Can Non-Realistic Shaders be used as an effective method to enhance a scene without sacrificing computer performance?

Jack Evans

Bsc (Hons) Computer Games Technology, 2014
University of Abertay Dundee

Permission To Copy

Author: Jack Evans

Title: Can Non-Realistic Shaders be used as an effective method to enhance a scene without sacrificing computer performance?

Degree: Bsc (Hons) Computer Games Technology

Year: 2014

(i) I certify that the above mentioned project is my original work
(ii) I agree that this dissertation may be reproduced, stored or transmitted, in any form and by any means without the written consent of the undersigned.

Signature ........................................................................................................

Date .................................
Abstract

This project explores the possibility of increasing efficiency in the art pipeline by producing the animations and shading for environment assets with shaders rather than in the art pipeline. Whereas most modern shaders focus on quickly rendering effects in addition to the work created by an artist, this dissertation will focus on reducing the artists involvement from the animation process as much as possible.

It is proposed that shading techniques could be used to replace the traditional rigging, skinning and animation process in certain circumstances, leaving art teams more time to focus on the creation of models, textures and advanced character animations, rather than focussing on smaller or less important environment assets.

The project will focus on the topic of non-photorealistic shaders of different types, and attempt to combine the techniques to create a custom shader that will handle the animation of an asset as well as its lighting and shading, whilst maintaining the level of efficiency of a standard diffuse shader. Implementation will occur in the form of a 3D scene, demonstrating the custom shader compared to the standard diffuse shader, and will be used to provide data regarding the artistic effect of the shader via user testing, and efficiency via the fps of the application.

Results obtained by the project will show that by the altering and reapplication of modern shading techniques, efficient and stylised shaders can be used to replace some aspects of the art pipeline.
Preface

I would like to give thanks to several people without whom, none of this would have been possible.

Firstly my project supervisor Mr Matthew Bett for his advice and guidance throughout the course of this project, which helped to develop a simple idea into something I can feel proud of.

Next I’d like to give my thanks to Joshua Henderson, not only for supplying me with models for testing and teaching me how to create my own assets, but also being a friend through recent years.

Additionally, I’d like to thank my parents and not so little brother for their support and many food packages over the past 4 years, without which I wouldn’t have survived my time at university.

And finally, Lee Jervis, for sitting on a coach for 14 hours just to spend a weekend playing computer games with me when I was stressed and just needed to relax.

Thanks to all of you, and many more I’m sure I’ve forgotten to thank.

— Jack Evans, 2014
# Table of Contents

1 – Introduction......................................................................................................................1  
2- Literature Review.............................................................................................................3  
  2.1 - Toon Shaders..............................................................................................................3  
    2.1.1 - Toon Outlining ....................................................................................................3  
    2.1.2 - Toon Lighting .....................................................................................................6  
  2.2 - Distortion Shaders.......................................................................................................8  
    2.2.1 - Traditional Animation Pipeline .........................................................................8  
    2.2.2 - Blend Shapes and Morph Shapes ........................................................................9  
    2.2.3 - Vertex Animation ...............................................................................................9  
  2.3 - Reflective and Refractive Shaders ..........................................................................11  
    2.3.1 - Reflective Shaders ............................................................................................11  
    2.3.2 - Refractive Shaders ............................................................................................13  
3 – Methodology .................................................................................................................14  
  3.1 - Distortion Shading ....................................................................................................15  
  3.2 – Toon Lighting .........................................................................................................18  
  3.3 – Toon Outlining .......................................................................................................21  
  3.4 – Optimisation ...........................................................................................................22  
  3.5 – Reflective Shading ...................................................................................................23  
4 – Testing and Results .......................................................................................................24  
  4.1 - Testing structure .....................................................................................................24  
  4.2 - User Testing ............................................................................................................24  
  4.3 - Technical Testing ...................................................................................................27  
5 - Discussion .......................................................................................................................30  
Final Words and Future Work ..........................................................................................35  
Appendices .........................................................................................................................36  
  Appendix A - Questionnaire given to testers to collect data ........................................37  
  Appendix B - Graphs to display questionnaire results ....................................................38  
  Appendix C - Debug Data ...............................................................................................41  
List of References ...............................................................................................................42  
Bibliography .........................................................................................................................45
# List of Figures

- Figure 1 – Demonstration of toon shading algorithm ........................................... 4
- Figure 2 – Demonstration of toon shading algorithm ........................................... 4
- Figure 3 – A photo before and after Sobel Filter ................................................... 5
- Figure 4 – A Ming head with crease outlining ....................................................... 6
- Figure 5 – Model lit with experimental lighting shaders ....................................... 8
- Figure 6 – Capsules lit with different shaders........................................................ 15
- Figure 7 – Results of the first iteration of the vertex animation shader ............. 17
- Figure 8 – Several frames showing vertex animation issues ................................ 17
- Figure 9 – The effect of the completed vertex animation shader ....................... 18
- Figure 10 – Toon lighting effect applied to a low polygon tree ....................... 20
- Figure 11 – A high resolution model with toon lighting and outlining .............. 22
- Figure 12 – The scene with diffuse shaders ......................................................... 25
- Figure 13 – The scene with custom shaders ........................................................ 26
- Figure 14 – Title screen ........................................................................................ 28
- Figure 15 – Application run at minimum and maximum graphics settings .......... 33
- Figure 16 – Animation issues caused by running at lo graphics settings .......... 33
- Figure 17 – Age range of testers .......................................................................... 38
- Figure 18 – How testers described the first scene ............................................... 38
- Figure 19 – How testers described the second scene .......................................... 39
- Figure 20 – Number of changed descriptive words .............................................. 39
- Figure 21 – Average percentage of changed descriptive words ....................... 39
- Figure 22 – What key differences were noticed between scenes ..................... 40
- Figure 23 – When asked if there was a change in performance ....................... 40
1 – Introduction

“The art challenges the technology, and the technology inspires the art.”
– John Lasseter, Pixar Animation Studios

This quote and this man seem the most appropriate place to begin when discussing the topic of shaders, as Pixar took some of the initial steps on the road to what shaders are now. Until massive steps were undertaken by this company, computer graphics projects were limited to the fixed pipeline architecture, which generally resulted in synthetic looking graphics.

In other fields such, as the film industry, high end rendering software was allowing more and more realistic results. A prime example of this software was Pixar’s very own RenderMan, a high level shading language developed to enable artists and graphics programmers to get the most control of the rendering process, and allowing results such as Toy Story and other high quality CG (St-Laurent 2004).

The work completed by Pixar combined with the introduction of DirectX 8.0 gave birth to Vertex and Pixel shaders. Home computers could now render images as never before, allowing artists and programmers to immerse players of computer games to a new degree.

Shaders are now commonplace in computer games, and game engines handle most graphics plotting without the programmer having to be concerned about base level coding, however, where shaders are used, they are only used for achieving rendering effects.

Where in the past, the people of Pixar worked to improve pipeline efficiency with their shaders, striving to break new ground and establish new techniques and uses with their knowledge, the pool of innovative ideas has nowadays, for most programmers, stagnated.
This paper will focus on finding the merits and issues of standing techniques that can be used to create effective and easy to apply shaders, in the hope of revitalizing the field of using shaders as a tool. If successful, this project should find ways to increase efficiency in the art pipeline by reducing the dependency of animation on the rigging and animation teams whilst answering the question:

*Can Non-Realistic Shaders be used as an effective method to enhance a scene without sacrificing computer performance?*

This dissertation will delve into the study of current shading techniques to find:

- What techniques currently exist in the field of non-realistic shaders and for what purpose are they used?
- How can these techniques be used to improve the current art pipeline?
- How effective can shaders be at changing a scenes atmosphere?
- How optimised can the shaders providing the solution become?
2- Literature Review

There are many different kinds of shaders with varying levels of complexity within the area of computer game development, each giving a different property to the game object that it is applied to and in turn altering the object's scene. The result of the shader is the 2D coordinates and colour values for each pixel that will be plotted to the screen from the viewpoint of the camera with the given effect applied, be it a simple lambert or specular shader, or a more advanced toon or depth of field shader.

This project focuses on three of the more advanced areas of shaders; Toon Shaders, Distortion Shaders and Reflective and Refractive Shaders

2.1 - Toon Shaders

Sometimes referred to as "Cel Shading", Toon Shading is a method of rendering used to imitate the 2D art style of cartoons or the appearance of comic books. The shader makes the 3D graphics appear flat by using several shades of flat colour to render the object rather than having gradual gradients between colours. The earliest use of toon shading in the games industry was for the Sega Dreamcast game “Jet Set Radio” (Smilebit 2000) in 2000, though this type of shader is still very popular in the computer game market. Recent games featuring toon shading include “Crackdown” (Realtime Worlds 2007), “Sly Cooper: Thieves in Time” (Sanzaru Games 2013) and to imitate the comic style in “The Walking Dead” (Telltale Games 2012). The process of toon shading is often broken down into 2 key components; lighting and outlining.

2.1.1 - Toon Outlining

The first toon rendering technique is used to create a black outline around the edge of the models in a scene to draw focus to them. This increases the user’s awareness of the edges to help them distinguish between primary objects in the foreground and images in the background that may share similar colour palettes. See Figure 1 below as an example of a scene where the edge has been
clearly defined with a toon shading algorithm and Figure 2 where the outline has been applied to the final render to enhance detail.

Figures 1 and 2 respectively from “Effective Toon-Style Rendering Control Using Scalar Fields” (Harvill 2007, Harvill 2007)

The results can be achieved by rendering the scene as a 2D image with toon lighting to give a flat coloured texture. The scene’s depth and world-space surface normals are then rendered to a texture to which an edge detection algorithm can be applied (St-Laurent 2004).

A common edge detection algorithm used for toon shading is the Sobel Filter. Implemented with the pixel/fragment shader, the Sobel Filter detects edges horizontally and vertically and a clear thickened outline is left behind after several passes.

The Sobel Filter performs 2D spacial gradient measurements on an image and emphasizes areas where there is a large gradient, these are the edges. It is an improvement over the previously popular “Robert Cross” operator, which until the Sobel Filter was the most common method of edge detection (Sobel 2014). The algorithm uses two 3x3 kernels which are convolved with the image. The second kernel is the same as the first but rotated by 90° counter clockwise. During each pass the filter is applied to the input image pixel by pixel to find gradients. Afterwards the results are combined to find the magnitude of the gradient at each point and the orientation of the gradient (Fisher et al. 2003). Below are the two kernels used for the algorithm;
If the first kernel is defined as $G^x$ and the second as $G^y$ then the gradient of the magnitude is given by:

$$|G| = \sqrt{G^x^2 + G^y^2}$$  \hspace{1cm} (1)

Because the operation is done in the fragment shader it will be done many times per pass so a more efficient approximation of the magnitude is used, this makes computing faster and less expensive to processing power;

$$|G| = |G^x| + |G^y|$$  \hspace{1cm} (2)

The angle of the edge can be found by the equation;

$$\theta = \arctan\left(\frac{G^y}{G^x}\right)$$  \hspace{1cm} (3)

The results of the Sobel method applied to a black and white image can be seen below in Figure 3 (Green 2002).

Shaders that take the density of colour at points on normal and depth maps to calculate where lines need to be rendered were further advanced upon in the paper “Fast silhouette and crease edge synthesis with geometry shaders” (Hajagos, Szécsi, and Csébfalvi 2012).

The method proposed in this paper splits the models into a triangular mesh as would be found in a normal game engine such as Unity and then defining which edges of triangles are edges of the object and which are “halfedges” that can be
used as creases. A half edge is defined as an edge that can be seen alongside another edge in the 2D perspective but isn’t actually adjacent as seen on the Ming head in Figure 4 (taken from the aforementioned paper).

![Ming head with crease outlining applied](image)

Figure 4 – A Ming head with crease outlining applied

The advantages of this method are that only a single pass is needed and there is no need for image processing filters. Issues arose in the authors’ testing of the theory though and it was found that clearly defined crease definitions are needed so models that deform, such as character models, would not be able to have the shader applied to them, making this method only suitable for rendered static background models.

### 2.1.2 - Toon Lighting

Another aspect of toon shading is the advanced lighting effects created using the fragment shader.

One technique to achieve the lighting required would be to use a series of masks (phil_potatoe 2013). A way of doing this is to create a specular lighting shader using 3 sets of values, colour and threshold for the lit, unlit and specular components of the models. Whereas colours are usually added together to create the final colour of the game object, in the toon lighting shader the colours use the values and threshold to create cut offs. These cut offs define where each colour starts and ends, using the inverted effected area as a mask.
This method provides an efficient but very simple toon lighting effect when added to a scene with directional lighting.

Another typical way that the effect can be achieved is through simple 3D lighting and a Lambertian shading model, with a 1D toon texture mapped to describe how the shaded area varies from light to dark areas. The animator has the ability to control the cartoon style with a 1D toon texture assigned with the required colours. The lighting is controlled in the scene with 3D direct lights.

In “Stylized Lighting for Cartoon Shader” (Todo, Anjyo and Igarashi 2009), advanced light dispersion techniques were incorporated into a shader to overcome issues caused by the normal 3D lighting and 1D texture techniques. H. Todo and his team’s paper deals with issues they call a “discontinuous light appearance problem” and a “straight lighting problem” that appear when using the previously stated method.

The report produced by the team presents a new lighting model, partially inspired by the X-Toon Shader (Barla, Thollot, and Markosian 2006), that has the same effect as using high quality bump maps without causing the artist extra work.

Using a light coordinate system defined by the origin $p_i$ and the coordinate axes $(du, dv$ and $dw$) a light vector field can be defined as:

$$L(p) = L_u(p)du + L_v(p)dv + L_w(p)dw \quad - (4)$$

Where $L(p)$ is a light vector at $p$ on a surface. From here the straight lighting on the flat surface is achieved with the function;

$$f_{st}(L(p)) = L_u(p)du + (1 - \alpha) L_v(p)dv + L_w(p)dw \quad - (5)$$

If $\alpha$ approaches 1 then the light vector will go along the $dv$ direction (where $\alpha$ is the directional scaling term). By using this equation in different ways and transforming and normalizing it the team was able to produce high quality toon lighting. The lighting in question, although fit for purpose, was unfortunately very taxing to the systems processing rate during the testing phase, causing
frame rates ranging from 6 – 20 fps, making this method unsuitable for this project. See Figure 5 for an example of the lighting at work.

![Figure 5 - Items on the left are lit with traditional lighting methods. Items on the right are lit with the new method developed by the team.](image)

It should be noted that the issues caused by the 1D toon texture technique could be solved by changing bump maps and normal maps on the models experiencing the problem, but this would create further work for the artist, something this dissertation focuses to overcome.

### 2.2 - Distortion Shaders

The next area of discussion is the process of distortion via animation. This was a topic that was only very lightly touched upon during the proposal phase, but will receive much more attention this time around.

#### 2.2.1 - Traditional Animation Pipeline

Typically the art pipeline is where all animation for a game happens. Artists model assets in programs such as Maya, ZBrush or 3D Studio Max and create
the UV’s for the model. Using the UV’s a texture is created by a texture artist and the model is then given to the technical art team. This team then creates a rig and skins the geometry to the skeletal frame, making it possible to animate with ease. After the rigging process the model will have a selection of control shapes (usually created with nurbs curves) that make it easier to control the animations of the model (Rodriguez 2013). The rigged model is now given to the animator who creates all animations needed and bakes them to a game object, normally a .fbx file (Edwards 2006). By following this pipeline complex animations can be ported easily into the engine and triggered by scripts.

2.2.2 - Blend Shapes and Morph Shapes

As part of the rigging process mentioned in section 2.2.1, assets can be given blend and morph shapes to enhance their animation capabilities. Although the end results of these two techniques appear very similar they are achieved in slightly different ways.

Morph shapes are one of the most popular techniques for animation. The method consists of creating several key poses sculpted into the model, such as different facial expressions. From here controls are used to interpolate between the shapes with a smooth motion to change the expression gradually (Pighin et al. 1998).

Blend shapes are very similar to this but rather than just moving between two sets of points as the morph shapes do they actually add to the existing model. Maya looks at the differences between the base shape and the blend target and adds the difference rather than finding a midpoint between the two shapes (Osipa 2010).

2.2.3 - Vertex Animation

Rendering effects are another way in which atmosphere can be changed within a game and inspiration for this can be taken from previous games or films. Effects can range from simple fog and haze such as in “The Elder Scrolls: Skyrim” (Bethesda 2011) to extreme screen distortions such as interactions with the
Scarecrow in “Batman: Arkham Asylum” (Rocksteady Studios 2009) and the initial encounter with the riders in “The Lord of the Rings” (The lord of the rings 2004). “Fog and other atmospheric effects blur out or obscure objects that are some distance away” (Feil and Scattergood 2005), this sort of effect can be easily created using shaders and/or other post processing techniques and can actually save processing power by reducing the amount of detail needing to be rendered in background models. For the purpose of this discussion only extreme distortions will be discussed further.

Based upon the idea of morph shapes and blend shapes, vertex animation is the process of animating models via changing the position of the vertices each frame with the use of code rather than by having them skinned to a rig. This process is similar to the main scripting of character controls for movement but instead of changing the position of the game object (Anon. 2014), the position of the points that make up the game object are being changed.

Very little exists in the way of journals and books regarding the topic but the concept is as follows. To start with a variable is created to store each component (x, y, z) of the position of the asset, this will later be used to store the new position values. Within the vertex shader an offset is calculated according to how the user wants to distort the asset, be it a general noise using random numbers or a mathematical function such as $\sin(x)$. At this point each offset is added to the initial value and stored in the new position variable, to be plotted by the remaining part of the vertex shader. See below for the pseudo code of this system.

\[
\text{OriginalPosition} = \text{AssetPosition};
\text{Offset}\.\text{xyz} = \text{RandomNumberGeneratorOutput};
\text{NewPosition}.\text{xyz} = \text{OriginalPosition}.\text{xyz} + \text{Offset}.\text{xyz};
\]

As with all code, there are certain limitations to the above method of animation. The key issue of this method is that although animation is being performed it will only add “noise” to the asset. Any complex animations such as
walk cycles would still have to be created in a suite such as Maya and exported as part of the model’s .fbx file. This being said, it is still a useful tool for creating large distortions at a small cost to computer performance.

2.3 - Reflective and Refractive Shaders

The final topic of discussion in this literature review is that of reflection and refractions. Whether it be looking at water or a mirror, or looking through glass and diamond, it always interrupts the immersion of a player if the surface doesn’t act as expected. This section will address these matters and discuss ways in which they can be achieved efficiently.

2.3.1 - Reflective Shaders

2.3.1.1 - Realistic Reflections

In a perfect mirror reflection, the light reaching a perfectly smooth surface is reflected in one direction, at an angle equal to the angle of incidence, with no scattering across the surface. Since the mirror surface is isotropic the reflected ray $R$ lies on the plane defined by the light ray vector $I$ and the normal $N$ (reference isbn:9781852339029). Therefore for a perfect equation;

$$\theta_r = \theta_i \quad (6)$$

Meaning;

$$\phi_r = \phi_i \pm \pi \quad (7)$$

2.3.1.2 - Cubemap Reflections

Cubemap reflections are a computationally simple and memory-friendly method of achieving professional looking, polished reflections without excessive calculations. Formed by six perspective images, cubemaps give the ability to render 3D models directly onto them using the graphics hardware (GPU) (Engel 2008). Given a unit vector, a value can be looked up from the cube map and used to plot to the model using the fragment shader.

Normally each cubemap face is uniformly partitioned into a 2D texture, but during the scaling up process used to map to a spherical surface the partitions
distort, destroying the uniformity. To fix this the cubemap faces are partitioned in their natural Cartesian space by correcting the texture coordinates \((x, y)\) using the function \(f\);

\[
f(x) = (1 - r)x + r \sin \left( \frac{\pi}{2} x \right)
\]

Where \(x \in [-1, 1]\) and \(r\) is an empirical constant of value 0.5512575. The corrected texture coordinates are stored as \((f(x), f(y))\).

To be more exact the correction function \(f\) is the linear interpolation of \(x\) and

\[
\sin \left( \frac{\pi}{2} x \right)
\]

Where \(r\) is the ratio controlling the interpolation. The ratio is chosen to minimise the discrepancy as much as possible. Discrepancy is given by;

\[
D(N) = \frac{1}{2 \sqrt{\pi N}} \left[ \sum_{i,j=1}^{N} \left( 1 - 2 \ln \left( 1 + \sqrt{\frac{1-n_i \cdot n_j}{2}} \right) \right) \right]^{\frac{1}{2}}
\]

Where;

\[
n_1, \ldots, n_n
\]

is an N-point sequence and \(n_i\) is a point on the sphere. The lower D is, the more uniformly distributed the sampling pattern will be.

When sampling the cubemap for a reflection the vertex normals can’t be used as they could sample the same points regardless of where the player is located. To overcome this, a reflection vector is calculated using the normal of the object and the incident vector that runs from the players view to the surface of the object, the same as when calculating the specularity of an object. To calculate the angle or reflection the following formula is used;

\[
R = I - 2N(N \cdot I)
\]

Where \(R\) is the reflected vector, \(I\) is the incident and \(N\) is the normal. This calculates an angle of reflection equal to the angle of incidence but pointing away from the reflective surface (Anon. 2013).
2.3.2 - Refractive Shaders

Refraction is the change in direction of a wave due to the medium through which it is travelling. It can also be calculated using cube map based shaders using a refractive index.

Snell’s law states that for a given pair of media and a wave with a single frequency, the ratio of the sines of the angle of incidence and angle of refraction is equivalent to the ratio of the phase velocities in the two media. This can be expressed as;

\[ \frac{n_1 \sin \theta_1}{n_2 \sin \theta_2} = \frac{\text{phase velocity in medium 1}}{\text{phase velocity in medium 2}} \quad (12) \]

Where the constants \( n \) are the indices of refraction for each of the corresponding media (Anon. 2001). By using the refractive indices and the angle of incidence the exit angle (angle of refraction) can be calculated.

A key example of where this is used is in glass simulation. As well as using a cube map to calculate the reflections for the glass, the refractions for looking through the glass are also calculated. The end result is calculated by blending the reflection and refraction using the Fresnel term (Pharr and Fernando 2005).
3 – Methodology

For the purpose of checking if shaders could be used as an effective method to change atmosphere within a game, a 3D scene was created that would allow for the testing of shaders and that could be later used for user testing. In order to maintain appropriate testing conditions the following requirements had to be met;

- The ability to change the texture of the assets from the GUI rather than directly in the code
- The ability to change the toon lighting of the assets from the GUI
- The ability to change the level of distortion of the assets from the GUI
- The ability to move around the 3D scene to allow the user to view all of the assets up close

For these reasons the Unity 3D engine (Unity Technologies 2014a) would be used to create the application as it would meet all of the above requirements and would also be a familiar programming environment to work with. Additionally, using Unity would speed up the process of creating a scene to apply the shaders, due to the engine already having the ability to load models and add lights to scenes. This would save time and allow more attention to be spent on the creation of assets and the scripting of shaders.

Because the aim of the project was to find out the capability of using shaders to enhance a scene’s atmosphere the first step was to set up a scene with sufficient lighting. A new unity scene was created with three lights; two directional lights and one point light. This would give a range of lighting styles and angles to ensure the shaders responded properly to the scenes when being used in the final product.

For the testing of the shaders a place holder model was added to the unity scene. A “capsule” model was chosen as the place holder for its curved top but straight sides. This would give the maximum amount of angles for light to reflect from in the specular lighting models, as opposed to a cube which would have limited reflective surfaces.
3.1 - Distortion Shading

One key aspect to the application is the distortion to the models in the scene. It was decided that the process of vertex animation would be used to achieve this effect, as it would allow for either a single distortion to be applied at initialisation, or a repeated implementation once per frame to allow for animation.

The shader was given input variables for the animation strength in each axis (x, y, z), an offset for each axis (x, y, z) so that each asset in the scene didn’t warp at the same time, a speed of animation to control how fast each model would distort and a frequency.

Using these values and the initial position of each vertex, a distorted model is created. To begin with a sine wave was used to create a general wave distortion;

\[
\text{Distortion} = \sin((\text{time} \cdot x \times \text{Animation Speed}) + (\text{Animation Offset} \times \text{Animation Frequency} \times \text{Animation Strength})) \quad - (13)
\]

As expected this returned a small wave like distortion that repeated continuously as shown in figure 6 with a comparison to how it looked before.

![Figure 6 - Capsule model with the standard diffuse shader from unity (left) and with sine wave vertex animation (right)](image)

Although there was a small distortion this was not the effect that was wanted in the final result, it would have to be enhanced. One issue was the repeating
nature of the sine wave. Even though this meant the model would always maintain its position in the scene, it felt too uniform to be considered distorted, it needed to appear more random. After much contemplation it was decided that this “randomness” would be accomplished with a procedural function call. Subsequent to deciding this, research was done on the topic of Unity functions that could be called. During this research a function called “noise()” was discovered that could be called from HLSL code, but upon further investigation, it was found that many GPUs don’t support the function. Furthermore, due to the limitations of the Cg library, the function might not work. A custom script would have to be used for the procedural element.

As the focus of the project was not procedural function, but rather the shading element, it was settled upon that the code for the procedural “noise” would be acquired from elsewhere. In the end a highly optimized version of the Perlin noise algorithm (Lex-DRL 2013) was used. The code was included along with the standard pragmas at the top of the script and was called using the function snoise();. This was used to replace the sine value to give the equation;

$$\text{Distortion} = \text{snoise}((\text{time}.x \times \text{Animation Speed}) + (\text{Animation Offset} \times \text{Animation Frequency} \times \text{Animation Strength})$$  \(\text{─ (14)}\)

To give a smoother distortion pattern time.x was later replaced with time.x/10 and the output value for the noise function was divided by 100. Further experimentation was done with the snoise(); function by replacing the input values to see the end results. This included multiplying the value of time/10 by the normal value for each vertex or alternatively by the value of the vertex positions. The functions can be seen below;

$$\text{Distortion} = \text{snoise}\left(\frac{\text{time}.x}{10} \times v.\text{normal}\right); \quad \text{─ (15)}$$

$$\text{Distortion} = \text{snoise}\left(\frac{\text{time}.x}{10} \times v.\text{vertex}.\text{xyz}\right); \quad \text{─ (16)}$$
Unfortunately, the results for these equations didn’t fulfil the requirements necessary regarding the third point of being able to control the level of distortion of assets, and so the initial distortion function (equation 14) was used. At this point in the project some of the assets had been completed, allowing the shader was tested on a model of a low polygon cartoon tree. The result can be seen below:

![Figure 7 - The result of the initial iteration of the vertex animation shader](image)

Whilst testing this, a major issue was spotted with the resulting effect of the shader. Due to the animation of all vertices in the model, the base of the tree would move around skewing the result from looking “non-realistic” to confusing and unrealistic. This can be seen in figure 8 below:

![Figure 8 - Several frames of the animation showing the negative effect caused by the shader.](image)
To overcome this issue a new variable was added to the shader to allow for a base limit to be added to the distortion. The variable is controlled with a slider, ranging from the entire model being affected, to none of the model being affected. If set to a low value, a threshold is implemented. An “if statement” is run for each vertex to check if the value lies above or below the threshold. v.vertex.y is compared to the value and the result of the statement determines whether or not the noise is added to the vertex position. This small statement enabled complete control of the animated area and completed the work on the vertex animation segment of the shader. The results can be seen in figure 9.

![Figure 9](image)

**Figure 9 - The effect of the completed vertex animation shader v1.3 on the low polygon cartoon tree asset**

The shader was saved and the lines of code that caused the animation were isolated to be used in the final shader.

### 3.2 – Toon Lighting

The next aspect to be addressed was the toon lighting shader. To decide upon a technique to use, the end purpose of the final “compound” shader was considered. The shader would have lots of different elements, and due to the complexity of the procedural function from the previous section, may become inefficient if the lighting model became too complicated. The other side of the issue would be that if the lighting was too simple then the effect wouldn’t be of
a high enough standard. For these reasons a specular lighting model, based on
the idea from the literature review, with thresholds for lit and unlit areas was
chosen. It would allow complete control of the lighting effect from the shader
GUI to meet the second requirement stated “The ability to change the toon
lighting of the assets from the GUI” and would also run efficiently enough to not
cause slow rendering times.

To achieve this effect, areas of the model would be culled using “masks”, using
values from the user defined variables for the lighting threshold and specular
threshold. It should be noted that whereas the code that was used to achieve
the vertex animation was executed within the vertex shader, the code used for
the lighting would be calculated in the fragment shader. The reason for this is
that although it requires more calls, per-pixel calculations create a much more
“polished” render than per-vertex calculations, due to smoother gradients.

To begin with, the dot product of the normal direction and light direction was
calculated to return a value of -1 to 1, as the value of the first vector
approached the second. Values above 0 are lit, with a gradient getting brighter
as the value approaches 1, and values below 0 are not lit. The result of the dot
product was saturated to negate negative values ensuring only values between
0 and 1 exist.

A cut off was then calculated. This was done by using the max( , ) function by
first finding the larger value; the result of the saturated dot product or the
threshold value as defined by the user. The value for the threshold was at this
point subtracted from the result of the max function giving either a value of 0, if
the threshold value was the largest, or a non-zero value if the saturated dot
product was larger. The result of this was saturated again to ensure the value
lay in the range on 0 to 1.

\[
Cut \ Off = \text{saturate}(\max(\text{threshold value} , \text{saturated dot product}) - \\
\text{threshold value}) \quad -(17)
\]
The same equation is then calculated for the specular lighting element but with the max ( , ) function being called for the specular threshold against the reflection as calculated from the view point of the player.

To allow the user to choose between blurred or hard edges for each of the lighting areas in the specular model, a range slider was added to the shaders GUI. The value for the slider would be taken to a power of 10 and multiplied by the output of each of the above cut offs, to allow the softening of the masked edges.

The masked areas were then multiplied by the colour values to give the coloured toon shading effect, and then multiplied by the texture value for the pixel to give the final toon lit value to return. The result of the shader can be seen below in figure 10:

![Image](image.png)

Figure 10 - The toon lighting effect applied to the low polygon cartoon tree asset

To allow for multiple lights a second pass was added. Because this pass would be applied on top of the first it would only return a value for the lighting and not the texture, furthermore it had to be blended. To give the best effect, Blend SrcAlpha OneMinusSrcAlpha was used.
In the second pass the Unity defined variable _WorldSpaceLightPos0.w became useful, as the returned value of 0 or 1 determined whether the additional lighting was further directional lighting or point light.

### 3.3 – Toon Outlining

As with the toon lighting, questions arose at the beginning of development for the outlining code. Standard edge detection algorithms, such as the Sobel Operator, might be too complex to be included in a shader that already had so many features. For this reason a technique that didn’t require edge detection would have to be created. The solution was found to be very similar to the lighting method.

By using the edge of the model as seen from the viewpoint as the external edge of the outline, the need for an edge detection algorithm is negated. The outline is calculated by finding the dot product of the normal direction and view direction, and saturating the result. The user defined variable for the thickness of the outline was then subtracted from the saturated result.

The mask was again given a diffusion slider, but to keep the blurred edge internal the outline thickness was added.

Drawing the mask internally also gave the added benefit of not distorting the outline, meaning it wouldn’t interfere with the vertex animation section of the shading. Furthermore, intricate edges on detailed models would not be blacked out to the point where they couldn’t be defined. The result of the completed toon lighting and outlining shader can be seen below on a higher poly count model (Henderson 2014).
3.4 – Optimisation

To create the final shader, all of the aspects from section 3.1 – 3.3 were applied within a single shader. Although there was very little difference in processing power, it was obvious there was a way in which the shader could be optimised. Firstly, all calculations that were done multiple times were stored as variables as soon as they had been calculated for the first time. This significantly reduced the number of calculations done within the shader, and as a result allowed the shader to be compiled for flash if necessary.

Additional optimisation took the form of replacing most floats used within the shader with half variables and fixed variables. Whereas a float is 32bits, a half is only 16bits and has a range of ±60000.000 with a 3.3 decimal precision. Furthermore, a fixed variable is only 11 bits with a range of ±2.0 with a precision of 1/256. For this reason fixed values were used for all normalised values. Most of the other variables were changed to the half type with very few exceptions.

The final optimisation was the lighting. Firstly, all lighting calculations were moved from the fragment shader to the vertex shader. This meant that the number of calls to calculate lighting was drastically reduced. Another example
of the light optimisation was using lerps for the attenuation. A lerp was created between the values 1 and \(1/(\text{distance from the vertex to the light source})\), using the Unity defined variable WorldSpaceLightPos0 as the blend value. This saved valuable processing power, allowing the fast implementation of the perlin noise function.

### 3.5 – Reflective Shading

Although initially being researched as a method of making the vertex animation appear even more warped due to moving reflections of the environment, the idea was discarded and afterwards re-applied in the form of a small body of water at the centre of the scene.

Using the cube map reflections discussed in the literature review, a second shader was created solely for the water. The inbuilt Unity assets were used to create a cubemap matching the skybox.

The cubemap was then sampled using a value calculated by using the Unity function \(\text{reflect}(,\, )\) to equate to equation (11). The values for view direction and normal direction were fed into the equation, and the resulting reflection direction gave a value that could be used by the \(\text{texCUBE}(,\,)\) function.

The final result for the cubemap texturing was multiplied by the value for the diffuse texture to blend the reflection over the painted cartoon style.

Later in the development attempts were made to add further detail to the water shader by adding a specular highlight to respond to the scenes lighting. This turned out not to be possible simply due to the asset the shader was being used on. The asset the material had been applied to was a 2D plane sitting on top of the ground fbx. Because it was completely flat, the lighting from the directional light hit the model at the same angle at all points, meaning the specular highlight had the choice to light all of the water evenly, which was too much, or not at all. For the sake of not overpowering the scene, the specular highlight was completely removed.
4 – Testing and Results

4.1 - Testing structure

The aim of the project can be broken down into two key areas; the artistic element and the computational element. For this reason, it was decided that two methods of testing should be used.

To test the artistic aspect of whether or not the shaders are fit for purpose, a group of 20 testers were selected at random to fill out a questionnaire (Appendix A). To test the computational element of the project, namely, if the shaders are efficient enough, an fps counter was added to a build of the project to measure frame rate.

4.2 - User Testing

The first set of testing was implemented as soon as the shaders were completed and all of the assets were placed in their final positions in the scene. Although some final scripts had not yet been implemented, e.g. the changing between the pre made and custom shaders, everything that was needed to collect feedback was in place. The users were provided with a questionnaire featuring instructions on what was to be done, and a laptop with two applications; one for the unity diffuse shader, and one for the custom shaders.

To limit the chance of influence from knowing how the effect was implemented, only people who hadn’t seen the project in development were chosen. It was also ensured that at no point in the questionnaire, or whilst asking for the subjects’ participation in testing, that the techniques used or name of the proposal were mentioned. The testing group had an age range of 10 years from the youngest to oldest, and the average tester age was 23 years old as can be seen in graph “Figure 17 - Age range of testers”.

Question were kept as unbiased as possible and in order to gather as much “discrete” data as possible, any question that could be asked as multiple choice, without biasing the result, was done so.
First, the testers were told to look around the first scene (Figure 12). They were then asked “How would you describe the atmosphere of island A?”

![Figure 12 - The scene, described to testers as “Island 1”, with diffuse shaders](image)

As shown by “Figure 18- How testers described the scene with inbuilt diffuse shaders”, the 3 main ways testers described the atmosphere of the scene was “Cheerful”, “Colourful” or “Pleasant”. This was the expected result of the how the scene should appear due to its basic, but brightly lit appearance. 25% of testers also ticked the box for “other”, describing the scene as “calming”, “calm”, “tranquil” “natural” and “bright”. Each of these words also fits the desired appearance of the scene.

After filling in the first question, test subjects were given instructions to view the scene where the custom shaders had been applied (Figure 13).
“Figure 19 - How testers described the scene when custom shaders were applied” shows the data that was collected. The data shows that the shaders were successful in changing the scenes atmosphere, with most testers reporting the scenes atmosphere to now be “Surreal”, “Psychedelic”, “Strange” or “Hallucinatory”. In fact, when comparing the two sets of feedback, every single tester changed their description of the island by between 2 and 9 descriptive words (Figure 20). This means that for the sample group of 20 users, there was an average change of 40.84% between the answers given in question 1 and question 3 (Figure 21).

Whilst testing, the participants were asked to describe any changes they noticed. In order to not point out changes that had been made, which could bias the data, this question was not multiple choice. After all results had been handed in, the data was parsed through and like answers were grouped to give a set of relative data. Fields for this data were changes caused by the shaders; Distorting Trees, Distorting Rocks, Reflective Water, Toon Shading and Darker Colours. Additionally, a field was added that counted changes that users thought they saw but weren’t actually present. The results of this question can be seen in graph “Figure 22- What key differences were there when custom
shaders were applied”. The testers noticed a range of different changes but it should be noticed that 100% of testers commented on the distortion of the trees and 85% of testers commented on the toon shading effect.

In comparison, only 55% of testers noticed the distortion of the rocks and a disappointing 35% of testers noticed the cubemap reflections in the water, despite the fact that the water and rocks were the centre piece of the island.

Interestingly however, 20% of testers noticed changes to the scene that didn’t actually occur. These included testers thinking there were “more rocks”, the level was “more spread out”, was “brighter” and one tester claiming the water was “flat with the ground”. This has a slight correlation with the descriptions being used in question 3, as half of the subjects that noticed non-existent changes, also described the scene as disorientating, which could potentially indicate a causal link.

The final question asked to the testers was whether they noticed a change in performance between the two applications. To keep this information as reliable as possible, the same computer was used for all testing regarding this question. Additionally, the application was run at the highest possible graphics quality so that all cases for testing were identical. Figure 23 shows that a very conclusive 85% of testers didn’t notice any change in performance and a further 10% said there might me, but they couldn’t be sure. This means that in a testing range of 20 subjects, only 1 person felt the shaders affected the processing of the project. These results will hopefully be confirmed by the technical testing.

4.3 - Technical Testing

In addition to the user testing, technical testing was also carried out to check the efficiency of the application. Whereas the user testing was carried out in 2 separate applications, the technical testing was carried out in the final completed application. The specification of the computer used for testing can be seen in appendix 3 - Debug Data.
The application was run at aspect ratio 1366 x 768 and at 2 graphics qualities to compare frame rates at different settings. The code used for checking the fps was taken from the Unity Wikipedia (Unity Technologies 2014b). The C# implementation was used and attached to a GUIText game object in the scene.

When tested at “Fantastic” quality the frame rate was a consistent 60FPS whether the custom or Unity shaders were used. This frame rate didn’t change regardless of how many game objects were displayed on the screen, or whether the user stood on the island or was orbiting it from the title screen figure 14.

![Figure 14 - The “island” as shown from the title screen](image)

Due to this, testing was implemented at a lower graphics quality. When run on “Fastest” settings, the most basic level of graphics, frame rate changes could be seen.

When looking at the “island” from afar, the title screen, the FPS ranged from 450.43 to 504.48 when standard Unity diffuse shaders were used, but only at 442.45 to 453.06 when using the custom shaders.

Furthermore, when viewing the scene up close, after entering the first person, the frame rate change was even more apparent. When using the diffuse shaders, frame rates ranged from 611.03 to 659.23 depending on how many
game objects were being rendered to the screen. Comparatively, when using the custom shaders the frame rates severely dipped to a range of only 425.72 to 502.19 fps, an average dip of 26.95% (to 2dp).

This range of results will be used in section 5, the discussion.
5 - Discussion

From observing the results gathered through various forms of testing, it can be concluded that shaders can be used as an effective and efficient method to enhance a scene, and can completely change an atmosphere. But whilst there appear large benefits to the technique, there are also substantial flaws in regard to the ideas posed in the introduction.

In regard to the user testing conducted, whilst results were positive in proving the artistic element of the research question there is also room for scrutiny vis-à-vis the multiple choice questions. Although, the questions regarding atmosphere (1 and 3) provided valuable discrete data for testing there is a chance that due to the overwhelmingly positive feedback that there may have been an element of unintentional bias. By providing a selection of answers to choose from, the results received, whilst easier to collate, may lead subjects to choose answers they wouldn’t usually have thought of. Choosing from multiple choice answers may lead subjects to assign higher “truth” values to false statements (Butler and Roediger III 2008) causing testers to believe answers that weren’t actually true. This could explain results such as in question 1, where one user described what should be a pleasant scene as “Threatening” which was only included to provide descriptions that shouldn’t have been applicable.

Furthermore, by limiting the answers available and having some words which clearly describe the scene, and others that should never be considered, with only a few bridging the gap between these two extremes, the testers may have been encouraged towards these answers through simply negating the wrong choices and choosing the from remaining selection, leading to the highly positive results.

Continuing from the theory of implanting ideas is the response to question 4. As stated before, all user testing was completed using the graphics quality of “Fantastic”, which was proved during the technical testing to not have frame
rate issues regardless of which shaders were used, and to always maintain a steady stream of 60 frames per second. Despite this, one tester responded to the performance question by saying the custom shader program was “laggy” and “unresponsive” compared to the diffuse shader program, and a further two testers became unsure and wrote answers equivalent to “maybe”. This furthers the idea that the “performance issues” were a psychological response to being asked if there was a change in performance, the testers instantly assuming that because they’re being questioned about the matter, that there must be something to be aware of.

Although all of the above questioning of the testing method is worth considering, it should be noted that it is only speculation due to the overwhelmingly positive results.

The scene was set up to have a cheerful and pleasant atmosphere using colourful assets, and the shaders were designed to change that atmosphere to be strange, hallucinatory and disorientating. The user testing undoubtedly proves that the techniques worked. The point of disorientation is further proved, as mentioned before, by the fact that four testers noticed changes that didn’t actually exist.

Regarding the technical testing, it is evident that the techniques proposed by this dissertation are undoubtedly efficient enough to be used in a professional setting. Whilst testing at high graphics quality there was no change in fps caused by the custom shaders at all. Additionally, although it was stated in the results that there was a large change in fps when testing at lower graphics qualities, due to the incredibly high frame rates achieved, these dips in performance are well within the realm of acceptable standards. This could be completely negated by limiting the frame rate of the application to 60 fps for all graphics qualities, due to this being the most common frame rate used in the industry anyway. Doing this would negate all frame rate issues caused by the shaders at lower qualities.
As a final point of discussion, the ideas proposed in the introduction with regard to reducing the involvement of art pipeline should be mentioned. First and foremost it should be noted that at no point should the work of the animators be completely removed from the art pipeline. As already stated in the literature review, the technique of vertex animation by code is no substitute for rigging an animation with respect to character animation. It would be highly inefficient to attempt to weight and animate full animation cycles, and would take far longer to do than the full rigging and animation process, with an end result that wouldn’t meet the standards needed. That said however, the process can be used as a viable replacement for parts of the animator’s job with some positive, but also some negative impacts.

With the scene in question, the process of rigging, skinning and animating the models seen would only take a matter of hours to complete. With that said, the results would vary drastically. Each asset would have a repeating animation cycle that continued throughout the playtime of the level with no variation, as opposed to the random movements provided by the shaders produced. Furthermore, all trees would animate in the exact same way unless further animation cycles were created.

Another advantage to the proposed method is the assurance that no matter what graphical level the application is played at, the results will always be the same (Figure 15). Compared to this, animated models imported into Unity using joint based rigs can have issues when turned to a lower graphics setting (Figure 16).
Finally, the frame rate would be reduced due to the implementation of the animations, and shaders would still be needed to achieve the toon shading effects, whereas the shader code has been proved to be efficient enough to be used in a professional environment, and fulfils all requirements on its own.

A negative effect of the shaders that would be solved by using prebaked animations is the issue of collision detection. Whereas a prebaked animation would collide with the player using mesh collision detection, the method
proposed is not able to do this. As far as the collision detection algorithm is aware the assets collision detection points are where the model is initially plotted. Any movement of the vertices is not registered and so players are able to intersect with geometry as it warps away from its initial position.

As a final point, whereas the shader may produce some movements that are not desired due to its random/procedural element, the pre-rendered approach would guarantee that all animations are acceptable in advance and so glitches are less likely to occur.
Final Words and Future Work

Whilst there are shaders than can achieve far more advanced rendering techniques, shaders like the one produced definitely have a place within the industry. The use of shaders as a time saving technique for other pipelines is not a widespread idea as of yet, but is definitely an area to which more time should be dedicated.

This dissertation was focussed primarily on the aspects of non-photorealistic shaders such as toon shading and vertex animation, but the ideas proposed and implemented are something that could greatly benefit the future of games production if expanded further into other fields. Whether it is through continuing the work of vertex animation with procedural elements to create a fast and efficient method of water simulation, or using shading techniques to alter textures to speed up the process of asset creation, there are many uses for shaders outside of the realm of basic effect rendering.

With newer and more power graphics rendering capabilities becoming available since the undertaking of this project, namely in the form of a new generation of consoles, graphics programming is a more important part of the pipeline than ever before.

The techniques introduced and advances made by this project, whilst promising, aren’t yet sufficient enough to warrant significant cutbacks in the art pipeline as initially proposed in the introduction, but are definitely a starting point that should be considered to improve the efficiency of the game creation pipeline.
Appendices
Appendix A - Questionnaire given to testers to collect data

Island Questionnaire

Name:
D.O.B.

Take a few minutes to look around Island A.

1. How would you describe the atmosphere of Island A? (Use as many of the descriptors as you want and underline your answers)

   Dark     Cheerful     Surreal     Tense     Psychedelic     Colourful     Strange
   Pleasant     Threatening     Hallucinatory     Disorientating     Other

   If "Other" please write here:

Now take a few minutes to look around Island B.

2. Describe any changes you notice:

3. How would you describe the atmosphere of Island B? (Use as many of the descriptors as you want and underline your answers)

   Dark     Cheerful     Surreal     Tense     Psychedelic     Colourful     Strange
   Pleasant     Threatening     Hallucinatory     Disorientating     Other

   If "Other" please write here:

4. Did you notice any performance change between Island A and Island B?
Appendix B - Graphs to display questionnaire results

**Figure 17 - Age range of testers**

**Figure 18 - How testers described the scene with inbuilt diffuse shaders**
Figure 19 - How testers described the scene when custom shaders were applied

Figure 20 - Number of changes of descriptive words between the 2 versions

Figure 21 - Average percentage of changed descriptive words between the 2 version
Figure 22 - What key differences were there when custom shaders were applied

Figure 23 - When asked if there was a change in performance
Appendix C - Debug Data

Initialize engine version: 4.3.4f1 (e444f76e01cd)
GfxDevice: creating device client; threaded=1
Direct3D:
  Version: Direct3D 9.0c [nvumdshim.dll 9.18.13.2702]
  Renderer: Intel(R) HD Graphics 4000
  Vendor: Intel
  VRAM: 944 MB (via DXGI)
  Caps: Shader=30 DepthRT=1 NativeDepth=1 NativeShadow=1 DF16=1
         INTZ=1 RAWZ=0 NULL=1 RESZ=1 SlowINTZ=0
Begin MonoManager ReloadAssembly
Platform assembly: C:\Users\Jack\Documents\Honours Project\Debug\WarpDebug_Data\Managed\UnityEngine.dll (this message is harmless)
Loading C:\Users\Jack\Documents\Honours Project\Debug\WarpDebug_Data\Managed\UnityEngine.dll into Unity Child Domain
Platform assembly: C:\Users\Jack\Documents\Honours Project\Debug\WarpDebug_Data\Managed\Assembly-CSharp-firstpass.dll (this message is harmless)
Loading C:\Users\Jack\Documents\Honours Project\Debug\WarpDebug_Data\Managed\Assembly-CSharp-firstpass.dll into Unity Child Domain
Platform assembly: C:\Users\Jack\Documents\Honours Project\Debug\WarpDebug_Data\Managed\Assembly-CSharp.dll (this message is harmless)
Loading C:\Users\Jack\Documents\Honours Project\Debug\WarpDebug_Data\Managed\Assembly-CSharp.dll into Unity Child Domain
Platform assembly: C:\Users\Jack\Documents\Honours Project\Debug\WarpDebug_Data\Managed\Assembly-UnityScript-firstpass.dll (this message is harmless)
Loading C:\Users\Jack\Documents\Honours Project\Debug\WarpDebug_Data\Managed\Assembly-UnityScript-firstpass.dll into Unity Child Domain
- Completed reload, in 0.061 seconds
desktop: 1366x768 60Hz; virtual: 1366x768 at 0,0

<RI> Initializing input.

<RI> Input initialized.

Platform assembly: C:\Users\Jack\Documents\Honours Project\Debug\WarpDebug_Data\Managed\Boo.Lang.dll (this message is harmless)
List of References


Lex-DRL 2013. *2D/3D/4D Optimised Perlin Noise Cg/HLSL Library (cginc)*.

Linear Curve Games 2013. *Far From Home*. Dundee, Scotland: Care Inspectorate.


Bibliography


