percentage of the number of times the new normal map was picked over the control in the second test

A box plot was created to show the distribution of the votes:

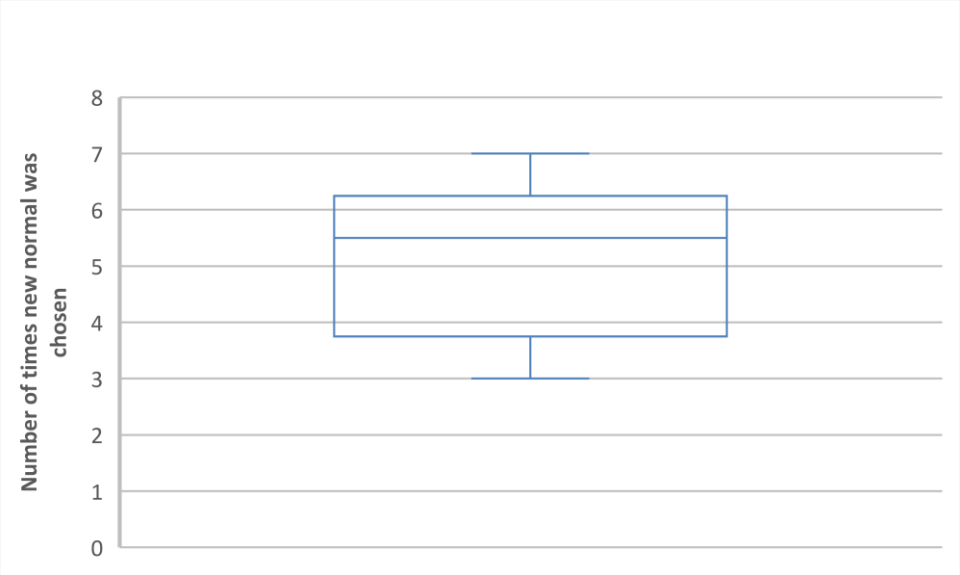

Figure 33: A box plot to show how participants voted in the second test

Overall, the average number of times the generated maps were picked over the control was higher than before (see box plot). Table 1 shows that in 7 out of the 10 textures, the generated normal map was either equally or more preferred than the control.

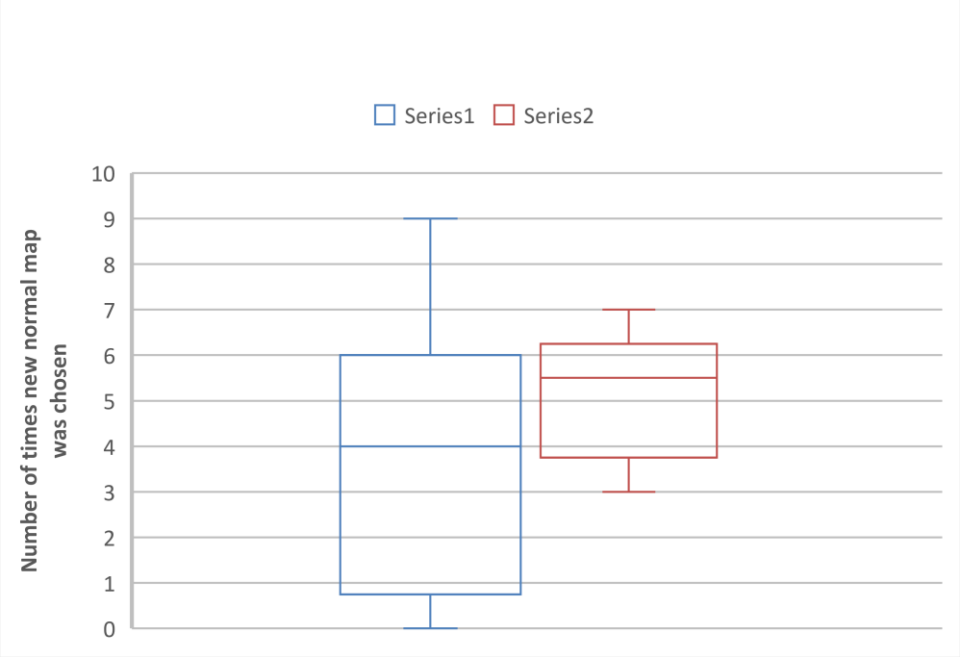

Figure 34: Box plot to show how often testers preferred the generated normal maps over the control

Therefore, it can be said that this current method – that uses the levelling technique – is at least as good as, if not better than, the control / competitor program.

Table 1: Table to show the percentage of respondents that preferred the generated normal map

Texture Number	Percentage preference for generated normal map
1	66%
2	33%
3	83%
4	50%
5	66%
6	50%
7	50%
8	33%
9	33%
10	50%

6. Conclusions & Future Work

6.1 Overview

This section includes what was learnt, developed and considers potential improvements of the project.

6.2 Future Work

In this project, there are three major aspects that are candidates for improvement, they are:

- The User Interface and User Experience
- The implementation of the method to find the modal colour
- Automating the levelling of images

Although it would limit the deployment of the program to just Windows machines, it would be much better for the program's U.I. if it was written in C#. There are, however, some problems with this. A large portion of the program would have to be re-written. It is desirable for the program to still use OpenGL so that it remains fast and responsive. To do this, a new external library would have to be used. One alternative is to use DirectX or Direct3D, which, since they were developed by Microsoft, have native support in C#.

Here is a concept for the design of the proposed U.I:

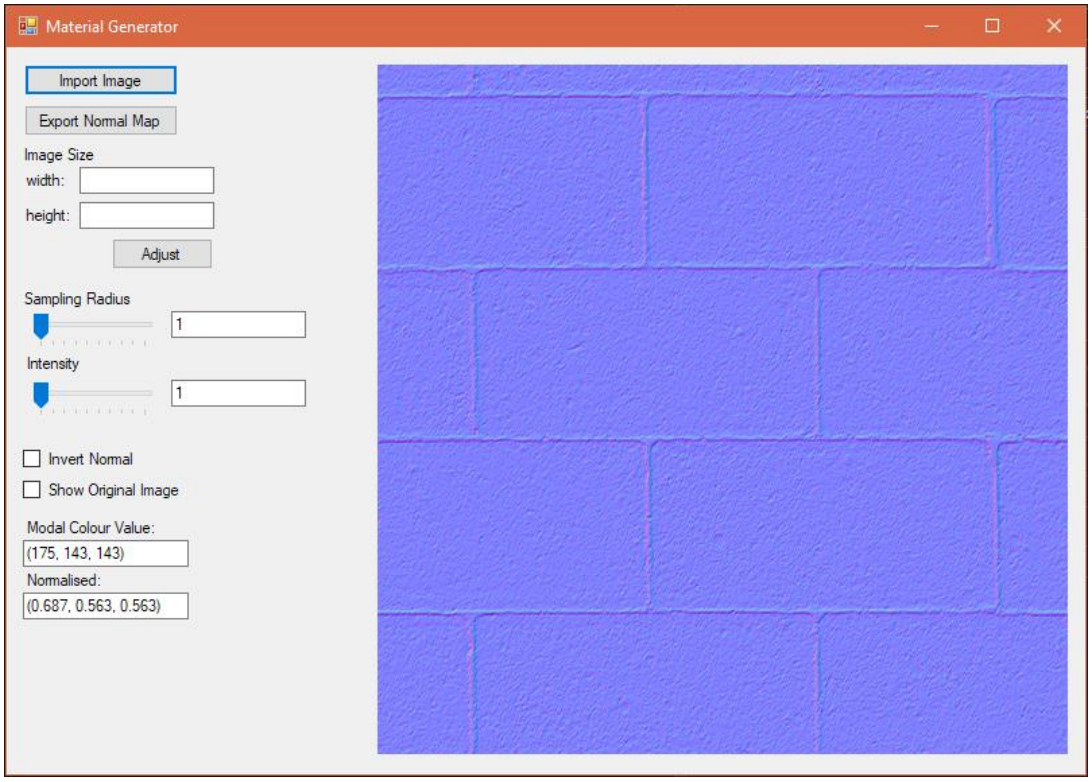

Figure 35: A design for the new user interface

The most recent implementation of finding the modal colour makes use of a 3D array. To save memory it is 8x8x8 in size (512 possible values). Each axis (x, y, z) represents a colour channel (r, g, b). The index of each position in the array represents a colour value. The value at each index represents how frequently the colour (within that band) appeared in the image. Because there were only 8 possible values for red, green, and blue, this method was considered inaccurate.

Here is a diagram to represent that data structure:

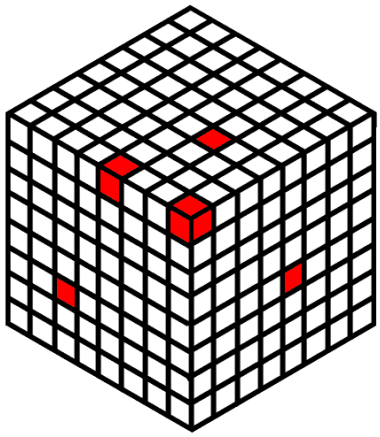

Figure 36: Diagram to show what a 3D array might look like

Some cells have been coloured to show the shape and scale of it.

Instead of using this method, one can take advantage of image compression techniques. One way in which images are compressed, is to use one colour multiple times if there is one like it elsewhere in the image. This means images can be represented using fewer colours, and so fewer would need to be loaded into RAM.

A 2D array should be used instead. The first value would be the 3D colour value, the second would be its frequency in the image. Like so:

Table 2: Table to show a 2D array holding the frequency of colours used in an image

rgb(66, 134, 244)	25
rgb(101, 130, 92)	13
...	...
rgb(n_1, n_2, n_3)	f_n

The only reason this technique was not used before, was because it was thought that it would use too much memory and would take too long to traverse.

The last technique that should be implemented in this program is levelling. Before, it had to be done manually. Not everyone has the specific image editing software required to do this, and it would be asking too much of the user to get them to do it themselves.

The proposed method to achieve this, is to first find the darkest and lightest pixel values in the image on the CPU. Then pass these values as uniform floating-point numbers to the GPU. Once the GPU has these values, it can then execute the following shader code to level the image:

```
37  vec3 gammaCorrection(vec3 colour, vec3 gamma)
38  {
39      return pow(colour, vec3(1.0 / gamma));
40  }
41
42  vec3 level(vec3 colour, vec3 minInput, vec3 maxInput)
43  {
44      return min(max(colour - minInput, vec3(0.0)) / maxInput - minInput, vec3(1.0));
45  }
46
47  vec3 levelWithGamma(vec3 colour, vec3 minInput, vec3 gamma, vec3 maxInput)
48  {
49      return gammaCorrection(level(colour, minInput, maxInput), gamma);
50  }
```

Figure 37: Code required to do automatic levelling

(Dura, 2009)

There are four parameters here: The colour, the gamma-correction colour, the minimum / darkest value, and the maximum / lightest value.

Gamma correction is optional. It is used as a bias to make the image lighter / darker, while maintaining contrast. It is recommended that the user specifies first, whether to use it or not, then its value.

The other parameters are found from the image. While the "colour" parameter varies across the image, the minimum and maximum are constant.

6.3 Conclusions

This project set out to see whether it was possible to generate normal maps from albedo textures that could be used in games, it found that this could be achieved by implementing an existing algorithm that was used as a starting point and developed so that it produced an enhanced output preferred by testers for its aesthetics.

This algorithm was translated into GLSL and adapted to use textures as inputs. It found that these normal maps were at least as good as ones generated using commercial software, according to those interviewed.

It was found that the process was improved when the images were levelled before processing.

The user interface was identified as a weak-point by testers; they did not like having to use only the keyboard to make changes to the output. If it were to be designed again, it would be using a different architecture that would allow a more easily-accessible interface to be created.

By using Unreal Engine to create a 3D environment to use for testing, I made better use of my time by spending it developing the algorithm rather than the test harness – which was the focus of the project.

Overall, this project was successful in generating normal maps programmatically, however some improvements could be made to the program's interface to increase its usability.

7. References

Ahmad, O. (2018, January 5). *Normal Map Generation*. Retrieved from Squircle Art: https://squircleart.github.io/shading/normal-map-generation.html

Blinn, J. F. (1977). Models of Light Reflection for Computer Synthesized Pictures. *ACM SIGGRAPH Computer Graphics*, 192-198.

Blinn, J. F. (1978). Simulation of Wrinkled Surfaces. *Computer graphics and interactive techniques* (pp. 286-292). New York: ACM SIGGRAPH.

Bois, A. (2010, May 5). *Normal Map Generation*. Retrieved from OpenGL Web site: https://www.opengl.org/discussion_boards/showthread.php/170895-Normal-Map-generation

Cignoni, P., Montani, C., Rocchini, C., & Scorpigno, R. (1998). A general method for preserving attribute values on simplified meshes. *Visualization* (pp. 59-66). Durham, NC: IEEE.

Cohen, J., Olano, M., & Manocha, D. (1998). Appearance-Preserving Simplification. *Computer graphics and interactive techniques* (pp. 115-122). Orlando: ACM SIGGRAPH.

Dura, R. (2009, January 28). *Levels control shader*. Retrieved from Mouaif: https://mouaif.wordpress.com/2009/01/28/levels-control-shader/

Dutton, C. (2013). Correctly and accurately combining normal maps in 3D engines.

Eisemann, M., Decker, B. D., Beckaert, P., Aguiar, E. d., Ahmed, N., Theobalt, C., & Sellent, A. (2008). Floating Textures.

Heidrich, W., & Seidel, H.-P. (1999). Realistic, Hardware-accelerated Shading and Lighting. *Computer Graphics and interactive techniques* (pp. 171-178). Los Angeles: ACM Press/Addison-Wesley Publishing Co.

Hillaire, S. (2018, March 21). Real-time Raytracing for Interactive Global Illumination Workflows in Frostbite. *Real-time Raytracing for Interactive Global Illumination Workflows in Frostbite*. San Francisco, California, USA: Electronic Arts.

Krishnamurthy, V., & Levoy, M. (1996). Fitting Smooth Surfaces to Dense Polygon Meshes. *Computer graphics and interactive techniques* (pp. 313-324). New Orleans: ACM SIGGRAPH.

Krušna, D., & Denisov, V. (2013). General purpose parallel programing using new generation graphic processors: CPU vs GPU comparative analysis and opportunities research. *Computational Science and Techniques*, 36-44.

Kuznietsov, Y., Stückler, J., & Leibe, B. (2016). Semi-Supervised Deep Learning for Monocular Depth Map Prediction.

Liao, J., Shen, S., & Eisemann, E. (2017). Depth Map Design and Depth-based Effects With a Single Image.

Lind, J. (2013, December 29). *Simple Normal Mapper*. Retrieved from Git Hub: https://github.com/juzzlin/SimpleNormalMapper/blob/master/src/Renderer.cpp

Mikkelsen, M. (2008). *Simulation of Wrinkled Surfaces Revisited.* Santa Monica: Naughty Dog.

Ondrúška, P., Kohli, P., & Izadi, S. (2015). Mobile Fusion: Real-time Volumetric Surface Reconstruction and Dense Tracking on Mobile Phones.

Saxena, A., Chung, S. H., & Ng, A. Y. (2006). 3-D Depth Reconstruction from a Single Still Image.

Vries, J. D. (2014, June). *Welcome to OpenGL*. Retrieved from Learn OpenGL: https://learnopengl.com/

Appendix A. Questionnaire

Survey

Section A

You will be shown a short demonstration of various materials in use in a games engine. For the following table, enter A or B appropriately in each empty box.

Material #	Most Realistic	Most suited for use in a game	Overall Preference
1.			
2.			
3.			
4.			
5.			
6.			
7.			
8.			
9.			
10.			

With a particular material in mind, why did you prefer it over its counterpart?
Material #: Material Letter: Reasoning:

Do you have any other comments?

Section B
Using the bricks texture, produce a normal map that is 512x512 pixels in size, has an intensity of 1.75, a sampling radius of 2 and is inverted, and save it to the file system.

Did you succeed?
☐ Yes
☐ No

How long did you spend on the task?
☐ A minute or under
☐ More than a minute

What was the value of the modal colour when you did this?

What difficulties did you encounter using the program?

What did you like about using this program?

What didn’t you like about using this program?

What would you do to improve this program?

Do you have any other comments?
