A Stylised Cartoon Renderer For Toon Shading Of 3D Character Models

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
in the
University of Canterbury
by
Jung Shin

University of Canterbury

Abstract

This thesis describes two new techniques for enhancing the rendering quality of cartoon characters in toon-shading applications. The proposed methods can be used to improve the output quality of current cel shaders. The first technique which uses 2D image-based algorithms, enhances the silhouettes of the input geometry and reduces the computer generated artefacts. The silhouettes are found by using the Sobel filter and reconstructed by Bezier curve fitting. The intensity of the reconstructed silhouettes is then modified to create a stylised appearance. In the second technique, a new hair model based on billboarded particles is introduced. This method is found to be particularly useful for generating toon-like specular highlights for hair, which are important in cartoon animations. The whole rendering framework is implemented in C++ using the OpenGL API. OpenGL extensions and GPU programming are used to take the advantage of the functionalities of currently available graphics hardware. The programming of graphics hardware is done using Cg, a high level shader language.

Keywords

Non-Photorealistic Rendering, Cartoon Shading, GPU programming, Hair Rendering, Painterly Rendering

Table of Contents

Chapter 1: Introduction 1

Chapter 2: Motivation for Proposed Research 5

2.1 Assisting artists . . . 5

2.2 Study of silhouettes . . . 5

2.3 Cartoon shading and hair rendering . . . 6

2.4 Hardware acceleration . . . 6

2.5 Our system . . . 6

Chapter 3: Related Work 8

Chapter 4: Reducing Computer Generated Artefacts In Toon Shading 14

4.1 Rendering of G-Buffer . . . 14

4.1.1 Colour . . . 14

4.1.2 Normal and Depth . . . 15

4.1.3 Other information . . . 16

4.2 Rendering of Silhouettes . . . 16

4.2.1 Colour Intensity discontinuity . . . 17

4.2.2 Normal discontinuity: Crease Edge . . . 19

4.2.3 Depth discontinuity . . . 20

4.2.4 Intensity of silhouette . . . 20

4.2.5 Bezier curve fitting: Potrace . . . 23

4.2.6 Restoring of intensity . . . 24

4.3 Filtering and Smoothing . . . 24

4.3.1 Diffuse Lighting . . . 25

4.3.2 Specular Highlight . . . 27

4.3.3 Smoothing of model contour . . . 27

4.4 Merging . . . 28

4.5 Comparison with Traditional Toon Shading . . . 29

Chapter 5: Introducing New Stylised Hair Model 33
5.1 Hair Model . . . 35
5.1.1 Mass-Spring Based Hair Model . . . 35
5.1.2 GPU based Particles . . . 35
5.2 Sorting the Hair Strands . . . 38
5.3 Rendering the Silhouette . . . 41
5.4 Shading . . . 44
5.4.1 Diffuse Lighting . . . 44
5.4.2 Specular Lighting . . . 44
5.5 Hair rendering result . . . 49

Chapter 6: Implementation Aspects 51
6.1 Performance case study: ATI draw buffers . . . 52
6.2 Performance case study: Particle number comparison . . . 55

Chapter 7: Conclusions and Future Work 57
7.1 Artistic level of detail . . . 57
7.1.1 Stylising the rendering using Noise . . . 58
7.1.2 Vector graphics . . . 58
7.1.3 Stylised strokes of silhouettes . . . 58
7.1.4 Skeletons . . . 59
7.2 Improving hair model . . . 59
7.2.1 Particles with depth issues . . . 59
7.2.2 Particles and diffuse shading . . . 59
7.3 Automatic hair animation . . . 59
7.4 Optimisation . . . 60
7.4.1 Level of detail . . . 60
7.4.2 Different hair strand models . . . 60
7.5 Shadow and Transparency . . . 60
7.6 Intuitive interfaces . . . 61
7.6.1 2D feature addition . . . 61
7.6.2 Visual effect in 2D . . . 61

Chapter 8: Acknowledgments 63

Chapter 9: Publications **64**

Appendix A: Cg Shader **65**

A.1 Cg and OpenGL . . . 65
A.2 Data types and functions in Cg shader . . . 65
A.3 2D Filters . . . 66
A.3.1 Edge detection: Sobel filter . . . 67
A.4 Billboard particle . . . 69
A.5 Multiple render target . . . 70

Appendix B: Splines **73**

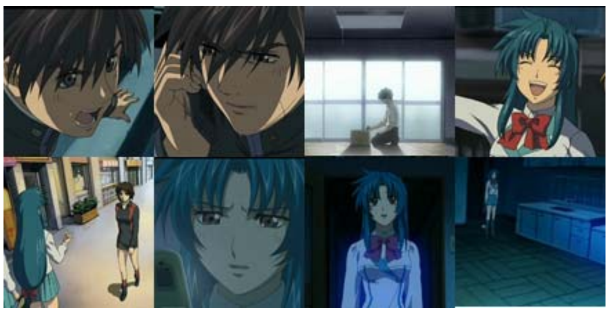
B.1 Bezier curve . . . 73
B.1.1 Quadratic Bezier spline . . . 73
B.1.2 Cubic Bezier spline . . . 74
B.2 Catmull-Rom spline . . . 74
B.2.1 Cubic Hermite spline . . . 74
B.2.2 Cardinal spline . . . 75
B.2.3 Catmull-Rom in the matrix form . . . 75

References **76**

Chapter I

Introduction

Toon shading is a technique to render objects in a cartoon style. Most cartoon shading algorithms researched so far are focused on real-time rendering and have been used for games and some real-time applications. Unfortunately, these shaders have not been used efficiently for producing real cartoons, with an inadequate number of tools to support designers during the drawing process. Cartoon shading has primarily been used for rendering some background scenes and mechanical objects, but not human characters which are one of the most important elements in cartoons (See figure 1.1). Most cartoons are manually drawn and involve labour intensive work.


(a) Hand-drawn characters

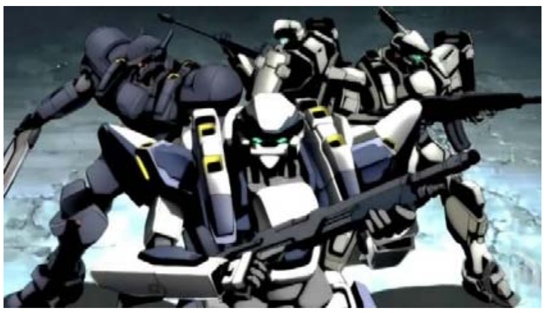

(b) Cartoon shaded robots

Figure 1.1: Japanese cartoon - Full Metal Panic: The Second Raid [18]

For example, the famous cartoon 'The Simpsons' is mostly manually drawn and takes six to eight months to create one episode with about 200 animators [1].

There are many issues in traditional toon shading to be solved to be used in real cartoons more efficiently. One is that the rendered characters are perceived as computer generated due to rendering artefacts. Smooth silhouettes and artistic shading are the main characteristics of a cartoon, and these features are usually hand-drawn. However, in cartoon shading, 3D models are usually polygon based and rendered silhouettes are rasterised and jaggy. The shading area also

[1] http://www.simpsonsfolder.com/production/letter.html

contains rasterised lines, rough shapes along shaded areas etc. Figure 1.2 shows some examples of cartoon shading. Aliasing problems are visible from both shading and silhouettes. Aliasing effect can be reduced using anti-aliasing functions of some latest graphics card, however, users can only apply the filter uniformly.

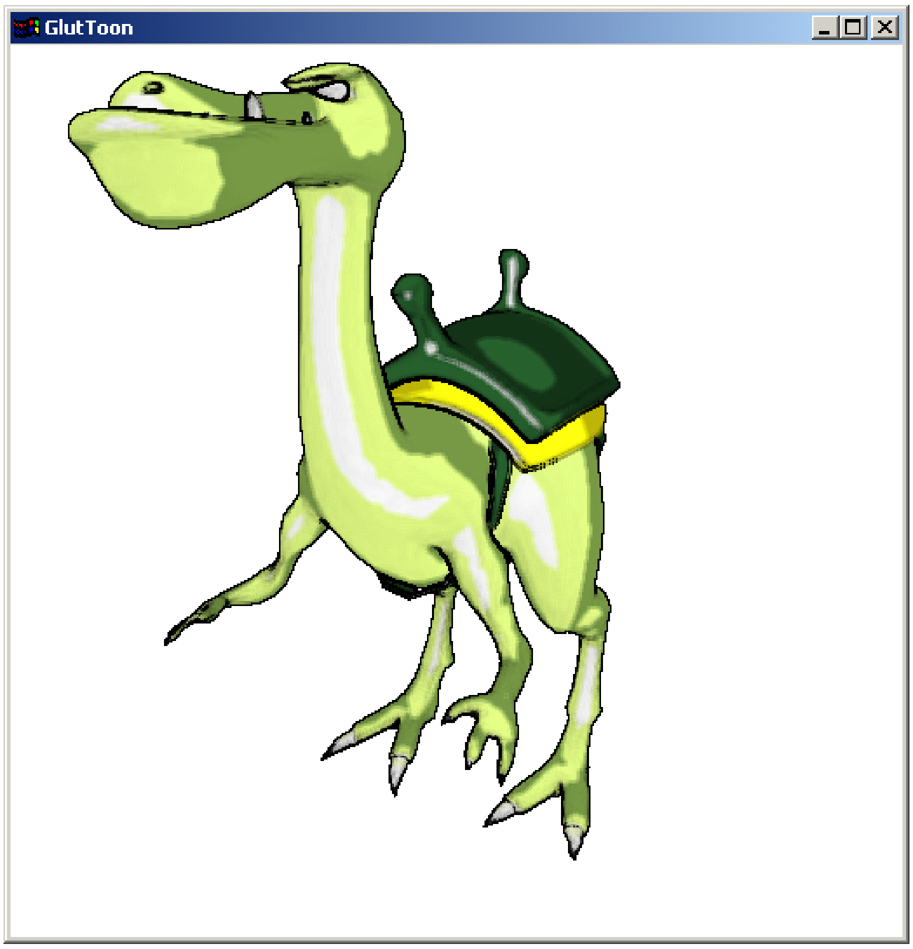
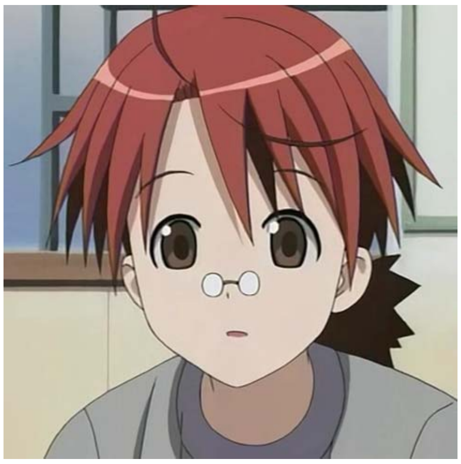
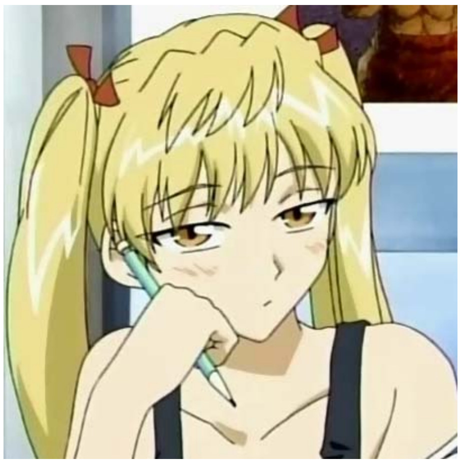


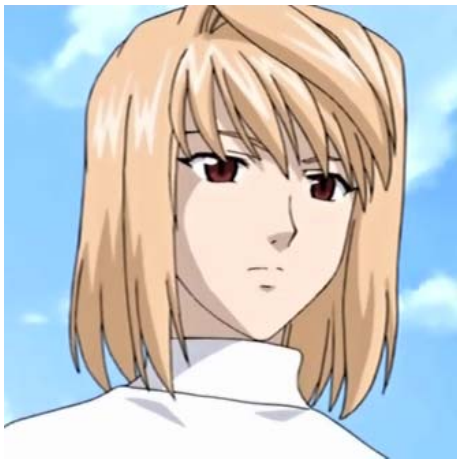
Figure 1.2: Toon shading examples

Therefore, a novel technique is introduced to control and apply the silhouette in this study. Different types of silhouettes and methods to change their intensities are studied and various image processing algorithms are used to minimise the computer generated artefacts and enhance the quality of toon shading. The silhouette coherency is improved through a novel technique which modifies the intensity of silhouettes automatically by utilising the depth information of the input geometry. The hand-drawn appearance of the silhouette is enhanced by sampling the silhouettes and reconstructing them with Bezier curve fitting. For the shading, various image filters are applied to smooth any rendering artefacts. The silhouette information is also utilised to reduce the aliased contour of the rendered images.

Each cartoon has their own styles of drawing which usually involve exaggeration and simplification of features. For example, in Anime, the Japanese version of cartoons, characters have simplified hair strands and stylised zigzag shaped specular highlights on their hair (See figure 1.3). However, exaggeration or simplification cannot be achieved easily with the traditional cartoon shading. To allow exaggeration or simplification, more specific models are needed rather than simple polygons.


(a)


(b)


(c)

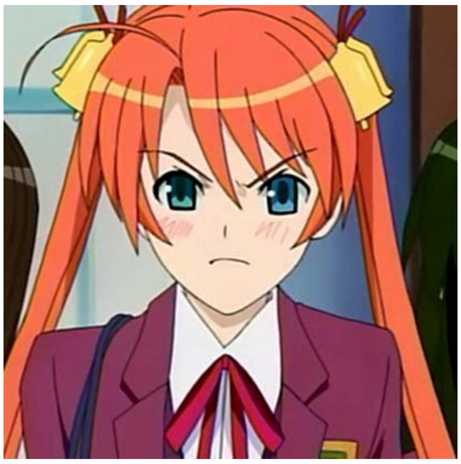

(d)

Figure 1.3: Anime hair examples

In this study, we introduce a new hair model and a rendering technique in Anime style. Hair is one of the most important features of cartoon characters. In this study, we use the Painterly rendering algorithm (section 3) with particles as the basic technique for rendering the hair shape. The hair model is rendered twice: once for rendering the silhouettes, and again for shading the model. A new specular highlight term is introduced and different styles of specular highlighting are achieved by finding approximate areas of specular highlighting and applying simple texture mapping with user defined textures.

Most of the operations except the Bezier curve fitting and the specular highlight of hair rendering are performed by the GPU taking advantage of the functionality provided by recent hardware. However, since the number of particles determines the quality of hair rendering, a large number of particles are used and the performance decreases accordingly.

In this thesis, methods to produce high quality cartoon character images are discussed, with the main focus being:

- an easy method of applying and controlling silhouettes
- smooth silhouettes and shading, stylised specular highlighting
- a simple hair model for efficient Anime rendering
- the generation of outlines for the hair strands
- an efficient specular zigzag highlighting, characteristic of Anime hair

The thesis is organised as follows. After presenting the motivation of this study in Section 2, an overview of the related work is given in Section 3. Section 4 introduces methods to enhance silhouette edges, and algorithms for diffuse shading and specular highlighting using image-based algorithms and section 5 contains the presentation of the hair model, the creation of the silhouettes, the diffuse shading, and the characteristic zigzag specular highlighting.

Chapter II

Motivation for Proposed Research

2.1 Assisting artists

Skilled artists draw high quality cartoon images, however producing a cartoon is labour intensive work and the production requires many animators to generate in between frames. Some animations in real cartoons are particularly labour intensive to generate. For example, animations involving characters or camera rotation, are hard to generate manually since the object appearance changes rapidly with only small change of the angle. Most of the character drawing is done manually since the quality of cartoon shading is not high enough. It not only takes a long time and requires a huge amount of effort but also causes ‘inconsistency’ problems. Since there are many animators drawing the characters, sometimes different animators have to draw the same cartoon characters. Even though animators try to draw the characters as similar as possible to the original artist, sometimes there are still differences. As a result the same character can look different in different episodes. Some meaningful inconsistency is necessary for exaggerations to convey emotions and to improve understanding. However, such meaningless ‘inconsistency’ is not desirable. Computer generated characters are consistent and require less labour.

Our aim is to improve the quality of cartoon shading so that computer generated characters can be used when labour intensive animation is needed.

2.2 Study of silhouettes

In [40] a hybrid geometry and image-based technique is introduced to remove unnecessary silhouettes from highly complex 3D input geometry in a Pen-and-Ink style. Silhouettes are very crucial features in many Non-photorealistic renderings including cartoon shading. Crowded silhouettes, however, cause darkening in the rendered images which does not necessarily correspond to the intensity of the reference image and gives misleading impressions. The study shows how images can be enhanced by adjusting the intensity of the silhouettes. Therefore we study the intensity of silhouettes in order to enhance the quality of cartoon shading.

2.3 Cartoon shading and hair rendering

Hair rendering in cartoon shading is very important since it is one of the most visual features for human characters. The cartoon hair algorithms presented in [27, 36] are focused on the animation of cartoon hair. However, they do not consider the specular highlighting. Developing a hair model, and algorithms for creating stylised specular highlights would significantly improve the rendering quality of toon shading. Therefore we introduce a new hair model and rendering algorithms to produce stylised silhouettes and specular highlighting.

2.4 Hardware acceleration

Currently available graphics hardware supports several functionality that are useful for toon rendering. Our approach relies on GPU programming for:

- efficient rendering of the geometry information into textures (section 4.1)
- simple and effective texture-lookup operations in the pixel shader (section 4.2.4, 4.3.1 and 5.4.1)
- per fragment operations for the edge detection and other image-based algorithms in the pixel shader (section 4.2, 4.3, 5.2, 5.3 and appendix A.3.1)
- efficient calculation of vertex positions for billboard particles in the vertex shader which are used for hair rendering (section 5.1.2 and appendix A.4)

2.5 Our system

We focus on the quality of the cartoon shading rather than the performance.

The most novel elements of our work include:

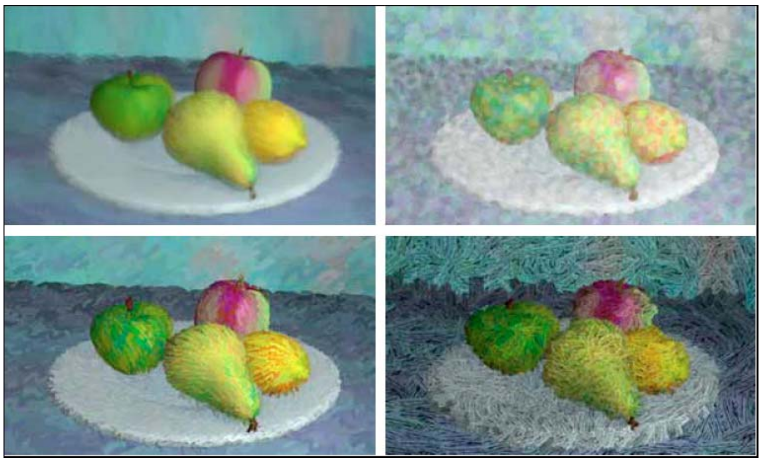
- an easy and effective method of applying and controlling silhouettes in the GPU
- enhanced hand-drawn appearance of silhouettes
- reduced computer generated artefacts

- improved coherence of silhouettes by controlling the silhouette intensity
- an efficient and easy-to-use hair model using particles rendered by the GPU,
- a silhouette renderer for hair strand
- the combination of diffuse lighting with zigzag specular highlighting effects

Chapter III

Related Work

Non-photorealistic Rendering (NPR) has become a popular topic because of its importance in cartoons and character animation. NPR rendering styles include painterly rendering, toon shading, Pen-and-ink illustrations, hatching and the generation of silhouettes. We review each of these in turn.


(a) Geometry-based painterly rendering[10]

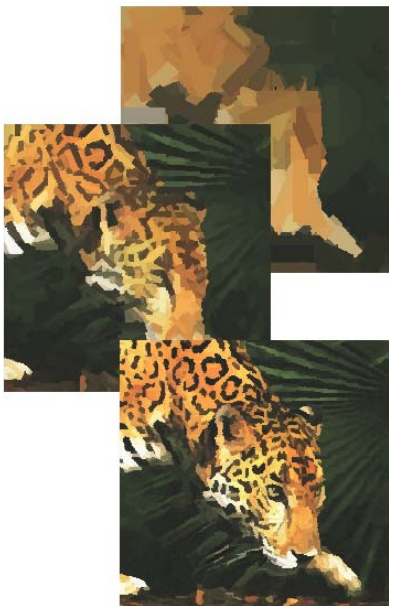
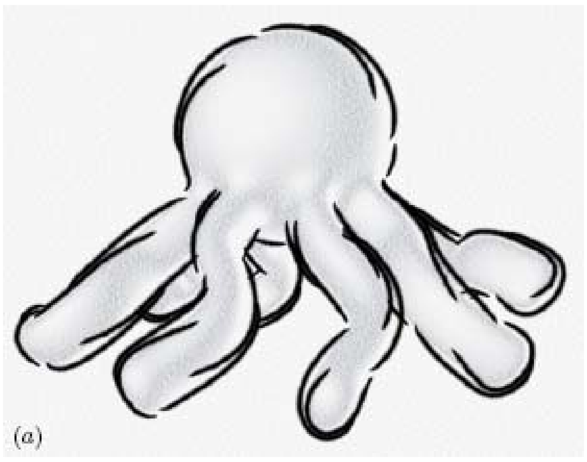

(b) Image-based painterly rendering[16]

Figure 3.1: Painterly rendering

Painterly rendering can be divided into two areas: geometry-based techniques and image-based techniques. Geometry-based painterly rendering is done in the object space where the input is 3D geometry. In [23, 10], particles are used to simulate the paint brush strokes. Three dimensional objects are converted into particle systems and each particle is painted with billboarded textures which are the primary units of the painting (See figure 3.1(a)). The image-based techniques take real photo images as the input. In [16], multiscale edges from the input are utilised to determine the orientation of painting strokes (See figure 3.1(b)).

Silhouettes are very important features in many NPR rendering algorithms, especially in cartoon shading. Kalnins et al. [11] present excellent work on coherent stylised silhouettes for animation (See figure 3.2(a)). Interesting and efficient algorithms for artistic silhouettes are proposed in [11, 28, 12]. In [12], a system with WYSIWYG interfaces is presented. Their system allows users to draw directly on 3D objects. Nienhaus and Döllner [26] present a technique for rendering 3D objects in a sketchy style using image-based techniques in real-time. The geometry properties are rendered and modified by using uncertainty values to achieve variation in styles.



(a) Coherent stylised silhouettes [11]

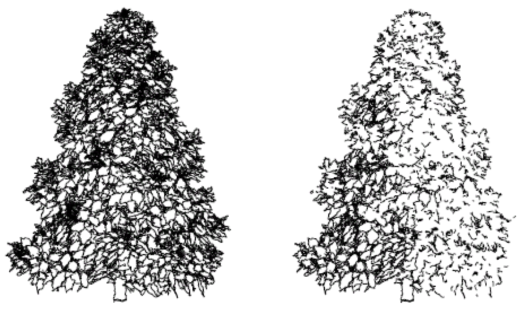

(b) Pen-and-Ink style (Left:Original, Right: Adjusted) [40]

Figure 3.2: Stylised silhouettes and Pen-and-Ink style

In [40] Wilson and Ma present an algorithm where complex 3D objects are rendered in a Pen-and-Ink style using hybrid geometry and image-based techniques. This paper concentrates on removing unnecessary detail of the Pen-and-Ink style when the underlying 3D object is complex (See figure 3.2(b)).

We also use a hybrid image-based approach similar to [40, 26]. We reduce the computer generated artefacts by rendering geometry properties to G-Buffers [32] and applying image-based algorithms including edge-detection (See figure 3.3). The G-Buffer technique is explained in detail later in section 4.1.

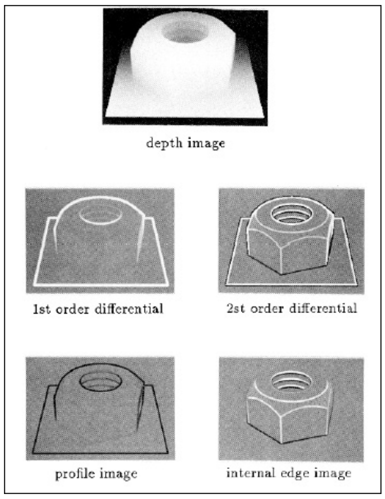
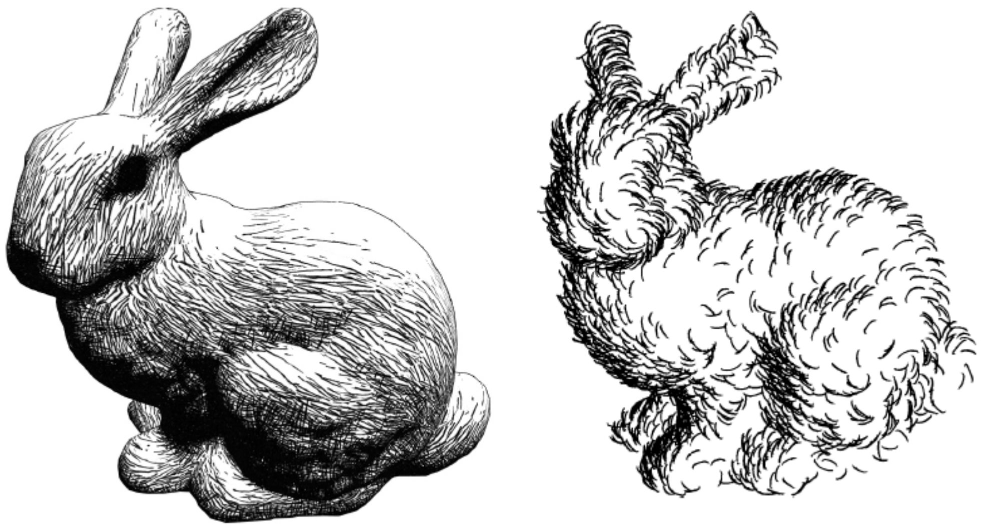


Figure 3.3: G-Buffer example [32]



(a) Coherence handled in object-space [39] (b) Coherence handled in image-space [33]

Figure 3.4: Hatching examples

A number of real-time hatching algorithms of 3D geometry are developed using recent hardware [30, 39, 33]. In [30, 39], geometry-based hatching with multitexturing techniques are used. The coherence is improved by the combination of multitexturing and blending techniques in object-space (See figure 3.4(a)). In contrast, [33] uses image-based hatching and maintains the coherence in image-space (See figure 3.4(b)).

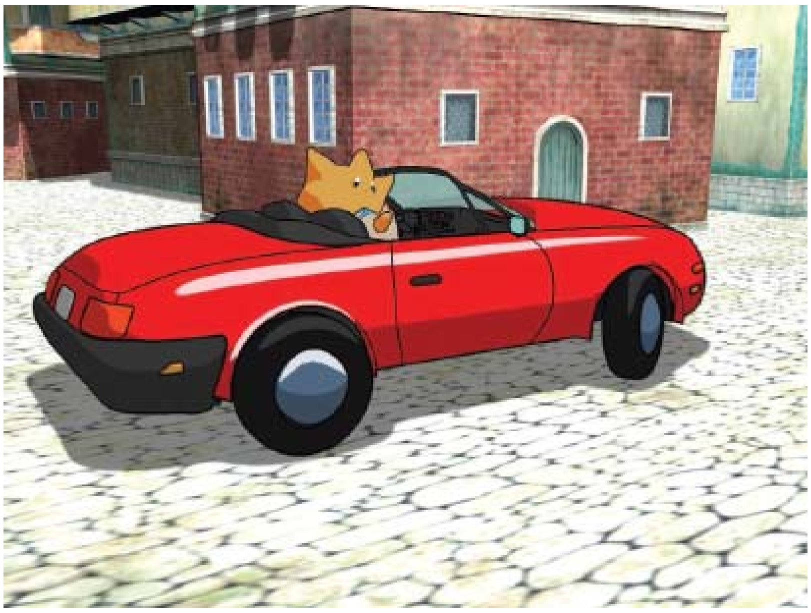


Figure 3.5: Cartoon style specular highlights [3]

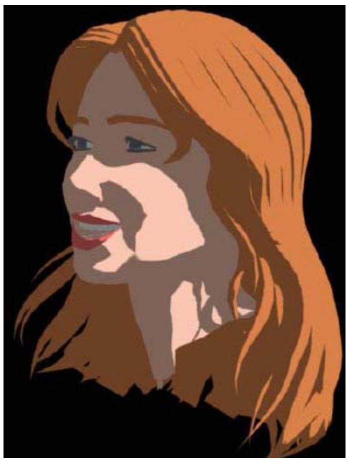
Current research in NPR is directed towards improving toon shading to achieve a more toon-like rendering. Nasr and Higgett introduce a new rendering shader to make rendered objects less glossy and shiny [25]. Anjyo and Hiramitsu introduced cartoon style specular highlights to make toon shading look more like cel animation [3] (See figure 3.5). In addition, Lake et al. [19] present a real-time animation algorithm for toon shading.

Hair plays an important role in cartoons. Hair is not only one of the most important visual features for human beings in real life, but also for human beings in comics and cartoons.

It is still difficult to render over 100,000 hair strands in real-time. Therefore, the hair model is an essential part of the hair shader. A good overview of different hair models are presented in [21].

Noble and Tang use NURBS surfaces for their hair geometry [27] (See figure 3.6(a)). In [20], the hair is modeled in 2D NURBS strips, where the visible strips are tessellated and warped into U-shape strips. In contrast, Kim and Neumann use an optimized hierarchy of hair clusters [13]. Mass-Spring based hair models [31] are commonly used for hair simulation, because they are simple and efficient to use.

However, previous publications in hair modelling have mainly been focused on impressive computer-generated hair simulation and rendering with realistic and photo-realistic behavior [38]. Several papers have been published to improve the appearance, dynamic and self-shadowing of hairs [15, 4], and the model of human hairs [13]. However, fewer researchers have


(a) Cartoon hair with NURBS surfaces [27]

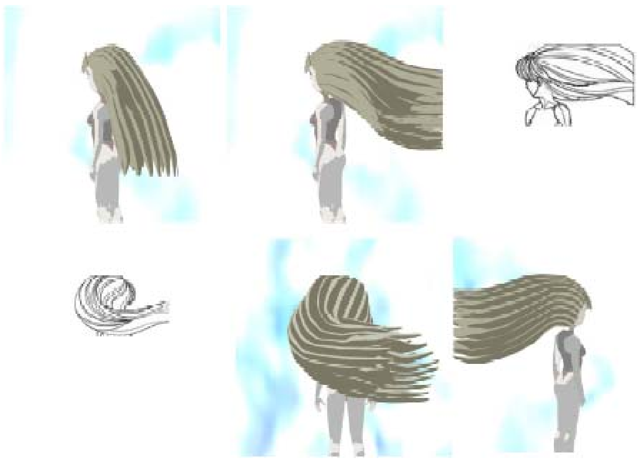

(b) Cartoon hair animation [36]

Figure 3.6: Cartoon hair examples

focused on cartoon hair models [22, 27, 6]. Mao et al. present an interactive hairstyle modeling environment for creating non-photorealistic hairstyles [22]. Their system includes a user-friendly sketch interface which generates hairstyles that can be sketched by modelers. Hairstyles are simply generated by sketching free-form strokes of the hairstyle silhouette lines. The system automatically generates the different hair strains and renders them in a cartoon, cel animation style. A similar approach with good results has been presented in [36] (See figure 3.6(b)). Côté et al. introduce a polygon-based technique to recreate an inking technique [6]. The system provides an intuitive interface to assist artists and produces cartoon-like hair using a hatching technique.

Our hair rendering approach is based on the painterly renderer model presented by Meier in [23] where a large number of particles are used to simulate brush strokes paintings (figure 3.1(a)).

Chapter IV

Reducing Computer Generated Artefacts In Toon Shading

The geometry buffer (G-Buffer) technique is a well known method to implement image-based algorithms [32]. Computer generated artefacts appear in 2D space so we use the G-Buffer as a basis for reducing computer generated artefacts. We apply various image processing algorithms to extract silhouette edges and to further enhance the rendering quality of toon shading.

4.1 Rendering of G-Buffer

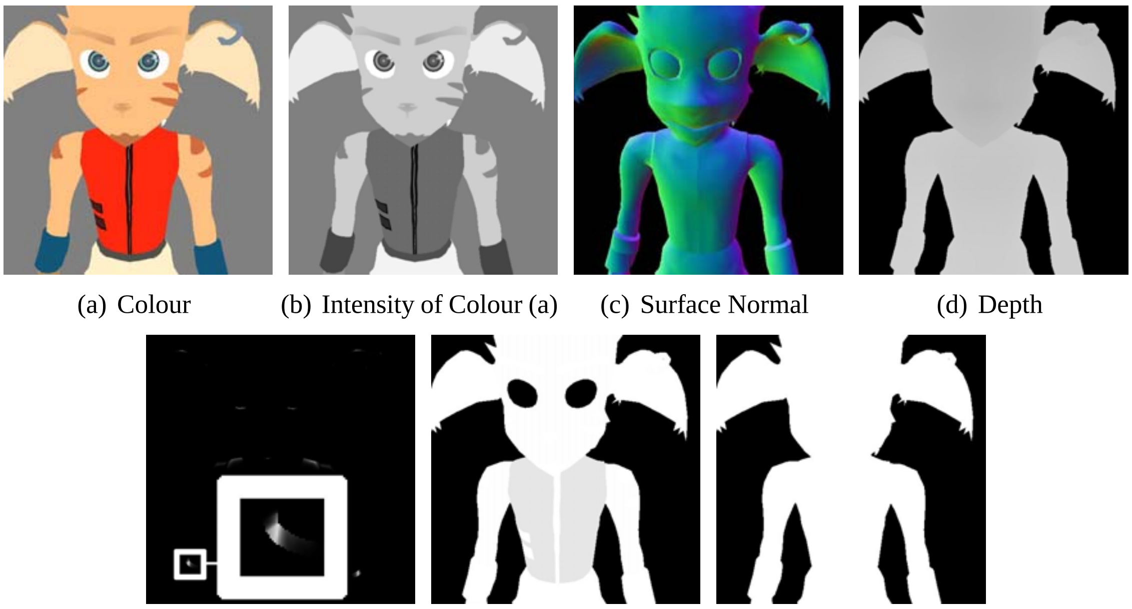
In [32], a G-Buffer is defined as following: “A G-buffer set is the intermediate rendering result, and used as the input data for enhancement operations. Each buffer contains a geometric property of the visible object in each pixel.”. In our implementation, we render the geometric properties into textures using the OpenGL API and the OpenGL Extensions “ARB render texture” which allows faster texture generation by rendering without copying the frame buffer manually.

In this phase, all geometry properties are rendered to textures (G-Buffers). The geometry information derived from the G-Buffer is used as inputs to the image-based algorithms. The rendered properties are packed into a series of four channels since a texture has four channels. The input model is rendered three times for different sets of information. Figure 4.1 shows the rendered G-Buffers.

This system could be re-implemented in the future using the latest the OpenGL extensions “Framebuffer object” and “ATI draw buffers” which did not exist when this study started. The Framebuffer object extension is similar to the ARB render texture extension but faster. The ATI draw buffers extension allows the GPU to render to multiple draw buffers simultaneously. It means the model does not need to be rendered three times but once.

4.1.1 Colour

First, we render the colour of the input geometry: (R,G,B,I). The colour includes the texture or material properties of the input model. In the fourth channel, the grayscale intensity of the colour (I) is stored. The intensity (I) is pre-computed in this phase for efficiency and is used for edge detection in later phases. We use the luminance value in the YIQ colour model [9] for getting the



(a) Colour (b) Intensity of Colour (a) (c) Surface Normal (d) Depth

(e) Specular highlight (f) Alpha of the input texture (g) Mask

Figure 4.1: Rendered information

gray-scale intensity (I). This value can be calculated using the following formula.

$$I = dot(RGB, (0.30, 0.59, 0.11))[29] \tag{4.1}$$

4.1.2 *Normal and Depth*

Second, we render the surface normal and depth buffer of the input geometry: (N_x, N_y, N_z, D). The surface normal is stored as colour in the texture. However, colour values in textures vary between [0,1] where the surface normal values vary between [-1, 1]. Therefore the surface normal is converted using equation 4.2 when it is rendered, and restored using equation 4.3 when the G-Buffer is used in later operations.

$$N'_{xyz} = (N_{xyz} + 1)/2 \tag{4.2}$$

$$N_{xyz} = (N'_{xyz} \times 2) - 1 \tag{4.3}$$

4.1.3 Other information

Finally, three other types of geometry related information are stored: (S,A,M). The specular highlight (S) is rendered and is modified later (section 4.3.2). The alpha value (A) of the input texture is not the rendered G-Buffer texture but the original input texture that is mapped on the input model, however, this alpha channel of the input texture is often redundant. In this paper, we use the alpha chanel to change the intensity of the colour silhouettes (section 4.2.1). Users directly change the alpha value of the input texture to change the silhouette intensity. The mask value (M) is set to one if the pixel is visible. It is necessary to store the mask value since the colour and normal textures are modified by applying filters. The mask is used to distinguish foreground and background for later use (section 4.4).

4.2 Rendering of Silhouettes

The aim of toon shading is to mimic hand-drawn cartoons and the silhouette is one of the most important features that creates a hand-drawn appearance. The silhouette is found by applying an edge detection algorithm on the rendered G-Buffers from the previous phase. Most of the image processing operations are performed in the GPU using a fragment shader.

In traditional toon shading, edge detection algorithms are often performed on the surface normal and the depth discontinuity [32, 5] but not the colour discontinuity.

In this study, we define three types of silhouettes: colour silhouettes, normal silhouette and depth silhouette. They are detected when there are colour intensity discontinuities, surface normal discontinuities and depth discontinuities. Colour discontinuities represent edges of material properties. Normal discontinuities correspond to crease edges, and depth discontinuities correspond to boundary edges on an object that separates the object from the background.

Figure 4.2 shows the pipeline of silhouette rendering. Note that colour intensity is already calculated from the previous phase (section 4.1.1). The first and the second step in figure 4.2 are performed in a single fragment shader. The third step however, is performed by the CPU which is the reconstruction of the silhouette. Unfortunately, the intensity variation from the second step is lost after the reconstruction (section 4.2.5). Finally, the traced silhouette and the original silhouette are merged to restore the intensity.

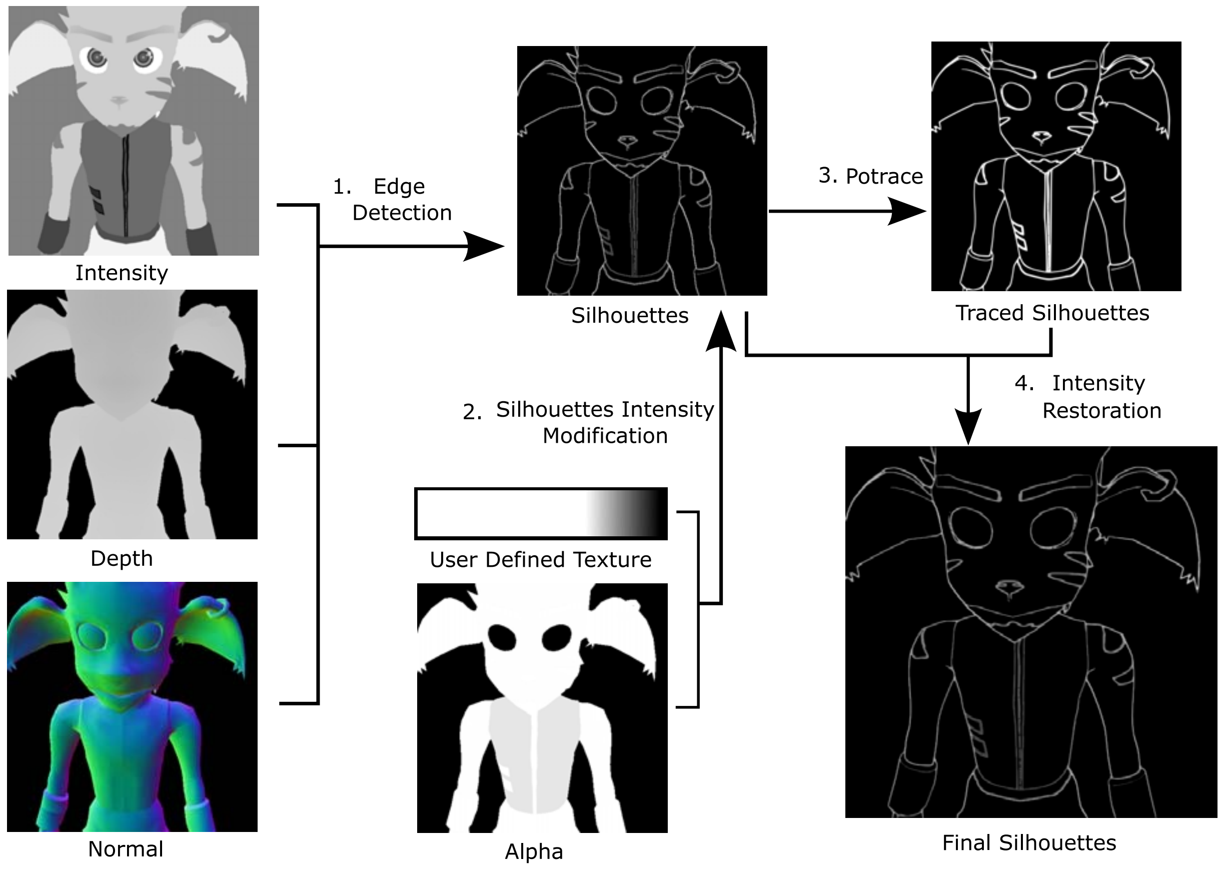


Figure 4.2: The silhouette Pipeline: (1) Edge detection on the colour intensity, the normal and the depth maps (2) Apply intensity modification using user defined input (3) Reconstruct the silhouette using Potrace (4) Restoring the silhouette intensity

4.2.1 Colour Intensity discontinuity

The Sobel filter is a well known edge detection algorithm which is simple and efficient. We apply the Sobel filters horizontally and vertically on the intensity of the colour (I).

```
Horizontal filter      Vertical filter    Pixel Value of Neighbor
   -1 -2 -1                -1 0 1                  A B C
    0  0  0                -2 0 2                  D X E
    1  2  1                -1 0 1                  F G H
```

$$Sobel_{value} = (|-A-2B-C+F+2G+H|+|C+2E+H-A-2D-F|)/8 \tag{4.4}$$

If the $Sobel_{value}$ is higher than a user defined threshold, the I_{edge} value is set to one where I_{edge} is the intensity value of the colour silhouette (See figure 4.3(a)).

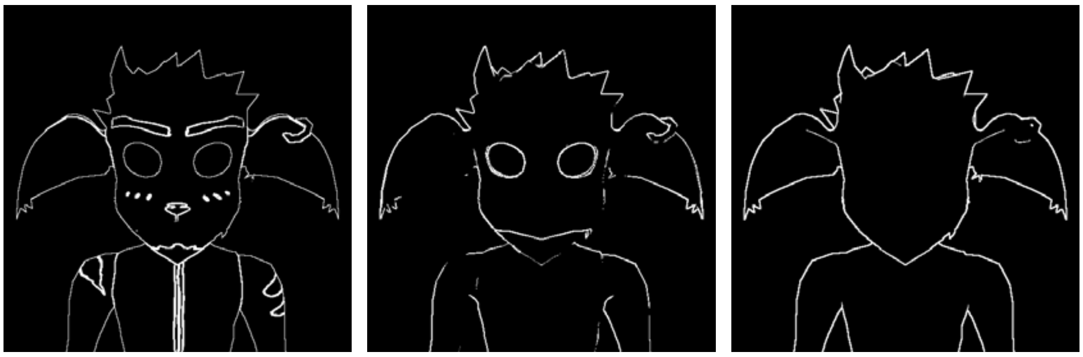
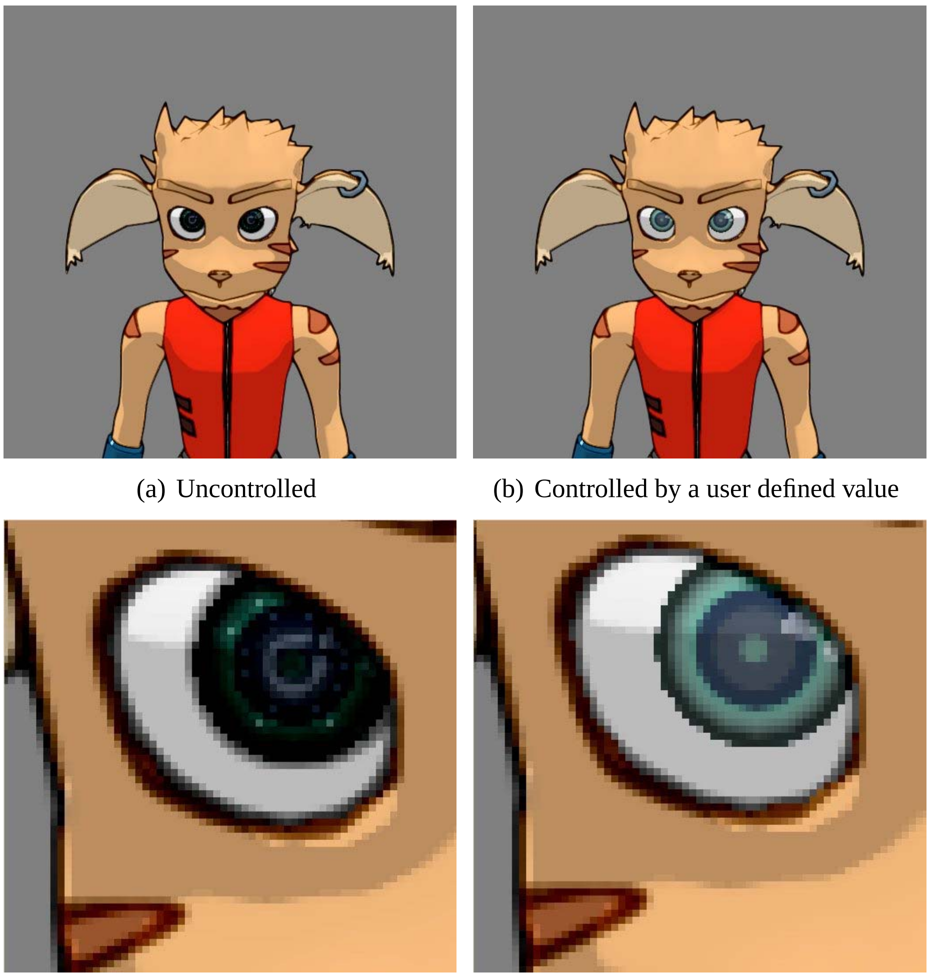

(a) Colour silhouettes (b) Normal silhouettes (c) Depth silhouettes

Figure 4.3: Edge detections

After the colour silhouette is found, users can change the intensity of the colour silhouette by changing value of the alpha (A).

$$I_{edgefinal} = I_{edge} \times A \tag{4.5}$$

It is important that users have full control of the colour edges for several reasons. First, users may not want the silhouette in certain areas. Many features of 3D characters can be achieved by texture mapping and can be enhanced with edge detection, However it is meaningless if users do not want it. In figure 4.4, the eyebrow, pupil of the eyes, lines on the cheeks and nose etc. of the model are done by texture mapping. However, silhouettes in certain areas might not be desirable; such as the pupil of the eyes. Figure 4.4(c) shows the original (uncontrolled) image where the silhouette is crowded in the pupil while figure 4.4(d) shows that redundant silhouettes have been effectively removed from the eye by appropriately adjusting the alpha value A. As figure 4.1(f) shows the alpha value is already removed around the pupil by the user. This task can easily be done by using image editing tools. High frequency features in the texture can cause coherence problems when animated. The high frequency features can appear and disappear depending on the distance and angle of the camera to the model when it is rendered. It creates unstable silhouettes which flicker as the object is moved with respect to the camera. The unstable colour silhouette can be removed easily by changing the value of (A). The edge detector can also detect noise from the input texture which can happen because the texture mapping is done from 2D space (UV) to 3D space (vertices). If the texture is stretched or the input texture is of low


(a) Uncontrolled (b) Controlled by a user defined value

(c) Zoomed image of (a) (d) Zoomed image of (b)

Figure 4.4: Colour edge control

resolution, it creates alias effects when it is rendered. The alias effect can be detected with the edge detector set at a certain threshold and depending on the intensity difference.

4.2.2 Normal discontinuity: Crease Edge

We detect the normal discontinuity by simply comparing the angles between neighboring pixels on the normal map.

```
Index
0 1 2    // 4 is the current pixel
3 4 5
6 7 8
```

$$CosAngle = abs(dot(1,7)) + abs(dot(3,5)) + abs(dot(0,8)) + abs(dot(2,6)) \tag{4.6}$$

CosAngle is the sum of the dot product of the horizontal, vertical and diagonal neighbors. $abs(dot(1,7))$ presents the dot product between the surface normals at index 1 and 7. The intensity of the normal silhouette N_{edge} is set to one if *CosAngle* is smaller than a user defined threshold (See figure 4.3(b)).

4.2.3 Depth discontinuity

The Sobel filters applied for determining depth discontinuity are the same as those applied for calculating the colour silhouettes. The depth silhouette shown in figure 4.3(c) is not only used for the final intensity of the silhouette (section 4.2.4) but also for the smoothing filter in section 4.3.3.

4.2.4 Intensity of silhouette

We modify the intensity of silhouettes due to two reasons. The first reason is to stylise the appearance as most traditional toon shading silhouettes are just black (See figures 4.5(a),4.5(b) and 4.5(c)). In this study, the intensity of the silhouette at a pixel depends on the number of discontinuities in the colour intensity, normal and depth maps (G-Buffer).

$$I_s = I_{edge} \times I_{weight} + N_{edge} \times N_{weight} + D_{edge} \times D_{weight} \tag{4.7}$$

$$I_{weight} + N_{weight} + D_{weight} = 1 \tag{4.8}$$

where I_s is the weighted sum of intensities and $I_{edge}, N_{edge}, D_{edge}, I_{weight}, N_{weight}, D_{weight}$ vary between [0,1].

The input texture is a useful guideline to vary silhouette intensity since the modeler creates the input texture to suit the input geometry. In other words, strong intensity of a feature in the input texture means the modeler wanted the feature to be emphasised. We blend the intensity of

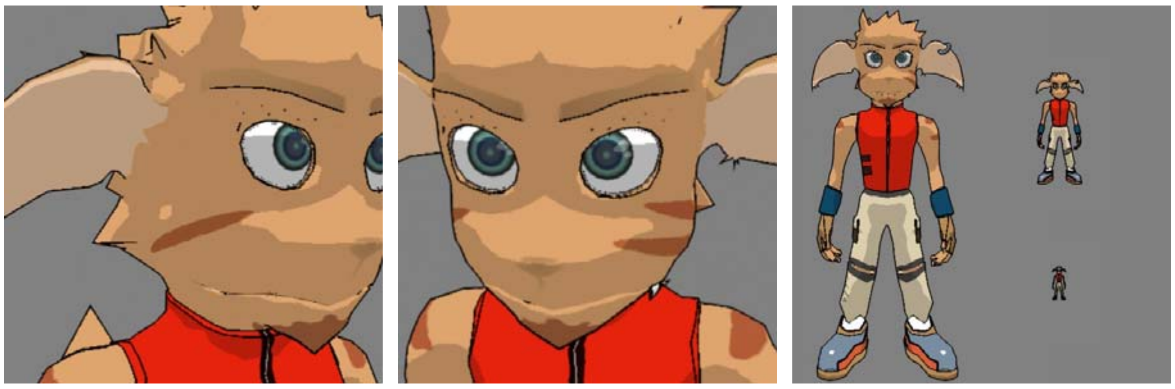

(a) (b) (c)

Figure 4.5: Traditional Black Silhouettes

the silhouette with the rendered texture intensity (I).

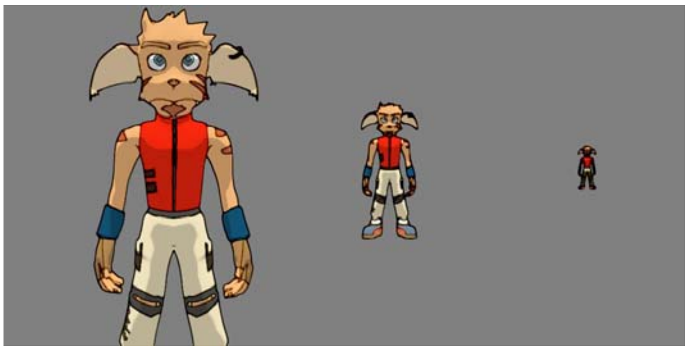
$$I_s^{'} = I_s \times w + I \times (1 - w) \tag{4.9}$$

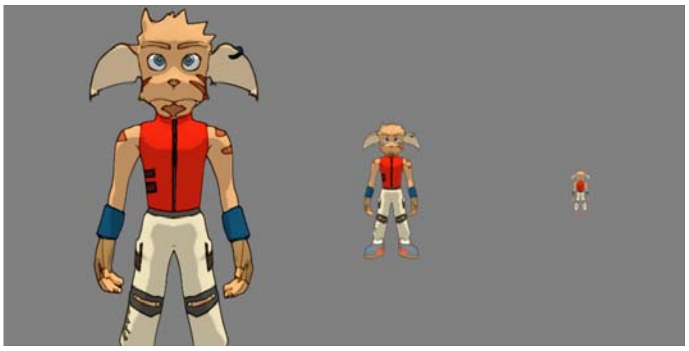
where $I_s^{'}$ is the modified silhouettes intensity and w is the weight between the intensity of silhouettes and (I).

The second reason to vary intensity is to eliminate cluttering of the silhouettes and to enhance silhouette coherence in animation, when the camera is moved towards or away from the character model. When the camera is moved away from the model, the size of the model on the screen gets smaller and it makes the silhouette unstable as more information is crowded in a smaller region. It also makes silhouettes crowded in small places, which is not visually pleasing (See figure 4.6(a)). We recognise, however, that the shape of a model is more important than the silhouettes when the camera is far away from the model since the model itself is too small. In this algorithm, we fade the silhouette away when the camera zooms out (See figure 4.6(b)). Users can easily change the rate of fading by creating a texture like that given in figure 4.6(c). A simple texture lookup is performed as follows.

$$I_{final} = I_s^{'} \times DT(d) \tag{4.10}$$

Here DT is the depth intensity texture look up, d is the depth value of the current pixel and I_{final} is the final intensity of the silhouettes. The intensity of the silhouette varies depending on the depth values, so the intensity of the contour silhouette can be modified using different depth value to perform the texture lookup. The depth values along the contour can be changed using


(a) Uncontrolled silhouette


(b) Controlled silhouette

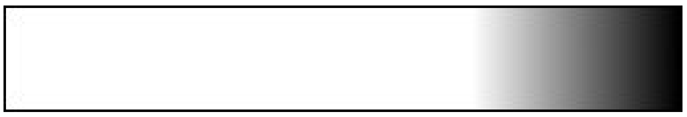
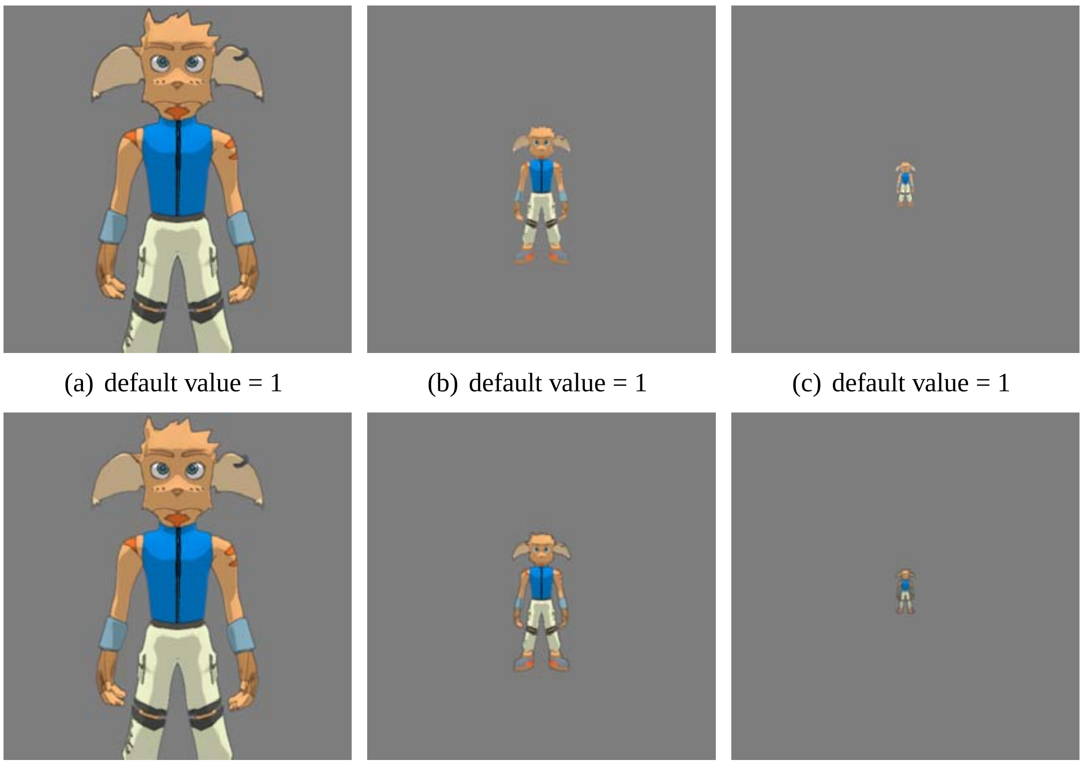

(c) User defined texture

Figure 4.6: Silhouette intensity control by depth information

the following methods.

The method depends on the background (default) value of the texture that the depth is rendered to. Then, the average, the minimum, or the maximum of depth values of neighboring values can be used for the texture look up, instead of the current depth value. This brings many interesting effects. For example, if the background value is zero and, the average depth value is used for the texture lookup, the intensity of the silhouette decreases because the depth values around the contour are all zero then the result of the texture lookup value will be near one. So the boundary silhouettes can remain while silhouettes inside are made to fade away (See figures 4.7(d), 4.7(e) and 4.7(f)). If the background depth value is set to one, then contour silhouette fades away together with other silhouettes (See figures 4.7(a), 4.7(b) and 4.7(c)).

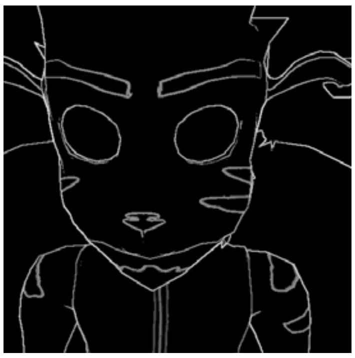

(a) default value = 1 (b) default value = 1 (c) default value = 1

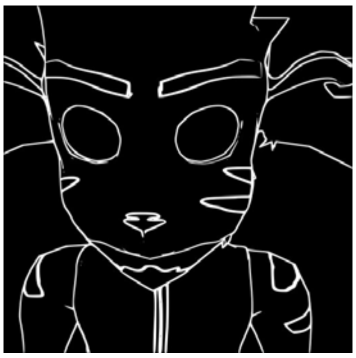
(d) default value = 0 (e) default value = 0 (f) default value = 0

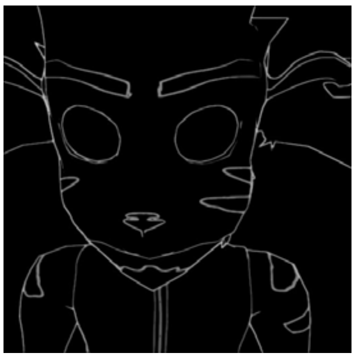
Figure 4.7: Silhouette difference depending on default depth texture values

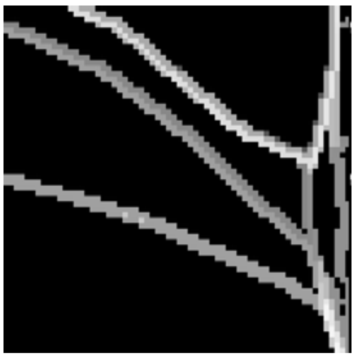
4.2.5 Bezier curve fitting: Potrace

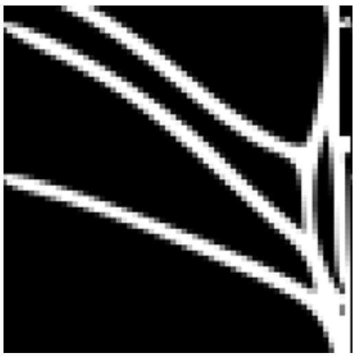
The silhouettes found from the previous phase are sampled and reconstructed with Bezier curve fitting (cf. appendix B.1). ‘Potrace’ is a wonderful library for tracing a bitmap, transforming it into a smooth, scalable image [34]. The Potrace library takes a binary image and produces a smooth grayscale image. The main advantage of using Potrace is that it produces nice lines with hand-drawn appearance. This process, however, cannot be performed by the GPU. Note that this phase is the only phase that is purely performed by the CPU in the computer generated artefacts reduction algorithms. Potrace takes a binary image as an input therefore the silhouette is needed to be thresholded and as the result, the intensity variation is now lost in the traced image (See figure 4.8(b), 4.8(e)).


(a) Original silhouettes


(b) Traced silhouettes


(c) Combined silhouettes


(d) Zoomed image (a)


(e) Zoomed image (b)

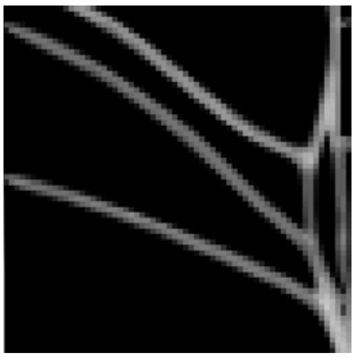

(f) Zoomed image (c)

Figure 4.8: Original Silhouettes, traced silhouettes and combined silhouettes

4.2.6 Restoring of intensity

The intensity variation is restored by combining the original and the traced silhouettes. The traced silhouette, however, is thicker than the original silhouette because of the anti-aliasing. The neighboring pixel values of the original silhouette are averaged and multiplied with the traced silhouette to restore the intensity (See figure 4.8(c), 4.8(f)).

4.3 Filtering and Smoothing

Toon shading characters are often easily perceived as computer generated because of the computer generated artefacts. In this section, we show how artefacts can be reduced by applying 2D filters. Smooth diffuse shading and stylised specular highlights are achieved by various 2D filters. Figure 4.9 shows the overall shading pipeline. Note that the depth silhouette from the previous phase is used as a guideline to blur the contour of the colour map.

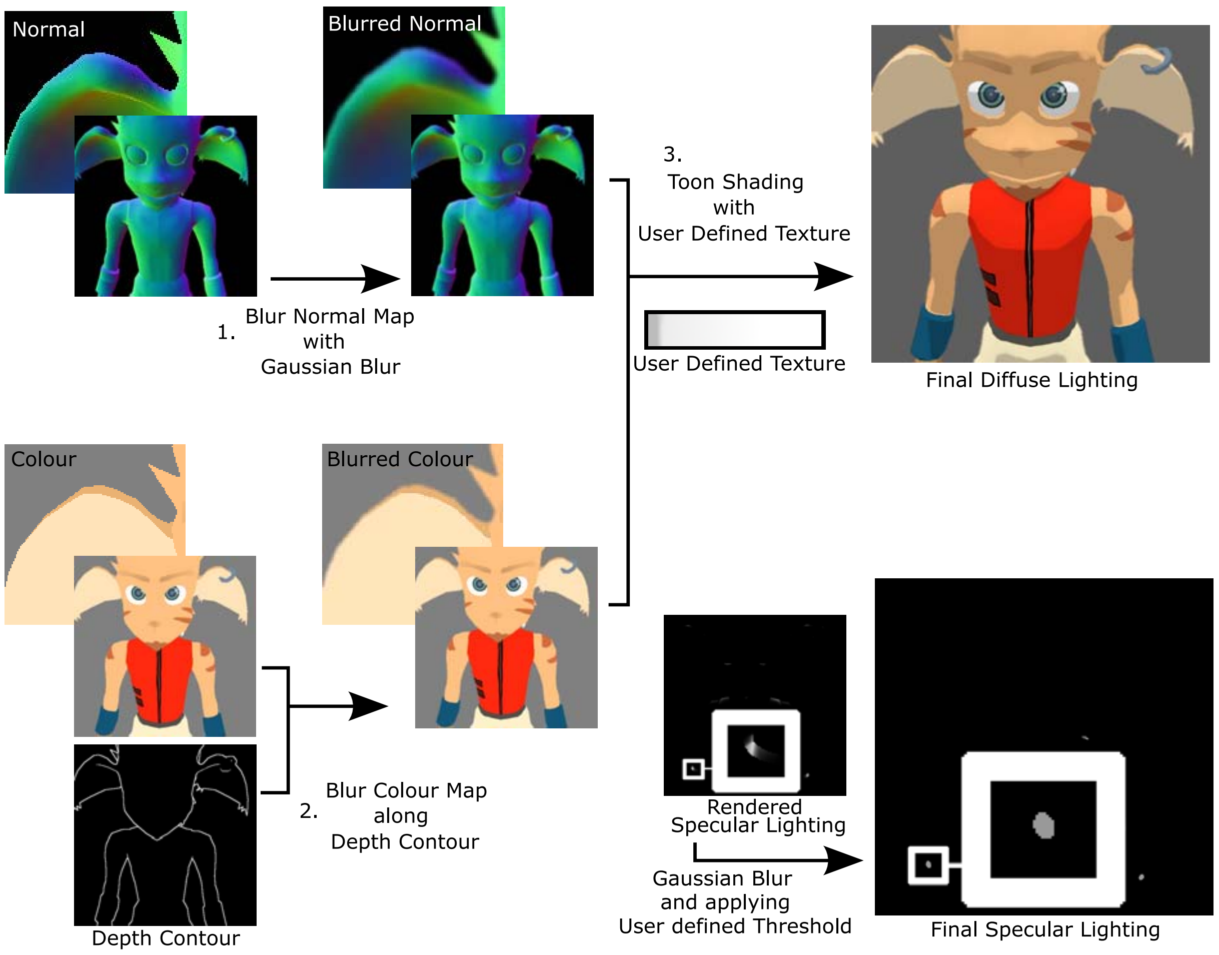

Normal
Blurred Normal
1. Blur Normal Map with Gaussian Blur
3. Toon Shading with User Defined Texture
User Defined Texture
Final Diffuse Lighting
Colour
Blurred Colour
Depth Contour
2. Blur Colour Map along Depth Contour
Rendered Specular Lighting
Gaussian Blur and applying User defined Threshold
Final Specular Lighting

Figure 4.9: Diffuse Lighting:(1) The normal map is blurred (2) the colour map is blurred along the depth contour (3) The diffuse value is calculated with the blurred normal map and the blurred colour map. A simple texture lookup is performed to achieve two-tone shading with the user defined texture. Specular Lighting: the rendered specular lighting is blurred with Gaussian blur and a user defined threshold is applied

4.3.1 Diffuse Lighting

The shading of a model largely depends on the geometry especially the surface normal. 3D human character models are usually required to have smooth surfaces. The shading areas, however, can have sharp, jaggy features when the model is of low polygon. Therefore we remove the high frequency components from the normal map by applying Gaussian blur with kernel size 11x11 pixels. Figure 4.10 shows the difference between the rendered results, using the original and the blurred normal maps. The shading area in figure 4.10(d) is smoother and contains less unnatural high frequencies than figure 4.10(c) does.

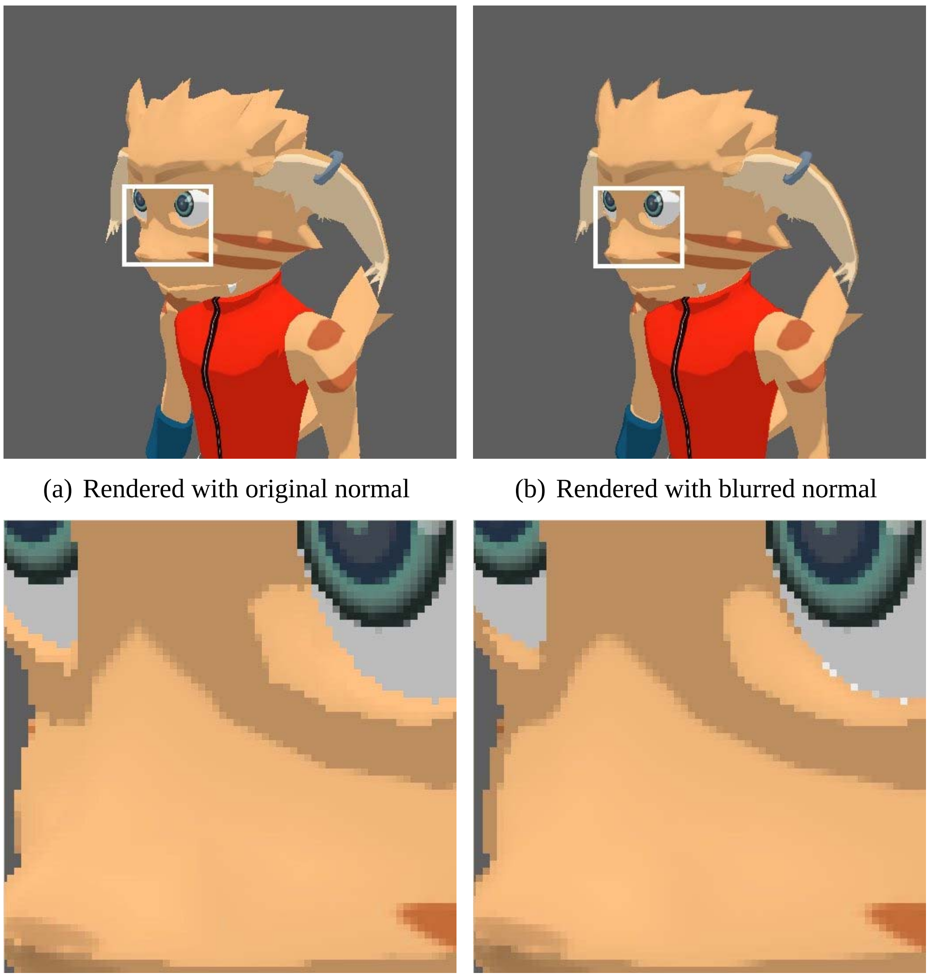

(a) Rendered with original normal
(b) Rendered with blurred normal
(c) Zoomed image of (a)
(d) Zoomed image of (b)

Figure 4.10: Shading comparison

The normal map is blurred instead of the rendered diffuse values because, if the diffuse lighting is blurred, it destroys the boundary of shading lines. Note that the blurring of the normal map is performed after the silhouette extraction so it does not affect the crease edges (the normal silhouettes). The diffuse value is calculated with the blurred normal map, and modified by toon texture look up with a user defined texture (See figure 4.9). Finally, the diffuse value is multiplied by the colour map to obtain diffuse lighting.

4.3.2 Specular Highlight

We modify the specular highlight to create cartoon specular highlight since changes in specular highlights affect the style of the rendering [25, 3]. The rendered specular highlight is blurred with Gaussian blur with kernel size 11x11 pixels and the resulting intensity values are thresholded using a user defined value (See figure 4.9). Our approach differs from the specular highlights of the traditional cartoon shading. In traditional cartoon shading, the specular lighting is usually done by the three-tone shading. The shading is divided into three shades: dark, bright and very bright (the specular highlights). The three-tone shading, however, is just a diffuse shading and is independent on the view vector direction. Therefore the highlights do not change depending on the view position. The advantage of our approach is that it is stylised and changes depending on the view position. The disadvantage is that it is slower than the traditional shading since specular highlights need to be rendered separately and blurred.

4.3.3 Smoothing of model contour

When the camera zooms out, the rasterisation of the model itself becomes more apparent, because the silhouettes which cover the aliased contour fade away (section 4.2.4). The rasterisation is more visible along the contour of the model.

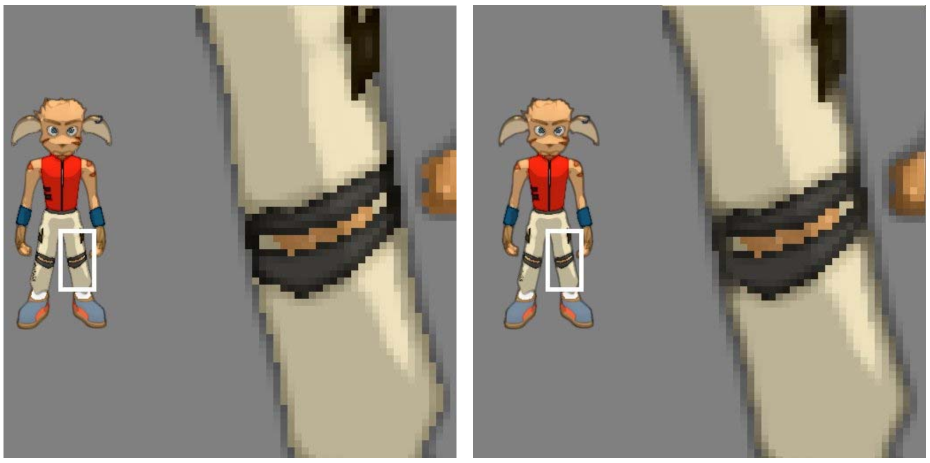

(a) Traditional (non-blurred) image (b) Contour blurred image

Figure 4.11: Contour blurred technique

In [7], applying Gaussian filters to smooth the image is suggested. Blurring the whole image, however, might not be desirable when users want some high frequency features on the model. Therefore we only apply blur the colour map (C) along the depth discontinuities. This blurring allows smooth contours even the silhouettes is faded out. Figure 4.11(b) shows that the contour blurred image is much smoother and cleaner compare to figure 4.11(a) the non-blurred one.

4.4 Merging

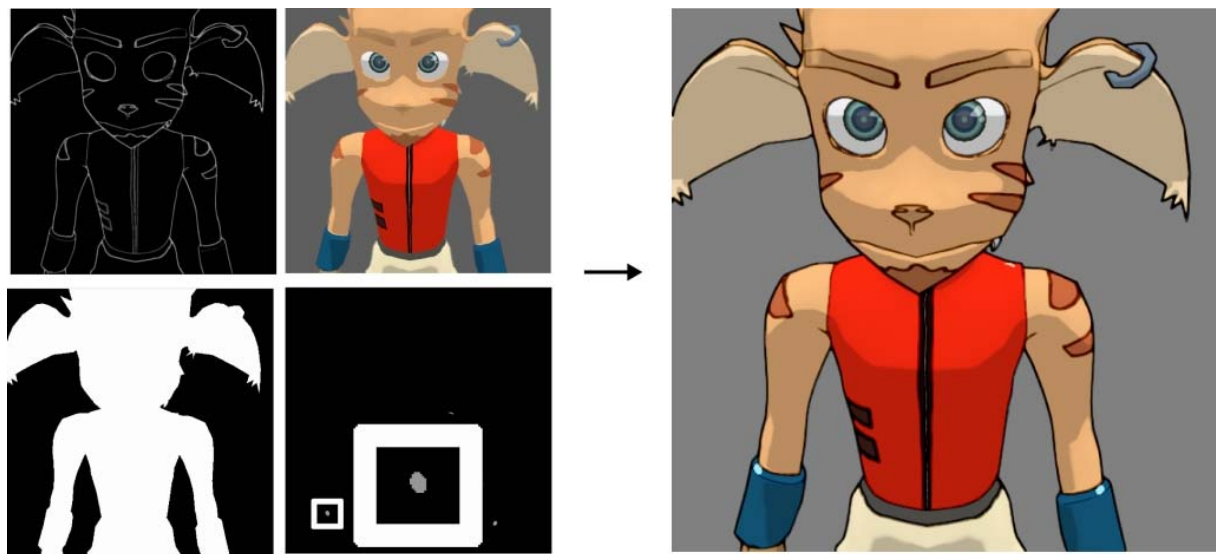


Figure 4.12: Merging and final result

Finally, all rendered images such as the diffuse lighting (R,G,B), the modified specular highlight (S) and stylised silhouettes are combined into one image.(See figure 4.12) The process is outlined by the following shader code.

```
if ( M > 0 ) // M is mask;
{
    OUT.color = DiffuseColor;
    OUT.color.w = M;
}

if ( silhouette > 0 )
{
    OUT.color = OUT.color * ( 1 - silhouette);
    OUT.color.w = OUT.color + silhouette;
}
OUT.color = OUT.color + specular;
```

4.5 Comparison with Traditional Toon Shading

In this section, we compare the quality of the new approach with the traditional toon shading. The same model is rendered with the traditional toon shading and the Pen-and-Ink style in 3DS Max. There is a little bit of perspective difference in figure 4.13 since they are rendered from two different systems. The following table lists the major difference between the traditional method and our approach.

	Traditional method	Our Approach
Edge detection on Colour discontinuity	Only Material	✓
Edge detection on Normal discontinuity	✓	✓
Edge detection on Depth discontinuity	✓	✓
Smooth Silhouettes	*X*	✓
Silhouette intensity control on specific area	*X*	✓
Silhouette intensity control depending on the depth	*X*	✓
Cartoon specular highlighting	view independent	view dependent

Our approach is more advanced because of the following reasons.

1 The colour edge detection is performed and users have full control over it. The figure 4.13(b) shows that many texture features such as eyebrows, lines on the cheek, the boundary of ears are enhanced with silhouettes. The line on the eyebrow in figure 4.13(a) is detected not because of the texture difference but the polygon group difference. The 3DS Max Pen-and-Ink technique performs edge detection on the normal difference and the polygon group difference (users can define polygons in different groups, and they can be used for assigning different material properties) but not the texture difference.

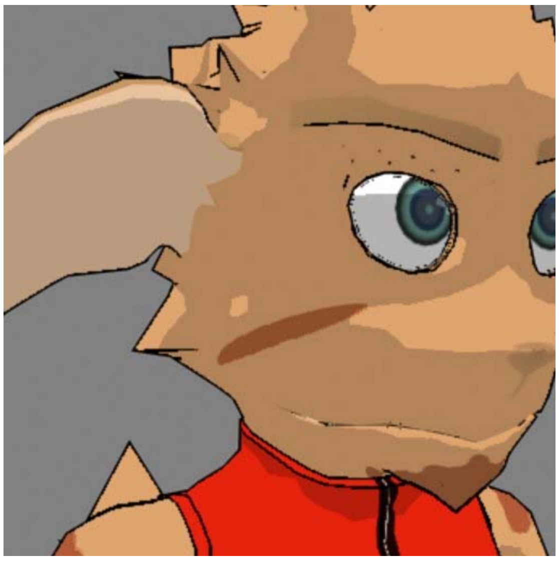
2 The silhouette coherence is strengthened by changing the intensity of silhouettes, when the camera zooms in and out while the traditional method suffers from crowded silhouettes (See figure 4.13(e)).

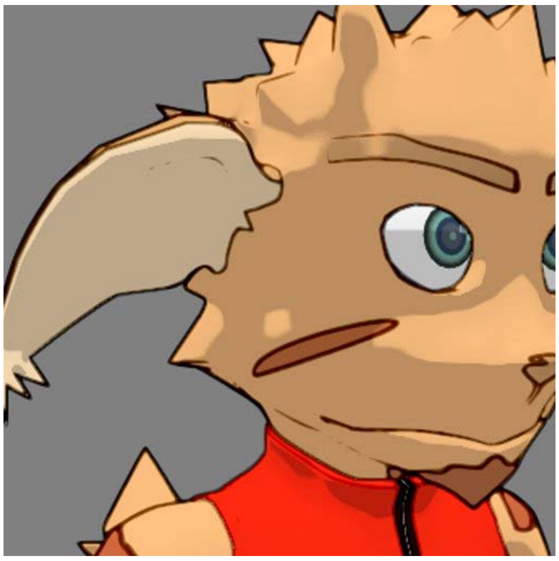
3 The rendering artefacts such as rasterised lines and jagged shading areas are reduced by

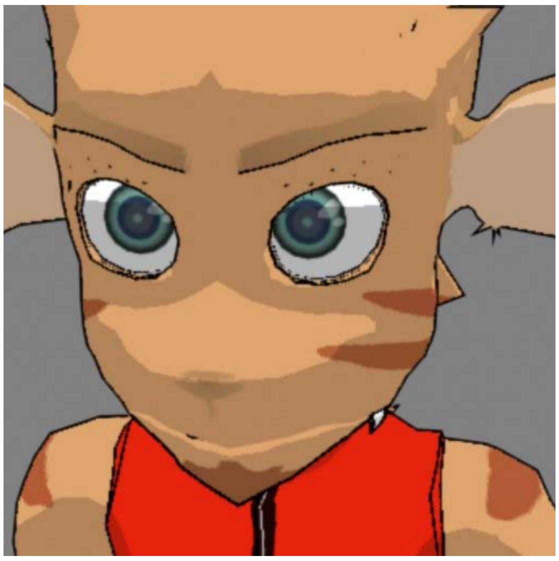
applying various image processing filters. Figure 4.13(c), 4.13(d) shows the difference between new approach and traditional toon shading.

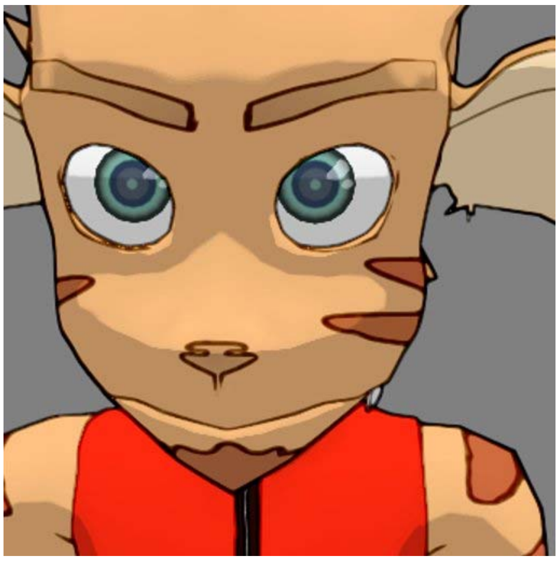
4 The specular highlighting is view dependent. Most of the traditional cartoon shading uses the three-tone shading and it is just a diffuse shading and view independent. However, users may want view dependent specular highlights so that the shape of the specular highlights change depending on the view position.

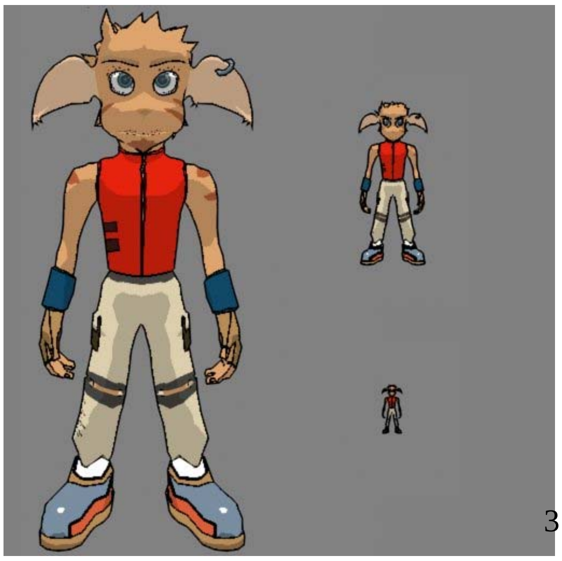
Figure 4.14 shows the result images of different 3D models with our approach.


(a) Traditional pen and ink style


(b) New texture feature enhanced style


(c) Traditional black silhouettes


(d) Smoothen and intensity varied silhouettes


(e) Traditional silhouettes

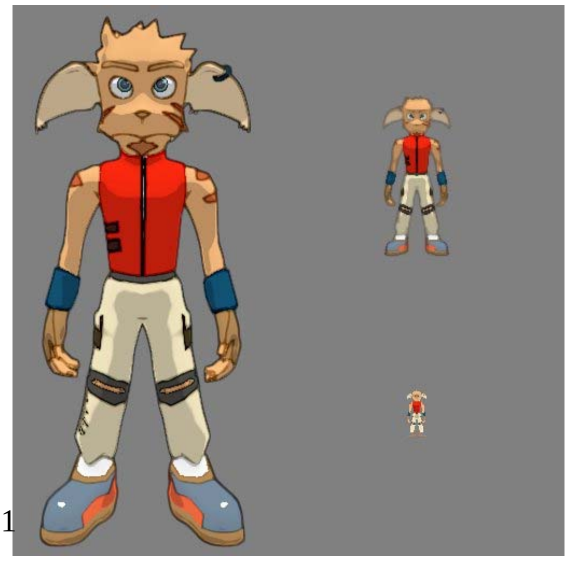

(f) Intensity fade out technique

Figure 4.13: Comparison with traditional method

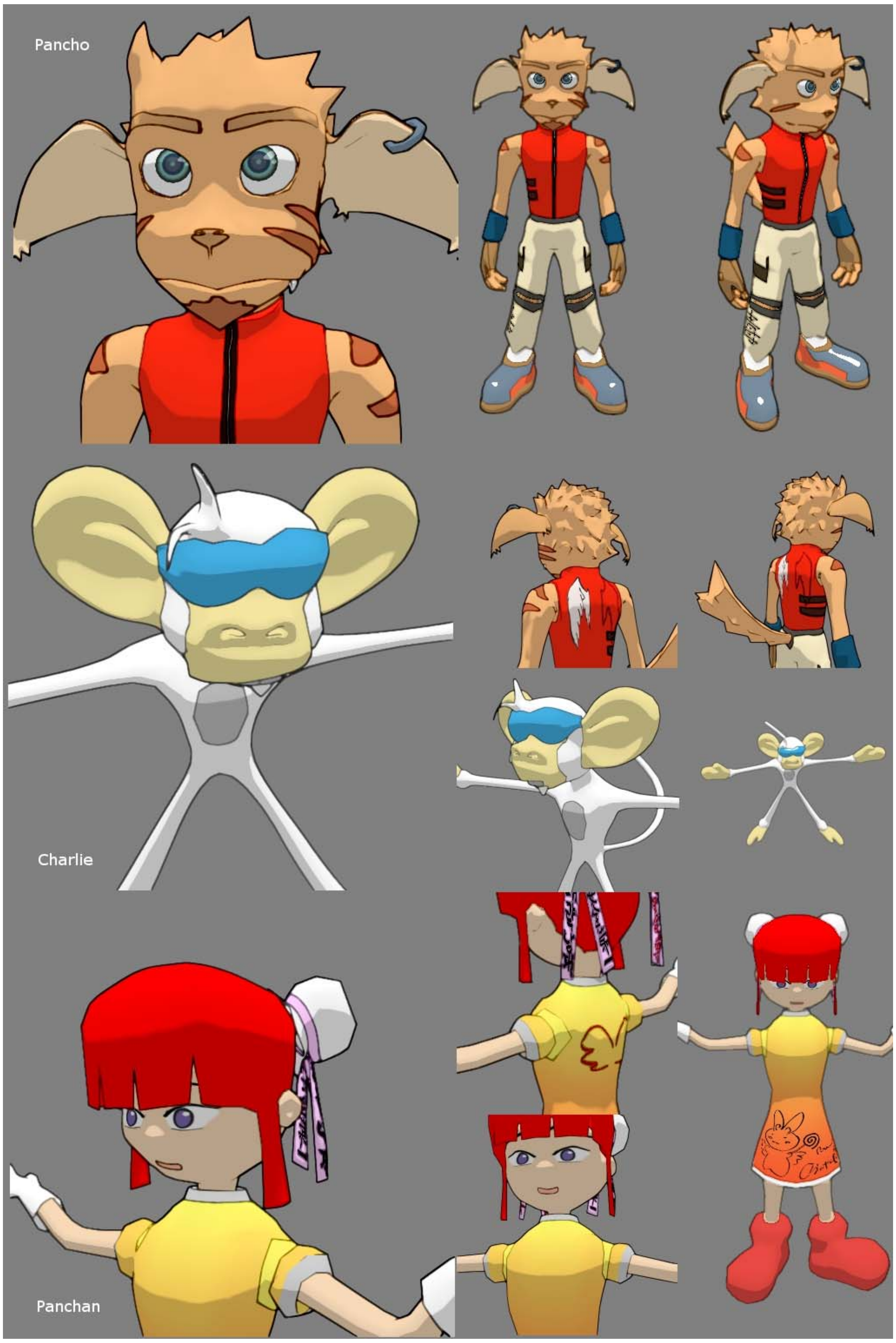


Figure 4.14: Rendered images of different 3D models

Chapter V

Introducing New Stylised Hair Model

Hair plays an important role in Anime, the Japanese version of Animation. After eyes, the hair is the feature that best shows the characters' individuality [24]. Hairstyle says a lot about personality and is characterised by the simplification of the hair strands and shading, the black outlines, and by special specular highlighting. The hair is a crucial feature used to distinguish between different cartoon characters, especially in Anime (See figure 5.1).

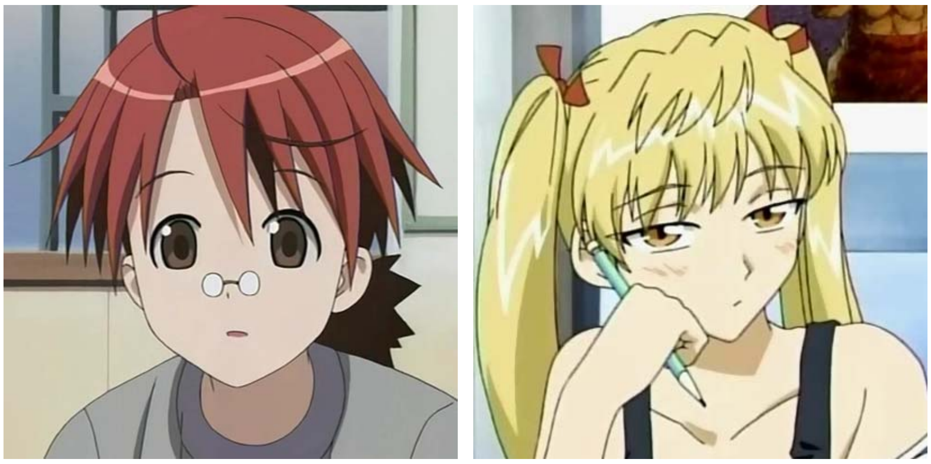

(a) Negima [1] (b) School Rumble [14]

Figure 5.1: Real Anime examples

Figure 5.2 shows the pipeline of our hair rendering. The rendering technique includes:

- a new hair model consists of mass-spring structure with Catmull-Rom splines
- generation of hair strands using billboarded particles and rendering of stylised silhouettes
- diffuse lighting issues with particles
- and stylised specular highlights

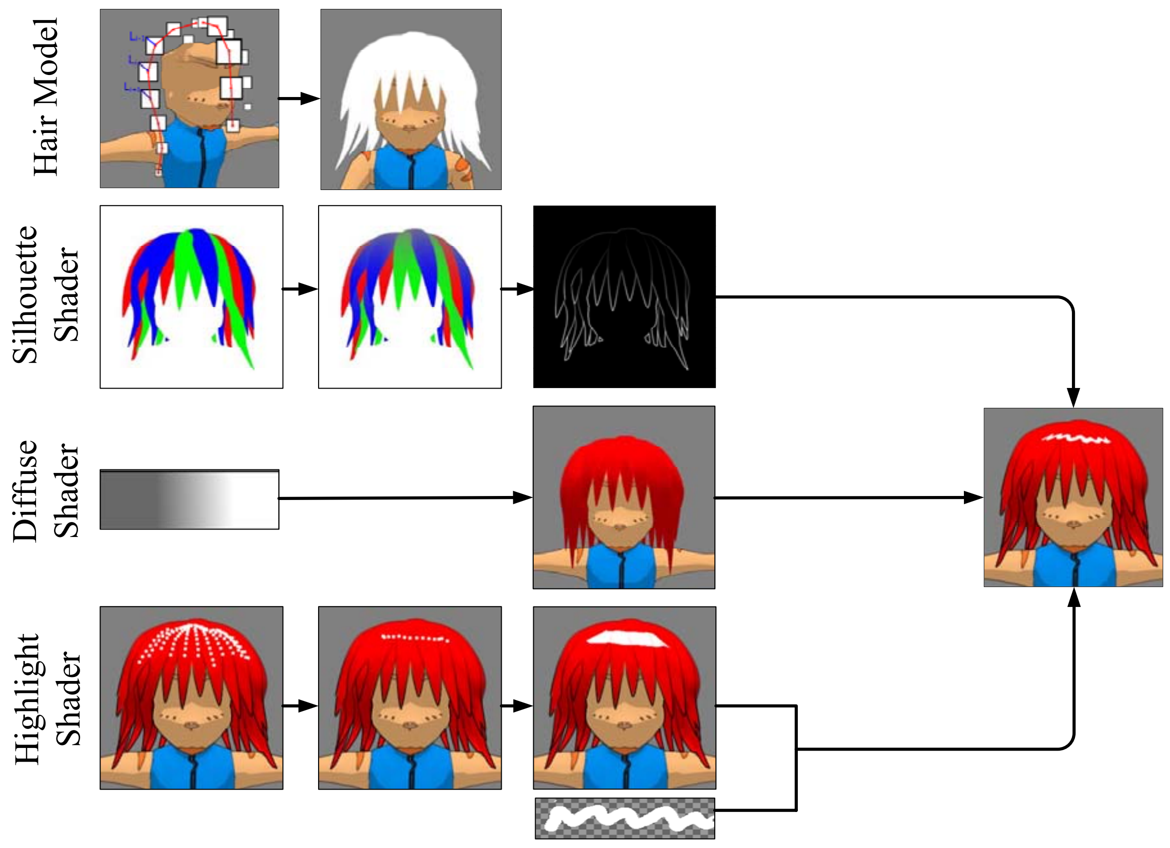


Figure 5.2: Overall Rendering Pipeline

5.1 Hair Model

5.1.1 Mass-Spring Based Hair Model

A mass-spring based hair model is often used in hair simulation since it is efficient and easy to implement. In this study, we use the mass-spring based hair model for two reasons. First, more specific information of hair is needed to improve the quality of the shading. We calculate the tangent of hair strands from the model (section 5.1.2) and use it for the specular highlight (section 5.4.2). Second, the mass-spring based model allows hair to be animated easily. In this study, each hair strand's physical characteristics are separately simulated. Simple air friction, gravity and collision detection with a sphere that represents the head is implemented. This is used when the hair model is initialised and animated.

5.1.2 GPU based Particles

For the rendering of hair strands, we implemented a special particle system, where a single strand consists of multiple particles. With the user defined control points, in between points are generated using a Catmull-Rom spline (cf. appendix B.2). Alpha blended particles are placed at the generated points as screen-aligned billboards. Thus, our approach generates a sparse-hair model using different particles that are connected together by using a Catmull-Rom spline (See figure 5.3). The points $p_1..p_n$ are user-defined and match the character's head.

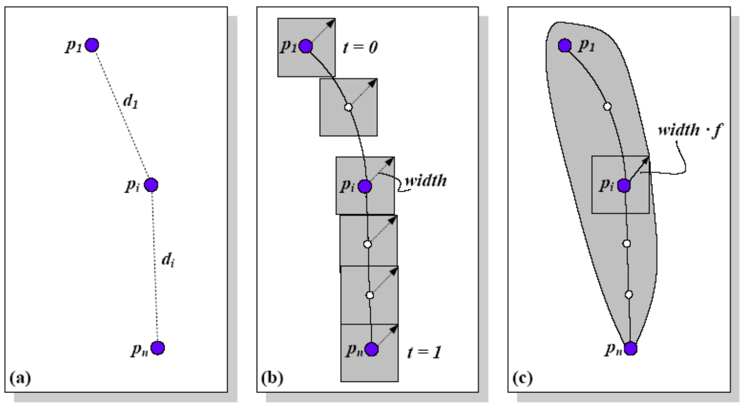


Figure 5.3: Creation of a hair strand: With user-defined particles (a), we define a Catmull-Rom spline on which the billboards are generated (b). Finally, the width of the billboard is re-sized to guarantee a better shape for the strand (c).

All generated particles are potential centers of the billboard particles that generate the "surface" of the strand. Thus, these particles will be used later to position the billboard texture that composes the strand. Next, the billboard vertices have to be generated. Similar to [10], this task is performed on the GPU: The coordinates of the center of the billboard, which is the coordinate of the particle, is sent to the vertex program four times, accompanied by the correct texture coordinates needed for the strand texture. These two coordinates can be used in combination to create each corner of the billboard.

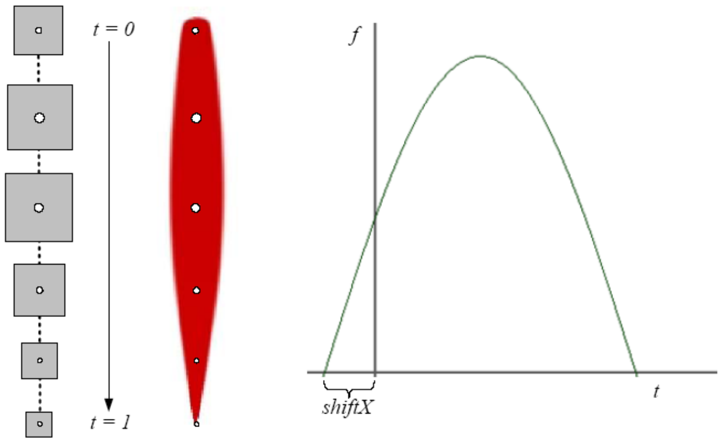


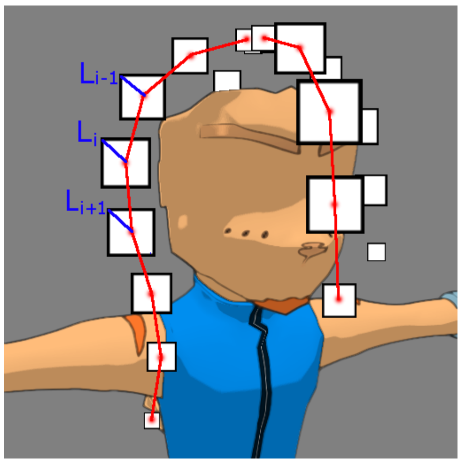
Figure 5.4: The billboard particles get re-sized by the factor f.

Finally, the billboard size (with its user-defined width) gets scaled by the factor

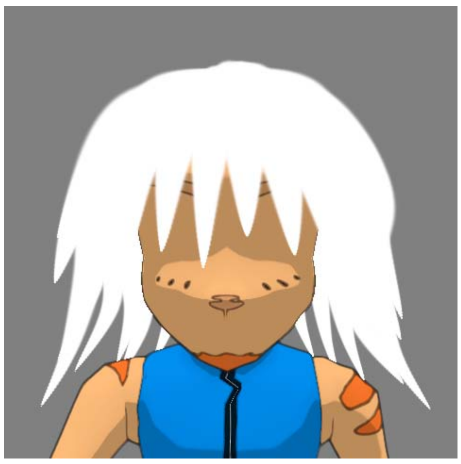
$$f = \sin\left(\frac{(t + shiftX)\cdot\pi}{1 + shiftX}\right) \tag{5.1}$$

where t ranges between 0 and 1. The user defined value $shiftX$ is the shift of the sine curve along the x-axis (See figure 5.4). The value $shiftX$ has been initialised to 0.2 by default. This is to achieve the shape of a hair strand shown in figure 5.4. As the result, the size of the hair-clump billboard gets scaled as depicted in figure 5.5. Different formulas can also be used to change the shape of the hair strands.

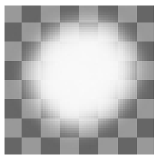
Since the hair strands do not have a real geometry (as proposed by [22]), and are defined by billboard particles, we have to calculate the normals for further shading. Figure 5.6 shows how both the tangents and the normals are calculated for each billboard particle.



(a) The simplified presentation of the billboard model.

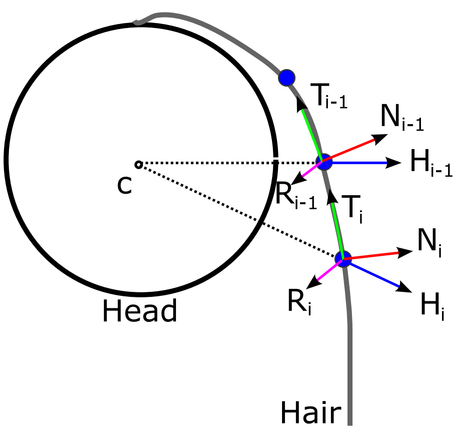


(b) The hair model consists of 2,500 billboard particles.

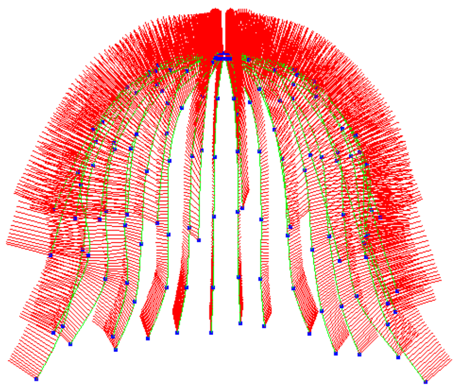


(c) Particle Texture used for (b)

Figure 5.5: The Hair Model



(a) Simplified presentation of the normal generation



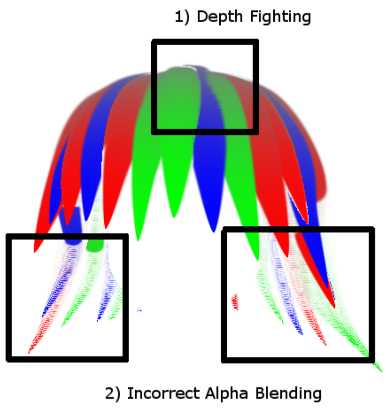
(b) Generated normals

Figure 5.6: Calculating the normal vector: (1) The tangent vector and the vector with its origin in the head's center are used to calculate the right vector. (2) The cross product of the right vector and the vector T_i are than used to calculate the normal vector.

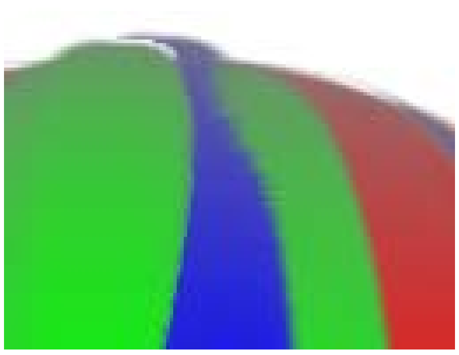
The user defined particle's position is denoted by $p_i(i = 1, 2, .., n-1)$. Obviously, the normal of a particle has to face away from the center of the hair model. Let C be the center of the hair model and H_i is the normalized vector facing away from C the center of the head. H_i, however, cannot be used as a normal vector for the billboard particle, because the particles do not necessarily have to be aligned along the head's surface. We, therefore, compute the tangent vector T_i which is simply calculated as $p_{i-1} - p_i$. To get the correct vector, we then calculate $R_i = T_i \times H_i$. Notice that the vector H_i has its origin in the head's center and is always pointing to the surface of the billboard (See figure 5.6(a)). Finally, the normal vector N_i is calculated by the cross product of the normalised tangent vector T_i and the normalised right vector R_i (See figure 5.6).

5.2 *Sorting the Hair Strands*

After discussing the creation of the particles and the billboards, this section mainly focuses on the rendering order of the billboards.



(a) Depth fighting and alpha blending problem



(b) Zoomed image of area 1)

Figure 5.7: Issues with Z-Buffer test

The alpha blending of particles causes problems in combination with the Z-Buffer (depth) test (See figure 5.7). The alpha blending between the particles gets disturbed unless particles are sorted manually according to the eye position. Although the shape of particles and the depth of particles can be rendered separately, it still causes z-fighting effects. This is because the distance between the particles is really small. In our approach, we deactivate the Z-buffer test.

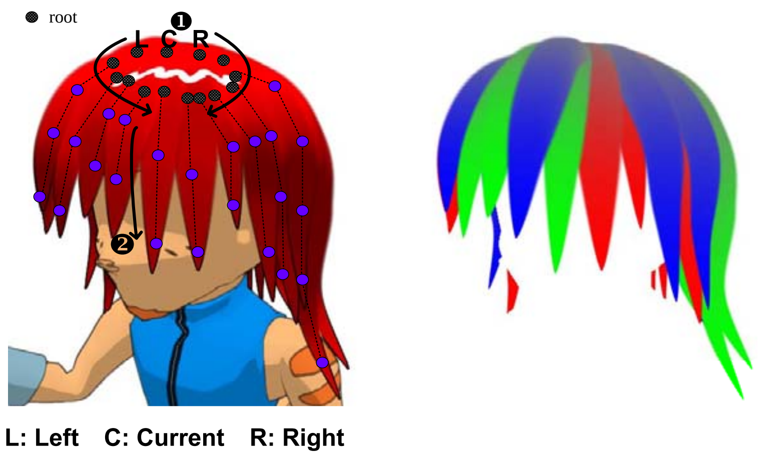


Figure 5.8: (a) The rendering order of the particles depends on the eye position. (b) Relative depth order between the neighbors of the current hair strand can still influence the final rendering order of the hair strands.

Therefore, the rendering order of the strands is very important to solve the depth issues and the root of the hair strands plays an important role in determining this. We sort the hair strands according to the distance from the eye position to the root position of hair strands (See figure 5.8(a)) before rendering the individual particle billboards. This prevents hair strands at the back of the head from covering hair strands at the front of the face.

In addition to the depth sorting, we also add a "relative" depth value, which is applied to the different hair strands. This is applied randomly at the beginning of the application and influences the final rendering order of the individual hair strands. We add the "relative" depth to achieve the typical Anime style of hair, a hair strand is covering neighbor hair strands (See figure 5.1). The relative depth level (See figure 5.9(a)) is coded in RGB-colour values with three depth values (R = back, G = middle, and B = front) and may change the rendering order of the hair strands. Therefore, before the current strand is rendered, the depth level of the two neighbors (the left and the right neighbor) have to be checked. The neighbors are determined by the root position of hair strands when the hair model is initialised. If the left or right neighbor is relatively behind the current hair strand (e.g. the current hair strand's colour is green, but the right hair strand's colour is blue), the neighbor strand has to be rendered first. The pseudo-code for the recursive rendering algorithm can be described as follows:

```
for each hair strand
    RenderRelativeOrder( current strand );
RenderRelativeOrder( strand ) {
    if ( strand.isRendered())
         return;
    if ( strand.left.level < strand.level)
         RenderRelativeOrder(strand.left);
    if ( strand.right.level < strand.level)
         RenderRelativeOrder(strand.right);

    render(strand);
}
```

Finally, some of the hair strands get occluded by the body's geometry (e.g. the head). The depth test between the hair billboards and the body is performed in the fragment shader. In a first step, the depth of the particles are calculated using the following formula.

$$ParticleDepth = ((particle\_centerPosition.z/particle\_centerPosition.w) + 1.0f)/2.0f \quad (5.2)$$

Here *particle_centerPosition* is the center position of the particle and *particle_centerPosition.z*, *particle_centerPosition.w* are the z and w coordinate values of the particle position.

The reference depth map of the body is forwarded to the fragment shader, where the depth test is performed. The particle billboard has only been drawn if it is in front of the 3D geometry. Thus, hair strands in the back of the head are not drawn.

```
frag2app FPshader(vert2frag IN,
                  uniform sampler2D depthRef) {
  frag2app OUT;
  ...
  refDepth = tex2D(depthRef, IN.texCoordScreen);

  // do not draw the particle if it is behind the
  // phantom geometry (e.g. head)
  if ( IN.particleDepth < refDepth ) {
    OUT.color = ParticleColor;
  }
```

```
    ...
}
```

As a result, the hair strand billboards are sorted according to their depth and relative position to their neighbor strands, and rendered into a texture for further use in the fragment shader. Note that the hair strands are rendered twice: first for creating the silhouettes and second for the diffuse lighting of the hair strands (section 5.4.1).

5.3 Rendering the Silhouette

Using the rendering order of the hair strands and the reference image, we can easily calculate a silhouette of a hair strand model. By applying a Sobel edge detection filter on the reference texture, the necessary data can be found for constructing the silhouettes (See figure 5.9(b), appendix A.3.1).

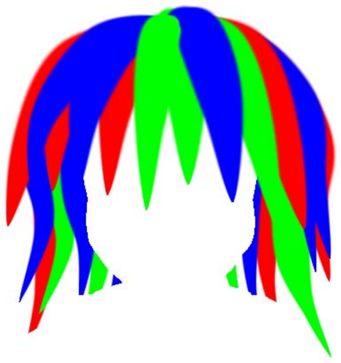
As described before, we use a two-step re-arranging approach for sorting the hair strands (in the first step, the strands are sorted from the back to the front of the head; in the second step, the hair strands can be slightly re-arranged according to the relative depth of their immediate neighbors). This re-arrangement of hair strands can be causing slight "jumping" effects especially near the root of the hair since the relative depth is applied only to its neighbors. This could be minimised by selecting proper width for the strands and proper relative depth.

However, a better solution is by fading out the intensity of the silhouettes which are close to the root of the hair by using a fading function. The intensity of the reference image is modified using the following function.

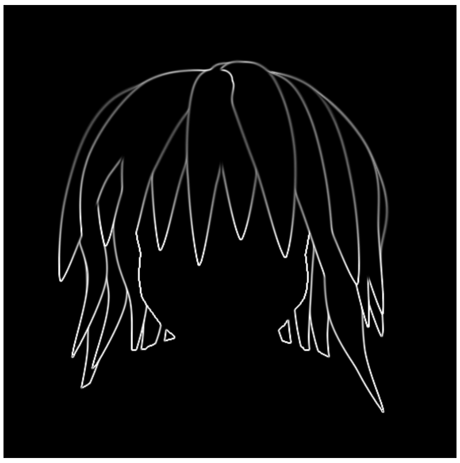
$$f = \frac{1.0 - cos(t \cdot \pi)}{2.0} \qquad \text{for all } t \in [0,1] \tag{5.3}$$

The above equation gives smooth intensity transition along a hair strand.

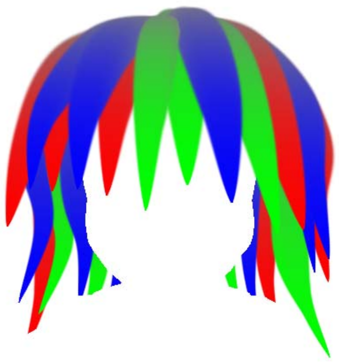
Again, the variable t represents the value from the first to the last particle of a single hair strand. Figure 5.10(c) shows the modified reference image and figure 5.10(d) shows the result silhouettes. As the "jumping" effects are most disturbing on the top of the head, we simply fade out the hair strand on the root of the hair. The results are shown in figure 5.10. Figures (a) and (b) demonstrate how the silhouettes get changed, especially on the top of the hair caused by the re-arrangement of the hair strands. In contrast, this effect can be hardly recognised by using the fading function (See figures (c) and (d)).



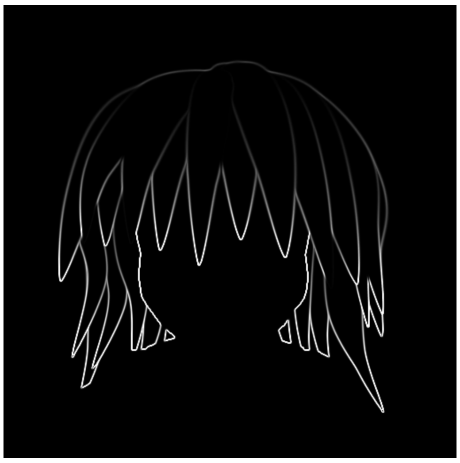
(a) Original Reference image



(b) Silhouettes obtained from (a)

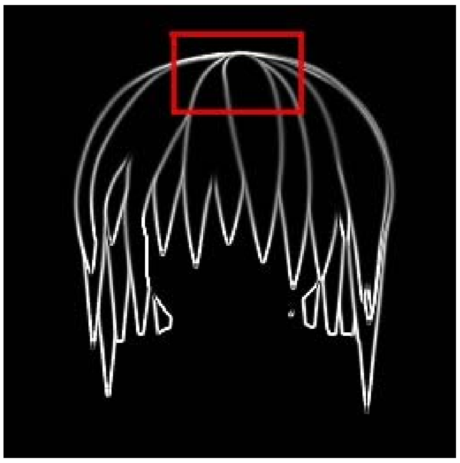


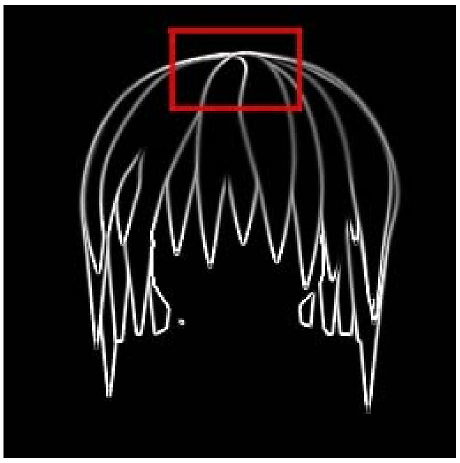
(c) Modified Reference image

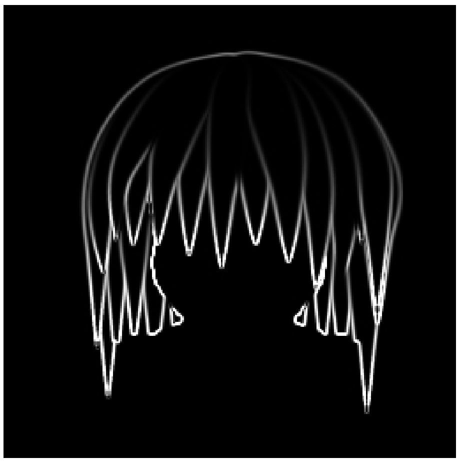


(d) Silhouettes obtained from (c)

Figure 5.9: Comparison between the original silhouettes and intensity modified silhouettes. The silhouette intensity is changed by modifying the reference images.


(a) Original Silhouettes


(b) Original Silhouettes at different eye position


(c) Intensity modified Silhouettes

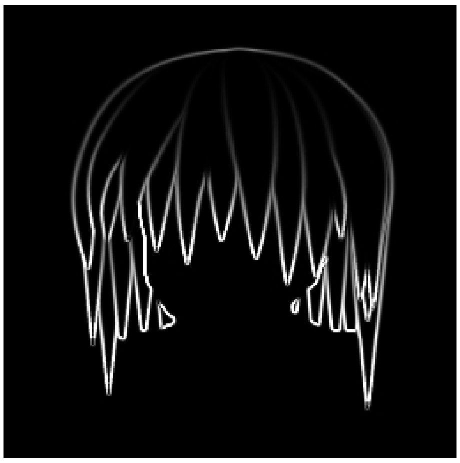

(d) Intensity modified Silhouettes at different eye position

Figure 5.10: Hair Silhouettes

5.4 Shading

5.4.1 Diffuse Lighting

Simplification of geometry and shading is important for generating Anime characters. Similar to the silhouettes, we use the reference image generated in the first step of the pipeline. In contrast, the order of how the particles are rendered within one hair strand is important for achieving a nice diffuse lighting effect. Therefore, the billboard particles need to be rendered from the furthest to the closest according to the eye position. To achieve a better performance, we simply sort the hair strands according to their distance from the eye position to the root and tip of the hair strand. The pseudo code shown below uses the following notation: d_{root} is the distance between the eye position and the root of a single hair strand. d_{tip} is the distance between the eye position and the tip of a single hair strand. The function `addToList` adds the actual hair strand to the corresponding `rootList` or `tipList`.

Algorithm 1 Hair Strand Sorting Algorithm

```
for $i = 0$ to $n - 1$ do
    $d_{root} \leftarrow$ distance(EyePostion, RootCurrentHairStrand[i])
    $d_{tip} \leftarrow$ distance(EyePosition, TipCurrentHairStrand[i])
    if $d_{root} \leq d_{tip}$ then
        addToList(rootList, hair-strand[i])
    else
        addToList(tipList, hair-strand[i])
    end if
end for
```

After sorting the hair strands, we can render the particles. First, the colour of the body's reference image is rendered, then the particles are rendered. Again the depth test is done in the fragment shader. However, diffuse shading causes a problem (See figure 5.11(c)). The combination of the Painter's algorithm with a step-function for the texture generates unwanted shading effects. Therefore, we used a different texture to generate the diffuse lighting (See figure 5.11(a)).

5.4.2 Specular Lighting

In [3], Anjyo and Hiramitsu present a novel highlight shader for cartoon rendering and animation. The specular highlight is an essential part of the Anime hair shading. In Anime, the specular highlight of the hair is mostly used to show the volume of the hair and it does not always show

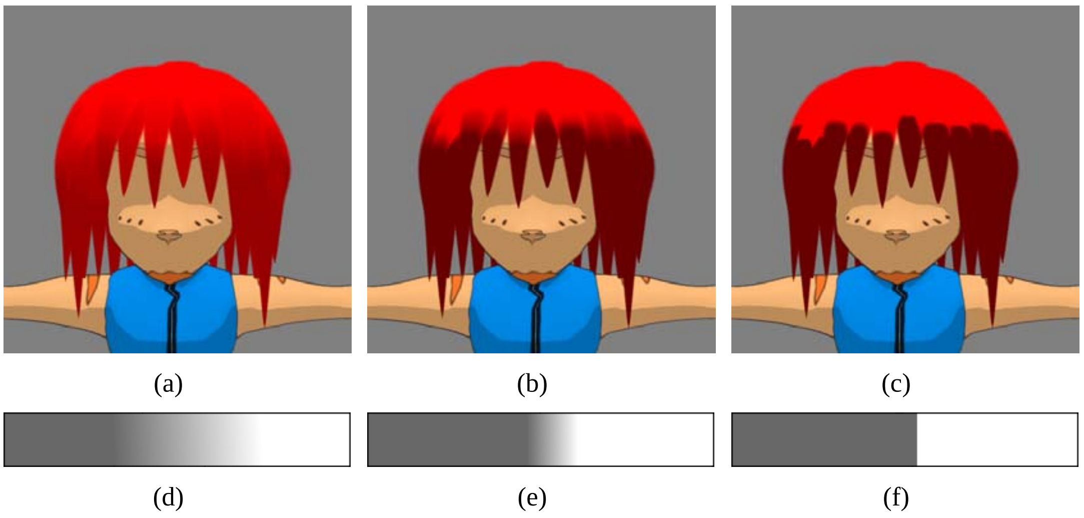


Figure 5.11: Diffuse Lighting with Texture Lookup

the accurate shininess of hair. It is just approximated areas of highlights. There are many different styles of cartoon specular shapes and they are usually exaggerated and simplified. The cartoon specular does not vary much depending on the eye position. Therefore, we recognised that in most characters, the current specular highlighting model cannot be used.

Instead of using the traditional Blinn's specular model [2], we propose a new highlight shader for the 3D object. Instead, our specular term is composed by using the tangent vector T and the normal vector N of the hair strand billboard:

$$specular = K_S \cdot lightColour \cdot (max(L \bullet W, 0))^{shineness} \tag{5.4}$$

where K_S is the reflection coefficient, L is the direction towards the light source, and W is an affine combination of the normal vector N and the tangent vector T. The vector W can be expressed by the vectors N and T, and the weight value w:

$$W = N \cdot w + T \cdot (1 - w) \tag{5.5}$$

Figure 5.12 illustrates the specular model. Note that the equation 5.4 is obtained by modifying the Phong-Blinn specular model:

$$specular = K_S \cdot lightColour \cdot (R \bullet V)^{shineness} \tag{5.6}$$

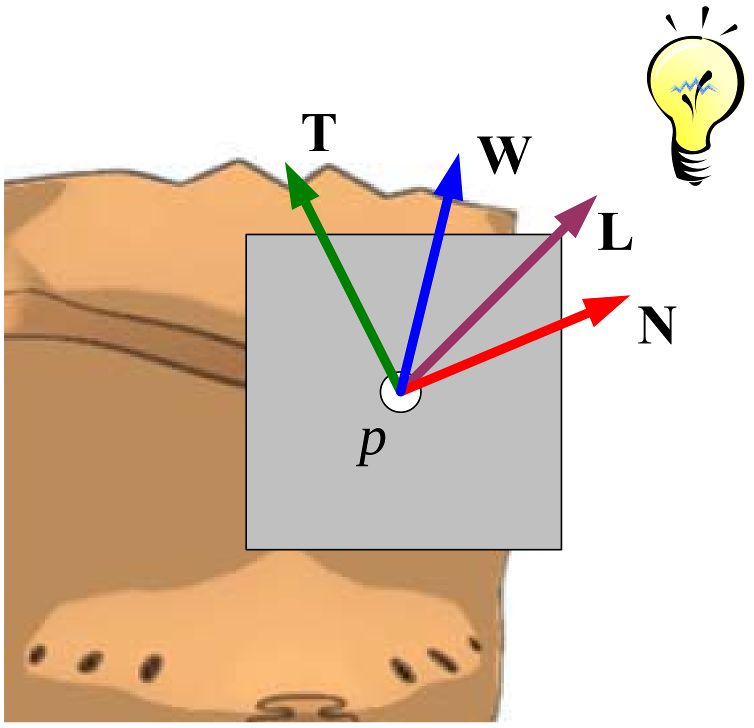


Figure 5.12: Calculating the specular term.

where R is the specular direction and V is the view vector.

The new specular term is introduced since the traditional specular model does not suit the specular highlights of the Anime hair. The specular highlight of the hair in Anime is an exaggeration to show the volume of the hair. It means the specular highlight may not be mathematically correct all the time. Therefore users should be able to change the position of the specular highlight. Figure 5.13 shows how the user defined weight value is influencing the final results. The light source is placed above the head. As a result, the highlight can be moved from the tip to the root of the hair strands by simply changing the weight value from 0 to 1.

User defined textures are used to achieve exaggerated, simplified, and cartoon style highlights. The specular hair model has its own structure which is the same as the original hair model but containing less points. Figure 5.15 shows the steps to generate a stylised specular highlight.

Here we explain the merging and linking of specular points. Our algorithm iterates over all strands and removes all particle links that have a larger value than the user defined distance to each particles (See figure 5.14(a)). Consequently, the particles of a single hair strand get merged (which is the average value of the particles of a single group) into one single particle (See figure

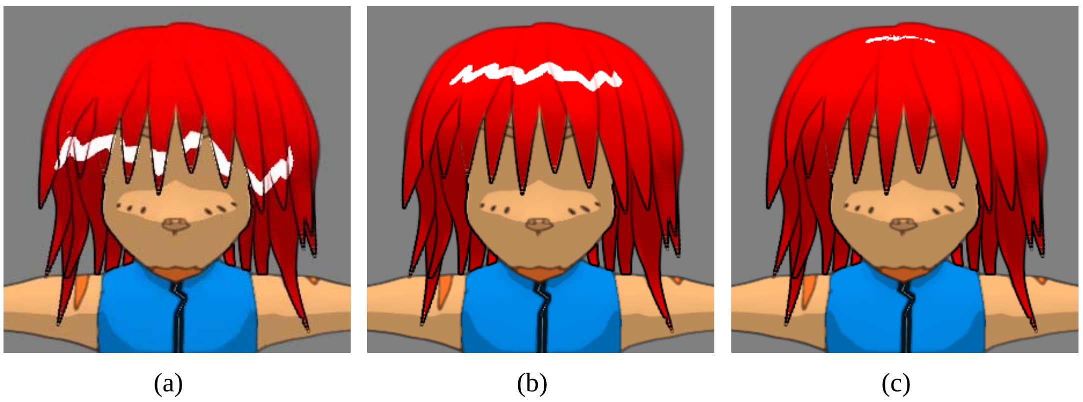


Figure 5.13: The different weight values (a) *weight* $= 0.0$, (b) *weight* $= 0.5$, and (c) *weight* $= 1.0$ can influence the position of the specular highlighting.

5.14(b)). Depending on the user defined threshold and the distance between the single particles, one or more groups are generated. Finally, we render the highlight textures as a series of triangle strips which are composed by the merged (averaged) particles (See figure 5.14(c)).

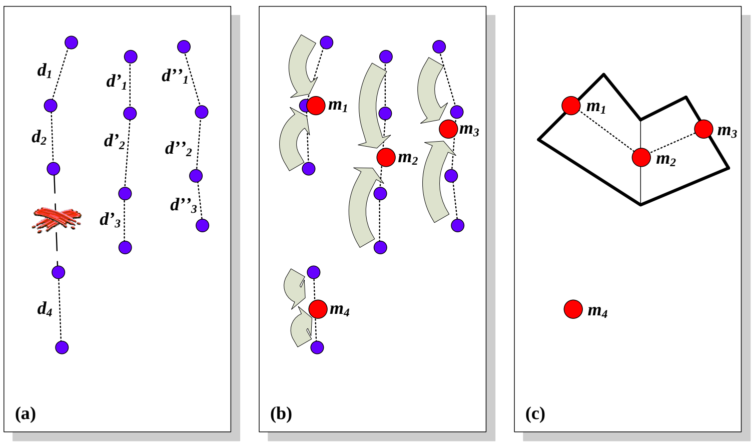
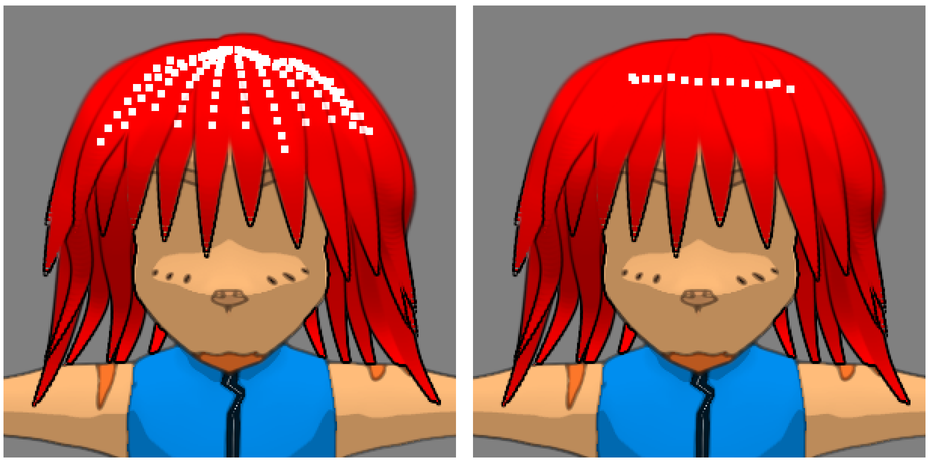


Figure 5.14: The pipeline of the specular highlight: after merging the particles (p_i), we achieve potential points m_i (b) which are representing potential points for creating the triangle mesh (c).

Notice that the linked specular texture needs more than just one texture depending on the amount of particles that are connected to generate one highlight. By using a single texture, the specular highlight gets stretched or squished which results in unwanted artifacts. Figures 5.17(c) and 5.17(d) show two example results of multiple specular highlight links with different lengths.


(a) (b)

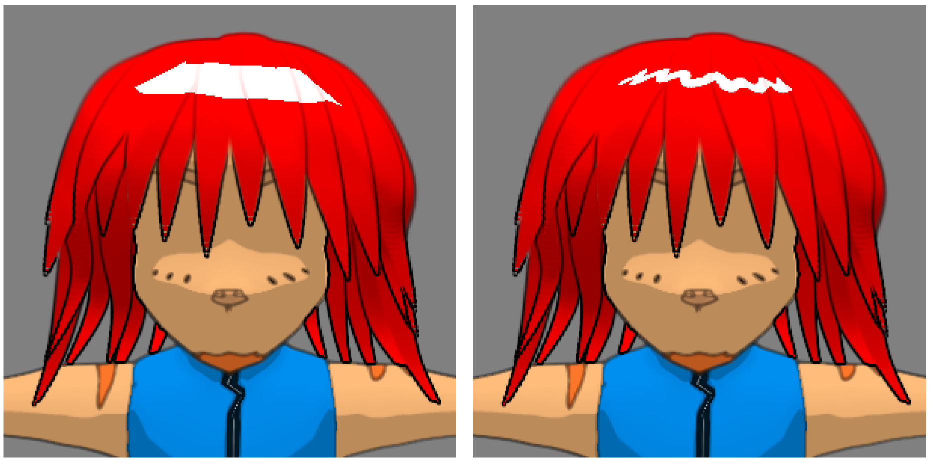

(c) (d)

Figure 5.15: The different pipeline steps for generating the specular highlight: firstly, all particles are marked with a special specuar highlight threshold (a). Potential particles are merged (b) and define the triangle strip (c), which is used for rendering the highlight texture (d).

The different thresholds are defined by the user and the threshold value is simply the minimum distance between the particles horizontally and vertically. The advantage is that the modeler can tweak the highlight to get great results. Users can change the style of the specular highlights by applying different textures (See figure 5.16).The disadvantage is that it may require too many user inputs (e.g. threshold value for the specular value, the minimum distance between merging points, and the linking of different textures) to generate nice renderings.

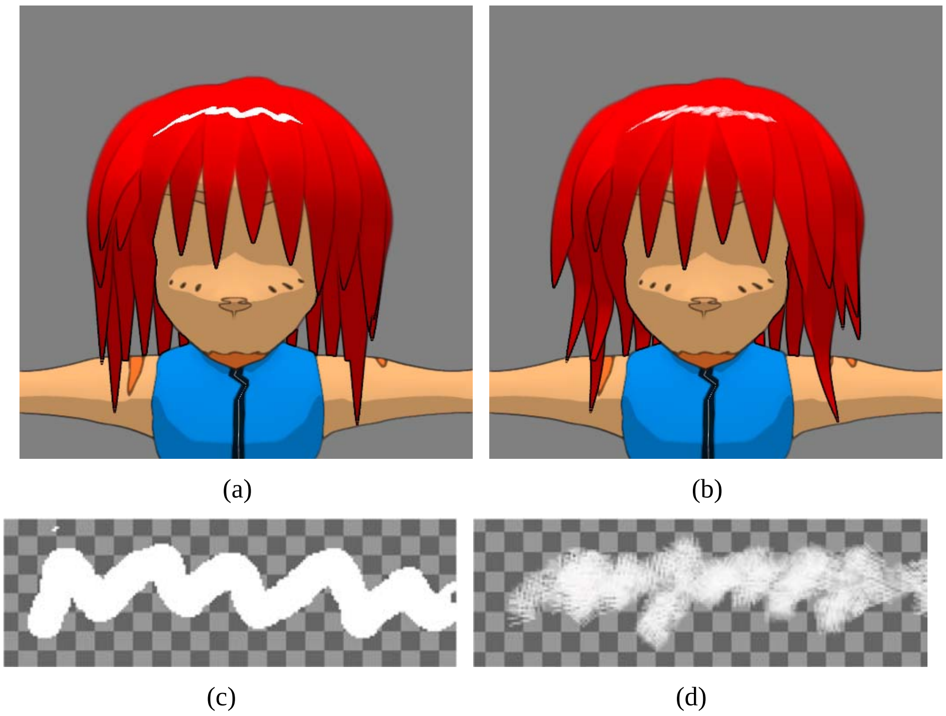


Figure 5.16: Stylised, specular highlights with the corresponding textures.

5.5 Hair rendering result

Figure 5.17 shows the hair rendering results with some background images. Users can easily change the properties of hair to achieve the different renderings. Users can change the intensity of silhouettes, the style of specular highlight and its position. Furthermore, users can achieve stylised multiple specular links (See figure 5.17(c), 5.17(d)).

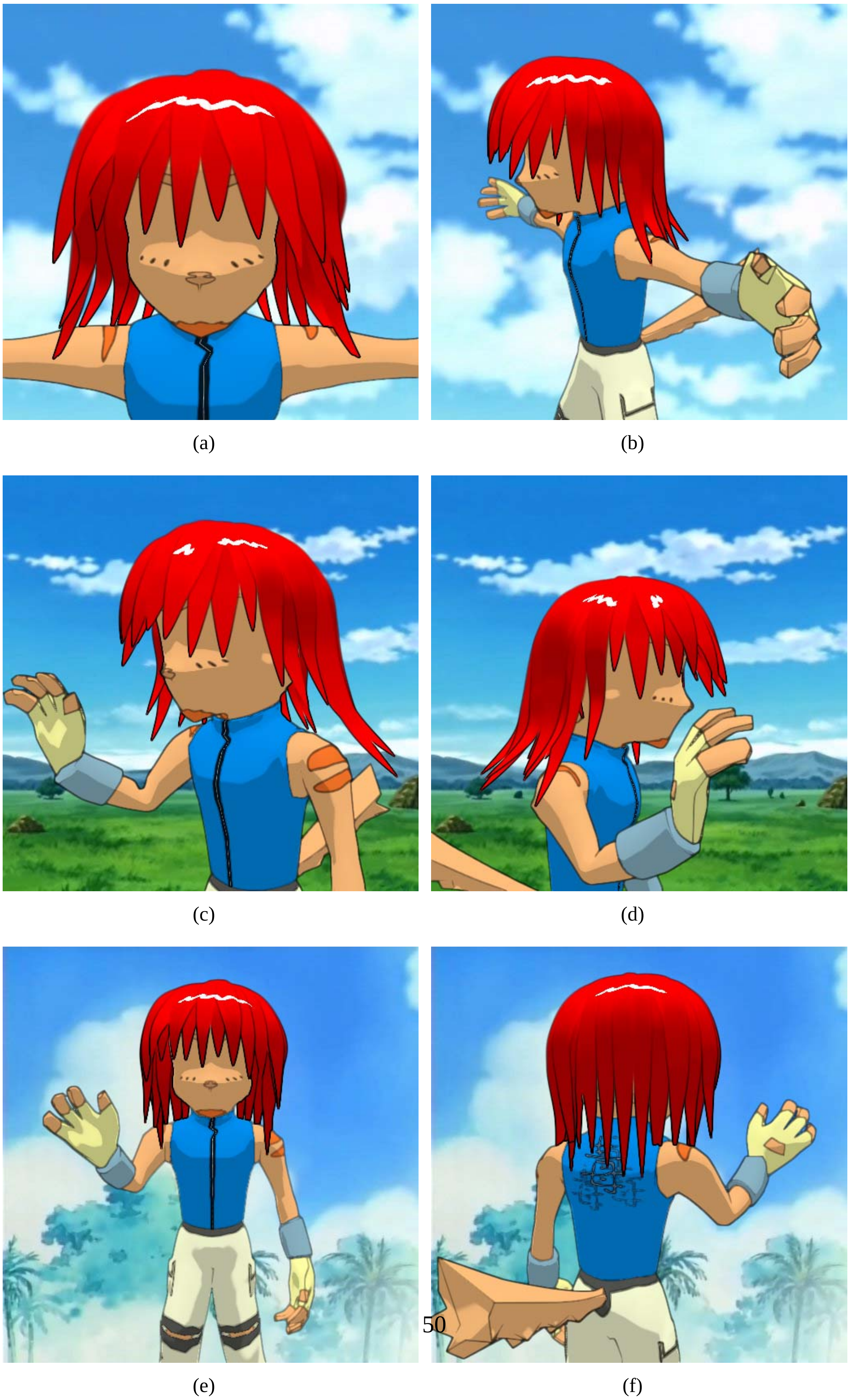


Figure 5.17: Results

Chapter VI

Implementation Aspects

This system is implemented in C++ using the OpenGL API [41] and nVidia's CG shader language (cf. appendix A). We tested our setup on a Pentinum 4, 2.60 GHz, and an nVidia GeForce 6800 graphics card with 256 MB of memory.

It is very difficult to evaluate the performance of this system since the performance depends on the user inputs. Note that we used a hybrid of 3D and 2D based techniques. The performance not only depends on the complexity of the 3D input geometry but also depends on the complexity of the features in the rendered G-Buffers in 2D spaces.

For the artefact reduction process, the complexity in 3D mainly depends on the number of polygons in the input geometry and the resolution of the input textures. However, there are many factors that change the performance of the 2D image-based techniques. The major factors include:

- the resolution of the G-Buffers
- the complexity of the input textures (the texture that is mapped on the input geometry)
- number of normal and depth discontinuity features on the input model
- the user defined threshold values for controlling the silhouettes
- the amount of projected (rendered) features in the 3D model stored in the G-Buffers

Obviously, the resolution of the G-Buffers affects the performance, since they are the inputs of image processing algorithms. As the size of the G-Buffer increases, the performace decreases accordingly. The most expensive process in this phase is the Bezier curve fitting of the silhouettes since it is performed by the CPU. If there are more silhouettes detected, it takes more time to reconstruct them. The G-Buffers, however, are rendered every frame and they change depending on the view position and the view angle. The number of silhouettes, and the number of pixels in the silhouettes image, can change rapidly depending on the visibility of the intensity features, as well as the surface normal and depth discontinuity features. The silhouette also changes depending on the threshold values for the silhouette intensity that the users control. If the viewer is far away from the model, the projected model is small in the G-Buffer. Therefore the number of visible silhouette pixels projected in the image space is small and vice versa.

6.1 Performance case study: ATI draw buffers

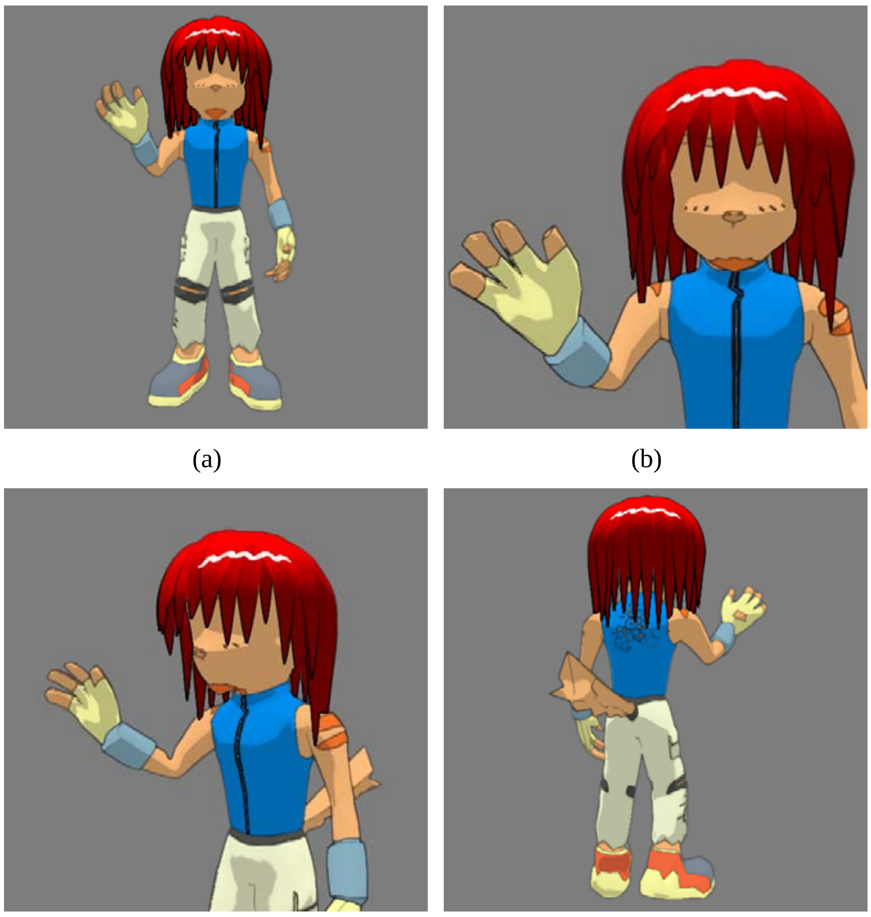
To run our system, a graphics card with pixel shader support is necessary. Some recent graphics cards, however, such as nVidia's G-force6 series (or equivalent graphic cards) support ATI's draw buffers extension which is also known as Multiple Render Targets (MRT). ATI draw buffers allow rendering to multiple draw buffers simultaneously. Instead of outputing a single colour value, a fragment program can output up to four different colour values to four separate buffers (cf. appendix A.5).

Therefore we re-implemented the system in order to optimise performance. The ARB render texture (RT) extension is replaced with the Framebuffer object extension (FBO) and the ATI draw buffers extension is used to render multiple G-Buffers together if possible. Replacing the ARB render texture extension with the Framebuffer object did not give any visible performance improvement, however, using the ATI draw buffers gave a significant performance improvement.

The performance of rendering the images in figure 6.1 were measured. The input 3D model contains 4245 polygons and the resolution of the texture mapped on the model is 1024x1024. The G-Buffer has a resolution of 512x512 data elements. The hair model was composed of 3753 particles and the number of sample points used for the specular highlighting contained 738 particles.

Table 6.1 shows both the model and the hair rendering performance are improved significantly, through the use of the new extensions. The rendering performance of the model is improved by 9.84 – 20.31% and the rendering of the hair is improved by 29.73 – 37.94%. The overall improvement is 27 – 33%.

The input model had to be rendered three times and the hair model in the old system needed to be rendered twice. The performance, however, does not double or triple and that is because, even though multiple buffers are rendered simultaneously, the same amount of information, such as the colour, textures, surface normals, the light position etc, needs to be passed to the vertex and fragment shader in the new system. The hair rendering is significantly improved since the old system, which had to render the hair model twice (7506 particles) where the new system only needs to render once (3753 particles).



(a) (b) (c) (d)

Figure 6.1: Rendered Images

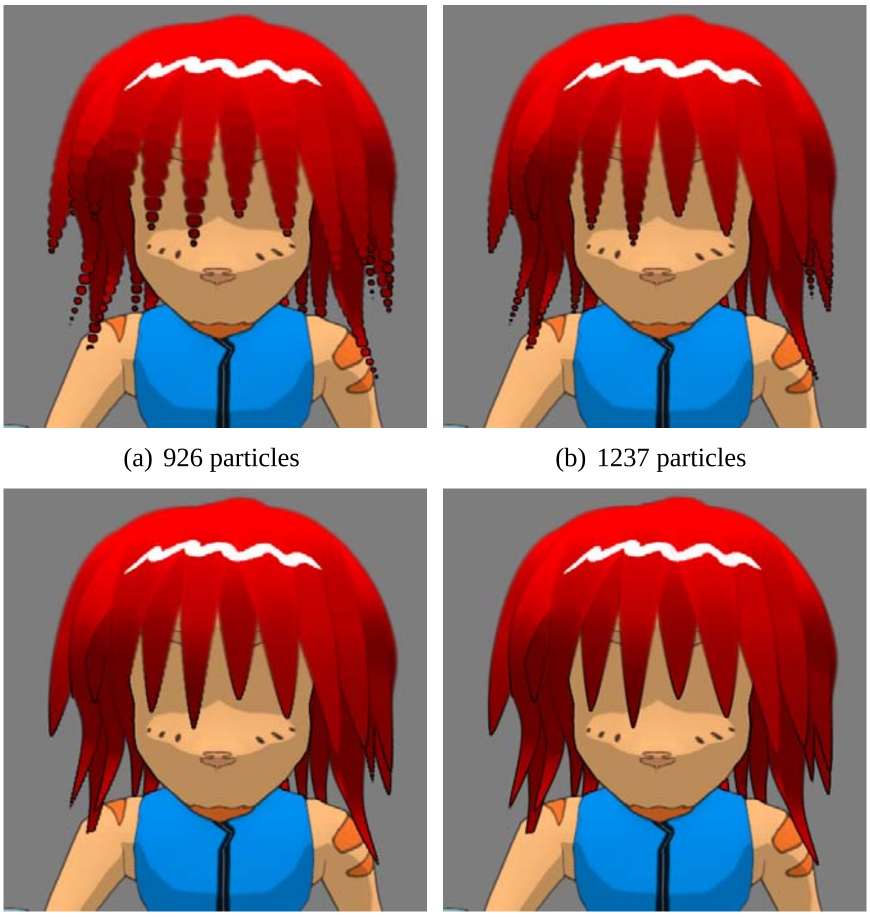
In the next section we compare the performance of the hair rendering depending on the number of particles.

figure 6.1(a)	Model	Hair	All
ARB render texture	5.992	2.200	1.816
Framebuffer object with ATI draw buffers	6.658	2.854	2.308
Improved %	11.11%	29.73%	27.09%
figure 6.1(b)	Model	Hair	All
ARB render texture	5.456	2.129	1.712
Framebuffer object with ATI draw buffers	5.993	2.854	2.220
Improved %	9.84%	34.05%	29.67%
figure 6.1(c)	Model	Hair	All
ARB render texture	5.461	2.101	1.712
Framebuffer object with ATI draw buffers	6.385	2.854	2.282
Improved %	16.92%	35.84%	33.29%
figure 6.1(d)	Model	Hair	All
ARB render texture	5.456	2.069	1.705
Framebuffer object with ATI draw buffers	6.564	2.854	2.267
Improved %	20.31%	37.94%	32.96%

Table 6.1: The performance of rendering the images in figure 6.1 using different extensions. Model: the rendering performance of the character's body only, Hair: the rendering performance of the hair model only, All: the performance of both the model and hair. The measurement is in frame per seconds.

6.2 Performance case study: Particle number comparison

The hair rendering performance mainly depends on the number of particles. Figure 6.2 shows the rendered results of the same hair model with different numbers of particles. In figure 6.2(a), the hair model contains too few particles and the hair strands are not smooth. As the number of particles increase, the quality of rendering increases and the performance drops. Figure 6.3 shows the performance of the hair rendering. Note that the performance results are with the new system only (MRT extension version).


(a) 926 particles (b) 1237 particles

(c) 1868 particles (d) 3751 particles

Figure 6.2: Quality of the hair rendering depending on the number of the particles

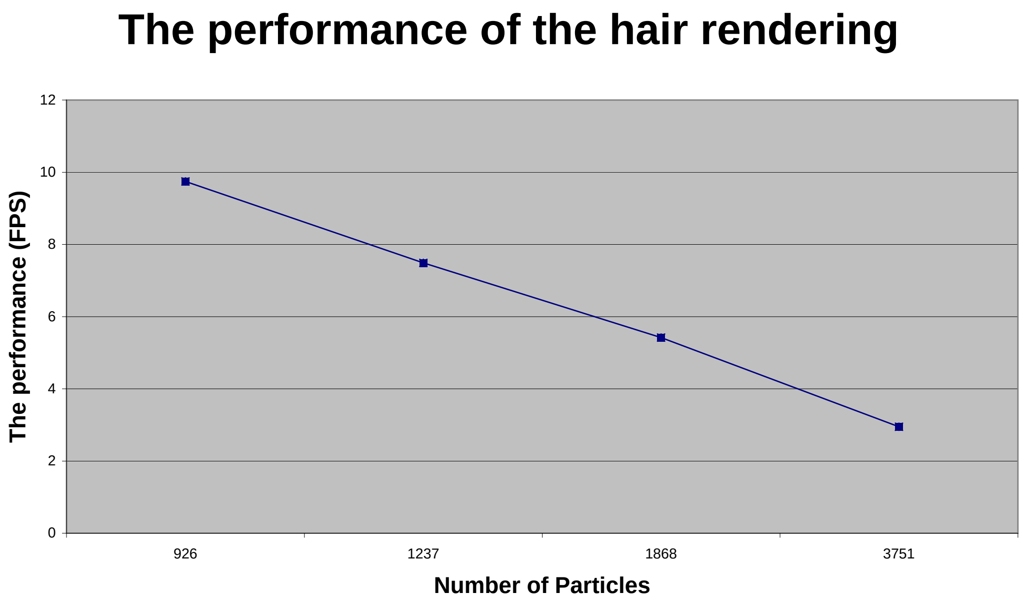


Figure 6.3: The hair rendering performance with different number of particles

Chapter VII

Conclusions and Future Work

In this study, new rendering techniques for cartoon characters have been presented. These include techniques for reducing computer generated artefacts and a novel rendering method for cartoon based hair models. The silhouette of the input geometry is improved through Bezier curve fitting and modifying the silhouette intensity. The quality of the rendered images are enhanced by applying various image-based algorithms. Moreover, we demonstrate an efficient method for rendering stylised hair in Anime style using a special hair model with particles. We also present a lighting model of hair using diffuse and the specular highlighting which are suitable for Anime style cartoon. The main goal of this study was to improve the rendering quality of cartoon characters. This was achieved by using enhanced silhouettes, smoothed shading and stylised hair rendering.

We have demonstrated different methods of applying and controlling the intensity of silhouettes in this study. The silhouettes, however, are found by applying edge detection on the G-Buffers, and the width of the silhouette mainly depends on the G-Buffers. Therefore it is hard for users to change the width of the silhouettes unless the input itself is modified. Also, the filters used to minimise the computer generated artefacts are applied uniformly to the G-Buffers, however the user may want to apply different filters non-uniformly.

The hair model with particles could be further improved. The main issues are with the depth between particles. In this study, the hair strands are partially sorted to avoid depth problems. Therefore it suffers the same problem as the Painter's algorithm, in that the hair model does not work for complicated hairs.

More research needs to be done to improve the quality of toon shading further. The following sections explain possible improvements of the current system and possible future studies.

7.1 Artistic level of detail

One of the main reasons why the rendered images are perceived as computer generated is the uniformness of rendering. Artists apply different levels of detail depending on the importance of the features. The current rendering method, however, renders all the features uniformly and it is hard to change the styles easily. Therefore, difference levels of detail need to be applied on different features depending on their importance.

Furthermore, an intuitive method is needed to assign the importance of features. An interface could be designed to let users select some area or volume in 2D screen coordinates or in 3D geometry space and assign importance values to the area. Then, the importance value would influence the width of the silhouette and also the parameters of the 2D filters in order to reduce the computer generated artefacts.

7.1.1 Stylising the rendering using Noise

In [26], noise (uncertainty values) textures are used to modify G-Buffers to stylise the rendering. The visual appearance is modified by controlling the uncertainty values. This approach can be used to achieve the artistic level of detail. The amount of noise is influenced by the importance values so that important features are not modified by the noise where less important features are strongly influenced.

7.1.2 Vector graphics

The varying width of silhouettes can change the style of the image significantly and break the uniformity. Therefore, the silhouette found from edge detection in raster graphics format needs to be reconstructed in vector graphics format. If the silhouettes are in vector graphics format, the width of the silhouettes can be easily changed and it is easy to modify the silhouette in 2D space with simple operations.

7.1.3 Stylised strokes of silhouettes

Even though many processes in rendering can be automated, the most important features of drawing must be strongly influenced by the artists. The simplest way is directly manipulating the input model. In [12], users can directly draw on the input geometry including the crease edge with WYSIWYG interfaces. To achieve the artistic level of detail, intuitive interfaces are necessary to apply different levels of detail.

7.1.4 Skeletons

Animation is one of the most important parts of toon shading, and skeleton animation is a common way of animating the 3D characters. The skeleton information can be utilised with each bone having their influence area, it would be intuitive for users to select different parts of the input geometry and set different importance values which will change the artistic levels of detail.

7.2 Improving hair model

There are many issues in improving our current hair model with most issues being caused by using particles. Particles are used to achieve the smooth shape of the simplified cartoon hair strands. However, the particles do not have real geometry, and cause problems with the depth testing. Another problem is with the diffuse lighting with the traditional discrete two-tone shading.

7.2.1 Particles with depth issues

Unfortunately, our current hair model does not allow twisted hair strands. This is because we solve the depth issues by sorting the hair strands. Similar to the Painter's Algorithm with teethed surfaces, we would have to split the individual hair strands. Reference geometry of the hair model is needed to use proper Z-buffer depth testing. A combination of particles with a polygon based reference geometry could be considered.

7.2.2 Particles and diffuse shading

A hair strand is a linked list of particles in our current hair model. Therefore the discrete two-tone shading leaves the shape of a particle. This is because the diffuse shading purely depends on the particles which do not have real geometry. Constructing the reference geometry can solve this problem as the proper diffuse shading can be achieved by rendering diffuse shading from the reference geometry and mask it with the particles to get a smooth shape.

7.3 Automatic hair animation

Hair animation becomes more problematic due to the demands of many animated features in cartoon productions. It is possible to generate natural hair animation with physics simulation as the hair model is based on the mass-spring model. The cartoon hair physics could be studied.

The hair of cartoon characters usually has its own shape and it does not change much. The hair, however, need to be animated in events such as during fast movement of the character, strong winds etc. Usually, the hair shape returns to the original shape after these events.

7.4 Optimisation

This study has mainly focussed on artistic styles in rendering human character models, and the current implementation is suitable only for offline rendering. The performance, however, could be further optimised.

7.4.1 Level of detail

The rendering of particles reduces the performance due to the large number of particles. In the current implementation, the hair model is rendered with the same number of particles regardless of the distance from the viewer to the hair model. However, when the hair model is far away from the viewer, a similar quality can be achieved with less particles.

7.4.2 Different hair strand models

Particles are used to achieve smooth outlines of hair strands. Using large numbers of particles, however, is not very efficient. Many different hair strand models could be considered.

- A hair strand model with a combination of polygon and particles in order to reduce the number of particles

- A purely polygon based hair strand model similar to graftals in [17]

7.5 Shadow and Transparency

This study did not consider how shadows and transparency should be handled in cartoon shading. The shadow is an important visual cue to understanding the environment. In cartoons, both the characters and the shadows are simplified and exaggerated.

7.6 Intuitive interfaces

7.6.1 2D feature addition

Many exaggerated and simplified features in cartoons are difficult to achieve through geometry-based techniques. For example, figure 7.1 contains many 2D cartoon features such as surprised eyes, a shining flare of the scissor, lines on the face which shows that the person is frustrated etc. These features are usually strongly influenced by the artists and their own styles. A simple but effective solution to achieve these stylisation is to use texture mappings with textures produced by the artists. The cartoon shading, however, is based on the 3D geometry and it would be necessary to have intuitive interfaces that allow users to easily add these 2D features and control them.

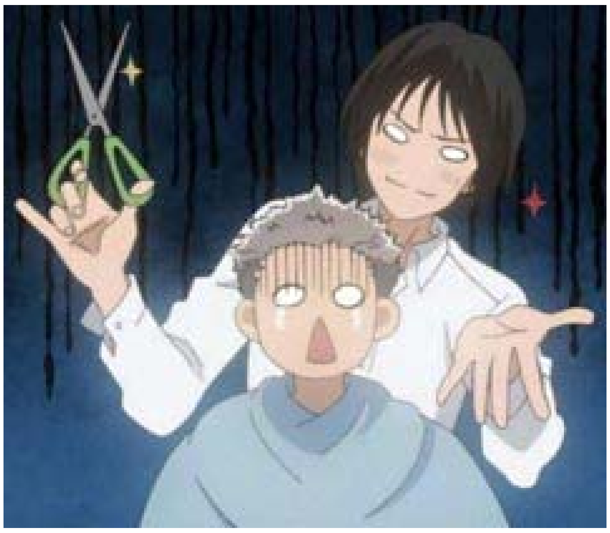


Figure 7.1: Exaggerations and simplification example [37]

7.6.2 Visual effect in 2D

Cartoons contain many 2D visual effects. Figure 7.2 shows that a character is shaded with a strong 2D light effect to show the brightness of the light source including lense flares. Many 2D effects are done manually in post production, however many steps could be automated. For example, let's assume that there is a 3D character and the character is waving his sword. If a user wants an aura around the sword, the path of visual effect can be automatically generated

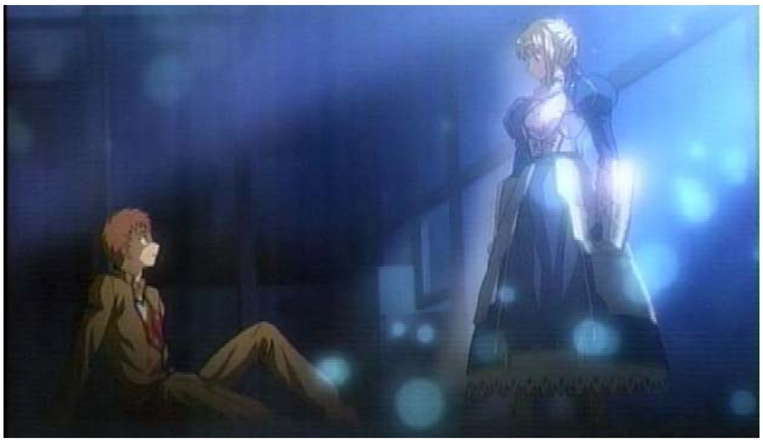


Figure 7.2: 2D effects example [35]

since the 3D position of the sword is already known. The users can map textures produced by the artists and produce animations of 2D effects easily. Therefore intuitive user interfaces are needed to map 3D information onto the 2D screen.

Chapter VIII

Acknowledgments

The author would like to thank Dr. Michael Haller of Upper Austria University of Applied Sciences for the local supervision and helpful discussions during the exchange programme, Dr. Mark Billinghurst for helpful discussions and proofreading of this thesis. The authors also like to thank Billy Chang for the great character models. The work is partly sponsored by the EU-New Zealand Pilot Cooperation in Higher Education under the project title “Leonardo”.

Chapter IX

Publications

A Stylized Cartoon Hair Renderer, Shin, J., Haller, M., Mukundan, M., Billinghurst, M., in ACM SIGCHI ACE 2006, ACM SIGCHI International Conference on Advances in Computer Entertainment Technology, Hollywood, USA.

The pdf version of the paper can be found at: `http://www.hitlabnz.org/fileman_store/2006-ACE-StylizedCartoonHairRenderer.pdf`

The pdf version of this thesis can be found at: `http://www.hitlabnz.org/people/jhs55/Thesis.pdf`

Appendix A

Cg Shader

Cg (C for Graphics) is a high level shader language created by nVidia for programming vertex and pixel shaders [8]. The syntax of Cg is similar to the programming language C. Cg can be used with two APIs, OpenGL or DirectX.

A.1 Cg and OpenGL

Cg library provides functions to communicate with the Cg program, like setting the current Cg shader, passing parameters, and such tasks. The followings are important functions for passing parameters.

cgGLSetStateMatrixParameter	passing a matrix to shaders
cgGLSetParameter*if*	passing float values to shaders
cgGLSetTextureParameter	passing textures to shaders

A.2 Data types and functions in Cg shader

Cg provides similar data types to C.

float	a 32bit floating point number
half	a 16bit floating point number
int	a 32bit integer
fixed	a 12bit fixed point number
sampler*	represents a texture object

float4x4 is used for a matrix, **float4** is used for a vector of four floats and **float2** is used for a texture coordinate etc. **sampler2D** is a 2D texture passed from the OpenGL API.

Cg provides many useful functions. **mul** and **tex2D** are used most often in this study. **mul** performs matrix by vector multiplication and **tex2D** performs 2D texture lookup with the given texture coordinate.

```
(eg.
float3 color = tex2D( myTexture, IN.texCoord).rgb;
float4 position = mul( myMatrix, oldPosition);
```

Cg also provides math functions such as **min**, **max**, **sin**, **normalize** etc.

A.3 2D Filters

In this study, filters are implemented using shaders. All filter operations are performed in the pixel shader, and both of the OpenGL API and a vertex shader are needed to be setup properly in order to operate the pixel shader. Firstly, both a vertex and a fragment shader are loaded and enabled. Then, the projection is changed to the orthogonal projection since it is a 2D operation. Third, necessary parameters are passed to the vertex and the fragment shaders. The following is the OpenGL code from the system.

```
// OpenGL preparation
glClear(GL_COLOR_BUFFER_BIT); // clear the buffer
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
// set to orthogonal projection
gluOrtho2D(0.0, textureWidth, 0.0, textureHeight);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
// change the view port to texture size
glViewport(0, 0, textureWidth, textureHeight);

cgGLSetStateMatrixParameter( cgModelViewMatrix,
   CG_GL_MODELVIEW_PROJECTION_MATRIX, CG_GL_MATRIX_IDENTITY);
   // pass the model-view-projection matrix to the vertex shader
// Also pass the necessary parameters including textures.
....
....

glBegin(GL_QUADS);          // render a polygon size of the input texture
glTexCoord2f(0.0f,    0.0f); glVertex2f( 0.0f,                      0.0f);
```

```
    glTexCoord2f(1.0f,    0.0f); glVertex2f( textureWidth,             0.0f);
    glTexCoord2f(1.0f,    1.0f); glVertex2f( textureWidth,    textureHeight);
    glTexCoord2f(0.0f,    1.0f); glVertex2f( 0.0f,            textureHeight);
    glEnd();
```

The vertex shader of a filter is very simple since only a polygon (the same size of the input texture) is needed to be rendered with the input texture. The following is the code of the vertex shader for filters.

```
struct app2vert {
    float4 position : POSITION;
    float2 texCoord : TEXCOORD0;
}; // parameters passed from the application to this vertex shader

struct vert2frag {
  float4 position : POSITION;
  float2 texCoord   : TEXCOORD0;
}; // parameters which will be passed to the fragment shader

vert2frag VPshader(app2vert IN,
                   uniform float4x4 modelViewProj)
{
  vert2frag OUT;

  // multiply the vertex position by the model-view-projection matrix
  OUT.position = mul(modelViewProj, IN.position);

  OUT.texCoord = IN.texCoord; // pass the texture coordinates
  return OUT;
}
```

A.3.1 Edge detection: Sobel filter

For the edge detection, a Sobel filter is implemented in the pixel shader. The following is the pixel shader for the hair silhouettes rendering. Note that the Sobel filter is applied to r, g, b channels (section 5.3).

```
struct vert2frag {
  float4 position : POSITION;
  float2 texCoord : TEXCOORD0;
}; // parameters passed from vertex shader to fragment shader

struct frag2app {
  float color : COLOR;
}; // output color (silhouette)

 frag2app FPshader(vert2frag IN, uniform sampler2D hairTex //hair reference image
                 , uniform float edgeScaler )
{
  frag2app OUT;

  const int NUM = 9; // current pixel (1) + neighbor pixels (8)
  const float unit= 0.001953125; // relative texture coordinates for the pixels
                                 // texture resolution is 512x512
                                 // therefore the minimum texture coordinate unit is
                                 // 0.001953125 = 1 / 512
  const float2 c[9] = {
            float2(-unit,  unit),
            float2( 0.0 ,  unit),
            float2( unit,  unit),
            float2(-unit,  0.0 ),
            float2( 0.0 ,  0.0 ), // current pixel
            float2( unit,  0.0 ),
            float2(-unit, -unit),
            float2( 0.0 , -unit),
            float2( unit, -unit)
  };

  float3  CI[9];
  int i;
  for (i=0; i < NUM; i++) {
      CI[i] = tex2D(hairTex, IN.texCoord.xy + c[i]).xyz;
  } // load values of current and neighbor pixels

  float x, y;
  // applying sobel filters horizontally and vertically
  x = CI[2].x+  CI[8].x+2*CI[5].x-CI[0].x-2*CI[3].x-CI[6].x;
  y = CI[6].x+2*CI[7].x+  CI[8].x-CI[0].x-2*CI[1].x-CI[2].x;
  float edge1 = (abs(x) + abs(y))/8.0f; // for R channel

  x = CI[2].y+  CI[8].y+2*CI[5].y-CI[0].y-2*CI[3].y-CI[6].y;
  y = CI[6].y+2*CI[7].y+  CI[8].y-CI[0].y-2*CI[1].y-CI[2].y;
  float edge2 = (abs(x) + abs(y))/8.0f; // for G channel

  x = CI[2].z+  CI[8].z+2*CI[5].z-CI[0].z-2*CI[3].z-CI[6].z;
  y = CI[6].z+2*CI[7].z+  CI[8].z-CI[0].z-2*CI[1].z-CI[2].z;
  float edge3 = (abs(x) + abs(y))/8.0f; // for B channel

  float total = max( edge1, edge2);
  total = max( total, edge3); // find the max intensity

  OUT.color = total * edgeScaler ; // scale by user defined value
  return OUT;
}
```

A.4 Billboard particle

The vetex positions of particle are calculated in the vertex shader. The center position of the particle is sent to the vertex shader with texture coordinates.

```
  // in OpenGL
  for ( int k=0; k < totalParticleNum ; k++)
  {
    Vector centerPoint = particlePosition[k];
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);
    glVertex3f( centerPoint[0],centerPoint[1], centerPoint[2]);
    glTexCoord2f(1.0, 0.0);
    glVertex3f( centerPoint[0],centerPoint[1], centerPoint[2]);
    glTexCoord2f(1.0, 1.0);
    glVertex3f( centerPoint[0],centerPoint[1], centerPoint[2]);
    glTexCoord2f(0.0, 1.0);
    glVertex3f( centerPoint[0],centerPoint[1], centerPoint[2]);
    glEnd();
  }
```

In the vertex shader, each vertex position is calculated.

```
vert2frag VPshader(app2vert IN,
                   uniform float4x4 modelViewProj,
                   ... )
{
  vert2frag OUT;

  float4 centerPosition =  mul(modelViewProj, IN.position);
  float4 vertexPosition;
  float2 billboard = ( IN.texCoord - float2(0.5, 0.5) ); // get the direction to
                                                         // shift on the screen
  ...
  vertexPosition.xy = centerPosition.xy + billboard * lengthOfParticle;
  vertexPosition.zw = centerPosition.zw;
  OUT.position = vertexPosition;
  ...
  return OUT;
}
```

A.5 Multiple render target

Multiple buffers can be rendered simultaneously by using the OpenGL functions and Cg fragment shader. The following is a simple code example for setting the multiple render target up.

```
  // In OpenGL
  GLenum buffers[4];
  buffers[0] = GL_COLOR_ATTACHMENT0_EXT;
  buffers[1] = GL_COLOR_ATTACHMENT1_EXT;
  buffers[2] = GL_COLOR_ATTACHMENT2_EXT;
  buffers[3] = GL_COLOR_ATTACHMENT3_EXT;

  // attach textures to buffers
  glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT, buffers[0],
                              GL_TEXTURE_2D, myTextureID00, 0 );
  glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT, buffers[1],
                              GL_TEXTURE_2D, myTextureID01, 0 );
  glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT, buffers[2],
                              GL_TEXTURE_2D, myTextureID02, 0 );
  glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT, buffers[3],
                              GL_TEXTURE_2D, myTextureID03, 0 );

  glDrawBuffers( 4, buffers); // draw 4 buffers at once!
  renderSometing();
```

After the textures(render targets) are attached to buffers and render targets are set, the scene is rendered with a pixel shader which outputs multiple colours.

```
struct vert2frag {
    ...
};

struct frag2app {  // output structure with 4 outputs
  float4 buffer1 : COLOR0;
  float4 buffer2 : COLOR1;
  float4 buffer3 : COLOR2;
  float4 buffer4 : COLOR3;
};

frag2app FPshader(vert2frag IN ,
                ... )
```

```
{
    frag2app OUT;
    OUT.buffer1 = float4(1,0,0,1); // buffer1 is filled with red
    OUT.buffer2 = float4(0,1,0,1); // buffer2 is filled with green
    OUT.buffer3 = float4(0,0,1,1); // buffer3 is filled with blue
    OUT.buffer4 = float4(0,0,0,1); // buffer4 is filled with black
    return OUT;
}
```

Appendix B

Splines

Splines represent curves. It is done by specifying a series of points at intervals along the curve and defining a function. The function allows to obtain the points in between the specified points using control points. The curve does not necessarily go through the control points. In this section, t is a parameter between the staring point $(t = 0.0)$ and the ending point $(t = 1.0)$ and $(t \in [0,1])$.

B.1 Bezier curve

Bezier spline is obtained by deCasteljau algorithm to interpolate a curve between $(n+1)$ control points $P_0, P_1 ... P_n$.

$$P_i^j = (1-t)P_i^{j-1} + tP_{i+1}^{j-1} \tag{B.1}$$

where $j = [1,n], i = [0, n-j]$.

B.1.1 Quadratic Bezier spline

The Quadratic Bezier spline has three control points $(n = 2)$: P_0, P_1, P_2. Therefore the curve is obtained by a linear interpolation between linear interpolation between control points P_0, P_1, P_2.

$$\begin{aligned} P_0^1 &= (1-t)P_0 + tP_1 \\ P_1^1 &= (1-t)P_1 + tP_2 \end{aligned}$$

After P_0^1 and P_1^1 are calculated by a linear interpolation, they are linearly interpolated again.

$$P(t) = (1-t)P_0^1 + tP_1^1 \tag{B.2}$$

Therfore we obtain the following formula.

$$\begin{aligned} P(t) &= (1-t)[(1-t)P_0 + tP_1] + t[(1-t)P_1 + tP_2] \\ &= P_0(1-t)^2 + 2P_1(1-t)t + P_2t^2 \end{aligned}$$

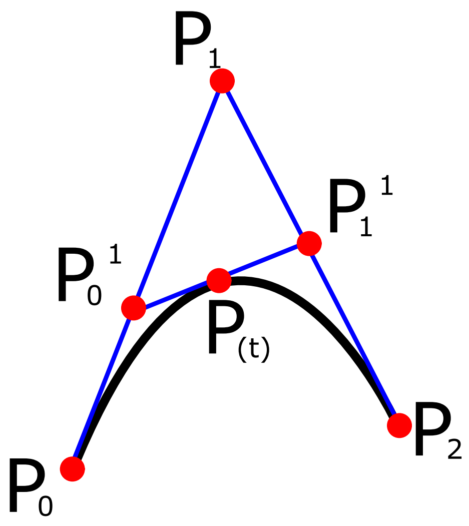


Figure B.1: Quadratic Bezier spline

B.1.2 Cubic Bezier spline

Similarly, the Cubic Bezier curve is obtained by interpolating control points P_0, P_1, P_2, P_3 and the formula is like the following.

$$P(t) = P_0(1-t)^3 + 3P_1(1-t)^2 t + 3P_2(1-t)t^2 + P_3 t^3 \tag{B.3}$$

The Cubic Bezier slpine is used in the Potrace library with few parameter restrictions.

B.2 Catmull-Rom spline

The Catmull-Rom spline has a unique property that the generated curve passes all the control points. A Catmull-Rom spline is a cardinal spline with a tension of 0.5 where a cardinal spline is a Cubic Hermite spline whose tangents are defined by the points and a tension parameter.

B.2.1 Cubic Hermite spline

The Cubic Hermite spline is defined:

$$P(t) = (2t^3 - 3t^2 + 1)p_0 + (t^3 - 2t^2 + t)m_0 + (-2t^3 + 3t^2)p_1 + (t^3 - t^2)m_1 \tag{B.4}$$

where p_0 is the starting point, p_1 is the ending point with starting tangent m_0 and ending tangent m_1.

B.2.2 Cardinal spline

A cardinal spline is a Cubic Hermite spline whose tangents are defined by the points and a tension parameter. The tangent m_i is define:

$$m_i = (1-c)(p_{i+1} - p_{i-1}) \tag{B.5}$$

where c is the tension parameter.

B.2.3 Catmull-Rom in the matrix form

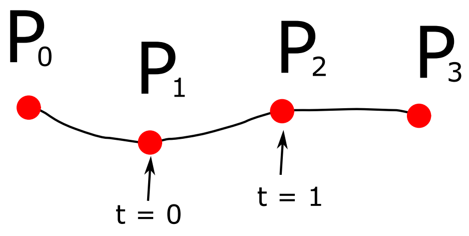


Figure B.2: Catmull-Rom spline

Given four points P_0, P_1, P_2, P_3, let P_1, P_2 be the starting point, ending point and let P_0, P_3 be the points to control tangent at each end (See figure B.2). Since the tension is 0.5, the formula becomes like the following:

$$\begin{aligned} P(t) &= (2t^3 - 3t^2 + 1)P_1 + 0.5(t^3 - 2t^2 + t)(P_2 - P_0) + (-2t^3 + 3t^2)P_2 + 0.5(t^3 - t^2)(P_3 - P_1) \\ &= 0.5(2P_1 + (-P_0 + P_2)t + (2P_0 - 5P_1 + 4P_2 - P_3)t^2 + (-P_0 + 3P_1 - 3P_2 + P_3)t^3) \end{aligned}$$

which can be presented in a matrix form.

$$P(t) = 0.5 \begin{pmatrix} 1 & t & t^2 & t^3 \end{pmatrix} \begin{pmatrix} 0 & 2 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 2 & -5 & 4 & -1 \\ -1 & 3 & -3 & 1 \end{pmatrix} \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix} \tag{B.6}$$

References

[1] K. Akamatsu. Magical teacher negima. *http://en.wikipedia.org/wiki/Negima!:_Magister_Negi_Magi*, 2005.

[2] E. Angel. *Interactive computer graphics: a top-down approach with OpenGL*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1996.

[3] K. Anjyo and K. Hiramitsu. Stylized highlights for cartoon rendering and animation. *IEEE Computer Graphics and Applications*, pages 54–61, 2003.

[4] F. Bertails, C. Menier, and M.-P. Cani. A practical self-shadowing algorithm for interactive hair animation. In *GI '05: Proceedings of the 2005 conference on Graphics interface*, pages 71–78, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. Canadian Human-Computer Communications Society.

[5] D. Card and J. L. Mitchell. Non-photorealistic rendering with pixel and vertex shaders. *http://www.shaderx.com/*, 2002.

[6] M. Côté, P.-M. Jodoin, C. Donohue, and V. Ostromoukhov. Non-Photorealistic Rendering of Hair for Animated Cartoons. In *Proceedings of GRAPHICON'04,* 2004.

[7] M. Eissele, D. Weiskopf, and T. Ertl. The G^2-Buffer Framework. In *Tagungsband SimVis '04, Magdeburg*, pages 287–298, 2004.

[8] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[9] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics, Principles and Practice. Second Edition*. Massachusetts: Addison-Wesley, 1993.

[10] M. Haller and D. Sperl. Real-time painterly rendering for mr applications. In *GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 30–38, New York, NY, USA, 2004. ACM Press.

[11] R. D. Kalnins, P. L. Davidson, L. Markosian, and A. Finkelstein. Coherent stylized silhouettes. *ACM Transactions on Graphics*, 22(3):856–861, July 2003.

[12] R. D. Kalnins, L. Markosian, B. J. Meier, M. A. Kowalski, J. C. Lee, P. L. Davidson, M. Webb, J. F. Hughes, and A. Finkelstein. Wysiwyg npr: Drawing strokes directly on 3d models. *ACM Transactions on Graphics*, 21(3):755–762, July 2002.

[13] T.-Y. Kim and U. Neumann. A thin shell volume for modeling human hair. In *CA '00: Proceedings of the Computer Animation*, page 104, Washington, DC, USA, 2000. IEEE Computer Society.

[14] J. Kobayashi. School rumble. *http://en.wikipedia.org/wiki/School_rumble*, 2004.

[15] M. Koster, J. Haber, and H.-P. Seidel. Real-time rendering of human hair using programmable graphics hardware. In *CGI '04: Proceedings of the Computer Graphics International (CGI'04)*, pages 248–256, Washington, DC, USA, 2004. IEEE Computer Society.

[16] L. Kovacs and T. Sziranyi. Efficient coding of stroke-rendered paintings. In *ICPR '04: Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume 2*, pages 835–838, Washington, DC, USA, 2004. IEEE Computer Society.

[17] M. A. Kowalski, L. Markosian, J. D. Northrup, L. Bourdev, R. Barzel, L. S. Holden, and J. F. Hughes. Art-based rendering of fur, grass, and trees. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 433–438, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[18] KyotoAnimation. Full metal panic!: The second raid. *http://en.wikipedia.org/wiki/Full_Metal_Panic!:_The_Second_Raid*, 2006.

[19] A. Lake, C. Marshall, M. Harris, and M. 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 Press.

[20] W. Liang and Z. Huang. An enhanced framework for real-time hair animation. In *PG ’03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, page 467, Washington, DC, USA, 2003. IEEE Computer Society.

[21] N. Magnenat-Thalmann, S. Hadap, and P. Kalra. State of the art in hair simulation. In *International Workshop on Human Modeling and Animation*, pages 3–9, 2000.

[22] X. Mao, H. Kato, A. Imamiya, and K. Anjyo. Sketch interface based expressive hairstyle modelling and rendering. In *CGI ’04: Proceedings of the Computer Graphics International (CGI’04)*, pages 608–611, Washington, DC, USA, 2004. IEEE Computer Society.

[23] B. 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 Press.

[24] H. Nagatomo. *Draw your own Manga*. Coade, 2003.

[25] N. Nasr and N. Higget. Traditional cartoon style 3d computer animation. 20th Eurographics UK Conference (EGUK ’02), June 11 - 13 2002. De Montfort University, Leicester, UK, p. 122.

[26] M. Nienhaus and J. Döllner. Sketchy drawings. In *AFRIGRAPH ’04: Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 73–81, New York, NY, USA, 2004. ACM Press.

[27] P. Noble and W. Tang. Modelling and animating cartoon hair with nurbs surfaces. In *Computer Graphics International*, pages 60–67, 2004.

[28] J. Northrup and L. Markosian. Artistic silhouettes: A hybrid approach. In *1st International Symposium on Non-Photorealistic Animation and Rendering (NPAR’00)*, pages 31–37, Annecy, France, June 05 - 07 2000.

[29] W. K. Pratt. *Digital Image Processing*. John Wiley and Sons, New York, 1991.

[30] E. Praun, H. Hoppe, M. Webb, and A. Finkelstein. Real-time hatching. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, page 581, New York, NY, USA, 2001. ACM Press.

[31] R. E. Rosenblum, W. E. Carlson, and I. E. Tripp. Simulating the structure and dynamics of human hair: Modeling, rendering and animation. *The Journal of Visualization and Computer Animation*, pages 141–148, 1991.

[32] T. Saito and T. Takahashi. Comprehensible rendering of 3-d shapes. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 197–206, New York, NY, USA, 1990. ACM Press.

[33] A. Secord, W. Heidrich, and L. Streit. Fast primitive distribution for illustration. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pages 215–226, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[34] P. Selinger. Potrace: a polygon-based tracing algorithm. *http://potrace.sourceforge.net/*, 2003.

[35] StudioDEEN, Type-Moon, and Geneon. Fate/stay night. *http://en.wikipedia.org/wiki/Fate_stay_night*, 2006.

[36] E. Sugisaki, Y. Yu, K. Anjyo, and S. Morishima. Simulation-based cartoon hair animation. In *13th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG'05)*, pages 117–122, 2005. Full Paper.

[37] C. Umino. Honey and clover. *http://en.wikipedia.org/wiki/Honey_and_Clover*, 2003.

[38] P. Volino and N. Magnenat-Thalmann. Real-time animation of complex hairstyles. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):131–142, 2006.

[39] M. Webb, E. Praun, A. Finkelstein, and H. Hoppe. Fine tone control in hardware hatching. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 53–ff, New York, NY, USA, 2002. ACM Press.

[40] B. Wilson and K.-L. Ma. Rendering complexity in computer-generated pen-and-ink illustrations. In *3rd International Symposium on Non-Photorealistic Animation and Rendering (NPAR'04)*, pages 129–137, 2004.

[41] M. Woo, Davis, and M. B. Sheridan. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.