ART DIRECTED SHADER FOR REAL TIME RENDERING - INTERACTIVE 3D PAINTING

A Thesis

by

CHETHNA KABEERDOSS

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirement for the degree of

MASTER OF SCIENCE

Chair of Committee,	Ergun Akleman
Committee Members,	Ann McNamara
	John Keyser
Head of Department,	Tim Mclaughlin

December 2016

Major Subject: Visualization

Copyright 2016 Chethna Kabeerdoss

ABSTRACT

In this work, I develop an approach to include Global Illumination (GI) effects in non-photorealistic real-time rendering; real-time rendering is one of the main areas of focus in the gaming industry and the booming virtual reality(VR) and augmented reality(AR) industries. My approach is based on adapting the Barycentric shader to create a wide variety of painting effects. This shader helps achieve the look of a 2D painting in an interactively rendered 3D scene. The shader accommodates robust computation to obtain artistic reflection and refraction. My contributions can be summarized as follows: Development of a generalized Barycentric shader that can provide artistic control, integration of this generalized Barycentric shader into an interactive ray tracer, and interactive rendering of a 3D scene that closely represent the reference painting.

To my grandparents and my suies.

ACKNOWLEDGMENTS

I would like to thank Dr. Ergun Akleman, my committee chair, for providing me continuous support and inspiration while I was working on my thesis. Dr. Ann McNamara and Dr. John Keyser, my committee members, for their constant support.

Special thanks to the faculty and staff of the department of Visualization for making this the awesome place it is for misfits like me.

Yomi Adenuga and Vivek George, thank you for being there for me through thick and thin. Ramakrishnan, Maragathmal, and Arumugam for the timely help because of which I am pursuing my dream. Amudha Kabeerdoss, Rajagopal Kabeerdoss , and Rahul Doss, thank you for your unconditional love and support that aids this curious heart to explore new worlds.

NOMENCLATURE

2D	Two-dimensional
3D	Three-dimensional
API	Application Program Interface
AR	Augmented Reality
CG	Computer Graphics
GI	Global Illumination
GPU	Graphics Processing Units
NPR	Non-Photorealistic Rendering
VR	Virtual Reality

TABLE OF CONTENTS

	Page
ABSTRACT	ii
DEDICATION	iii
ACKNOWLEDGMENTS	iv
NOMENCLATURE	v
TABLE OF CONTENTS	vi
LIST OF FIGURES	viii
LIST OF TABLES	xi
1 INTRODUCTION	1
1.1 Motivation	1
1.2 Goal	2
2 BACKGROUND	5
2.1 Evolution of NPR	5
2.2 Previous Work in Non-Photorealistic Rendering	9
3 METHODOLOGY: STRUCTURE OF GENERIC BARYCENTRIC SHADER	27
3.1 Effect Parameters	29
3.2 Barycentric Formulation for Back-End Shaders	31
3.3 Extension	32
3.3.1 Style Control with Control Images	32
3.3.2 Style Control with Basis Functions	33
3.3.3 Rectangular Box Property	34
3.3.4 Painter's Hierarchy	35
4 BARYCENTRIC SHADER USED FOR THE REAL-TIME NPR SHADER	38

4.1 Hierarchical Zero-Degree Bezier Basis Functions to Represent Simplices in Color Space 38
4.2 Extended Hierarchical Zero-Degree Bezier Basis Functions 39
4.3 The Particular Hierarchy Used in Our Shaders 40
4.3.1 Effect of Control Textures 43
4.3.2 Computing Ω_i Terms in Front-End Shader 43

5 VISUAL ANALYSIS 49

5.1 Visual Analysis 50
5.1.1 Diffuse Reflection Shader 51
5.1.2 Specular Highlight Shader 52
5.1.3 Silhouette Shader 52
5.1.4 Transparency Shader 53
5.2 Shadow 55

6 IMPLEMENTATION 56

6.1 Scene Modeling and Development 58
6.1.1 Rendering Scene Using Mental Ray 58
6.1.2 Digital Compositing/Post Processing 60
6.2 Creating a Generic Barycentric Shader for Non-Photorealistic Real-time Rendering 61
6.3 Results 66

7 CONCLUSIONS AND FUTURE WORK 70

REFERENCES 71

APPENDIX A SHADER CODE 77

LIST OF FIGURES

FIGURE Page

1.1 Chinese ink-and-brush painting with reflection created using 3D computer graphics . . . 3

1.2 Charcoal Shader look-and-feel of hand-drawn charcoal drawing effect for a variety of materials and shapes . . . 3

2.1 Interactive Pen-and-Ink Illustration created by Salisbury et al: A single scene, drawn in a variety of styles . . . 5

2.2 Watercolor effects . . . 6

2.3 Simulated watercolor effects . . . 6

2.4 Curtis et al's Automatic watercolorization . . . 7

2.5 Litwinowicz's technique: an image produced showing the impressionist effects . . . 7

2.6 An example of Philippe Decaudin's cartoon images directly rendered from a 3D model using his cartoon shader. . . . 8

2.7 Gooch shader examples . . . 9

2.8 Flowchart showing the Evolution of NPR . . . 10

2.9 An example of Paul Haeberli's NPR images. . . . 11

2.10 Barbara Meier's technique: Frame from a painterly rendered animation 12

2.11 Barbara Meier's painterly rendering pipeline . . . 13

2.12 Litwinowicz's technique for obtaining impressionist effects . . . 14

2.13 Painterly Rendering with Curved Brush Strokes of Multiple Sizes. . . . 15

2.14 Brick-a-Brac (1995) by Cassidy Curtis. Computer generated sloppy ink rendering style . . . 16

2.15 The New Chair (1998) directed by Cassidy Curtis : Loose and Sketchy NPR . . . 17

2.16 Fishing (1999) by David Gainey, watercolor look done by Cassidy Curtis 17

2.17 A truffula tree top with static graftals organized in a three-level hierarchy. 18

2.18 Black ink diffusion renderings using ink 19

2.19 Artists directly annotated the same 3D teacup model to produce four distinct rendering styles. 20

2.20 Interactive Vector Fields used for Painterly Rendering 21

2.21 Real-time 3D cartoon smoke 21

2.22 Watercolor Inspired Non-Photorealistic Rendering for Augmented Reality 23

2.23 Main conceptual steps of Lu et al's algorithm. 24

2.24 Variety of styles created using Lu et al's algorithm. 24

2.25 Example fins extruded from mesh edges 25

2.26 This painterly cat rendered using fin texture approximation 25

3.1 An example demonstrating the concept of control and weight images. . 33

3.2 A comparison of the effect of the number of discontinuities in changing the final style. 34

3.3 A comparison of convex hull vs. box property. 35

4.1 Barycentric NPR shader breakdown 42

4.2 Monotonically increasing function represented as diagrams 44

4.3 Surface normal(N) and eye vector/camera(I) ray are used to silhouette edge detection 45

4.4 Specular computation 47

4.5 Clamping the specular value to obtain the desired specular highlight . . 47

4.6 How to compute a standard reflection ray for an incoming ray 48

4.7 How to compute a standard refraction ray for an incoming ray 48

5.1 'Decido' - Primary visual reference. Digital painting by Rachel Cunningham 49

5.2 Color analysis of the main object in Cunningham's painting, Decido . . 51

5.3 Specular highlight analysis in Cunningham’s painting, Decido 52
5.4 Color analysis of the silhouetted edge in Cunningham’s painting, Decido 53
5.5 Color analysis of the water in Cunningham’s painting, Decido 54
5.6 Color analysis of the shadows in Cunningham’s painting, Decido 55
6.1 A flowchart showing the process for creating the interactive 3D painting 56
6.2 3D scene modeled in Autodesk Maya 57
6.3 Passes rendered of the Creature from Autodesk Maya using Mental Ray 58
6.4 Post-processing of the 3D scene in Nuke 59
6.5 Final image of 3D scene 60
6.6 Real-time render of the main body of the creature with our custom shader 61
6.7 Main body of the creature with silhouette edge variations 63
6.8 Main body of the creature with transparency variations 63
6.9 Main body of the creature with refraction variations 64
6.10 Custom textures used in the Body Shader 65
6.11 Water geometry rendered real-time with custom shader 66
6.12 Custom textures used in the Ant Shader 66
6.13 Custom textures used in the Cactus Shader 67
6.14 Shadow intensity variation seen on the ground and small creature 67
6.15 Clouds and bird rendered real-time with the custom shader 68
6.16 Frame of the final render 68

LIST OF TABLES

TABLE Page

6.1 Comparison between mentral ray shader - maya rendering and barycentric shader - real time rendering 69

1 INTRODUCTION

Early Computer Graphics(CG) rendering primarily focused on creating complex and realistic images. Over the last two decades, a substantial amount of non-photorealistic rendering (NPR) methods has been developed to enable the simulation of non-photorealistic styles of art forms. According to David et al. [34] to better communicate complex digital art, some form of visual abstraction is needed. Digital art benefited from advancements in CG and rendering technology. Creating a non-photorealistic illustration proved to be more often effective, information was better conveyed by omitting extraneous details and only focusing on the relevant features.

However, there's been a limited amount of research that explored real-time rendering with global illumination(GI) effects, including reflection and refraction for NPR. This is partly due to the unavailability of adequate software and hardware to support rendering real-time reflection and refraction. The parallel processing powers of graphics processing unit (GPU) coupled with the inherently parallel nature of ray tracing makes it possible for an interactive raytracer with GI to exist [29]. Now it is possible to realize a real-time art directable non-photorealistic shader.

1.1 Motivation

Development of a rendering and shading framework to obtain the desired look-and-feel is not an easy task in practice. As explained [5], replication of a style acts as a measure

of our technique and understanding. To the degree that we lack the ability to make a picture look realistic, we also lack the artistic control over it. Shade Trees architecture, a flexible tree-structured shading model that can represent a wide variety of shading characteristics, has laid the foundation for procedural shader concept to create any desired look-and-feel [5] . Despite the success of shaders and shading languages [16], rendering the desired style is a challenge even for highly qualified lighting technical directors. Recently, a new approach called Barycentric Shaders, was developed by Akleman et al. [1] to simplify shader development. They define barycentric operations in the form of parametric functions, which satisfy partition of unity and guarantees that colors calculated by the functions always stay inside the convex hull of a set of control colors. The method has been successfully used in some specific artists' styles including a recent Chinese-ink painting [25], see Figure 1.1, and Charcoal rendering [12], see Figure 1.2. In this work, we present a more general application of Barycentric Shaders that can be used to obtain any style that resembles a painting specifically in real-time rendering.

1.2 Goal

This work, demonstrates the possibility of translating the visual aesthetics of a painting into real-time rendering. This approach is a streamlined process. We have developed a generic Barycentric shader that can be used in an interactive raytracer developed by NVIDIA. The OptiX- Application Program Interface (API) [29] is an application framework for achieving optimal ray tracing performance on the GPU. We chose to use this

Figure 1.1: Chinese ink-and-brush painting with reflection created using 3D computer graphics

Figure 1.2: Charcoal Shader look-and-feel of hand-drawn charcoal drawing effect for a variety of materials and shapes

framework to create painterly rendered scenes in real-time using GI. One main goal of this shader is to ensure its design allows artists to achieve desired styles with the help of

intuitive and interactively modifiable shader control parameters.

Although Barycentric shading methods allow us to obtain the desired look-and-feel, the current rendering pipeline does not allow interactive interaction with such artistic results that includes GI effects such as shadow, reflection, and refraction in real-time. There is a need for the development of rendering methodology that allow interactive interaction with artistically rendered virtual worlds i.e three-dimensional(3D) scenes. This work develops a methodology that allows users to interact with such artistically rendered 3D scenes. Gaming, augmented reality and virtual reality are industries where this shading methodology is applicable. We demonstrate the effectiveness of this approach by transforming an existing painting into a 3D painterly rendered scene.

2 BACKGROUND

The evolution of technology has lead to the creation of non-photorealistic shaders with GI for real-time rendering. In this section we discuss previous research in NPR.

2.1 Evolution of NPR

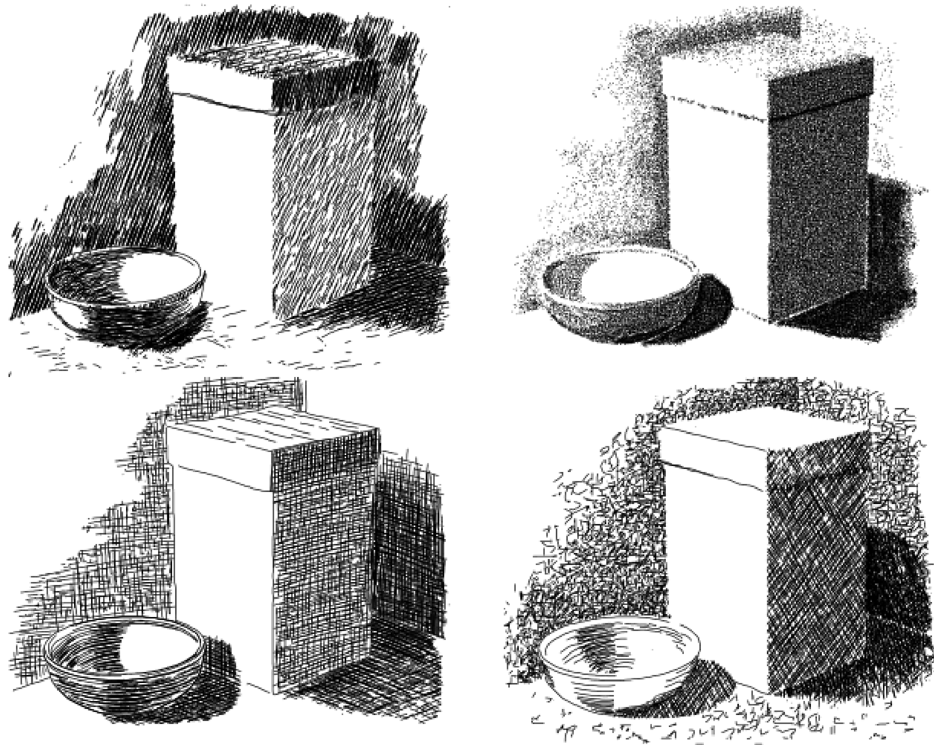


Figure 2.1: Interactive Pen-and-Ink Illustration created by Salisbury et al: A single scene, drawn in a variety of styles

According to Stuart Green, NPR is referred to as the creation of images without aspiring to be realistic [31]. Some of the well-known NPR categories include:

• Pen and Ink illustrations Techniques: cross hatching, outlines, line art. Figure 2.1

shows an example of this technique.

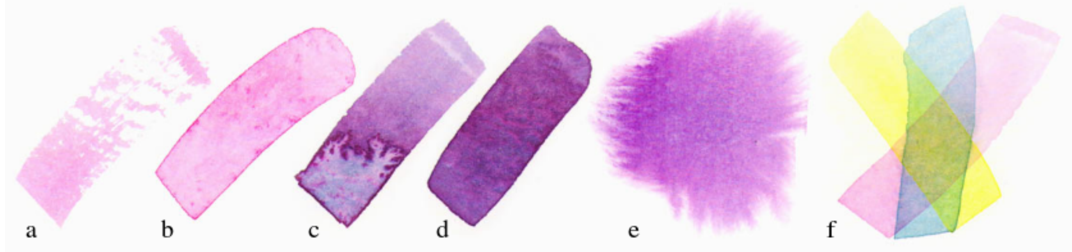


Figure 2.2: Watercolor effects. (a) drybrush, (b) edge darkening, (c) backruns, (d) granulation, (e) flow effects, (f) glazing

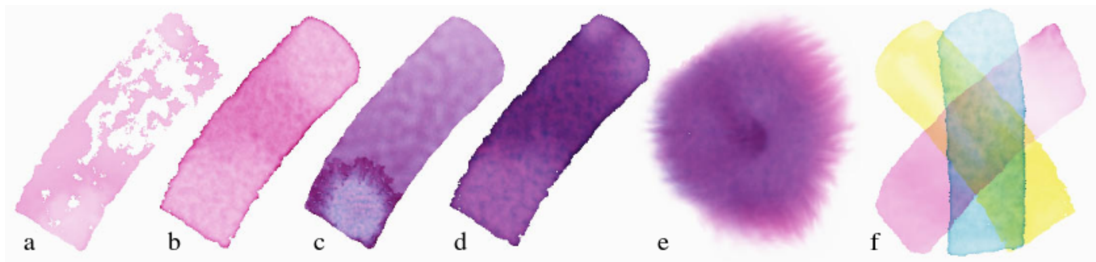


Figure 2.3: Simulated watercolor effects. (a) drybrush, (b) edge darkening, (c) backruns, (d) granulation, (e) flow effects, (f) glazing

- Painterly Rendering Styles: watercolor, ink, impressionist, expressionist, pointilist. Figures 2.2, 2.3, 2.4, 2.5 show examples of this technique.
- Cartoons Effects: cartoon shading, distortion. Figure 2.6 shows an example of this technique.
- Technical Illustrations Characteristics: Matte shading, edge lines. Figure 2.7 shows an example of this technique.

The flow chart shown in Figure 2.8 describes the evolution of NPR. It shows how computer generated images in the form of pixel art, a term published by Adele Goldberg

(a) Automatic watercolorization of the finished painting

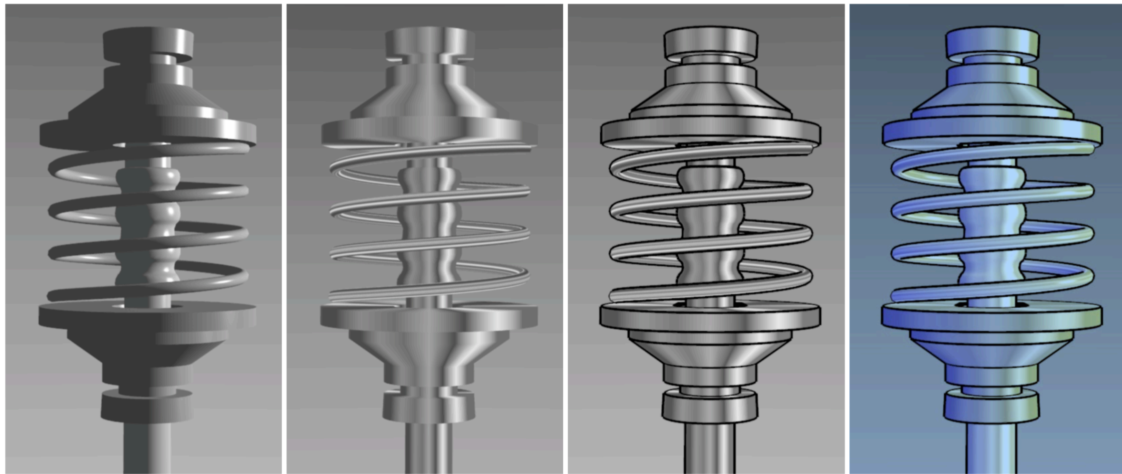
(b) Low resolution image

Figure 2.4: Curtis et al's Automatic watercolorization

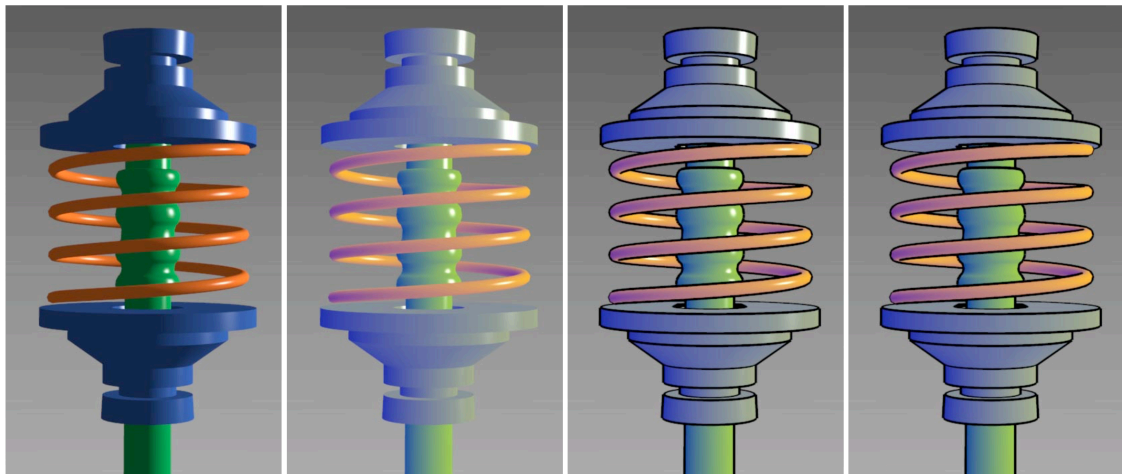
Figure 2.5: Litwinowicz's technique: an image produced showing the impressionist effects

Figure 2.6: An example of Philippe Decaudin's cartoon images directly rendered from a 3D model using his cartoon shader.

and Robert Flegal[13], was a form of traditional digital art. The images were edited on the pixel level, this kind of art can be seen in early 2D games. Pixel art is commonly divided into two subcategories: isometric and non-isometric. The isometric pixel art is created in 2.5D projection, this is commonly seen in games to provide a pseudo 3D viewing effect. Non-isometric pixel art is created in orthographic views (top, side, front, bottom). It involves the conversion of 3D wire frame geometry into 2D rendered images. These images were either rendered as photorealistic (close approximation to the real world) or non-photorealistic images (a stylized abstract of it's real world counter-part).



(a)



(b)

Figure 2.7: Gooch shader examples. (a) Left to Right: Phong shaded object; New metal-shaded object without edge lines; New metal-shaded object with edge lines; New metal-shaded object with a cool-to-warm shift. (b) Left to Right: Phong model for colored object; New shading model with highlights, cool-to-warm hue shift, and without edge lines; New model using edge lines, highlights, and cool-to-warm hue shift; Approximation using conventional Phong shading, two colored lights, and edge lines.

2.2 Previous Work in Non-Photorealistic Rendering

A significant amount of work has been published on real-time and non-real-time NPR. One common goal is to produce images that closely mimic artistic or expressive

depictions.

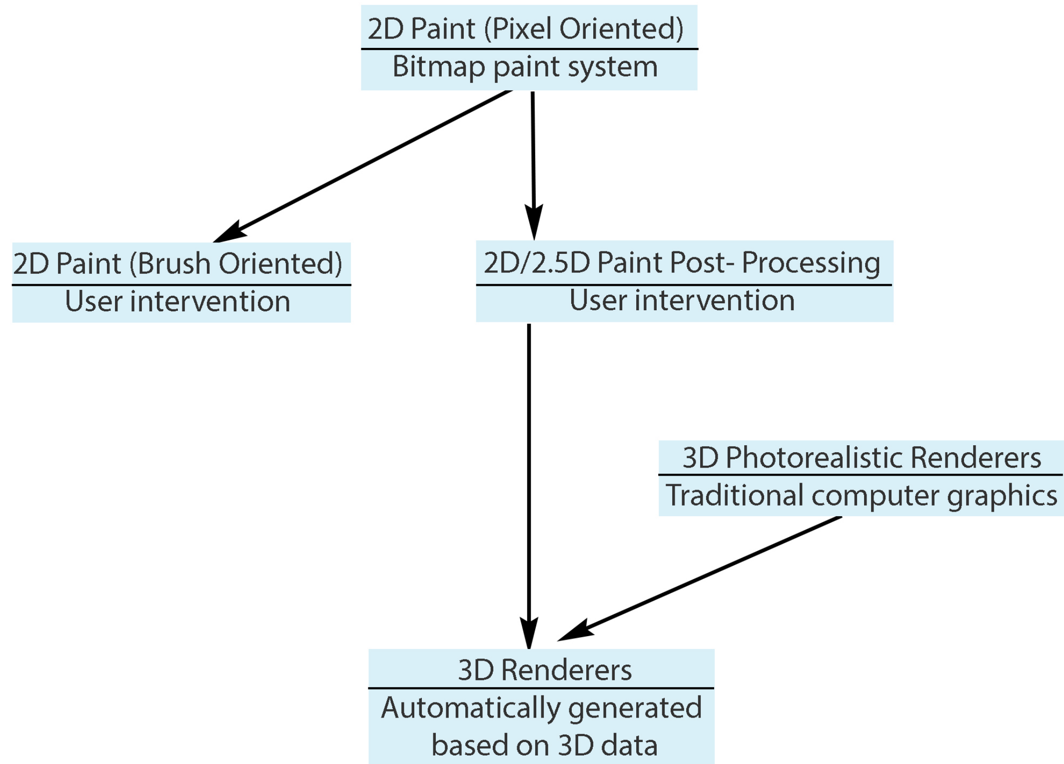
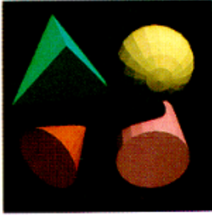


Figure 2.8: Flowchart showing the Evolution of NPR

Haeberli introduced the first automatic and semi-automatic painterly rendering algorithms for NPR to replicate impressionistic paintings in CG [15]. He developed a program that interactively selects and manipulates visual information from a source image or 3D scene. He describes a painting as an ordered list of brush strokes. Each brush stroke represents a collection of attributes which includes the location, color, size, angle and shape of the brush stroke. An example is shown in Figure 2.9. Several commercial painting software, such has Adobe Photoshop, Fractal Design Painter and Matador Paint

System, have incorporated Haeberli's NPR algorithms into their systems.



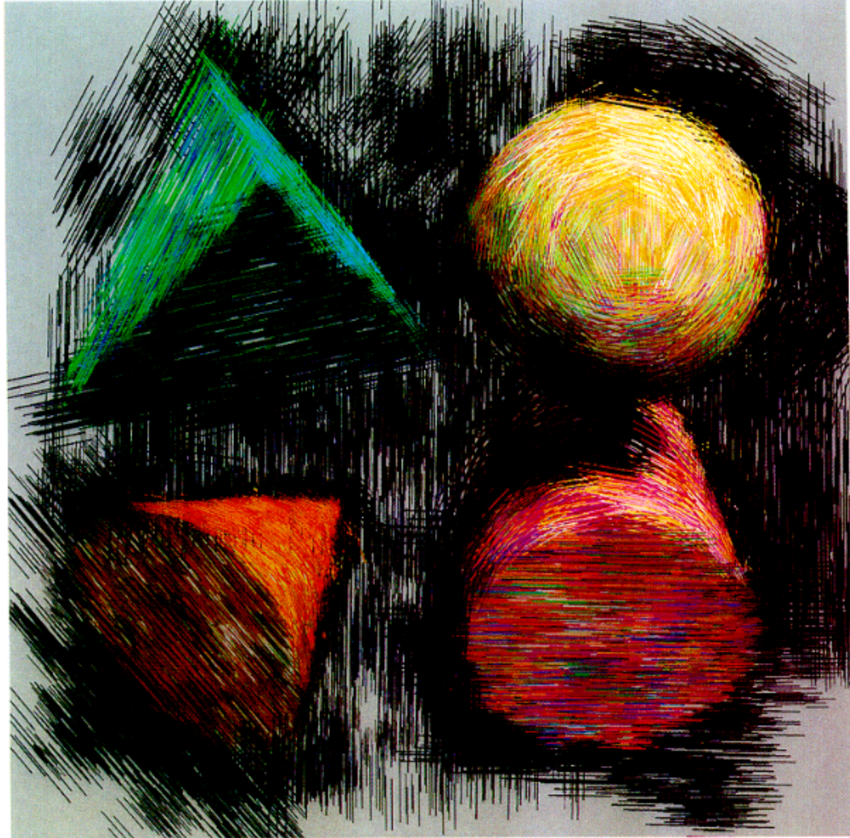


Figure 2.9: An example of Paul Haeberli's NPR images. These images are obtained as a result of non-linear filtering 2D images.

Decaudin introduced cell shading by using a combination of multi-pass rendering and image processing techniques to create stylized images from data represented in a 3D scene [11]. An example is shown in Figure 2.6. In cell shading, the characters and objects in the scene are represented by their respective outlines. The outline varies in thickness. The color inside can be uniform or a gradient between two colors. This shading also has a shadow effect attribute. Decaudin also stated that, achieving cell shading requires two

separate outlines. One outline is created for the object's silhouette and a second to crease edges.

Meier [28] developed on Haeberli's [15] work by attaching particles to 3D objects and placing brush strokes to coincide with the particles. This resulted in images that resembled impressionist paintings as shown in Figure 2.10. Figure 2.11 shows the painterly rendering pipeline introduced by Meier.

Figure 2.10: Barbara Meier's technique: Frame from a painterly rendered animation

Salesin et al. [34] implemented a number of principles of traditional pen-and-ink illustrations as part of an automated rendering system. This was extended to an interactive rendering system by Salisbury et al. [32]. Collections of strokes arranged in different pat-

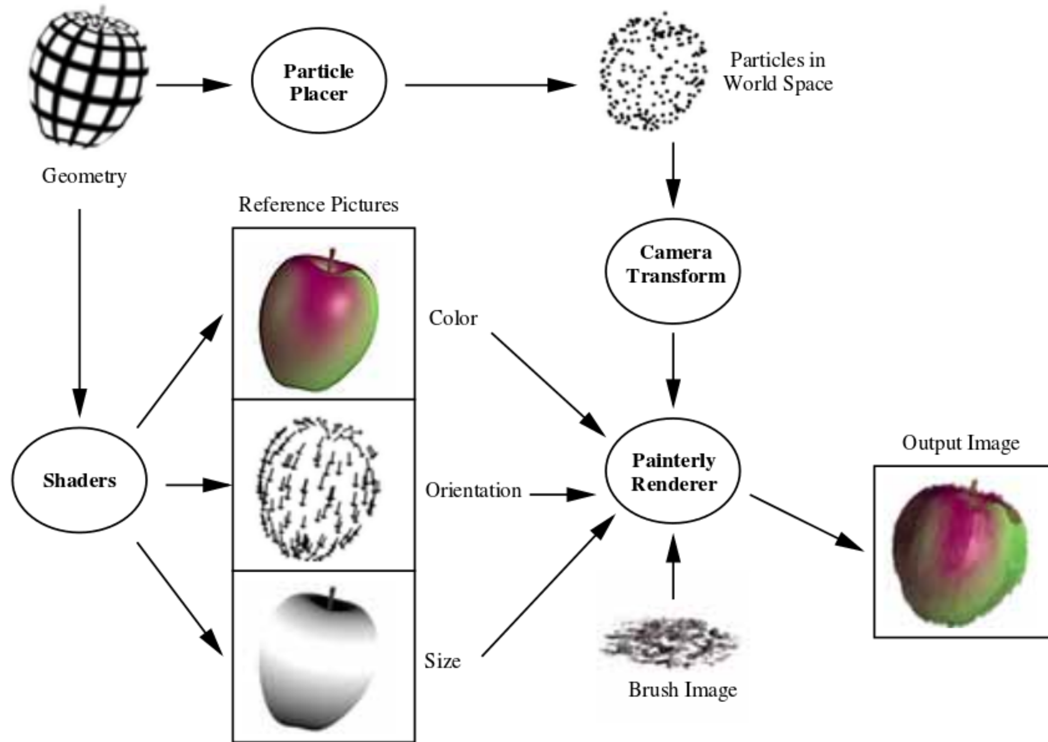


Figure 2.11: Barbara Meier's painterly rendering pipeline

terns are used to generate texture and tone. This paper created a mouse-based interface that generated strokes (single and multiple) automatically. This can achieve texture, tone, and shape. The user *painted* using textures and tones, and the computer generated individual strokes. This system also allows edge extraction from images, useful for outlining. Figure 2.1 shows a scene created, in a variety of styles, from this system.

Litwinowicz's paper [24], explores the technique of transforming ordinary video segments into animations with a hand painted effect. This paper makes use of the underlying principles of Haeberli's [15] techniques for transforming images to painterly animations. This conversion process includes a brush stroke orientation algorithm and a technique to move the brush strokes for each frame. This produced a temporally coherent

painterly animation. An optical flow field was used to push short brush strokes along scene movements and provided various tools for editing/correcting flow and layering to obtain painterly/impressionist effects, see Figure 2.12. Figure 2.5 shows an image produced with this technique.

Figure 2.12: Litwinowicz's technique for obtaining impressionist effects. Two frames and the optical flow field that maps pixels from one frame to another

Hertzman [17] presented a method that creates painted appearance from a photograph, by automating the paint and draw process without human intervention. The method involves painting with varying brush sizes to express various details in an image. According to Hertzman, achieving details in a painting requires the use of a layering technique. This consists of a rough large brush sketch that is then painted over progressively with smaller brushes until a visual resemblance to the source image is achieved. Figure 2.13 shows the progression of the layering technique.

Cassidy Curtis made a computer-generated short film, *Brick – a – Brac*, in 1995, which had a sloppy ink rendering style. He created this as an exercise to demonstrate personality through animated lines, [7]. Figure 2.14 shows the animation rendering process.

In 1998 Curtis [8] created a *Loose and Sketchy* rendering style by drawing only what

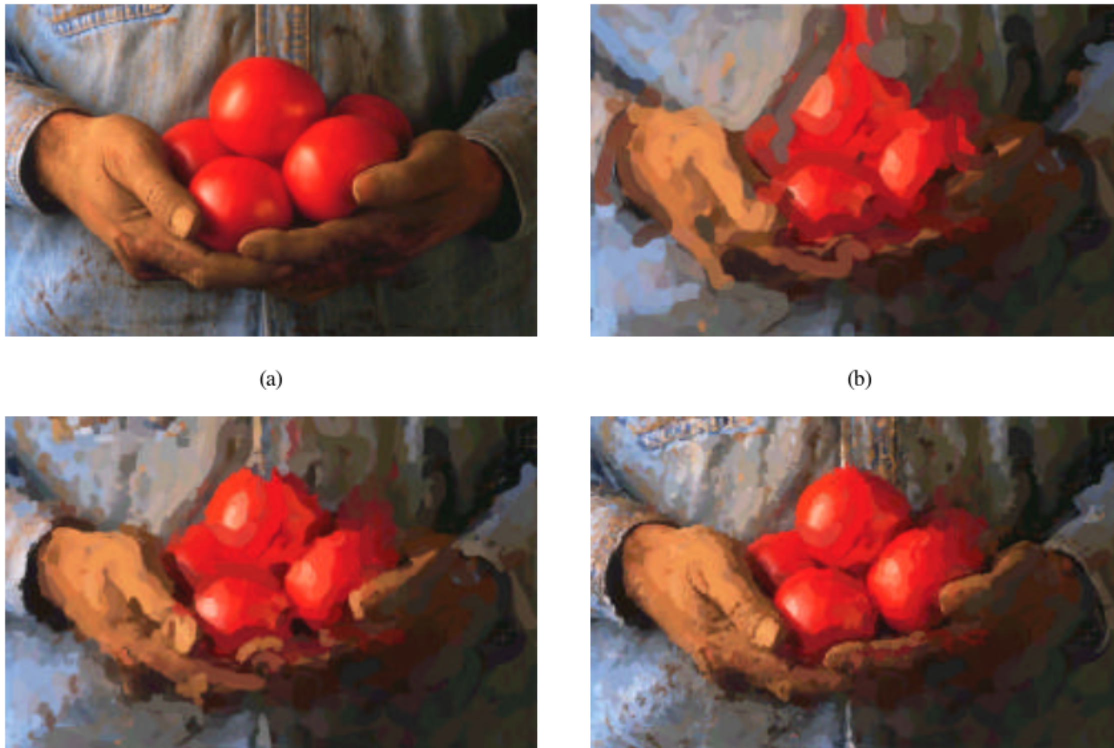


Figure 2.13: Painterly Rendering with Curved Brush Strokes of Multiple Sizes. (a) A source image. (b) The first layer of a painting, after painting with a circular brush of radius 8. (c) The image after painting with a brush of radius 4. (d) The final image, after painting with a brush of size 2. Note that brush strokes from earlier layers are still visible in the painting.

is necessary. A hand drawn sketch is an artist's interpretation, it encourages the viewer to participate in the completion of the image. This engages the viewer's mind unlike a photorealistic image. The line gesture conveys the character's mental state. To achieve this effect, the shader requires a user input depth map of the 3D models, and the conversion of the animation into a sequence of gestural sketches. This tool was used for motion testing and for creating finished animations in a non-photorealistic style. Figure 2.15 shows a screen capture from the *The New Chair* [9] short film using this NPR style.

Curtis also developed a watercolor look for the film *Fishing* [10], directed by David

Gainey, in 1999. He achieved the watercolor look using image processing and noise synthesis. Figure 2.16 show a screen capture from the film.

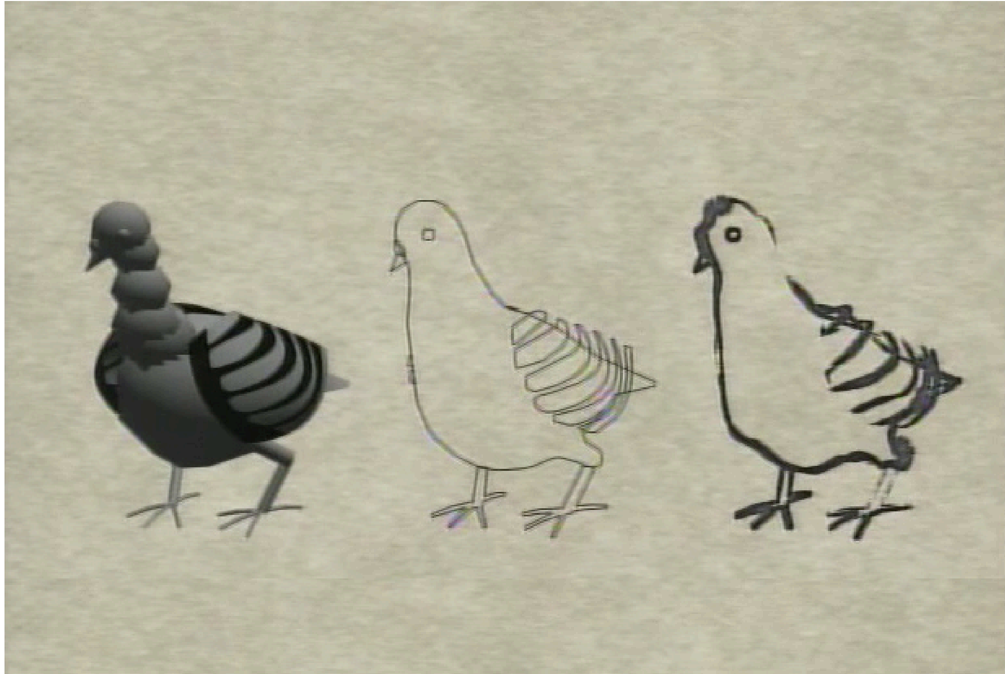


Figure 2.14: Brick-a-Brac (1995) by Cassidy Curtis. Computer generated sloppy ink rendering style

Gooch et al. [14] developed a shading approach that provides art directable control of the color of the output image. This technique is now popularized as Gooch shading. It involves the surface computation of a light source and the representation of objects in the scene as two tone shades in the final image. This helps identify the surface orientation of the object. Shading occurs only in mid-tones so that edge lines and highlights remain visually prominent. The paper talks about technical illustrations occupying the middle ground of abstraction, where the important properties of 3D objects are accented while the extraneous detail are diminished. See Figure 2.7 for an example image.

Figure 2.15: The New Chair (1998) directed by Cassidy Curtis : Loose and Sketchy NPR

Figure 2.16: Fishing (1999) by David Gainey, watercolor look done by Cassidy Curtis

Kowalski et al. [22] in their 1999 paper proposed an algorithm to render stylized fur, grass and trees. This stylized look suggests the complexity of the scene without representing it explicitly. They use *graftal* textures which places fur, leaves, grass or other geometric elements into the scene procedurally to achieve a particular effect. *Graftal* is a stroke-based procedural texture that generates detailed elements such as fur, hair, and grass. The important aspect of this paper is the algorithm that implements the placement of these *graftals* to match the aesthetic requirements of a particular texture. Figure 2.17 shows an example of the *graftal* placement with relation to the camera distance.

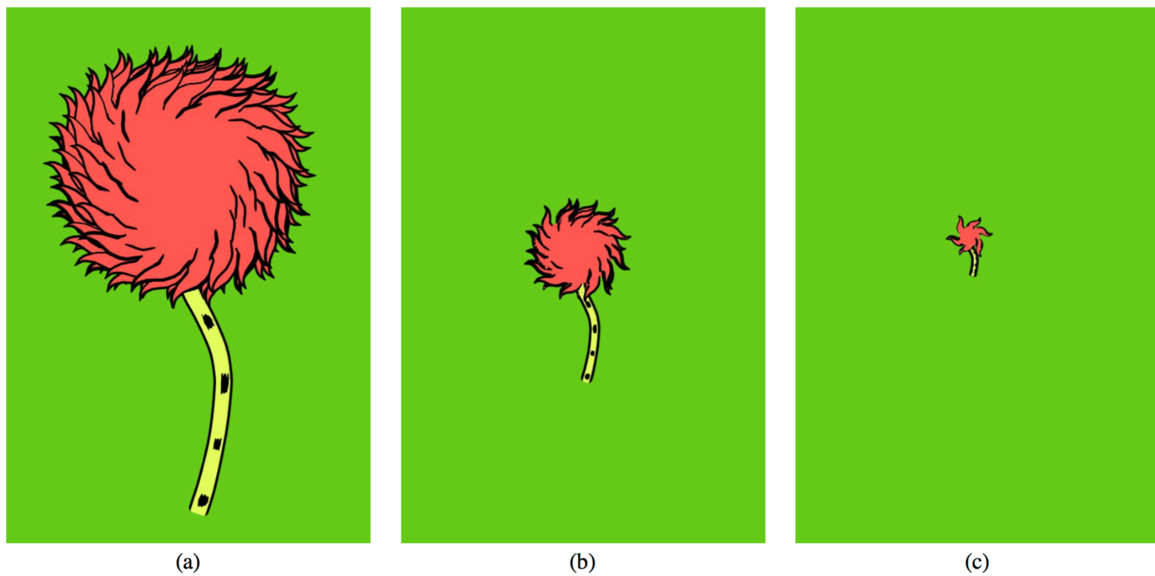


Figure 2.17: A truffula tree top with static graftals organized in a three-level hierarchy. (a) When the camera is close, all three levels are drawn. (b) As the camera zooms out, only two levels are drawn, and (c) finally just the base level is drawn.

In the 2000s, research on NPR branched into different applications. [33], [18], [21]. Most notable of them included the NPR for interactive rendering and videos/animation

sequence. Lee [23] implemented a technique for rendering oriental black ink paintings with realistic diffusion effects. This is based on a model that simulates a variety of paper types and black ink properties. This explores real-time dynamic simulations of realistic ink diffusion. Amount of water, coarse particles, density of paper/cloth are some of the parameters used to explore the ink diffusion. See to Figure 2.18 for an example.

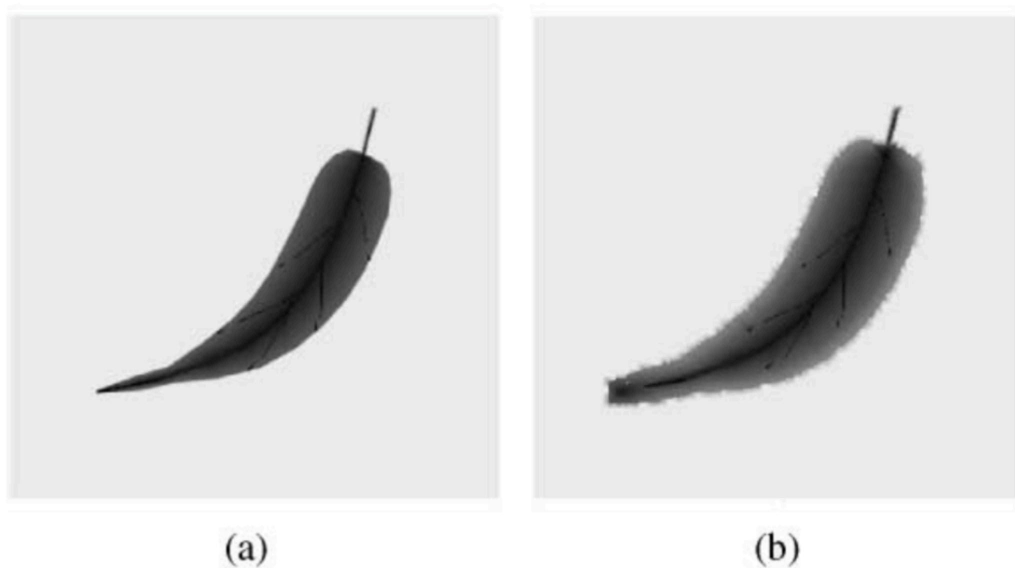


Figure 2.18: Black ink diffusion renderings using ink. (a)only a limited amount of water, producing weak diffusion and (b) an abundant amount of water producing strong diffusion.

Kalnins et al. [20] presented a system for drawing stroke-based NPR styles directly on 3D models. The system offers the choice of brush styles, paper textures, background and base-coats. The artist decides a *brush* style, then draws strokes over the model from one or more viewpoints. When the system renders the scene from any new viewpoint, it adapts the number and placement of the strokes appropriately to maintain the original look. See Figure 2.19

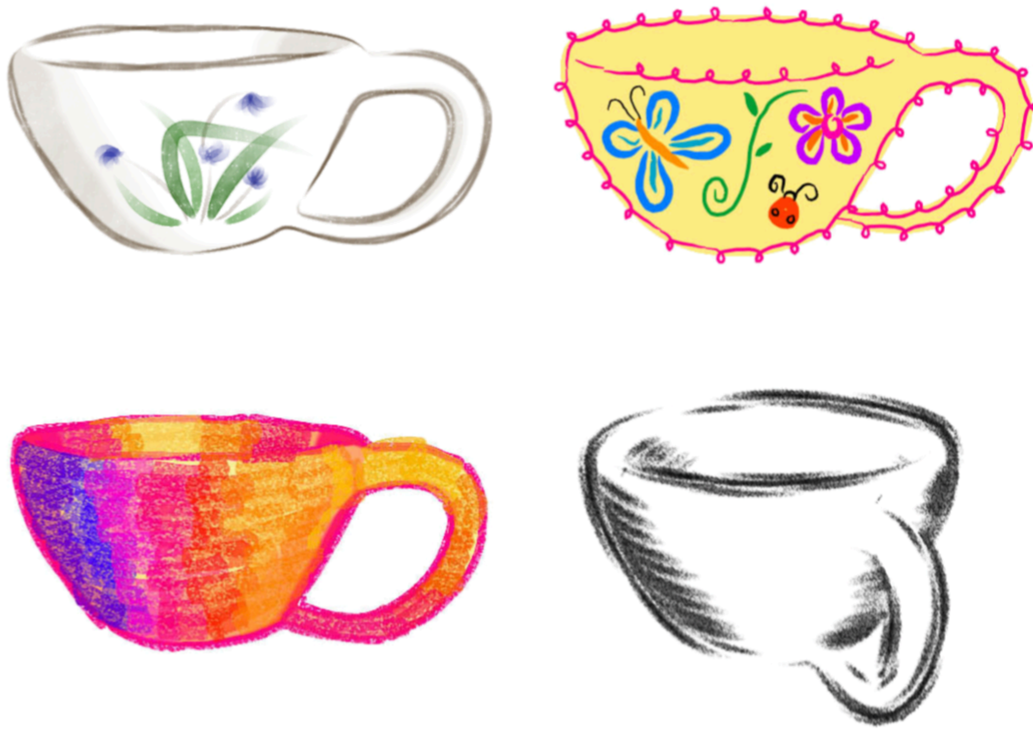
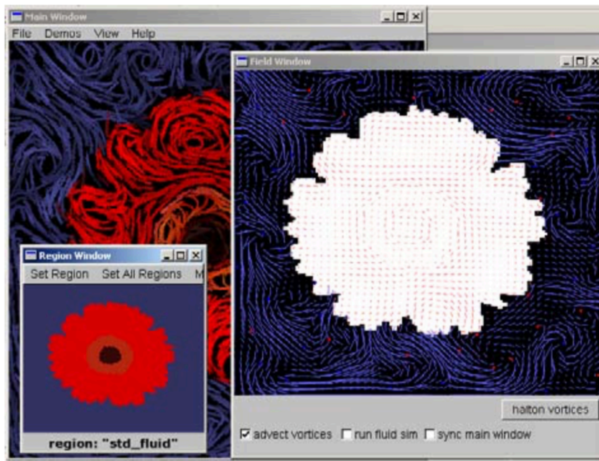


Figure 2.19: Artists directly annotated the same 3D teacup model to produce four distinct rendering styles.

Chu et al. [4], presented a physically-based method for simulating ink dispersion in absorbent paper. They developed an ink flow model based on the method of lattice Boltzmann equation (LBE). They implemented a painting system called *MoXi* using OpenGL and the CG shading language. The ink simulation operations are implemented in fragment programs running on the GPU (parallel processors).

The trend of using physics to produce artistic rendering was adopted by Olsen et al. [30]. The research makes use of vortex dynamics and semi-Lagrangian fluid simulation to create controlled vector fields. The vector field is used in the creation of painterly images and animations. See Figure 2.20 for an example.



(a)

(b)

Figure 2.20: Interactive Vector Fields used for Painterly Rendering. (a) The application showing the field and region editing windows, (b) Rendering of a red poppy

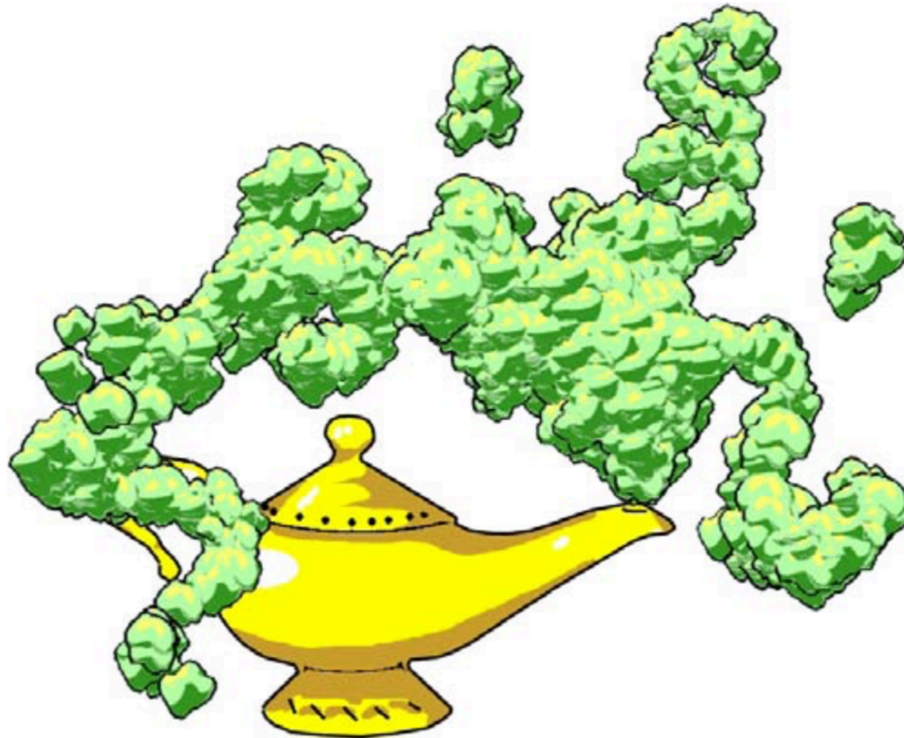


Figure 2.21: Real-time 3D cartoon smoke

McGuire et al. designed an algorithm for rendering real-time interactive smoke animations in stylized cartoon [27].The rendering algorithm implements four effects: cell shading, shadowing, black outlines, and gradient fills. These can be selectively used to approximate different cartoon styles and provide artistic control. Figure 2.21 shows an example of this style of rendering.

Chen et al. [3] explored watercolor NPR in real-time Augmented reality using voronio diagrams to mimic watercolor effects. To maintain visual coherence, edge detection and re-tiling of the voronoi cells is needed. This work did not explore the inclusion of GI. See figure 2.22

Lu et al. [26] explored the possibility of converting images, video, and 3D animation sequences into painterly renderings in real-time. The parallel computing power of GPU enables artists to execute the placement of brush strokes in real-time. See Figure 2.23 which shows the steps of the algorithm used for this type of rendering, and Figure 2.24 showing the variety of styles created using the software.

Imhof et al. [19] present a method for real-time painterly rendering by interpreting off-surface paint strokes as volumetric data, near the surface of the 3D mesh. Their method is based on fin textures in which mesh edges are extended orthogonally off the surface and textured to replicate the painterly renders of the custom offline rendering method. See Figure 2.25 and Figure 2.26

Recently, Akleman et al. [1] generalized Gooch shading [14] using Barycentric algebra to provide art-directed control of colors in the final images. This type of shader

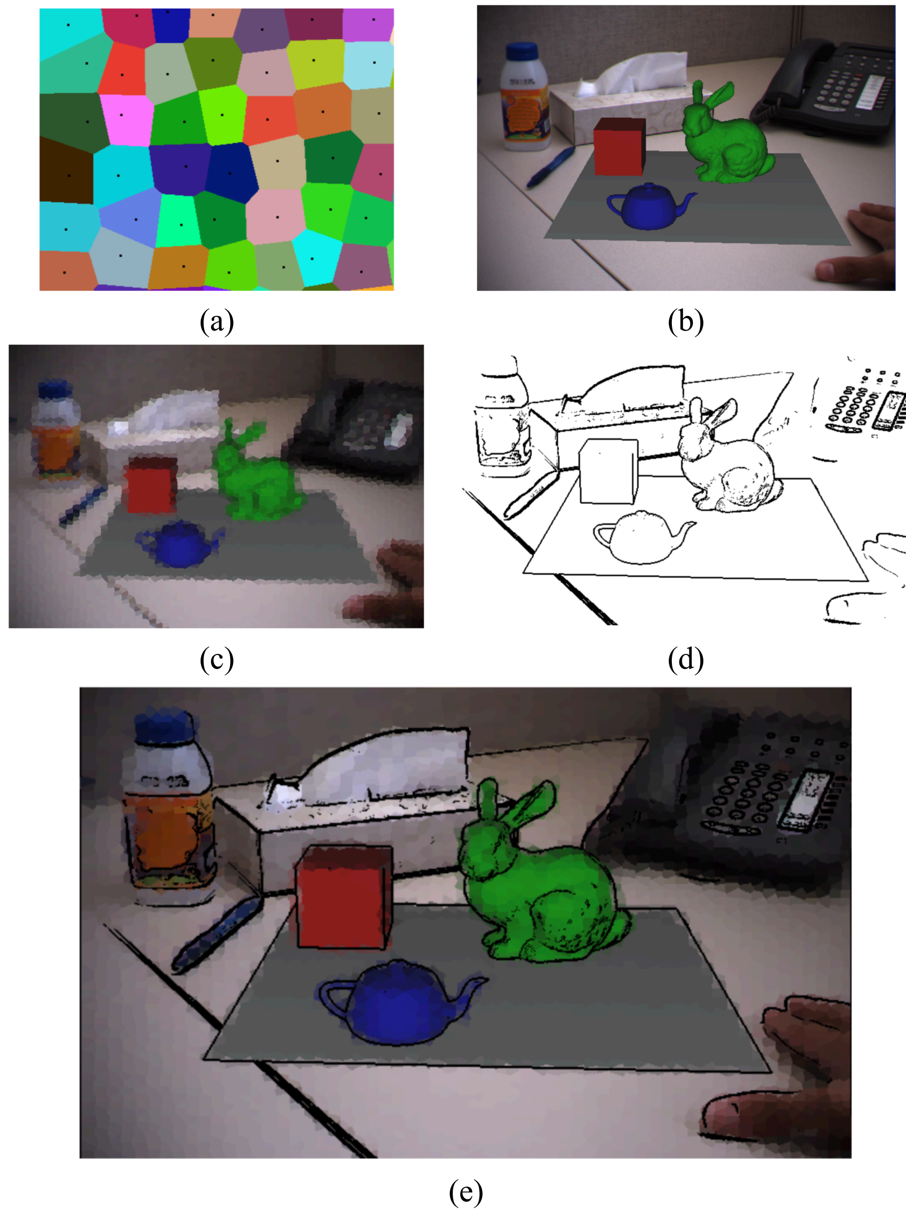


Figure 2.22: Watercolor Inspired Non-Photorealistic Rendering for Augmented Reality. (a) A Voronoi pattern, (b) An original AR frame , (c) Image processed by a Voronoi pattern, (d) Detected edges in an AR frame , and (e) A final rendering, combining tiling and edges

guarantees the desired style irrespective of the underlying renderer and illumination model.

This method has successfully been used to generate shaders for Chinese ink-and-brush paintings with reflection [25], refer to Figure 1.1. Similarly, the same barycen-

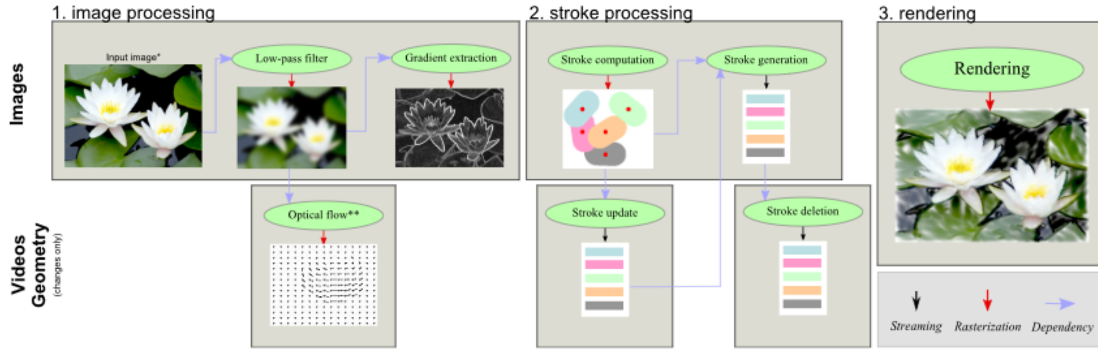
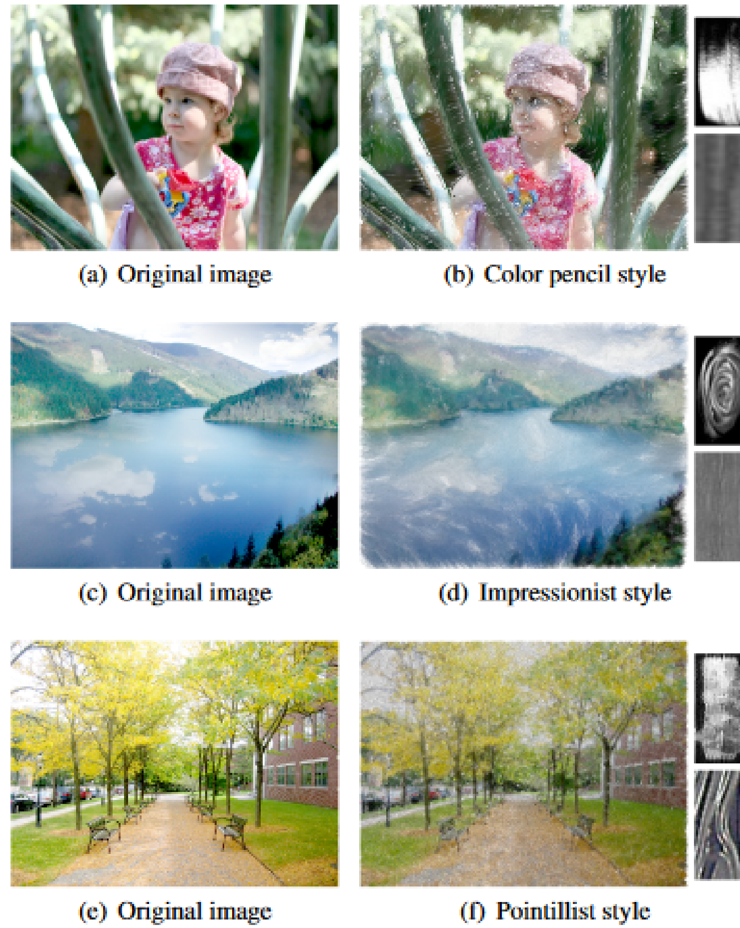


Figure 2.23: Main conceptual steps of Lu et al's algorithm.

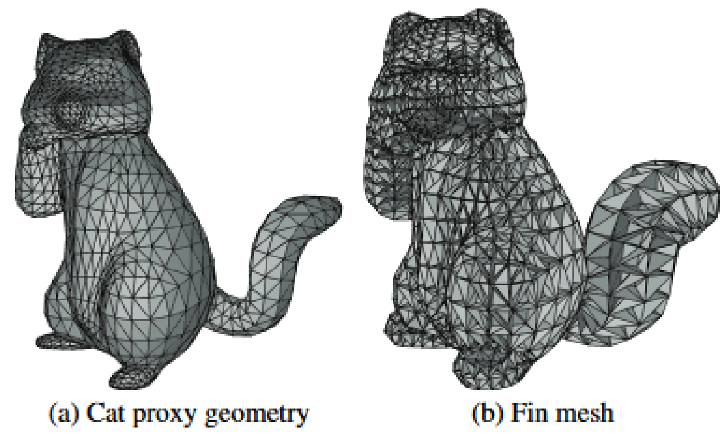


(a) Original image (b) Color pencil style

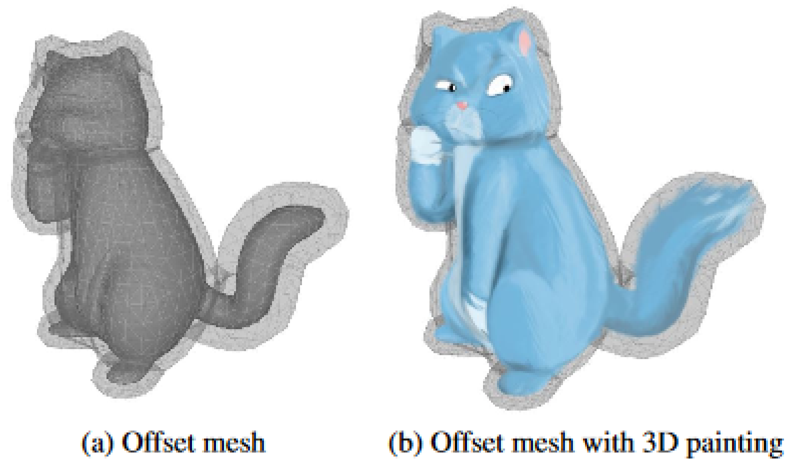
(c) Original image (d) Impressionist style

(e) Original image (f) Pointillist style

Figure 2.24: Variety of styles created using Lu et al's algorithm.

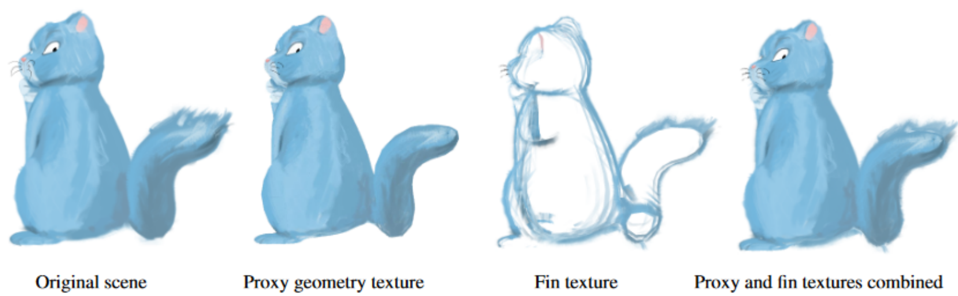


(a) Cat proxy geometry (b) Fin mesh



(a) Offset mesh (b) Offset mesh with 3D painting

Figure 2.25: Example fins extruded from mesh edges



Original scene Proxy geometry texture Fin texture Proxy and fin textures combined

Figure 2.26: This painterly cat rendered using fin texture approximation

tric shader was adopted in charcoal rendering with reflection [12], refer to Figure 1.2. Liu's work demonstrated global illumination effects such as reflections using Yang Ming's paintings as visual reference. Liu's research provided a non-photorealistic water reflection shader. Du's work demonstrated effects such as reflection in charcoal renderings. These shading approaches can only be used in post-production with no real-time interaction.

In this work, I provide a real-time interactive approach to rendering 3D versions of paintings with GI while providing direct artistic control of resulting colors.

3 METHODOLOGY: STRUCTURE OF GENERIC BARYCENTRIC SHADER

In this section, we present Barycentric Shaders a shading framework based on barycentric algebra. Barycentric shaders provide an intuitive art-directed control over the final output. Akleman et al. [1] demonstrated example scenarios where this framework was used to produce user desired style of 3D scenes regardless of the underlying rendering method and illumination model.

According to Akleman et al. [1], introducing an addition/subtraction operator in shader development may result in negative values. The resulting color value is the difference between two points referred to as conceptual vector. The difference of two colors may not result in a positive real value. Negative color values are difficult to understand hence not intuitive to work with as control parameters.

The Barycentric Shader method poses a solution that uses operations that satisfies partition of unity such as mixing operations. The results obtained are from colors defined with positive real numbers. Using the barycentric algebra in shaders does not require a significant conceptual change, it only requires the shader operator to be of the form given by equation 3.1

$$c = \sum_{i=0}^{M} \omega_i C_i \quad where \quad \sum_{i=0}^{M} \omega_i = 1 \quad and \quad \forall\, \omega_i \geq 0 \tag{3.1}$$

where C_i's are colors, i.e. n-tuples of positive real numbers. The restriction that the weights ω_i are all positive and sum to 1 is called the partition of unity property. This property

guarantees that the solution c stay inside of the convex hull defined by the colors C_i. This restriction does not impose any limit over the polynomials that can be used. Basis functions of parametric polynomials used in geometric modeling, such as Bezier, B-splines and β-splines satisfy this property.

Parametric rational, irrational or piecewise polynomial can be converted to a form that satisfies the partition of unity property [2]. Choosing $\omega_0 = (1 - t)$ and $\omega_1 = t$ with $0 \leq t \leq 1$, which results in the mix operator, utilizing the basis functions of first degree Bezier curves, which satisfy partition of unity.

In addition, the convex hull property that comes with partition of unity, is particularly useful in practical shader development applications. It provides an intuitive control mechanism for obtaining desired results.

The extension of the Barycentric algebra into shader development can be implemented by adapting primitives similarly used in geometric modeling. There are practical aspects of shader development that differentiate this problem from modeling curves and surfaces. In order to resolve this Akleman et al. [1] separated the rendering architecture into three phases. The front-end shader, interface between Front and Back-end shader, and the back-end shader.

- Front-end shaders:

  Compute information about the geometric surface including displacement, bump, normal vectors and angles. Akleman et al. [1] use classical shader calculation to deal with negative and complex numbers. This step is carried to produce parameters

for the back-end shaders, that computes the final colors.

- Interface Between Front and Back-end Shaders:

  The Front-end shaders combine all incoming radiance that produce a certain effect as a lump sum term and turn this lump sum into a single color-point. Front end shader also produce a single effect parameter for each effect. Let k_i(p) $\in [0; 1]$ denotes effect parameter that corresponds to the weight of importance of that particular effect i in a given shading point p.

- Back-end shaders:

  They are constructed using the barycentric operations to compute the color based on the parameters passed from the front-end shaders. While developing the back-end shader, a reference image is used to ensure the desired style is maintained regardless of how the parameters from the front-end shaders are computed.

3.1 Effect Parameters

The number of effect parameters corresponds to the dimension of the shading functions which is dependent on the chosen painting. The front-end shader provides a single color-point C_i and an effect parameter k_i. C_i gives a lump sum of all light reaching a given point and produces a particular effect, while k_i gives the importance of that effect in a given shading point. Akleman et al. [1] discuss 6 different effects. In this research, we discuss only the effects that are relevant to our goal.

- Diffuse Reflection Shader:

A diffuse front-end shader provides a single diffuse color-point that gives a lump sum of all diffusely reflected light that reaches a given point and a diffuse effect parameter that provides importance of that effect in that particular point. The lump sum color point can be computed by including illumination coming from point lights, ambient occlusion, environment illumination or final gathering or incoming photons. It is helpful to compute shadow separately using either ray tracing or shadow maps. Shadow parameter is used to adjust corresponding diffuse color-point, which gives a better control over results.

- Specular Highlight Shader:

  A specular highlight front-end shader provides a single color-point as a parameter that is a lump sum of all specular reflected illumination for a given view direction and specular highlight effect parameter. Specular highlights and true mirror reflection are not compatible. A shader designer should decide how to compute the lump sum parameter from the specular highlights and true mirror reflections.

- Silhouette Shader:

  A silhouette front-end shader provides the silhouette color and effect parameter. It provides information about the distance from the shading point to the silhouette edge of an object from a given view direction. This effect parameter can be obtained as a monotonically increasing function of $n \bullet n_o$.

- Transparency Shader:

  Transparency front-end shader provides a color-point which is computed by com-

bining refraction and mirror reflection. It makes use of the fresnel and transparency effect parameter to describe the transparency of the point. Fresnel parameter is computed as a function of the angle between normal vector, incident ray direction, and index of refraction. Fresnel turns reflection and refraction into a lump sum parameter that can be included as a single parameter.

3.2 Barycentric Formulation for Back-End Shaders

Barycentric shaders and parametric shapes originate from the same theoretical framework, but there are significant differences between them, which can be broadly categorized as follows:

- Position Dependency:

  Every surface or volume point of a barycentric shader behaves differently. The shader functions should consist of parameters of either texture coordinates (u, v)'s, that correspond to surface positions, or volumetric positions (x, y, z)'s.

- Non-Scalar Parameters:

  In shape modeling, parameters are single dimensional. For instance, a parametric surface is defined by two single parameters. On the other hand, shading parameters for each effect consists of one color-point and one effect parameter.

- Constraints on Colors:

  For modeling objects, the entire parameter space is used to compute shapes. On the other hand, in rendering we construct only a subset of allowed colors.

- Dimension:

  In shape modeling, the number of parameters can be at most three; one for curves, two for surfaces and three for volumes. On the other hand, the number of independent shading effects can be much higher than three.

3.3 Extension

The parametric curve and surface methods in geometric modeling cannot be used directly in the implementation of barycentric shaders. Akleman et al.[1] present solutions that help in the development of these barycentric shaders.

3.3.1 Style Control with Control Images

The first approach Akleman et al. presents is based on using texture images as coefficients of parametric equations in Barycentric form. This formulation allows every shading point to have different material property that can be derived from a set of texture or control images.

Barycentric form provides partition of unity and guarantees that the final color of any given shading point stays in the convex hull of control colors. The final color is just a weighted average of the control colors. Thus a function computing the final color is a weighted average of a set of control images, given as

$$I = \sum_{i=0}^{M} \Omega_i \; I_i, \quad \text{and} \quad \sum_{i=0}^{M} \Omega_i = 1, \tag{3.2}$$

where I_i's are the control images and Ω_i's are the weight images, that are computed from basis functions and parameters which ensures they satisfy extended partition of unity. 1 is

a white image and I is the final rendering.

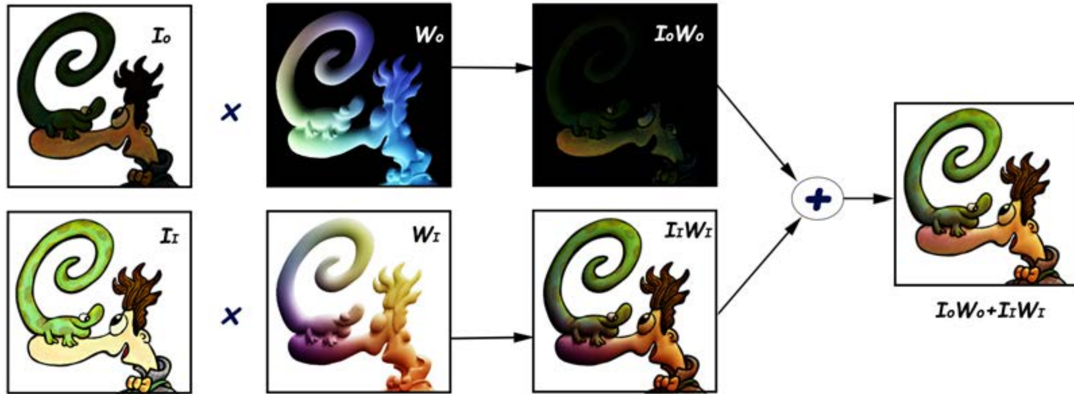
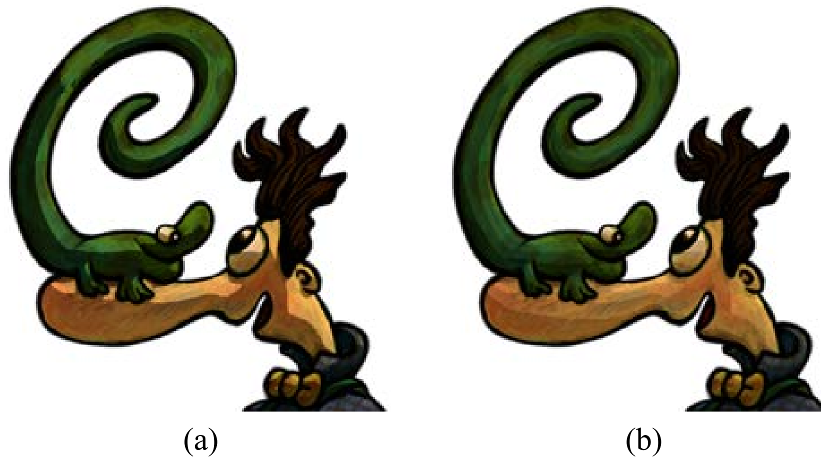


Figure 3.1: An example demonstrating the concept of control and weight images. I_0 and I_1 are control images, Ω_0 and Ω_1 are weight images that satisfy partition of unity. $I = I_0\Omega_0 + I_1\Omega_1$ is the final rendering obtained by taking a weighted average of the two control images.

3.3.2 Style Control with Basis Functions

According to Akelman et al. control images determine overall style, and the basis functions play an important role in obtaining desired styles. This is achieved by controlling how colors are distributed across the final image. An important element of visual style is the functional continuities underlying the parametric function. $G2$ continuous function appears significantly different *in style* than a function that is $G0$ discontinuous in some regions. Apart from the number of discontinuous regions, the shapes and the sizes of the functions influences our perception of style. The control texture images may be discontinuous, which makes it difficult to evaluate the continuity of control images. Where as we can always analyze properties of the basis functions that are used to create

the final images. For instance, consider the two renderings presented in Figure 3.2(a) and (b). Although these images are discontinuous, it is easy to see in (a) there are only two and in (b) there are four distinct regions. These perceived regions result from the number of zero-degree B-splines. Note that when the number of zero degree basis functions increases, the final image approaches a linear interpolation.



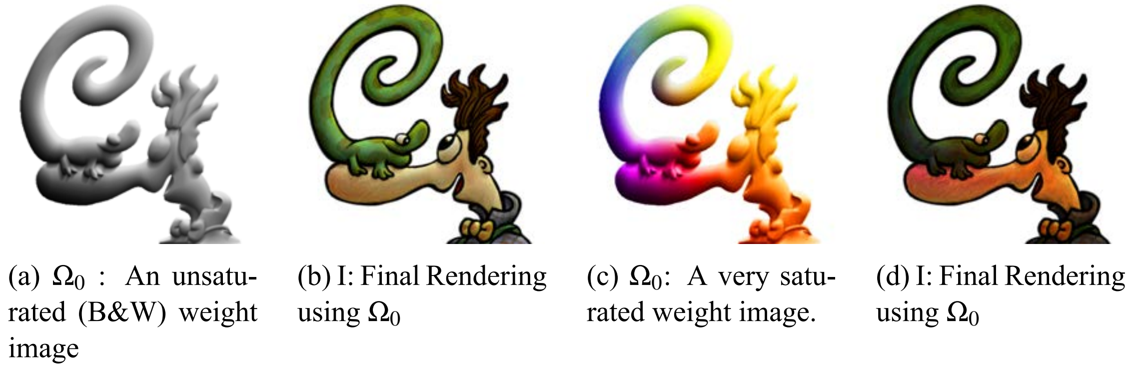
(a) (b)

Figure 3.2: A comparison of the effect of the number of discontinuities in changing the final style. To create consistency, we obtained the two new control images needed in (b) by using linear interpolation of the original control images.

3.3.3 Rectangular Box Property

If weights for a given shader point (u, v) are unsaturated, i.e. greyscale colors, then the convex hull property obtained is dependent on the colors, see Figure 3.3a. This is similar to geometric modeling where the convex hull is dependent on the defined points. However, if weights are just random colors, each dimension gives a convex hull property independent from one another. This results in a wide range of color. An example is shown

in Figure 3.3b.



(a) Ω_0 : An unsaturated (B&W) weight image

(b) I: Final Rendering using Ω_0

(c) Ω_0: A very saturated weight image.

(d) I: Final Rendering using Ω_0

Figure 3.3: A comparison of convex hull vs. box property. A greyscale weight image provides a convex hull. On the other hand, a very saturated weight image extends color possibilities to a bounding box.

3.3.4 Painter's Hierarchy

Handling a high number of effect parameters with a parametric equation can be challenging. This is simplified by using the painter's hierarchy, i.e. following the layering order the painter uses. A painter first creates a base image of the painting, which corresponds roughly to diffuse reflection. Next, the other effects such as specular highlights, silhouette edges, transparency and shadows are created. Similarly a hierarchy of effect parameters can be created, starting with the most essential effect and the rest is added in sequence. The most essential effect is denoted as I_0 , the next one as I_1 and so on.

$$
\begin{aligned}
\mathbf{I'}_0 &= \mathbf{I}_0\, k_0\Omega_0 \\
\mathbf{I'}_1 &= \mathbf{I'}_0(\mathbf{1}-k_1\Omega_1)+\mathbf{I}_1\;\; k_1\Omega_1 \\
&\ldots \\
\mathbf{I'}_m &= \mathbf{I'}_{m-1}(\mathbf{1}-k_m\Omega_m)+\mathbf{I}_m\;\; k_m\Omega_m \\
&\ldots \\
\mathbf{I'}_M &= \mathbf{I'}_{M-1}(\mathbf{1}-k_M\Omega_M)+\mathbf{I}_M\;\; k_M\Omega_M
\end{aligned}
\tag{3.3}
$$

where $k_n \in [0,1]$ is the effect parameter that is used to control importance of a given effect and $I_n\Omega_n \in [0,1]^3$ is the total radiance received by the shading point that produce that particular effect. This particular equation provides rectangular box property.

In some cases when the user does not want to restrict $I_n \in [0,1]^3$ and wants to provide convex hull property, the equation can be simplified to a bilinear equation for simplices.

$$
\begin{aligned}
\mathbf{I'}_0 &= \mathbf{I}_0\, k_0\Omega_0 \\
\mathbf{I'}_1 &= \mathbf{I'}_0(\mathbf{1}-k_1)+\mathbf{I}_1\;\; k_1\Omega_1 \\
&\ldots \\
\mathbf{I'}_m &= \mathbf{I'}_{m-1}(\mathbf{1}-k_m)+\mathbf{I}_m\;\; k_m\Omega_m \\
&\ldots \\
\mathbf{I'}_M &= \mathbf{I'}_{M-1}(\mathbf{1}-k_M)+\mathbf{I}_M\;\; k_M\Omega_M
\end{aligned}
\tag{3.4}
$$

where $k_n \in [0,1]$ is the effect parameter that is used to control importance of a given effect and $I_n \in [0,\infty]^3$ is the total radiance received by the shading point that produces that

particular effect. This particular equation provides convex hull property. An additional advantage of this formulation is that we do not have to make a significant conversion from front-end shader to back-end shader. On the other hand, we must still be careful in handling negative numbers.

Establishing the painter's hierarchy is critical in achieving the desired style of rendering, i.e having the structure of the shader closely follow the exact hierarchy of the chosen painting. Choosing the parametric equations is equally important, i.e establishing the barycentric operations and the control colors. In simplex equations each barycentric operation is a mixture of 2 colors, this number can vary. For instance in cartoon shading, discontinuous functions such as zero-degree B-splines can be used as barycentric basis functions. For smoother color changes functions such as first-degree B-splines, quadric or cubic Bezier curves can be used. The control colors can be identified by carefully sampling the colors in the chosen painting. Any software, such as Adobe Photoshop, with a color picker can be used for this process.

4 BARYCENTRIC SHADER USED FOR THE REAL-TIME NPR SHADER

In this section, we present a simple non-photorealistic Barycentric shading network that can be used in a real-time ray tracer. Barycentric shaders are capable of representing a wide variety of non-photorealistic styles [1]. In this work, we focus on a real-time application of Barycentric shaders for recreating expressive paintings. This section also describes a basic structure of our generic Barycentric shader.

4.1 Hierarchical Zero-Degree Bezier Basis Functions to Represent Simplices in Color Space

As discussed in the previous section, we use Barycentric basis functions to get a controlled shader network. In this work, we use a specific type of Barycentric equations that provides simplices, which are higher dimensional extensions of lines and triangles. An M-simplex is a M-dimensional polytope defined as the convex hull of its M+1 points. An M-simplex interpolates its M+1 end-colors denoted by $\mathbf{C}_i$, where $i = 0, 1, \ldots, M$. Simplices can be given using a high dimensional version of bilinear form, which is given iteratively as follows:

$$
\begin{aligned}
\mathbf{C'}_1(t) &= \mathbf{C}_0(1-t_1)+\mathbf{C}_1 t_1 \\
& \cdots \\
\mathbf{C'}_i(t) &= \mathbf{C'}_{i-1}(1-t_i)+\mathbf{C}_i t_i \\
& \cdots \\
\mathbf{C}(t) &= \mathbf{C'}_{M-1}\,(1-t_M)+\mathbf{C}_M\, t_M
\end{aligned}
\tag{4.1}
$$

As far as $0 \leq t_i \leq 1$, Barycentric rule is satisfied. This gives us an additional advantage of controlling hierarchy, which is particularly important in our case to obtain desired results.

Note that in this case the heart of the hierarchical form is degree one Bezier basis functions $(1-t)$ and t. This set of basis functions guarantees that if a color is chosen from the given options, the output color remains in the convex hull. To turn this equation into a shading equation changes need to be made.

4.2 Extended Hierarchical Zero-Degree Bezier Basis Functions

Let us now denote control texture of the most essential effect as I_0, the next one as I_1 and so on. Then,

$$
\begin{aligned}
\mathbf{I'}_0 &= \mathbf{I}_0\, k_0\Omega_0 \\
\mathbf{I'}_1 &= \mathbf{I'}_0(\mathbf{1}-k_1\Omega_1)+\mathbf{I}_1\;\, k_1\Omega_1 \\
&\cdots \\
\mathbf{I'}_m &= \mathbf{I'}_{m-1}(\mathbf{1}-k_m\Omega_m)+\mathbf{I}_m\, k_m\Omega_m \\
&\cdots \\
\mathbf{I'}_M &= \mathbf{I'}_{M-1}(\mathbf{1}-k_M\mathbf{I}_M)+\mathbf{I}_M k_M\Omega_M
\end{aligned}
\tag{4.2}
$$

where $k_n \in [0,1]$ is the effect parameter that is used to control importance of a given effect and $t_n\Omega_n \in [0,1]^3$ is the total radiance received by the shading point that produces that particular effect. This particular equation provides rectangular box property.

4.3 The Particular Hierarchy Used in Our Shaders

As discussed earlier, our shader consists of four effects and need five control images to obtain results as follows:

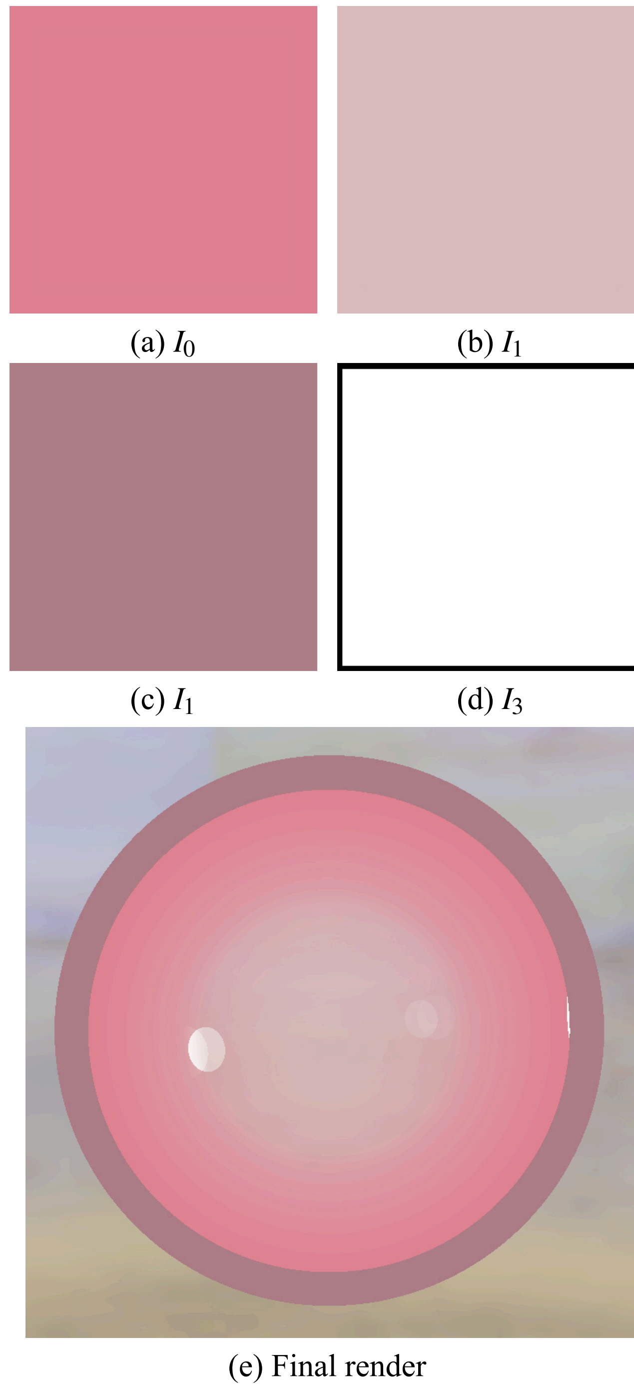
$$
\begin{aligned}
\mathbf{I'}_0 &= \mathbf{I}_0\, k_0\Omega_0 \\
\mathbf{I'}_1 &= \mathbf{I'}_0(\mathbf{1}-k_1\Omega_1)+\mathbf{I}_1\;\, k_1\Omega_1 \\
\mathbf{I'}_2 &= \mathbf{I'}_1(\mathbf{1}-k_2\Omega_2)+\mathbf{I}_2\;\, k_2\Omega_2 \\
\mathbf{I'}_3 &= \mathbf{I'}_2(\mathbf{1}-k_3\Omega_3)+\mathbf{I}_3\;\, k_3\Omega_3 \\
\mathbf{I'}_4 &= \mathbf{I'}_3(\mathbf{1}-k_4\Omega_4)+\mathbf{I}_4\;\, k_4\Omega_4
\end{aligned}
\tag{4.3}
$$

Ambient Term: This is the zeroth term given as $\mathbf{I}_0\ k_0\Omega_0$ where $\mathbf{I}_0$ is the control texture choosen $k_0\Omega_0 = 1$. If necessary $k_0\Omega_0$ term can be used to to create darker regions.

Diffuse Reflection Term: $\mathbf{I}_1$ is the diffuse control texture that gives us how we want the object to be rendered when there complete diffuse illumination. Ω_1 represents the lump sum of all diffusely reflected light that reaches a given point and k_1 is a percentage that controls the amount of diffuse reflection. If $k_1 = 0$, then there is no diffuse reflection. The lump sum color point Ω_1 can be computed by including illumination coming from point lights. In this work we do not include ambient occlusion, environment illumination, final gathering or incoming photons. Shadow is computed separately using ray tracing. Shadow parameter is used to adjust corresponding diffuse illumination Ω_1, which gives a better control over the results.

Silhouette Highlight Term: $\mathbf{I}_2$ is the silhouette control texture that determines how the object is to be rendered in silhouette regions. Ω_2 represents the term that controls silhouette regions and k_2 is a percentage that controls the the amount of silhouette. If $k_3 = 0$, then there is no silhouette.

Specular Highlight Term: $\mathbf{I}_3$ is the specular highlight control texture that controls how the object is rendered when there is full specular illumination. Ω_3 represents the lump sum of all specularly reflected light that reaches a given point while k_3 is a percentage that controls the the amount of specular highlight. If $k_3 = 0$, then there is no specular highlight. The lump sum color point Ω_3 can be computed by including specular illumination coming from point lights.



(a) I_0 (b) I_1 (c) I_1 (d) I_3 (e) Final render

Figure 4.1: Barycentric NPR shader breakdown. Single colors used for I_0, I_1, I_2, and I_3. We chose $k_0 = k_1 = k_2 = k_3 = 1$. We ignored transparency by choosing $k_4 = 0$. Final image shows the shader applied on a sphere.

Transparency Term: $\mathbf{I}_4$ is the transparency control that determines how the object is rendered in transparent regions. Ω_4 is the term that combines incoming refraction and reflection using Fresnel. In addition, k_4 is a percentage that controls the amount of transparency. In other words, if $k_4 = 0$, then the object is opaque, otherwise it is transparent.

4.3.1 Effect of Control Textures

Figure 4.1 demonstrates the effect of control textures on final rendering. Note that this particular shader does not have transparency since we chose $k_4 = 0$.

4.3.2 Computing Ω_i Terms in Front-End Shader

Ω_i terms are standard terms computed by standard shaders. We frequently used monotonically increasing functions to convert computed illumination to Ω_i. The simplest and one of the most useful monotonically increasing function is clamp, which can be defined as,

$$y = Clamp(t, t_0, t_1) = \begin{cases} 0 & \text{if } \; t_0 \leq x, \\ \dfrac{t - t_0}{t_1 - t_0} & \text{if } \; t_0 < t < t_1, \\ 1 & \text{if } \; x \leq t_1, \end{cases} \tag{4.4}$$

This function monotonically increases only $t_0 \leq t_1$. We can now provide how Ω_i terms are computed in Front-End shader. Figure 4.2 shows a graphical representation of the clamp equation

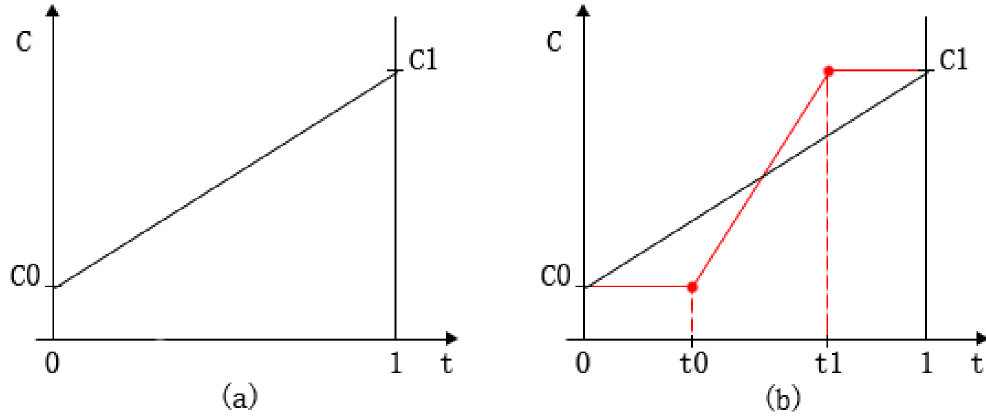


Figure 4.2: Monotonically increasing function represented as diagrams. (a) represents interpolation between C_0 and C_1, (b) add t_0 and t_1 parameters to provide flexible controls to blend the two colors C_0 and C_1

Diffusely Reflected Illumination

Ω_1 is the lump sum of all diffusely reflected light that reaches a given point. It is computed as

$$\Omega_1 = \sum_{i=0}^{N} sh_{L_i}\ t_{L_i}\ C_{L_i} \tag{4.5}$$

where N is the number of lights in the scene, C_{L_i} is the color of light source L_i, sh_{L_i} is the shadow parameter for the light L_i computed by ray tracer and t_{L_i} is the monotonically increasing function. This is the $\cos\theta$ that is computed by dot product of surface normal $\vec{n}$ and light vector for the light L_i $\vec{n_{L_i}}$ as

$$t_{L_i} = clamp(\vec{n} \cdot \vec{n_{L_i}}, -1, 1) = clamp(\cos\theta, -1, 1). \tag{4.6}$$

Note that, in this case, the monotonically increasing function $clamp(\cos\theta, -1, 1)$ is

given by the following equation

$$clamp(\cos\theta, -1, 1) = \frac{\cos\theta + 1}{2} \tag{4.7}$$

Silhouette

Ω_2 is the silhouette illumination. To find out if a point on the surface of the 3D object is the outline or silhouette, we check if the surface normal (N) is perpendicular to the eye vector(camera ray), see Figure 4.3. It is computed as

$$\Omega_2 = clamp(1 - \vec{n} \cdot \vec{n_E}, min, max) \tag{4.8}$$

where $\vec{n_E}$ is the eye vector and *max* and *min* are used to control the size of the silhouette and smooth transition region.

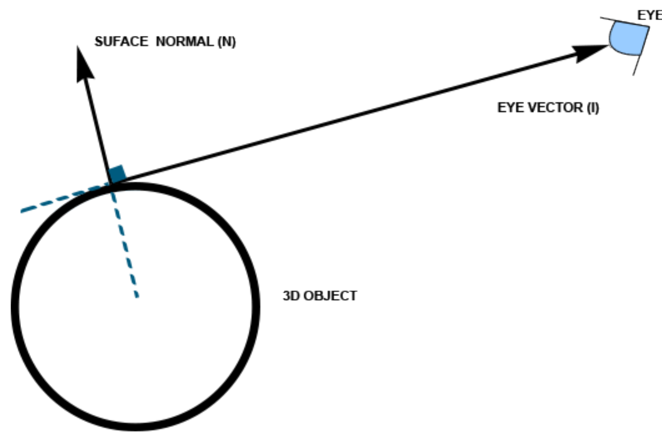


Figure 4.3: Surface normal(N) and eye vector/camera(I) ray are used to silhouette edge detection

Specular Illumination

Ω_3 is the lump sum of all specularly reflected light that reaches a given point. It is computed as

$$\Omega_3 = \sum_{i=0}^{N} sh_{L_i} \; s_{L_i} \; C_{L_i} \tag{4.9}$$

where N is the number of lights in the scene, C_{L_i} is the color of light source L_i, sh_{L_i} is the shadow parameter for the light L_i computed by ray tracer and s_{L_i} is the monotonically increasing function of $\cos\phi$ that is computed by dot product of $\vec{r_{L_i}}$, which is reflection of light vector $\vec{n_{L_i}}$ $\vec{n}$ and eye vector $\vec{n_E}$ as

$$s_{L_i} = clamp(\vec{n_E} \cdot \vec{r_{L_i}}, min, max) = clamp(\cos\phi, min, max) \tag{4.10}$$

In this equation, *max* and *min* are used to control the size of the specular highlight and its sharpness, shown in Figure 4.5.

Refraction and Reflection

Incorporating real-time reflection and refraction in a Barycentric shader is one of the main contributions of this thesis.

To get the reflection color C_R of a given point, reflection ray at that point is calculated. Figure 4.6 shows how we can use incoming ray (I) and the surface normal (N) to calculate the reflected ray (R).

To obtain the refraction color C_T at any point on the object, we first need to calculate

the refracted ray using the incident/incoming ray, surface normal and index of refraction, see Figure 4.7. The index of refraction (IOR) in nature typically ranges from 1/2 to 2. An examples is vaccum that has an IOR of 1 and water with an IOR of 1.333.

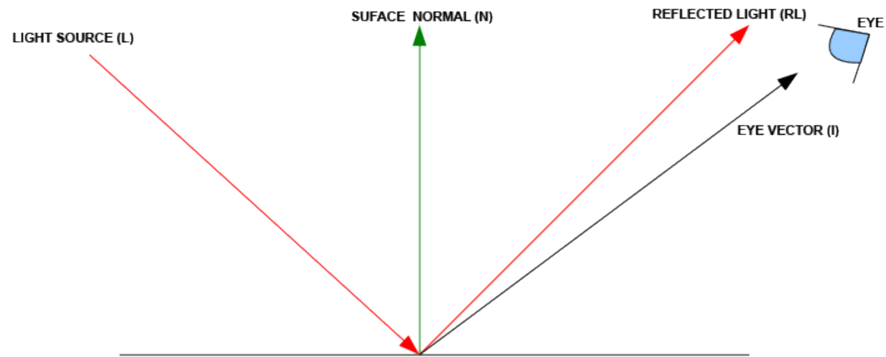


Figure 4.4: Specular computation

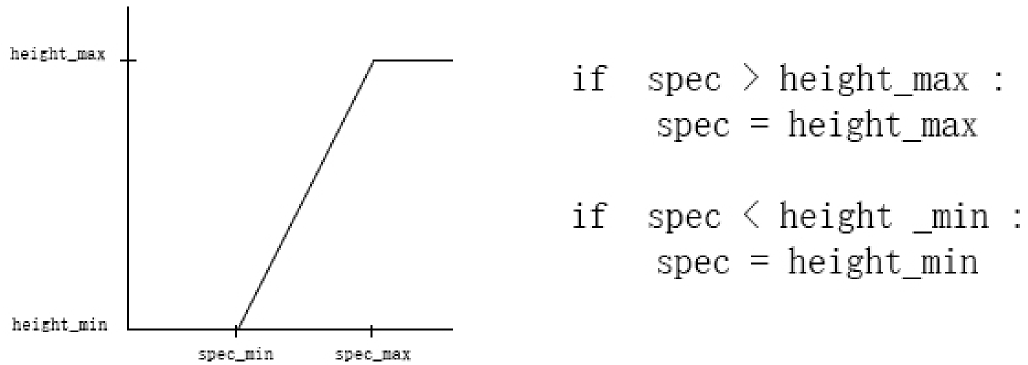


Figure 4.5: Clamping the specular value to obtain the desired specular highlight

The OptiX API [29] has a routine that calculates the refracted ray, based on the given incident/incoming ray, surface normal and index of refraction inputs, for a given point.

Ω_4 is the combination of reflection and refraction using Fresnel term f. It is computed as

$$\Omega_4 = fC_R + (1-f)C_T \qquad (4.11)$$

where C_R is the color of reflection and C_T is the color of refraction.

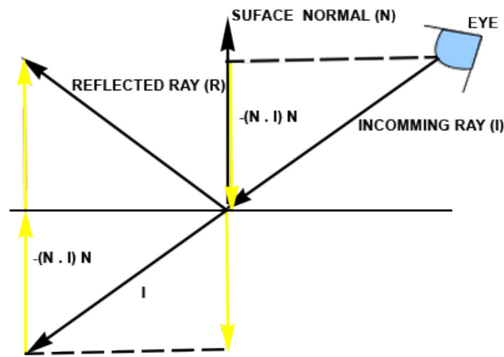


Figure 4.6: How to compute a standard reflection ray for an incoming ray

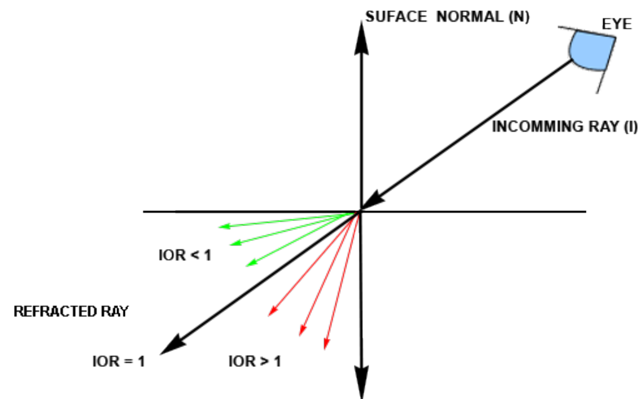


Figure 4.7: How to compute a standard refraction ray for an incoming ray

5 VISUAL ANALYSIS

In this work, we reconstruct Rachel Cunningham's digital painting, *Decido*, shown in Figure 5.1. For this we need to develop a set of shaders for every object in the scene that can produce a look-and-feel that is consistent with the objects in the painting.

Figure 5.1: 'Decido' - Primary visual reference. Digital painting by Rachel Cunningham

The generic Barycentric shader presented in section 4 provides a framework to simplify the process of creating shaders for the objects in the scene. In order to create a new

shader, we identify all the effects pertaining to the scene. Then analyse the layering technique used by the artist. This helped identify the shading hierarchy. The shader developed using this information can be used for all the objects in the scene. This can be done by using different texture maps, colors, and adjusting the effect parameters to achieve the desired result. Therefore, the main problems are reduced to the identification of effects and it's order of application.

This identification of effects can be achieved by careful visual analysis of the *paintings* we intend to reconstruct.

5.1 Visual Analysis

The purpose of this visual analysis is to discuss the artists' use of shading, form, shadow, reflection and refraction principles, that are then abstracted as effects. Since the goal of this research is to explore the possibility of creating a real-time art-directed 3D painting, the analysis emphasizes the artists style of painting, that is represented using the generic barycentric shader. The result of this yielded a list of essential characteristics and artistic aspects i.e effects and effect parameters used as guidelines for recreating the painting in CG. These can be categorized as follows:

1. Diffuse Reflection Shader
2. Specular Highlight Shader
3. Silhouette Shader
4. Transparency Shader

5.1.1 Diffuse Reflection Shader

In this subsection, we discuss color and tones of Cunningham's painting. In this analysis, we extract the darkest and brightest color from each region and create a color ramp for each region using the color values we sampled correspondingly. We use the Barycentric shading method, which interpolates between two colors/images; the brightest(illuminated) and darkest(unlit) color(image), that we call bright image and dark image. The resulting color (diffuse color) is an equation derived from $RGB\alpha$ channel of the bright and dark images.

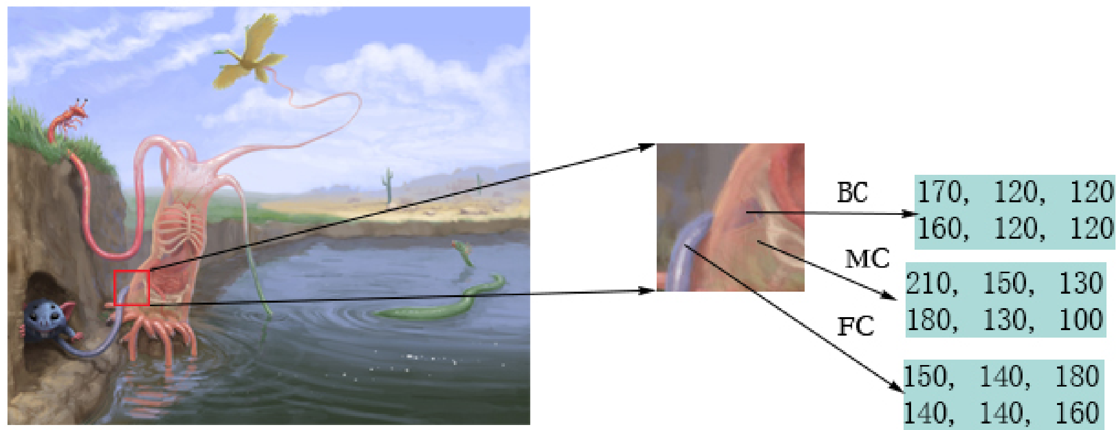


Figure 5.2: Color analysis of the main object in Cunningham's painting, Decido

In Figure 5.2, *FC* corresponds to the darkest and lightest foreground color of one of the tentacles. *BC* corresponds to the darkest and lightest color of the same tentacle behind the semi-transparent creature. And *MC* corresponds to the darkest and lightest color of the creature. The color values obtained by taking samples from the image are used in producing the texture map inputs for the Diffuse Reflection shader equation.

5.1.2 Specular Highlight Shader

The specular highlight in Cunningham's painting is different for different parts of the creature. On the body of the creature the specular is mostly diffused and, with a softer fall off. On the eye of the rodent and snake the specular highlight is sharp. See figure 5.3.

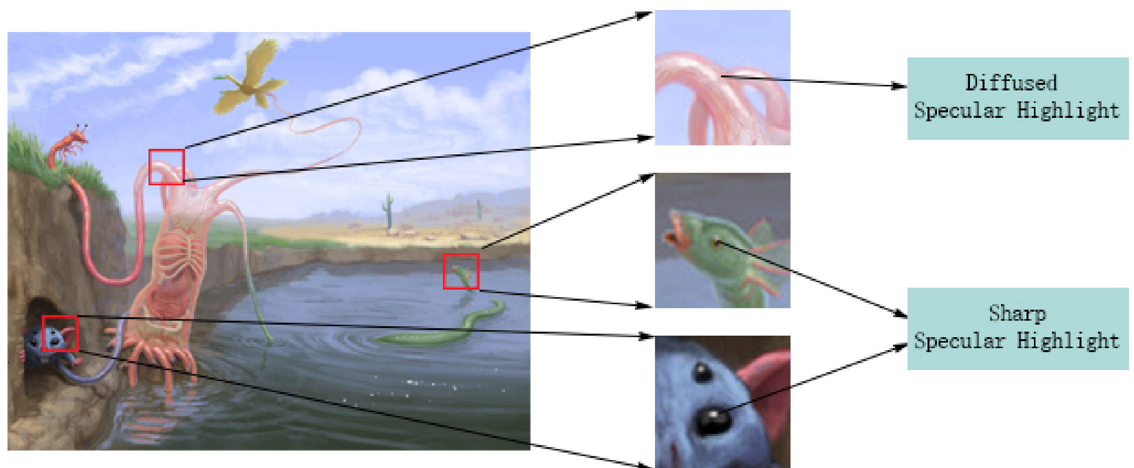


Figure 5.3: Specular highlight analysis in Cunningham's painting, Decido

5.1.3 Silhouette Shader

In this subsection, we discuss the edge or border that surrounds each character in the painting. On close examination of the painting the character is surrounded by a thin slightly jagged edge. The edge color is a slightly saturated value as compared to color of the body. See Figure 5.4

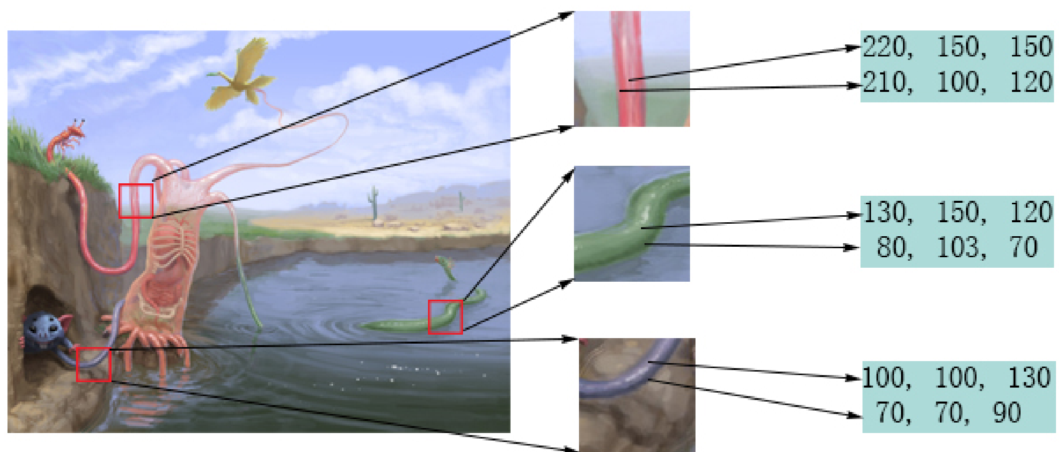


Figure 5.4: Color analysis of the silhouetted edge in Cunningham's painting, Decido

5.1.4 Transparency Shader

Reflection

The reflection in Cunningham's painting is mainly seen in the water. The water closer to the viewer has a hue that is darker as compared to that which is farther away from the viewer. The water is murky as there is no representation of what could be below the water surface. This reflection depicted is not physically accurate. By taking 2 sets of color samples (the darkest and the lightest) from the region close to the viewer and farther away from the viewer, we can map out the color range for the water based on the distance from the camera. See Figure 5.5. The reflection as we can see is mostly objects near the water, and there is no distinct sky reflection seen on the water surface. This can be handled by using a hand painted environment map that is used while calculating the water reflections.

f is the Fresnel parameter, that is calculated using Schick's approximation of Fresnel reflectance. This routine is provided by the OptiX API [29], this parameter is used

to determine how to combine the reflection and refraction colors. We can customize the parameter for the purposes of our thesis to make use of this routine and achieve the desired look and feel for the final result.

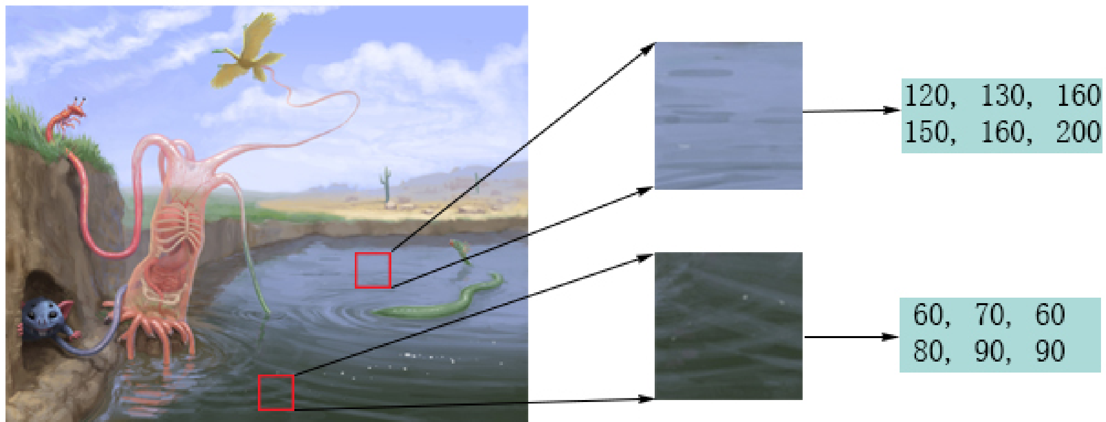


Figure 5.5: Color analysis of the water in Cunningham's painting, Decido

Refraction

In Cunningham's painting, refraction although is not seen in the water, she has incorporated it in the main creatures body. The color breakdown in Figure 5.2 shows one of the tentacles from behind the creature's body.

Transparency

We notice that the color of the tentacle is influenced by the a certain alpha factor or the color of the body. The creatures limbs are opaque and minimal refraction almost nearing transparency in the main body. To replicate this look, we hand painted a transparency map. For each point on the surface, the transparency value is obtained from the map, the value of which ranges from 0 to 1. 0 being completely transparent and 1 being completely opaque.

5.2 Shadow

Another goal of this thesis is the inclusion of artistic shadows. We compute shadows separately using ray tracing, provided by the Optix API [29]. While analysing the shadows in the painting, we notice that the color value of the shadowed regions in the painting do not go completely black (0,0,0). See Figure 5.6. This quality is very important to achieve in CG in order to achieve the painterly CG rendering. In order to replicate this effect using Barycentric shaders, we took two color/texture maps values for the same object, similar to how the diffuse reflection shader is calculated. In order to have artistic control of the shadows we introduced a shadow intensity parameter, Ω_1. This is then used to adjust the corresponding diffuse illumination, giving more control over the results.

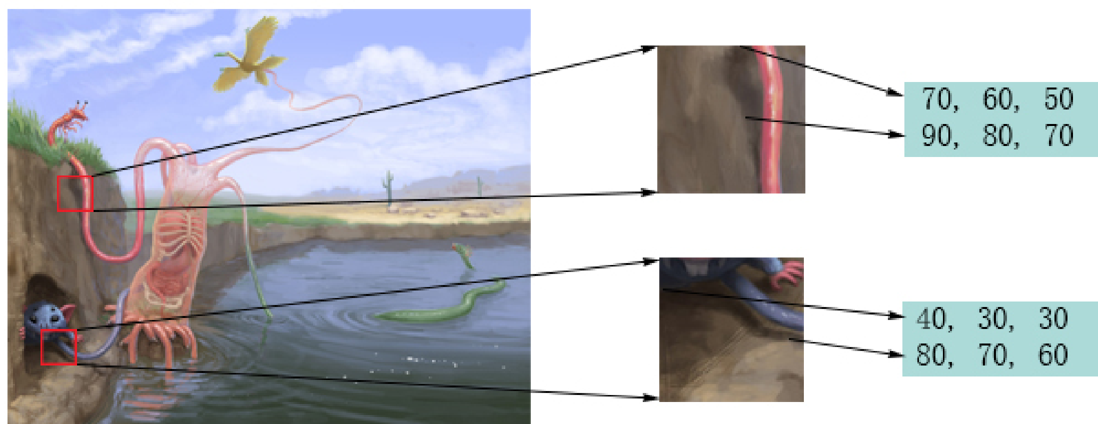


Figure 5.6: Color analysis of the shadows in Cunningham's painting, Decido

6 IMPLEMENTATION

This section is about the implementation of the Barycentric shading method for NPR as well as the process of creating the final 3D painting. Figure 6.1 illustrates the general process and the stages to complete the interactive painting.

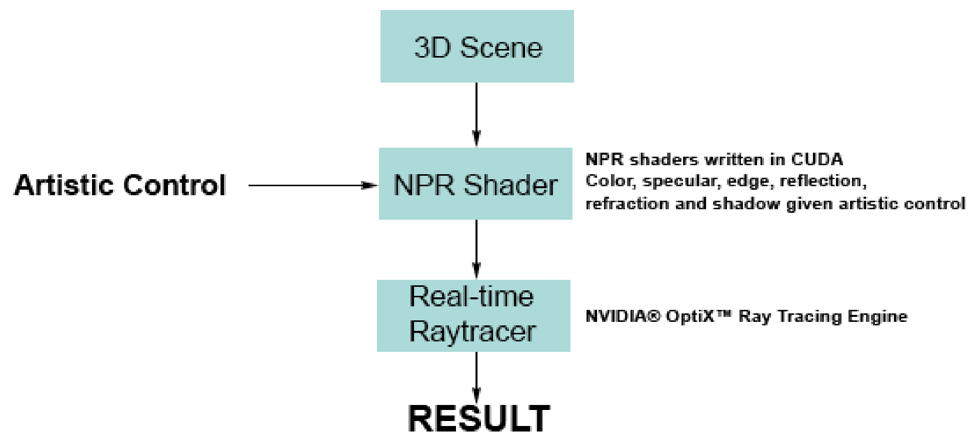


Figure 6.1: A flowchart showing the process for creating the interactive 3D painting

A 3D scene is created to represent the objects in the painting. This is shown in Figure 5.1. It consists of the main creature with the organs and its appendages, the land and the water. Then, the Barycentric shader for NPR is developed in CUDA and attached to each 3D mesh. The user has real-time artistic controls to adjust the effect parameters.

In this research we use Autodesk Maya for 3D modeling, CUDA for developing the shader and NVIDIA® OptiX™ ray tracing engine [29] to generate the interactive 3D painting. Adobe Photoshop and The Foundry Mari were used for painting and editing textures maps which were used in the Barycentric shader.

The primary visual reference is a digital painting created by Rachel Cunningham using Adobe Photoshop, called 'Decido', see Figure 5.1. Cunningham mavkes uses of the layering technique to add richness and depth to this painting. The characters in the painting have silhouetted edges, artistic shadows, reflections and transparency. These attributes made this painting a perfect choice to use as a test bed for this research.

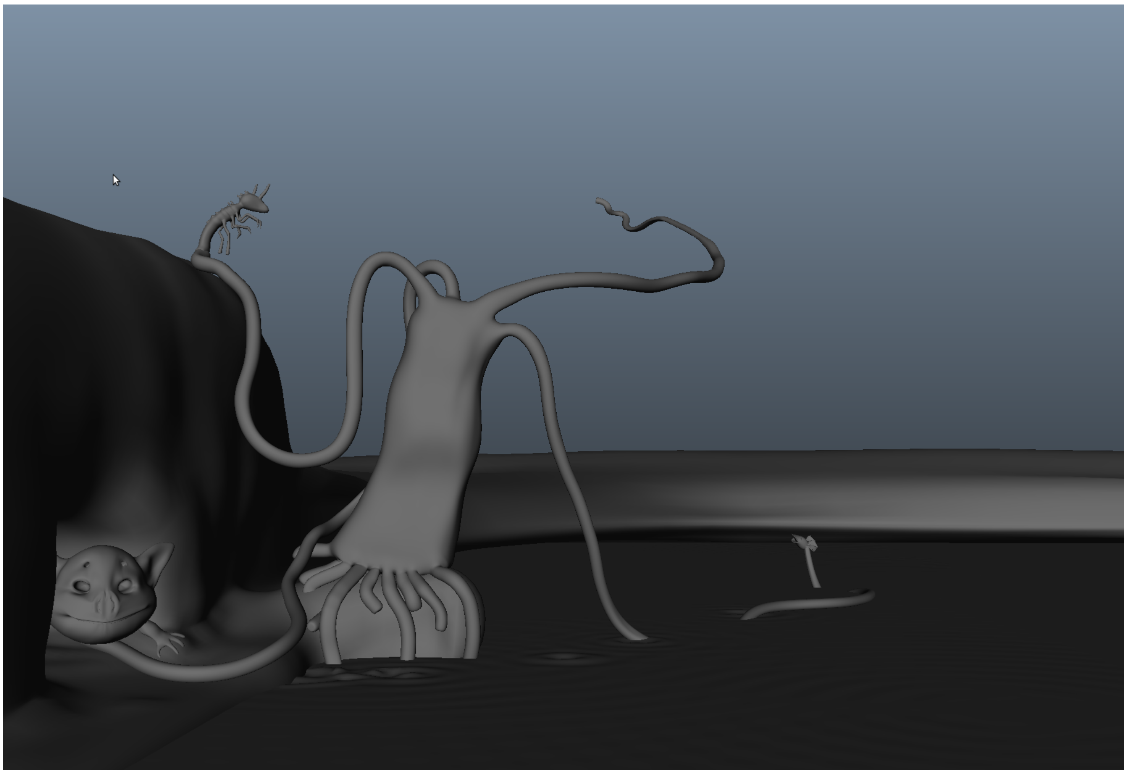


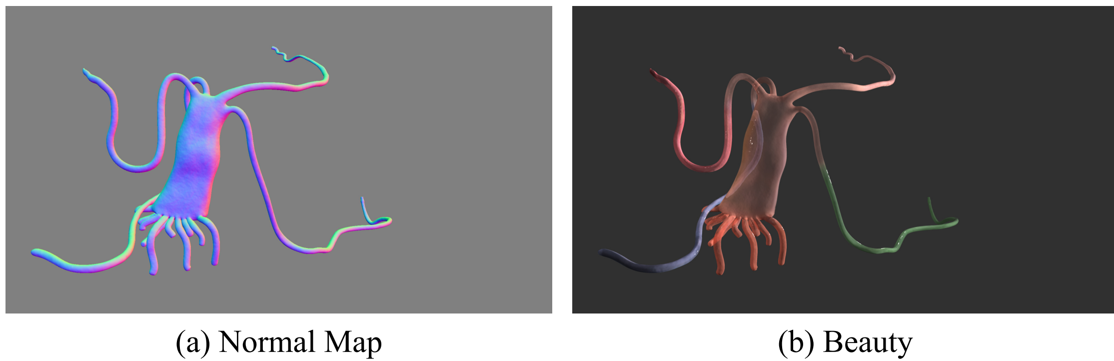
Figure 6.2: 3D scene modeled in Autodesk Maya

NVIDIA® OptiX™ ray tracing engine [29] is a general purpose ray tracing API. It increases the ray tracing speed using the NVIDIA® CUDA™ GPU computing architecture. It has built in routines that handle recursive ray generation, collision detection, environment mapping. Taking advantage of this technology, we developed a generic Barycentric shader

for non-photorealistic real-time rendering. This shader handles real-time reflections, refractions and shadows. This shader is largely based on the principle of the Barycentric Shaders developed by Akleman et al [1].

6.1 Scene Modeling and Development

We modeled the characters in the 3D scene to match the characters in the reference painting. The layout of the scene was designed to ensure the position of the characters were similar to that of the reference painting. Finally a virtual camera was created in Autodesk Maya to match the references' point of view. Figure 6.2 shows an image capture of the 3D scene.



(a) Normal Map (b) Beauty

Figure 6.3: Passes rendered of the Creature from Autodesk Maya using Mental Ray

6.1.1 Rendering Scene Using Mental Ray

To evaluate the visual aesthetics and the volume of the reconstructed scene, we first rendered the geometry out of Autodesk Maya using Mental Ray renderer. The textures for the 3D mesh were hand-painted, to achieve the same look and feel of the visual reference.

These textures were plugged into Mental Ray's shaders. We illuminated the scene with carefully placed virtual lights mimicking the lighting in the original painting. Shaders in Mental Ray didn't offer enough artistic control. We rendered out beauty(contains diffuse, specular, reflection, refraction and shadow color information), and normal passes for every object in the scene. This was done to have more control of the final image in post production. Figure 6.3 shows the beauty and normal map passes of one of the characters of the painting modeled in 3D.

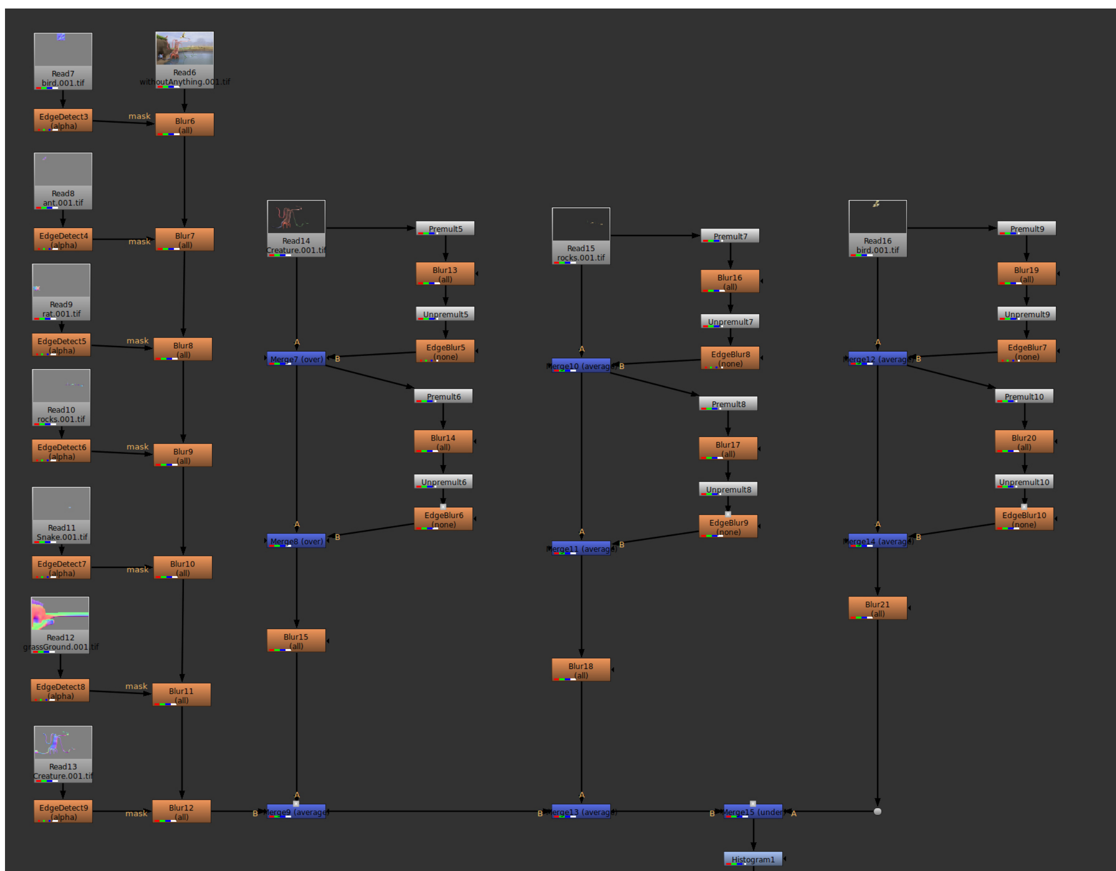


Figure 6.4: Post-processing of the 3D scene in Nuke

Figure 6.5: Final image of 3D scene

6.1.2 Digital Compositing/Post Processing

The rendered passes are imported into The Foundry's Nuke, a compositing software where post processing is done. Built-in nodes in Nuke which execute the basic 2D image operations are used to achieve the final image. We used the blur, grade, and edge detect nodes to assemble the passes for each of the objects in the scene. These are then combined using the merge node. See Figure 6.4 showing the post-processing done in Nuk. Figure 6.5 which shows the final image.

The scene was rendered with a moving virtual camera. Several passes were needed in order to have artistic control during post production. Each pass has a render time that ranges from a few seconds up to 2 minutes, in Autodesk Maya. The final sequence is rendered out of Nuke taking a minimum of 2 minutes per frame.

6.2 Creating a Generic Barycentric Shader for Non-Photorealistic Real-time Rendering

The crux of this thesis is to create an art-directable Barycentric shader for real-time rendering with GI. This shader is aimed at replicating the results, that were achieved in post production, in real-time. The NVIDIA® OptiX™ ray tracing engine [29] gave us a platform to explore this idea. The API is well suited for building custom shaders in CUDA. CUDA is a parallel computing platform and an API model created by NVIDIA®. We make use of CUDA to a build generic art-directable Barycentric shader for NPR in real-time.

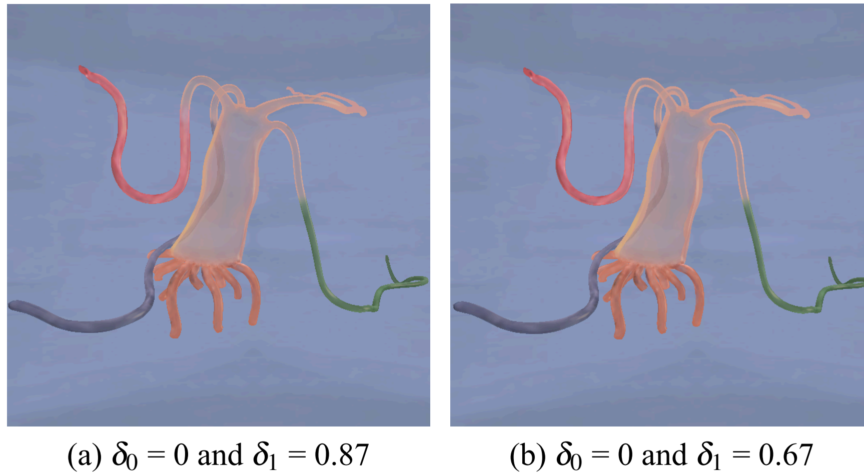
Figure 6.6: Real-time render of the main body of the creature with our custom shader

Based on the visual analysis done in section 5, we developed a procedural shader which computes the diffuse reflection, specular highlight, silhouette, transparency and shadow. While testing the shader on the objects, we quickly realised that not every object made use of every effect in the shader. This can be controlled by enabling the appropriate effect parameter. Below we describe the generic shader used for each object and the parameters provided to the user for art-directable control.

The customised barycentric shader that was used for the main creatures body, see Figure 6.6 contained effects which computes the diffuse reflection, specular highlight, silhouette, and transparency. The effect parameters for all were set to 1. The shader included controls for specific dark and light color/texture input, controls for the color of the silhouetted edges, and thickness of the edge. A control for transparency of the object, the index of refraction and controls for setting the specular quality. To compute the diffuse reflection, two hand painted textures, one dark and one light were given as inputs. Similarly, to compute the silhouette, a hand painted texture map was given as input and the edge parameter δ_0 and δ_1 values were defined. Figure 6.7 shows different thickness variations of the silhouetted edge.

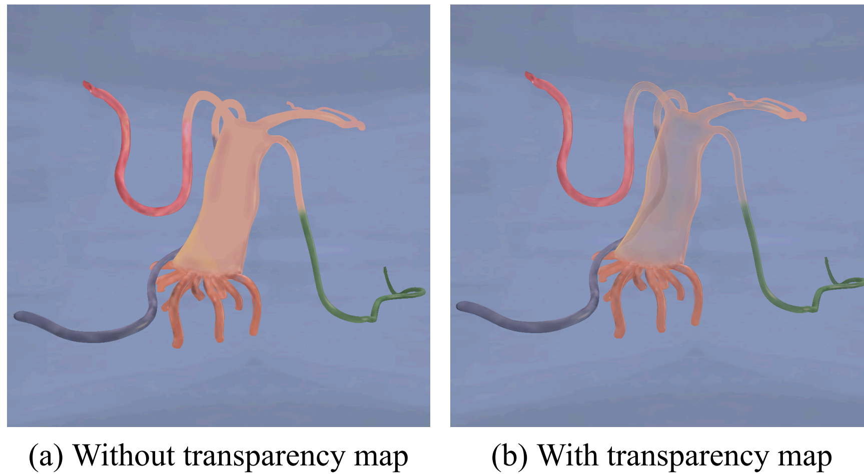
In order to control the transparency, a hand painted transparency map was provided as input to the shader, see Figure 6.8.

And finally to control the amount of refraction, we defined the index of refraction, fresnel minimum(0 to 1), fresnel maximum(0 to 1), fresnel exponent parameters, see Figure 6.9 (fresnel min = 0.1, fresnel maximum = 1.0, fresnel exponent = 3). NVIDIA's API takes



(a) $\delta_0 = 0$ and $\delta_1 = 0.87$ (b) $\delta_0 = 0$ and $\delta_1 = 0.67$

Figure 6.7: Main body of the creature with silhouette edge variations



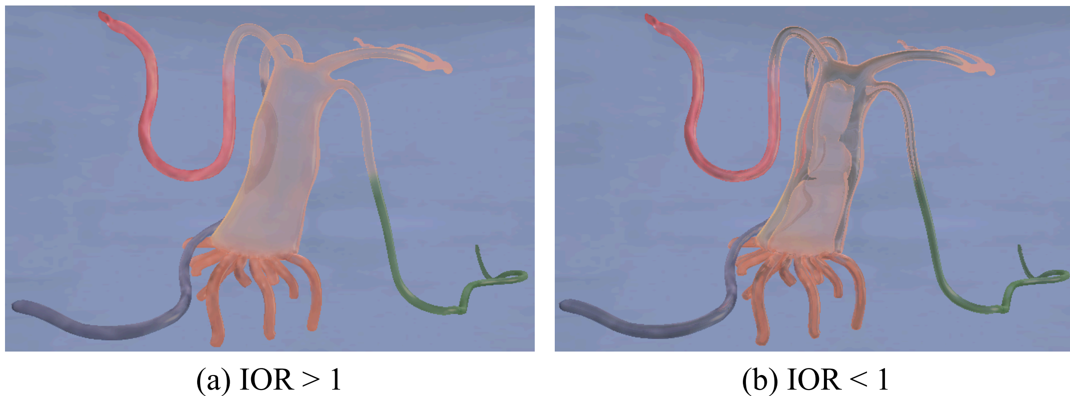
(a) Without transparency map (b) With transparency map

Figure 6.8: Main body of the creature with transparency variations

in the value of these inputs to compute the refraction and reflection color for a given point on the surface of the object.

Figure 6.10 shows hand painted textures for the creature's body. The color values for the textures were obtained by sampling colors from the original painting.

The customised barycentric shader used for the water in the scene computed for the transparency (reflection & refraction) and specular effects. Figure 6.11 shows the reflection



(a) IOR > 1 (b) IOR < 1

Figure 6.9: Main body of the creature with refraction variations

of the environment map on the surface of the water. The water geometry was created with ripples to match the look of the painting. Based on the visual analysis done in section 5, we created a custom sky map for the reflection on the water.

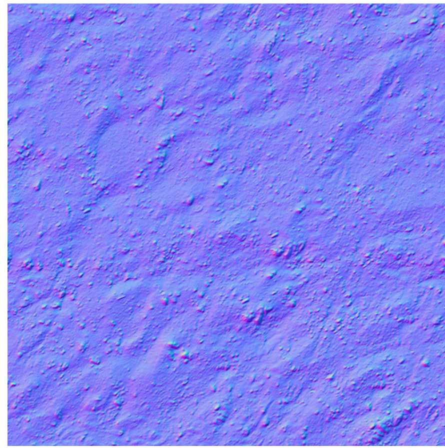
The customised barycentric shader used for the ant, rodent, ground, and cactus handled shadows, and bump mapping. The visual analysis discussed on shadows in section 5 indicates that shadows aren't completely black. In order to handle this the shader takes in a dark diffuse and a light diffuse color or texture map. And based on the shadow intensity or effect parameter (Ω_0) we interpolated between the two colors or texture maps. Figures 6.12, ??, ??, 6.13, show the custom texture maps used for the ant, rodent, ground and cactus geometry. These textures were hand painted, the color values for them were obtained by sampling colors from the original painting. Figure 6.14 which shows the shadow interpolations.

For the clouds and the bird, we used hand painted textures and normal maps. This shader computed the diffuse reflection and transparency effect to achieve the desired look. As seen in Figure 6.15.

(a) Diffuse - light

(b) Diffuse - dark

(c) Silhouette


(d) Normal

(e) Transparency

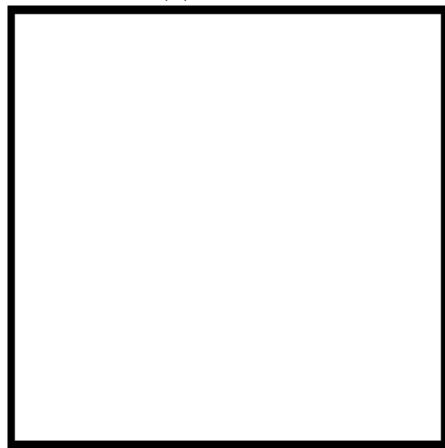

(f) Specular

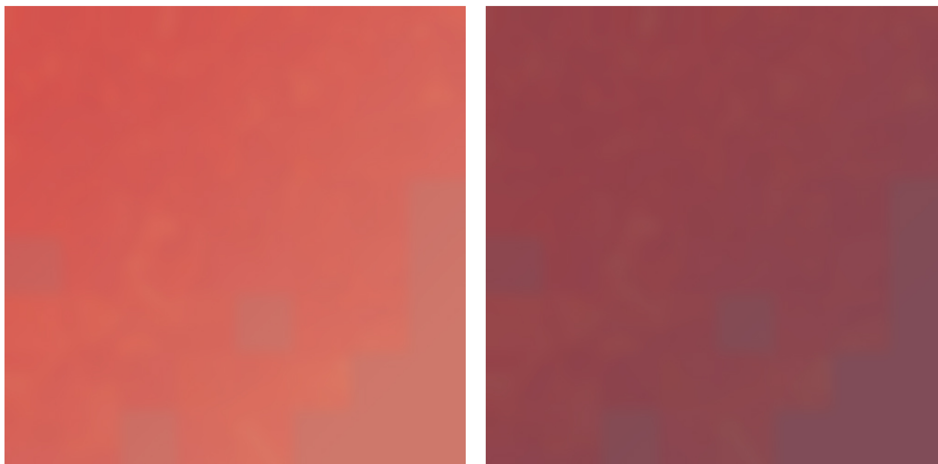
Figure 6.10: Custom textures used in the Body Shader

6.3 Results

Figure 6.16 shows a screen capture of rendered frame. This is the final result of the interactive 3D scene. The scene renders at 5 frames per second.

(a) Reflecting custom sky map (b) Reflecting environment map

Figure 6.11: Water geometry rendered real-time with custom shader



(a) Diffuse - light (b) Diffuse - dark

Figure 6.12: Custom textures used in the Ant Shader

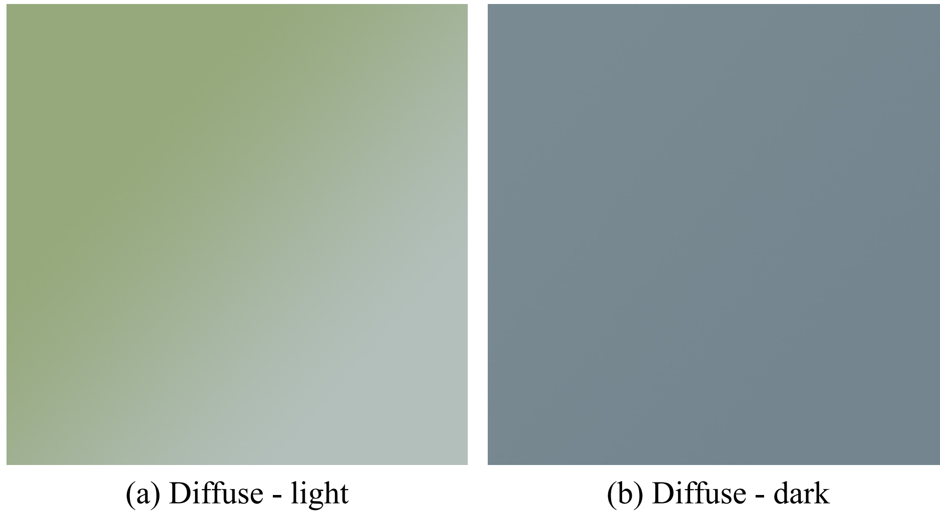

(a) Diffuse - light (b) Diffuse - dark

Figure 6.13: Custom textures used in the Cactus Shader

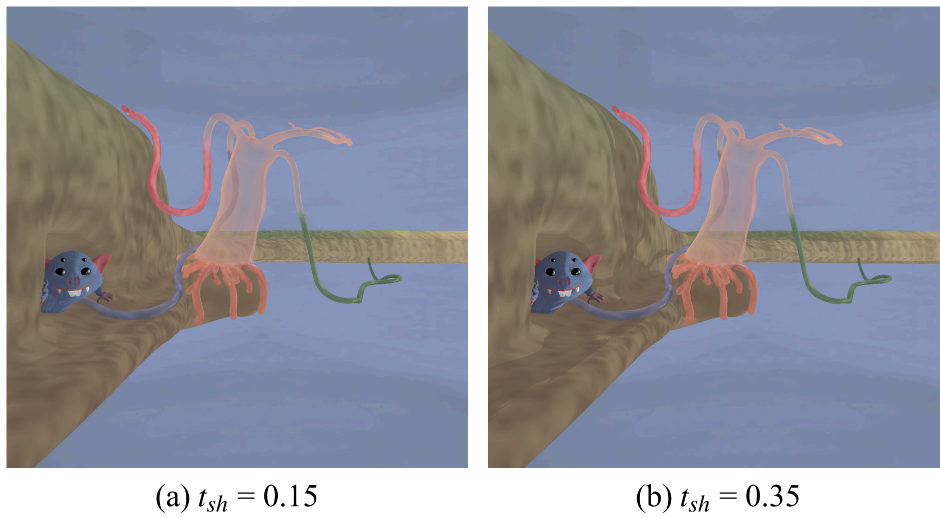

(a) t_{sh} = 0.15 (b) t_{sh} = 0.35

Figure 6.14: Shadow intensity variation seen on the ground and small creature

Figure 6.15: Clouds and bird rendered real-time with the custom shader

Figure 6.16: Frame of the final render

Comparison		
Property	In Maya using mental ray	Real-time using barycentric shaders
Time to render one frame	20s - 2 mins	0.2s
Real time interaction	No	Yes
Artistic Control	No	Yes
Color space control	No	Yes

Table 6.1: Comparison between mentral ray shader - maya rendering and barycentric shader - real time rendering

7 CONCLUSIONS AND FUTURE WORK

The goal of this research was to create a barycentric shader for NPR with GI for real-time raytracing. The shader we created also allows user control over diffuse, specular, reflection, refraction, transparency, silhouette and shadow effects. The shader was used in an example scenario to replicate the look of a painting. We used a digital painting by Rachel Cunningham *Decido* [6], as the primary visual reference.

In order to achieve the look and feel of the 2D painting in 3D, a visual analysis of the painting was conducted to understand the essential effects. We then substituted our findings in the generic barycentric shader to customize it for the characters in chosen painting. This work shows a proof of concept that a barycentric shader can be art directable in real-time NPR with GI. The shader and the control parameters it provided was effective in matching the look and feel of the painting.

Although this approach provides artistic control in real-time, this process still requires human efforts in terms of providing the 3D model of the painting, textures and normal maps in order to get the desired look. The extension of this work would include applying this shader concept to achieve NPR style real-time rendering with GI in augmented and virtual reality settings. Another extension of this work could include developing a barycentric shader to render grass, hair or fur in real-time.

REFERENCES

[1] E. Akleman, D. H. House, and S. Liu, “Barycentric shaders: Art directed shading using control images,” Proceedings of Expressive’2016: Computational Aestetics Conference, p. Accepted, 2016.

[2] R. H. Bartels, J. C. Beatty, and B. A. Barsky, An Introduction to Splines for Use in Computer Graphics &Amp; Geometric Modeling. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987.

[3] J. Chen, G. Turk, and B. MacIntyre, “Watercolor inspired non-photorealistic rendering for augmented reality,” in Proceedings of the 2008 ACM Symposium on Virtual Reality Software and Technology, ser. VRST ’08. New York, NY, USA: ACM, 2008, pp. 231–234. [Online]. Available: http://doi.acm.org/10.1145/1450579.1450629

[4] N. S.-H. Chu and C.-L. Tai, “Moxi: Real-time ink dispersion in absorbent paper,” ACM Trans. Graph., vol. 24, no. 3, pp. 504–511, Jul. 2005. [Online]. Available: http://doi.acm.org/10.1145/1073204.1073221

[5] R. L. Cook, “Shade trees,” SIGGRAPH Comput. Graph., vol. 18, no. 3, pp. 223–231, Jan. 1984. [Online]. Available: http://doi.acm.org/10.1145/964965.808602

[6] R. Cunningham, “Decido(decisions),” Digital Painting, Available: http://www.rachelcunninghamwang.com/digital-painting.html, 2012, [Online].

[7] C. J. Curtis, “Brick-a-Brac,” Available:http://otherthings.com/uw/pigeons/brick2.html, 1995, [[Online]; https://vimeo.com/118034969].

[8] ——, “Loose and sketchy animation,” in ACM SIGGRAPH 98 Electronic Art and Animation Catalog, ser. SIGGRAPH ’98. New York, NY, USA: ACM, 1998, pp. 145–. [Online]. Available: http://doi.acm.org/10.1145/281388.281913

[9] ——, “The New Chair,” 1998, [[Online]; https://vimeo.com/119315695].

[10] D. Curtis, Cassidy J.and Gainey, “Fishing,” 1999, [[Online]; https://vimeo.com/118042611de].

[11] P. Decaudin, “Cartoon looking rendering of 3D scenes,” INRIA, Research Report 2919, Jun. 1996. [Online]. Available: http://phildec.users.sf.net/Research/RR-2919.php

[12] Y. Du and E. Akleman, “Charcoal rendering and shading with reflections,” in ACM SIGGRAPH 2016 Posters, ser. SIGGRAPH ’16. New York, NY, USA: ACM, 2016.

[13] A. Goldberg and R. Flegal, “ACM president’s letter: Pixel art,” Commun. ACM, vol. 25, no. 12, pp. 861–862, Dec. 1982. [Online]. Available: http://doi.acm.org/10.1145/358728.358731

[14] A. Gooch, B. Gooch, P. Shirley, and E. Cohen, “A non-photorealistic lighting model for automatic technical illustration,” in Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, ser. SIGGRAPH

’98. New York, NY, USA: ACM, 1998, pp. 447–452. [Online]. Available: http://doi.acm.org/10.1145/280814.280950

[15] P. Haeberli, “Paint by numbers: Abstract image representations,” in Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques, ser. SIGGRAPH ’90. New York, NY, USA: ACM, 1990, pp. 207–214. [Online]. Available: http://doi.acm.org/10.1145/97879.97902

[16] P. Hanrahan and J. Lawson, “A language for shading and lighting calculations,” SIGGRAPH Comput. Graph., vol. 24, no. 4, pp. 289–298, Sep. 1990. [Online]. Available: http://doi.acm.org/10.1145/97880.97911

[17] A. Hertzmann, “Painterly rendering with curved brush strokes of multiple sizes,” in Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, ser. SIGGRAPH ’98. New York, NY, USA: ACM, 1998, pp. 453–460. [Online]. Available: http://doi.acm.org/10.1145/280814.280951

[18] A. Hertzmann and K. Perlin, “Painterly rendering for video and interaction,” in Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering, ser. NPAR ’00. New York, NY, USA: ACM, 2000, pp. 7–12. [Online]. Available: http://doi.acm.org/10.1145/340916.340917

[19] N. Imhof, A. Milliez, F. Jenal, R. Bauer, M. Gross, and R. W. Sumner, “Fin textures for real-time painterly aesthetics,” in Proceedings of the 8th ACM SIGGRAPH

Conference on Motion in Games, ser. MIG '15. New York, NY, USA: ACM, 2015, pp. 227–235. [Online]. Available: http://doi.acm.org/10.1145/2822013.2822021

[20] 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 Trans. Graph., vol. 21, no. 3, pp. 755–762, Jul. 2002. [Online]. Available: http://doi.acm.org/10.1145/566654.566648

[21] M. Kaplan, B. Gooch, and E. Cohen, "Interactive artistic rendering," in Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering, ser. NPAR '00. New York, NY, USA: ACM, 2000, pp. 67–74. [Online]. Available: http://doi.acm.org/10.1145/340916.340925

[22] 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 Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, ser. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 433–438. [Online]. Available: http://dx.doi.org/10.1145/311535.311607

[23] J. Lee, "Diffusion rendering of black ink paintings using new paper and ink models," Computers Graphics, vol. 25, no. 2, pp. 295 – 308, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0097849300001321

[24] P. Litwinowicz, "Processing images and video for an impressionist effect," in Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, ser. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 407–414. [Online]. Available: http://dx.doi.org/10.1145/258734.258893

[25] S. Liu and E. Akleman, "Chinese ink and brush painting with reflections," in SIGGRAPH 2015: Studio, ser. SIGGRAPH '15. New York, NY, USA: ACM, 2015, pp. 8:1–8:1. [Online]. Available: http://doi.acm.org/10.1145/2785585.2792525

[26] J. Lu, P. V. Sander, and A. Finkelstein, "Interactive painterly stylization of images, videos and 3d animations," in Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, ser. I3D '10. New York, NY, USA: ACM, 2010, pp. 127–134. [Online]. Available: http://doi.acm.org/10.1145/1730804.1730825

[27] M. McGuire and A. Fein, "Real-time rendering of cartoon smoke and clouds," in Proceedings of the 4th International Symposium on Non-photorealistic Animation and Rendering, ser. NPAR '06. New York, NY, USA: ACM, 2006, pp. 21–26. [Online]. Available: http://doi.acm.org/10.1145/1124728.1124733

[28] B. J. Meier, "Painterly rendering for animation," in Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, ser. SIGGRAPH

’96. New York, NY, USA: ACM, 1996, pp. 477–484. [Online]. Available: http://doi.acm.org/10.1145/237170.237288

[29] NVIDIA, “Nvidia® optix™ ray tracing engine,” Available:https://developer.nvidia.com/optix, 2016. [Online]. Available: https://developer.nvidia.com/optix

[30] S. C. Olsen, B. A. Maxwell, and B. Gooch, “Interactive vector fields for painterly rendering,” in Proceedings of Graphics Interface 2005, ser. GI ’05. School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada: Canadian Human-Computer Communications Society, 2005, pp. 241–247. [Online]. Available: http://dl.acm.org/citation.cfm?id=1089508.1089548

[31] F. Pfenning, “NPR,” Available:http://www.cs.cmu.edu/~fp/courses/graphics/pdf-2up/21-npr.pdf, 2003, [Online]; pp. 3–, accessed 17 April 2003.

[32] M. P. Salisbury, S. E. Anderson, R. Barzel, and D. H. Salesin, “Interactive pen-and-ink illustration,” in Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, ser. SIGGRAPH ’94. New York, NY, USA: ACM, 1994, pp. 101–108. [Online]. Available: http://doi.acm.org/10.1145/192161.192185

[33] M. Shiraishi and Y. Yamaguchi, “An algorithm for automatic painterly rendering based on local source image approximation,” in Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering, ser. NPAR ’00.

New York, NY, USA: ACM, 2000, pp. 53–58. [Online]. Available: http://doi.acm.org/10.1145/340916.340923

[34] G. Winkenbach and D. H. Salesin, “Computer-generated pen-and-ink illustration,” in Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, ser. SIGGRAPH ’94. New York, NY, USA: ACM, 1994, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/192161.192184

APPENDIX A

SHADER CODE

```
static __device__ void nprShader(    float importance_cutoff,
                                     float3 cutoff_color,
                                     float fresnel_exponent,
                                     float fresnel_minimum,
                                     float  fresnel_maximum,
                                     float refraction_index,
                                     int refraction_maxdepth,
                                     int reflection_maxdepth,
                                     float borderWidthParameter,
                                     float3 refraction_color,
                                     float3 reflection_color,
                                     float3 extinction_constant,
                                     float3 diffuse_light,
                                     float3 diffuse_dark,
                                     float3 spec_color,
                                     float3 border_color,
                                     float3 normalmap_color,
                                     float3 transparencymap_color,
                                     boolean k1,
                                     boolean k2,
                                     boolean k3,
```

```
                                    boolean k4)
{

    //————surface normal————//
    const float3 normal = normalize(rtTransformNormal(
   RT_OBJECT_TO_WORLD, shading_normal));

    //————front hit point————//
    const float3 fhp = rtTransformPoint(RT_OBJECT_TO_WORLD,
   front_hit_point);

    //————back hit point————//
    const float3 bhp = rtTransformPoint(RT_OBJECT_TO_WORLD,
   back_hit_point);

    //————world geometric normal————//
    float3 world_geometric_normal = normalize(rtTransformNormal(
   RT_OBJECT_TO_WORLD, geometric_normal));

    //————incident direction ————//
    const float3 i = ray.direction;

    //————transmission direction ————//
         float3 t;
```

```
//————reflection direction———//
        float3 r;
//————tangent to the shading normal———//
float3 shading_tangent = make_tangent(shading_normal);

//————world shading normal—————//
const float3 world_shading_normal  = normalize( rtTransformNormal(
RT_OBJECT_TO_WORLD, shading_normal ) );

//————tangent to the world shading normal————//
const float3 world_shading_tangent  = normalize( rtTransformNormal(
RT_OBJECT_TO_WORLD, shading_tangent ) );

//————reverse normal—————//
const float revn = copysignf( 1.f, dot(-ray.direction,
world_geometric_normal));

//————face forward normal———//
float3 ffnormal = world_shading_normal * revn;

//————————texture/color inputs————————//
float3 I0 = diffuse_light;
float3 I1 = diffuse_dark;
float3 I3 = spec_color;
float3 I4 = border_color;
```

```
//-----------calculated color outputs----------//
float3 I1_dash;
float3 I2_dash;
float3 I3_dash;
float3 I4_dash;

float reflection = 1.0f;
float3 result = make_float3(0.0f);
float3 cT = make_float3(0.0f);
float3 cR = make_float3(1.0f,1.0f,1.0f);

const int depth = prd_radiance.depth;
float3 beer_attenuation;

//------------Beer's law attenuation---------//
if(dot(normal, ray.direction) > 0) {
  beer_attenuation = exp(extinction_constant * t_hit);
} else {
  beer_attenuation = make_float3(1);
}

//-------refraction calculation ----//
if (refraction_index != 0.0){
    if (depth < min(refraction_maxdepth, max_depth))
```

```
    {
      if ( refract(t, i, normal, refraction_index) )
      {
        //————check for external or internal reflection ———————//
        float cos_theta = dot(i, normal);
        if (cos_theta < 0.0f)
          cos_theta = −cos_theta;
        else
          cos_theta = dot(t, normal);



        //———fresnel_schlick method provided by the OptiX API
————//
        reflection = fresnel_schlick(cos_theta, fresnel_exponent,
fresnel_minimum, fresnel_maximum);

        float importance = prd_radiance.importance * (1.0f−
reflection) * optix::luminance( refraction_color * beer_attenuation
);
        float3 color = cutoff_color;
        if ( importance > importance_cutoff ) {
          color = TraceRay(bhp, t, depth+1, importance);
        }
        cT += (1.0f − reflection) * refraction_color * color;
      }
```

```
    }


    //------reflection calculation------//
    float3 cutoffColor = cutoff_color;
    if (depth < min(reflection_maxdepth, max_depth))
    {
      r = reflect(i, normal);


      float importance = prd_radiance.importance * reflection *
optix::luminance( reflection_color * beer_attenuation );
      if ( importance > importance_cutoff ) {
        color = TraceRay( fhp, r, depth+1, importance );
      }
    }


    cT += reflection * reflection_color * cutoffColor;


    I2 = cR*cT;
 }




 float3 p_normal = faceforward( normal, -ray.direction,
world_geometric_normal );


 //--------reflected ray----------//
```

```
float3 reflectedRay = reflect( ray.direction , p_normal );


//--------surface hit point--------//

float3 hit_point = ray.origin + t_hit * ray.direction;

unsigned int num_lights = lights.size();


//-------Diffuse Reflection effect calculation-------//

const float3 temp = make_float3(0.5f,0.5f,0.5f);

const float3 normalMapConvert = 2.0f*(normalmap_color - temp);


const float3 ndash = normalize((normal+normalMapConvert)/2.0f);


float tnew  = fabs(dot(-normalize(ray.direction),normalize(normal))
);

omega_0= pow(tnew,4.0f);



//-------Silhouette effect calculation-------//

float bnew = 1 - tnew;

if(bnew>borderWidthParameter){

    omega_1 = 1.0f;

}

else{

    omega_1 = 0.0f;

}
```

```
//———transparency effect calculation ——————//
float omega_2  = (transparencymap_color.z);

//———Mixing Diffuse Colors ——————//
I1_dash = I0*omega_0*k1 + I1*(1-omega_0*k1);

//———Mixing Silhouette with Diffuse ——————————//
I2_dash = I4*omega_1*k2 + I1_dash*(1-omega_1*k2);

//———Mixing transparency color with Border and Diffuse ————————//
//——————— transparency omitted in the silhouette ———————————//
if(omega_1 == 0.0f){
    I3_dash =  I2*(omega_2*k3) + I2_dash*(1-omega_2*k3) ;
    }



//———Specular highlight calculation ——————//
I4_dash = I3_dash;
//———looping through lights ——————//
for(int i = 0; i < num_lights; ++i) {
  BasicLight light = lights[i];

  //————light vector —————————//
  float3 L = optix::normalize(light.pos - hit_point);
```

```
    //------Specular Component------//
    float snew =  dot(reflectedRay ,L);
    snew= (snew  -0.90f)/(0.99f-0.90f);
    if(snew > 1.0f){
        snew = 1.0f;
    }
    if(snew < 0.0f){
        snew = 0.0f;
    }
    omega_3=snew*0.3f;

    //----mixing specular highlight with color calculated thus far
  ----//
    I4_dash = I3*omega_3*k4 + I4_dash*(1-omega_3*k4) ;
  }

  //----------returning calculated final color--------//
  prd_radiance.result = I4_dash;
}
```

Listing A.1: Barycentric shader function written in CUDA