Adaptive Abstraction using Non-Photorealistic Rendering in XNA

by

Conor Hanratty, B.A.,B.A.I.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Computer Science

University of Dublin, Trinity College

August 2009

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Conor Hanratty

September 7, 2009

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Conor Hanratty

September 7, 2009

Acknowledgments

I would like to thank my supervisor, John Dingliana, for all his help and advice throughout the project as well as everyone from GV2 for their input. I would also like to thank all of my friends and family for their support.

Conor Hanratty

University of Dublin, Trinity College
August 2009

Adaptive Abstraction using Non-Photorealistic Rendering in XNA

Publication No. _____

Conor Hanratty
University of Dublin, Trinity College, 2009

Supervisor: Dr. John Dingliana

The purpose of this dissertation is to demonstrate a system which uses adaptive abstraction in the rendering of a 3D scene.

Scenes can be drawn with focus on certain objects or certain regions in the scene by removing extraneous detail from unimportant areas. This project uses non-photorealistic rendering (NPR) to stylize and remove detail from unimportant areas. In doing so, a framework is created to attract user focus to certain areas of the scene based on distance from the viewer and overall importance. Edge-detection and edge darkening are used on abstracted objects to assist in emphasizing important details on an object.

This dissertation discusses research done into the state of the art in the fields of non-photorealistic rendering and adaptive abstraction. Various methods of NPR are discussed and compared.

A demonstration project using this algorithm is created which demonstrates this functionality. The demo contains versions of the application on both the PC and the Xbox360 games console. The application is interactive, allowing a user to roam around a scene and change the levels of abstraction of a number of objects. The details of the creation and results of this application are discussed.

Contents

List of Figures

Chapter 1

Introduction

Scenes can be drawn with focus on certain objects or certain regions in the scene by removing extraneous detail from unimportant areas. This is known as abstraction. This project uses painterly non-photorealistic rendering (NPR) to stylize and therefore remove detail from unimportant areas. In doing so, a framework is created to attract user focus to certain areas of the scene based on distance from the viewport and overall importance. Edge-detection and edge darkening are used on abstracted objects to assist in emphasizing important details on an object.

Chapter 2 of this dissertation discusses research done into the state of the art in the fields of non-photorealistic rendering and adaptive abstraction. Various methods of NPR are discussed and compared.

A demonstration project using this algorithm is created which demonstrates this functionality. The demo contains versions of the application on both the PC and the Xbox360 games console. The application is interactive, allowing a user to roam around a scene and change the levels of abstraction of a number of objects. The details of the creation of this application are in Chapter 4 and the results are discussed in Chapter 5.

1.1 Motivations

Non-photorealistic rendering (NPR) is a popular field of computer graphics which, as its name suggests, does not primarily concern the realistic representation of 3D environments. Instead, it focuses on using more stylistic and expressive artistic styles to represent a given scene. These styles can be reminiscent of artistic illustration (sketching, pen and ink), or of paintings (painterly rendering). NPR can be used as a medium to add more information about a scene or it can create simpler versions of complicated scenes making them easier to comprehend. For example, a blueprint of a building is not photorealistic, but can convey much more visual information about the building.

Though much work has been done to date in the field of non-photorealistic rendering in real-time, very little has been done in relation to XNA. Aside from a simple demonstration of pencil shading

and cel-shading which can be found on the XNA Creators Club website[1], comparatively little work has been done in this field.

This seems like a large gap considering both the abundance of NPR on other platforms (predominantly OpenGL and Cg/GLSL) and the potential major benefits of NPR in the field of video games. Numerous games such as *Valkyria Chronicles*(Playstation 3, 2008), *Madworld*(Nintendo Wii, 2009), *Ōkami*(Playstation 2 & Nintendo Wii, 2006) and *The Legend of Zelda: The Wind Waker*(Nintendo Gamecube, 2002) use NPR styles to improve their artistic style and gameplay.

Research into NPR could be undertaken using the XNA development toolkit, allowing a framework for PC and Xbox360 games which use NPR styles. From this *video game* perspective, an examination into a subject related to video games would be quite useful. For example, user attention can be drawn to certain areas of the screen using NPR.

Halper et al.[5] discuss how NPR can be used to influence user perception and judgment. It was shown how users could be influenced in their choice of navigating a certain scene. By adding more detail to one path than to another, a group of users tended to choose a path in a scene with greater graphical detail than another path (see figure 1.1). This removal of detail from less important objects and addition of more detail to more important objects in computer graphics is known as abstraction.

Figure 1.1: Example images from Halper et al.[5] When prompted, users tended to choose the more detailed paths to reach a goal.

In games, such a system to exploit this phenomenon could be very useful in encouraging a player to follow a certain path or perform a certain action. *Adaptive* abstraction, where the levels of abstraction in a scene change depending on the view of the scene or the actions of the user, could be used to assist a users perception of gameplay. This could be as simple as providing more detail in a scene to objects that are closer to the screen, perhaps because the player has moved closer to the object in order to examine it visually.

[1]This demo can be found here - http://creators.xna.com/en-US/sample/nonrealisticrendering

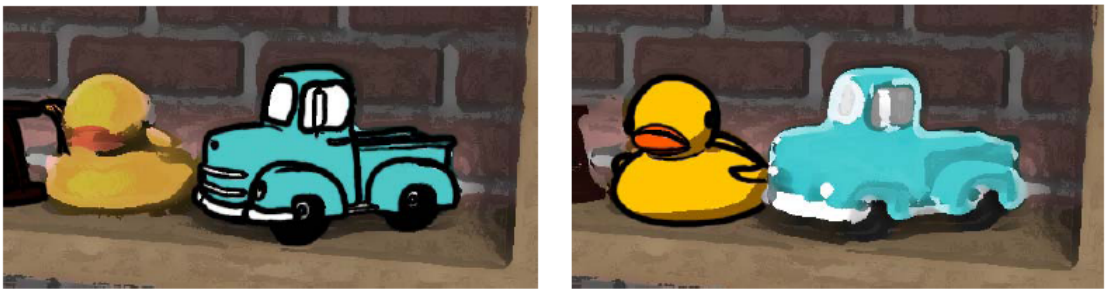

Figure 1.2: Example images from Halper et al.[5] Two different rendering styles (cel shading and oil paint) are used to define an object in a scene.

1.2 Objectives

The purpose of this dissertation is to demonstrate a system which uses real-time painterly NPR techniques to render an interactive scene in XNA. Furthermore, different levels of adaptive abstraction are used depending on the importance of an object in the scene or the distance an object is from the viewport (for example, closer objects receive more detail than distant ones). An application has been created which demonstrates this functionality. It is an interactive application for the PC or Xbox360 that gives the user control of a small spaceship in a three dimensional scene. Objects close to the ship are rendered normally, while more distant objects appear less detailed and more stylistic.

The processes of selective abstraction and stylization are handled on the Graphics Processing Unit (GPU) using HLSL shaders. Firstly, the application renders the depth information of the scene to a texture. This stores the distance from the viewport of each visible object for the later rendering passes. The scene is then rendered as normal (with textures), followed by a post-processing pass which abstracts certain areas of the scene depending on information from the depth texture.

Furthermore, certain models in the scene can be specified to be of a certain level of salience (importance) and be rendered normally regardless of position in the scene. Thus, the players attention can be drawn to a clearly rendered object in the distance against an abstracted background.

Different image processing effects are used depending on the levels of abstraction. These methods include Sobel filter edge detection, Gaussian blur and Kuwahara filters.

The primary contribution of this dissertation is an algorithm and accompanying program that

1. uses adaptive abstraction to direct user focus,
2. generates a scene using a painterly NPR system,
3. has a stable framerate for a number of resolutions,
4. is deployable on Xbox360 an PC.

Chapter 2

State of the Art

This chapter aims to explore and discuss previous research done in the fields of non-photorealistic rendering and abstraction.

2.1 Non-photorealistic Rendering

Computer graphics has made great strides in recent years in rendering scenes that are as close to realistic as possible. This field of computing has had a profound impact on the development of movies, animation and video games. However, in some cases, it is desirable not to render a scene photorealistically.

Non-photorealistic rendering or NPR is an area of computer graphics whose focus is primarily not on photorealism but on mimicking existing artistic styles. These styles include painting, sketching, cartoon or 'toon' shading and technical illustrations (for example, blueprints). Such styles can be adopted for a number of reasons. For example, in film making, NPR can be used to create a 'comic book' feel to the cinematography, allowing the artistic presentation of a film to be linked to the themes of the film, for example in *300*(2007) or *A Scanner Darkly*(2007).

Although in film NPR is almost always pre-rendered, much research has also gone into the implementation of real-time NPR. This research can have many uses. For example, a video game may be better served by being rendered as a watercolour painting if such a style would fit the theme of the game. Examples of such rendering can be found in the games *Ōkami*(2006) and *Prince of Persia*(2008). In *Ōkami*(see figure 2.1), a watercolour style is used to complement one of the gameplay elements where the main character must fight off enemies and solve puzzles using a magical paintbrush. In *Prince of Persia*, the story is presented as an epic fairy tale, and cel-shaded graphics and painterly rendering help to reinforce the storybook feel of the game.

Figure 2.1: Image from the game *Ōkami*. Copyright Clover Studio & Capcom

2.1.1 Cel-shading/pencil sketching

Lake et al.[7] discussed the method of hard shading (setting the light level of each object manually)) in relation to cel-shading (see figure 2.2). Instead of calculating colours per vertex, a texture map with a limited number of lighting shades from black to while is created. First, one must find the dot product of the normal of a vertex with the normalized vector of the light direction. If this value falls below a certain threshold, the object is in shade and is coloured accordingly from the texture map. If not, then the object is set to be a brighter colour. By using only a few colours, visual detail is reduced to give a hard edge of light/darkness that follows the contours of the object.

Lee et al.[8] discuss pencil rendering in real time to simulate the effect of pencil sketches on a 3D model (see figure 2.2). Their method involves detecting contours using depth and normal images acquired from the scene (a similar method is discussed in Chapters 3 & 4). The system then uses this information to create contours. Before being rendered, the contours are first 'shaken' using a sine function to approximate the shaking errors that would be produced by a human hand. The scene is then shaded by using a pre-rendered texture which mimics the colour and material of pencil strokes on paper. The final image does however suffer from a certain degree of temporal incoherence, where a certain line may not correspond to where that line should appear to be from one frame to the next.

2.1.2 Painterly rendering

Physical Simulation

Much research has gone into NPR which mimics watercolour and painterly styles. The work of Curtis et al.[2] extensively documents a system which simulates the artistic effects of watercolour. It does this by creating a physically correct shallow-water fluid simulation to represent the interactions between the

Figure 2.2: Left: Demonstration of cel-shading from Lake et al[7]. Right: Demonstration of pencil-shading from Lee et al[8].

paper and fluid on a watercolour canvas. They investigate many of the tricks used by real painters to create a scene, such as dry brush effects, edge darkening, intentional backruns, separation of pigments and use of flow patterns.

Among the numerous physical properties simulated are the velocity and pressure of the water, concentration of pigment, viscosity and viscous drag, etc. Furthermore, the paper is simulated as multiple layers; a shallow-water layer, a pigment-deposition layer and a capillary layer. This simulation creates a very detailed and realistic result but is also very computationally expensive. This makes it unsuitable for a real-time application like the one proposed here.

Laerhoven et al.[12] attempted a similar system, recreating the physical interactions between paint and paper, but with a real-time, interactive implementation. Their system involves a 3 layer paper model similar to one proposed by Curtis et al.

Paint is considered to consist of pigment and water. In this model, three different states exist for them (see Figure 2.3);

- Water and pigment on top of the paper
- Pigment deposited on the surface of the paper
- Water absorbed into the paper

For each of these states, a different layer is simulated as a 2D grid of cells. Each of these grids is simulated in real time. On top of the paper, the paint is simulated as a shallow layer of water. In this layer, the velocity at each field is expressed as a vector field and the fluid flow is simulated by a

two-dimensional Navier-Stokes equation. In the pigment layer, pigment is deposited by the fluid and can move back and forth between the other layers. The capillary layer mimics the internal structure of the paper diffusing the water and pigment to replicate the irregularity of the paper texture. The texture itself is procedurally generated.

The final result of this method was a much faster, but still realistic simulation of a watercolour painting. However, the results presented show scenery which maintains video-like frame rates for grid sizes of 400x400. These would be much lower than the resolutions of the hardware used for this dissertation (the maximum resolution of the Xbox360 is 1280x720). For this reason, a physically based painterly rendering system was not used for this project.

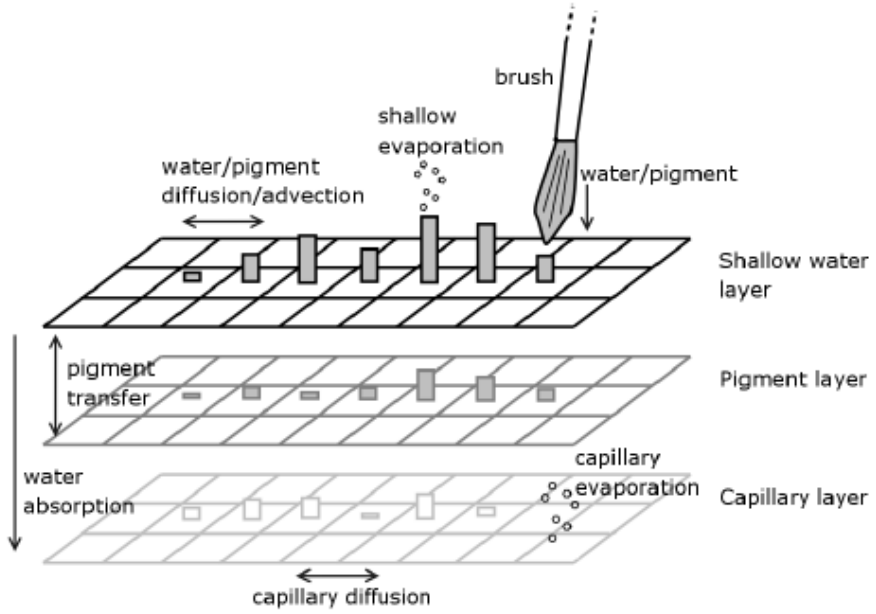

Figure 2.3: The 3-layer paper model used by Laerhoven et al.[12]

Real time painterly rendering

Meier of Walt Disney Animation[10] developed a painterly rendering system for animation which uses 3D particle systems which are rendered as 2D brush strokes in screen space.

First, a particle set is created which defines the geometry of the object being rendered. The parametric surface of the object is tessellated into triangles in which the particles are placed. The ratio of the area of each triangle to the surface area of the whole surface determines the number of particles per triangle. Each particle is transformed into screen space and sorted in order of depth. For each particle, a brush stroke is drawn. The appearance of each of the strokes is determined by a number of parameters (color, shape, size, orientation, etc.) specified by a set of reference images (see 2.4). The renderer examines the brush stroke attributes in the reference images at the screen space

location of the particle and renders a brush stroke that is used in the final rendered into the final image.

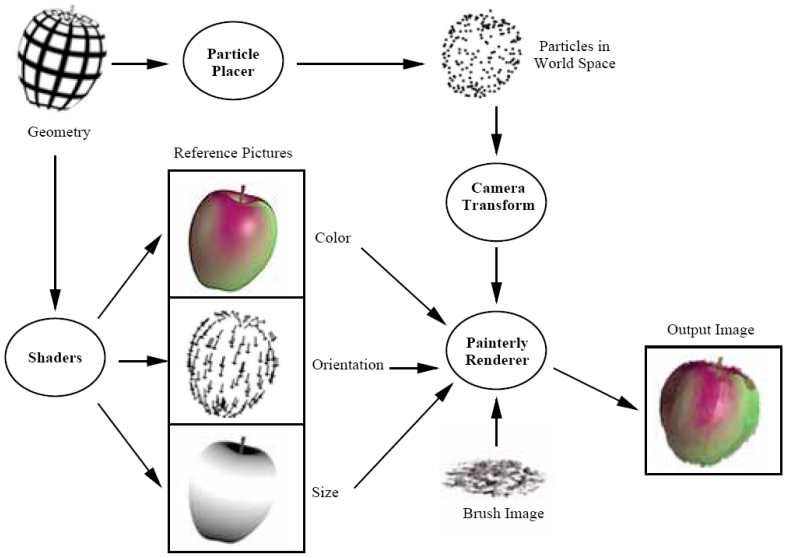

Figure 2.4: Rendering pipeline used by Meier.[10] Upon locating the particles on the surface of the object, shaders are used to calculate the parameters for the painterly rendering. A brush texture is then drawn to each particle in a manner consistent with the parameters.

The results were impressive, but the system was not designed for real-time use. Thus all rendering must take place offline. However, Sperl created a system which was based on Meiers use of particle systems but which ran in real time[11]. The system accelerates the rendering process using the GPU and Cg, a shading language. In addition, the polygon meshes of the objects are converted into particle systems in a pre-processing step to save time. The system contains two passes; the first writes depth and colour information to textures, while the second uses this information to render the brush strokes to the particles as billboards in a similar manner to the Meier method. This method of writing colour and depth to textures at different stages would later be used in the algorithm for this dissertation.

Using a Pentium 4 with 2.4 Ghz and a GeForce Ti 4600, a scene containing 100,000 particles could be rendered at 8 fps. While implementing an offline method like Meiers' at such a speed is impressive, it could be considered too slow for an application like a video game. However, the idea of writing depth information in a first rendering pass is a useful concept which is employed throughout this project.

Another notable example of real-time painterly rendering is the work of Bousseau et al.[1](see figure 2.5). This system is designed to take still photographs or 3D models as input and apply a number of effects to give an interactive watercolour system. It uses a series of post-processing effects to apply watercolour-like layers to the final image. These stages include toon shading, edge 'wobbling' (distortion effect along the edges of an object to mimic paper granularity) and edge darkening (to imitate pigment naturally migrating towards the edges of an object). Furthermore, noise textures are

applied to simulate pigment dispersion, turbulent flow of paint and the texture of paper. This idea would later be investigated for this dissertation.

Figure 2.5: Example of painterly rendering of a 3D model from Bousseau et al.[1]

Lei & Chang[9] propose a watercolour NPR system which works in two stages. First, using a method originally used by Kubelka et al[6], the colour in RGB space of a pixel is found using the optical mixing of pigments. This involves setting each pigment a set of absorption and scattering coefficients and a constant of granulation. The Kubelka-Munk model is then used to compute the resulting RGB colour. Next, edge darkening takes place using a Sobel filter (which ended up as one of the techniques tried for this project) and a grainy texture is applied to the whole scene to simulate the peaks and valleys present on a piece of paper. Figure 2.6 shows the effect of a teapot rendered with 4096 polygons. Their results indicate real-time scenes rendering at 20 frames per second at 512x512 pixels.

2.1.3 Adaptive abstraction

To render an object with abstraction implies an effort on the part of the artist to remove certain unimportant details from a scene. For example, if rendering a brick wall, though the more physically correct solution would be to render it in such a way that every brick was visible, in certain circumstances it may be more appropriate to render only a few bricks across the wall. Doing so can still convey to the user the existence of a wall without showing too much visual information. This can encourage the user not to focus too much attention upon it, allowing them to divert more attention other objects in the scene. This can allow a viewer to more quickly identify other, more important items in the scene that the artist was trying to draw the viewer's attention to.

Figure 2.6: A teapot, rendered in the style of Lei & Chang[9].

Adaptive abstraction implies a system whereby the amount of abstraction being applied to a certain object can be changed depending on whether or not that object's importance within the scene changes. In the context of video games, there are numerous reasons a developer may wish to direct a users focus towards a certain object in a scene. For example, if a player needs to collect a certain item in an area in order to progress, the item could be highlighted by abstracting the scene around it. Or if a player gets lost in a certain area and doesn't know where to go for the game to progress (a common problem in video games), abstraction can be used to draw their attention to the correct path out of the area. Furthermore, based on the idea that a player will naturally move closer to an object or area they think is of importance to get a better view, abstraction could be used on far away objects in a scene in order to bring an object the player is closer to into greater focus.

Winnemöller et al.[13] created an abstraction framework that abstracts imagery by changing the contrast and detail of visually important features. This method works by minimizing contrast in low-contrast areas which are not likely to be particularly important and maximizing contrast in areas of high contrast using difference of Gaussian edge-detection.

The basis of the algorithm is to find and emphasize perceptually important information. The paper is based on the assumptions that

1. Human visual systems are designed to work on features
2. Changes to these features are important perceptually
3. Polarising such changes will allow for automatic image abstraction to take place

Furthermore, the paper suggests that visual salience (importance) can be determined by the effects of luminance, color opponancy and orientation on low-level human vision. To maintain the real-time aspect of the algorithm, it works by only dealing with luminance and colour opponancy. It assumes that a large change in either of these two values indicates a salient feature such as a boundary.

Redmond and Dingliana presented[3] a method of using real-time non-photorealistic rendering for abstraction which uses object and image based techniques. The result is a scene which used edge detection and painterly rendering to apply different levels of abstraction to different objects in a scene, allowing unneeded detail to be removed and allowing the saliency of important objects to be increased if necessary.

To achieve NPR painterly rendering, a version of the Kuwahara filter is applied to the scene using image space GPU pixel shaders (an detailed explanation of the Kuwahara method can be found in section 3.2.2). Different levels of this painterly image processing are then applied to an object in a scene depending on the level of abstraction required. Furthermore, object based edge detection and difference of Gaussian image based edge detection (see section 3.1.2) are used to find detailed edges. This is done because of the benefit edges present in directing user focus.

Depending on where in a scene an artist may want to draw the attention of the user different levels of abstraction were used by implementing a tiered system of focus. For example, unimportant objects could be rendered with large amounts of painterly rendering, as well as having its RGB colour values faded slightly and having edges slightly transparent. By doing this, visual detail on the object can be significantly reduces, contrasting heavily with a more important object scene which could receive no abstraction at all. This would draw a user's focus to the more important item, allowing a new level of artistic control on the presentation of a scene.

As explained in another publication, Redmond and Dingliana[4] conducted user tests to investigate how using adaptive abstraction can influence user gaze. In their experiments, they asked participants to take part in simple search and recognition tasks.

The first experiment involved each participant to view a set of animations which contained a number of spheres of the same size moving about a central axis. Their placement was random and textures were applied to each sphere. The first 8 animations used normal local illumination after which the participants would view 56 more using 7 different types of non-photorealistic rendering (8 animations each). The NPR filters included Kuwahara, Winnemöller, edge detection, colour quantization and changes in colour, brightness and luminance. The participants had been shown a particular type of textured sphere before the experiment and the goal was to find that specific type of sphere somewhere in each scene. Half of each of the abstracted scene sets used multiple abstraction levels to remove unwanted detail and add salience to the target object.

The results of the experiment indicated that the average recognition speeds of the participants was 2.53 seconds for when the target object was focused and 2.79 seconds for when rendered normally. Thus, participants had improved reactions when NPR filters were used.

Another experiment was done focusing on how using multiple styles could influence user times. The second experiment was run with static scenes of varying size. The participants were shown one hundred images of one hundred and fifty randomly sized and positioned spheres. The spheres were also given a random texture. The first twenty images were rendered normally, while the next eighty were rendered with four types of NPR, using a number of techniques for each method. These methods included

1. Kuwahara filter, variation in saturation and brightness
2. Kuwahara filter, edge detection, colour quantization, variation in luminance
3. Winnemöller filter, variation in saturation and luminance
4. Winnemöller filter, edge detection, brightness variation

Half of the images focused on the target object while the other half focused on another object at random. The experiment found that reaction time for when the target object was focused (1.45 seconds) was somewhat faster than in a scene rendered normally (1.64 seconds) and significantly faster than when a random object had been abstracted. Furthermore, it was found that the methods of NPR used had an effect on reaction times, with the first group of techniques having the fastest average time.

Further experiments were performed using eye-tracking to gauge the users' gaze behavior. Using a similar set of animations to the first experiment, in which one sphere out of a large number received a specific brick texture. The users would have to count the number of these textured spheres in each scene. In half of the animations the target spheres are the focus of the abstraction algorithm, while in the other half, a random object was the focus. When the experiment was completed, the eye-tracking software evaluated how far from the abstracted objects the users gaze was. The experiment found that abstraction can heavily influence user gaze, in particular, drawing a users attention to a region in focus in the first second of each scene.

The final experiment used the eye-tracking technology on an urban scene. Participants were shown images of urban scenes a number of times; some rendered normally, some abstracted. In the abstracted scenes some buildings made the focus on one of four buildings of varying prominence within the scene. The participants were simply made to look at each scene while the eye-tracking technology recorded their gaze. The test proved that each user spent more time looking at a target object if it was in stylized focus.

2.2 XNA & HLSL

2.2.1 XNA

Microsoft XNA is an application framework for computer game development on the Xbox360 and PC. It is based on a native implementation of the .NET Compact Framework 2.0 for Xbox360 and .NET Framework 2.0 on Windows. It runs using C# and XNA Game Studio Express IDE. It was primarily created for students and independent game developers to develop games as a hobby or for commercial use and share them over Xbox Live or Windows Live.

Its primary benefits to this project concern the ease with which interactive 3D games can be created.

2.2.2 HLSL

High Level Shading Language or HLSL is a shading language created and developed by Microsoft. It was developed as a similar language to GLSL, the shading language primarily used for OpenGL development, for use with the Microsoft Direct3D API. Furthermore, it is very similar to the Cg shading language developed by NVIDIA, but HLSL will only compile in DirectX, whereas Cg can compile in OpenGL also.

Chapter 3

NPR Overview

For this project, a wide variety of non-photorealistic techniques were used to create a varied rendering environment. This chapter explains and compares some of the methods considered for this project.

3.1 Edge Detection

Firstly, a number of different styles of edge detection were researched. This was done so that edges which suggest the shape of the object can be easily seen by the user and the shape and texture of the object can be conveyed. Edge detection is primarily either object based or image based.

3.1.1 Object-based edge detection

Object based edge detection involves identifying edges based on the geometric properties of the 3D model; i.e. vertices, edges and polygons, etc. This method of edge detection can be very useful as it discerns exact edges and is completely view-independent. This means that from every perspective, all visible edges appear and are coherent.

Depth & Normal mapping

This method is based on an edge detection algorithm created by Microsoft[1].

It involves finding edges on an object based on two parameters. For every pixel on screen, whether that pixel is at the silhouette edge of an object or not is determined by the distance from the viewport of the vertex at that pixel and the value of the normal of the vertex at that pixel. This information is accessed by the GPU by storing it in a texture before the scene is rendered.

First, the scene is rendered to a texture with a shader that records the normal of each vertex as the RGB values of the output and the depth of each vertex as the alpha value of the output. The resulting texture is sent to the GPU for the main rendering pass. As the scene is being rendered, a

[1]The demo itself can be found on the XNA website at http://creators.xna.com/en-US/sample/nonrealisticrendering.

calculation is done to determine the change in depth and normal values across that pixel. Figures 3.1 and 3.2 demonstrate the different stages of the process.

(a) The source image (b) Depth information as a texture (c) World space normal information

Figure 3.1: Steps taken for edge detection. Images from public domain.

(a) Large changes in normal and depth indicate edge pixels (b) Source image composited with edge image

Figure 3.2: Steps taken for edge detection. Images from public domain.

Section 4.2.4 explains the method as used in this dissertation.

Wireframe enhancement

One popular trick to mimic the appearance of dark edges is to draw the model twice; once for edges and once for the rest of the object. This can be done by first setting the backface culling parameter to reverse in the renderer and rendering the wireframe of the object in solid black. This draws the back-facing polygons of the model as solid black lines. Furthermore, if the translations of the vertices is changed, the wireframe can be dilated to appear slightly larger than normal. Alternatively, the back-faces may be solid filled and the vertices translated slightly along their own normals.

Once this has been done, the backface culling parameter is set back to normal and the object is drawn with normal shading and texture information calculated. When the whole scene is drawn with Z-buffering the back facing wireframe or solid black object will lie deeper in the scene than the front

facing object. Since the black outline object is slightly larger than the normally drawn object, the darkened edges of the black object appear to 'peek' out around the edges of the textured model. This gives the impression of darkened edges around the model.

This method was not used for the final application of this dissertation because it would involve rendering all objects twice. While this would not be a problem for small scenes, for the large scene used in this project (and most video game scenes), the memory requirement would be too high.

3.1.2 Image-based edge detection

Image based edge detection involves taking a view of an object and examining its appearance for areas where there are, for example, a difference in brightness or colour to determine if an edge exists. This method can be cheap and easy to apply, especially if done as a post-processing effect, but comes at the disadvantage of sometimes being temporally and spatially incoherent in an interactive application.

For this dissertation, all image processing is done on the GPU as a postprocessing effect. Having rendered the scene as normal to a texture once, this texture is passed on to the GPU for post-processing. During a post-processing pass, only the pixel shader is used. Using the scene texture as input, it may perform some calculation with the pixel values and output a new value which will be the final pixel value for the output. Post-processing is useful because image calculations can be performed which involve every pixel of the scene. Thus, to calculate a pixels lighting and colour value, the values of the surrounding pixels can also be taken into account.

This is useful for the edge detection algorithms in sections 3.1.2 and 3.1.2 as well as the painterly rendering algorithm described in section 3.2.2.

Difference of Gaussians / Gaussian blur

To understand how this method of edge detection works, it is important to understand the concept of Gaussian blur.

In mathematics, a Guassian function is a mathematical function of the form

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}} \tag{3.1}$$

They describe the graph of a bell curve, which has a peak at a certain point and then drops off gradually in either direction. These functions are commonly used in statistics and signal processing, where they are used to describe normal distributions, heat equations and diffusion equations.

In image processing, Gaussian functions are used for image blurring. This is done by convolving the image with a two-dimensional Gaussian function, and can be used to reduce noise or lessen image detail. It is used to calculate the transformation of each pixel in the image.

Equation 3.1 shows the general formula for a Gaussian function in one dimension, however for image processing, where blurring must be done in both the horizontal and vertical directions, the correct function to use is one which is the product of two Gaussian functions:

$$f(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3.2}$$

where σ is the standard deviation, and x and y are the distances from the origin along the horizontal and vertical axes respectively. Values from this formula are used to create a convolution matrix which is applied to the image. For each pixel, a value is calculated as the weighted average of all other pixels within a certain neighborhood. The original pixel has the highest Gaussian value, and thus, has the highest weight, while surrounding pixels have lower weights relative to their distance from the center. As the distance from the center pixel grows, the closer the contribution of that pixel approaches zero. Thus, to save computational power, pixels a distance of 3σ are considered to be effectively zero and not calculated.

The effect of Gaussian blur is usually generated by convolving the source image with a kernel of Guassian values. Below is a rough approximation for the Gaussian mask corresponding to $\sigma = 1.0$.

$$M_1 = \frac{1}{273} \begin{vmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{vmatrix}$$

Each pixel is set to the sum of the surrounding pixels weighted by the Gaussian mask. Note that the sum of all the mask values is equal to one. This ensures that the intensity across the image remains constant.

Difference of Gaussian edge detection involves finding the difference between one Gaussian blurred version of an original image from another. The subtraction of the two blurred images preserves edge information which lies between the frequency ranges that are preserved in the two blurred images.

Sobel Operator

The Sobel operator is an image processing technique primarily used in edge detection. It is a differential operator which computes the 2D spacial gradient measurement for an image.

The Sobel operator is used to find the absolute gradient magnitude at a point in a greyscale image. The method uses two 3x3 convolution masks, one for establishing horizontal and the other for establishing vertical gradient; both of which are much smaller than the source image. The values of the masks used are

$$G_x = \begin{vmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{vmatrix}$$

for the x-direction(horizontal) and

$$G_y = \begin{vmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{vmatrix}.$$

for the y-direction(vertical). For each pixel, the masks are applied for it and the pixels adjacent to it. Each of the two kernels can be convoluted with the source image separately and then combined to determine the absolute gradient for that pixel. The final output will be an image showing where the edges exist.

Figure 3.3: A picture of an palm tree and the same image when convoluted with the Sobel operator. Copyright 2008 RoboRealm.

Laplacian

Similar to the Sobel filter is the Laplacian filter, which also acts as a convolution mask. Unlike the Sobel filter, which approximates the gradient of an image, the Laplacian filter approximates the second derivative. The Laplacian mask is a single 5x5 mask which approximates the second derivative in both the horizontal and vertical directions. An example can be seen below.

$$LaplacianMask = \begin{vmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & +24 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{vmatrix}.$$

The Laplacian filter is simpler to implement than the Sobel filter as it uses one filter instead of two, however it is much more sensitive to noise. Thus, though it was considered, it was not implemented for this dissertation.

3.2 NPR methods

3.2.1 Cel-Shading

Cel-shading is a type of lighting model in non-photorealistic rendering. It's primary use is to make computer graphics look like a hand-drawn animation such as a cartoon or comic book. It is prominently used in real time and interactive applications, in particular as the most widely used form of NPR in video games. The name cel-shading comes from when classic cartoon animation was done by drawing on and colouring acetate *cels*.

The process of cel shading first requires the conventional lighting and texture calculations of each pixel for a scene. The lighting values are then compared to a 1D texture containing only a certain number of discrete lighting levels. If the light value is within a certain threshold, its lighting level is set to a discrete light level in the texture. This leads to the lighting in a cel-shaded scene to appear more like blocks of colour rather than a smooth transition from darkness to light.

Cel-shading usually is used with some kind of edge-detection to accentuate the effect.

3.2.2 Kuwahara

A key non-photorealistic technique used in this dissertation is the use of the Kuwahara filter. It was used in the painterly rendering algorithm of Redmond and Dingliana [3].

The easiest way to provide abstraction in a scene is to remove unimportant detail from an object. This can be done using a smoothing/blurring filter such as a median filter.

Though its primary purpose is to clean up noisy images, the Kuwahara filter (a technique of computer vision) can be used to create painterly abstraction effects.[3] Its benefit to NPR is its ability to smooth out edges in a scene without distorting the position or sharpness of the edges. Thus it is an *edge-preserving* filter.

For each pixel, the Kuwahara filter measures the variance of a number of subregions of pixels surrounding that pixel (see figure 3.4). The variance of a region, in this case, is taken to be the difference between the mean of the squares of each pixels luminance (the amount of light from 0.0 to 1.0 in grayscale) and the sum of the squares of the means of each pixels luminance. The mean of whichever subregion has the lowest variance is taken as the value for that pixel.

3.2.3 Paper textures

As stated in section 2.1.2, Bousseau et al. used a system that incorporated a number of stages to create their painterly effect. These stages included the use of texture to simulate pigment dispersion, turbulence flow and the feel of the paper. For these stages grey-level textures were applied, using Gaussian noises, scanned pieces of paper and perlin noise textures.

Perlin noise is a type of procedurally generated texture used to add realism in computer graphics. Its function is pseudo random but its visual details (alternating regions of dark or light pixels) are of roughly the same size allowing it to be scaled and used in a wide variety of mathematical functions

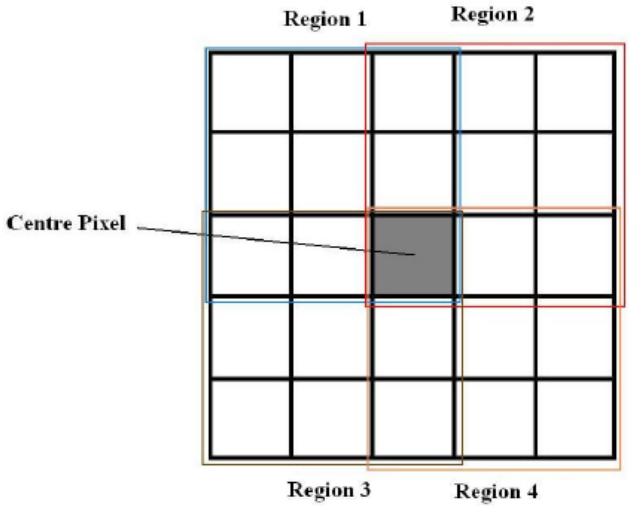

Figure 3.4: Example of a 5x5 filter with 3x3 sampling regions. The center pixels value is set to the mean of the region with the lowest variance. Image courtesy of Redmond and Dingliana.[3]

(see figure 3.5). An artificially generated texture can be made to look more realistic by combining it with Perlin noise; thereby mimicking the natural random appearance of textures in nature.

Figure 3.5: An example of Perlin noise. Note how each region of light is roughly the same size as the others. Copyright 2006-2009 Filter Forge Inc.

In NPR, it can be used to simulate the different levels of say, paint on a canvas, by appearing similar to the random contours of a page or flow of paint.

For this project, a perlin noise texture was considered. It was to be applied to objects in the background scene as their colour textures were being applied. However, due to the interactive implementation the project would adopt, it proved difficult to implement in a natural looking way.

Chapter 4

Design & Implementation

The application presented in this dissertation implements a number of visual effects on a scene in order to create the desired abstraction effects. This chapter explains how the algorithm was originally conceived and how the application was implemented. The concept of the algorithm as it was originally designed is discussed. A step by step guide to how the demo application works is provided.

4.1 Design

From the beginning of the design of this system. Four primary goals were considered. The final implementation should

1. use adaptive abstraction to direct user focus,
2. generate a scene using a painterly NPR system,
3. have a stable framerate for a number of resolutions,
4. be deployable on Xbox360 an PC.

Early on in development, a clear idea of how to achieve these objectives was formed. The application itself takes the form of an interactive scene in Trinity College where the user controls a spaceship which can navigate around the scene. Controls are added to allow the player to change the abstraction levels of certain objects.

Adaptive Abstraction to Direct User Focus For the purpose of this dissertation, abstraction was implemented as a method of rendering certain areas of the scene based on certain criteria. These criteria are

- how important (salient) an object in a scene is and
- how far away from an object the player is.

This is implemented by drawing a depth image of the scene to represent object proximity. This depth image is then used to gauge how distant a certain pixel is from the viewport. This information is then used in a post-processing pass to evaluate what level of abstraction to use in that region of the screen. Salient features of the scene are also defined as such before the depth pass and set with a parameter to define their level of abstraction before the post-processing pass.

Abstraction of an object can involve Gaussian blur, edge detection, and painterly rendering using a Kuwahara filter.

Painterly NPR In this algorithm, painterly rendering is implemented using the Kuwahara edge-preserving filter. This method removes detail across the image while still maintaining the shape boundaries of an object. It can, therefore, provide a roughly uniform level of abstraction across an image in a painterly style.

Some parts of the scene will be rendered differently to the rest, depending on a certain objects depth and importance. While Gaussian blur will be used for distant objects as they would be perceived by the user to be unimportant, and simple texture mapping is used for nearby objects, painterly rendering is used for regions that lie in between. By using painterly rendering for abstraction, visual information is reduced while still maintaining important details like edges. Thus the viewer can discern a certain amount of visual information but in a way that doesn't distract from other, more salient objects in the scene.

Framerate In order to maintain the experience on interactivity in video games and other interactive applications, a stable, relatively high frame rate is necessary. The algorithm would be designed to run on a number of different resolutions for both the Xbox360 and PC. Furthermore, effort would be been made to streamline the implementation on the CPU and the GPU so that the time taken for each complete cycle could be as low as possible.

Xbox360/PC compatibility One of the benefits of XNA is that a program written for one of XNA's supported platforms (Xbox360, PC or Zune) can automatically be compiled by XNA into a version which runs on another platform using the same source code. In this manner, a version would be designed to run on both the Xbox360 and PC simultaneously.

4.2 Implementation

To describe the process through which the scene is rendered, it is a good idea to recount the stages the application must go through when it starts.

4.2.1 Initialization & Object Creation

At this stage the viewing parameters of the scene are created. These include the view and projection matrices, the near and far clipping planes and the setup of the 'ChaseCamera' object, which controls

the position and movement of the viewport. It is based on the chase camera design created by Microsoft [1]. It simulates a third person perspective camera focusing on a player-controlled ship. The camera lags behind when the ship is turned, moving back into position based on a simulation of spring-like physics.

Furthermore, a 'Cursor' object is created which defines the movement and effect of a movable cursor onscreen, which is used to interact with objects in the scene. It is based on a Cursor class created by Microsoft[2].

It is important to note for future reference that for the remainder of the program two projection matrices exist. One is for the main rendering passes, while the other is for the depth pass. The reason for this is that the near and far clipping planes for the projection matrix in the depth pass must be relatively small in order for the depth calculations to have a meaningful effect. If the range is too large then the difference between two objects say, 10 meters away and 15 meters away become insignificant.

Next, a 3D scene is loaded into the application. The default scene is Trinity College with a number of objects scattered around. The objects themselves are set in a custom class which stores their position, orientation, model meshes and whether or not the object is 'interesting' enough to be rendered with or without abstraction on. By implementing models using this 'CustomItem' class, the importance, position and size of a large number of models can be stored in a single 'List' of these objects. By simply adding new CustomItem objects to this list, new objects can be added to the scene in a single line of code:

```
1  targetObjects.Add(
2          new CustomItem(
3                  Content.Load<Model>("saucer"),
4                          new Vector3(60f, 10f, -150f),
5                                  Matrix.CreateRotationY(Math.PI),
6                                          true,
7                                                  0.005f));
```

Here, a new object is created and added to the list of objects to be drawn using the 'saucer' model, at the geometric location (60,10,-150), with a net rotation of π radians about the y-axis, an importance value set to true and scaled to 0.005 of its original size.

4.2.2 Depth/Normal pass

The first draw pass that occurs is the pass that draws the depth and normal information to a texture. First, a render target is set to receive the texture that the information will be drawn to. A render target is simply a buffer where the GPU draws pixel information. The default render target is the back buffer, i.e. where the next frame is drawn, but other render targets can be drawn to and the output of these draws can be extracted as a texture.

All models already have shaders and textures associated with them, but for the depth pass, these

[1] Found on the XNA Creators Club website - http://creators.xna.com/en-US/sample/chasecamera

[2] Which can also be found at XNA Creators Club at this address - XNA http://creators.xna.com/en-US/sample/picking

are not necessary as the position and normal of each vertex are sufficient. For this reason, when the draw function is called in XNA for the depth pass, the shader used for all models is temporarily changed to the shader used for the depth calculations. This shader ('ABSpost') contains all the functions necessary for a depth/normal pass and all the functions for the image post-processing (which wont be necessary until a later pass).

The program goes through every ModelMeshPart in every ModelMesh for every CustomItem in the list defined in section 4.2.1 and finds the local 'Effect' associated with that part. It then adds the default effect to a list of effect objects and sets the effect in the model object to the depth shader effect. The GPU must be passed the World, View and Projection matrices for the object, and the value set in the CustomItem class indicating if that item should be considered an interesting feature or not. The World matrix is given by converting the Position parameter of the CustomItem class to a 4x4 translation matrix, while the View matrix is given by the Camera object. The Projection matrix is the special depth pass matrix defined earlier in section 4.2.1. The object is then drawn with this shader, storing depth and normal information to a texture. Once this texture has been created, the application goes through the ModelMeshParts again, assigning them their original effects which had been stored in the list.

Below is the HLSL code corresponding to the vertex and pixel shaders in the depth/normal pass of the application.

```
1  NormalDepthVertexShaderOutput NormalDepthVertexShader(VertexShaderInput input)
2  {
3      NormalDepthVertexShaderOutput output;
4
5      // Apply camera matrices to the input position to accurately place object in scene
           .
6      output.Position = mul(mul(mul(input.Position, World), View), Projection);
7
8      float3 worldNormal = mul(input.Normal, World);
9
10     // The output color RGB values hold the normal, scaled to fit into a 0 to 1 range.
11     output.Color.rgb = (worldNormal + 1) / 2;
12
13     // The output alpha holds the depth, scaled to fit into a 0 to 1 range. If the
           object
14     // is set to "interesting", this is automatically set to 0.
15     if(specialItem)
16     output.Color.a = 0.0f;
17     else
18     output.Color.a = output.Position.z/ output.Position.w;
19
20     return output;
21 }
22
23 // Simple pixel shader for rendering the normal and depth information.
24 float4 NormalDepthPixelShader(float4 color : COLOR0) : COLOR0
25 {
```

```
26      return color;
27  }
```

In the vertex shader, each vertex is taken in by the shader and projected into world space by multiplying it be the World-View-Projection matrix so that its position relative to the viewport is correct. The value of the normal of each vertex is then found (relative to the vertex position in world space). The x, y and z values are then scaled to fit between zero and one and stored as the RGB channel of the output colour.

Next, depending on the boolean value set previously indicating the importance of the item, the depth parameter is set.

If the value is set to false, the value for the depth is set to the z-buffer value. This is the distance from the viewport to a vertex. Once it is found, the depth value of each pixel is normalized to between zero and one by dividing it by the greatest depth value visible, which is stored in the w value of the position semantic.

If, however, the importance value is set to true, the depth value of the vertex is set to zero. This is because zero depth corresponds to values that are close to the viewport. When the depth information is used in later passes, objects that are close will be rendered without any modifications, whereas more distant objects will be rendered with greater and greater levels of abstraction. The closest objects will have depth values close to zero and be rendered fully detailed with no abstraction. Thus, by setting the depth values of important but distant objects to zero, they are also rendered in full detail.

Once the depth is found, it is written to the alpha channel of the output colour. In the pixel shader, the input colour, which now contains the normal and depth information, is simply passed on and written to the texture.

When this information is returned to the CPU as a render target, the texture is then extracted and saved for the postprocessing pass.

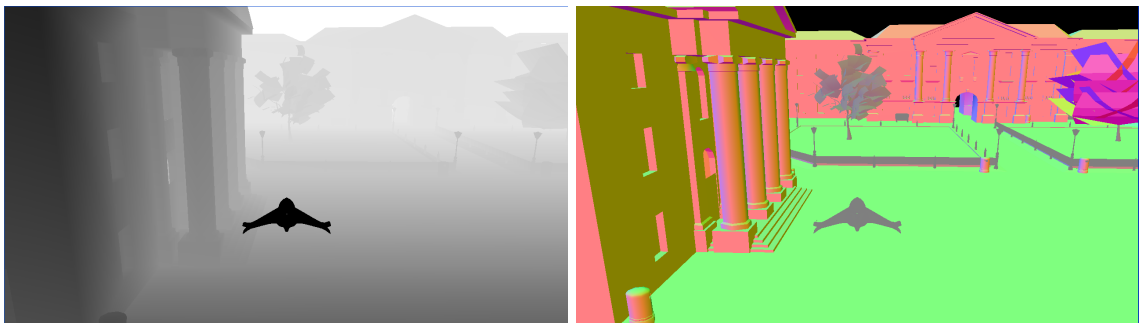

Figure 4.1: Represented graphically, the depth and normal information of the scene respectively.

4.2.3 Cel-shading/Texture shading pass

In the next rendering pass, all models are rendered with the shaders and textures that are associated with them to another render target. Each model uses effects that are associated with their own meshes

to apply diffuse lighting and textures to the surfaces of the objects.

However, for the environment models of Trinity College a more sophisticated method was used. In an effort to boost the non-photorealistic nature of the background to make the other models stand out more, the lighting model and textures were changed to incorporate several methods from other NPR papers.

For example, the cel-shading algorithm of Lake et al.[7] was used as the lighting model for the background. For this, only a small number of lighting values were possible for any given part of the scene depending on the location of the light source. However, instead of storing the range of possible lighting values in a 1-D texture as described by Lake et al., an array of threshold values are saved in the shader. When the local lighting value is calculated in the shader, it is thresholded to one of the lightness values in the array, allowing only a discrete number of light levels to be visible.

```
1  output.LightAmount = dot(worldNormal, normalize(LightDirection));
```

Here, in the vertex shader, the amount of light is found by finding the dot product of the normal of the vertex and the direction of the light source. This value is stored in the output structure of the vertex shader to be used in the pixel shader.

```
1  float4 color = tex2D(DiffuseSampler, psIn.TextureCoordinate);
2
3    float light;
4
5    if (psIn.LightAmount > ToonThresholds[0])
6        light = ToonBrightnessLevels[0];
7    else if (psIn.LightAmount > ToonThresholds[1])
8        light = ToonBrightnessLevels[1];
9    else
10       light = ToonBrightnessLevels[2];
11
12   color.rgb *= light;
13
14   return color;
```

Here, the value for light striking a particular vertex, having been calculated previously, is compared with the three possible values for shading (the 'ToonBrightnessLevels'). Whichever threshold value the light amount corresponds to dictates the light level at that point. This light level value is then multiplied by the colour value from the objects texture to create a shaded, coloured, object. An example screen can be seen in figure 4.2.

Once the shading and texturing pass is completed, the pixel colour information is saved to a render target and a texture is extracted by the CPU. This texture, along with the depth/normal texture, are passed as parameters into the post-processing function.

4.2.4 Image post-processing

The depth/normal and cel-shaded textures are then passed to the GPU for the final rendering pass of post-processing. Using this information, the postprocessing effect alters the cel-shader texture and

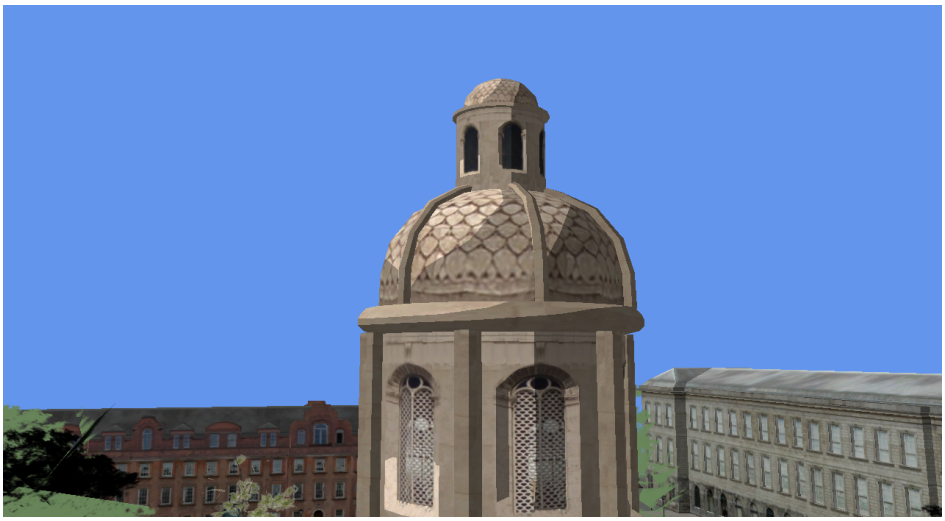

Figure 4.2: Cel-shading in the Trinity College scene. Note how the lighting levels appear as solid 'blocks'.

draws an image to the screen; the final, abstracted image. Image post-processing takes place in the same 'ABSpost.fx' HLSL shader file as the shader functions for the depth/normal pass.

```
1     // Activate the appropriate effect technique.
2     KEffect.CurrentTechnique =
3                                 KEffect.Techniques["PostProcess"];
4
5     // Draw a fullscreen sprite to apply the postprocessing effect.
6     spriteBatch.Begin(SpriteBlendMode.None,
7                       SpriteSortMode.Immediate,
8                       SaveStateMode.None);
9
10    KEffect.Begin();
11    KEffect.CurrentTechnique.Passes[0].Begin();
12
13    spriteBatch.Draw(image.GetTexture(), Vector2.Zero, Color.White);
14
15    spriteBatch.End();
16
17    KEffect.CurrentTechnique.Passes[0].End();
18    KEffect.End();
```

Here, the 'spriteBatch' object is used by the shader to draw its output as a fullscreen sprite onto the screen. The post-processing effect takes place as a single pixel shader, with no need for a vertex shader as the only information needed for calculations are provided by textures.

To perform the image processing and abstraction on the GPU, a number of steps are followed.

Depth Separation To perform the abstraction, the depth information from the depth/normal texture is extracted for each pixel and separated into four distinct regions (as can be seen in figure 4.3). Depending on its value from zero to one (close to distant), a pixel will be said to represent a position in one of these four pixels. Each of these pixels will then have a different image processing function applied to it. For depth d, the functions applied are:

- $1.0 \geq d \geq 0.9$ - Very distant objects; Gaussian blur.
- $0.9 > d \geq 0.8$ - Distant objects; Kuwahara blur with 5x5 sampling regions.
- $0.8 > d \geq 0.6$ - Close objects; Kuwahara blur with 3x3 sampling regions.
- $0.6 > d \geq 0.0$ - Very close objects; No changes.

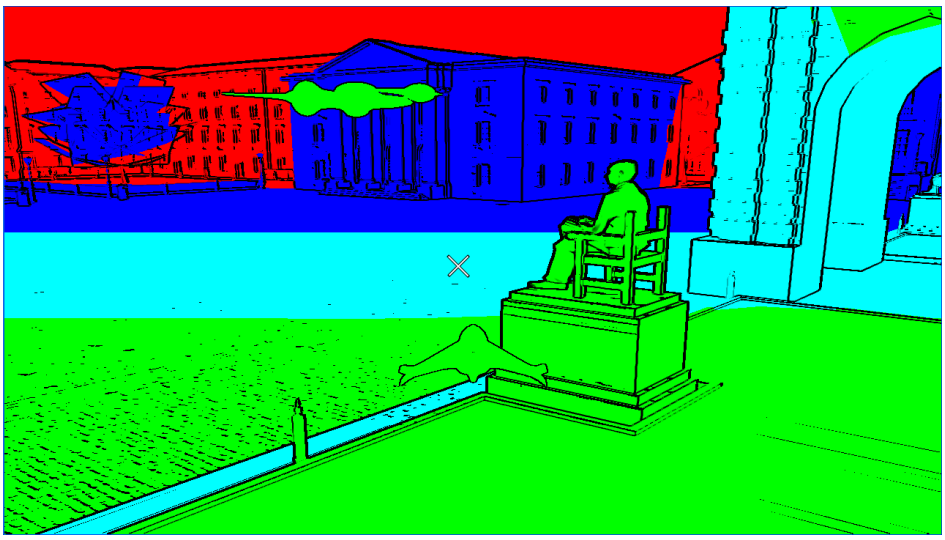

Figure 4.3: Scene is separated into 4 different regions of depth. Note how the important object in the distance (the flying saucer) belongs to the same region as objects that are very close. In order, the four regions are rendered as; red - Gaussian Blur, dark blue - Kuwahara with 5x5 sampling regions, light blue - Kuwahara with 3x3 sampling regions, green - no change.

Gaussian Blur As mentioned in section 3.1.2, Gaussian blur is accomplished by convolving a Gaussian mask with each pixel in an image to calculate its new value. Using the formula from equation 3.2, a Guassian mask was created with a σ value of 1. The values found for this mask were approximately;

$$Mask = \begin{vmatrix} 0.003663 & 0.014652 & 0.025641 & 0.014652 & 0.003663 \\ 0.014652 & 0.058608 & 0.0952381 & 0.058608 & 0.014652 \\ 0.025641 & 0.0952381 & 0.15018315 & 0.0952381 & 0.025641 \\ 0.014652 & 0.058608 & 0.0952381 & 0.058608 & 0.014652 \\ 0.003663 & 0.014652 & 0.025641 & 0.014652 & 0.003663 \end{vmatrix}$$

For each of these positions relative to the pixel being rendered, the pixel is found, multiplied by its corresponding value, and added to the total. For example, in the case of the top left pixel, which, relative to the object pixel is 2 up in the vertical axis and 2 to the left on the horizontal axis, the function would calculate;

```
1 val = tex2D(texSampler, uv + float2(-2/screenRes.x, -2/screenRes.y));
2 sigone += 0.003663*val;
```

where texSampler is the texture received from the previous pass, uv is the texture coordinate of the current pixel being calculated and 1/screenRes.x is the width of one pixel and 1/screenRes.y is the height. When all these values are added up, they give the value of the final, blurred pixel.

Kuwahara As stated in section 3.2.2, the Kuwahara method involves isolating a number of regions of pixels around the target pixel, finding the variance of these regions, and setting the target pixels value to the mean of that region. This method is adapted from Redmond and Dingliana[3].

Two different levels of Kuwahara filtering exist in this project. Depending on the depth of the pixel, it can either be calculated using 5x5 pixel sampling regions or 3x3 pixel sampling regions. Figure 4.4 shows the comparison between these region sizes alongside an unaltered scene. As can be seen in figure 3.4, a 3x3 sampling region corresponds to a 5x5 grid of pixels necessary to calculate each pixels value. Similarly, a 5x5 sampling region will correspond to a 9x9 grid. For the 3x3 sampling case, 25 texture calls need to be made to calculate the colour output for one pixel, whereas in the 5x5 sampling case, 81 calls are necessary. As is expected, this can put a significant strain of the speed of the GPU to make so much use of this already slow function. As such, the Kuwahara algorithm is the slowest part of this rendering cycle, and optimizations are necessary to keep the application moving smoothly in real time. For example, the range of depth values for which 5x5 Kuwahara sampling can take place ($0.9 > d \geq 0.8$) is kept as low as possible to reduce the number of pixels rendered this way while still producing the desired visual effect.

Dynamic Branching In earlier versions of HLSL, it was impossible for code to branch dynamically, i.e. to calculate a variable inside the shader and use that variable to evaluate an *if/else* statement. In newer versions, including the one used for this dissertation (Vertex/Pixel Shader model 3.0), it is possible, but not as straightforward as on a CPU.

For example, if there were a piece of code with two possible branches, say an *if/else* statement, a CPU would evaluate the *if* condition of the branch and, depending on whether the statement was true or not, follow one path of code. However, on the GPU in HLSL, due to latency if one branch takes longer than another, *both* branches are evaluated and once completed the GPU evaluates the if condition. It then selects the result of the code path that corresponds to the correct *if* condition. Even if both paths are quite slow and computationally expensive, both must be followed.

On the Xbox360, there is a way to speed things up.

```
1 #ifdef XBOX
2 [branch]
3 #endif
```

```
4       if(depth.a >= 0.9)
5       {
6         result = Gaussian(uv);
7       }
8       else
9       {
10        result = runKuw(uv,5.0);
11      }
```

The '[branch]' attribute, when placed before a flow control statement such as *if*, will force the GPU to execute the branch dynamically if it can (contrary to '[flatten]' which forces the GPU *not* to branch dynamically). Unfortunately, this method is not available on the PC.

Object-Based Edge Detection As stated in section 3.1.1, object-based edge detection can be done by examining the normal and depth data visible by the viewport. This information is stored in the depth texture saved on the previous pass.

```
1
2         float4 n1 = tex2D(depthSampler, uv + float2(-1, -1) * edgeOffset);
3         float4 n2 = tex2D(depthSampler, uv + float2( 1,  1) * edgeOffset);
4         float4 n3 = tex2D(depthSampler, uv + float2(-1,  1) * edgeOffset);
5         float4 n4 = tex2D(depthSampler, uv + float2( 1, -1) * edgeOffset);
6
7         float4 diagonalDelta = abs(n1 - n2) + abs(n3 - n4);
8
9         float normalDelta = dot(diagonalDelta.xyz, 1);
10        float depthDelta = diagonalDelta.w;
11
12        normalDelta = saturate((normalDelta - NormalThreshold) * NormalSensitivity);
13        depthDelta = saturate((depthDelta - DepthThreshold) * DepthSensitivity);
14
15        float edgeAmount = saturate(normalDelta + depthDelta) * EdgeIntensity;
16
17        color *= (1 - edgeAmount);
```

The above piece of code indicates that the change in normal and depth across a pixel is measured by the pixels diagonally adjacent to the pixel; i.e. the differences between the top left and bottom right pixels and the top right and bottom left pixels (note that the edgeOffset variable is set so that a larger radius can be chosen if necessary). The difference in normals is then expressed as a float by finding the dot product between it and the unit vector (1,1,1). If the delta values do not exceed a certain threshold, then the result of:

```
1   normalDelta = saturate((normalDelta - NormalThreshold) * NormalSensitivity);
2   depthDelta = saturate((depthDelta - DepthThreshold) * DepthSensitivity);
```

will both be zero due to the saturate function (the sensitivity value determines how much a change in normal or depth contributes to an edge). The variable edgeAmount is then used to calculate the "edginess" of the pixel based on the previous values for normalDelta and depthDelta. This gives a

value between 0 and 1 indicating how strong the edge is. This value is then used to change the colour of the pixel, which had been previously calculated, and darken it if necessary.

Figure 4.5 shows the edges based on normals and depth changes. The image shown demonstrates how, although accurate for object edges, edges on textures wont register using this method. For example, the cobblestones on the ground are implemented as a flat plane. Because there is no change in normals and depth across a flat surface, no edges are registered. In the interest of accuracy, a method that detects changes based on the appearance of a texture.

Thus, to attain these edges as well, another, image based solution is also necessary.

Sobel Filter Edge Detection As stated in section 3.1.2, Sobel edge detection involves using 3x3 convolution masks to find the gradient of change of image intensity. Since the Sobel edge detection method is image based, textures that have already been applied to objects have edges extracted from them also. Thus, when applied to the rendered scene texture which has been passed into the GPU for the post-processing pass, edges in the textures themselves, for example, brickwork or windows, are picked up.

```
1  float4 g = (tex2D(texSampler, uv + float2(-1/screenRes.x, -1/screenRes.y))
2           +(2*tex2D(texSampler, uv + float2(0/screenRes.x, -1/screenRes.y)))
3             +tex2D(texSampler, uv + float2(1/screenRes.x, -1/screenRes.y))
4               -tex2D(texSampler, uv + float2(-1/screenRes.x, 1/screenRes.y))
5                 -(2*tex2D(texSampler, uv + float2(0/screenRes.x, 1/screenRes.y)))
6                   -tex2D(texSampler, uv + float2(1/screenRes.x, 1/screenRes.y)))+
7                     (tex2D(texSampler, uv + float2(-1/screenRes.x, 1/screenRes.y))
8                       +(2*tex2D(texSampler, uv + float2(0/screenRes.x, 1/screenRes.y))
                            )
9                         +tex2D(texSampler, uv + float2(1/screenRes.x, 1/screenRes.y))
10                          -tex2D(texSampler, uv + float2(-1/screenRes.x, -1/screenRes.
                                y))
11                            -(2*tex2D(texSampler, uv + float2(0/screenRes.x, -1/
                                  screenRes.y)))
12                              -tex2D(texSampler, uv + float2(-1/screenRes.x, 1/
                                    screenRes.y)));
```

Figure 4.6 shows the edges detected by the Sobel edge detector used in this project. The image on the left shows how the Sobel edge detector finds edges of various different colours and luminances, and sets to black areas where no edges exist. In order to represent to edges as constant dark lines, the image is thresholded. The result is an image where a pixel is either an edge pixel (set to zero luminance or black) or not (set to one luminance or white).

The small details expressed in the textures are picked up, but edges like at the sides of the buildings are harder to detect. This unreliability leads to some 3D edges being missed due to the similarity of colours that can exist where an edge should be.

Combined methods By combining the results of the object-based image detection (which is good at finding edges at the borders of solid objects) and the Sobel edge detection algorithm (which can pick

up the edges on patterns in textures, a more comprehensive edge detection algorithm is implemented.

When the results are multiplied with the result from the colour result the resulting image will be a colour image with the edges superimposed. this is due to the edge information either being zero or one (black of white). Multiplying zero by a colour pixel reduces its value to zero, indicating an edge on the main image. Multiplying one by the colour pixel will return that pixels value, indicating no edge. A sample screenshot can be seen in figure 4.8.

For the main application, edges are drawn only on objects with a depth value of 0.1 or higher. Thus, nearby (like the players ship) and important objects are not drawn with edges, so the detail of the textures are not concealed in any way (allowing the user to inspect it closely).

However, for the benefit of comparison, the system can also be implemented with edges only affecting the *un*-abstracted objects, with the background left abstracted but without edges. This can make detail on the background harder to see, but allows the important objects to stand out even more(the result can be seen in section 5.1.1).

4.2.5 Update Function

The update function serves primarily to handle user input into the application. The application is implemented as a third person camera controlling a small spaceship. The spaceship is controlled by the player via a USB Xbox360 gamepad (which works on both platforms). The orientation of both the spaceship and the camera is controlled by the left analog stick. The right trigger adds thrust, moving the ship directly forwards through the scene. The left directional pad can move around the light source in the sky which affects the cel-shading of the Trinity College models.

Furthermore, the player also has control over a cursor in screen space which can be moved about the screen by the right analog stick. By moving this cursor over an object and pressing the Y button, the user can toggle on and off whether or not abstraction should be applied to that object. For example, the player can highlight one of the added models and change the way it is rendered so that it, like the scene around it, is abstracted.

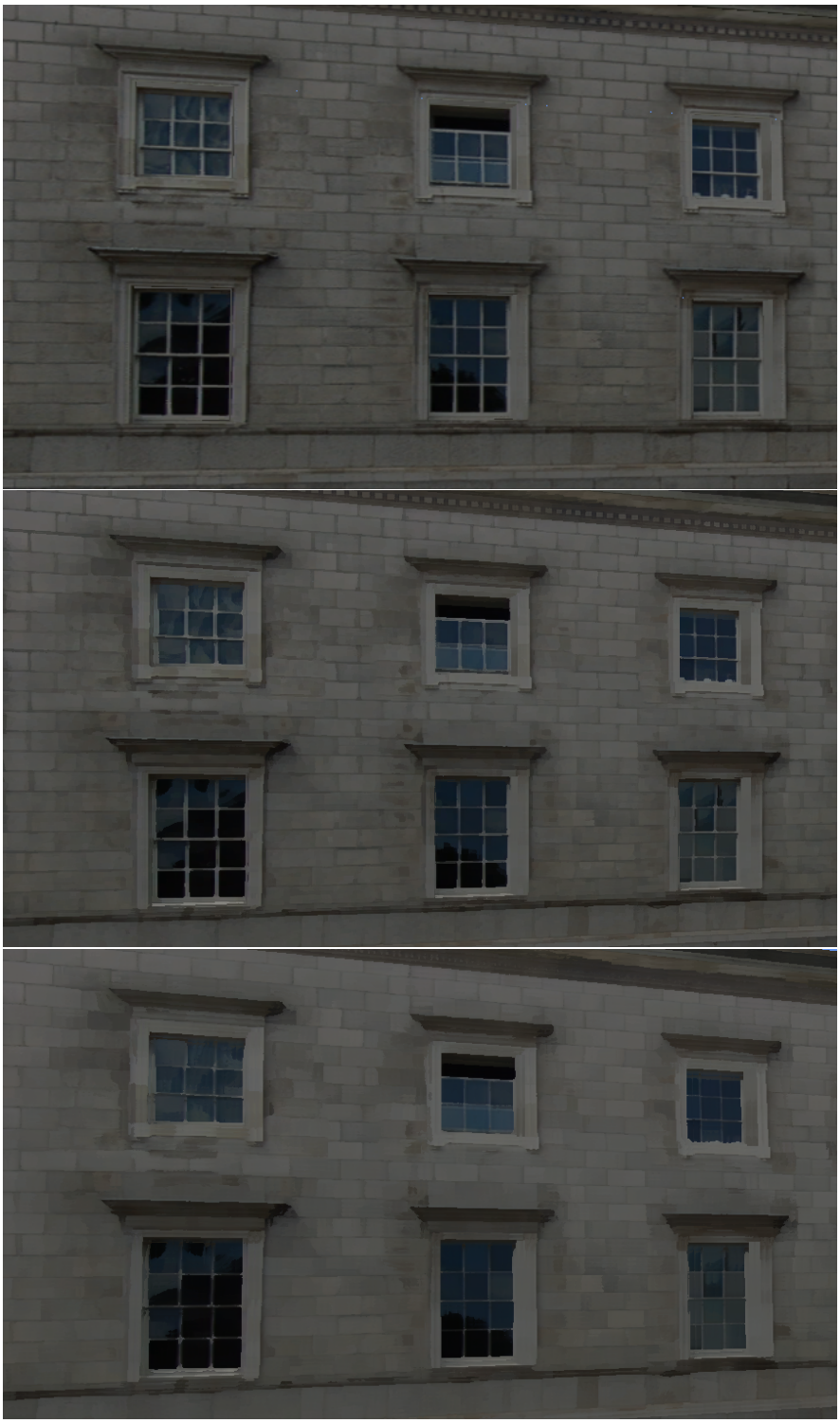

Figure 4.4: Above: An example of the unaltered scene. Middle: Kuwahara filter applied with 3x3 sampling region. Below: Kuwahara filter applied with 5x5 sampling region.

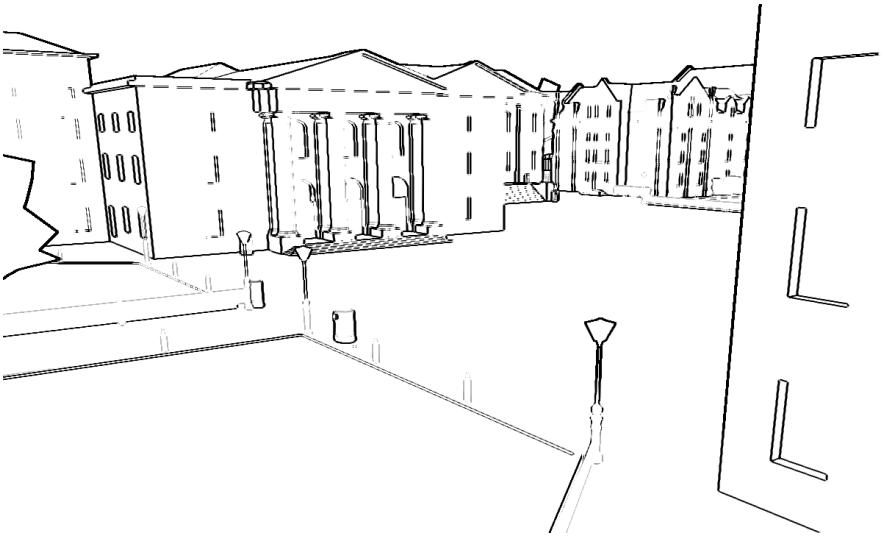

Figure 4.5: Object based edge detection.

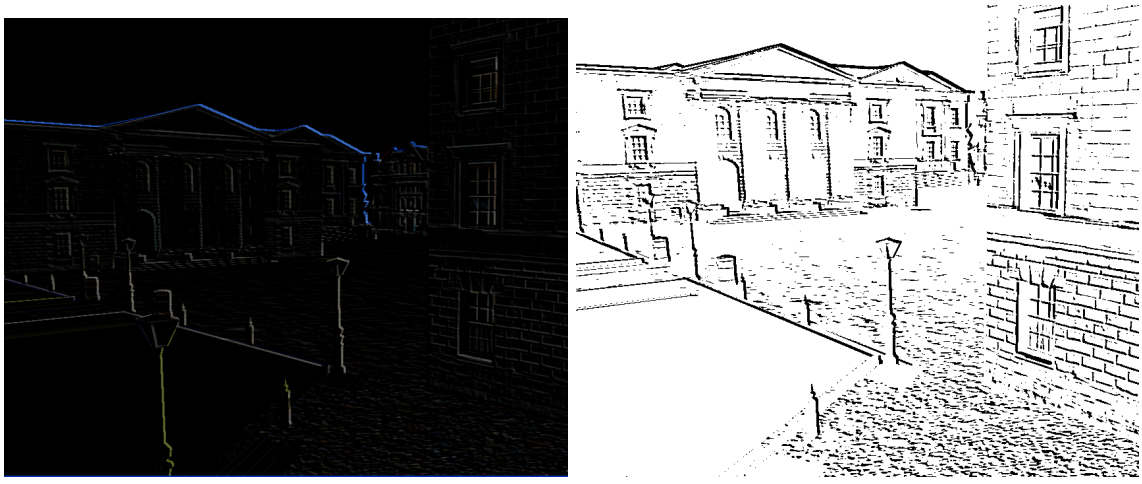

Figure 4.6: Left: Sobel edge detection. Right: After thresholding.

Figure 4.7: Combined Sobel edge detection and object edge detection

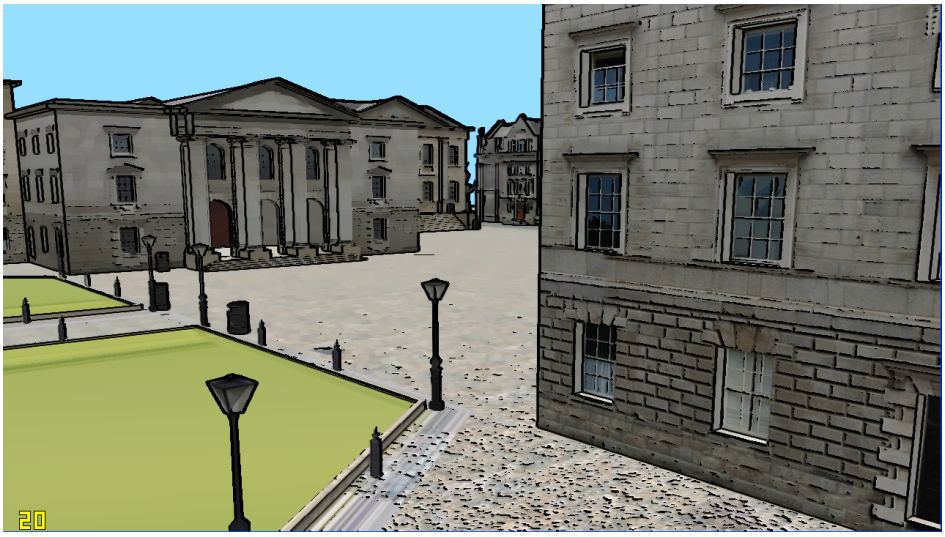

Figure 4.8: Final abstracted image overlaid with edges

Chapter 5

Results & Conclusions

This chapter explains the results of the final implementation of the application. A Youtube video showing some of the features of the application discussed here can be found under the username hanrat77 with the title "Adaptive abstraction with non-photorealistic rendering (NPR)" [1].

5.1 Application

The final application is a model of Trinity College's front square with models of other assorted objects scattered around the scene. The player can move around the screen in a spaceship which is positioned in a third-person video game style view relative to the camera.

Controlling the ship with the gamepad, using the left analog stick for turning and the right trigger for thrust. As stated in Chapter 4, the scene is rendered with the adaptive abstraction algorithm running for the background of the scene (the model of front square). The algorithm doesn't affect the objects in the scene, allowing them to stand out against a less detailed, abstracted background. By hovering a cursor over one of the objects and pressing the Y button, the salience of that object can be changed. When this happens, the abstraction algorithm is applied to the object as well. This allows the user a degree of control over the visual information available in a certain scene.

5.1.1 Appearance

Figure 5.1 shows a sample image from the final application showing abstraction for a range of different objects, rendered at different levels of abstraction. The building on the left is quite close to the viewport so it is rendered with edge detection but no abstraction, just simple texture mapping. As objects get further away, the depth information associated with them will reflect that in the post-processing pass. Thus, objects will achieve greater levels of abstraction as they grow more distant. The Campanile (the large monument visible on the right hand side of the screen) is more distant, and is abstracted with a 5x5 Kuwahara filter. Similarly, the red brick building in the distance has also

[1] The exact URL can be found here - http://www.youtube.com/watch?v=sx8LQI7nJUg

Figure 5.1: Sample image showing how level of abstraction changes with depth.

been abstracted. However, in the case of both of these objects, it is still easy to discern details such as windows due to the darkened edges. These edges appear at points of importance, giving the viewer enough information to see what they are without being a distraction from other objects in the scene.

Figure 5.2 shows the effect of using abstraction on an object to direct user focus. All three images show a car in the Trinity Scene rendered in different ways. The first image shows both the car and the scene rendered normally without any abstraction (for the remainder of this section, this will be referred to as method #1). In the second image, the scene is abstracted but the car is left unabstracted (for the remainder of this section, this will be referred to as method #2). As the image shows, the stylized look of the background allow the sharper details of the car to stand out better. The final image shows how the car looks when abstracted with the background. The car appears blurred and doesn't contrast as heavily with the background as before.

In order to compare this implementation with another style, an alternate implementation of the abstraction algorithm was created. This implementation is similar to the idea used by Redmond and Dingliana[3], where edge detection is performed on the important object instead of the abstracted

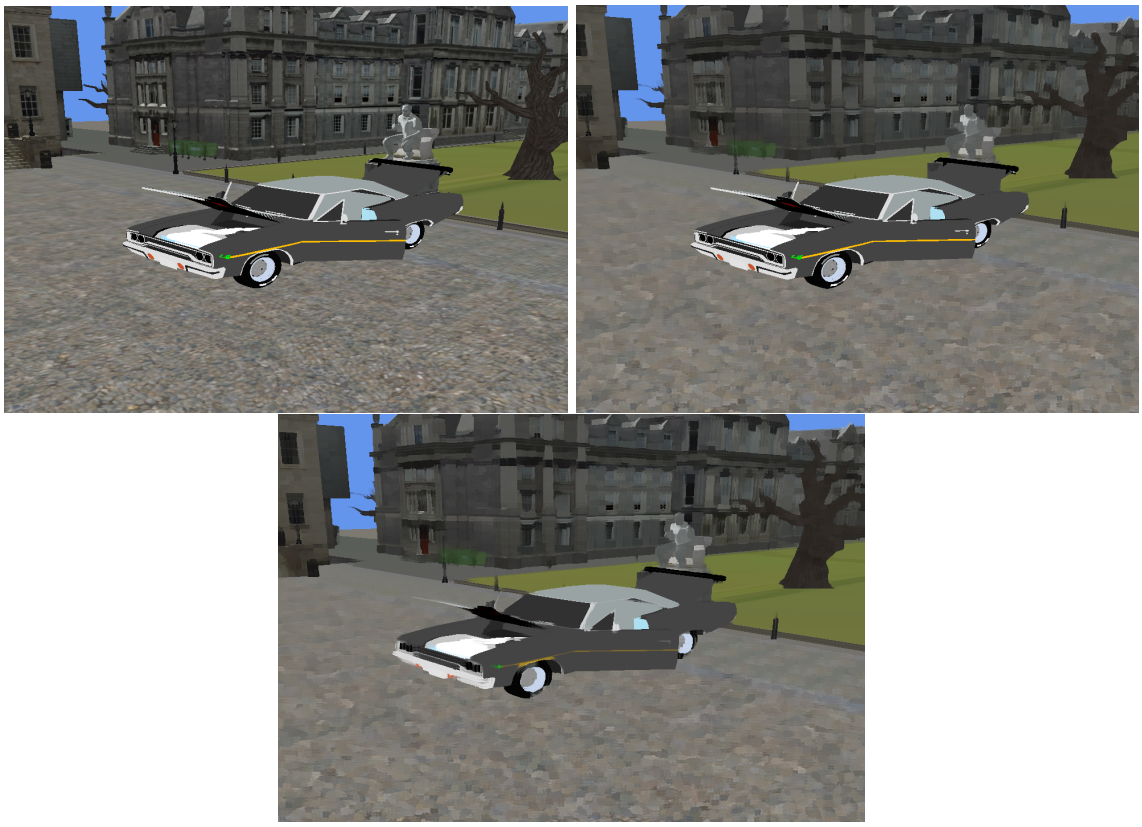

Figure 5.2: Comparison of effects of abstraction on an object. For simplicity, edge detection has been switched off. Top: Normal texturing; Middle: Scene abstracted, car normal; Bottom: Both car and scene abstracted.

background. As figure 5.3 shows, an unabstracted car with edge detection stands out from the scene significantly more than when the car is also abstracted.

Figure 5.4 shows a comparison between rendering without abstraction with the two methods of abstraction described here. The image shows a model of flying saucer in front one of the buildings in the scene. The images show the subtle differences between the different rendering styles, for example, how the red brick building is rendered in various levels of detail behind the saucer model.

Finally, figure 5.5 shows a difference image between normal texturing and the scene rendered using method #2. The difference is at its greatest around the important objects, proving that the adaptive abstraction technique brings a visible difference to the rendering of these objects in relation to the rest of the scene.

5.1.2 Framerate

To test the framerate of the application, the program was run at a number of different resolutions on both the PC and Xbox360 under a number of different sets of conditions. The PC is operating with

Figure 5.3: Alternative implementation (method #2) where edge detection affects the focus object

an Intel Core2 running at 1.86 GHz and 3 GB of RAM. The graphics device it uses is an NVIDIA GeForce 8600 GTS.

The implementation of the Kuwahara algorithm mentioned in section 4.2.4 indicated that it ran at two different sizes depending on the level of abstraction needed. These involved the use of either a 3x3 pixel sampling method or 5x5. As previously stated, for the 5x5 method, four grids of 5x5 sampling regions are necessary per pixel, leading to 81 texture reads. This, even if implemented efficiently, can be a slow process. Thus the framerate tests were implemented both with and without using 5x5 sampling. Though the level of abstraction would be less without it, doing so could allow the application to run faster.

Furthermore, the tests were run both with and without edges to see how using two different edge detection algorithm would affect processor performance. To find out the framerate, a frame rate counting class was modified from one created by Shawn Hargreaves of Microsoft [2] and implemented into the application.

Figure 5.6 shows the graphs of framerate of an implementation against the resolution. The resolutions chosen were 680x480, 800x600, 1024x768 and 1280x720. These four resolutions were chosen as they range from the minimum to maximum resolutions usable by the Xbox360 console.

As the graphs show, the PC framerates show a predictable decline proportional to the increase in resolution. The top left graph of figure 5.6 shows that on the PC, even with the expensive 5x5 Kuwahara sampling, the framerate never dropped below 20 fps. In the case of the Xbox360, however, an interesting feature is apparent. While the curve shows the same profile of decreasing performance as its PC counterpart (though to a lesser degree), it appears that the maximum upper framerate allowable is stuck at 18 fps. It was found in all tests that the implementation used could work no faster. This is further apparent when viewing the top right and bottom right graphs of figures 5.6. The curve appears as a straight line at 18 fps.

Since it seems unlikely that the same piece of hardware could run at the same speed running at

[2] An overview of the class can be found here - http://blogs.msdn.com/shawnhar/archive/2007/06/08/displaying-the-framerate.aspx

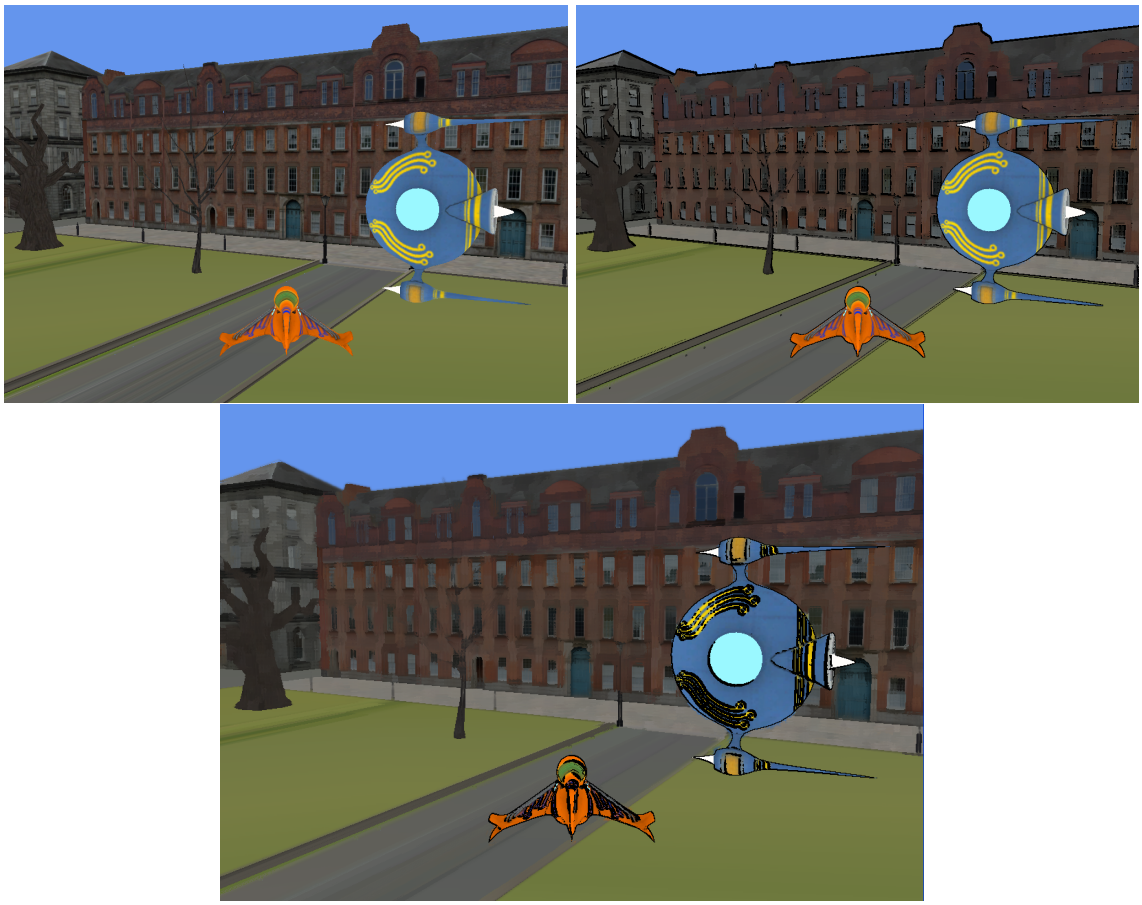

Figure 5.4: Comparison of normal texturing, method #1 of adaptive abstraction and method #2 of adaptive abstraction.

680x480 pixels as 1280x720 pixels, it is reasonable to assume that the framerate of the 640x480 case has been capped by the implementation of code. This is likely due to the branching involved on the shader to determine the abstraction level of each pixel (see section 4.2.4). In HLSL, while the '[branch]' command can improve dynamic branching on the Xbox360 hardware, there is a limit to the amount of speedup possible.

Even using pixel version 3.0 of HLSL, which has support for dynamic branching, branches can still cause slowdowns, especially if the branch paths contain expensive tex2D calls. In this application, the paths lead to functions which run Guassian or painterly image processing, which involve numerous texture reads and mathematical calculations. One possible solution to this problem could be to transfer the branching logic from the CPU back to the GPU. However, this was not feasible for this implementation since the condition that led to branching (information read from a depth texture) would take even longer to calculate per pixel on the CPU than the GPU.

Figure 5.5: A difference image between normal texturing and one of the adaptive abstraction methods. The areas of greatest difference tend to be around the areas where salient objects are located.

5.1.3 Generic Data

When adding new object models to the system, all that is necessary for the user to do is add the model to the list of active members of the 'targetObjects' list as a 'CustemItem' object. All of the depth pass shader code and post-processing is done in the single 'ABSpost.fx' file. By keeping this code in a single universal set of functions in a single file, the graphical implementation is kept as general as possible. This allows any object to be abstracted in the scene without any need to alter its associated textures or shaders.

5.2 Conclusions

The primary goal of this dissertation was to design an algorithm that

1. used adaptive abstraction to direct user focus,
2. generated a scene using a painterly NPR system,
3. had a stable framerate for a number of resolutions,
4. was deployable on Xbox360 an PC.

In these areas, this project has largely succeeded. In some areas, improvements could be made.

If this dissertation were to be repeated, one of the major differences in the approach would be the inclusion of user tests to prove (or disprove) conclusively that a certain object in the scene is more easily registered as visible and important by the user. Unfortunately, there was not enough time to organize experiments of this nature to discern if the effect was improved by abstraction. However,

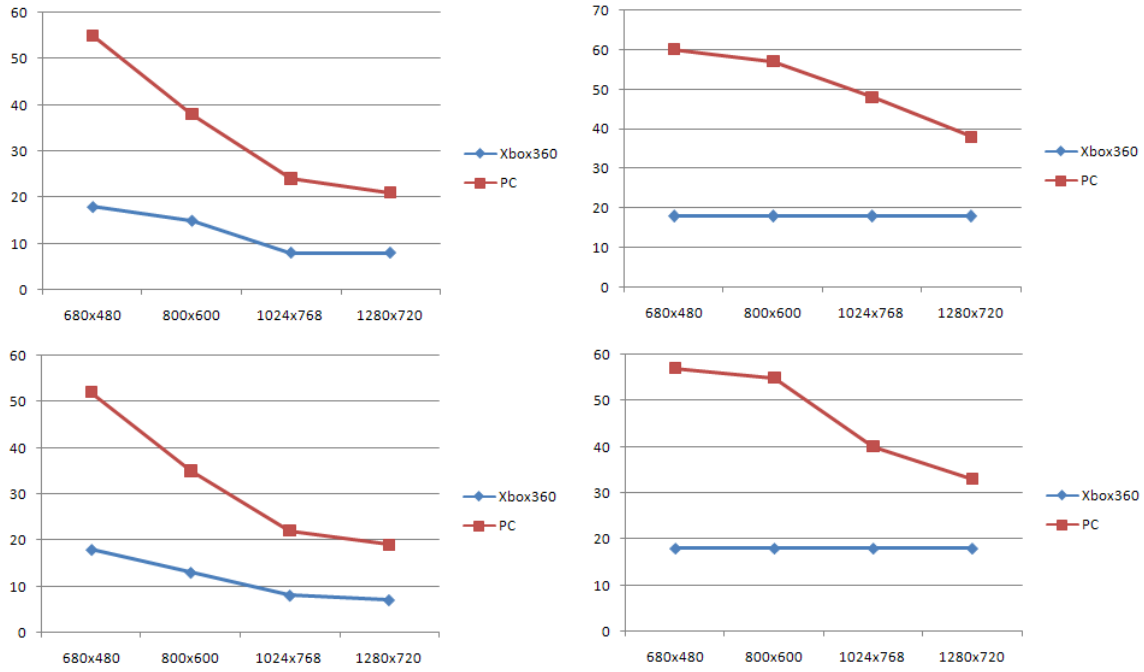

Figure 5.6: Four graphs of framerate against resolution with different rendering criteria. Top Left: No edge detection, with 5x5 sampling. Top Right: No edge detection, without 5x5 sampling. Bottom Left: Edge detection, with 5x5 sampling. Bottom Right: Edge detection, without 5x5 sampling

figure 5.5 does show that the level of visual distinction around the important objects is greater when abstraction is used.

The framerate issues, in particular the 18fps limit on the Xbox360, would also have been addressed more fully. While the same source code can be compiled in XNA for both the PC and Xbox360, a certain program will never run exactly the same way on both hardware setups. The reason for the limit to the cap in framerate on the Xbox360 is likely due to the implementation of the application. The Xbox360 hardware can be utilized better if the application takes advantage of its natural strengths, for example, the way its GPU is directly connected to RAM (allowing faster memory access), and how its multiple cores allow for powerful threading. An application could exploit these advantages and, potentially, work as fast or faster than its PC counterpart. In any case, 18 fps is still real time, and close enough to game-like framerates that it is quite usable.

This dissertation has proven that the algorithm does work on the Xbox360 and PC. These two major games platforms could benefit greatly from games that implemented adaptive abstraction to influence player behavior. Furthermore, the implementation is an example of a real time non photo-realistic rendering in XNA, a relatively new field.

5.3 Future Work

Advancements on Abstraction Algorithm

This project mostly deals with abstraction that adapts based on an objects proximity to the user and importance. However, an object's significance may also be determined by other factors.

If an object is moving, it may be of more interest to the user than if it was static. For example, in a video game, if an enemy is standing still in the distance, it may be of less concern to the player than an enemy which was moving very fast, possibly because it is preparing to attack the player. Thus, velocity or acceleration could also be used as a deciding factor in what level of detail is applied to an object.

A system that uses movement as a basis for abstraction could be implemented quite simply using the algorithm used in this dissertation as a basis. Similar to the way each model in this algorithm is implemented as a complex object with parameters such as position, rotation and importance, other parameters could include velocity and acceleration. Indeed, in the context of video games, such parameters would be essential to create dynamic obstacles and enemies. If the relevant criteria (velocity/acceleration) is simply sent to the Depth/Normal pass as a parameter (similarly to how the importance value was passed in in this dissertation), the value for depth can be adjusted to account for the object's movement (i.e. a faster object could be brought to the users attention by rendering it in more detail).

This leads to a large number of possible advancements to the algorithm about what criteria is allowed to alter abstraction. If a player was surrounded by enemies in a fighting game, for example, an 'intent' parameter could be taken into account. In this way, an enemy which was preparing to launch an attack could be rendered in more detail, giving the player more time to respond to the imminent threat. Such a system could be beneficial to newer, less experienced gamers who could use the extra time to plan their actions.

Pop-In

Throughout the dissertation, effort was made to keep the level of 'pop-in' (when one type of rendering suddenly and noticeably changes to another) to a minimum. Thus, while some pop-in is evident (if one pays close attention, the dividing line between 5x5 Kuwahara and 3x3 Kuwahara is visible on the side of some buildings), the problem is only detrimental if one pays close attention. The existence of the 3x3 Kuwahara region between the most detailed and 5x5 Kuwahara regions allows a smoother transition between levels.

One solution to this problem that would have likely removed pop-in completely would have been to have 'sub-regions' at the transitions between abstraction layers. This would, for example, involve having a small region between the 3x3 and 5x5 Kuwahara levels where the result for that pixel would be interpolated between the result of these two methods. Doing this would smooth out the transition between layers considerably, but would also cause a large drop in framerate due to the need to run *both* algorithms for just one pixel. Since the ability to maintain a real-time framerate was so important,

the idea was dropped, but future work in this area could examine this optimization.

Automated Parameters

The levels of abstraction chosen, and the visual range relative to the viewer in which each level exists have been selected to suit the Trinity College scene. While it is possible that another generic scene could work well with the same parameters, it is not certain, and different parameters may be necessary.

For example, the projection matrix used for the depth/normal rendering pass was made with clipping planes that could roughly fit the entire Trinity College scene into them. For a larger scene, it may be more useful to use larger clipping planes. One possible optimization would involve evaluating the size of the background model as the program initializes and set the clipping planes relative to the size of the model. This would have the dual benefit of ensuring the different abstraction regions are spread evenly across the model, while also ensuring that no part of the model would be unnecessarily clipped when rendered.

Bibliography

[1] Adrien Bousseau, Matt Kaplan, Joëlle Thollot, and François X. Sillion. Interactive watercolor rendering with temporal coherence and abstraction. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pages 141–149, New York, NY, USA, 2006. ACM.

[2] Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. Computer-generated watercolor. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 421–430, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[3] Niall Redmond & John Dingliana. A hybrid technique for creating meaningful abstractions of dynamic 3d scenes in real-time. In *WSCG '08: The 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2008*, pages 477–484, Plzen - Bory, Czech Republic, 2008. WSCG.

[4] Niall Redmond & John Dingliana. Influencing user attention using real-time stylised rendering. In *TPCG 2009: Theory and Practice of Computer Graphics*, Cardiff, UK, 2009. TPCG.

[5] Nick Halper, Mara Mellin, Christoph S. Herrmann, Volker Linneweber, and Thomas Strothotte. Psychology and Non-Photorealistic Rendering: The Beginning of a Beautiful Relationship. In Jürgen Ziegler and Gerd Szwillus, editors, *Mensch & Computer 2003: Interaktion in Bewegung (September 8–10, 2003, Stuttgart)*, pages 277–286, Stuttgart, Leipzig, Wiesbaden, 2003. Teubner Verlag.

[6] Paul Kubelka. New contributions to the optics of intensely light-scattering materials. part ii: Nonhomogeneous layers. *J. Opt. Soc. Am.*, 44(4):330–334, 1954.

[7] Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. Stylized rendering techniques for scalable real-time 3d animation. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 13–20, New York, NY, USA, 2000. ACM.

[8] Hyunjun Lee, Sungtae Kwon, and Seungyong Lee. Real-time pencil rendering. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pages 37–45, New York, NY, USA, 2006. ACM.

[9] Eugene Lei and Chun fa Chang. Real-time rendering of watercolor effects for virtual environments. In *In PCM 04: Proceedings of the 5th Pacific Rim Conference on Multimedia*, pages 474–481, 2004.

[10] Barbara J. Meier. Painterly rendering for animation. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 477–484, New York, NY, USA, 1996. ACM.

[11] Daniel Sperl. Painterly rendering for realtime applications. Master's thesis, Hagenberg College of Information Technology, Austria, 2003.

[12] Tom Van Laerhoven, Jori Liesenborgs, and Frank Van Reeth. Real-time watercolor painting on a distributed paper model. In *CGI '04: Proceedings of the Computer Graphics International*, pages 640–643, Washington, DC, USA, 2004. IEEE Computer Society.

[13] Holger Winnemller, Sven C. Olsen, and Bruce Gooch. Real-time video abstraction. *ACM Trans. Graph*, 25:2006, 2006.