Exploring Real-Time Image Space Painterly Rendering

by

Michael P. Krazanowski
B.Sc., Malaspina University College, 2001

A Thesis Submitted in Partial Fullfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Michael Patrick Krazanowski, 2011
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

Exploring Real-Time Image Space Painterly Rendering

by

Michael P. Krazanowski
B.Sc., Malaspina University College, 2001

Supervisory Committee

Dr. Brian Wyvill, Supervisor
(Department of Computer Science)

Dr. Amy Gooch, Departmental Member
(Department of Computer Science)

Dr. Paul Lalonde, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Brian Wyvill, Supervisor
(Department of Computer Science)

Dr. Amy Gooch, Departmental Member
(Department of Computer Science)

Dr. Paul Lalonde, Departmental Member
(Department of Computer Science)

ABSTRACT

Artists have been using brush strokes to generate abstractions and interpretations of the world they view, however this process is very time consuming. Many recent and diverse techniques attempt to mimic the process on a computer that an artist would go through to generate a painting; an area of research is affectionately nick-named “painterly rendering”. These applications emulate the effects of an artist’s handiwork with dramatically less time invested in the process. I present a method to adapt painterly rendering for real-time simulations or video games, such that the images may appear to have been painted by hand. The work described in this document focuses on three problem areas posed for a real-time solution for painterly-rendering: brush stroke generation, temporal coherence between frames and performance. The solution presented here intends to solve these three fundamental issues with the intent to layer these methods on top of real-time applications using current generation consumer hardware.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents iv

List of Tables vii

List of Figures viii

Acknowledgements x

Dedication xi

1 Introduction 1

1.1 Real-Time Painterly Rendering . . . 1

1.2 My Contributions . . . 4

1.3 Document Overview . . . 5

2 The Problem to be Solved 7

2.1 Non-Photo-Realistic Rendering . . . 9

2.2 The Photorealistic-Nonphotorealistic Gap . . . 11

2.3 Real-Time Stroke-Based Painterly Rendering for Video Games . . . 13

2.3.1 Image-Space Real-Time Painterly Rendering (IsRaPtRS) . . . 14

2.3.2 Performance: Hardware Acceleration . . . 14

2.3.3 Quality . . . 15

3 Painterly rendering for real-time applications 17

3.1 Performance . . . 17

3.1.1 Performance Overview . . . 17

3.1.2 Performance Requirements 18
3.1.3 Performance Plan 19
3.2 Painterly Rendering 21
3.2.1 Paint Stroke Attributes 22
3.2.2 State Of The Art 23

4 Experiments 25
4.1 High Level Algorithm 25
4.1.1 Algorithm Discussion 28
4.2 Multiscale Colour and Intensity Buffers 28
4.2.1 Multiscale Colour 29
4.2.2 Multiscale Intensity 30
4.2.3 Performance Considerations 30
4.3 Multiscale Orientation and Detail Buffers 30
4.3.1 Gradient Calculation 31
4.3.2 Mutiscale Orientation 31
4.3.3 Mutiscale Detail 31
4.4 Orientation and Detail Performance Considerations 32
4.5 Mutiscale Performance Considerations 33
4.6 Frame Coherence 33
4.6.1 Animating Paint Strokes 34
4.6.2 Optical Flow 35
4.6.3 Image-Space Optical Flow Quality and Usage Considerations . 37
4.7 Paint-Stroke Rendering 40
4.7.1 Paint-Stroke Detail 40
4.7.2 Paint-Stroke Orientation 40
4.7.3 Paint-Stroke Colour, Shape and Material 40
4.7.4 Paint-Stroke Position 41

5 Evaluation, Analysis and Comparisons 46
5.1 Performance Analysis 46
5.1.1 Calculating Paint Stroke Data Buffers 47
5.1.2 Cost of Rendering Paint Strokes 48
5.2 Qualitative Analysis 49
5.2.1 Multi Scale Colour Error Analysis 49

5.2.2 Optical Flow Error Analysis 50
5.3 Memory Cost 52
5.4 The User Interface 53
5.4.1 Animation Type 55
5.4.2 Optical Flow Display Threshold 57

6 Conclusions 63
6.1 Performance 63
6.2 Image Quality 66
6.2.1 Personal Observations on Quality 66
6.3 Motion 67
6.4 Future Work 68
6.4.1 Comparing Geometric Advection Methods 68
6.4.2 Discrete Particle Image-Space Advection 69
6.4.3 3D Models and Video 69
6.4.4 Surface Properties 69
6.4.5 Performance Profiling 70
6.4.6 Measure Optical-Flow vs Geometric Movement Results 70
6.4.7 Measure The Error Introduced Through Vector Compression . 71

A Additional Information 72
A.1 Formulas 72
A.2 Example Images 73
A.3 Glossary 74

Bibliography 82

List of Tables

Table 5.1 Calculation of painterly buffer cost 53

List of Figures

Figure 3.1 GPU Pipeline 19

Figure 4.1 Diagram of painterly rendering data calculation 27
Figure 4.2 Multiscale Gradient Calculation, <x, y> displayed as <red, blue> respectively. 42
(a) Very-high 42
(b) High 42
(c) Medium-high 42
(d) Medium 42
(e) Medium-low 42
(f) Low 42
Figure 4.3 Local window optical flow example 43
Figure 4.4 Paint stroke parameters 43
Figure 4.5 Vacation picture of the Oregon coast 44
Figure 4.6 Oregon coast that uses multi scale paint strokes 44
(a) Very-high 45
(b) High 45
(c) Medium-high 45
Figure 4.7 Multi scale paint strokes 45
(a) Medium 45
(b) Medium-low 45
(c) Low 45

Figure 5.6 Screenshot of the user interface 56
Figure 5.1 Comparing Different Source Images and Different Source Image Resolutions 58
Figure 5.2 Comparing Different Source Images as They are Rendered with Varying Paint Stroke Lengths 59

Figure 5.3 Comparing (per-pixel) the painterly image colour deviation from the source image. 60
Figure 5.4 Measuring the percentage of pixels that are breaching their extents at any given time due to advection. 61
Figure 5.5 Measuring the percentage of pixels that have more than one advection vector competing for transfer at any given time due to advection. 62

(a) Butterfly 77
(b) CraterLake 77
(a) Dunes 78
(b) Flowers 78
(a) Geese 79
(b) OregonCoast 79
Figure A.1 Example painterly images 80
(a) Rafting 80
(b) Sanfrancisco 80
Figure A.2 Example painterly images 81
(a) SteamTrain 81
(b) Waterfall 81

Acknowledgements

I would like to thank all those people and organizations that helped make this thesis possible, in particular my supervisory committee for agreeing to oversee the following research.

I would like to thank Brian Wyvill for giving me this opportunity to work on research academically, for securing funding for this endeavor and for his encouragement and support throughout my time at the University of Victoria. I would also like to thank him for suggesting a topic outside of my bailiwick, giving me the opportunity to grow in areas that I wouldn't be able to otherwise.

I would also like to thank the computer science department, its professors and its industry partners at Intel for sharing their knowledge and expertise and always going above and beyond to help me in whatever way I needed. The enthusiasm for the material and willingness to assist their students to learn the material was exemplary.

A special thanks goes out to the graphics and games group for their continued commitment to try to promote the education of that magical grey area between science and art and to convey the practical knowledge between the needs of academia and industry. In particular I would like to thank Herbert and Pourya for being great lab-mates, for their assistance in teaching and research and for sharing their knowledge in areas I was unfamiliar with.

I would like to thank Dr. Micaela Serra specifically for taking time out of her busy schedule to help answer those questions that aren't easily answered elsewhere, to advise me on all aspects of life and for always having good chocolate on hand.

Finally I would like to thank the organizations that helped to fund the research contained within this thesis as well as the other work I did during my time at the University of Victoria. I would like to thank the National Sciences and Engineering Research Council (NSERC) and the Graphics Animation and New Media (GRAND) for providing the bulk of the funding for this research.

DEDICATION

To my boys, Corbin and Malcolm and my parents.

Chapter 1

Introduction

For thousands of years, humans have been creating pictorial images and have been mastering the skills necessary to effectively share ideas through this medium [1]. For most of our history these images have been created by painting or drawing, a very time consuming process. Only in very recent history have we been able to use the technology of the camera to create images but at a much faster rate.

It has been argued [2] that the photograph simply captures the world as it is, limiting the artist's ability to convey their interpretation or abstraction of the world. During the time before the camera, artists would spend years learning how to interpret, abstract and represent the world through paintings and how to use paintings to direct the viewer's focus. Images created by paintings are potentially more effective than an image created through a camera.

Context can be defined only by the content in a photograph and its effectiveness is limited compared to its painted counterpart. Artists have more control over interpretation and abstraction in a medium like painting than with a camera.

1.1 Real-Time Painterly Rendering

The following document describes my algorithm for a stroke-based painterly-rendering applicable for real-time applications.

My research is an attempt to define a system that allows stroke-based painterly-rendering (SBR) to generate images that look like paintings (hand-painted images) that can be updated at an interactive rate. Stroke-based-rendering, as surveyed by Hertzmann [3], is an automatic non-photo-realistic painterly rendering approach that defines the characteristics of and renders discrete paint-strokes in an attempt to mimic the work of a painter with their paint and brush.

A wide availability of computer graphics makes it possible to transform a photographic image to an image that mimics a hand-drawn or hand-painted one with the speed of a camera. The field of computer science that encompasses the transformation from raw visual data to an abstract visual representation (such as a painting) is called non-photorealistic rendering (NPR). A good reference for this broad field is encompassed in the books: Gooch & Gooch *"Non-photorealistic rendering"* [4] and Strothotte & Schlechtweg *"Non-photo-realistic computer graphics: modeling, rendering, and animation"* [5].

There has been a lot of research into how to create NPR images in many different styles but many of these are only appropriate for off-line, or non-time-sensitive applications. There has been comparatively little work in real-time painterly rendering research. Real-time painterly rendering can be too computationally expensive, can require too much authoring to create the paintstrokes or can require too much user interaction to adjust the system parameters to create the content that results in an NPR image.

Recently there has been work attempting to create real-time applications that produce a painterly rendering style [6], but there has been no research to determine the performance cost of the NPR transformation, no formal definition of quality and very little work that reports on the cost in memory of these algorithms.

My research also measures the execution time for my stroke-based painterly rendering system for performance, quality and memory costs.

For any real-time application to be successful it must succeed for the following set of criteria. My research attempts to accomplish all of these criteria to an acceptable level.

1. **Performance:** An application can only considered real-time unless it has some measure of the time it takes to perform its task and is bounded. For the purposes of the research contained in this document, real-time is defined as a graphics application whereby the entire update and rendering loop can complete and restart within the industry acceptable periods of time (which is discussed later). Update time identifies how a given algorithm fits into the time-budget to allow rendering to update at a periodic rate.

2. **Memory Footprint:** Most real-time graphics programs are run on hardware that has strictly bounded memory resources. As such, any successful real-time application needs to define a memory budget to identify how much of the available resources are being used and by what process. For many real-time graphics applications, the memory budget is dependent on the memory available on the hardware and the use of this memory is dependent on the requirements of the application.

3. **Usage:** Ease of use and implementation cost are critical for acceptance in a commercial project. Any application will have a resistance to its use if there is an exceptional amount of work required to implement or use the application. There needs to be some consideration of how much additional human intervention is required to make it work.

4. **Quality:** An application must meet the quality requirements desired will not likely meet the requirements of the application. There needs to be a measure to identify whether or not the application meets the desired quality threshold. For this research, there are two quality metrics that need to be considered: *static quality* defines how the application looks for a single not animating image and *dynamic quality* defines the quality associated with how well the sequence of images maintain coherency as the application updates.

In complex systems the criteria necessary for its success are frequently competing with each other and the criteria defined in this section are no different. For instance minimizing the memory use may have a necessary impact on quality or performance, simplifying the usage requirements may hinder the user's ability to meet desired quality requirements or meet a performance budget.

Changing the parameters of one aspect of a real-time rendering system may adversely affect other competing aspects. The following research identifies the major competing criteria and discusses the expected outcome for various scenarios.

The decision to prioritize which criteria are important lies with those that use the system (as with any complex system). This research attempts to define the criteria quantitatively so they could be used as a guide for forming decisions about the application's priority and the trade-offs associated with those decisions.

1.2 My Contributions

My approach to stroke-based painterly-rendering makes it possible to render painterly images at interactive rates, without additional burden on the user to define and refine parameters. It avoids the need to burden the application development team with the requirement to add additional information to the scene to support the algorithm by defining high-level parameters that are used to generate the paint-strokes.

The algorithm used in this research generates stroke information from the images output from an existing rendering method, uses the high-level parameters exposed to the user and renders the paint strokes. This algorithm behaves as a post-process to an existing rendering method.

In short, my contributions are:

- **Real-time stroke-based-painterly rendering classification:** I evaluate the previous work relating to painterly rendering methods with the focus on real-time updating of paint strokes. This evaluation categorizes the parameters used by the existing research in the field and many of the parameters are discussed for their feasibility for real-time painterly rendering.

- **Real-time, animated stroke-based-painterly rendering algorithm:** I combine the various parameters defined by the previous work of others to construct the individual paint strokes. I use a fast approximation of optical flow to carry paint strokes around the canvas in an attempt to maximize paint-stroke coherency between frames.

- **Painterly rendering measurement:** I define a set of tests to evaluate the decisions used to create a painterly rendering system. These tests measure the performance impact of various parameters, measure of quality and measure of cost in memory usage.

My research has shown that painterly rendering is plausible for real-time applications and my implementation executes faster than any other method published to date. In addition I have identified quantitatively the trade-offs between performance and quality for the reported painterly rendering algorithm and discuss these findings. I also classify existing research methods by the parameters used for their methods and discuss their usefulness in a real-time system.

1.3 Document Overview

The rest of the document is a description of the various algorithms that were experimented with to achieve the goals set out here. It then itemizes some interesting considerations from the research that was considered from the state of the art. Finally the document describes the results of this research including pictures, runtime analysis, relevant code sections and my conclusions on how this research compares to existing published work.

Chapter 1: This chapter, a brief overview of the problem that was tackled in this research, motivating usefulness and a summary of the work completed for this research. The 4 main criteria that needs to be considered for any real-time application: performance, memory, quality and usage are identified.

Chapter 2: The work published by others, describing how and why this research is important and how the research was carried out. Lays out the plan for achieving the 4 criteria described previously.

Chapter 3: The new approach given in this document, classifies previous work according to the parameters used.

Chapter 4: The experiments performed for this research. Describes the algorithms used, the data flow through the system and the various stages of processing that are necessary to calculate and used to create a painterly image at interactive rates.

Chapter 5: The results of the experiments carried out. Shows the data captured to identify the impact of the use of various parameters on the 4 main criteria described in chapter 1. Describes the method of measurements and motivation for which parameters were measured.

Chapter 6: Analysis of the data shown in chapter 5. Identifies the trends in the data, compares the effects of parameter usage and discusses the trade-offs of the use of the various parameter values.

Chapter 2

The Problem to be Solved

In recent years, photographic images have become more prevalent in our society, due in part to the invention of the camera which allows a greater percentage of the population to create and recreate market-ready images. As a result the majority of the images we see on a daily basis are now photographic as opposed to hand-drawn or hand-painted.

During the thousands of years of perfecting static paintings and sketchings the creators of these pictorial artifacts have learned and passed on to subsequent generations the skills and lessons learned about how to effectively communicate through these abstractions. These images can be pleasant to look at, but also the artist can exploit the human vision system to guide the attention of the viewer to look at specific areas of the image. The artist can direct the viewer to pay attention (orient) to some aspect of the image.

Despite their prevalence, camera generated images limit the artist's ability to direct the user's attention to specific areas of the scene. The properties of a photographic image are constrained to camera direction, focus, field-of-view, depth-of-field and the scene itself. There isn't a simple way for the photographer to artificially attract the user's gaze to a specific area of the image without modifying the image in a post-process due to the constant fidelity of the light as it is projected onto the photograph.

Taking a photograph is convenient and much faster than hand-drawing or painting an image but has the drawback that the resulting images are simply a copy of

the world at the point in time and space that it was taken. The artist has lost the ability to add their interpretation to the image. They are also less likely to be able to orient the viewer's low-level visual system to focus on the desired aspect of the image. Finally, since a photographic image is so similar to the world we see around us all the time, these images are not likely to engage the viewer to spend more than a few seconds viewing them.

The paper “Perceptually-based brush strokes for non-photo-realistic visualization” [2] describes how the human visual system identifies differences in luminance, hue, orientation, texture and motion to direct focus. Their claim is that “Human vision is designed to capitalize on the assumption that the world is generally a quiet place” and that our psycho-visual system orients on areas violating the *quiet space* assumption.

Painters take advantage of the visual-system's attraction to areas of change by defining objects with high levels of difference in the scene that are intended to be focused on. A claim made in [2] was that painters would have more than one object in the scene to look at which would engage the user, causing them to look back and forth between the small set of objects the artist intended to be looked at.

The result is that artists have been harnessing the power of our visual systems to orient our attention to a specific area of an image through the selective use of detail. They also direct us to engage in a subconscious analysis if the scene through a small set of objects and how they relate to the rest of the scene. In other words, they have a lot of control over how the viewer views the image.

For the remainder of this document I will use wikipedia's definition of “style” as: *the aspects of visual appearance of a work of art and how these aspects may relate it to other works*. These aspects refer to potentially many characteristics of the art world, but I will be focusing on the subset of the definition of style that relates to how the visual appearance of a work of art is defined by the parameters used to construct the individual paint strokes.

2.1 Non-Photo-Realistic Rendering

In recent years, computer rendering of images and geometry has allowed the artist to create a scene with greater control over specific areas of the produced images. This advancement potentially allows for painterly-like images to be plausible and therefore can be used to effectively communicate painterly images as well as the perspective of the artist. Computer rendering also improves the completion time of these images [4]. Instead of taking weeks to years to complete a painting, minutes to hours to take a photo, add artistic modifications and develop/print it into a photograph, a computer can generate images at a rate of milliseconds. Video games or visualizations exploit the speed of creating computer images to improve the rate the user can interact with the application.

The majority of new research for real-time interactive applications attempt to create a photo-realistic style of image. A phenomenon that is likely due to the familiarity we have with objects captured in a photo and how they coincide with what we see in our every day life. Photo-realistic images of solid objects also have a one-to-one correspondence between their position and colour after they are projected to the image plane. The relationship between position and colour in photo-realistic rendering aids in making the transformations from 3D geometry to the image plane a relatively simple operation.

The use of photo-realistic rendering is still used in the majority of the real-time rendering applications despite the finding in [2] that abstract images can be as effective or more effective than a photo to convey information to the user. They found that the use of a stroke-based data set to display multivalued weather data was the same or more effective than the traditional method of visualizing weather data in almost all cases.

The drawback of non-photo-realistic rendering (NPR) is that it is more ambiguous than photo-realistic rendering. Creating a cartoon, a sketch, a painting or some other form of artistic interpretation of the world is not as simple as “project the colour of an object from 3D to the 2D image plane”. The object being abstracted will likely change shape, change colour, be quantized to discrete strokes or lines or some other non-trivial transformation. As such, the work necessary to create an NPR

image is dependent on the set of transformations used to create the parameters for the abstraction and how those parameters are interpreted. As can be expected, there are many different transformations used for even more different algorithms to create these NPR images.

There is little research on real-time painterly-rendering, and only very recently in [6] was there an attempt to define an image-space painterly algorithm to execute at acceptable real-time speeds. Prior to this paper, there were attempts to render images in a painterly style by attaching paint-strokes, like particles, to the surface of the geometry in the scene. The result does produce something that resembles the effect of a painting, when the scene moves it becomes obvious that the paint-strokes are particles and everything looks furry.

The lack of real-time painterly rendering doesn't seem to have dissuaded video game developers and there have been several attempts to mimic the effects of painted images in a few video games. Games such as Ubisoft's XIII and Prince of Persia (2008), Sucker-Punch's Sly-Cooper series, Capcom's Okami and more recently Electronic Arts' Death Spank all made an obvious attempt to mimic the feel of the painted or drawn art styles. These are in addition to the thousands of video games that were required to generate a painterly style due to the limitations of the rendering hardware available.

For most of the video games that attempt to create a painterly-like rendering style, the effect is simply that, a style. Other than making the game consistently look like it was created by sketches or paint strokes, the style has no grounding in the medium it is mimicking.

Ubisoft's **XII** attempted to ground its rendering style in the comic book style. XII has a comic book rendering style for rendering the scene and character. In addition the game renders sequences of in-game scenarios in a set of smaller panes to further mimic the style and layout of a comic book.

Capcom/Clover's **Okami** further grounds its painterly-like rendering style by making a the entire game look like a Japanese style water colour painting that animates over time on an obvious paper background. The purpose of the style of Okami

is to give the illusion that the user is a painter watching their painting animate from the beginning of the adventure to its conclusion.

The game has additional grounding in the painterly-like rendering style by defining a game play mode that actually requires the user to manipulate a paint brush to draw shapes *on* the canvas that the scene is rendered on. The effect is to give the user the illusion that they are a painter that has the ability to influence the outcome of the adventure by painting specific modifications on the adventure. An additional effect tilts the camera slightly to show the artist's desk (with ink on it), the scene is rendered as a ink drawing/painting on a piece of paper with the adventure displayed on it and the paintbrush can be manipulated to draw on the scene. The result is to modify the adventure to cause a spell to be cast within the game.

There are many other video games that mimic a painterly rendering style in the game. The effect does allude to the user taking part in a painted story-book adventure, creation of the painterly looking effect is a time consuming process whereby artists must construct the entire world to look like it is painted. The artists are required to not only create the 3-dimensional geometry for the world, but also the textures applied to them need to be constructed to look like they were painted.

2.2 The Photorealistic-Nonphotorealistic Gap

In the world of video games and other interactive or real-time media, despite the desire for non-photo-realistic rendering, there is a very limited set of rendering styles currently exercised by their authors. Despite recent hardware advances, most video games and other real-time graphics applications generate images that would fall into the photo-realistic rendering or cartoon rendering categories. Prevalent use of photorealistic rendering limits the expressiveness of the media resulting in all of the art created with it being visually very similar.

Research has shown that non-photo-realistic rendering can be as effective or more effective to communicate to the user through images than photo-realistic rendering [2]. A fact likely not surprising to professional artists since they have been perfecting the painted medium for thousands of years as opposed to the tens of years for inter-

active computer generated art.

So the question remains, "why do most interactive real-time applications still strive for a photo-realistic rendering style as opposed to non-photo-realistic rendering?".

The answer may simply come down to the relative complexity of transforming raw data to the style of these two genres. 3D geometric modeling, texturing and animation all have a set of deterministic transformations from representation to display. In many cases, there are limited dependencies between the atomic objects being rendered or there are methods for resolving these dependencies within a limited degree of complexity.

Some aspects of NPR rendering don't have deterministic transformations and the complexity of resolving conflicts or ambiguities can be much greater than a photo-realistic rendered scene.

There can be order ambiguities for painterly rendering. For example, if a paint-stroke is rendered, it matters which order the paint-strokes are rendered in. Sorting is frequently a problem for performance in computer science, especially for a large set of data. Approaches such as [7, 8, 9, 10] define multiple scales of paint-stroke resolutions and simply render these multiple scales from lowest resolution to highest resolution in several passes. Other approaches avoid the depth aliasing problem by attaching the paint strokes to the surface of a 3-dimensional object and treat each paint-stroke as a particle [11, 12, 13]. An other approach to dealing with the occlusion ambiguity is to use the colour-gradient of the image to define paint-stroke boundaries and cut paint strokes short if they attempt to render across these boundaries [14].

There can be movement ambiguities for painterly rendering. If a paint-stroke is to move from one position on the screen to the next, it can have its characteristics changed dramatically. It may need to be oriented differently. The paint strokes may benefit from fading in and out or simply drawing over old strokes [15].

As paint strokes defined from image parameters move across the canvas, the paint strokes may bunch up resulting in a lot of paint strokes trying to draw over each

other, or they spread out leaving gaps in between them. Solving this coverage problem requires that a measure of paint-stroke density be calculated and paint-strokes be created or destroyed based on the density measure. Even recent research [14, 6] are subject to the paint-stroke coverage problem and although they may offer a potential solution, none describe the cost of their proposed solution.

2.3 Real-Time Stroke-Based Painterly Rendering for Video Games

Of the 4 criteria that must be upheld for a painterly rendering method to be useful for a video game or other real-time rendering algorithm, performance is paramount. Development cost and memory requirements will also hinder the feasibility of any rendering style if either of them become too cumbersome. If the proposed algorithm adds significant cost to rendering an existing scene, or requires significant artist time to implement it will get shelved as not-feasible for the application at hand. Finally a poor quality result makes the output unacceptable for the desired application.

The time cost of a rendering method must always be considered when rendering for interactive applications. The user must be able to react/interact to events in the images they are seeing is related to the rate at which they receive those images. An application becomes less interactive as the amount of time for rendering increases.

Ideally the rate at which the images should be updates should match the update rate of the device the images are being displayed on. Unfortunately, there seems to be no previous work that has achieved this lofty goal.

Although dependent on the hardware used, approaches such as [15] claim to have an update rate of 1-4fps (250-1000ms), [16] claim to have an update rate of 80,000-300,000ms, [17] claims less than 10,000 ms and [18] claim that the advection and filtering alone take 35,000 ms combined. All of these approaches greatly exceeds the ideal target of 4-8ms except [6] that claim a somewhat reasonable 30-50ms update rate.

2.3.1 Image-Space Real-Time Painterly Rendering (IsRaPtRS)

The algorithm used in this research exploits the high parallelism of commercially available rendering hardware on the market today in an attempt to keep the perceived cost to the viewers and authors of the transformation as low as possible. In relation to our goals, the algorithm transforms the images output from a typically rendered scene into a painterly rendering style. Using the output of an existing rendering pipeline should reduce the overall cost of authoring that the artists would have to do using other methods. Finally, I use algorithms from previous work described throughout this document to execute the transformation from T(i): PR $\rightarrow$ NPR while also using optimizations and hardware features to also keep the time cost of the transformation as low as possible.

There are several reasons for choosing to define the painterly rendering using image-space operations using the output of an existing rendering pipeline.

1. Appending the painterly rendering pipeline after an existing rendering pipeline makes use of existing functionality with little changes necessary to the existing functionality.
2. Should reduce the burden on the artists to create content to support the rendering method since the transformation works on images that are already being generated.
3. Since the bulk of the transformation occurs in image space, the algorithm can exploit the features of rendering-hardware from existing consumer video cards.

2.3.2 Performance: Hardware Acceleration

This work describes the design and implementation of a subset of stroke-based painterly-rendering algorithms (SBR) for execution within a soft real-time constraint. In order to allow real-time rendering of SBR painterly images, there must be an attempt to minimize the cost of processing a frame. The algorithm is designed to use the vertex,

fragment and geometry shaders of the graphics processing unit (GPU) to take advantage of its powerful and scalable processing unit for highly parallelizable operations.

The painterly algorithm is designed to make heavy use of the rendering resources we have available from the GPU to the fullest extent possible. The algorithm takes advantage of available rendering hardware and intentionally avoiding operations that have a high level of computational complexity or dependency.

The fragment shader is used to calculate the paint-stroke parameters necessary to render individual paint strokes such as: paint stroke colour, paint stroke center and direction, paint stroke movement characteristics and paint stroke surface properties. The vertex and geometry shader is used to take the paint-stroke parameters and generate the many paint strokes that comprises the final images.

All of the work described in this document is using reasonably powerful graphics hardware that is commercially available in the form of a NVidia 470 GTX graphics card and running on a 64-bit Windows Vista computer. Although not documented, this work was also executed on an ATI Radeon 5570 HD graphics card and running on a 64-bit Windows 7 computer.

2.3.3 Quality

In order to determine whether or not the output images are acceptable, or at least comparable, some measure of quality must be made to allow for comparison. Image-space painterly rendering a transformation from a source image $S \Rightarrow P(S)$ to a painterly image, it should be possible to evaluate the errors that are produced by the transformation.

It is arguable that painterly rendering is intended to create errors through careful aliasing of the source image into paint-strokes. The output image should resemble the input image, so measuring their similarity is not an entirely inappropriate metric to evaluate.

The objects in the painterly image are intended to animate and move. The re-

search described within this document focuses on image-space operations and as such is expected to be subject to aliasing errors as does any other image-space algorithms. The quality of the resulting image, or sequence of images can be evaluated with respect to the knowledge of the expected aliasing issues.

It should be stated that the cost of the transformation from T(i): PR $\rightarrow$ NPR will not be free; my method appends extra processing to existing rendering methods. Independent on how much of an improvement my method turns out to be over previous work, it will still be more costly performance-wise and memory-wise than not performing the PR $\Rightarrow$ NPR transform. It is not my intent to provide a solution that has no impact on a rendering system; but it is my intent to create an algorithm for creating painterly images that are reasonably temporally coherent, plausibly efficient for real-time rendering while not adding significantly more work for the people creating content for the application.

Chapter 3

Painterly rendering for real-time applications

3.1 Performance

The motivation of the painterly rendering algorithm described in this document is to produce a rendering algorithm that creates a painterly rendering style that runs fast enough for interactive applications. The intent is to have a rendering algorithm act as a post-processing effect to allow a painterly rendering style to be appended to an existing rendering system such as a video game or visualization system without significant modifications to the existing rendering system; to this end, the painterly rendering process must be exceptionally fast.

3.1.1 Performance Overview

A real-time application that cannot meeting its refresh deadlines is unusable for its intended purposes. Real-time applications such as video games must always be considering the cost of the time required to produce output. These applications are usually not considered hard-real time applications since nothing catastrophic will occur if a deadline is not met but missed deadlines degrade the usability of the application. In the case of video games, missed deadlines may hinder the ability of the user to be able to react to in-game situations.

Interactive applications need to define a budget for the time used to produce out-

put. Many applications don't have an explicit budget for its update time, but try to update fast enough to avoid annoying the user.

Video games require a predictably periodic update rate and are usually defined as 60 Hz, 30 Hz or even 20 Hz or15 Hz and can also be described as frames-per-second (FPS). These numbers are chosen to match some fraction of the refresh rate of an NTSC display device, such as a TV, which is assumed to be refreshing the screen at a rate of 60 Hz (60/2=30, 60/3=20, 60/4=15, etc).

It should be noted that the PAL signal standard has a refresh rate of 50Hz, that computer monitors can run at an almost arbitrary rate and that newer TVs can run at 120 Hz and even 240 Hz. For most video games in recent history, the desired update rate has been almost unanimously 60Hz with many video games falling back on 30 Hz when necessary. The rest of this document will be assuming these two update rates.

The periods of the 60 Hz and 30 Hz rates described previously are 16.7ms and 33.3ms respectively. These periods describe the amount of time that a single update must occur in without missing its time deadline.

3.1.2 Performance Requirements

A real-time painterly rendering process cannot be a hindrance to the entire rendering system necessary for the application. The calculation and rendering for the painterly rendering effect must not only share the meager 16.7ms (or 33.3ms) time budget but also cannot consume a large fraction of this budget.

The interactive application must have the sum of all its processes' time costs be less than the prescribed time budget. The "sum of all its processes" includes everything necessary for the application to perform correctly such as: Character rendering, World rendering, Particle rendering, UI rendering, Physics updates, Input polling, AI calculations, File I/O, Asset management, Network I/O, etc. The painterly rendering process is arguably only a very small part of the entire pipeline, and as such, the time budget for it must not be disproportionately large for its usefulness. If the cost of

rendering a painterly rendering effect is too high it will simply not be useful for the task it was intended for.

A reasonable time budget for a visual effect such as painterly rendering would be 25% of the budget for the endure update budget as a worst case (roughly 4.2ms (60Hz) or 8.3ms (30Hz)).

3.1.3 Performance Plan

There have been many painterly rendering algorithms created to simulate painting effects on the computer. Very little of the existing painterly rendering research consider their algorithm's usefulness for time-sensitive applications. Transforming images, video or 3D objects into a painterly image can be a non-linear operation depending on the algorithms used and/or the amount of data required. In order to mimic a painted image, paint stroke direction, width, length, curvature, texture and order all must be taken into account. Generating the information to define all these parameters can require a significant amount of calculations.

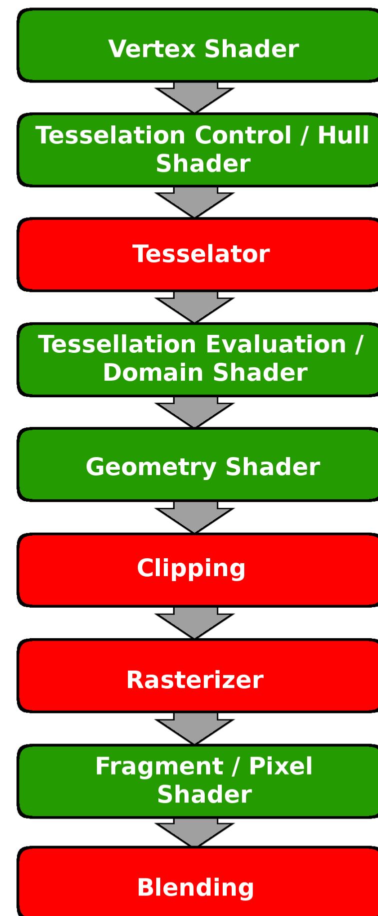


Figure 3.1: GPU Pipeline

The GPU is an essential tool to allow not only for the colour of pixels to be decided and presented to the frame buffer, but also to handle the load of a large amount of calculations that are parallelizable. For example the GeForce GTX 470 has 448 stream processors clocked at 625MHz. The calculations needed for painterly rendering could potentially run at orders of magnitude faster [19] with a highly parallelized algorithm implemented in a GPU shader program.

Recent graphics hardware provide GPUs that are highly programmable. In the last decade, consumer graphics hardware have been available that allow the user to define custom programmable "shader" programs. Shader programs allow for complex computation on the various stages of the rendering pipeline that had been previously hard-wired or hard-coded outside of the user's control. The stages of the rendering pipeline that are available for shader program definition are shown in Figure 3.1 as the green (dark) elements.

Other advancements in GPU programmability have allowed for general purpose GPU programming through APIs such as OpenCL and CUDA. These interfaces allow for more general purpose "kernel" programs to be defined and used on the GPU.

Lu, Sander and Finkelstein [6] have recently developed an algorithm that generates paint strokes in the GPU using OpenGL based on the vertex coordinates supplied to it, and generates a set of output vertices based on optical flow statistics to be used in subsequent frames. One major drawback to their solution is that an additional process must be introduced to handle the cases where moving paint strokes either "cluster unnecessarily or leave holes in the canvas" over time.

Similar to the method used in [6], my approach uses the geometry shader to generate the actual paint strokes on the GPU and my approach calculates optical-flow vectors to maximize frame coherency whenever possible. Unlike their method, I am consuming the image rendered after an existing rendering system as input to define the paint strokes to be produced as output every frame.

My method used the images created for the current frame and the previous frame to not only define optical flow, but to also define the paint strokes themselves. A consequence of the decision to use images to define paintstroke positions and movement results in avoiding the clustering and hole creation seen in other optical-flow-based methods by ensuring that at least 1 and at most 2 paint-strokes are considered for each pixel provided to the rendering system.

I use the GPU to calculate optical flow statistics to calculate how a pixel moves over time. For optical flow calculations, the movement information for pixels that are known are used to define optical flow information for pixels that don't have any

known optical movement for their immediate neighbours.

3.2 Painterly Rendering

Painterly rendering on a computer attempts to mimic the work done by painters; simulating the application of paint on a canvas. The process of painting with pigment, brush and canvas takes a very long time to complete, on the order of days to years. Painterly rendering performs a similar process, can produce a similar result but only takes milliseconds to seconds to complete.

There are many ways to create and arrange paint strokes on a virtual canvas using a computer. How the paint strokes are created defines the artistic style of the painting. Most of the research for computer-generated painterly rendering revolves around how to define the paint strokes, how to arrange the paint strokes on the virtual canvas or how to fix the problems caused by the previous generation or arrangement steps.

One of the primary focuses of my research is to create a painterly rendering system for real-time systems. Unfortunately much of the existing research on painterly rendering is of limited use due to the time-cost necessary to calculate the paint-stroke parameters or to render the paint-strokes themselves. Computationally expensive paint stroke parameters or rendering methods cannot be currently considered for a real-time painterly rendering system. Aspects of existing research that are too costly for real-time applications are discussed and are clearly defined how and why these algorithms needed to be discounted.

As a rule, any algorithm that is not O(1) or at least a highly parallelizable $O(\log_2 n)$, I seriously considered the feasibility of its use in my application. In most cases these algorithms could be considered too costly or unpredictable for use in a real-time application.

3.2.1 Paint Stroke Attributes

Painterly rendering algorithms use a set of parameters to define the paint strokes, and in some cases the set of parameters defines the artistic style of the image. For example pointillist paintings are defined by many small and point-like paint "strokes". Many of the existing painterly rendering algorithms focus on the definition and use of these parameters.

Paint-stroke position defines where on the canvas a specific paint stroke will be placed. For many painterly rendering systems, the paint-stroke-position defines the center of the paint stroke or some other position within the area of the paint stroke. Moment-based painterly rendering uses paint-stroke positions combined with paint-stroke orientations to define simple particle-like paint strokes that are used to construct the painterly image [6, 15, 16, 14, 7, 8, 20, 21, 22, 11, 12, 23, 24].

Paint-stroke orientation defines the direction of the paint stroke on the canvas. The direction parameter may represent the orientation of the whole stroke, or for defining the orientation of the paint stroke at control points for non-linear paint strokes. The orientation parameter represents the direction that the paint brush traveled on the canvas to create the paint stroke [6, 16, 14, 7, 8, 20, 21, 22, 25, 11, 23, 24].

Paint-stroke width and length defines the size and trivial shape of the paint strokes. For paint strokes that generate rectangles for the paint-stroke's representation, the width and height directly defines the width and the height of the rectangle [14, 7, 8, 20, 22, 11, 23, 24, 26, 2, 27, 16, 12, 13].

More complex paint stroke rendering systems may define a width and height parameter to represent the arc-length of the paint stroke and the width to define the maximum paint stroke width [17, 15, 28, 22, 29, 27, 25, 26].

Paint-stroke shape defines the geometric complexity of the paint strokes. The shape can be as simple as a rectangle, can be formed from a curve such a s a spline, can be tapered on the ends and may be shaped through the use of a texture [23, 29, 30, 26].

Paint-stroke texture defines how the colour of the paint stroke is to be repre-

sented [17, 6, 8, 21, 23, 29, 30]. Images can be used to define the actual colour of the paint stroke, the outline of the area the paint stroke influences (stencil) or even how the paint stroke interacts with the canvas, with other existing paint-strokes [30, 28] or the external lighting conditions affecting the painting [17, 16, 31].

Paint-stroke coverage defines how much of the canvas the paint strokes are intending to cover. Coverage may be defined transitively by the relative size of the paint-stroke size to the area of the canvas the paint stroke is being applied to, by the relative positions of the paint-strokes or by a combination of both [15, 16, 14].

Paint-stroke motion defines how a paintstroke will move over time [6, 15, 18, 14, 11]. This motion may influence position, size, coverage, shape implications.

3.2.2 State Of The Art

Most of the existing research uses a rendered image to inform the paint stroke parameters. Pixel information is used to define the colour of the paint strokes. Frequently a pre-pass filter is executed on the raw image to condition it for use in the subsequent operations on it. As an example, Gaussian convolution kernels are frequently used to blur the image and to reduce the influence of noise in the source image for subsequent operations.

Painters take advantage of the psychophysical properties of human perception to draw attention to areas of the picture they are painting. As such, most real-world paintings have a varying size and density of paint strokes on their surfaces. Multi-scale information, similar to mip-maps, are used to define a varying level of paint-stroke size and density [9, 7].

Paint strokes are usually oriented to align with object boundaries, or colour boundaries of the image being mimicked. Many of the painterly rendering methods use the colour or intensity (luminance) gradient for a specific image pixel in the image to define the orientation of the paint stroke. A simple method to calculate the gradient is to use some difference highlighting algorithm that calculates the greatest difference between the pixel in question and its neighbours. A vector orthogonal to the

gradient vector can be calculated by rotating the gradient vector by 90 degrees (in 2 dimensions). Many algorithms use a Sobel convolution kernel [32] to estimate the image gradient due to its use of convolution to make it a local operation that is highly parallelizable and relatively ease to implement and to execute on a GPU.

Chapter 4

Experiments

4.1 High Level Algorithm

A large part of any real-time painterly rendering algorithm's success revolves around quickly calculating the statistics needed to set the position, orientation and size of the brush strokes. Rendering the brush strokes has a good basis in the work done by the previous research described in this document but still needs significant optimization to make it appropriate for real-time applications. These algorithms analyze the source image, generate brush stroke geometry and submit all of them back to the graphics system to define the paint-stroke properties and render the painterly image.

The algorithm that is needed is one that produces a painterly result with as little dependency on such variables as possible. Transformation costs, buffering delays, overdraw costs, texture sampling and other pipeline-related bottlenecks are the main hindrances to performance of any such algorithm.

IsRaPtRS performs a statistical analysis of the input image's colour information and calculates a generalizing multiscale representation of this statisical data. These multiscale buffers store progressively generalizing colour and other statistical information in a pyramid structure of images similar to mip-maps. Each multiscale level contains the most relevant data that represents the image for that level of resolution and from the next lowest hierarchy level. Using the multiscale data a paintstroke can ask the hierarchy of data that it relates to information such as:

1. "How far is it to the closed pixel that is different in colour to pixel[i]?"

$$texture_dim_M(width, height) = texture_dim_0(\frac{width_0}{2^M}, \frac{height_0}{2^M}) \quad (4.1)$$

Equation 4.1: Multiscale texture dimensions relative to the 0th multiscale level

2. "How does pixel[i]'s colour relate to pixels that share the same colour as pixel[i] within its local neighbourhood?"
3. "What is an approximate center of mass of the colour region pixel[i] is in?"
4. "What is a rough approximation of the gradient for pixel[i] colour?"
5. "How has pixel[i] moved since the last frame for various resolutions of view?"

These questions can be used to solve problems about the stroke position, orientation, colour, texture and movement. The painterly algorithm will make use of multiscale buffers to answer these questions and to generate the brush strokes in a similar method to the approach described by Nehab & Velho [9]. Performance-wise, using these multiscale buffers these questions can be answered in O(log(n)) time for each paintstroke at a cost increase of 4/3 to create the additional buffers.

In defining the algorithm for the image-space real-time painterly-rendering system (IsRaPtRS) I use the following naming convention:

The variable f is the frame index the data belongs to. As the animation progresses, each refresh of the source image increases its f value. In order to minimize the quantity of data captured, the value f will only use the current value of f and the value f-1 to compare the currently rendered image and the previously rendered image.

The variable M is used to define the multiscale (or mipmap) level that is being used and relates to the buffer dimensions as shown in Equation 4.1.

Some post processing effects that benefit from similar multiscale rendering are: deferred lighting/shading [33], bloom [34, 35], depth of field [36, 37], volumetric post-processing effects [38, 39], image warping such as refraction or mirage effects [40] and potentially many more.

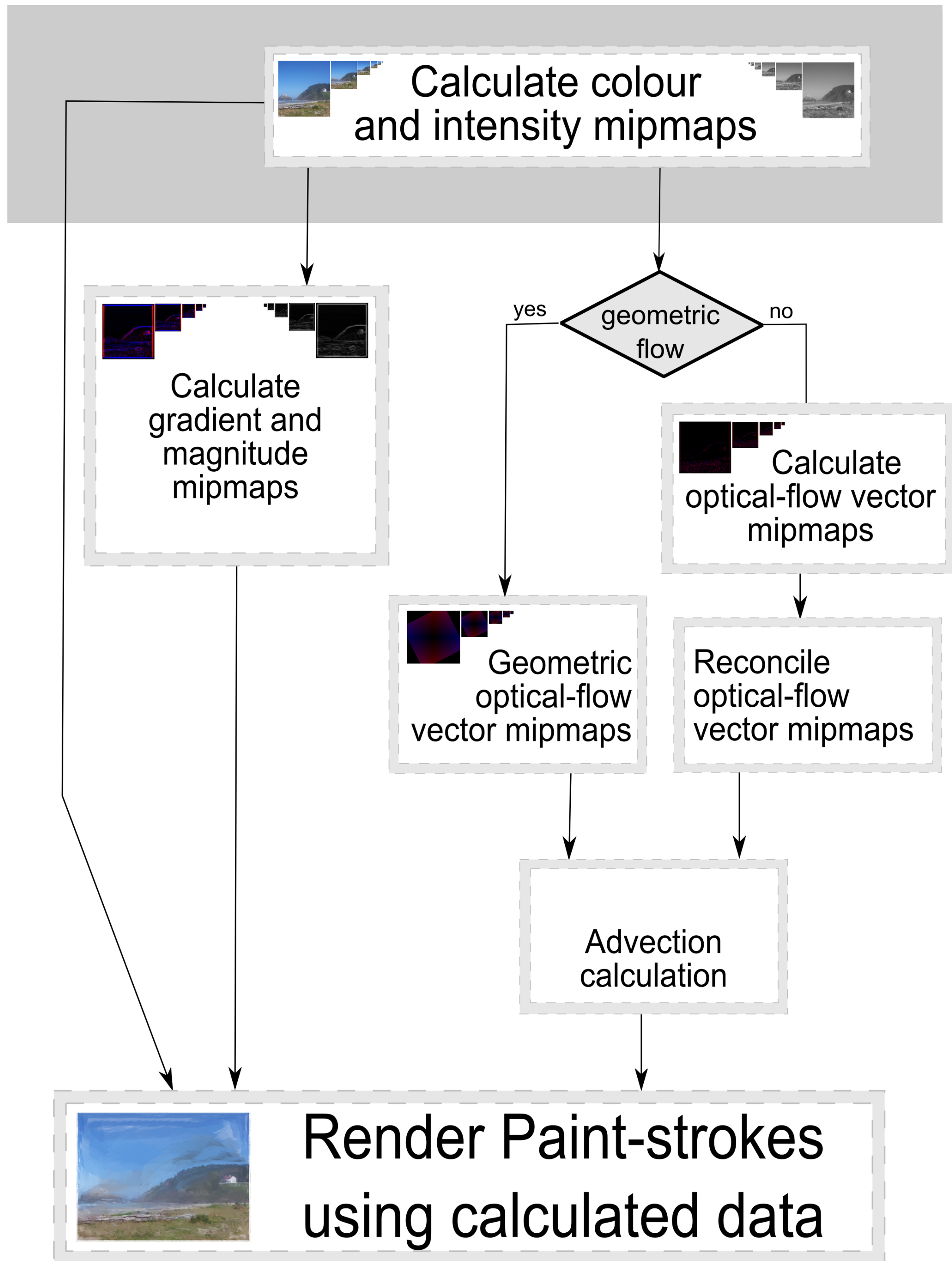


Figure 4.1: Diagram of painterly rendering data calculation

4.1.1 Algorithm Discussion

For each painterly rendered frame, a set of paint stroke parameters needs to be calculated to define the paint strokes. These parameters define the position, orientation, size, shape and material properties for each of the potential paint strokes that are to be rendered.

First, the scene needs to be rendered into a texture that will likely be the output of an existing rendering pipeline. The texture produced will define all of the remaining characteristics of the painterly rendering system.

The multiscale colour and multiscale intensity buffers are derived entirely from the colour buffer or from the next highest multiscale layer.

The multiscale orientation and multiscale detail buffers are derived from a gradient calculation on the multiscale intensity buffers.

The multiscale optical flow buffers are defined by comparing the colour differences between adjacent pixels for a defined pixel vector offset with a similar pixel vector offset from the previously rendered image that has been offset in one of 8 directions (as shown in Figure 4.3 on page 43).

4.2 Multiscale Colour and Intensity Buffers

Multiscale buffers are calculated in order to ensure that certain calculations can occur in $O(log_2(n))$ by using the data buffers that progressively generalize higher-detail buffers. Using multiscale buffers is not uncommon in real-time computer graphics. Multiscale image data called 'mip-maps' are frequently used for real-time rendering. Mip-maps take a high-resolution image and divides each dimension of the image by powers of two (eg. $M_0 : 1024 \times 1024, M_1 : 512 \times 512, M_2 : 256 \times 256, M_3 : 128 \times 128 ... M_N : 1 \times 1$).

The motivation for mip-map creation is that it contains a representation of the image that is smaller in size and still appropriately representative of the source image

$$C_M(w,h) = 1/4 \sum_{i=w-1}^{i<w+1} \sum_{j=h-1}^{j<h+1} C_{M-1}(i+1/2, j+1/2) \tag{4.2}$$

Equation 4.2: Multiscale Colour Buffers

for the situation using it. For instance, looking in 3D at a wall orthogonal to the view direction will require many different resolutions of the image applied to it when the camera is at different distanced from it. Pixels on the wall close to the camera require a high-resolution representation of the image, whereas the need for high-resolution images drops off very quickly as the pixels on the wall get further away from the camera. The cost of sampling a high-resolution representation of the wall texture for pixels that are rendered far from the camera (and therefore don't need the higher resolution) costs more memory than is needed and therefore increases the likeliness of a data cache miss when sampling that image. Ideally an object that is rendered would have an image representation that matches the resolution of the rendered object as closely as possible.

Creating a full mip-map set increases the total data size of the image by 4/3 as is shown in the calculation in Equation A.7 but can be used increase the likelihood of avoiding a data cache miss for a greater number of texels.

For each texel in a multiscale image's representation (greater than 0), its value can usually be created from some combination of the 4 pixels surrounding it from the next-highest multiscale resolution. The calculation of a given pixel in a multiscale image can be computed from a set of 4 coherent pixels from the next higher multiscale image.

4.2.1 Multiscale Colour

The multiscale colour buffers for multiscale levels of m=1..n are calculated from the 4 closest pixels from the multiscale image m-1. The IsRaPtRS algorithm calculates a multiscale box filter which is simply defined as an average of the 4 pixels of interest from the next-highest multiscale image as shown in Equation 4.2.

$$I_M(i,j) = C_M(i,j)[red, green, blue] * [0.2126, 0.7152, 0.0722] \tag{4.3}$$

Equation 4.3: Intensity Calculation

4.2.2 Multiscale Intensity

Multiscale intensity buffer contains the luminance of the results of the colour buffer for the same multiscale image level as described in Equation 4.3.

4.2.3 Performance Considerations

For the calculations of the multiscale colour and multiscale intensity buffers, the shader packs the <red, green, blue> values from the colour calculations into the <red, green, blue> values of the output texture respectively and packs the intensity value into the alpha component of the output texture.

The texture that is being written to (as the output texture) is defined as a RGBA8 texture.

The use of all 4 components of the texture being written to is to minimize the number of separate images that need to be set up for the render target.

4.3 Multiscale Orientation and Detail Buffers

The multiscale orientation buffers are the set of vectors orthogonal to the gradient of the image. These orientation buffers are used to orient the paint strokes to the isolines (contour lines) of the image. The orientation buffers ensure that the major axis of the paint strokes stay along the pixels that are most likely to share colour characteristics with the paint stroke being rendered.

The multiscale detail buffers are defined as the magnitude of the gradient for a given pixel and is used to determine the size of the paint strokes. The paint stroke sizes are defined by at least 2 criteria, which multiscale level that is appropriate for the resolution of the paint stroke and a measure of blend between the multiscale levels.

4.3.1 Gradient Calculation

The orientation and detail buffers are generated from a gradient calculation. The Sobel convolution operator is used to approximate the colour gradient of the image [32]. The result of the Sobel convolution calculation defines the vector direction that corresponds with the largest difference in colour around the given pixel. The magnitude of the resulting vector is also a measure of the colour differences.

$$grad(x, y) = \nabla(f(x, y)) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) \tag{4.4}$$

Equation 4.4: Gradient formula

where $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ can be approximated with the equations Dx and Dy respectively show below.

$$Dx = 1/8 \begin{bmatrix} -1.0 & 0.0 & +1.0 \\ -2.0 & 0.0 & +2.0 \\ -1.0 & 0.0 & +1.0 \end{bmatrix}, Dy = 1/8 \begin{bmatrix} +1.0 & +2.0 & +1.0 \\ 0.0 & 0.0 & 0.0 \\ -1.0 & -2.0 & -1.0 \end{bmatrix} \tag{4.5}$$

Equation 4.5: Sobel convolution kernel

4.3.2 Mutiscale Orientation

The multiscale orientation buffer is a 90 degree rotation of the gradient vector. A 2 dimensional rotation of 90 degrees is simply calculated as:

$$rotate_{90^\circ}\Big(f(x, y)\Big) =< y, -x >.$$

4.3.3 Mutiscale Detail

The multiscale detail buffers define which base resolution of the multiscale buffers that a given paint stroke will be rendered at. The detail buffers define a stencil from which the paint-stroke rendering code will decide whether or not to draw a paint stroke or not for a given multiscale level.

The decision of whether or not to render a paint stroke is calculated with a threshold using the step function (Equ. A.1, page 72). For any gradient value that has a length smaller than a threshold, the paint stroke is discarded, otherwise the paint stroke is rendered using the buffer data calculated previously in the process. As is shown in Figure 4.7 (page 45), paint strokes that have very little gradient contribution simply don't render a paint stroke quadrangle. The set of multiscale paint strokes show the large/broad paint strokes for areas with little detail and detailed paint strokes are rendered progressively over top where the image has a high variance in its colours.

4.4 Orientation and Detail Performance Considerations

For the calculations of the multiscale orientation and multiscale detail buffers, the shader packs the <Dx, Dy> into the <red, green> values of the output texture respectively and the shader packs the detail value into the blue component of the output texture.

Packing the orientation data and magnitude data in the same texture makes use of as many of the 4 components of the texture being written to and to minimize the number of separate images that need to be set up for the render target.

Currently the output texture for the orientation and detail buffers is defined as a RGBA_Float16, but since the Dx and Dy values could be normalized to the range [-1..1] and the detail values are in the range 0..1 and none of these need excessive precision, they could be stored in a RGBA8 texture. Of course the Dx and Dy values would need to be transformed to the range [0..1] via: $packvalue = (rawvalue + 1)/2$ and would then need to be unpacked when used with a: $unpackedvalue = packedvalue * 2 - 1$.

4.5 Mutiscale Performance Considerations

For all multiscale colour, intensity, orientation and detail, the operations required to calculate these buffers are relatively simple convolution kernels and so their calculations scale linearly with the number of pixels being rendered.

There is quite a lot of sampling occurring in order to fetch the pixel values that are used within the window that the convolution kernel is operating on. For the multiscale colour (mip-map) calculation, there are 4n pixels being sampled from the higher resolution image [m-1] to calculate the box filtered colour and intensity values. There are 4n pixels being sampled due to the fact that 4 pixels from the higher resolution image [m-1] shares the same area for each pixel in the lower resolution image [m]. For the multiscale gradient calculation, there are 9n pixels being sampled from the same resolution image [m] to calculate the orientation and detail buffers. 9n pixels refer to the pixel sharing the same coordinate as well as the 8 surrounding pixels for that coordinate, sometimes referred to as the 8-window for that pixel.

Fortunately since these calculations are being done on the GPU, they can all be done in batches and can possibly also avoid data cache misses due to adjacent pixels performing almost exactly the same data accessing.

4.6 Frame Coherence

Static painterly images can be made beautiful and there are many algorithms that can make painterly images that compare to their real-world counterparts, paintings. A logical step is to attempt to make these painterly images move in a similar way that static photographs seem to move in a video sequence. The most obvious problem with making a painted image dynamic is that paint strokes cannot be considered as infinitely small particles of colour as their photograph counterparts are frequently treated. The paint strokes have individual size, shape, texture and occlusion properties that significantly contribute to their own identity.

Moving these paint strokes from one image frame to the next requires that the properties that define an individual paint stroke stay coherent between frames and

stay consistent over the lifespan of the paint-stroke. Temporal coherence is not a trivial task since individual paint strokes' parameters can have complex side effects to other paint strokes when they are used to render the paint stroke. For example, simply moving a paint stroke, changing its size or changing its orientation will likely have an effect on the occlusion properties of previously adjacent paint strokes. The new parameters may cause adjacent paint strokes to become completely occluded, partially occluded or cause gaps between paint strokes. Additional processing is required to fix undesirable properties of these side-effects.

4.6.1 Animating Paint Strokes

Many painterly rendering methods attempt to maintain temporal coherence by operating on a sequence of images such as video or subsequently rendered frames. One way to achieve coherent movement of paint strokes is by calculating optical flow vector fields and advecting the paint strokes from one frame to the next. As was noted, advection of paint strokes may cause paint strokes to pile up or to cause gaps in between them. Both of these situations must be fixed by eliminating redundant paint strokes or adding new paint strokes, respectively.

Adding new paint strokes or removing existing paint strokes requires that the topology of the paint-stroke adjacencies be analyzed and corrected to fit a desired coverage property. Ensuring a desired coverage criteria can be expensive and iterative.

Some approaches proposed to solve this problem revolve around generating a triangle mesh of adjacent particles and calculating the area of each triangle to determine if it is too small or too large to represent a single particle [41]. The triangle mesh method requires that the adjacency mesh be created and updated each iteration, a costly operation.

Some research uses 3D geometry to define the position, orientation and colour of the paint strokes. The creation of paint strokes from 3D geometry has the benefit of keeping paint-strokes naturally coherent when the scene is animated [11, 12, 13]. Since the paint strokes are defined as particles attached to 3D geometry, as the geom-

etry is transformed so too will the positions of the paint strokes. The paint strokes move as smoothly and naturally as does the surface of the object. Unfortunately the drawback of these methods are that the positions of these particles must be defined in addition to the 3D geometry. While the particle position calculation can be done off-line with a preprocessing step, if the artist responsible for the look of the scene wants to change how the paint strokes are represented, it could be a cumbersome task to do so if there is a lot of authoring or iteration required. An additional drawback to these methods is that there can potentially be dramatically more geometry to transform and render, leading to an increased probability of excessive transformation costs and overdraw issues adding to render time costs.

Another approach simply paints over old images with new paint strokes where the image is changing [15]. An image based approach does guarantee coverage of the canvas and improves coherency, but it suffer from aliasing problems due to the discrete nature of pixel data. The aliasing problems of an image-based painterly rendering approach can result in paintstrokes popping into and out of existence due to this aliasing.

The method used is to calculate a trivial optical flow vector field in image-space and advect the paint strokes also in image-space . The decision to follow the image-based approach was due to the availability of the highly-parallelizable GPU to perform the optical flow calculation. Also an image-based approach favours a measure of paint-stroke coverage as opposed to consistency between frames.

4.6.2 Optical Flow

I implemented two distinct optical flow calculations, one using the source image data only and one using the transformation information of the geometry. Both methods have their benefits and drawbacks and will be discussed in this section.

Image-Space Optical Flow

The first optical flow method implemented calculates colour difference (CD) between a pixel and its 8-neighbours (pixel offset vector (PO)) and then compares that difference vector to the same pixel offset vector starting at a different pixel for the

previously rendered frame. A similarity measure is calculated between the CD[f] and its counterpart for CD[f-1] as shown in Figure 4.3. If the similarity exceeds a defined threshold, then the optical flow vector PO is defined as the representative for the optical flow vector.

For images that are consistent, colour-wise, the optical flow vectors will only match the threshold criteria for areas with large discontinuities. Those pixels that cannot calculate optical flow, due to lack of information within the convolution window, require optical flow information from outside their convolution window. Fortunately the information calculated for the highest multi-scale level can be progressively passed on to lower multi-scale levels to be used as a reasonable approximation of the optical flow for pixels that cannot calculate it for itself using local data.

Once the generalization of optical flow has been calculated for all the multi-scale levels, the high-frequency multiscale buffers still contain areas where there should be optical flow but no optical flow information was calculated. Ideally a process should be able to sample an optical flow buffer for any multiscale level to retrieve its estimated optical flow value is for a given pixel coordinate. The optical flow buffers have not calculated the optical flow estimation for areas with small or no local optical flow information calculated.

Optical Flow Reconciled

The optical flow values need to be reconciled for those pixels that could not calculate an optical flow vector previously. The optical flow vectors calculated previously need to be propagated back into the optical flow buffers from the lowest resolution multi-scale level up to the highest resolution multi-scale level used.

Reconciling can be estimated by sampling the optical flow from the lower resolution multiscale buffer [M+1] and combining it with the higher resolution multiscale buffer with a simple average or weighted average. For higher resolution buffers with no optical flow estimate, the lower resolution multiscale optical flow estimate can be used as the representation of the optical flow value if the colour values for both buffers match within a reasonable error tolerance.

4.6.3 Image-Space Optical Flow Quality and Usage Considerations

Calculating image-space optical flow vectors and even reconciling them does not result in an accurate representation of the scene's actual pixel motion. The problem is that the motion vector is calculated from matching colour differences between pixels and over time. While comparing colour information between adjacent pixels can frequently estimate the motion of the pixel, it doesn't necessarily predict correct pixel motion information.

The problem stems from the fact that colour information does not correspond to motion. For instance if the image has a large colour difference vector that is orthogonal to the actual motion of the pixel, the estimated pixel motion will be calculated as orthogonal to the actual pixel motion. The result is a false-positive for a motion signal.

Similarly a pixel that has actual motion will not necessarily result in an estimated motion vector. For instance if a pixel is actually moving orthogonal to the colour difference vector and the 8-neighbourhood of pixel information does not change, then a small or 0-length optical flow vector will be calculated.

Image-space optical flow calculations are not representative to actual object motion, at least for small convolution windows. For areas of the source image that lacks enough colour deviations or has colour deviations that result in an orthogonal vector to the actual motion of the pixel, the resulting optical flow vectors can be incorrect.

These negative side-effects can be minimized by ensuring that the source image has a sufficient level of variance consistently over the image to statistically result in a larger percentage of correct optical flow signals.

A sufficient level of colour variance fixes the lack-of information problem, but not the false-signal problem. The solution to false-signals is to increase the number of samples to ensure that outliers are made insignificant to the final result and means that the 8-neighbour method of calculating optical flow is insufficient. There needs

to be additional signals considered to confirm or deny any optical flow calculation.

Much of the literature on optical-flow expands the convolution window or employs iterative methods to perform the statistical confirmation operation. Unfortunately for a real-time application, neither an increased convolution window or iteration are appropriate directly [42].

The method used in IsRaPtRS the mip map buffers to calculate the confirmation step. The mip map's pixels contain a windowed average of the optical flow vectors for those pixels contained within the extents of it. Reconciling information from multiscale buffers is a single iteration operation to allow for holes in the forward optical flow calculation to be filled where possible.

Finally, the reconciled optical flow method is again just an approximation of the actual optical flow of the image. There are trivial cases where no amount of image analysis will result in the correct optical flow. For example stripes moving along the stripe direction or concentric circles rotating around the center of the circles provide no information about the actual object motion from the output images.

For sufficiently variant images, optical flow calculations do provide the key benefit that they require no additional information about the image to estimate how objects are moving. All that is needed is the set of source images to estimate the optical flow from.

Geometric Image-Space Motion

The second method of calculating image-space motion vectors was done by saving the actual transformations for the objects being rendered in the scene to reconstruct the transformation differences of the object from previous renderings of the output image to the current rendering as done for motion blur effects in [43]. For the purposes of this document this method of image-space motion estimation is called geometric image-space motion.

The geometric image-space motion method has the benefit of calculating the object

motion in the scene with exact precision. The screen-space positions of the fragments that will become output pixels can be calculated exactly and vectors between the previous rendering and the current one. The vertices simply need to be transformed for the previously rendered image and compared against the transformed vertices for the currently rendered image. The difference between the two transformed objects is interpolated across the object and the result is an exact representation of the optical flow motion in image-space.

Geometric image-space motion only requires the storage of the transformation data between rendered frames. For scenes with a small number of objects that are moving the result is a potentially substantial savings in memory usage since subsequent frames do not need to be collected for the optical flow calculation just the transformation data.

For scenes with a lot of objects being transformed there is potentially similar costs to the image space optical flow method in memory use and likely an increase in performance cost to transfer all of the transformation data between the rendering processes.

Geometric image-space motion also requires that all moving objects in the scene keep track of not only the transformation for the currently rendered image, but also previous transformations. The implication being that the rendering system for an existing application would need to potentially change dramatically to buffer all of the transformation information and then be used to calculate the motion data. Calculating optical flow in this way will likely not be able to be trivially appended to an existing rendering system.

4.7 Paint-Stroke Rendering

Finally after calculating multi scale statistics for colour, intensity, orientation, detail, (gradient) and optical-flow, it is now possible to transform the original scene into a stroke-based painterly-rendered scene. Multi scale generalization is used as a set of best-fit parameters for each scale similar to the method used in [31].

4.7.1 Paint-Stroke Detail

For each pixel on each multi scale level, a detail value is calculated by thresholding the gradient buffer to decide whether or not to generate a paint stroke or not. The geometry shader makes the decision to generate a paint-stroke quadrangle from the value stored in the detail buffer.

4.7.2 Paint-Stroke Orientation

Paint-stroke quadrangle orientation is defined by the value stored in the orientation buffer. The quadrangle for the paint_stroke (PS) is defined by the points: $advection_position(PS) \pm orientation(PS) * length(PS) \pm Rotate_{90^\circ}(orientation(PS)) * width(PS)$. These points will define the 4 coordinates necessary to construct the quadrangle oriented orthogonal to the colour gradient of the image and along the colour-isoline of the image.

The geometry shader generates each quadrangle using the orientation buffer to define the orientation of the paint stroke quadrangle..

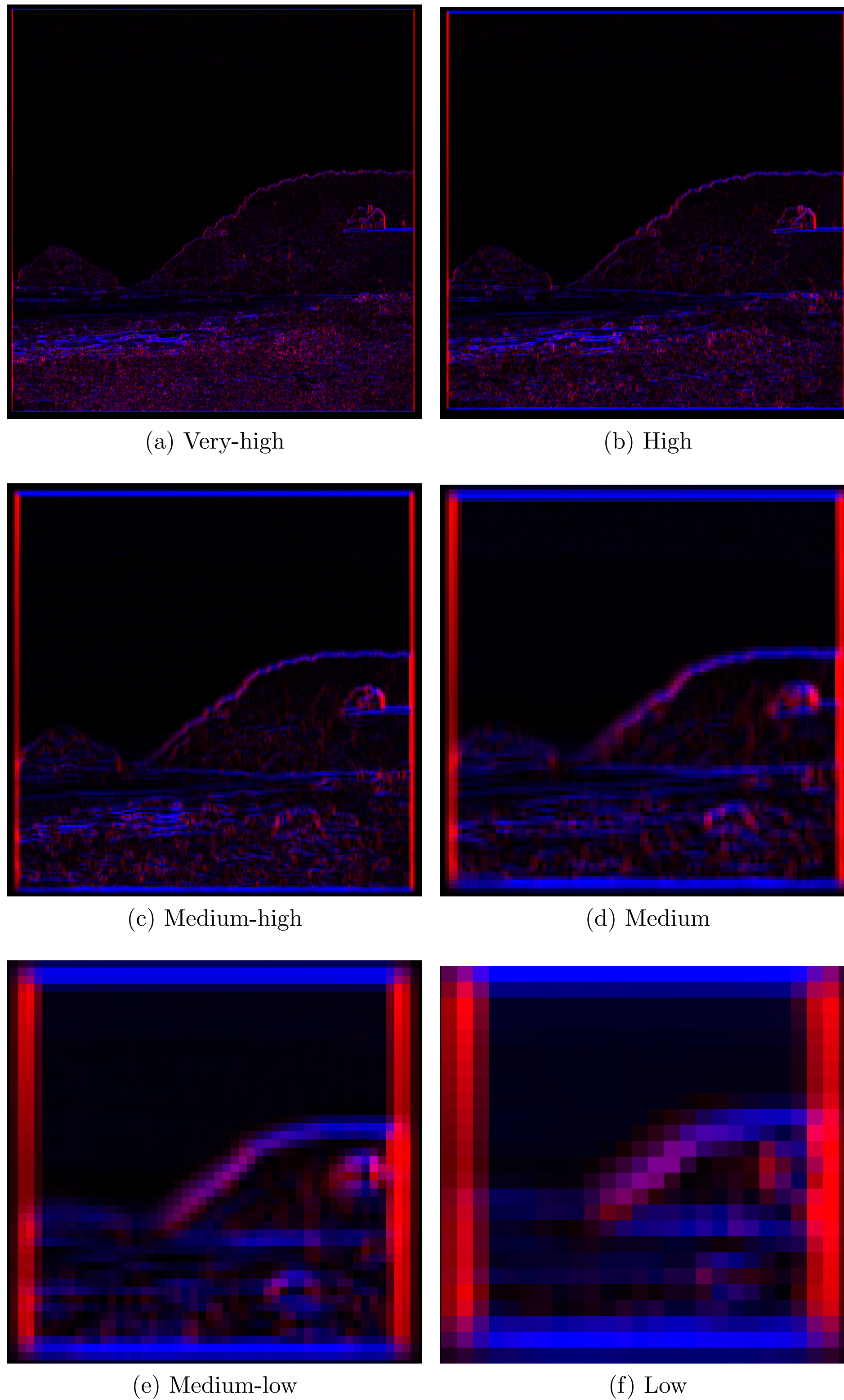
4.7.3 Paint-Stroke Colour, Shape and Material

The colour data drives the paint-stroke colour and will be passed through the paint-stroke rendering code to provide the vertex colours of the paint stroke-quadrangles. The paint-stroke quadrangles will also have a grey scale texture applied to them to define the shape and semi-transparency of each individual paint stroke as shown in

Figure 4.4.

4.7.4 Paint-Stroke Position

The paint strokes are positioned relative to the center of the pixel that the paint stroke is associated with. The position is offset by the advection vector calculated previously.



(a) Very-high

(b) High

(c) Medium-high

(d) Medium

(e) Medium-low

(f) Low

Figure 4.2: Multiscale Gradient Calculation, <x, y> displayed as <red, blue> respectively.

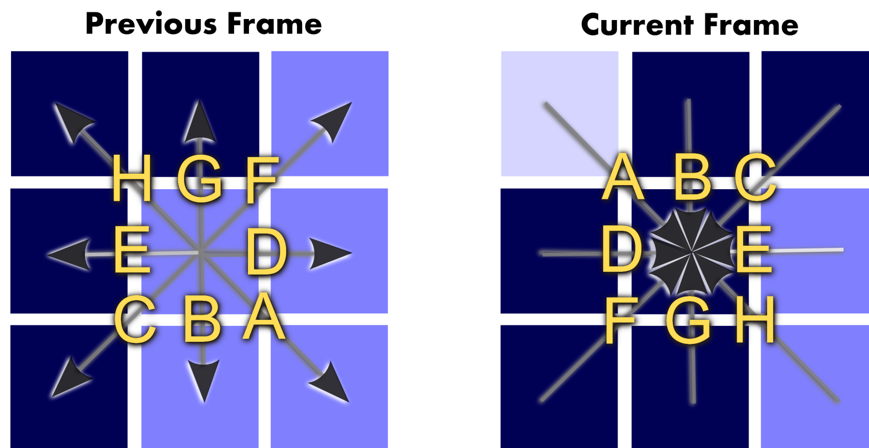


Figure 4.3: Local window optical flow example

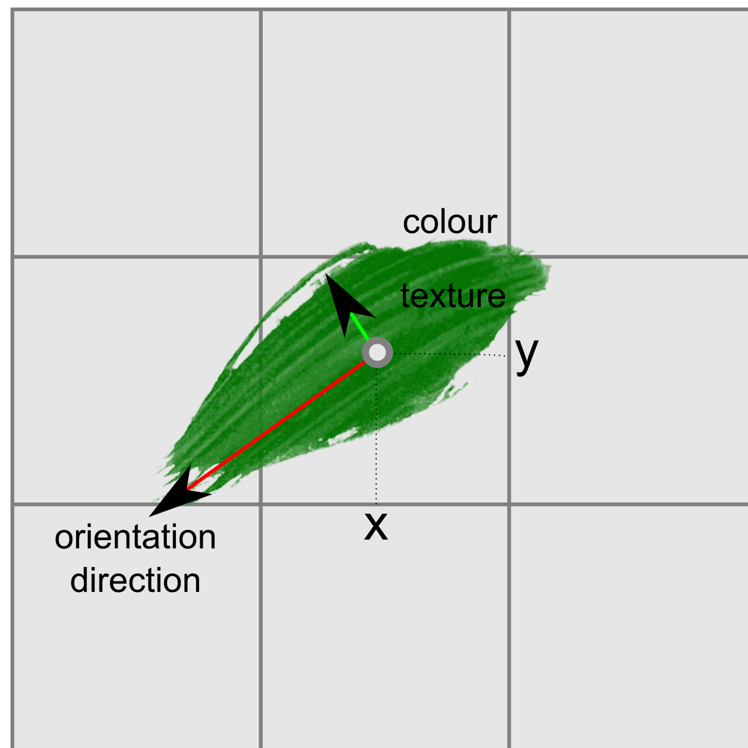
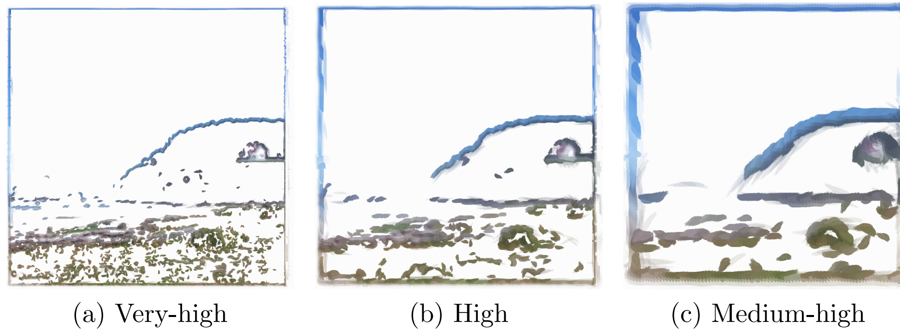


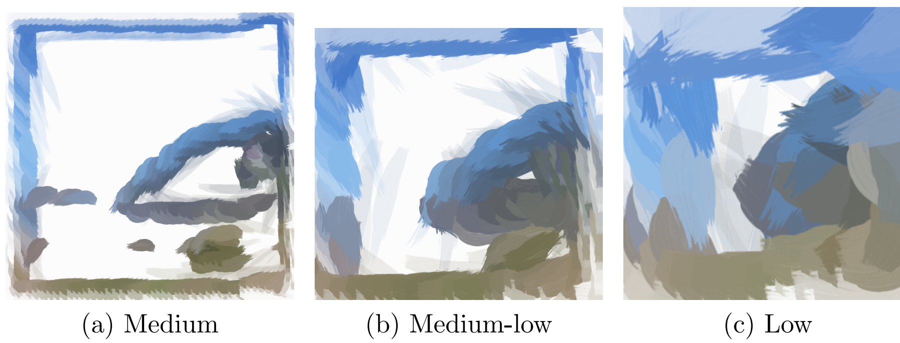
Figure 4.4: Paint stroke parameters

Figure 4.5: Vacation picture of the Oregon coast

Figure 4.6: Oregon coast that uses multi scale paint strokes



(a) Very-high (b) High (c) Medium-high



(a) Medium (b) Medium-low (c) Low

Figure 4.7: Multi scale paint strokes

Chapter 5

Evaluation, Analysis and Comparisons

Developing a real-time painterly rendering system is a complex task with many variables to be considered and evaluated. Like most complex systems, changing one variable can have dramatic consequences on others. This chapter is an attempt to quantify the effects of three of the primary output variables that were defined in section 1.1 as critical to painterly rendering: performance, frame coherency and quality.

For all of these experiments, measurements are taken from IsRaPtRS as unmodified as possible. The measurements are collected over at least 500 frames and the arithmetic mean is used to define the average as a metric for measurement. It is this average that is the measurement shown in this section.

5.1 Performance Analysis

The time cost of an algorithm is paramount for any real-time system. Without understanding what contributes to the consumption of time, a real-time system will likely miss deadlines having dire consequences on the performance of the system. This chapter shows the measurements of IsRaPtRS, summarizes and explains these results.

From a performance perspective, there are obvious contributors to the time-cost for a real-time rendering system: raster operation costs, primitive operation costs,

data access costs, state manipulation costs and any stalls in the rendering pipeline caused by a delay on the CPU.

Raster operation costs include the costs required to render the pixels of the output image. For the purposes of my analysis, raster operation costs include image sampling costs and fragment processing costs. In more practical terms, raster operation costs are mainly the costs associated with anything that is processed from within the fragment shader or anything that is an input to the fragment shader.

5.1.1 Calculating Paint Stroke Data Buffers

One way to measure raster operation costs is to modify how many pixels need to be rendered and calculate the relative cost per pixel. There is a relationship between the density of the rendered output image and the density of the texture's pixels (texels), and the texel/pixel density relationship may relate to a performance cost for rendering paint strokes. This experiment measures how much of an effect that the texel density of the source image has on the algorithm overall.

By modifying the resolution of the textures applied to the geometry in the scene to affect the texel/pixel relationship it is possible to measure whether the source texel density has a major impact on the resulting painterly rendered scene. This experiment was carried out and the results of this experiment are shown in Figure 5.1.

From the results of these measurements it can be concluded that the relationship between pixel and texel densities have only minor impact on the performance of painterly rendered images except when the pixel/texel relationship becomes obviously disjoint. In cases where the painterly image resolution is 1024x1024 but the texture image size is 128x128, there is a slight but noticeable increase in time required to render the painterly image.

An interesting point to be made from the measurements shown in Figure 5.1 is that the majority of the variance of the painterly rendering pipeline is in the stage that actually renders paint strokes. Wildly varying values in the RenderPaintStrokes().Average values in Figure 5.1 shows the mojority of the variance is

in the paint stroke rendering. The other stages are reasonably consistent despite the differing source images applied. These stages of the pipeline treat each pixel as a single input and produce a single output with the same shader program, so it isn't surprising that there is very little variance in their cost.

5.1.2 Cost of Rendering Paint Strokes

It was shown in the experiment described in section 5.1.1 that the majority of the variance in the time required to render a painterly image was in the stage that actually rendered the paint strokes. With that in mind there are two factors that could influence that cost, the raw number of pixels required to be rendered per frame (fill rate) and the amount of times each pixel is required to be redrawn (overdraw).

To measure the raw cost of raster operations the scene was rendered for the set of source images and varied the number of fragments rendered by changing the size of the render targets but keeping the other parameters constant between tests.

In the following test the integer values are scaled to be 10 times to allow for integer scalars to be used on the user panel meaning that a value of 10 produces a quadrangle with a length that matches the pixel's dimensions.

To measure the cost of multiply rendered pixels, the scene is rendered using the same set of source images but varied the relative size of each paint stroke. The test starts from a paint stroke size scalar value of 3, which does not cover the background, to a value of 27 which provides a significant amount of paint-stroke overdraw.

These *PaintStrokeSize* scalar values range from 3 to 30 and scale the length of the major axis of the paint stroke by the interpolated width/height value which is by default 0.5. (The major axis of the paint strokes are normalized to the size of the pixel for that multi resolution level). Relating the coverage of the paint stroke texture to the size of the geometric quad being rendered, a minimum value of 16 is required to get the majority of the canvas covered by paint strokes.

Figure 5.2 shows the output of the results of the varied paint-stroke size test with

respect to the cost of time to render the scene. As the paint stroke sizes increase, so does the cost in time to render them with an increasing cost at a rate of M^2 where M is the multiscale level. An M^2 growth in cost compared to a linear growth in paintstroke dimensions makes sense since the area of the paint strokes will increase as both axes of the paint stroke grows linearly. (Area = width * height or Area = aspect_ratio * height^2) .

Otherwise the comparative cost of the rest of the painterly rendering pipeline remains very close to constant overall despite the dramatically differing source images used in the pipeline. The fact that the combined cost in time of all of the previous rendering stages to the paint-stroke rendering stage remain quite constant between tests is shown in Figure 5.2.

5.2 Qualitative Analysis

Many papers define the algorithms for creating painterly rendered images, but almost none of them attempt to identify the quality of the images being produced. While it can be argued that paintings are subjective and a qualitative analysis can be dependent on the context, painterly rendering is a transformation which can be measured for quality.

This section attempts to define a quality measure for the painterly rendering transformation but makes no serious attempt to measure the quality of this transformation. A qualitative measure would require feedback from a user study, and while that may have merit. such user studies are sensitive to the users' personal perspectives, their cultural background, their abilities or training and doesn't address whether or not the transformation from photo realism to non-photorealism is within desired error thresholds.

5.2.1 Multi Scale Colour Error Analysis

Painterly rendering is a transformation that takes a source image and produces a painterly style output image: S $\Rightarrow$ P(S). I argue that a painterly image produces

pixel colours that represent aliased versions of the source images' pixel colours. I argue that it should be valid to compare the source image's colour values against the painterly image's colour values to create a measure of similarity.

The result of a painterly algorithm is to intentionally modify the placement and colour of the source image's pixels but that the output pixels intentionally relate to the source image's pixels. The chart shown in Figure 5.3 shows how much the painterly image's pixel colours deviate from the source image's pixel colours.

5.2.2 Optical Flow Error Analysis

For IsRaPtRS, the paint stroke advection operates in image space for both the optical-flow and the geometric estimations. The decision to use images to store paint stroke state was made in part to the efficiency of relating image space operations to the way that the GPU's fragment shader operates. While it has been shown that the image-space pipeline described here is faster than any existing published animated painterly rendering method, there is expected to be aliasing discontinuities or errors as a result of using any image-space advection method.

To calculate the errors encountered by using image-space advection I compute those pixels that have been moved, through advection, far enough away from the pixel's center to exceed the extents of the pixel's boundary (breach event). Paint strokes that have breached the extents of the containing pixel have to be passed to the adjacent pixel to which it has traveled.

The percentage of paint strokes that have breached its containing pixel's boundaries is shown in Figure 5.4.

Firstly, it can be seen that as the pixel size increases via the use of the multi scale (mip map) buffers, the frequency of breach events decreases at a rate of M^2 where M is the multiscale level. The M^2 decrease in breach events should be expected since the amount of advection vectors that reach the extents of the pixel's range should be expected to decrease at M^2 since the pixel's range grows at M^2 with increased scale due to the multiscale level.

Secondly, between the same images with the same multi scale level, the frequency of breach events increases as the optical flow vectors increase. In other words, as the object increases in speed in screen-space so does the likelihood of advection breach events. Again the increase in breach events is not unexpected since the relative range to the extents will decrease as the optical flow vectors increase, resulting in a shorter period of time to a breach event and therefore increasing the frequency of breach events.

While the advection buffer stores 2 advection vectors [Px, Py, Sx, Sy], a paint stroke that breaches its pixel's extents can be stored and used in the adjacent pixel's secondary advection storage space.

For paint strokes that have a low likelihood of a breach event due to a slow moving object in the scene, or a low resolution multi-scale buffer, the secondary buffer can allow paint strokes to smoothly advect through the image-space buffer with fewer discontinuities. For paint strokes that have a high likelihood of a breach event, such as the high-frequency multi scale buffers, it is likely that there may be multiple paint strokes advecting to the same pixel's extents.

Since there is only 1 secondary advection storage space, multiple breach events that result in the same pixel's extent range results in a collision that requires that only 1 of the advection vectors succeed in the image-space advection. In other words, a collision event will result in one or more paint strokes simply disappearing due to lack of storage space for it.

In the default configuration two-dimensional advected vectors can be contained within a standard four-channel, two-dimensional texture buffer since there is only room for a primary vector and a secondary vector to handle a single collision event. It is possible to add additional buffers to handle additional collision events, but as Figure 5.5 shows, the likelihood of a collision due to multiple breach events is a very small percent of the paint strokes being rendered.

The percentage of paintstrokes that result in a collision due to multiple advection vectors breaching at the same time and resulting in the same adjacent pixel range is

shown in Figure 5.5.

The first thing to notice about the advection collision data is that the likelihood of a collision event is an order of magnitude less likely than a breach event. A dramatically smaller collision likelihood is good news since a collision event cannot be solved without additional buffers to store the collisions in.

Also like the breach events, the likelihood of a collision event decreases at a rate of M^2 with the increased range of the multi scale buffers. Also like the breach events though, the likelihood of a collision event increases with an increased optical flow vector magnitudes.

5.3 Memory Cost

The buffers used to calculate and store the paint stroke properties are in some cases in addition to the memory cost of an existing rendering pipeline. While the source image and its mip-maps are likely to be available from an existing rendering pipeline, the orientation/detail buffer, the optical flow, reconciliation and advection buffers are likely to be in addition to an existing real-time rendering system.

To measure the additional memory cost of each of these buffers, only the source image dimensions are required as well as the pixel size. Since all of the buffers used to calculate the paint stroke properties use multiscale (mip-map) buffers the formula can be used for all to determine the memory cost of all of these buffers using the scalar MMS as is derived in Equation A.7.

$$\text{memory(I)} = \text{buffercostperpixel(I)} * \text{width(I)} * \text{height(I)} * \text{MMS}$$

where MMS is the scalar derived from the formula in Equation A.7.

For my painterly rendering system the buffers use the following pixel sizes:

1. **Colour/Intensity:** RGBA8 (4 bytes)

2. **Orientation/Detail:** RGBA8 (4 bytes)

3. **OpticalFlow:** RGBA8 (4 bytes)

4. **OpticalFlowReconciled:** RGBA8 (4 bytes)

5. **Advection:** RGBA16 (8 bytes)

A painterly rendered image that has the following source images and uses the geometrical optical flow method will cost the following in storage buffers alone:

Render Target Size	Optical Flow Method	Calculation	Memory Cost
RT($I_{1024x1024}$)	Geometric OF	(4+4+4+16)*1024*1024*1.333	37.333 MB
RT($I_{512x512}$)	Geometric OF	(4+4+4+16)*512*512*1.333	9.333 MB
RT($I_{256x256}$)	Geometric OF	(4+4+4+16)*256*256*1.333	2.333 MB
RT($I_{1024x1024}$)	Image OF	(4+4+4+4+16)*1024*1024*1.333	42.666 MB
RT($I_{512x512}$)	Image OF	(4+4+4+4+16)*512*512*1.333	10.666 MB
RT($I_{256x256}$)	Image OF	(4+4+4+4+16)*256*256*1.333	2.6666 MB

Table 5.1: Calculation of painterly buffer cost

5.4 The User Interface

The experiments that were performed made every attempt to expose the parameters described in section 3.2.1 and make them available for use in these experiments. Figure 5.6 shows a screen shot of the user interface provided to the user to achieve the desired goal.

As is the case with most user interfaces, if they are visible within the application window, then they are occupying valuable screen space that could be rendering the painterly images. In addition, the preparation and rendering of the UI itself can have a non-trivial performance cost to it, which can skew any performance metrics taken for the painterly rendering algorithm.

These issues motivated the design decisions made for implementing the UI for the painterly rendering application.

The user interface in fig 5.6 is implemented in C# as a separate application from the application that renders the painterly images. The UI communicates with the painterly rendering application over a TCP connection via local-host. The communication between the user interface and the renderer is a unidirectional, master/slave type relationship where the C# UI gives commands through the TCP connection and the painterly renderer receives those commands and adapts its rendering parameters accordingly.

The following describes the elements shown on Figure 5.6 and their function in the painterly rendering application.

1. **Display mode:** Selects which rendering mode is to be used. Most of the options given in the drop-down simply display the intermediate buffers used to define the painterly rendering parameters.
2. **Show profiler:** Toggles on/off the performance profile display.
3. **Resolution Min/Max:** Defines which mip-maps will be used for rendering paint strokes. The Min value defines the highest detail multi resolution buffer and the Max value defines the lowest detail multi resolution buffer.
4. **Paint stroke length:** Defines the magnitude of the primary axis of the paint strokes.
5. **Width/Length Max/Min:** Defines the ratio of paint-stroke width to length. (i.e. 1.0 is a square and 0.01 is a thin line.) The Max value applies to the Max multi resolution level, the Min value applies to the Min multi resolution level and all multi resolution levels in between are linearly interpolated between these two values.
6. **Pos Rand:** Defines a random positional offset.
7. **Orient Rand:** Defines a random orientation offset.
8. **Size Rand:** Defines a random size scalar.
9. **Opacity Max/Min:** Defines the opacity of the paint stroke geometry. The opacity is multiplied with the texture opacity applied as the paint stroke texture.

The Max value applies to the Max multi resolution level, the Min value applies to the Min multi resolution level and all multi resolution levels in between are linearly interpolated between these two values.

10. **Hier Thresh Max/Min:** Defines the cutoff for the given detail value calculated in the *orientation_detail* pass. The cutoff defines whether or not to render a paint stroke for the given region. The Max value applies to the Max multi resolution level, the Min value applies to the Min multi resolution level and all multi resolution levels in between are linearly interpolated between these two values.

11. **Geometric Optical Flow:** Toggles whether or not to use geometrical movement vectors as opposed to optical flow approximations for calculating advection.

12. **Optical Flow Match Threshold:** Defines the threshold to accept or reject the optical flow approximation vectors.

13. **Animation Speed:** Defines how fast the object will move on the screen.

14. **Animation Type:** Defines the type of animation is applied to the object in the scene as described in section 5.4.1.

15. **Optical Flow Display Threshold:** Defines a scalar to base the optical flow vectors on as explained in section 5.4.2.

16. **Source Image:** Defines which of the 16 images to use as the source image.

17. **Source Image Type:** Defines whether to used the source images in BMP format or DXT (DDS/S3TC) format. DXT was originally u to determine the rendering cost difference between compressed and uncompressed texture formats, but it turned out there was no noticeable difference so only BMP is currently used.

18. **Source Image Resolution:** Defines the size of the source image to use. The choices are 1024x1024, 512x512, 256x256 or 128x128.

5.4.1 Animation Type

The animation types applied to the source images were intentionally simple. These basic transformations were defined to identify the basic types of transformations that would be applied to more complex geometry in a painterly rendered scene. For the purposes of this description, the Z vector defines the camera's at vector, the X vector defines the camera's Right vector and the Y vector defines the camera's up vector. The transformations that were defined were:

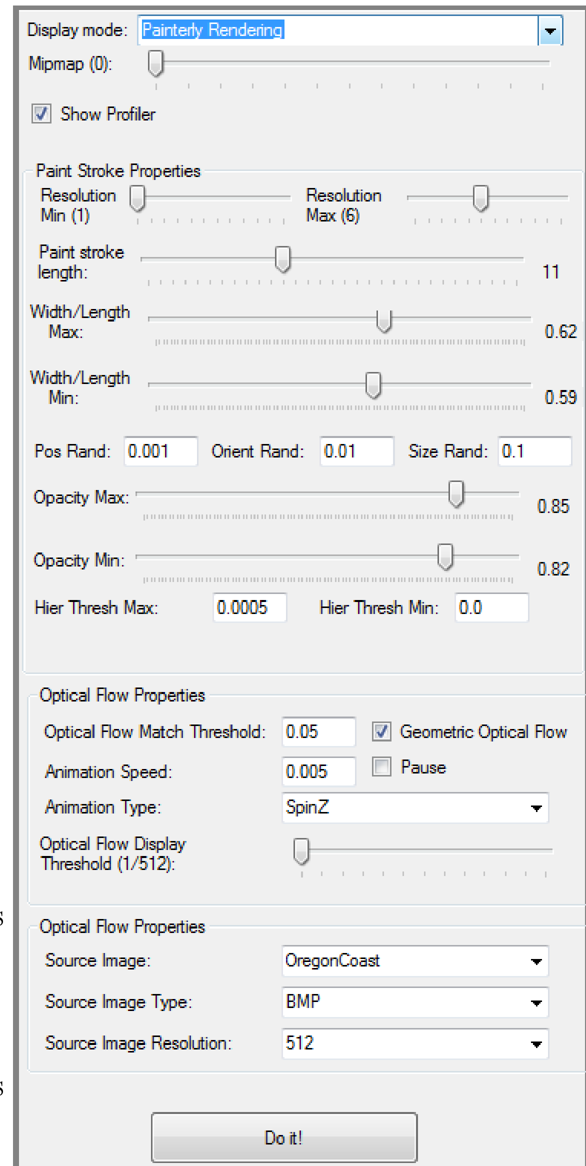


Figure 5.6: Screenshot of the user interface

1. **SpinZ:** Transformation spins the object around the Z axis with an offset to the center of view.

2. **RotateZ:** Transformation rotates the object around the Z axis.

3. **TranslateX:** Transformation moves the object back and forth along the X axis.

4. **TranslateY:** Transformation moves the object back and forth along the Y axis.

5. **RotateX:** Transformation rotates the object around the X axis.

6. **RotateY:** Transformation rotates the object around the Y axis.

7. **ScaleZ:** Transformation scales the object along the XY plane.

5.4.2 Optical Flow Display Threshold

It is useful to be able to visualize the various buffers that are being calculated along the rendering pipeline for IsRaPtRS. Being able to visualize the optical flow vectors or the geometric movement vectors are no exceptions but due to their nature, it can be more complex to display them than the other data buffers calculated for painterly rendering.

The optical flow vectors that are calculated for slow-moving pixels can result in vectors that are smaller in magnitude than is able to be displayed in an RGB8 display format or is discernible by the human eye if floating point formats are used for storage of these vectors. The optical flow vectors need to be scaled to a length that better matches the 0..1 range that the display hardware is using as input.

The "Optical Flow Display Threshold" value allows the user to define the scalar of which to scale the optical flow vectors to get them closer to the range 0..1. The values provided are S=[1, 1/2, 1/4, 1/8, ..., 1/512, 1/1024, 1/2048]. These values define the maximum value that will be mapped to a value of 1. In other words, the optical flow and geometric movement vectors will be scaled by the inverse of these values and clamped to the range 0..1.

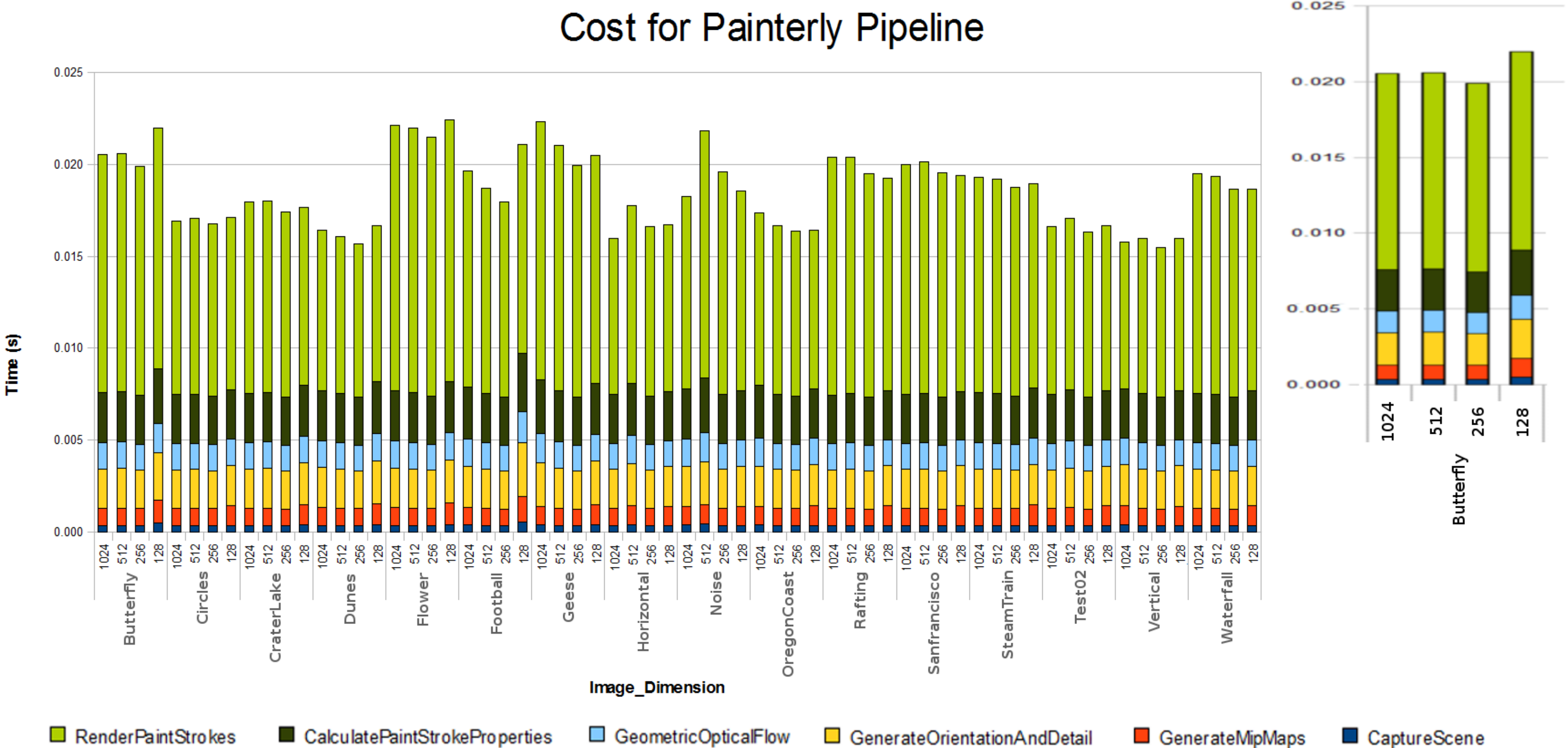


Figure 5.1: Comparing Different Source Images and Different Source Image Resolutions

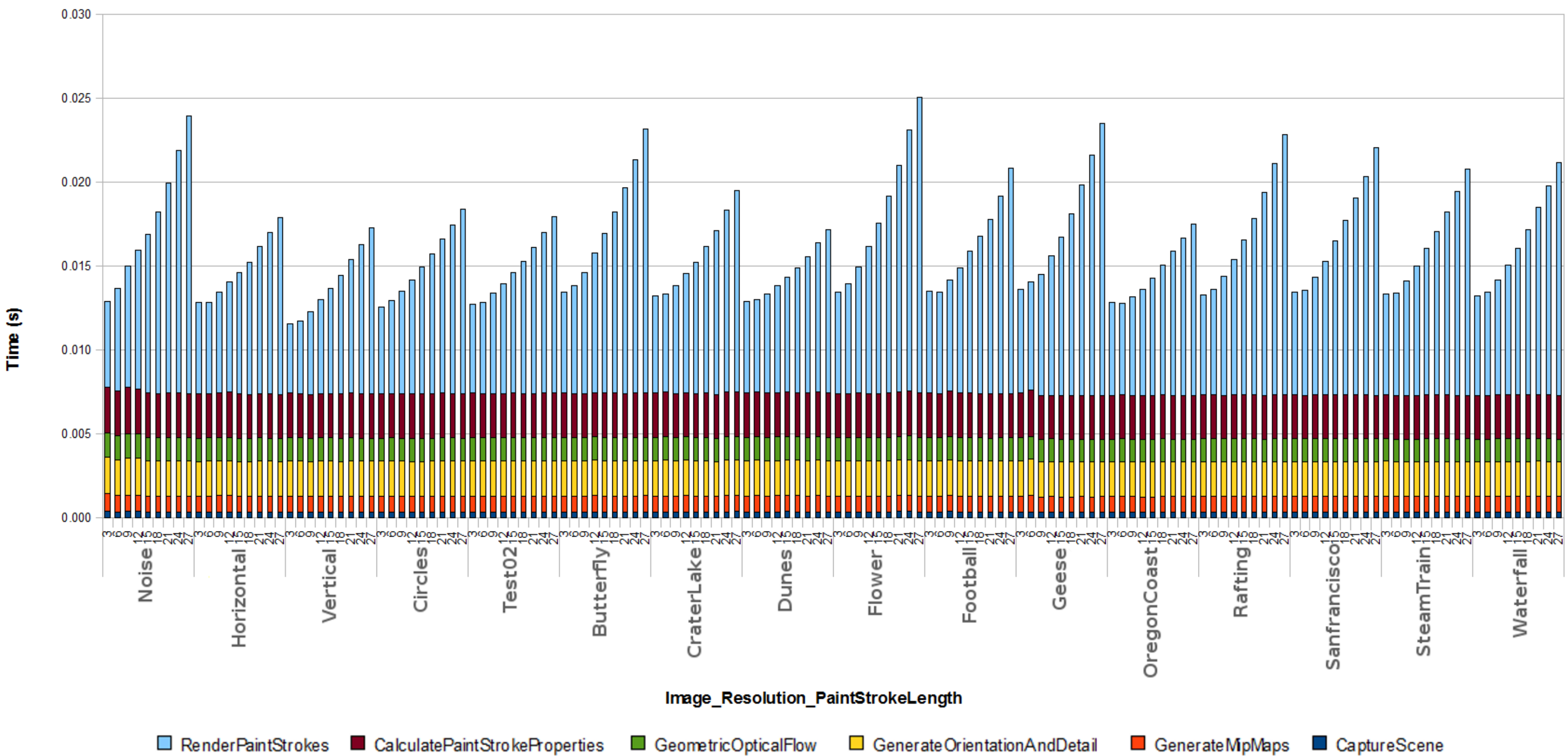


Figure 5.2: Comparing Different Source Images as They are Rendered with Varying Paint Stroke Lengths

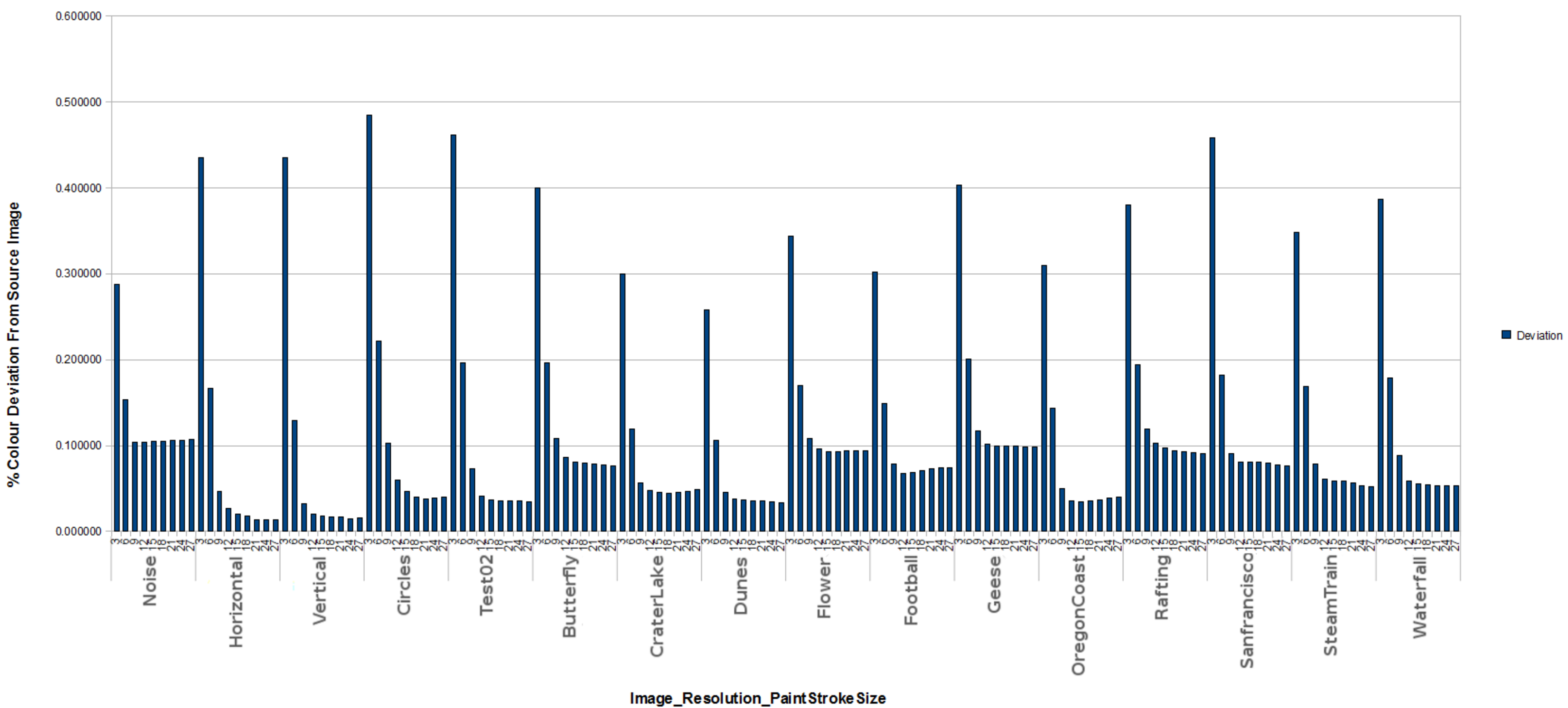


Figure 5.3: Comparing (per-pixel) the painterly image colour deviation from the source image.

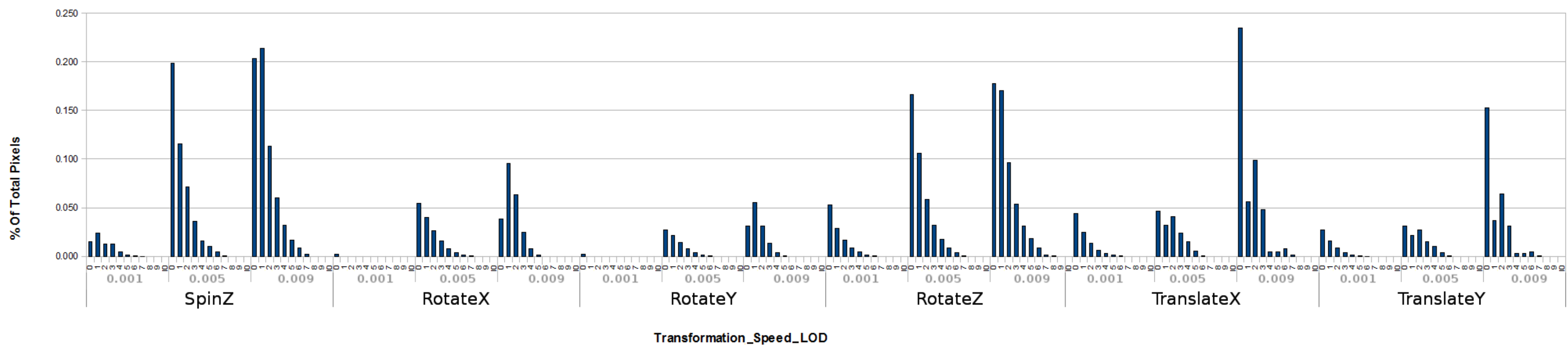


Figure 5.4: Measuring the percentage of pixels that are breaching their extents at any given time due to advection.

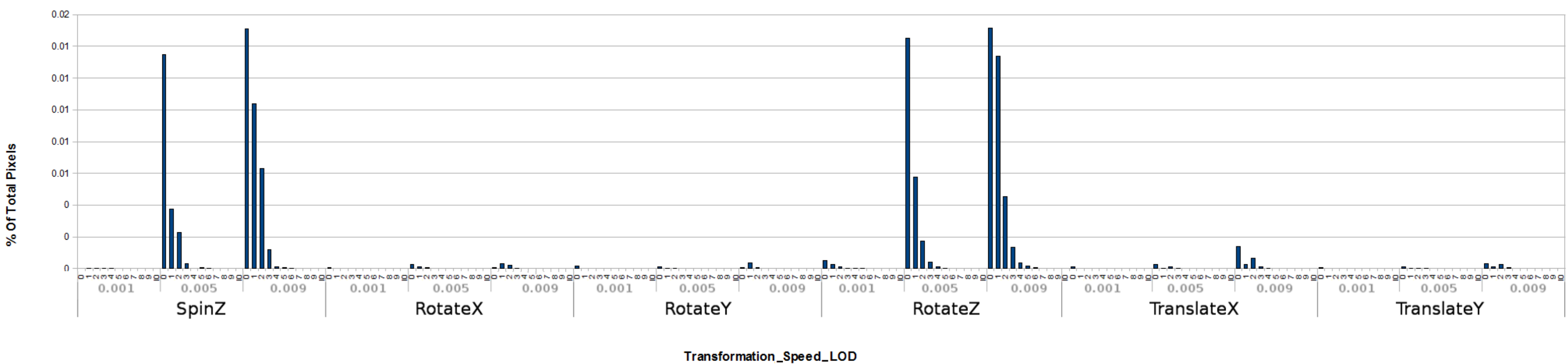


Figure 5.5: Measuring the percentage of pixels that have more than one advection vector competing for transfer at any given time due to advection.

Chapter 6

Conclusions

Painterly rendering has been proven to be possible for trivial real-time applications, but the transformation cost is not trivial. My research improves upon the cost in time of previous work and my results further prove that painterly rendering is feasible for real-time applications. Unfortunately this research also has shown that the cost of painterly rendering in both time and quality has a dependency on how the paint-strokes are rendered and as with most complex algorithms there is clearly a trade-off between performance and quality.

The numerical results show that transforming an image to a set of paint-strokes has an increasing cost at a rate of M^2 in time depending on the size of the paint-strokes (where M is the multiscale level). Fortunately, these results show that the errors of the transformed image also decreases at a rate of M^2 depending on the size of the paint-strokes. Taking these two properties into account, it should be possible to find a paint-stroke size that satisfies a reasonable trade-off between performance and quality.

6.1 Performance

The cost in time for rendering the necessary paint-stroke properties and the paint-strokes themselves may not be an acceptable amount of overhead for an application designer. Despite these results showing the performance/quality trade-off, the total overhead for painterly rendering may still be too expensive to add to an existing real-

time application without the need for substantial amounts of changes made prior to the painterly rendering pipeline.

The point that is likely to be the deterrent to a developer of a real-time application is the significant basic overhead to construct the paint-stroke properties and the sensitivity of the quality/performance trade off.

Referring to Figure 5.2, the minimum cost for transforming an image to a painterly rendered one is 11.6ms. For an application that is intended to run at 60 frames-per-second (FPS), this update time consists of 70% of the entire update period. (60FPS = 16.666 ms update period). With an existing rendering pipeline taking place within the CaptureScene and GenerateMipMaps portion of the intended rendering pipeline, and most likely will approach a 16ms update period itself, adding a painterly rendering post-processing stage will take a 60FPS application and make it work to run at 30FPS.

Fortunately I am not making the claim that I have done everything possible to make my application run as efficiently as possible. It is very likely that it may be possible to shrink the scale of the performance values shown in this document, resulting in an increased likelihood of a stroke-based-painterly rendering system being acceptable for specific real-time applications.

What continues to be troubling is that the update cost of the last stage of the rendering pipeline (from fig.5.2), RenderPaintStrokes, contributes as much or more than the entire cost of all the previous stages combined in most cases. While, in hind-sight, the paint stroke rendering cost is not surprising, it does mean that the feasibility of any stroke-based painterly-rendering system is most likely limited to the raw processing power of the hardware it is being run on. Unfortunately this conclusion means that no amount of preprocessing or cleaver parameter generation or usage is likely going to significantly improve the performance of a painterly rendering system without cutting corners somewhere.

So if it is true that a real-time painterly-rendering system is dependent on the processing power of the GPU it is running on or can some sort of optimization *could* be done to improve the situation?

One such possibility for minimizing the fill rate cost could be to remove blending on many of the pixels being rendered. Without having to perform a mix operation A.3, the cost of many of the fragments written would have slightly less work to do when outputting their results.

Unfortunately the quality of the resulting image is in many cases arguably greatly improved when semitransparent paint- strokes are rendered. The effect is more obvious for the larger paint strokes, where semitransparent paint-strokes give an almost water-colour look when rendered. Fortunately as the paint-strokes get smaller, the effect is less pronounced.

It may be possible to disable blending on the high-frequency paint-strokes without significant negative effect and improve the performance in the process.

Another technique that may improve performance a bit would be to be more careful of the use of paint-stroke images for the various multi scale levels. Choosing paint-stroke images that provide the visual features desired for each multi-scale level may allow for less paint-strokes to be necessary to maintain a similar level of quality. While varied paint stroke textures will not likely fix a significant amount of the fill-rate cost, some portion of the previously mentioned 11.6ms cost may be avoided by allowing for a better set of parameters to be selected for a similar resulting quality.

There wasn't a lot of analysis of the subtle changes to performance that were made while there was a lot of time spent on tuning this painterly rendering application. For instance, it is widely known that conditional branching in a shader program is not friendly to performance since the program will likely have to execute both branches of the condition. Using the step function and mix functions it may be possible to remove conditionals from the shader and take control over how the flow control is processed. Unfortunately for the experiments done for this research, there was an almost unnoticeable change in performance characteristics between using branches and its serialized alternative.

6.2 Image Quality

It is no surprise that the overall image quality is dictated by the parameters that define the paint-strokes. Attributes such as paint-stroke width/height, opacity, position, orientation and texture all contribute to the overall image quality. While a painted image's quality is subjective by definition, the parameters that define the creation of the paint-strokes influence the subjective and objective quality of the image.

This research has shown that the objective quality of the painterly transformation process is strongly influenced by the coverage of the paint-strokes themselves. Obviously if there lacks a reasonable amount of coverage, the transformation is going to lack resemblance to the original image. This research has also shown that too much coverage results in an over abundance of paint-stroke aliasing which too decreases resemblance to the original image.

6.2.1 Personal Observations on Quality

While I performed no formal subjective analysis of IsRaPtRS I feel it is useful to describe some of my own observations I collected as I performed the research and development.

A naive painterly rendered system looks too procedural. Simply taking data as input and outputting paint-strokes results in an image that looks like a broken bitmap. I found that adding a very small amount of randomness to the paint-stroke position, orientation and size severely decreased the procedural look of the output image which resulted in a more human made looking image.

While adding randomness increases the objective errors measured, the resulting images simply looked better. This may be an agreement to the comments made in [2] whereby high-frequency interference patterns created by the lack of randomness tends to solicit the viewer's attention, but in this case, to the incorrect area of the image. Having your view attracted to an unexpected or innocuous area of the image due to repetitive interference patterns simply makes the image look wrong.

6.3 Motion

I chose to calculate paint-stroke motion in image-space for my research since the GPU handles processing per-pixel operations very well. An additional motivation for using image data as source data and storing paint-stroke parameters is that pixel data is the primary output of most rendering systems. This means that the results of my research could be appended to an existing rendering pipeline.

The most obvious problem with using pixel data for calculations is the fact that each pixel is an alias for the area around its location. This discrete definition of a pixel means that there are no actual representations of the data except at the exact coordinates of the pixels. While there are methods to estimate the values between the pixel coordinates, they are just estimates.

Another way to look at this pixel aliasing problem is that the data that is stored at a pixel location describes the area around that coordinate. Again, as the application samples the pixel data there must be a way to blend between the values represented by adjacent pixels.

For paint-stroke motion my application calculates a pixel motion vector either by optical flow, or by calculating the geometric motion in screen-space. These values are stored in pixel data for convenience to match the data structures the GPU and the rest of the pipeline is working with. With other pixel data this storage method means that there needs to be some way to represent the data as the application attempts to sample from between pixel coordinates.

Fortunately, optical flow and geometric motion results in a simple 2 dimensional vector for each pixel. These vectors can be accumulated and used for paint-stroke advection. As long as the accumulated vectors stay within the extents of the pixel's range, the paint-stroke's motion is guaranteed to be as continuous as the motion itself.

When the paint-stroke breaches the pixel's extents there is a possibility of discontinuities and unfortunately since this system uses a discrete and periodic coordinate system via the pixel buffers, these discontinuities result in an instantaneous paint-stroke appearance or disappearance.

The discontinuities in the advection buffers have been shown to be predictable in their frequencies. Using multi-scale buffers, it has been shown that these discontinuities occur with M^2 greater likelihood in the higher frequency multi-scale buffers. This turns out to be a good result since discontinuities in the low-frequency buffers show through the large paint-strokes, which in turn are much more obvious to the user. Discontinuities in the high-frequency data buffers (and smaller paint-strokes) are not as noticeable which means they are more likely to be acceptable when they do occur.

I have also identified that while breach events can result in discontinuities, multiple breach events for an adjacent data buffer that results in advection to the same data element is unsolvable without additional storage space. My results show that these "collision" events do occur, they occur with an order of magnitude smaller frequency than the breach events and like the breach events they decrease with M^2 frequency as the multi-scale buffer decreases in frequency.

6.4 Future Work

While there were many experiments done and a large amount of observations and data taken, this research is by no means complete. This section describes some of the obvious next steps that should be taken to continue the work started here.

6.4.1 Comparing Geometric Advection Methods

It would be useful to perform some of the geometric-centered advection calculations outlined in [6] and compare with the image-space advection experiments done here. Although they do describe their algorithm quite well, and they have achieved performance unparallelled by previous work, they don't itemize where the costs of the various stages of their experiments are taken from.

6.4.2 Discrete Particle Image-Space Advection

The advection calculation performed in this research is admittedly simplistic and could probably take advantage of more analysis. Unfortunately there is a lot of work that has been done for tracking particle advection from a statistical perspective. My method requires that discrete particles are tracked in a uniform grid, of which there was no research found that fulfilled these conditions.

The reason I believe that fulfilling these conditions is useful is because, unlike the existing methods used, by tracking discrete particles through a uniform grid, there is a guarantee of some level of particle density with the side-effect that it doesn't require a variable sized buffer to store the results in. Of course removing the guarantee of a particle's existence in favour of a particle density guarantee has side effects, but in for some applications this trade-off may be acceptable.

6.4.3 3D Models and Video

The bulk of the work presented in this research is attempting to determine if there exists a method to allow for real-time painterly rendering at the most fundamental level. With the proof that it is possible to perform the transformation from an arbitrary image to painterly rendered image in real-time, it would be useful to evaluate how IsRaPtRS performs with more complex data.

As has been done in several of the other painterly rendered research, the input to the algorithm used video or 3D geometry as the source of the animation. While complex 3D geometry was intentionally left out to reduce the number of variables to be quantitatively measured, it would be useful to compare with this research.

6.4.4 Surface Properties

I attempted to implement all of the paint-stroke properties that require only simple computation to render. While I was able to implement most of them, surface properties such as paint stroke diffuse lighting properties and shininess have yet to be completed. Surface lighting effects were left out of this research mainly because

it is guaranteed to only add to the rendering cost of the algorithm and that only a small handful of the existing research has included surface lighting. Of the existing research, it is questionable of whether or not it would benefit this research or simply add more noise to the discussion posed here.

With that said, adding lighting effects to the other effects presented here may result in a reduction in the number of created or destroyed paint strokes through IsRaPtRS.

6.4.5 Performance Profiling

I always feel that a rendering algorithm can benefit from more performance profiling and this research is no different. For the entirety of researching and developing IsRaPtRS I always felt that the profile data I was getting back was a little higher than I expected. The general consensus with my colleagues and my industry contacts are that my research may benefit from using DirectX as opposed to OpenGL.

In the case of the reconciled optical flow, I was never able to get a performance profile that was logical. For the complexity of the shader to calculate the reconciled optical flow, there seems to be no logical reason for its cost. The only conclusion I could speculate on is that the reconciliation shader is the only one that needs to sample to a lower mip-map level while sampling from a higher mip-map level although most of the other shaders used easily sample from a lower mip-map level and write to higher mip-map levels.

6.4.6 Measure Optical-Flow vs Geometric Movement Results

It may be beneficial to determine the amount of error produced by the various test cases between the single-pass optical flow calculation and the equivalent geometric movement calculation. It is known that the optical flow calculation can approach a 100% error with specific test cases such as horizontal lines image with a translateX animation or circles image with a rotateZ animation. I believe that comparing the image noise level with the optical flow error result would be a useful exercise.

6.4.7 Measure The Error Introduced Through Vector Compression

Most of the optical flow / geometric movement vectors ended up using the compression scheme where the XY vector components were scaled by $E=2^Z$ and the results stored as XYE8: [RGB]=[X*E, Y*E, 1/Z].

I ended up using the XYE8 compression scheme for all the vector buffers because it reduced the time cost of the shaders by half. Unfortunately compressing vectors in this way is a lossy compression scheme and although the compression benefits the performance argument it really needs a measure of the error introduced by using it.

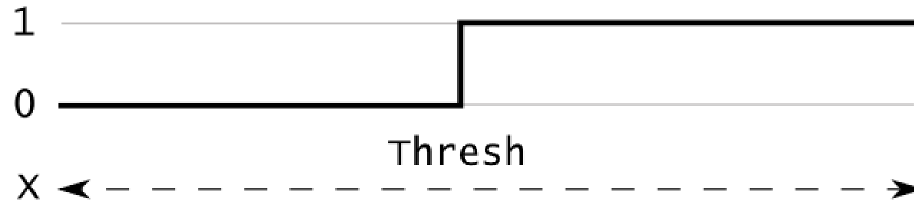
My research was focused on the performance considerations primarily and since the XYE8 vector compression scheme is a reasonable approximation of the original vector for the purposes of image-space advection, the actual error metric was ignored. Ideally the error should be calculated or measured as well for this compression scheme.

Appendix A

Additional Information

A.1 Formulas

$$step(x, threshold) = \begin{cases} 0.0 & \text{if } x < threshold \\ 1.0 & \text{otherwise} \end{cases} \tag{A.1}$$

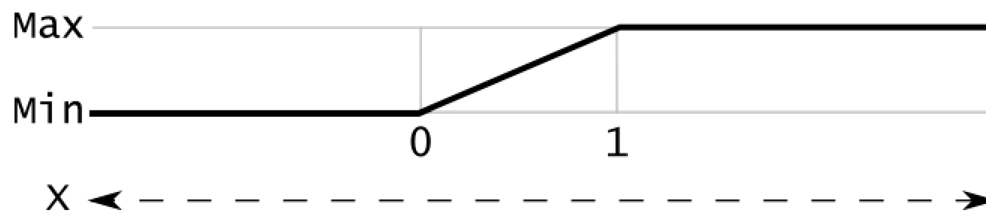


Equation A.1: Step function.

$$window(x, min, max) = \begin{cases} 0.0 & \text{if } (x < min) \text{ or } (x > max) \\ 1.0 & \text{otherwise} \end{cases}$$
$$= step(min, x) * (1.0 - step(max, x)) \tag{A.2}$$

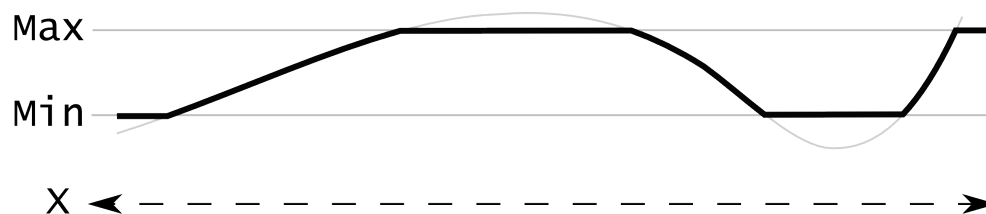
Equation A.2: Window function

$$mix(x, min, max) = \begin{cases} min & \text{if } x < 0.0 \\ max & \text{if } x > 1.0 \\ min * (1.0 - x) + max * x & \text{otherwise} \end{cases} \tag{A.3}$$



Equation A.3: Mix function

$$clamp(x, min, max) = \begin{cases} min & \text{if } x < min \\ max & \text{if } x > max \\ x & \text{otherwise} \end{cases} \tag{A.4}$$



Equation A.4: Clamp function

A.2 Example Images

$$compressToXYE8(X,Y) =<< \frac{X*2^E+1}{2}, \frac{Y*2^E+1}{2}, \frac{E}{32} >> \\ whereE = \lceil max(-log_2(X), -log_2(Y)) \rceil \tag{A.5}$$

Equation A.5: Compress <x,y> to XYE8

$$decompressFromXYE8(X,Y,E) =<< \frac{2*X-1}{2^{32*E}}, \frac{2*Y-1}{2^{32*E}} >> \tag{A.6}$$

Equation A.6: Decompress from XYE8 to <x,y>

$$\begin{aligned} MMS &= 1 + 1/4 + 1/16 + 1/64.... \\ (1/4)MMS &= 1/4 + 1/16 + 1/64 + 1/256.... \\ MMS - (1/4)MMS &= 1 \\ MMS &= 1/(1-1/4) \\ MMS &= 1.3333 \end{aligned} \tag{A.7}$$

Equation A.7: Mipmap size calculation

A.3 Glossary

Advection:

1. The transfer of a property of the atmosphere, such as heat, cold, or humidity, by the horizontal movement of an air mass.

2. The rate of change of an atmospheric property caused by the horizontal movement of air.

3. The horizontal movement of water, as in an ocean current.

Deadline:

1. A time limit for completion of a task which if exceeded is deemed to be considered a failure of the task to complete within its allotted time.

Graphics Processing Unit (GPU):

1. A specialized microprocessor that offloads and accelerates graphics rendering from the central (micro-)processor.

Interactive:

1. Allowing or relating to continuous two-way transfer of information between a user and the central point of a communication system, such as a computer or television.

Isoline:

1. A function of two variables is a curve along which the function has a constant value (also contour line).

Medium:

1. A means or instrumentality for storing or communicating information.

2 An intervening substance through which signals can travel as a means for communication.

Non-Photorealistic Rendering (NPR):

1. An area of computer graphics that focuses on enabling a wide variety of expressive styles for digital art. In contrast to traditional computer graphics, which has focused on photorealism, NPR is inspired by artistic styles such as painting, drawing, technical illustration, and animated cartoons.

Photorealism (PR):

1. Is the genre of painting based on using the camera and photographs to gather information and then from this information, creating a painting that appears to be very realistic like a photograph.

Psychophysical:

1. A discipline within psychology that quantitatively investigates the relationship between physical stimuli and the sensations and perceptions they affect.

Stroke Based Rendering (SBR):

1. An automatic approach to creating nonphotorealistic imagery by placing discrete elements called strokes, such as paint strokes or stipples.

Style:

1. The aspects of visual appearance of a work of art and how these aspects may relate to other works.

Texel:

1. A texel, or texture element (also texture pixel) is the fundamental unit of texture space[1], used in computer graphics. Textures are represented by arrays of texels, just as pictures are represented by arrays of pixels.

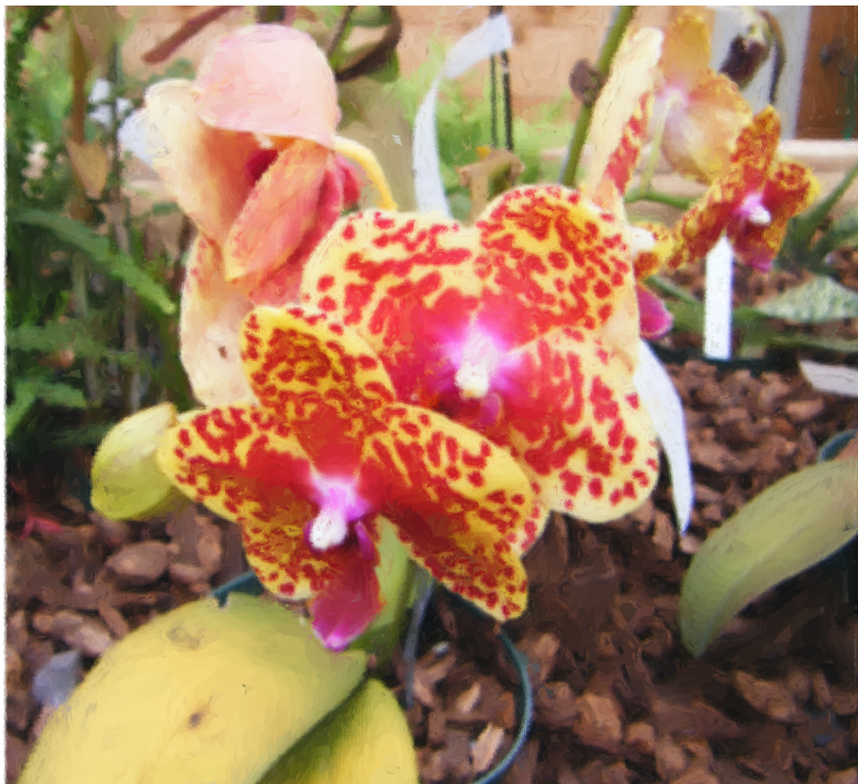
Topology:

1. The study of the properties of geometric figures or solids that are not changed by homeomorphisms, such as stretching or bending. Donuts and picture frames are topologically equivalent, for example.

(a) Butterfly

(b) CraterLake

(a) Dunes



(b) Flowers

(a) Geese

(b) OregonCoast

(a) Rafting

(b) Sanfrancisco

Figure A.1: Example painterly images

(a) SteamTrain

(b) Waterfall

Figure A.2: Example painterly images

Bibliography

[1] V. D. John Charles, *A Text-Book of the History of Painting.* Dodo Press, Aug 2007.

[2] C. G. Healey, L. Tateosian, J. T. Enns, and M. Remple, "Perceptually based brush strokes for nonphotorealistic visualization," *ACM Trans. Graph.*, vol. 23, no. 1, pp. 64–96, 2004.

[3] A. Hertzmann, "A survey of stroke-based rendering," *Computer Graphics and Applications, IEEE*, vol. 23, no. 4, pp. 70 – 81, 2003.

[4] B. Gooch and A. Gooch, *Non-Photorealistic Rendering.* AK Peters Ltd, 2001. ISBN: 1-56881-133-0.

[5] T. Strothotte and S. Schlechtweg, *Non-Photorealistic Computer Graphics: Modeling, Rendering and Animation.* Morgan Kaufmann, June 2002.

[6] J. Lu, P. V. Sander, and A. Finkelstein, "Interactive painterly stylization of images, videos and 3d animations," in *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, (New York, NY, USA), pp. 127–134, ACM, 2010.

[7] L. Kovács and T. Szirányi, "Painterly rendering controlled by multiscale image features," in *SCCG '04: Proceedings of the 20th spring conference on Computer graphics*, (New York, NY, USA), pp. 177–184, ACM, 2004.

[8] M. Shiraishi and Y. Yamaguchi, "An algorithm for automatic painterly rendering based on local source image approximation," in *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, (New York, NY, USA), pp. 53–58, ACM, 2000.

[9] D. Nehab and L. Velho, "Multiscale moment-based painterly rendering," in *Computer Graphics and Image Processing, 2002. Proceedings. XV Brazilian Symposium on*, pp. 244–251, 2002.

[10] D. Nehab, "Moment based painterly rendering," 2003. Princeton University.

[11] B. J. Meier, "Painterly rendering for animation," in *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 477–484, ACM, 1996.

[12] D. Sperl, "Realtime painterly rendering for animation," 2003.

[13] S. Bhattacharjee and P. Narayanan, "Real-time painterly rendering of terrains," in *Computer Vision, Graphics & Image Processing, 2008. ICVGIP '08. Sixth Indian Conference on*, pp. 568–575, December 2008.

[14] P. Litwinowicz, "Processing images and video for an impressionist effect," in *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 407–414, ACM Press/Addison-Wesley Publishing Co., 1997.

[15] A. Hertzmann and K. Perlin, "Painterly rendering for video and interaction," in *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, (New York, NY, USA), pp. 7–12, ACM, 2000.

[16] J. Hays and I. Essa, "Image and video based painterly animation," in *NPAR '04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, (New York, NY, USA), pp. 113–120, ACM, 2004.

[17] A. Hertzmann, "Fast paint texture," in *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, (New York, NY, USA), pp. 91–ff, ACM, 2002.

[18] A. Bousseau, F. Neyret, J. Thollot, and D. Salesin, "Video watercolorization using bidirectional texture advection," in *SIGGRAPH '07 ACM SIGGRAPH 2007 papers*, (New York, NY, USA), p. 104, ACM, 2007.

[19] N. Zhang, Y. shan Chen, and J. li Wang, "Image parallel processing based on gpu," vol. 3, pp. 367 –370, March 2010.

[20] M. Obaid, R. Mukundan, and T. Bell, "Enhancement of moment based painterly rendering using connected components," in *CGIV '06: Proceedings of the International Conference on Computer Graphics, Imaging and Visualisation*, (Washington, DC, USA), pp. 378–383, IEEE Computer Society, 2006.

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

[22] J. Hellstn, "Image-space painterly rendering," 2008.

[23] P. Haeberli, "Paint By Numbers: Abstract Image Representations," vol. 24, no. 3, pp. 207–214, 1990.

[24] S. C. Olsen, B. A. Maxwell, and B. Gooch, "Interactive vector fields for painterly rendering," in *GI '05: Proceedings of Graphics Interface 2005*, (School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada), pp. 241–247, Canadian Human-Computer Communications Society, 2005.

[25] J. Hays, "Interactive tensor field design and visualization on surfaces," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 1, pp. 94–107, 2007. Member-Zhang, Eugene and Member-Turk, Greg.

[26] B. Gooch, G. Coombe, and P. Shirley, "Artistic vision: painterly rendering using computer vision techniques," in *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, (New York, NY, USA), pp. 83–ff, ACM, 2002.

[27] R. M. Kirby, D. F. Keefe, and D. H. Laidlaw, "Painting and visualization," 2003.

[28] A. Hertzmann, "Painterly rendering with curved brush strokes of multiple sizes," in *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 453–460, ACM, 1998.

[29] S. Strassmann, "Hairy brushes," in *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 225–232, ACM, 1986.

[30] N. Chu, W. Baxter, L.-Y. Wei, and N. Govindaraju, "Detail-preserving paint modeling for 3d brushes," in *NPAR '10: Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, (New York, NY, USA), pp. 27–34, ACM, 2010.

[31] A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin, "Image analogies," in *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 327–340, ACM, 2001.

[32] N. Kanopoulos, N. Vasanthavada, and R. Baker, "Design of an image edge detection filter using the sobel operator," *Solid-State Circuits, IEEE Journal of*, vol. 23, pp. 358 –367, Apr. 1988.

[33] S. Hargreaves and M. Harris, "Deferred shading," 2004.

[34] G. James and J. O'Rorke, *Real-Time Glow*, pp. 343 –362. GPU Gems, Addison-Wesley, 2004.

[35] T. Sousa, *Adaptive Glare*, pp. 349–355. Shader X^3: Advanced Rendering with DirectX and OpenGL, Charles River Media, 2005.

[36] T. Scheuermann and N. Tatarchuk, *Improved Depth-of-Field Rendering*, pp. 363–377. Shader X^3: Advanced Rendering with DirectX and OpenGL, Charles River Media, 2005.

[37] D. Gillham, *Real-Time Depth-Of-Field Implemented With A Postprocessing-Only Technique*, pp. 163–175. Shader X^5: Advanced Rendering Techniques, Charles River Media, 2007.

[38] D. Filion and S. Boisse, *Volumetric Post-Processing*, pp. 571–577. Game Programming Gems 5, Charles River Media, 2005.

[39] M. Krazanowski, "A more accurate volumetric particle rendering," *Intel Software Network and Gamasutra.com*, June 2009.

[40] C. Oat and N. Tatarchuk, *Heat and Haze Post-Processing Effects*, pp. 477–486. Game Programming Gems 4, Charles River Media, 2004.

[41] O. M. Pastor and T. Strotthote, "Graph-based point relaxation for 3d stippling," in *ENC '04: Proceedings of the Fifth Mexican International Conference in Computer Science*, (Washington, DC, USA), pp. 141–150, IEEE Computer Society, 2004.

[42] B. K. P. Horn and B. G. Schunck, "Determining optical flow," *ARTIFICAL INTELLIGENCE*, vol. 17, pp. 185–203, 1981.

[43] G. Rosado, *Motion blur as a post-processing effect.*, pp. 575–581. GPU Gems, Addison-Wesley, 2007.