AN ABSTRACT OF THE THESIS OF

Mizuki Kagaya for the degree of Master of Science in Computer Science presented on May 1, 2009.

Title: Painterly Rendering Using Space-Time Varying Style Parameters

Abstract approved: _____

Eugene Zhang

Automatic painterly rendering systems have been proposed but they opted for selecting a single style to generate paintings from images, which lacks the ability of creatively using multiple styles to focus important objects and deemphasize unimportant part of the scenes. We provide a multi-style painting framework to address this issue as well as techniques to enhance the quality of the painting. To summarize, this project makes the following contributions.

- Multi-style painting framework: A canvas is considered as a collection of non-overlapping regions that correspond to objects or background of input images and videos. Within each region stroke orientation can be specified and user-defined painting style parameters can be assigned. Those assigned attributes are also used to define values for regions with any attribute unspecified.

- Color-based stroke ordering: By relating the drawing order of strokes to their colors, one can create more coherent paintings rather than a randomly assigned order.
- Implicit renderer: We propose a renderer with a different perspective from existing explicit approaches. Brush strokes are simultaneously generated by advecting images of seeds of the strokes in a way that they are repeatedly morphed according to stroke orientation.

Painterly Rendering Using Space-Time Varying Style Parameters

by

Mizuki Kagaya

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented May 1, 2009
Commencement June 2009

Master of Science thesis of Mizuki Kagaya presented on May 1, 2009.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Mizuki Kagaya, Author

ACKNOWLEDGEMENTS

I would like to thank all the people who have contributed to my thesis work. Eugene Zhang has been my major advisor since the beginning of my graduate school life. He has diligently advised me uncountable materials related to research and computer graphics. Without his tensor field design framework, my project would not be formed as it is. I have a great gratitude for Greg Turk, FreeFoto.com and Pics4Learning.com for various kinds of photographed image sources for my work. Artbeats has also cordially provided high-quality commercial video sources for my video paintings. Without those, this project would have been delayed by a considerable amount of time. Patrick Neill has analyzed performance of my framework and suggested various ways of optimization. He also helped with shader programming to accelerate my renderer, making it possible to paint images at interactive rate. Sinisa Todorovic, William Brendel, and Qingqing Deng collaborated on video productions as well as image/video segmentation coding. It is a great pleasure to have the multi-style funtionality included in my current framework. Todd Kesterson has provided his artistic point of view to evaluate painted results. His advice and suggestions were significantly helpful to develop my work further. I would also like to appreciate for all faculty members who willingly served as my committee members for the final oral examination. Eugene Zhang as the major advisor, Yuji Hiratsuka and

Sinisa Todorovic as committee members, and Michael Scott as a Graduate Council Representative.

Lastly, I am glad to have studied abroad for my undergraduate/graduate period at Oregon State University. This would never be possible without any cordial support from my family, friends I have met in college, graphics lab mates, and faculty members.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

Chapter 1 – Introduction

Painting is not an easy task. We look at a subject, choose a brush with a appropriate size and color, draw a stroke in a certain direction for a desired length, and then repeat this until we are able to capture sufficient details in order for the painted image by itself to tell what it is about. There is not a unique way to paint an image. We rely on our intuition and sometimes subconsciousness to decide the next strokes. Algorithmically instructing machines to simulate a process of drawing like we do is not straightforward. Bringing machine-drawn paintings toward a visually pleasing level is an additional hurdle to develop a good algorithm.

It was nearly two decades ago when Haeberli published a paper that introduces an idea of using brush strokes to paint from a single photographs [15]. Later, Hertzmann has developed an automatic algorithm for image paintings, achieving various painting styles using winding strokes with different sizes [19]. Hays et al. pushed a painting algorithm toward generating video paintings with temporal coherency [17]. Processing a video input requires a significant amount of time since it can consist of hundreds of static images. Temporal coherency of strokes must be paid attention; they should not move around randomly nor appear and disappear too frequently to be perceived as visual artifacts. Many other novel graphics papers discussing image/video paintings have also been published. However, to the best of my knowledge, none of them have focused on exploiting a set of multiple styles to depict within a single image or a video; two exceptions remain. Haeberli introduced a binary operator which takes two paintings with the same number of brush strokes to interpolate/extrapolate them [15]. Hertzmann demonstrated a temporal style

transition by interpolating two different sets of style parameters [19]. However, it applies to a whole canvas, not arbitrary segmented regions.

Many painting algorithms use a template to control stroke shapes. Short line segments or smooth spline curves are de facto a standard of stroke representation. To add more realistic details, they can be texture-mapped with opacity maps. Having multiple such opacity maps will give randomness to stroke appearances, but geometry that forms strokes is still simple. Their shapes are computationally determined by following contours from starting points. My project has started with a motivation of achieving the computation in parallel. Stroke drawing order must be re-ordered to diminish regular overlaps but it is the case that the order is randomly permutated so that strokes individually appeal themselves rather than coherently describe a content of a painting.

To summarize, this thesis provides a high-level idea of a framework that gives a new open to users to exploit multi-style paintings, plus a different approach of painterly rendering and a possibility of stroke drawing order. Contributions on this thesis are the followings.

- A multi-style framework of automatic painterly rendering is proposed, which produces both image and video paintings. The system gives to users interactive abilities to design or modify painting attributes and stroke orientation, as a form of keyframing, per segmented region. With keyframes the system automatically computes the rest of information for unspecified regions.

- Apart from the framework is a renderer based on that multiple strokes are

simultaneously rendered. Instead of computing stroke shapes in order, an image based approach [55] is used to indirectly compute them. The renderer is able to produce multi-layer paintings that show curved strokes with different sizes, similar to [19].

- Color-based stroke ordering comes in renderers. By taking care of a color bias in preference of draw order, similar colors are likely to show up in painting results so that stroke overlap looks more coherent.

This thesis consist of eight chapters, including this introductory session. In Chapter 2, we are going to review past related work. My framework borrows technical ideas from some of their works that will be introduced in that chapter. Overview on the multi-style painting framework is discussed in Chapter 3. This contains input specifications, capabilities of the framework, and workflow examples so readers are able to grasp pictures of what it can produce and how they use it.

More on algorithmical details are described in the following chapters. Chapter 4 shows how multi-style framework is built, including automatic computation of styles as well as stroke orientations. Implementations of renderers and color-based stroke orderings are described in Chapter 5. We introduce two types of renderers. The first one, called an explicit renderer is nothing new about implementation but it is used to compare with the other type in terms of rendering results. The second type, which could be imagined to be called an implicit renderer, was designed in a different perspective of generating brush strokes. Stroke ordering based on colors has risen up during development of the implicit renderer. There are more than

one way to define the ordering. We have populated a set of ordering rules to know which one would stand out.

Chapter 6 exhibits rendering results generated from my framework, including visual comparison between two renderers and different color ordering rules. A lot of discussion comes in Chapter 7, which will be leveraged for future development. Finally, as closing words, we spare some space to wrap up this thesis in Chapter 8.

Chapter 2 – Related Work

Many papers have been published on the topic of non-photorealistic rendering (NPR), but there are only a few formats as inputs they can take: 2D media such as images/videos and 3D models. My work falls into the former category, and it is painterly rendering using brush strokes, as opposed to other means such as using image filters. For those who are interested in painterly rendering techniques that are stroke-based, we refer to the survey of Hertzmann [22]. Other types of painting methods are image analogies [23], stylization and abstraction [6,10], watercolorization [3,4,9], charcoal sketch [2,40], pen-and-ink sketch [25,45,58], hatching [25,44], stippling/halftoning [11,43], shard-like painting [31], and mosaics [13,16,29].

2.1 NPR of Images

To the best of my knowledge, Haeberli is the first researcher who described an approach to image abstraction using a set of brush strokes. He mentioned basic rendering steps, computation of stroke attributes, and even interpolation/extrapolation of two paintings with different styles. While this framework involves manual stroke placement, the framework has inspired much research.

Automatic painting systems requiring less user interaction have been published. Cohen has designed artificial intelligence painter, AARON, which is capable of drawing human figures and plants [7]. Winkenbach introduced an automatic pen-and-ink sketch system for 3D models [58], and Salisbury et al. built an interactive pen-and-ink sketch system, where users supply a directional field to guide stroke

orientations for image inputs [45]. Curtis et al. built automatic watercolor simulation, which is applied to an automatic watercolorization for images and 3D models [9]. The system allows users to control pigment distribution in order to generate pleasing results. Litwinowicz's automatic painting system targeted for both image and video inputs [38]. Users do not have to use mouse to specify stroke positions nor orientations; they are instead automatically computed. Later, Hertzmann used a set of long strokes with different stroke sizes into his automatic painting system [19]. His idea was to capture much details from high frequency area while preserving low frequency area as much blurred as possible, and he achieved this introducing layers; at the bottom layer is a rough sketch painted with the largest stroke size. At each upper layer, a new set of finer brush strokes is generated only for regions where details are missing.

Hertzmann's idea of using layers implicitly but partially determines stroke ordering. Larger strokes should be drawn first, and they are painted over by upcoming strokes with smaller sizes. Collomosse leveraged salience of image feature for the ordering. Strokes around high salience regions are rendered last so the painted image still retains image content [8]. Other stroke based methods or stroke order within each layer in Hertzmann's approach just randomly permuted strokes in a draw list to avoid regular placements [19, 38].

Stroke orientations are typically determined by edges detected from a source image. They can be normal to local gradient of a blurred image [15, 19, 38], or they can be interpolated from only strong gradients [17]. Kovács extracted edges at multi-scale levels so that orientations at each pixels is computed from the closest

edge [34]. Gooch et al. estimated stroke orientations by finding medial axes from features of a segmented image [14]. Zhang et al. provided an interactive framework for both vector and tensor field design. Underlying field can be edited by putting design elements to fix orientation, canceling two singularities of different types, and smoothing inside a closed region [59, 60].

Using different stroke shapes gives different stylistic appearance on results. Haeberli offered a set of different geometries for brush strokes [15]. Litwinowicz used short line segments with a constant width. If they go across an edge detected from an input image, they are clipped to it in order to preserve details; thus their lengths vary [38]. Thickened spline curves are used for Hertzmann's brush stroke model so that they form arbitrary curved shapes to always follow image contours [19, 24]. For brush strokes to appear in a more realistic way, they can be textured with a stroke image while being rendered. Additional lighting effect can be further added as a post processing [21].

In many painterly rendering algorithms, computing stroke positions relies on a regular grid [19, 38], stroke areas [48] or a stochastic selection [17, 53]. Another approach is to treat stroke placement as an optimization problem. In other words, we look for a painting that best capture features in a source image [20]. This takes a form of non-linear energy optimization and is computationally expensive. Similar optimization procedure has been used for stippling [11], halftoning [43], or tile mosaics [16].

Some perceptual models or computer vision related techniques are employed in order to convey or analyze structure inside an source image. Santella and

DeCarlo built a system that takes in human visual perception upon an image to paint [10, 46]. An eye tracker records the eye movements of a user to know the content in an image. The collected perceptual data is used to determine stroke colors so that subjects in the image are painted in colors with more contrast and saturated fashion; otherwise, grayish unsaturated colors are used. A work of Gooch et al. segmented an image for medial axes estimation as mentioned above [14]. Lecot's work produces artistic vectorized images close to an input bitmap, using image segmentation to divide it into various regions [37].

Semet et al. built a distributed system using artificial ants [47]. Users specify various colonies, and then ants in the colonies navigate and sense environments of a reference image to deposit inkmarks on canvas.

2.2 NPR of 3D Models

Researches on painterly rendering have been advanced to 3D models. Kawata et al. developed an algorithm for a point set [28]. Kolliopoulos's painting system segments an image rendered from a 3D scene to abstract extra details in each object [33].

Generating an animated painting is not a simple extension from a image painting. Meier pointed out that fixing brush stroke positions for successive frames will produce so called 'shower-door' effect, meaning that strokes are always sticking onto a viewing place. Her work rather focused on making brush strokes always stick onto moving objects, using particle systems to maintain temporal coherency [41].

Kaplan et al. combined her work and Kowalski's work and proposed an interactive painting system capable of feathery rendering using geograftals [27, 35].

2.3 NPR of Videos

Litwinowicz demonstrated that how brush strokes can stick onto moving objects in a video sequence. He computed optical flow as a mean of motion estimation of the objects. At each frame brush strokes are displaced by optical flow in order to achieve temporal coherency. In his method, new strokes are randomized and then uniformly distributed among an old stroke set to avoid a clear boundary between those two sets [38]. Hertzmann et al. applied his image painting algorithm to video inputs. Paintings on successive frames are produced by painting over the previous frames, carefully drawing where on a canvas radical changes occur [24]. In a work of Hays et al., brush strokes are constrained over time and added new strokes at the end of draw order. New strokes are appearing from behind of existing strokes, and they are smoothly faded to avoid scintillation. Similar to Hertzmann's previous works, their system is capable of producing different artistic styles by using a different set of parameters [17]. Bousseau et al. used optical flow for texture warping to make abstracted videos in a watercolor style exhibiting high temporal coherency [4].

There are painting algorithms that are not relying on optical flow to achieve temporal coherency. Klein et al. treated a video input as volumetric data where time comes in as the third dimension. They defined 'rendering solids' to form

shards or short brush strokes when rendered. Since they extend over intervals of time, rendered videos maintain temporal coherency [32]. Agarwala et al. showed how roto-curves computed from a video source can be conformed to moving brush strokes [1]. Snavely et al. combined a video source with its depth information in order to produce a 3D motion flow that is spatio-temporally smooth [49]. A video tooning algorithm, a work by Wang et al., computes spatio-temporal 3D segmentation from a video source to help interpolate user-specified semantic regions [57].

2.4 Painting as Visualization

While painting is artistic representation of an image, it also provides a way to describe geometric structure embedded in the image. Scientific visualization has been developed to display scientific models in a meaningful way. For example, temperature distribution can be color-coded, or a vector field can be visualized by drawing trajectories or streamlines in evenly spaced fashion [39]. LIC achieved to visualize arbitrary 2D vector fields in a continuous smooth fashion [5]. At each point a noise texture is sampled along a trajectory in both forward and backward directions and convolved with a filter. The method was also demonstrated to process an image into a pictorial version. Later IBFV has been introduced by Wijk; it can emulate a wide variety of visualization technique, including LIC and arrow plots, in a high performance on machines with standard graphics features. Idea behind this technique is that within each iteration a current texture is warped along a vector field by some amount, and it is then combined with a noise texture

to create a new texture for the next iteration [55]. Zhang et al. developed a tensor field visualization method based on IBFV, which has inspired this thesis [59].

Some researchers have integrated artistic essence into scientific visualization in order to display data in more pleasing way. In non-photorealistic visualization technique of Healey, et al., multiple strokes are used to represent each element of a given data set. Those strokes' appearance depend on element's attributes [18]. Interrante has focused on visualizing multivariate data field, using a partially ordered multidimensional palette of natural textures [26]. Laidlaw attacked the same problem but instead used brush strokes, referred to as human-processed textures. He has put ideas after observations on Van Gogh paintings into visualization experiments [36].

Chapter 3 – System Overview

My system provides an ability of creating image/video paintings, where the style parameters and stroke orientations can be designed in both space and time. A still image or a video clip is acceptable as input. Users will be working with the user interface shown in Figure 3.1. A sub-window on upper-left is provided where the current painting can be displayed and some types of editing (described in short) are performed. Tabular panels on right organizes a set of editing operations.

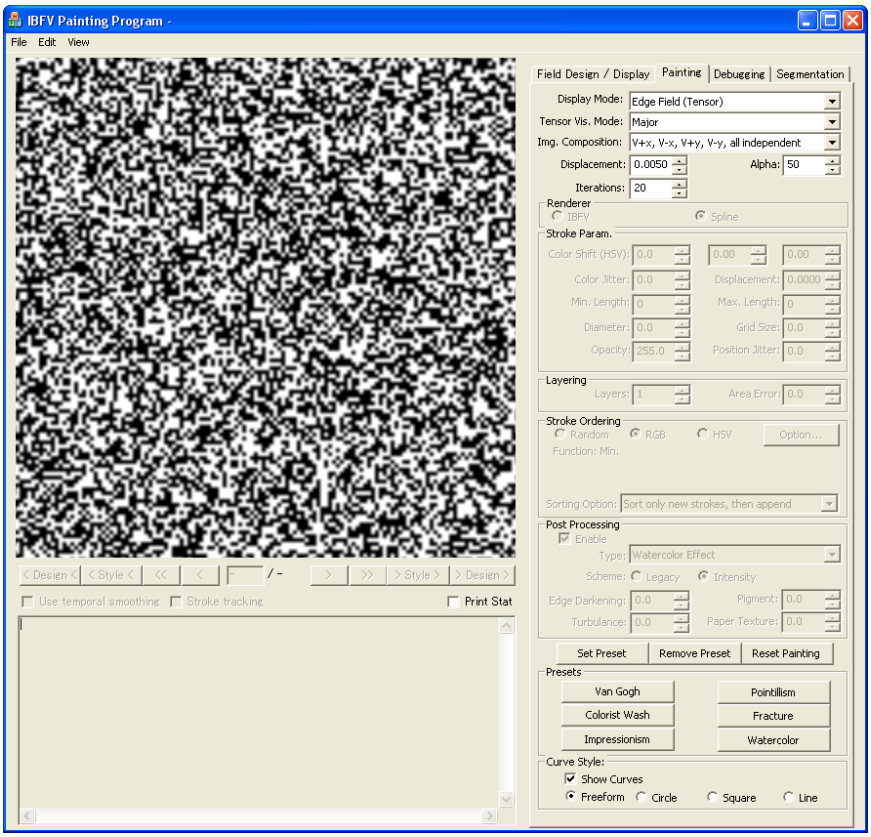


Figure 3.1: User interface of painting program

My painterly rendering framework is summarized as shown in Figure 3.2. With this framework, we are able to do the following items:

- Region creation
- Style parameter design
- Edge field design
- Stroke ordering
- Rendering

In this chapter we are stepping through each item, describing capabilities and ideas of my system. Possible workflows are discussed afterwards. For the rest of this thesis, we first discuss the design of style parameters and stroke orientations in Chapter 4 before describing a new rendering algorithm for both images and videos in Chapter 5.

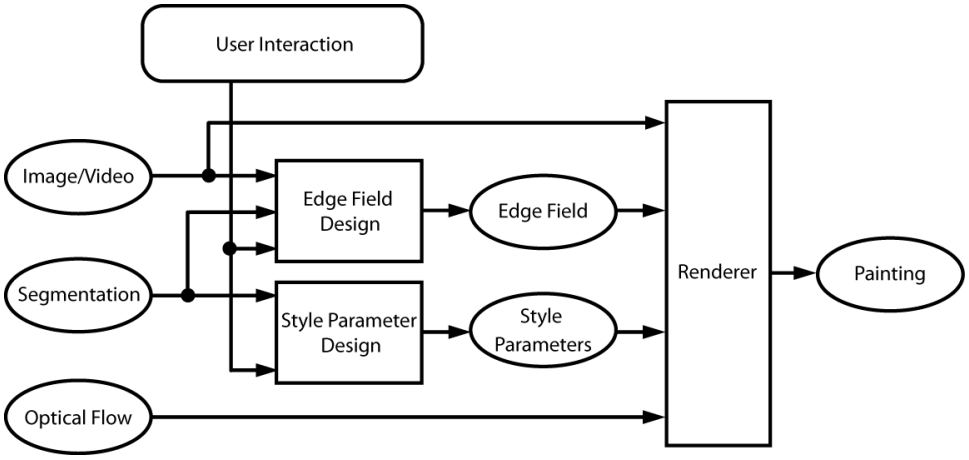


Figure 3.2: Block diagram of multi-style rendering system

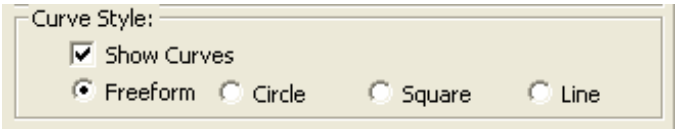
3.1 Region Creation

In my system, a region is a subset of a rectangular canvas, which represents an object in a scene or background. In other words, a canvas is made up of disjoint regions each containing pixels. Region creation takes an important role of achieving spatially varying style paintings. To do this, users need to supply a segmentation file for an input. For image input, it is possible to create such data over a view window in my system. One way is to manually draw closed curves, each bisecting a region into smaller ones. A better way is use an automatic segmentation tool with computer vision technique and user interaction for refinement (Figure 3.3). For video inputs, a video segmentation tool is used as an offline process to create a set of segmentation files for each frame. We are not going to step in any detail about segmentation algorithm since it is outside of my achievement for this thesis.

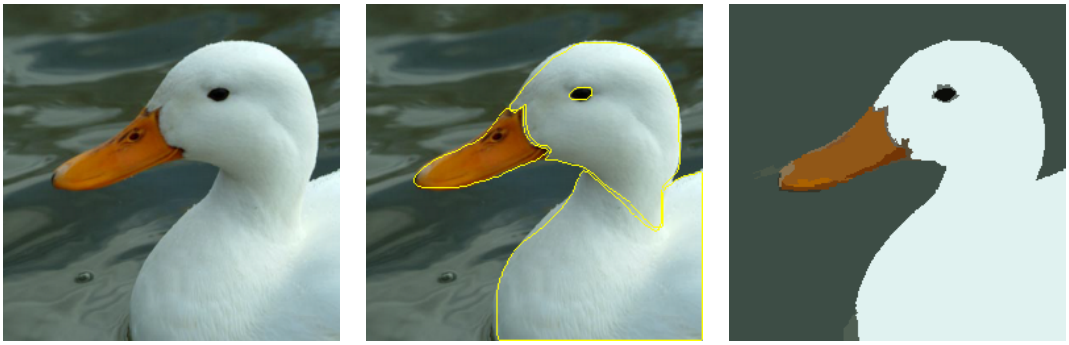
For my system to know which region will be modified through style parameter or edge field design, the region must be clicked to select. Regions not selected will be dehighlighted. Clicking on a selected region will deselect it so the view window shows all regions highlighted.

3.2 Style Parameter Design

For artists to start painting, they have to decide a number of attributes for a style such as brush size, length, color choice, and so on. It is the same for my framework as well. At least one style definition needs to be specified. We call the definition 'style parameters'. It is merely a collection of numerical values each



(a)



(b) (c) (d)

Figure 3.3: An example of region creation. (a): Hand drawing tool for image segmentation. (b): Input image. (c): Segmentation using the hand drawing tool. (d): Segmentation using computer vision algorithm. Original image from FreeFoto.com.

defining painting attributes such as stroke attributes, multi-layer settings, and post processing.

There are a number of stroke attributes (See Figure 3.4). In my system, we use a term 'stroke size' for width (a). Both width and length can change style appearance. Some styles such as Impressionism tend to have long strokes whereas Pointillism has zero-length strokes, or dots (b). Density is how strokes should be close to each other. If density is too low then they can leave many gaps uncovered (c). Opacity decides the transparency of a stroke. Giving lower opacity will look wash-style paintings (d). Strokes are usually placed in the center of regular grids, which makes artificial looking. Thus a control to jitter those grids are also given

to reduce it (e). It is possible for users to jitter stroke colors (f) as well as shift them by some amount (g∼i). Note that these types of techniques such as using jittered grid and stroke colors have already been suggested in several painting papers [15, 17, 19, 38].

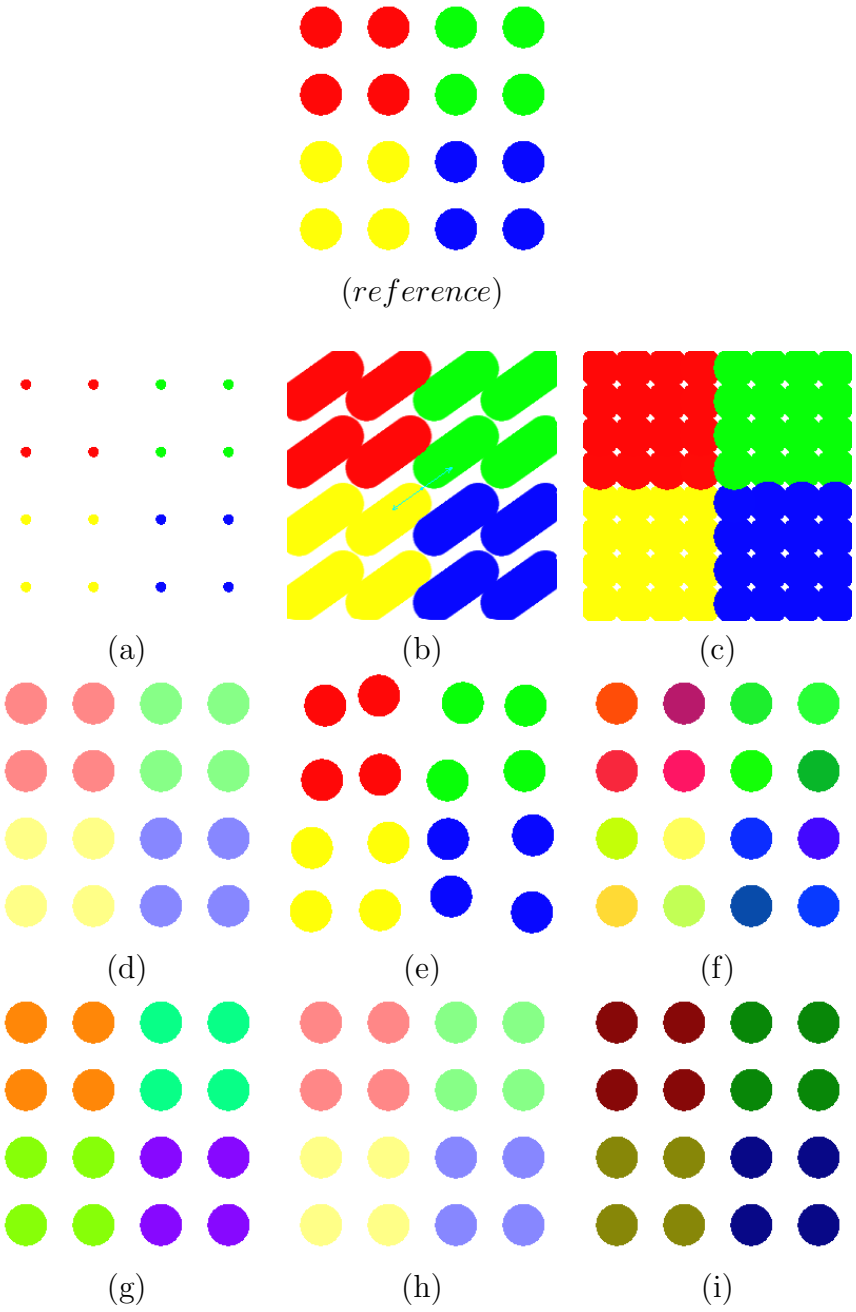
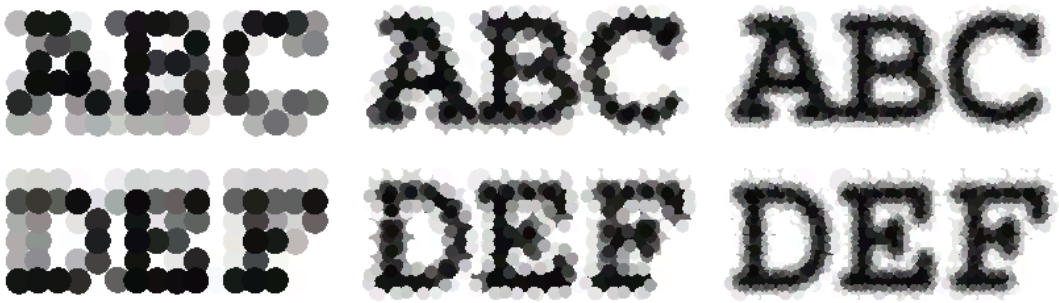


Figure 3.4: Stroke attributes include (a) size, (b) length, (c) density, (d) opacity, (e) position jitter, (f) color jitter, and (g∼i) color change in HSV values.

Multi-layer option introduces additional layers over the underlying painting and fix areas where the result deviates from the original input over some threshold. Two parameters are given – the number of layers and error tolerance. Each upper layer uses a smaller stroke size. This means that it can add more details that have not captured in lower layers (Figure 3.5). How much details should be added depends on the error tolerance. If it does not allow any error, then upper layer could hide whatever is drawn onto the underlying painting. The idea of using multi-layers is originally from Hertzmann [19].

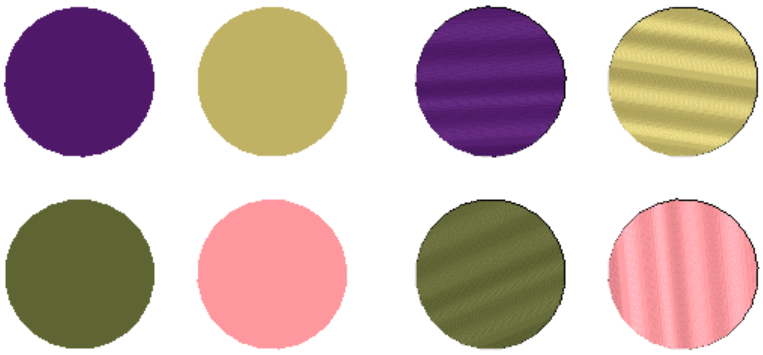


(a) (b) (c)

Figure 3.5: Multi-layer example. Smaller strokes are added to upper layers, giving finer details. Images above are rendered with different number of layers. (a): 1 layer, (b): 2 layers, and (c): 3 layers.

Post processing gives a lighting effect on color paintings to give more realistic appearance. This includes light angle, stroke texture scale and roughness, and stroke height (Figure 3.6). This is the idea from Hertzmann [21].

As users can expect, there are many parameters to make up one single style. Those not familiar with my framework can apply one of style presets which it



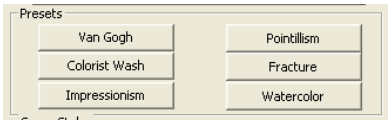
(a) (b)

Figure 3.6: Post processing example. (a): a color image without lighting. (b): the same image with lighting applied.

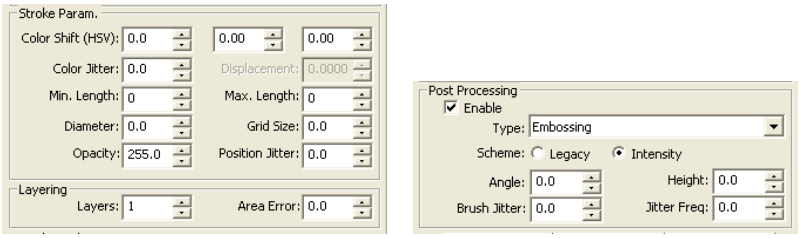
provides. If they are not satisfied, specific parameters can be controlled from text boxes (Figure 3.7).

Style design is performed in region basis, but it does not mean that each single region must have its own style. If there is some region without any style assigned to it, my system automatically computes a style for the region by using 'style propagation'. It is similar to the idea of heat diffusion. Any region to which some style is assigned is considered as a heat source. Heat will propagate through entire region until heat distribution is stabilized. Thus style within unassigned regions seems to become style transition among those heat source (Figure 3.8).

Another issue about style parameters is that they can be keyframed. Creating a painting video with style transition over time can be achieved by assigning multiple style parameters to different frames. Any style between two keyframes can be derived from interpolation over the keyframes.



(a)



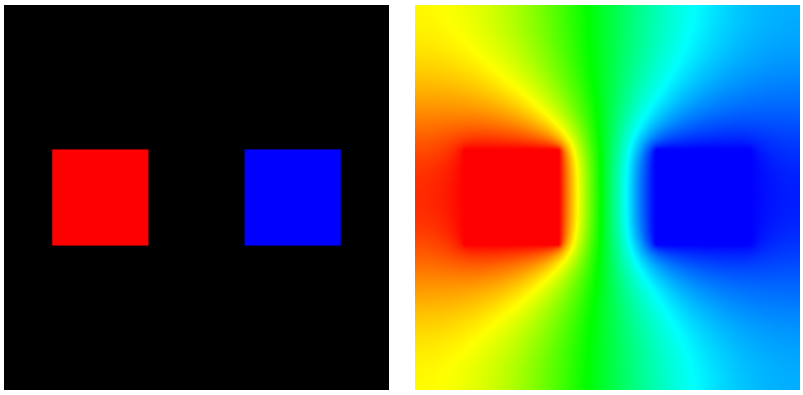
(b)

Figure 3.7: User interface of style parameter design. (a) Style can be applied from one of presets, and (b) can be further modified through edit boxes.

Example: We show a sequence of style design process. See Figure 3.9 for visual guidance. (a) Suppose an input image with a corresponding segmentation data dividing the input into three regions. (b) To apply a style, we select a region and then chooses one of presets (Figure 3.7(a)). (c) A style has been applied. Note that styles are observed even outside the selected region. This is due to the aforementioned style propagation through heat diffusion. (d∼f) We repeat this for other regions. In this example, a style for each region is specified, in order of petals, background, and the center of a flower.

3.3 Edge Field Design

We use the edge field which dictates stroke orientation over input domain, and it is typically computed from an input source, using the tensor field design frame-



(a) (b)

Figure 3.8: Spatial interpolation. (a): Heat sources are shown as color squares (red and blue) and the rest of areas are empty (black). (b): The empty area is filled with values from surrounding sources.

work [59]. However, one might sometimes think it is necessary to fix the orientation in some area or to add special effects such as ripples on water where a dolphin comes out. To do this, we rely on a design tool developed by Zhang et al. [59]. In their approach, a group of special units called 'design elements' are used to guide stroke orientations. Stroke orientations can be locally modified by using them. All type of design elements and operations can be selected from user interface as shown in Figure 3.10. Users will be using a mouse to add, modify or delete elements over a selected region on a view window.

Like style parameter design, edge field design can be performed per region. Regions with specified design elements will propagate its edge field outwards. Design elements are treated as keyframeable objects as well as style parameters. Elements specified at any frame also affects all other frames. Objects specified at different keyframes are interpolated to make a smooth transition from one to another.

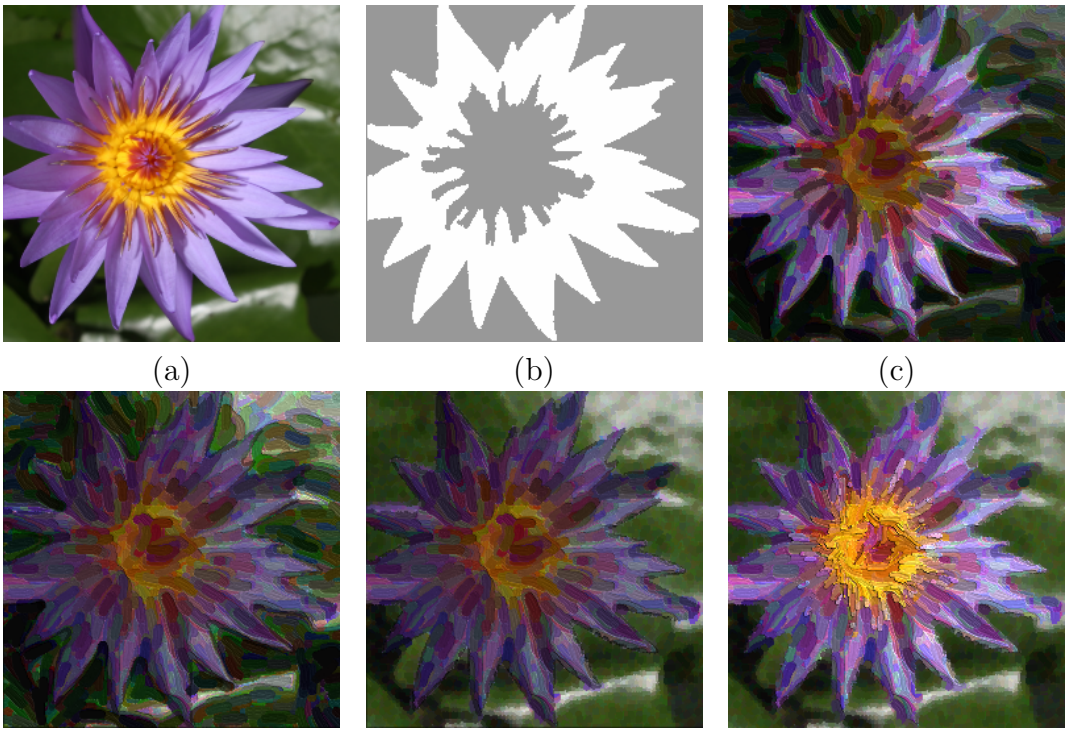


Figure 3.9: An example of style design process. Original video courtesy of Artbeats.

Example: We show a sequence of edge design process. See Figure 3.11 for visual guidance. (a) Suppose we have an input image. (b) The system computes an edge field from the input. (c) Orientation can be fixed by putting a design element. Each element is visualized as either a color box or an arrow. (d) We repeat this process until we fully modify orientation for other regions. (e) The modified edge field is then used in a renderer to produce a painting.

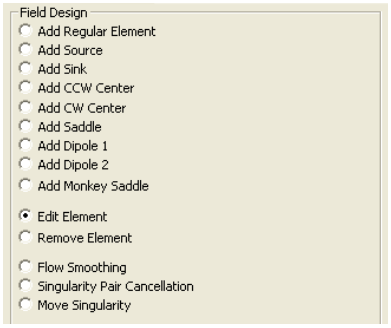


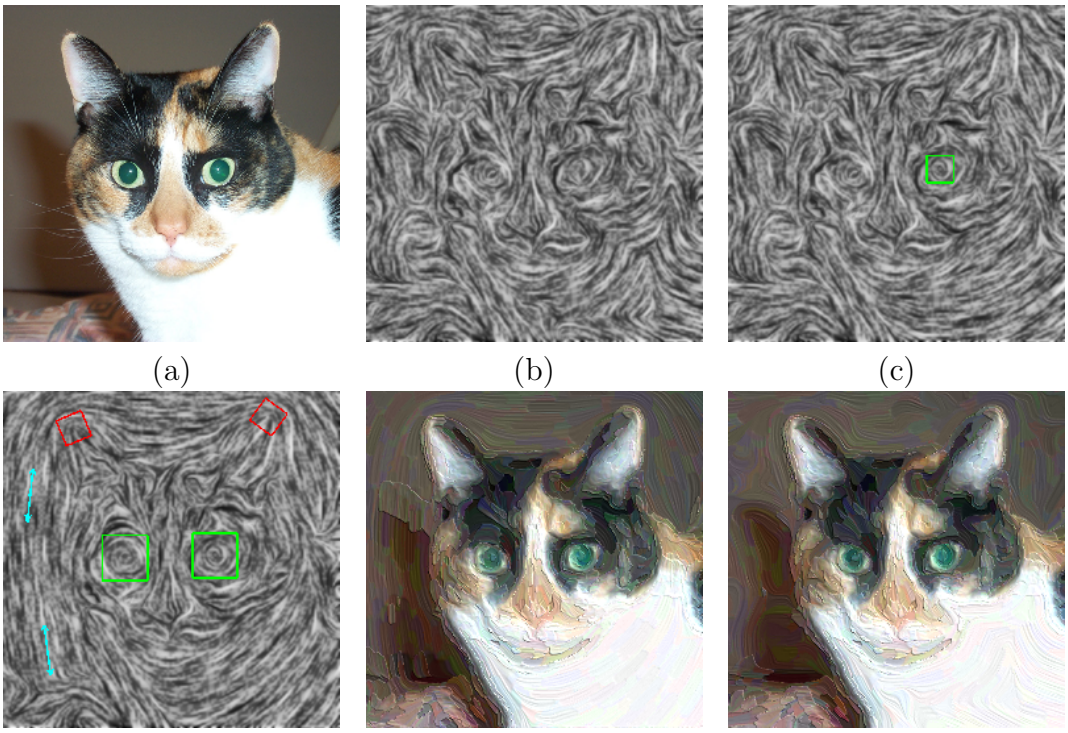
Figure 3.10: Edge field design pane. Various types of design elements and edit operations are accessible from this pane.

3.4 Stroke Ordering

Stroke ordering determines draw order of all strokes that are generated based on all style parameter settings. This setting is accessible from a stroke ordering pane (Figure 3.11), and users can choose one from the following three options:

- *Random*: Strokes are sorted in ascending order based on random numbers assigned to each stroke.
- *RGB*: Strokes are sorted according to an ordering function based on RGB channels
- *HSV*: Strokes are sorted according to an ordering function based on their HSV values.

A function for a color-based ordering is customized via an option dialog (Figure 3.12). By default, ordering applies to newly spawned strokes. However, it can extend to a whole set of strokes as well.



(a) (b) (c)

(d) (e) (f)

Figure 3.11: An example of edge field design. (a)∼(d): A sequence of the design process. (e): Result. (f): Another painting with no design elements added (for a comparison with (e)). Original image courtesy of Greg Turk.

3.5 Rendering

Two different renderers have been implemented for my system and can be toggled before painting production (Figure 3.13(a)). The first one is a geometry-based (explicit) renderer, where spline curves represent long curved strokes. The work has already been introduced in a multi-layer image painting paper authored by Hertzmann [19]. My version of an explicit renderer deals with video inputs, and we also borrowed some ideas from Hays's video painting paper [17]. For convenience,

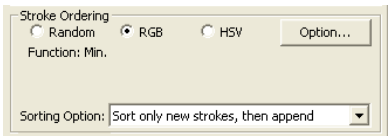
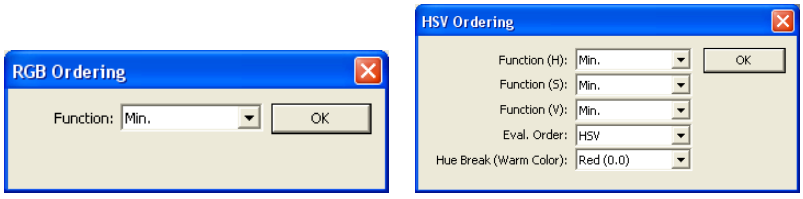


Figure 3.11: Stroke ordering pane.



(a) (b)

Figure 3.12: Customization dialog for color functions. (a): RGB. (b): HSV.

we use a term 'explicit' for this approach.

The other renderer is an image-based renderer, inspired from Image Based Flow Visualization [55], which has been a primary subject of my thesis. The aim was to simultaneously render a set of strokes, instead of placing strokes one after another like in an explicit method. It turned out that brush strokes produced from this looked more natural than spline curves. As opposed to the former approach, we call this 'implicit'.

My renderers are capable of creating two different images based on given style parameters. One is a color painting, and the other is a height image that is used at a post processing stage.

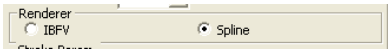


Figure 3.13: Renderer pane. 'IBFV' refers to an implicit method whereas 'Spline' is an explicit method.

3.6 Workflow

Considering all aforementioned items, we describe some case scenarios using my framework. Depending on which case to go with, users need to pass in additional data along with an input image/video.

3.6.1 Single Style

The simplest case is a single style painting, which consists only two stages as shown in (Figure 3.14). Since only one style dominates in both time and space domains, there is no need to process segmentation and only one keyframe suffices. Output is either an image or a video sequence where a single style dominates from the beginning to the end.

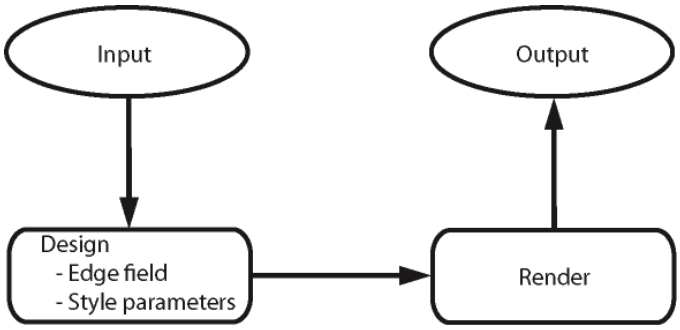


Figure 3.14: Single style workflow.

3.6.2 Spatial Multi-Style

For spatially varying paintings, segmentation data needs to be supplied as an additional input along with an image. It is still possible to create a segmentation data on my system, instead of not passing it as an external file. In this case, the segmentation pattern applies to all frames. Design process is iterative since there are multiple regions. Figure 3.15 shows workflow for this case.

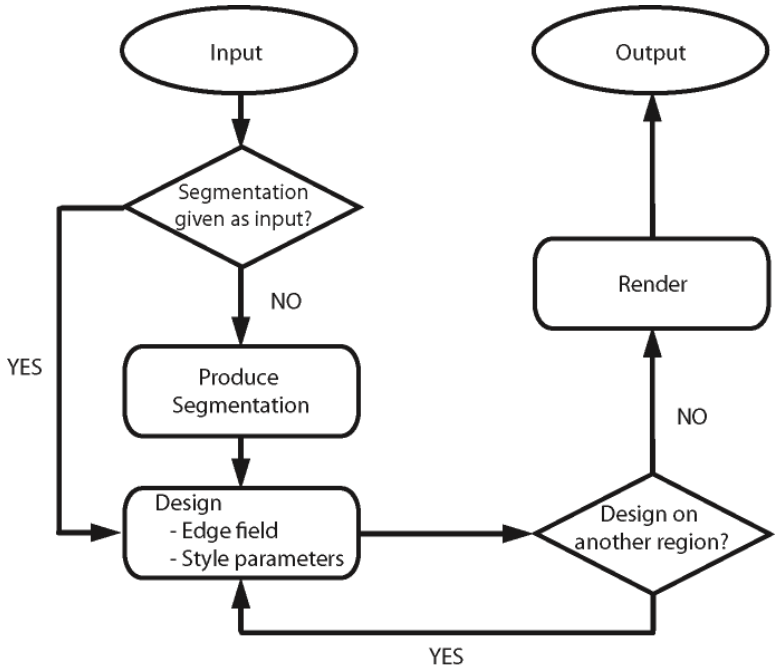


Figure 3.15: Spatial multi-style workflow.

3.6.3 Temporal Multi-Style

For this type of painting, input is typically a video sequence. However, it is still possible to use a still image, treating it as a video sequence with the same image content at each frame. Design process is again iterative but it resides in temporal domain (Figure3.16). For a video input, motion data of objects in the input also needs to be supplied. This data will be used to move brush strokes at each frame to follow corresponding objects.

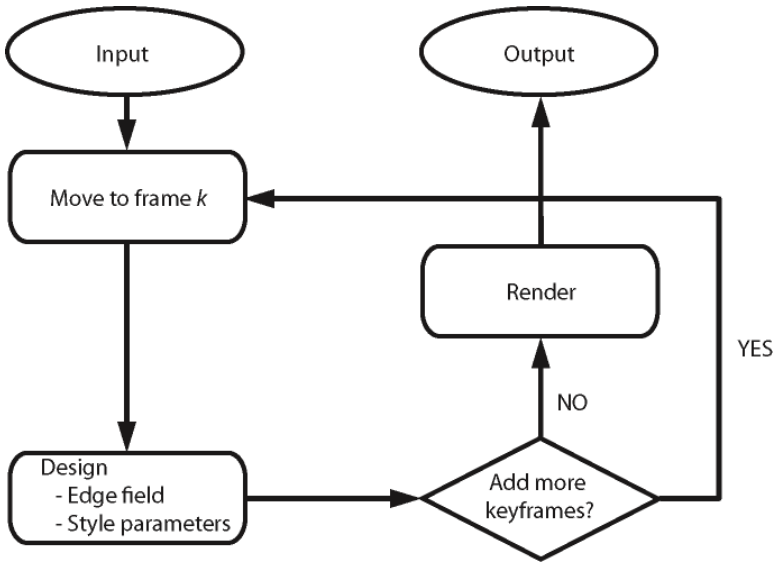


Figure 3.16: Temporal multi-style workflow.

3.6.4 Spatial and Temporal Multi-Style

Combining subsections 3.6.2 and 3.6.3, we obtain a complete workflow for space-time varying multi-style workflow as shown in Figure 3.17. Note that it also inherits extra input requirements.

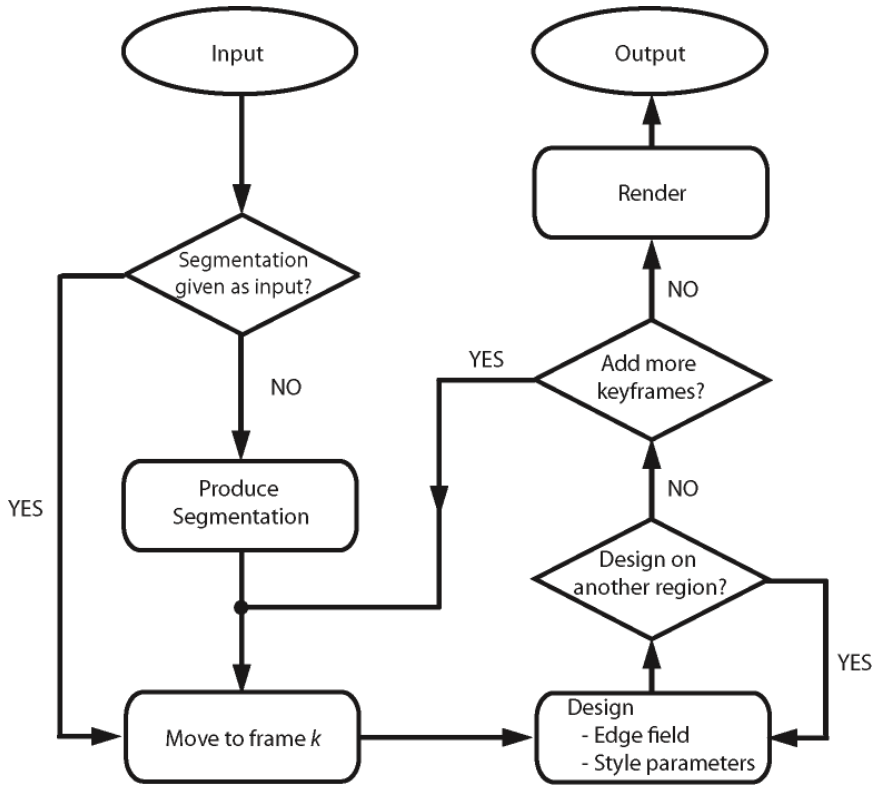


Figure 3.17: Spatial and temporal multi-style workflow.

Chapter 4 – Style Specification

Recall from Chapter 3 that spatially varying paintings can be created through the idea of keyframing. The technique is widely used in animation where several 'key' poses are assigned to objects at particular frames so that motion at the rest of frames are synthesized [1, 51]. The similar idea is used here. Two different environments for keyframing in my painting system are style and an edge field. Style parameters are used as keyframeable objects for the former whereas design elements play the same role for the latter.

Users might want to assign style parameters to every region. However, it can become a tedious work if there are tens of regions, or users may want to have an effect of style transition from one place to another. My system addresses this by propagating style parameters from assigned regions to regions without specified style parameters. This propagation technique is also used in edge field design.

For style propagation, the idea of heat diffusion is employed. An environment is initialized with a zero temperature, to which multiple heat/cold sources representing one of the style parameters are put into various locations in the environment. Once stabilized, the distribution of the parameters covers the whole domain which matches the values at assigned regions. There are several numbers of numerically stable methods to simulate it, such as the Gauss-Seidel method or the Conjugate Gradient method [42]. For this thesis, the Gauss-Seidel approach is used [50]. Any region with keyframes is marked as 'fixed' so that its value will not be overridden from the propagation. A direct approach is to solve a heat equation in a 3D space where the time comes in as the third coordinate:

$$\frac{\partial u}{\partial t} = \kappa \nabla^2 u = \kappa \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \tag{4.1}$$

, where κ is some constant and u is a scalar function defined on volume.

However, this volumetric diffusion requires much space since each pixel needs to have values of both style parameters and edge fields. Consider a case of propagating style parameters for a video sequence with a dimension 512×512 pixels^2 and 100 frames. 17 values of type either an integer or a floating point number make up a single style. Assume each value takes up 4 bytes of memory. Thus style parameters at each pixel requires 68 bytes. This follows that there must be 17 megabytes of data to store for each frame, which eventually leads to approximately 1.7 gigabytes to save for all frames. This number proportionally increases as a video input has more number of frames. Direct implementation of a stable diffusion method results in setting up a huge linear system, which also requires additional space to save updated style parameters. Another issue is that the propagation must be carried out every time a keyframe is updated. My system needs to be responsible for giving feedback at promising interactive rate.

To avoid such computational and memory cost, a two-step approach is taken which divides the three-dimensional problem into temporal and spatial subproblems. Given a frame number k, we first compute style parameters or the edge field of every region where corresponding keyframes are defined. In the second step, those computed values are propagated through the rest of regions at that frame. Thus, that propagation turns into two-dimensional heat diffusion, which requires

less memory space. We can also expect that reasonable computation time since area necessary for diffusion is typically smaller than the whole canvas.

For the rest of this chapter, we show details of propagation algorithms for both style parameters and an edge field, respectively.

4.1 Style Parameters

As mentioned before, style parameters are just a collection of numeric values. Thus, applying heat diffusion technique is straightforward, using Equation 4.1 for each value. However, there are two issues to note. First, some of style parameters are of integer type, such as the number of layers used in rendering. However, they are more discretized than floating point numbers so that using a tiny step size has no effect to diffuse them. Such values must be temporarily treated as floating point numbers before entering a diffusion step and snapped back to the closest integer afterwards. Second, for some algorithmical reason on my renderers, we fix a value of stroke density for multi-style painting for both space and time domains. Once the stroke density in one keyframe is changed, it affects other keyframes. Thus all style parameters except the density will be propagated. Chapter 5 lists attributes of the parameters and describes where they are used to render a painting.

4.1.1 Approach

Given a frame number k_0, a style applied to a canvas needs to be computed. The pseudocode is given as follows.

```
computeStyle(R = {R_0, R_1, ...R_n}, k_0) {
  for (each region R_i in R) {
    if (R_i has no keyframe for style parameters) {
      // Style will be computed in diffuse step.
      updateDiffusionFlag(true, R_i)
    }
    else {
      if (R_i has a keyframe at k_0) {
        S_0 ← style parameters at k_0
      }
      else {
        S_n ← style parameters at the closest keyframe after k_0
        S_p ← style parameters at the closest keyframe before k_0

        if (S_n exists and S_p exists) {
          n_0 ← frame number of S_n
          p_0 ← frame number of S_p
          ω ← (n_0 − k_0)/(n_0 − p_0)

          S_0 ← ωS_p + (1 − ω)S_n        // Interpolate over two keyframes.
        }
        else if (S_n exists) {
          S_0 ← S_n        // Use from a forward keyframe.
        }
        else {
          S_0 ← S_p        // Use from a backward keyframe.
        }
      }

      // Update a painting style of R_i.
      udpateStyle(S_0, R_i)
```

```
        // Styles of pixels in R_i will not be modified
        updateDiffusionFlag(false, R_i)
      }
    }

    // Propagate styles to regions marked as true.
    diffuseStyle(R)

    return
}
```

This operation takes a region set R, which are segmentations of a whole canvas. It checks each region in R to see whether it has any keyframe assigned for the region; if not, a 'true' flag is set by updateDiffusionFlag() function, showing that style parameters for the region will be computed later. Otherwise, a 'false' flag is set and styles are computed from keyframes. If there is a keyframe at a specified frame, then values of the frame will be used. Otherwise, it looks for the two closest keyframes: one from forward and the other from backward. If both are found, styles are computed by linearly interpolating over them. If not, that means only one keyframe is found. Thus style parameters from the keyframe are used.

Once all styles for every region with keyframes are identified, styles for rest of regions (i.e., those marked as 'true') will be computed from two dimensional heat diffusion (equivalent to diffuseStyle() function in pseudocode). Thus Equation 4.1 reduces to the two-dimensional version:

$$\frac{\partial u}{\partial t} = \kappa \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{4.2}$$

To compute this, a canvas is discretized into a two-dimensional grid, where each cell corresponds to a pixel. We denote values in a cell as $U^n_{i,j}$, meaning the values after n-th time step of a cell at the i-th row and the j-th column. With this notation, heat diffusion can be visualized as shown in Figure 4.1. At each moment, the center cell $U^n_{i,j}$ diffuses outward so a fraction of a value it stores is given to neighboring cells. On the other hand, those cells also give some fraction of values to the center cell. Therefore, values at each cell can be computed as a simple explicit scheme:

$$U^{n+1}_{i,j} = U^n_{i,j} + \left(U^n_{i,j+1} + U^n_{i,j-1} + U^n_{i+1,j} + U^n_{i-1,j} - 4U^n_{i,j}\right)\Delta t \tag{4.3}$$

, where Δt is step size. However, this method is numerically unstable against a larger step size. A better approach is to apply an implicit scheme shown as Equation 4.4. The Gauss-Seidel method is used to solve a linear system of such equations for each cell, which is relatively easy to implement and stable against arbitrary step size although additional iterations are required in order to values converge under some threshold.

$$U^{n+1}_{i,j} = \frac{1}{1+4\Delta t}\left\{U^n_{i,j} + \Delta t\left(U^{n+1}_{i,j+1} + U^{n+1}_{i,j-1} + U^{n+1}_{i+1,j} + U^{n+1}_{i-1,j}\right)\right\} \tag{4.4}$$

Even though the Gauss-Seidel method is stable against large step size, it is not fast due to many iterations if we want a grid such that style parameters are completely diffused over it. To reduce such iteration amount, we beforehand assign an initial guess of style parameters to each cell, using a push-pull method.

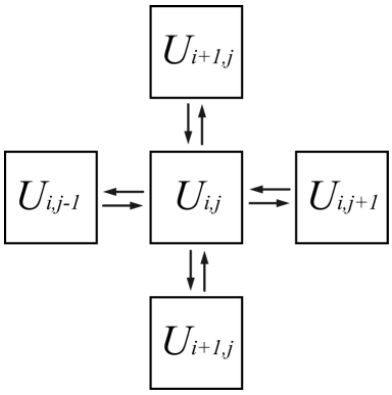


Figure 4.1: 2D heat diffusion in a discrete case. Adjacent cells exchange values by a small amount.

A push-pull method is a pyramid method used for image processing such as interpolation over scattered pixel data to fill missing pixels. The method we used is described in [52]. The idea is to build a pyramidal structure, where the original data sits in the bottom. In the push process, low resolution of a grid is constructed at each level such that each cell is computed by averaging known cells. In the pull process, undefined cells are determined from its upper level. Our method involves heat diffusion at each level before filling values. This means diffusion flags need be assigned to upper levels. If there is no known cell to average, then a corresponding pixel at upper level will have a 'true' flag; otherwise a 'false.' Once initial guess values are computed, heat diffusion is applied for the final style parameter distribution for a canvas.

4.2 Edge Field and Design Elements

To the best of my knowledge, there are two ways to represent an edge field. One way is to represent each orientation in terms of a vector. Another approach is encode it as a tensor as shown in [59]. Let me spare some space to briefly review what a tensor is. A tensor is a way to represent a quantity, like a scalar or a vector. With a simple example, a second-order tensor is represented as follows.

$$\mathbf{T} = \begin{pmatrix} t_{00} & t_{01} \\ t_{10} & t_{11} \end{pmatrix} \tag{4.5}$$

Notice that it is actually a 2-by-2 matrix. It is symmetric if $t_{01} = t_{10}$. It can be decomposed into a unique combination of an isotropic part $\mathbf{S}$ and an anisotropic part $\mathbf{A}$ such as

$$\mathbf{T} = \mathbf{S} + \mathbf{A} = \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \mu \begin{pmatrix} \cos 2\theta & \sin 2\theta \\ \sin 2\theta & -\cos 2\theta \end{pmatrix} \tag{4.6}$$

, where $\mu \geq 0$. It typically has two orthogonal eigenvectors equal to the ones of $\mathbf{A}$. Unlike a vector which shows one direction, eigenvectors of a tensor are bidirectional. The major eigenvector v_0 is written as

$$v_0 = \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} \tag{4.7}$$

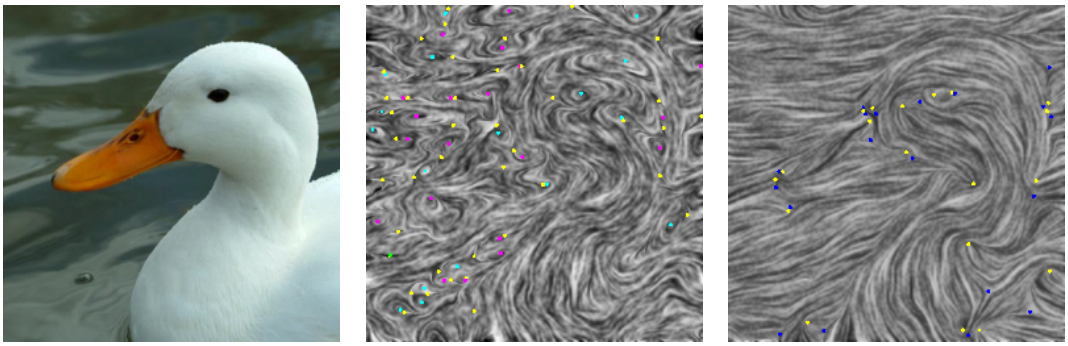
, and its eigenvalue is equal to μ. On the other hand, the minor eigenvector v_1 looks

$$v_1 = \begin{pmatrix} \cos(\theta - \pi/2) \\ \sin(\theta - \pi/2) \end{pmatrix} \tag{4.8}$$

with its eigenvalue $-\mu$. Analogous to a vector field, there is also a tensor field where at each point a tensor value is defined. Any point with two equal eigenvalues is called a degenerate point, or a singularity. Singularities are important aspect of the tensor field topology, and there are several types of such points. Some of them are described as design elements, which will be introduced shortly.

The edge field in my system is the second-order symmetric tensor field, and consists of only the anisotropic part. Namely, $\mathbf{T} = \mathbf{A}$. Its eigenvectors, v_0 and v_1, correspond to stroke orientation and gradient direction, respectively. Tensor is considered as a good choice of representing edge information or modifying it since propagating or smoothing a tensor field is less likely to introduce additional singularities than vector field propagation [59]. Figure 4.2 describes an example for this.

Computing our edge field not only involves propagation but also tracking of design elements. Those elements need to follow assigned regions. For instance, if a region moves, all of associated design elements must follow it. If it rotates, they should also move such that they stay in the same position with the same orientation relative to the region (see Figure 4.3 for detail).



(a) (b) (c)

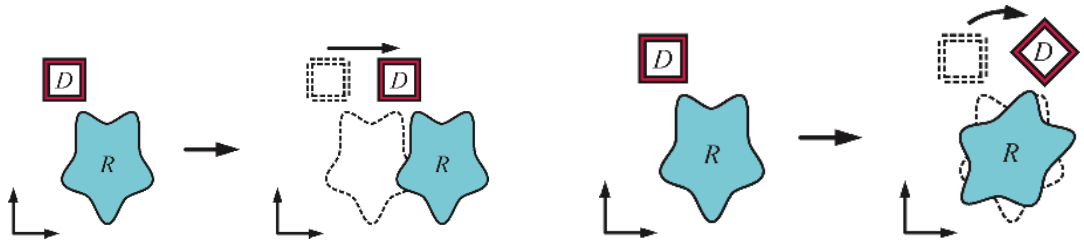
Figure 4.2: An example of smoothing on vector/tensor-based edge fields. (a): Original image. (b) and (c): Vector/Tensor-based edge fields after smoothing, respectively. Color dots show different types of singularities. Notice that the vector-based edge field contains more number of singularities than the other.

4.2.1 Design Element Types

Design elements locally modify the underlying edge field. In other words, they add either a degenerate point or a regular flow. Such elements doing the former are called 'singular elements' and elements of the latter kind are called 'regular elements' [59]. We now introduce all types of design elements with shapes and mathematical forms. Figure 4.4 shows screenshots of each type.

4.2.1.1 Wedge

It is one of primitive singularities in a tensor field. Given a point at $\mathbf{p}_0 = (x_0, y_0)$, the wedge pattern at $\mathbf{p} = (x_\mathbf{p}, y_\mathbf{p})$ is described as follows.



(a) (b)

Figure 4.3: Design elements must follow motion of assigned regions in terms of (a) translation and (b) rotation. In this example, a square with a label D corresponds to a design element and a blue area with a label R is a region.

$$\mathbf{T}(\mathbf{p}) = e^{-d\|\mathbf{p}-\mathbf{p}_0\|^2} \begin{pmatrix} x & y \\ y & -x \end{pmatrix} \tag{4.9}$$

, where d is a decay factor and $(x, y) = (x_{\mathbf{p}} - x_0, y_{\mathbf{p}} - y_0)$. This element will be useful to enhance edges of objects with sharp ends.

4.2.1.2 Trisector

It is another primitive singularity, which divides a plane into three hyperbolic sectors.

$$\mathbf{T}(\mathbf{p}) = e^{-d\|\mathbf{p}-\mathbf{p}_0\|^2} \begin{pmatrix} x & -y \\ -y & -x \end{pmatrix} \tag{4.10}$$

4.2.1.3 Node

A node pattern resembles sink/source patterns in a vector field. This would be useful to add explosion effects.

$$\mathbf{T}(\mathbf{p}) = e^{-d\|\mathbf{p}-\mathbf{p}_0\|^2} \begin{pmatrix} x^2 - y^2 & 2xy \\ 2xy & -(y^2 - x^2) \end{pmatrix} \tag{4.11}$$

4.2.1.4 Center

Ring patterns exhibit within a center. This element is useful to enhance an edge field of circular objects such as the moon and eyes.

$$\mathbf{T}(\mathbf{p}) = e^{-d\|\mathbf{p}-\mathbf{p}_0\|^2} \begin{pmatrix} y^2 - x^2 & -2xy \\ -2xy & -(y^2 - x^2) \end{pmatrix} \tag{4.12}$$

4.2.1.5 Saddle

A saddle is made up of four hyperbolic sectors. Due to its distinctive pattern, usage might be limited.

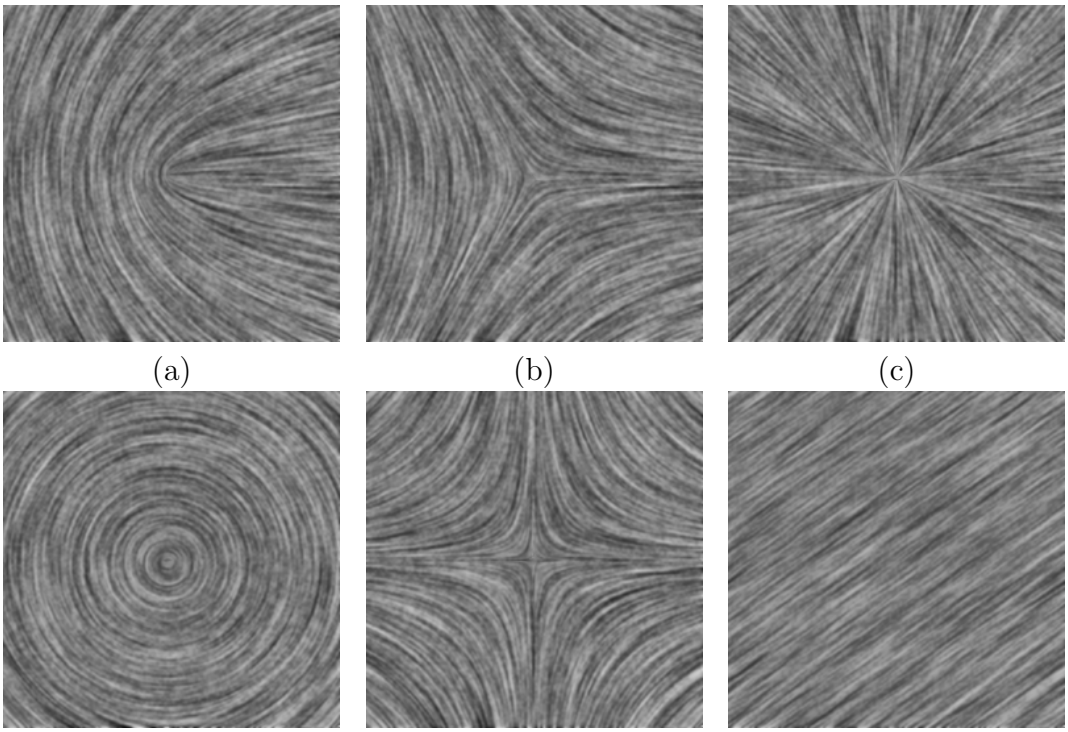
$$\mathbf{T}(\mathbf{p}) = e^{-d\|\mathbf{p}-\mathbf{p}_0\|^2} \begin{pmatrix} x^2 - y^2 & -2xy \\ -2xy & -(y^2 - x^2) \end{pmatrix} \tag{4.13}$$

4.2.1.6 Regular

A regular element adds a straight orientation to a tensor field. It is different from aforementioned items in that it does not contain a degenerate point. Given a direction (V_x, V_y) defined at $\mathbf{p}_0$,

$$\mathbf{T}(\mathbf{p}) = e^{-d\|\mathbf{p}-\mathbf{p}_0\|^2}\mu \begin{pmatrix} \cos\theta & \sin\theta \\ \sin\theta & -\cos\theta \end{pmatrix} \tag{4.14}$$

, where $\theta = 2\tan^{-1}\left(\frac{V_y}{V_x}\right)$, and $\mu = \sqrt{V_x^2 + V_y^2}$. This element might be useful to sweep out complex patterns on background.



(a) (b) (c)

(d) (e) (f)

Figure 4.4: Design element types. (a): Wedge. (b): Trisector. (c): Node. (d): Center. (e): Saddle. (f): Regular.

4.2.2 Edge Field Computation

To the best of my knowledge, there are two ways to compute an edge field – local and RBF. With a local method, an image gradient at each pixel is estimated using the Sobel filter. A gradient field is a vector field describing in which direction change in intensity is the greatest. Note that its perpendicular direction shows a direction with the least intensity change, which corresponds to edge orientation. Thus each vector in the gradient field needs to be rotated by 90 degrees. This approach is used in works of Litwinowicz and Hertzmann [19, 38]. On the other hand, Hays believed that using only strong edges for a whole domain would produce a more coherent result. Regions with weak edges are updated by using a radial basis function (RBF) which propagates strong edges into weak areas [17].

We use the latter approach but it needs to be accustomed to a tensor field version. To begin with, a gradient field is estimated by using the Sobel filter with a kernel size 3x3 pixels^2. We take the rotated gradient field as the major eigenvectors of our edge field. In summary, given a gradient $< G_x, G_y >$, we compute a tensor $\mathbf{T}$:

$$\mu = \sqrt{G_x^2 + G_y^2} \tag{4.15}$$

$$\theta = 2\left(\text{atan2}(G_y, G_x) + \pi/2\right) \tag{4.16}$$

$$\mathbf{T} = \mu \begin{pmatrix} \cos\theta & \sin\theta \\ \sin\theta & -\cos\theta \end{pmatrix} \tag{4.17}$$

Then we use a discrete Laplacian operator to propagate strong edges:

$$T_0^{n+1} = T_0^n + \sum_{j \in J} \left(T_j^n - T_0^n \right) \Delta t \tag{4.18}$$

, where J is a set of neighboring pixels. We want to modify only pixels with small magnitude so pixels with strong values are fixed to prevent the smoothing operation from modifying values.

This method considers only a single frame, rather than taking neighborhoods into account. It makes edge fields of a video input discontinuous over time. This will give visual artifact in which every stroke shakes instead of rotating in a pleasing way. To remedy this, we smooth out the edge field in both spatial and temporal domains. Given a frame number k, we compute edge fields from 5 contiguous frames: $(k-2)$, $(k-1)$, k, $(k+1)$, and $(k+2)$. An edge field at the frame is then calculated by smoothing them with a 3D Gaussian filter with a kernel size $5 \times 5 \times 5$ pixels^3.

Another option would be to apply angular constraints, which originally comes from Hays and Essa [17]. Their idea is to set angular constraints to strokes so that they will not rotate more than a specified threshold. We instead modified our edge field so that at each pixel tensor is recomputed if angular difference of eigenvectors between consecutive two frames is beyond a threshold. We have experimented with various thresholds but found that it was not working better than just smoothing out the edge field.

4.2.3 Approach

Pseudocode for computing an edge field has a similar structure to the one computing style parameters. Thus the code looks as follows.

```
computeEdgeField(R = {R_0, R_1, ...R_n}, k_0) {
  for (each region R_i in R) {
    R_i.edgefield ← the edge field from input frame at k_0 or the zero field

    if (R_i has no keyframe for design elements) {
      updateDiffusionFlag(true, R_i)
      continue
    }
    else {
      D_k ← R_i.key_elements[k_0]

      // At keyframe
      if (D_k.size() > 0) {
        for (each design element D_j in D)
          applyElement(D_j, R_i)

        updateDiffusionFlag(false, R_i)
      }
      else {
        D_f ← R_i.forward_elements[k_0]
        D_b ← R_i.backward_elements[k_0]

        if (D_f.size() > 0 and D_b.size() > 0) {
          // Interpolate over forward/backward elements.
          for (each design element D_j in D_f)
            applyElement(R_i.weight[i] × D_j, R_i)

          for (each design element D_j in D_b)
            applyElement((1 − R_i.weight[i]) × D_j, R_i)
        }
        else if (D_f.size() > 0) {
```

```
            // Apply only forward elements.
            for (each design element D_j in D_f)
              applyElement(D_j, R_i)
          }
          else {
            // Apply only backward elements.
            for (each design element D_j in D_b)
              applyElement(D_j, R_i)
          }

          updateDiffusionFlag(false, R_i)
        }
      }
    }

    // Propagate the edge field to regions marked as true.
    diffuseEdgeField(R)

    return
}
```

An option is given for users to assign either the edge field computed from input or the zero field to each region. Unlike the code in the previous subsection, design elements and their rotations/translations at each frame are pre-computed and stored in *key_elements*, *forward_elements*, *backward_elements* lists. The first one holds only design elements at keyframes. If the frame number k is a keyframe, then *key_elements*$[k]$ should contain a list of design elements; otherwise it is an empty list. The latter two are lists of design elements warped forward/backward in time from the nearest keyframe. Those warped design elements tend to have different positions and orientations. Each design element assigned to a region will modify the edge field within the region, which corresponds to invoking ap-

plyElement() function. If there are both forward and backward elements, they are multiplied by weight factors before modifying the edge field. Those three lists and weight factors for all frames are updated each time design elements are added, modified, or deleted. Below is the algorithm to compute all such data.

```
propagateDesignElement(R = {R_0, R_1, ...R_n}, D = {D_0, D_1, ...D_m}) {
  // Initialize all frames to have no design elements.
  for (each region R_j in R) {
    for (i ← 0; i < number_of_frames; i++) {
      R_j.forward_elements[i] ← empty list
      R_j.backward_elements[i] ← empty list
      R_j.key_elements[i] ← empty list
    }
  }

  // assign design elements D1 ... Dm to lists for appropriate regions
  for (each element D_j in D) {
    R_p ← D_j.region
    frame ← D_j.frame_number
    R_p.key_elements[frame].append(D_j)
  }

  // Traverse through to fill in all design elements and record the
  // distance of each frame from its nearest keyframe
  for (each region R_j in R) {
    prev_keyframe ← −1  // no previous keyframe
    next_keyframe ← −1  // no next keyframe

    // Step 1: Traverse the video forward
    for (i ← 0; i < number_of_frames; i++) {
      if (R_j.key_elements[i].size() > 0) {
        // A new keyframe is encountered
        prev_keyframe ← i
        R_j.dist_from_prev_keyframe[i] ← 0
        continue
      }
```

```
    if (prev_keyframe ≥ 0) {    // There was a keyframe before this one
      R_j.dist_from_prev_keyframe[i]
        ← R_j.dist_from_prev_keyframe[i − 1] + 1

      if (i == prev_keyframe + 1)
        E_p ← R_j.key_elements[i − 1]
      else
        E_p ← R_j.forward_elements[i − 1]

      for (each element E_j in E_p) {
        R_j.forward_elements[i].append(E_j)
        R_j.updateGeometry(E_j)
      }
    }
  }

  // Step 2: Traverse the video backward
  for (i ← number_of_frames − 1; i ≥ 0; i − −) {
    if (R_j.key_elements[i].size() > 0) {
      // A new keyframe is encountered
      next_keyframe ← i
      R_j.dist_from_next_keyframe[i] ← 0
      continue
    }

    if (next_keyframe ≥ 0) {    // There was a keyframe after this one
      R_j.dist_from_next_keyframe[i]
        ← R_j.dist_from_next_keyframe[i + 1] + 1

      if (i == next_keyframe − 1)
        E_p ← R_j.key_elements[i + 1]
      else
        E_p ← R_j.backward_elements[i + 1]

      for (each element E_j in E_p) {
        R_j.backward_elements[i].append(E_j)
        R_j.updateGeometry(E_j)
```

```
            }
        }
    }


    // Step 3: Compute a weight factor for each frame
    for (i ← 0; i < number_of_frames; i++) {
        if (R_j.key_elements[i].size() > 0) continue
        if (R_j.forward_elements[i].size() == 0) continue
        if (R_j.backward_elements[i].size() == 0) continue

        R_j.weight[i]
            ← R_j.dist_from_next_keyframe[i]/
            (R_j.dist_from_next_keyframe[i] + R_j.dist_from_prev_keyframe[i])
    }
  }
}
```

Inputs are a set of regions and design elements. We assume that each design element already has a reference to a region where it is assigned. $forward\_elements$ and *backward_elements* are computed based on a result of the *key_elements* list. To make a list of the first kind, a search for a keyframe starts from the beginning of a video sequence. Each time it is found, its frame number is recorded and its elements are copied to the $forward\_elements$ list for successive frames until a new keyframe is found. updateGeometry() corresponds to updating a design elements' position as well as orientation, in order to make them follow moving regions. Transitional and rotational components for this update are from a segmentation input, where regions are approximated to deforming ellipses for estimation of frame-to-frame transformations.

Each transformation is performed in a frame local to regions. Two positions,

starting and ending points, need to be updated for regular elements. Let P_i and T be positions of a regular element and the center of an ellipse E approximating a region, and (ΔT, $\Delta\theta$) be translation and rotation of E from the current to the next frame. Then position of P_i for the next frame can be found from the equation:

$$P_f = R(\Delta\theta)(P_i - T) + T + \Delta T \tag{4.19}$$

, where R is a transformation matrix. For singular elements, center positions and orientations need to be changed. To find a new orientation of a singular element, we simply add $\Delta\theta$ to its current orientation.

Chapter 5 – Painterly Rendering

Basic process of my paint rendering algorithm is as follows which is based on Hertzmann's idea [19].

```
paint() {
   Initialize a canvas.

   for (each layer) {
      Create a reference image from input.
      Compute a stroke list.
      Call renderLayer().
   }
}
```

As described earlier, three types of input need to be given beforehand for an image painting: input image, edge field, and style parameters. For video painting, optical flow is required as an additional input. Input image is used to generate reference images for each layer as well as compute stroke colors. Curvy strokes with various lengths are generated from edge field and style parameters. Style parameters are defined at each pixel and aggregated into a two-dimensional array. They should have reference to regions to which they belong to. In video paintings, it is preferred that strokes should follow moving objects instead of sitting at a place all the way. Optical flow is a way to estimate how far each stroke can move at each frame.

In this section for a rendering process, names of style parameters are often referenced. The following list are aliases for style parameters which altogether define a single painting style P:

d_w : stroke size
l_{min}, l_{max} : minimum/maximum stroke length (explicit renderer only)
l_{disp} : stroke length (implicit renderer only)
μ : stroke density
α_{init} : initial opacity
j_{pos} : position jitterness
j_{rand} : random color jitterness in RGB color space
j_h, j_s, j_v : color shift in HSV color space
n_{layers} : number of layers to use
Δc : area error (tolerance)
$pp_1 \ldots pp_4$: post processing parameters

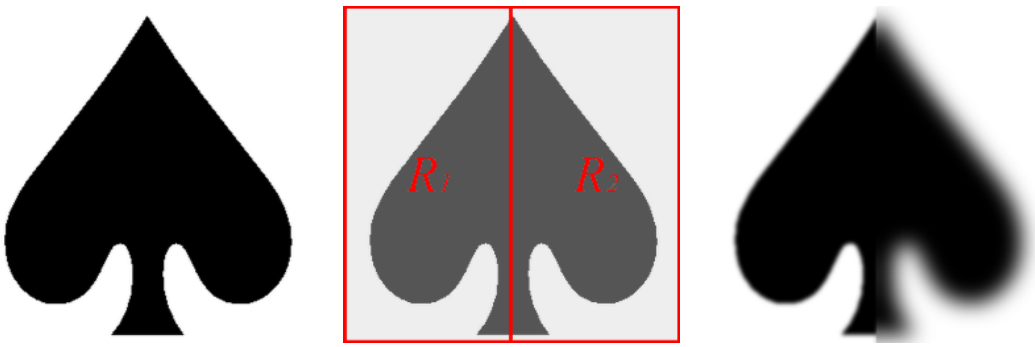
, plus a reference to a region.

At the beginning of our rendering routine, a canvas is initialized by filling a constant color $(r, g, b) = (1.0, 1.0, 1.0)$. Then a loop iterates to start from the bottom layer through upper layers. The number of iterations is computed by taking a maximum of n_{layers} among regions with style parameter keyframes.

5.1 Reference Image Generation

A reference image is an image to which we want to approximate, using the current stroke size. This means that the image shows only details that are at least as large as the stroke size [19]. To compute a reference image, the Gaussian filter is applied at each section of an input image corresponding to a region (see Figure 5.1). The kernel size for each region is determined to be a value of d_w of a single pixel of the

region. The value is rounded off to some integer. If it is even, then it is snapped to an odd number. However, this approach looks awkward if a region has no style parameter keyframe, since d_w varies within the region.



(a) (b) (c)

Figure 5.1: An example of reference image creation. (a): Input image. (b): Two regions where the Gaussian filter is applied with different kernel sizes: 3x3 pixels^2 for R_1, and 15x15 pixels^2 for R_2. (c): Result.

5.2 Stroke List Computation

A single painting consists of a number of brush strokes. They are not rendered onto a canvas unless a list of such strokes is ready at each layer. Each element in the stroke list contains some stroke attributes such as position and color, etc.

5.2.1 Image Painting

For seeding points of strokes at a bottom layer (or, the 0th layer), centers of cells in a regular grid, each separated by stroke density μ, are calculated. They are

then jittered by some amount based on j_{pos} to be seeding points, where j_{pos} is a parameter value at grid centers. For upper layers, strokes should be generated at places where errors caused by a painting at lower layers are large. To identify such areas for the i-th layer ($i \geq 1$), a canvas is subdivided into a finer grid where each cell is separated by $\mu/2^i$. Next, a difference image is built by computing color difference between the current painting and a reference image for the $(i-1)$-th layer. At each cell, a threshold Δc and the number of layers n_{layers} are read from its center. A new stroke is created for each cell if area error exceeding Δc as well as $i < n_{layers}$. Stroke location is then determined to a pixel with the largest error inside the cell.

Once stroke position is found, style parameters $P_{\mathbf{p}}$ and a tensor value T_p are read from the position. Stroke size should depend on which layer we are painting at, which leads to $d_w/2^i$. Stroke color is computed such that each HSV component of a primary color sampled from a reference image is shifted by amount of j_h, j_s, and j_v, respectively unless a hue value is undefined, followed by that each RGB component is then jittered by $j_{rand} \cdot p$, where p is a random number in $[-1, 1]$. We also store the jitter amount in RGB channels to each stroke. Finally, orientation of a stroke is determined by finding the major eigenvector of T_p.

5.2.2 Video Painting

Generating a list for the first frame is identical to the way for image painting, but a different routine must proceed for successive frames in order to maintain

temporal coherency. That means a stroke list needs to be updated rather than regenerated, which results in more strokes added into a list for each layer. However, having too many strokes raises two problems. First, too many brush strokes in upper layers from the previous frames can clutter and overlap to obscure strokes in lower layers [17]. Second, holding increasing number of strokes slows down rendering speed for subsequent frames. Thus it is necessary to cull strokes that are longer contributing to a final painting. In order to prevent upper layers from being crowded by too many strokes, each stroke is given a certain amount of lifetime, which is decremented at each frame [17].

Basic process of updating a list is a sequence of move, removal, update, and addition. In the first phase, seeding point of each stroke is updated by optical flow. Any stroke whose position is pushed away from a canvas will be subject to a removal.

In the removal phase, a stroke is removed if it is now outside a canvas, its lifetime goes to zero, or it is considered to be in a cluster with other strokes. The first two cases are trivial to identify and the last one requires a simple spatial binning. A canvas is divided into grid cells. Separation between the cells differs at each layer; for the bottom layer, a constant proportional to the minimum stroke size is used (see Figure 5.2) whereas $\mu/2^i$ is used for other layers. Each stroke is binned into cells according to their seeding positions. If there are exceeding number of strokes more than the maximum capacity of each cell, then extra strokes to be rendered early are removed.

After unnecessary strokes are removed, remaining ones are updated. Each

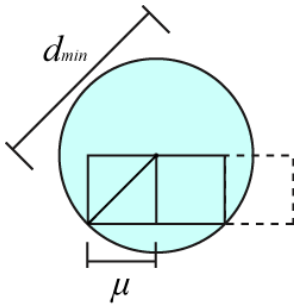


Figure 5.2: Given a minimum stroke size d_{min}, density for the bottom layer μ is set to $d_{min}/2\sqrt{2}$. This ensures that strokes are seeded enough to hide gaps.

stroke color is recomputed by using a primary color and color shift values sampled at a new position, plus a jitter value carried by the stroke. Orientation is updated in the same way for the image painting. However, eigenvectors of a tensor are bidirectional so that one additional check is necessary; a direction is flipped if newly sampled and old eigenvectors make some angle larger than 90 degrees.

Now new strokes are added as follows. At each layer a stroke image is rendered to a temporary buffer, where background and strokes are colored as white and black with their assigned opacity, respectively. This image will be used to locate uncovered regions on canvas. Finding stroke positions is the same as the one for the image painting except two things. First, grid cells in the previous spatial binning is used instead. Second, one additional check comes in to see whether the seeding position $\mathbf{p}$ is already covered by other strokes by looking up to a stroke image, using a following function:

$$isCovered(\mathbf{p}) = \begin{cases} false & \text{if } stroke\_image[\mathbf{p}] > c_{bgfg} \\ true & \text{otherwise} \end{cases} \tag{5.1}$$

, where c_{bgfg} is a threshold kept constant. Finally, new strokes are added to back of stroke list for temporal coherency.

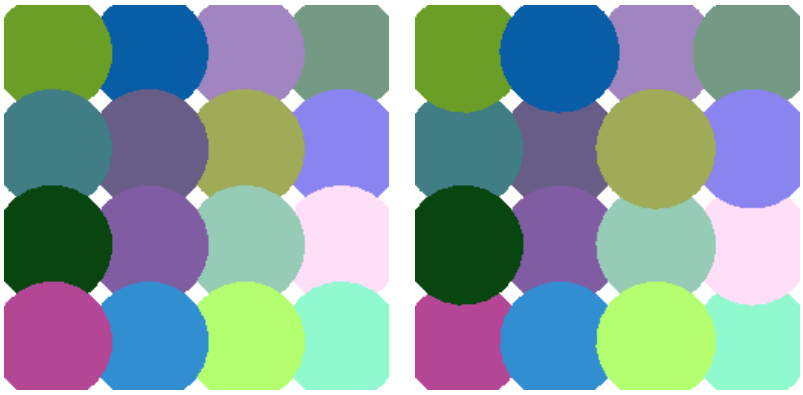
5.3 Stroke Ordering

New strokes are generated in scanline order, from left to right and from top to bottom, and they are sequentially stored in a list. Those strokes are sequentially rendered in backward order. Therefore, stroke overlap would look too regular if there were no permutation in our stroke list (see Figure 5.3). Thus sorting strokes according to some rule is necessary. Typically the sorting is applied to all strokes that are generated at each frame. It is still possible to sort in a whole set after each time of list update, but flickering will result due to recomputation of stroke colors at every frame.

A simple but effective way is to assign a random depth value to each of strokes and sort them in order.

Random: A random value is used as stroke depth. All strokes in a list are sorted in ascending order according to the assigned depth values.

Another way to sort strokes is to use their colors. There are several color representations, in which RGB and HSV color spaces are commonly used for artistic design or graphics purpose. We have designed ordering rules for each color space.



(a) (b)

Figure 5.3: An example of the stroke order. (a): Drawing strokes in sequence is unpleasant due to a regular overlap. (b): Changing draw order gives natural stroke placements.

RGB: Ordering based on RGB color space originated from a paper [54]. Their original idea is to employ the ordering as a part of hiding data into an image. Our stroke ordering follows their idea but takes a slightly different form. Suppose we have two colors $C_1 = (r_1, g_1, b_1)$ and $C_2 = (r_2, g_2, b_2)$. The *min* function is defined as follows.

```
min(C1, C2) {
  // Compute squared color intensity
  I1 ← r1^2 + g1^2 + b1^2
  I2 ← r2^2 + g2^2 + b2^2

  if (I1 < I2) return true
  else if (I2 < I1) return false
  else {
    if (g1 < g2) return true
    else if (g1 == g2 and r1 < r2) return true
    else if (g1 == g2 and r1 == r2 and b1 < b2) return true
    else return false
```

```
    }
}
```

This basically suggests that we bring a darker color in front of lighter colors. If they have the same intensity, we look at each color channel. If a green value of one color is smaller than the other, then the first color is considered smaller. If they are equal again, then we compare red channels, and so on (Figure 5.4).

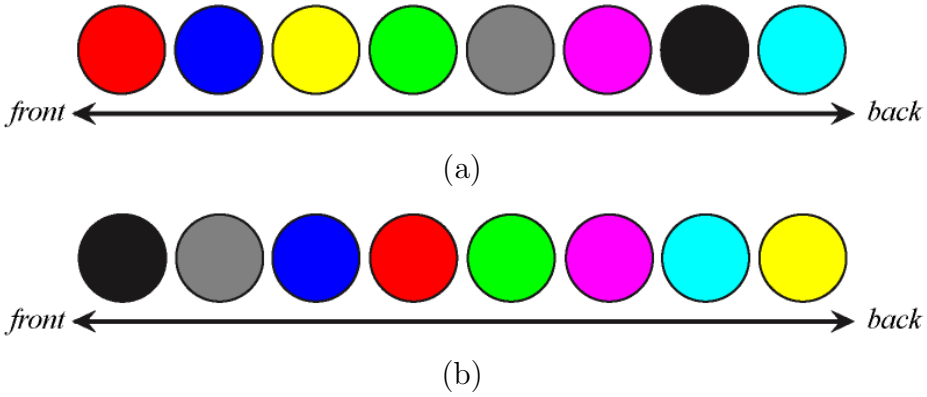


Figure 5.4: An example of stroke ordering. Color disks represent stroke primitives. (a): Unsorted stroke list. (b): Sorted list after color ordering.

The *max* function is defined in a similar fashion.

```
max(C1, C2) {
  // Compute squared color intensity
  I1 ← r1^2 + g1^2 + b1^2
  I2 ← r2^2 + g2^2 + b2^2

  if (I1 > I2) return true
  else if (I2 > I1) return false
  else {
    if (g1 > g2) return true
    else if (g1 == g2 and r1 > r2) return true
```

```
    else if (g1 == g2 and r1 == r2 and b1 > b2) return true
    else return false
  }
}
```

HSV: Ordering in HSV space is designed in more complex way than the RGB ordering. The each color component (hue, saturation, and value) is evaluated in some evaluation order. There are six possible permutations: HSV, HVS, SVH, SHV, VHS, and VSH. Two colors $C_1 = (h_1, s_1, v_1)$ and $C_2 = (h_2, s_2, v_2)$ are evaluated component-wise by using corresponding evaluators.

```
evalHSV(C1, C2) {
  // Evaluate each color component.
  // eval_order[] is an array containing an evaluation order.
  result[0] ← evalComponent(eval_order[0], C1, C2)
  result[1] ← evalComponent(eval_order[1], C1, C2)
  result[2] ← evalComponent(eval_order[2], C1, C2)

  if (result[0] == 1)
    return true
  else if (result[0] == 0 and result[1] == 1)
    return true
  else if (result[0] == 0 and result[1] == 0 and result[2] == 1)
    return true
  else
    return false
}


evalComponent(channel, C1, C2) {
    switch (channel) {
      case HUE:
        return evalHue(h1, h2)

      case SATURATION:
```

```
            return evalSat(s1, s2)

        case VALUE:
            return evalVal(v1, v2)
    }
}
```

The code above shows how two color are evaluated in HSV color space. evalHue(), evalSat(), and evalVal() are evaluators for hue, saturation, and value, respectively. The latter two provides two options to evaluate colors:

$$min(\alpha_1, \alpha_2) = \begin{cases} 1 & \text{if } \alpha_1 > \alpha_2 \\ 0 & \text{if } \alpha_1 == \alpha_2 \\ -1 & \text{otherwise} \end{cases} \tag{5.2}$$

$$max(\alpha_1, \alpha_2) = \begin{cases} 1 & \text{if } \alpha_1 > \alpha_2 \\ 0 & \text{if } \alpha_1 == \alpha_2 \\ -1 & \text{otherwise} \end{cases} \tag{5.3}$$

, where α is s for saturation, or v for value.

It naturally makes sense that both saturation and value components can be ordered. However, one might question that how to make a similar ordering for hue, which is generally visualized as a color wheel. One way is cut the wheel at some degree h_c to open up as a ribbon as shown in Figure 5.5. We use angles measured from h_c to some hue counterclockwise, and call it *map1* function. Thus the min/max functions can be defined as follows.

$$min(h_1, h_2) = \begin{cases} 1 & \text{if } map1(h_1) < map1(h_2) \\ 0 & \text{if } map1(h_1) == map1(h_2) \\ -1 & \text{otherwise} \end{cases} \tag{5.4}$$

$$max(h_1, h_2) = \begin{cases} 1 & \text{if } map1(h_1) > map1(h_2) \\ 0 & \text{if } map1(h_1) == map1(h_2) \\ -1 & \text{otherwise} \end{cases} \tag{5.5}$$

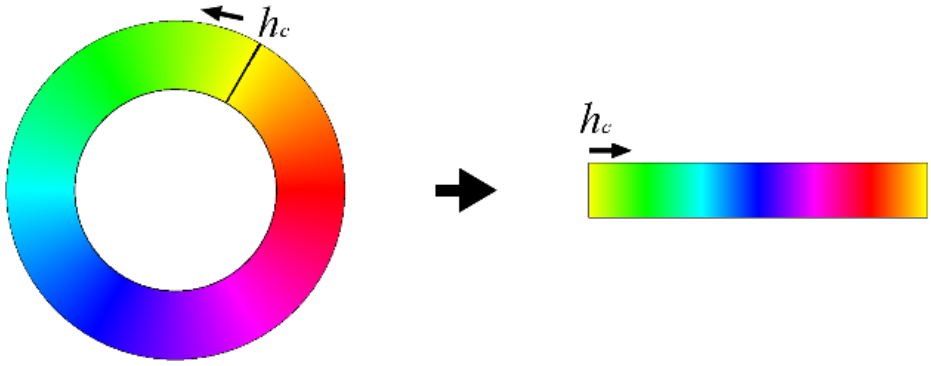


Figure 5.5: A color wheel representing hue (left) is cut at h_c, and opened into a ribbon (right).

Other possible way is compare warmness of hues. Suppose h_c refers to the warmest hue and create a function *map2*, which returns a smaller angle between h_c and a given hue. Then the warm/cool functions look like

$$warm(h_1, h_2) = \begin{cases} 1 & \text{if } map2(h_1) < map2(h_2) \\ 0 & \text{if } map2(h_1) == map2(h_2) \\ -1 & \text{otherwise} \end{cases} \tag{5.6}$$

$$cool(h_1, h_2) = \begin{cases} 1 & \text{if } map2(h_1) > map2(h_2) \\ 0 & \text{if } map2(h_1) == map2(h_2) \\ -1 & \text{otherwise} \end{cases} \tag{5.7}$$

5.4 Rendering

After draw order of stroke lists is finalized, they are used to make a painting for the current frame. In this stage, stroke shapes are computed before they are rendered onto a canvas. Those shapes are computed by finding trajectories from seeding points. In a tensor field, those trajectories are called hyperstreamlines, which are curves along an eigenvector field and analogous to streamlines in a vector field [59]. Two different renderers have been implemented: explicit and implicit. The former explicitly computes each stroke shape by tracing a hyperstreamline of major eigenvector field, whereas the latter simultaneously warps stroke attributes along the field to draw multiple strokes.

5.4.1 Explicit Approach

Mainframe of our explicit approach is based on algorithms described in Hertzmann's papers [19, 21]. Curvy strokes are represented as trianlge strips whose

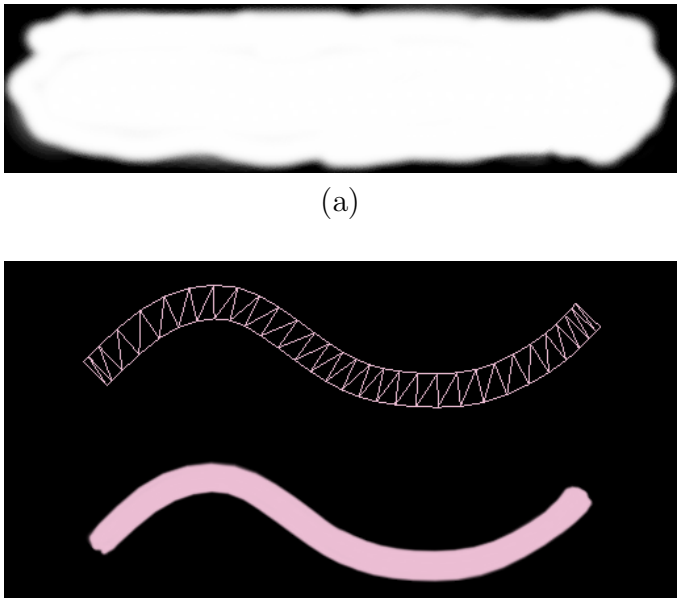
spines are uniform B-spline curves traced from a seeding point. Below are steps to render a painting per layer.

```
renderLayer() {
   Call traceStrokes() to generate all stroke shapes.
   Render strokes onto a canvas.
}


traceStrokes() {
   for (each stroke S in stroke list) {
      P ← style parameters at seeding point of S
      min_itr ← P.l_min
      max_itr ← P.l_max
      (x_0, y_0) ← seeding position of S
      (last_dx, last_dy) ← (0, 0)
      R_0 ← reference to a region containing (x_0, y_0)
      l ← current layer
      step_size ← P.d_w / 2^(l+1)

      // Calculate alpha
      S.color.α ← calcAlpha(S.lifetime, P.o_init)

      for (i ← 1; i < max_itr; i++) {
         d_0 ← colorDifference(reference_image[x_0, y_0].rgb, S.color.rgb)
         d_1 ← colorDifference(reference_image[x_0, y_0].rgb, canvas[x_0, y_0].rgb)

         // Stroke color is deviated from the color in a reference image
         if (i > min_itr and d_0 < d_1)
            continue

         // Stroke is at a vanishing point
         if (major eigenvector at (x_0, y_0) == 0)
            continue

         // Stroke goes beyond a region
         if (reference to a region containing (x_0, y_0) is not equal to R_0)
```

```
            continue

        // Outside of a canvas
        if ((x_0, y_0) is outside a canvas)
            continue

        (dx, dy) ← major eigenvector at (x_0, y_0)
        normalize(dx, dy)

        // Flip a direction if necessary.
        if (last_dx · dx + last_dy · dy < 0)
            (dx, dy) ← (−dx, −dy)

        // Find a new control point.
        (x_0, y_0) ← (x_0 + step_size · dx, y_0 + step_size · dy)

        (last_dx, last_dy) ← (dx, dy)

        // Add a control point.
        S.append(x_0, y_0)
    }

    // Build a triangle strip
    S.tessellate()
  }
}
```

This approach is inherited from Hertzmann's multi-layer painting [19]. The inner loop iterates to find a new control point until it reaches max_itr unless it is terminated from several break conditions. It stops when the current stroke color does not approximate a reference image better or the current control point is right at a degenerate point. Curves are clipped at the boundary of a region to maintain its shape detail and to prevent curves from overriding styles at neighboring regions.

Triangle strips are computed after all spline curves are generated from each set of control points. Each curve is thickened so its width to be that of a stroke. Strip width corresponds to its stroke size. Once all shapes become available, they are rendered with alpha-blending onto a canvas in backward order. Opacity map is textured on each strip to depict a realistic brush stroke appearance (Figure 5.6). Note that this idea is originally from Hertzmann [21].



(a)

(b)

Figure 5.6: An illustration of stroke rendering in the explicit renderer. (a): Opacity map. (b): A tessellated strip (top) is textured with an opacity map, giving a brush stroke shape (bottom).

5.4.2 Implicit Approach

Main difference from the previous explicit renderer is that our implicit render generates a number of strokes at once instead of sequentially draw them. To make this happen, we borrow an idea from Image Based Flow Visualization (IBFV). The method renders continuous hyperstreamlines by warping an image on a screen space to visualize a vector field, which has been further extended to support on a tensor field as well [55, 59]. In our implicit renderer, a texture mapped on a triangular mesh is repeatedly warped along an edge field. At each time, the warped texture is composited with an *initial image* to add lengths to growing strokes. Initial image contains discs with various colors and sizes according to stroke attributes. Such circular objects become winding long strokes as they get stretched. Basic steps to render a painting per layer is described as follows.

```
renderLayer()  {
   Create an initial image.
   Render an image to a temporary canvas.
   Composite the image with a canvas.
}
```

Initial image is a RGBA texture where some stroke attributes reside as circular discs. To create this, it is first initialized to $(r, b, g, a) = (1.0, 1.0, 1.0, 0.0)$ for all pixels. The last alpha component is used to determine whether a pixel belongs to a stroke (i.e. foreground) or is just background. It is foreground if the alpha value of a color $\mathbf{c}$ exceeds a threshold α_{bgfg}; otherwise background. This is described in terms of a function as follows.

$$b(\mathbf{c}) = \begin{cases} foreground & \text{if } c.a > \alpha_{bgfg} \\ background & \text{otherwise} \end{cases} \tag{5.8}$$

In the rendering step, we use the initial image to create a four different images, which are later combined into one single image as a result of painting on the current layer.

```
renderImage() {
   n ← warp iterations
   I ← initial image
   edge_field[0] ← V_{x+}, edge_field[1] ← V_{x−}
   edge_field[2] ← V_{y+}, edge_field[3] ← V_{y−}

   // Initialize each current image with the initial image.
   for (i ← 0; i < 4; i++) {
      J[i] ← I
   }

   // Repeat warp and composite for each current image.
   for (j ← 0; j < n; j++) {
      for (i ← 0; i < 4; i++) {
         tmp ← Warp J[i] with edge_field[i].
         J[i] ← compositeImages(tmp, I)
      }
   }

   // Combine all four current images.
   return combineImages(J[0], J[1], J[2], J[3])
}
```

Note that texture warping is performed on a triangular mesh whereas all compositions and combination of images are performed on pixel-basis. As preparation, four vector fields, $\mathbf{V}_{x+}$, $\mathbf{V}_{x-}$, $\mathbf{V}_{y+}$, and $\mathbf{V}_{y-}$ are computed from a major eigen-

vector field of the current tensor based edge field $\mathbf{T}$. Each of them is defined as follows.

$$\mathbf{V}_{x+} = \begin{cases} \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} & \text{if } \cos\theta \geq 0 \\ \begin{pmatrix} -\cos\theta \\ -\sin\theta \end{pmatrix} & \text{otherwise} \end{cases} \qquad \mathbf{V}_{x-} = -\mathbf{V}_{x+} \tag{5.9}$$

$$\mathbf{V}_{y+} = \begin{cases} \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} & \text{if } \sin\theta \geq 0 \\ \begin{pmatrix} -\cos\theta \\ -\sin\theta \end{pmatrix} & \text{otherwise} \end{cases} \qquad \mathbf{V}_{y-} = -\mathbf{V}_{y+} \tag{5.10}$$

This approach is similar to the idea of tensor field visualization [59]. Components of computed vector fields are normalized, and reverse flows of $\mathbf{V}_{x+}$ and $\mathbf{V}_{y+}$ are also used to grow strokes in both directions during texture warping. To warp a texture, we compute texture coordinates on each vertex [56]. Suppose $\mathbf{r}$ is a position of a vertex on a mesh and $\mathbf{v}$ is a vector value at $\mathbf{r}$ of some vector field $\mathbf{V}$. Thus the texture coordinate $\mathbf{t}$ can be computed as

$$\mathbf{t} = T(\mathbf{r} - \mathbf{v}\Delta t) \tag{5.11}$$

, where T is a function to convert into a texture space from a mesh space. Since multiple vector fields are used to produce separate images, a set of texture coordinates for each edge field needs to be computed for all vertices. Equation 5.11

is applied, substituting l_{disp} at tech vertex for Δt. Notice that stroke length is proportional to the parameter value. If the scaling factor is zero, that means texture is not warped at all, giving the Pointillism style, whereas a larger number for the parameter makes longer strokes. A new image J_{k+1} is synthesized by warping its previous image J_k along a corresponding vector field $\mathbf{V}$ and then combining the warped image with the initial image I with a composite function compositeImages().

Figure 5.7 explains how strokes are simultaneously generated. The initial state is (a), which is $J_0 = I$. J_0 is warped by a horizontal vector field (b), and J_1 is created as an image of the warped J_0 and I under the composite function (c). The new image is warped again (d) to produce J_1 (e), which will be further used to get J_2 (f).

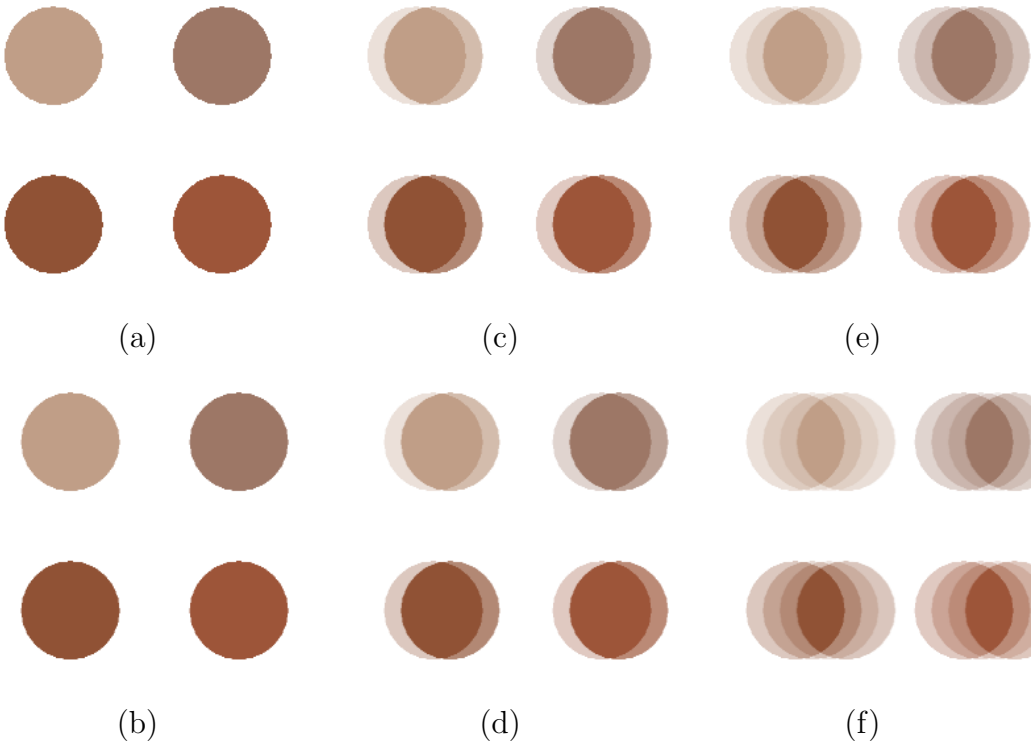


Figure 5.7: An illustration of stroke generation. For clarity of how two images are composited, the alpha-blending function is used in this example. A process of rendering is shown as a sequence from (a) to (f) (top to bottom, left to right). Stroke tips are faded out so quickly and this cannot be solved by changing a blending factor.

It is equivalent to how the original IBFV generates an image for one flow if a composite function is a linear blending function which is

$$\text{compositeImages}(I_0, I_1) = \alpha I_0 + (1 - \alpha) I_1 \tag{5.12}$$

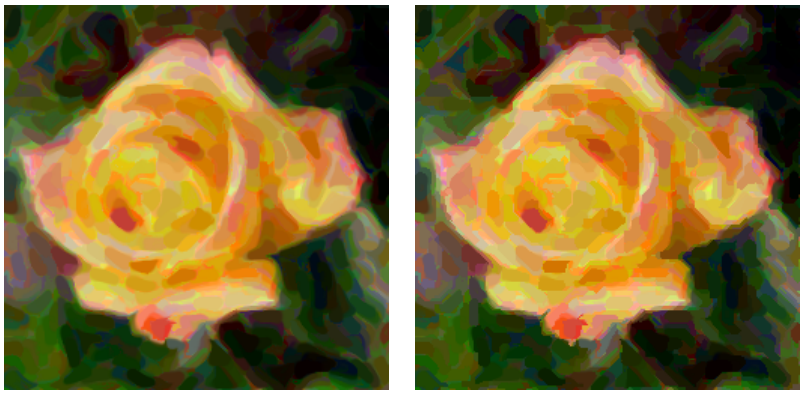
, where α is constant and input parameters I_0 and I_1 are images. However, it fails to create strokes each having consistent colors because of the color blending. Our composition function needs be based on color selection of two different colors at each pixel. Thereby, our heuristic approach has been designed as follows.

```
compositeImages(I0, I1) {
  for (each pixel p) {
    c0 ← I0(p)
    c1 ← I1(p)

    if (b(c0) is background) {
      result(p) ← c1
    }
    else if (b(c1) is background) {
      result(p) ← c0
    }
    else {
      // Evaluate with one of color order functions described in Section 5.3
      if (colorFunc(c0, c1) == true)
        result(p) ← c0
      else
        result(p) ← c1
    }
  }

  return result
}
```

Our color selection starts with checking whether each pixel in two images is background. If exactly one of them is background, then a color of the other is used. If both are background, then the pixel for output is also background but color values are copied from one of input images. If they are both foreground, then a color-based ordering function is used to compare colors in either RGB or HSV space to determine a color to select. The function helps with how two simultaneously growing strokes should be overlapped.

This heuristic approach has an issue. While this composition method supports color-based ordering, it does not support depth-based ordering. It is somehow possible to convert each depth value into a unique color and use a set of those encoded colors to generate an initial image. However, a crucial part is that the value can be easily changed when a texture image is warped. Color at a pixel is determined by computing a weighted average over texel colors close to the pixel's texture coordinate. For example, an average over white and black texels bleeds a gray. What it means is that color-coded depth can be easily changed. To avoid this, one could instead select a texel color closest to a pixel's texture coordinate. This eliminates the color bleeding effect, but introduces an artifact in which individual strokes suffer from making natural orientations. Figure 5.8 describes an example of rendering results, using different color sampling methods. Even though the weighted average exhibits color bleeding, it is in fact not that obvious. Furthermore, stroke edges are made soft due to the color bleeding, which makes brush strokes more coherent with each other.

After four different stroke images are rendered, they are combined into a one



(a) (b)

Figure 5.8: Different texture filtering during texture warp. (a) Weighted average filtering. Colors are bled around stroke edges. (b): Nearest filtering. Stroke orientations look unnatural.

single painting for the current layer. This is simply done by repeatedly calling a composition function three times.

```
// Called within renderImage()
combineImages(J_{x+}, J_{x-}, J_{y+}, J_{y-}) {
    J_x ← compositeImages(J_{x+}, J_{x-})
    J_y ← compositeImages(J_{y+}, J_{y-})
    return compositeImages(J_x, J_y)
}
```

Zhang et al. used a weight function based on eigenvector directions to combine two images generated from V_{x+} and V_{y+}. This eliminates discontinuities and creates smooth tensor field visualization [59]. In the implicit renderer, a stroke color is kept constant so a color selection is necessary rather than weighted blending.

At the last step of rendering, a layer image on a temporary buffer is composited onto a canvas. The image for the bottom layer is copied over, and images for upper

layers are alpha-blended to avoid aliasing effects around stroke edges.

```
compositeLayers() {
  C ← canvas
  L ← rendered image for current layer
  β ← fall off factor ∈ [0, 1)

  if (L is rendered for the bottom layer) {
    // Just copy over from each pixel
    for (each pixel p) {
      C(p).rgb ← L(p).rgb
    }
  }
  else {
    // Alpha-blend L with C
    for (each pixel p) {
      S ← style parameters at p

      if (S.α_init < L(p).a)  r ← 1
      else  r ← L(p).a/S.α_init

      // Compute a blending factor α.
      if (r > β)  α ← S.α_init · (r − β)/(1 − β)
      else  α ← 0

      C(p).rgb ← (1 − α) · C(p).rgb + α · L(p).rgb
    }
  }
}
```

It basically checks that if the opacity of L exceeds $\beta \cdot S.\alpha_{init}$ at each pixel. If it is not, the color at a pixel on a canvas is left unchanged; otherwise, the color on L is considered to be a part of a stroke so that it is composited over a corresponding pixel on a canvas, and $L.a$ is linearly mapped to $[0, S.\alpha_{init}]$, which will be a blending factor α. A falloff value β is a constant that has been empirically

determined. Lowering the value means that more pixels on L will be blended, but it reveals visual artifact as shown in Figure 5.9.

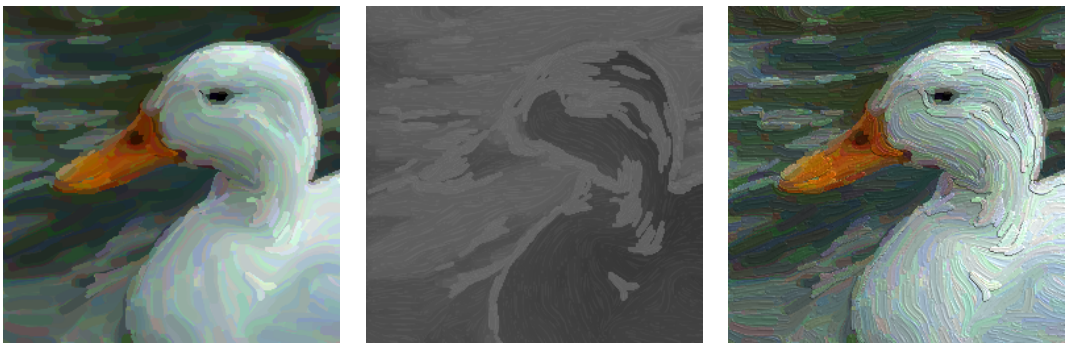
(a) (b)

Figure 5.9: An example of layer composition. The second layer is put over the first layer. (a): Default falloff value is used. (b): Extremely low falloff also reveals colors too close to background.

5.5 Post Processing

Lighting effect is added to enhance realistic appearance of paintings. The algorithm is based on [21]. Basic steps are the following (see Figure 5.10); a color image is first rendered from a set of tessellated brush stroke models with stroke colors (a). A height field is then generated from the same set of strokes with a height texture plus offset values (b). Finally, each pixel of a color image is illuminated according to the Phong shading model, with the height information of the second image (c). In my system, each style owns four style parameters for post processing ($pp_1 \ldots pp_4$) that are lighting angle, brush scale, brush height, and stroke height scale. The first

parameter is determines an azimuth of a directional light source, which illuminates a canvas at 45-degree zenith. The second parameter is to scale height texture within its space. The last two parameters are used for height exaggeration of brush texture and stroke shape, respectively. Setting the values to zero eliminates shading effect.



(a) (b) (c)

Figure 5.10: Workflow of lighting algorithm. (a): Color image. (b): Height field. (c): Shading result.

With the explicit renderer, it is straightforward to produce a height field simply by replacing a texture and giving a constant color as offset. On the other hand, it is not applicable to our implicit renderer. Remember the renderer makes a decision of stroke overlapping based on stroke colors, not a seeding order. Because of that, overlap appearance on image painting is not always consistent with that on its height field. To resolve this issue, we instead use an intensity map as a replacement of a height field. First off, a height texture is attached to each stroke during the initial image creation. The texture is used to add color variations within each stroke (Figure 5.11). Then the same rendering routine is used to generate the

second color image. Afterwards, that image is grayscaled by computing intensity at each pixel (Figure 5.12). This way is compatible with our explicit renderer so this pseudo-height map approach is set by default in our framework.

(a) (b) (c)

Figure 5.11: An illustration of the initial image creation for the post processing. (a): Initial image for a color image. (b): Height texture is used to add color variations in each stroke. (c): Initial image to render the second color image.

(a) (b) (c)

Figure 5.12: Height field creation in our implicit renderer. (a): Color image with height texture. (b): Intensity of (a) as height field. (c): Shading result.

Chapter 6 – Results

6.1 Painting Styles

We have designed 5 different style presets: Van Gogh, Colorist Wash, Impressionism, Pointillism, and Fracture. Each of them is accessible from my painting framework to apply onto regions. Each style is numerically defined by style parameters based on the following observations.

Van Gogh:

- Stroke Attributes: short, opaque, moderate amount of stroke position/color jittering, no HSV color shift.
- Multi-layering: 1 layer.
- Post Processing: Both stroke edges and inner texture are highlighted.

Colorist Wash:

- Stroke Attributes: long, semi-translucent, moderate amount of stroke position/color jittering, no HSV color shift.
- Multi-layering: 3 layers.
- Post Processing: Only stroke edges are highlighted.

Impressionism:

- Stroke Attributes: long, opaque, moderate amount of stroke position/color jittering, no HSV color shift.

- Multi-layering: 3 layers.
- Post Processing: Both stroke edges and inner texture are highlighted.

Pointillism:

- Stroke Attributes: rounded, opaque, large position jitter, small color jitter, no HSV color shift.
- Multi-layering: 1 layer.
- Post Processing: Neither stroke edges nor inner texture is highlighted.

Fracture:

- Stroke Attributes: rounded, opaque, large position jitter, small color jitter, no HSV color shift.
- Multi-layering: 1 layer.
- Post Processing: Both stroke edges and inner texture are highlighted.

6.2 Single-Style Painting

Painted results from explicit and implicit renderers are shown in Figure 6.1. For a comparison between two renderers, the same style parameters and stroke ordering were used. With the first three styles (Pointillism, Fracture, and Van Gogh), both renderers clearly show individual strokes. Visual difference is much more obvious

on multi-layer styles (Impressionism and Colorist Wash). With an implicit renderer, strokes seems more fluidic than the ones from an explicit renderer. However, they are appearing as clusters on upper layers, which makes it harder to distinguish each shape of them.

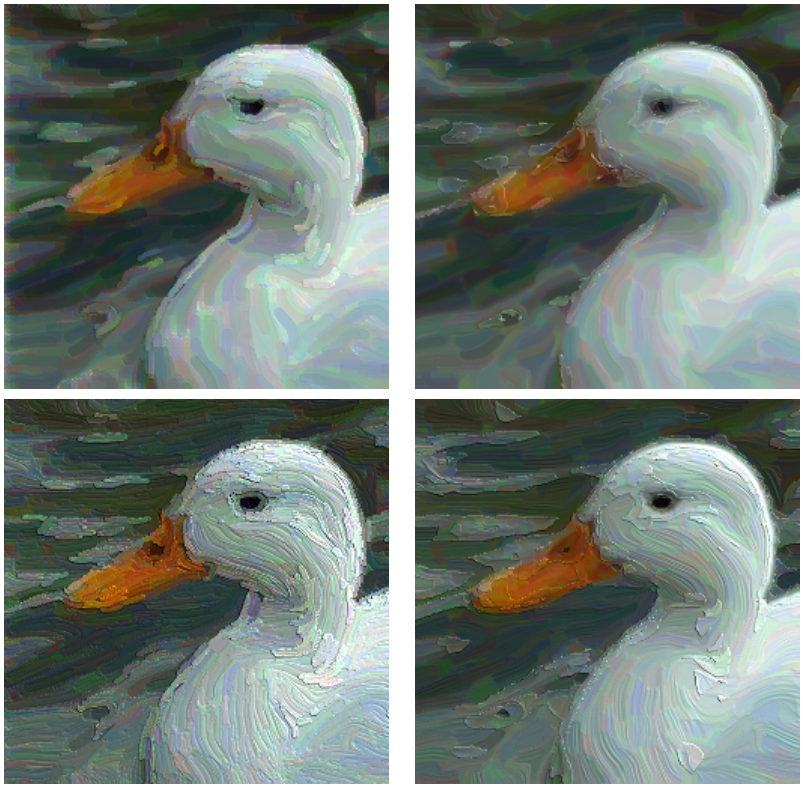


Figure 6.1: Image painting result of the explicit method (left) and the implicit method (right). Top: Colorist Wash. Bottom: Impressionism.

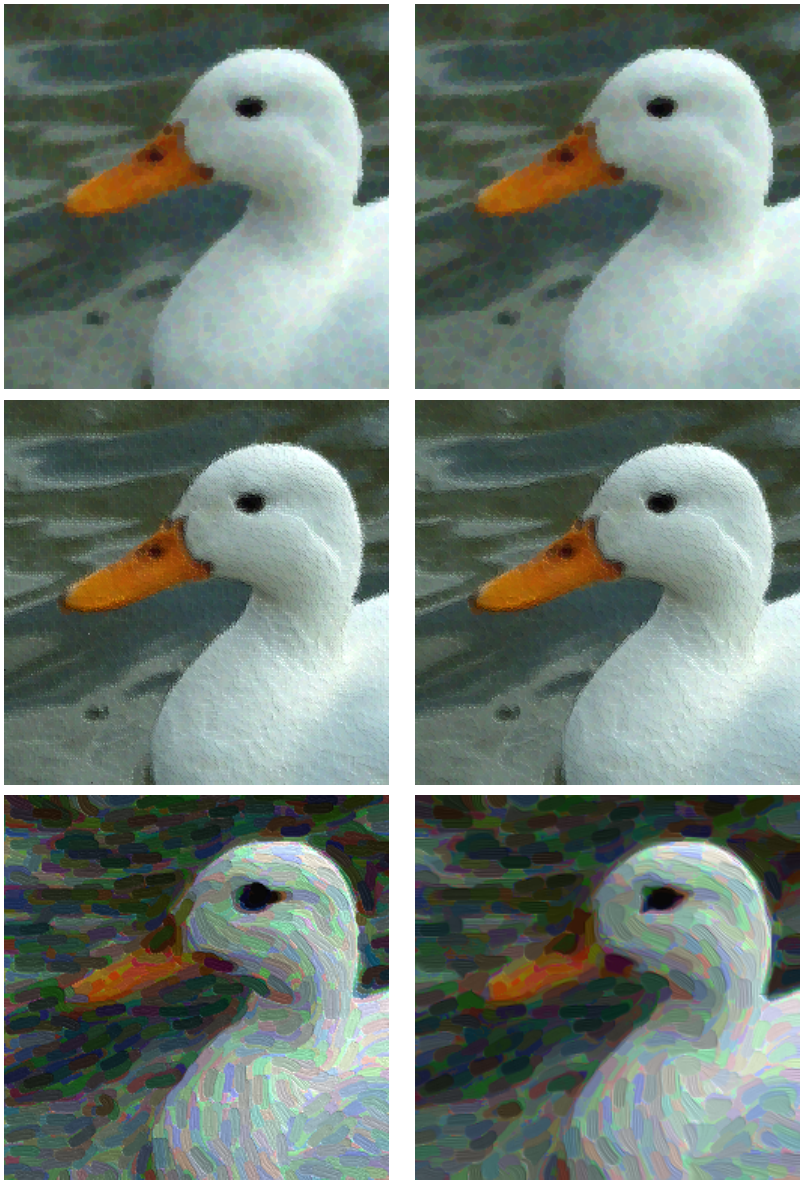
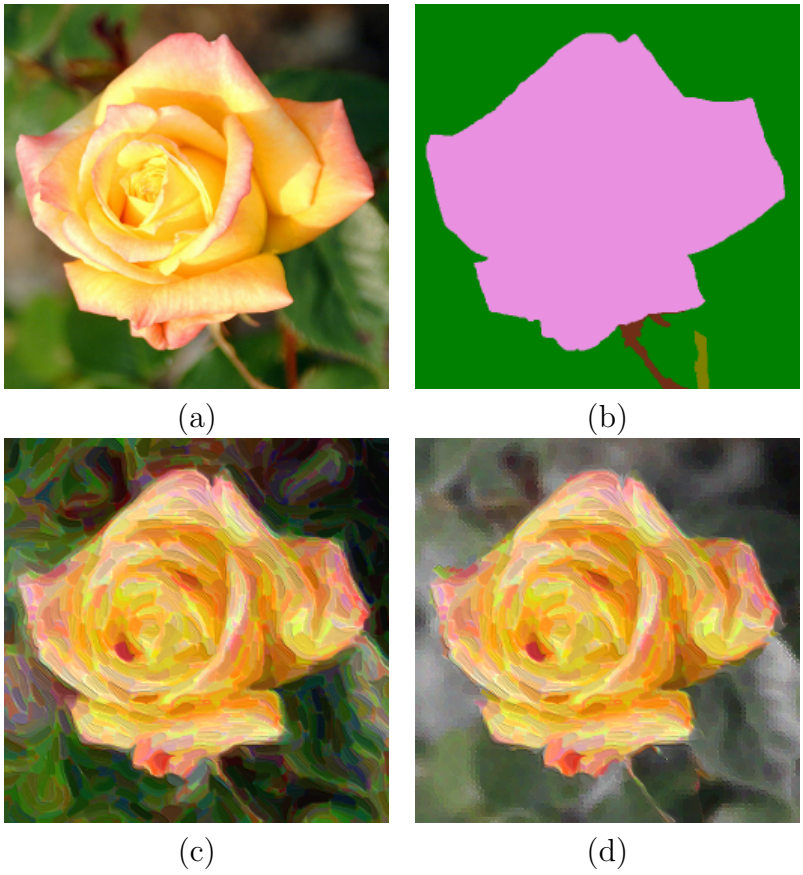


Figure 6.1: (Continued) Image painting result of the explicit method (left) and the implicit method (right). Top: Pointillism. Middle: Fracture. Bottom: Van Gogh.

6.3 Multi-Style Painting

One effective way of multi-style painting is to pay viewer's attention to a main object in a scene by choosing a style for background different from foreground. Figure 6.2 shows an example of a single flower image.



(a) (b)

(c) (d)

Figure 6.2: Multi-style painting result. (a): Input image. (b): Segmentation data. Background is colored with green. (c): Single-style painting. (d): A multi-Style result. Background is painted with punctate strokes of desaturated colors to enhance a flower. Original image from FreeFoto.com.

In Figure 6.2, (c) was generated by applying a single style to a whole image,

and (d) was generated by using different styles to each segmented region according to (b). With a background scene having a style containing desaturated colors, the flower object can stand out more than (c).

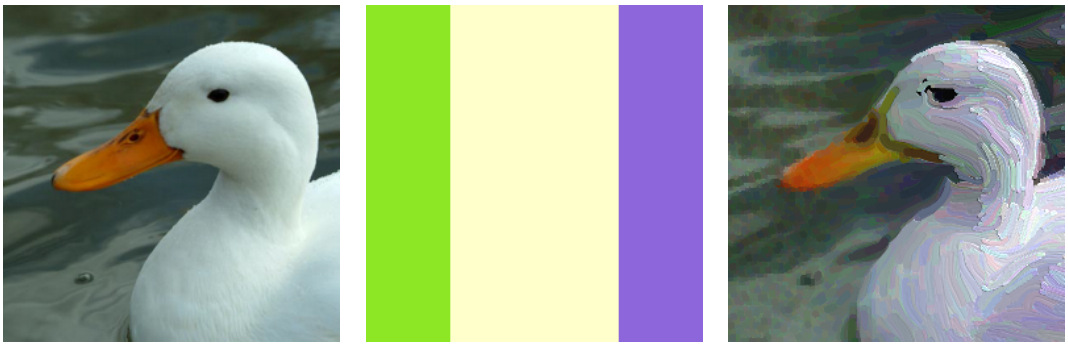
Another positive usage of multi-style is focusing/defocusing multiple targets in a video clip. For example, smaller strokes are used to finely describe objects to be focused whereas larger strokes are applied to get out of viewer's focus. Our video painting example shows that a lady and a child are alternately focused over time (Figure 6.3).

(a) (b) (c)

Figure 6.3: Using multi-style technique to change targets to be focused. (a): A lady is focused. (b): Focus is changed to a child by changing stroke sizes for a lady and a child. (c): Both figures are focused. Original video courtesy of Artbeats.

Figure 6.4 shows a style interpolation within a single image. Different styles were applied to two regions on sides (green and purple) as shown in (b), and the last region has left unspecified. The result (c) shows that the two styles are smoothly blended to define a varying style within the third region.

Benefit from using design elements is that stroke orientations can be modified



(a) (b) (c)

Figure 6.4: Spatial interpolation on style parameters. (a): Input. (b): Segmentation image (3 regions). (c) Painting result after each region colored in green and purple is assigned a different style. See how a beak color and stroke length spatially change.

to remove undesirable noise or introduce new features. Figure 6.5 shows snapshots of a video painting result along with edge fields. One regular element has been set for a dolphin to create a regular flow inside the object. Multiple design elements have been spread over time for a water region to simulate animating ripples when a dolphin jumps into and out of the surface. Quality of design element's object tracking depends on accuracy of motion data supplied from segmentation data. For example, a regular element for a dolphin reasonably follows her. On the other hand, although we could add moving ripple patterns to water surface, we also saw that some elements were pushed as opposed to our expectation. It required further user interaction to fix the issue. In our example, additional elements were set to relocate a ripple center.

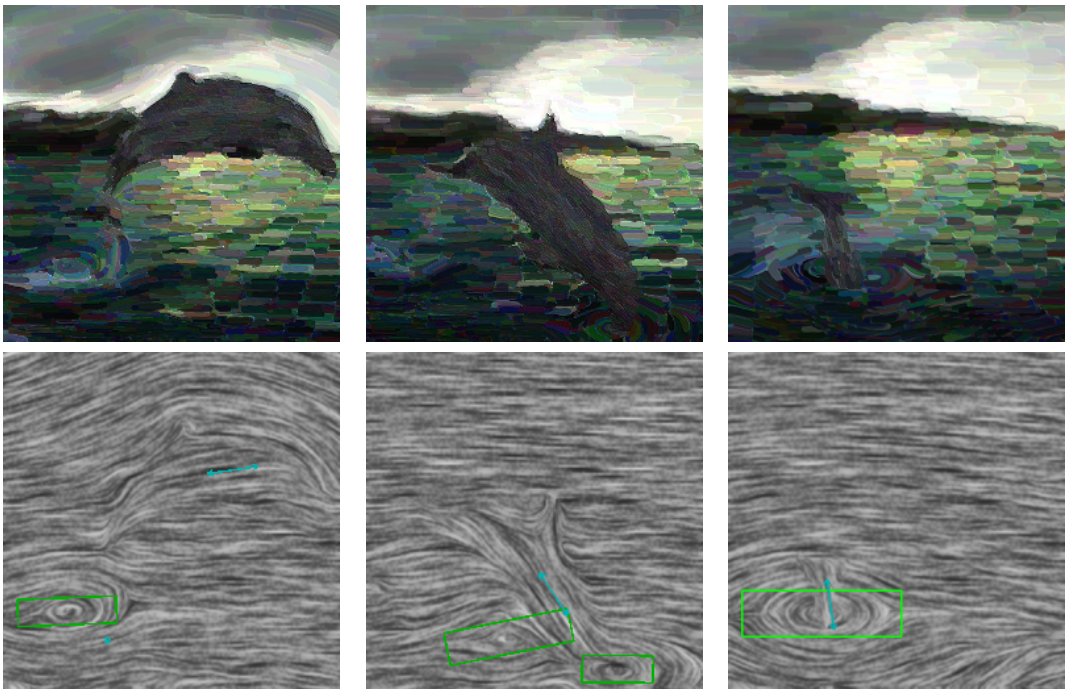


Figure 6.5: Painting video result (top) and its edge field (bottom). Design elements are represented as green boxes and cyan arrows. Original video courtesy of Artbeats.

6.4 Stroke Ordering

Figures 6.6 $\sim$ 6.8 show painting results with different stroke ordering applied to an image and a video. The first figure contains image paintings with the explicit renderer displaying random, RGB-based, and 8 different HSV-based stroke orderings. Note that this shows only a part of all possible orderings. HSV ordering has six ways to evaluate in order, and there are four functions to compare hue and two for the other components. This means we have $(3!{\cdot}4{\cdot}2{\cdot}2) = 96$ combinations for HSV-based stroke ordering. Moreover, the warmest hue h_c can be set any value between 0 and 360, which means there are a quite number of HSV-based color ordering. The painted results from the color-based ordering gives an effect of strokes with similar colors coherently appearing on a canvas.

Choice of stroke ordering affects overall appearance such as brightness, contrast and color variations. Moreover, it matters under the implicit renderer in terms of quality. We observed that certain types of orderings based on HSV color space give granular look. For example, it fails to produce slick strokes when hue is evaluated first with warm/cool functions or saturation is evaluated first with a min function (Figure 6.7). Interestingly, we did not encounter this kind of obvious artifact with RGB-based ordering rules.

The effect of color ordering is predominant on image paintings as well as very beginning frames in video paintings. However, it diminishes in successive frames so rendered videos with different ordering methods eventually get to similar visual appearance (Figure 6.8). To avoid this, a whole set of strokes in a stroke list,

instead of ones that are newly created, could be sorted at every a certain number of frames, but it would be inevitable that some strokes would pop up. Another way would be to make a stroke color static. In other words, once a new stroke is generated, its color should be fixed. Unfortunately we have never experimented the latter approach and it is unsure that how much effectively it would alleviate the disappearance of color ordering effect.

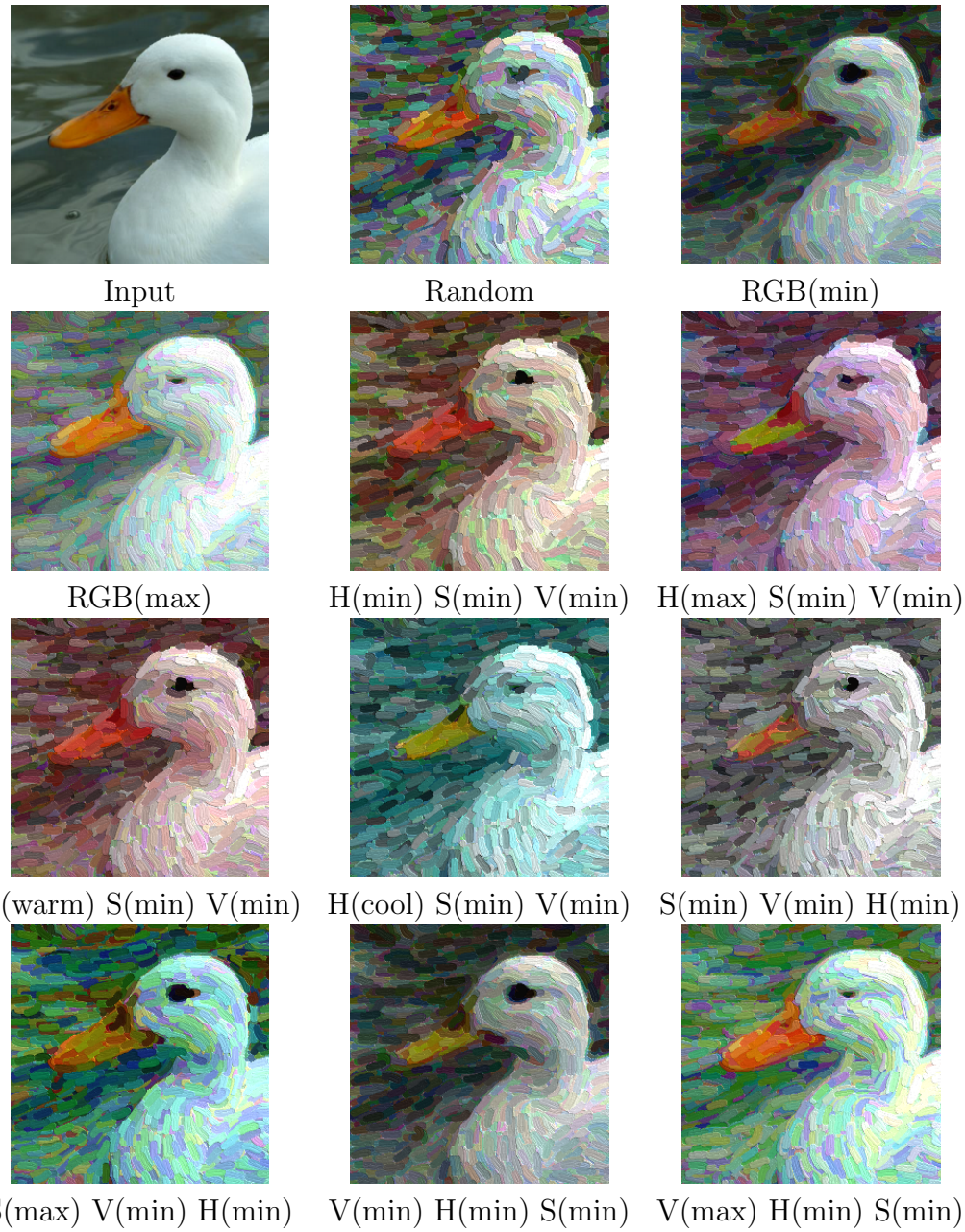


Figure 6.6: Image paintings generated by the explicit renderer with different color ordering. The same Van Gogh style has been applied with parameter values unchanged. $h_c = 0°$ was used for HSV-based ordering.

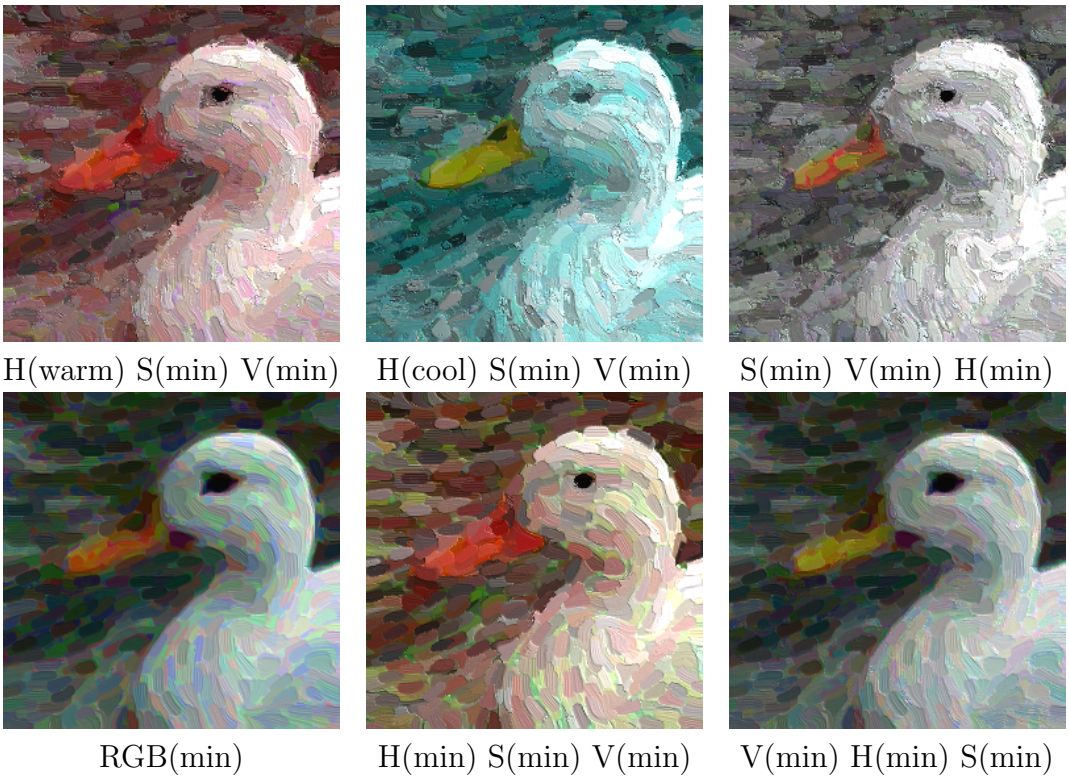


Figure 6.7: Examples of stroke ordering that produce high granularity on an implicit renderer (top) and that does not (bottom).

Figure 6.8: Snapshots of video paintings (Frame 0, 100, and 190) generated by an explicit renderer with different color ordering. Top: Random. Middle: RGB(min). Bottom: H(max) S(min) V(min). Original video courtesy of Artbeats.

Chapter 7 – Discussion

7.1 Implementation

Our rendering system was built based on OpenGL and OpenCV. Both implicit and explicit renderers were implemented to run on a fixed graphics pipeline as well as programmable shaders. The implicit renderer benefits from GLSL to avoid pixel reads from texture memory during warping, compositing, and combining texture images. OpenCV provides a set of functionalities suited for computer vision research, such as image smoothing, optical flow computation, and even video loading/saving.

7.2 Image Dimension

The current version of my painting framework automatically adjusts the input image size into 512×512 pixels^2. Extra rows/columns are automatically added to any input smaller than the size, but rendered outputs have the same dimension as the inputs. Any input exceeding 512 pixels in one dimension will be cropped, and the possible output size is limited to at most 512×512 pixels^2.

7.3 Style Specification

7.3.1 Diffusion

Push-pull method helps with fast spatial propagation of style parameters and an edge field, but it still takes a while to compute if one wants to define styles for vast areas, using style parameters defined at a tiny fractional regions. In addition to that, the method uses a pyramidal approach where upper levels are in the half the size of their one lower levels. This forces us to introduce additional rows and columns to make a grid size be in the power-of-two only for a diffusion process. If arbitrary size for input will be allowed, we need to allocate extra memory for those padded pixels as well as for pixels corresponding to the input size.

Style propagation works if style parameters are assigned to at least one region; otherwise it fails since there is nothing to transmit. Consider a scenario of a video sequence in which a dolphin keeps jumping out of water and diving back. Suppose the video's segmentation data shows two distinct regions describing a foreground object (a dolphin) and a background scene. If a style is applied to a dolphin, the style for background will be determined with the propagation from the animal, but what happens if the animal completely hides under the water at some frame? The frame contains nothing to propagate for the background so we are unable to generate a painting consistent with other frames. One way to avoid the situation is to make sure at least one style is defined and always visible on a canvas at all frames.

7.3.2 Segmentation

Video segmentation is processed at the offline stage, and the precomputed result is passed to my framework, which assumes that the input is well enough defined; both the segmentation is perfect and motion data associated to each region are promising. If a user wants to change the current segmentation data completely, the worst scenario is that all the design processes must be started over. Such a case happens when segmentation structure in the new data is different from the old one: for example, a different number of regions. Redoing video segmentation several times will become a lot of user intervention to generate a single video painting.

Even if one could produce the best segmentation, another concern is quality of motion estimation for each segmented region. For example, when we were designing edge field for a dolphin video, design elements assigned to a dolphin reasonably tracked the animal. However, at some frames design elements simulating water ripples were displaced to the opposite direction of our expectation, which made us to introduce additional elements to relocate the ripples at a right position.

For some video inputs, estimated rotations are noisy and we observed unpleasant angular oscillation back and forth. This will not visually influence under certain kinds of design elements such as a center and a focus. If elements of the other type are used, we will observe unwilling stroke orientation changes. Exploration of a better region-wise motion estimation algorithm will be one of the next steps to push our painting results better in temporal coherency.

7.3.3 Style Parameter Design

In my opinion, shifting a stroke color on a region is powerful in that stroke colors are directly manipulated by users. For instance, a dusk image can be easily converted into the night sky by changing a color hue from red to bluish purple. We opted to HSV color space for this operation since it is commonly used in major graphics application and it is easily converted to the RGB color space back and forth. In the HSV space, a color is described as three attributes: hue, saturation, and value. Within the space, colors are more intuitively and easily controlled than in the RGB space which more or less measures content of red, green, and blue components.

One concern on color shifting is that we are not able to perturb any grayish color. Consider how the HSV color space can be visualized as a single cone. At the bottom rim corresponds to hue color wheel. Saturation is namely a distance from the center of a color wheel, and value is a length of a altitude measured from a cone vertex. Any grayish color lies on the altitude, where hue is really undefined. Thus visual artifact shows up when color shift is applied to a region containing any zero-saturation color: a glare from the sun, for instance. Possibly pre-filtering input to avoid inclusion of such colors would help.

In the current version of my framework, only a single regular grid is used to compute all stroke positions at a layer, even though different stroke density values can be defined at each style. An alternative way could be to repeat the computation for each region, using grids with different resolutions, but this works only if every region has its own style parameters. One way to think about this com-

putational problem is that we treat it as a particle-based glyph packing problem, which is for instance described in [30]; stroke density could be interpreted as an isotropic second-order tensor field. Particle systems are used to represent glyph locations, having potential energy field formed by local tensor values. Alignment of the particle systems is iteratively updated to minimize glyph overlaps. Then its stabilized solution can be used as a set of seeding points of all strokes. However, computational speed is an issue; the method will require thousands of iterations until the systems are stabilized, which is less favoring if we want to make our framework interactive.

7.3.4 Edge Field Design

Design elements are provided to users in order to modify local stroke orientations, such as adding circular curves or streamline flows. However, they do not have a strong control to shape an edge field into arbitrary forms. Combination of multiple design elements would solve this issue, but it is likely that users would be asked to study formula of such element combination for patterns they want to achieve. One could suggest to use a brush-based interface discussed in [12].

Smoothing an edge field over consecutive frames increases temporal coherency of stroke orientations, but many rendered strokes are still jumping around over frames. Hays put constraints on brush strokes so that they cannot rotate more than a specified amount. This is not applicable to the implicit renderer since strokes are generated by warping an image, instead of explicit curve computations.

As I explained earlier, applying the angular constraint on an edge field is not a successful solution for the approach.

7.4 Painterly Rendering

7.4.1 Reference Image

Remember that a reference image is generated from an input image by applying a Gaussian filter with varying window sizes proportional to local stroke sizes. Typically the widow size of such a filter remains fixed within a single operation. This means multiple iterations are required to apply the filter with different window sizes to each region. If every segmented region has its own style parameters, then the number of iterations is exactly the number of the regions. However, what if some region needing style propagation also exists? More number of iterations is necessary, but obviously it is not a novel approach. One way is to use as a window size an average of stroke sizes within a region. This will not charge extra iterations and choice of the size is more reasonable than my current approach. Another possibility is to use a summed area table [24]. It gives a box-filtering but it is possible to specify different window sizes at pixel level. Thereby only one iteration suffices although constructing an image integral is required as a pre-processing step.

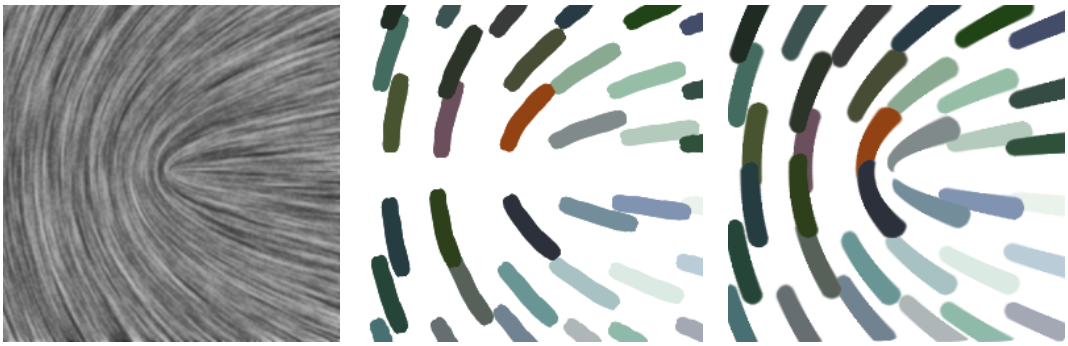
7.4.2 Stroke Shapes

In the explicit renderer, every stroke is represented as a tessellated strip with a shape texture mapped on. Reusing a single texture for all strips does not create stroke variation at all. An easy way to resolve this issue is we create multiple textures each depicting a different stroke shape. Textures are loaded from files by a painting program so that it is easy to replace the current shapes with a new one if users do not like them. However, it remains that each stroke still keeps its constant width.

On the other hand, with the implicit method, stroke width can vary around degenerate points. For example, strokes are tapered as they are approaching toward a wedge point, or they can be blown up or even split at a trisector. Inputs for my painting system are images or videos that are photographed or captured from real scenes. Thus edge fields computed from those inputs are topologically complex so they exhibit many spreading degenerate points, which gives more random varieties on stroke shapes.

7.4.3 Color Bleeding

Each time after image warping in the implicit renderer, new colors easily appear at places where different colors are facing: i.e. around stroke edges. On one side, this is gladsome that strokes nicely and coherently overlap. Brush strokes are still individually recognized but they altogether serve themselves as a whole painting image without each stroke shape standing out too much for viewers. On the other



(a) (b) (c)

Figure 7.1: Stroke shapes rendered from different renderers. (a): A wedge as a sharp edge field (b): The explicit method keeps stroke width constant. (c): With the implicit method, stroke width varies under the edge field topology.

hand, it is a nuisance because it is hard to render completely solid-colored strokes.

To see why that happens, consider two strokes with a color of the first larger than the other. As they get warped, new colors are introduced around their edges. Then the warped image will be composited with the initial image. At some pixel, the new color of the first stroke will be compared with the original color of the second. It can happen that the former is now smaller than the latter. This causes splotchy dots within a single stroke or a banding effect as shown in Figure 7.2.

It would be difficult to render blur-edged strokes without having the visual artifact. With the implicit renderer, stroke shapes are neither computed nor stored in some other place so some novel image operation is necessary to help our hands. Due to limited knowledge of image editing, I have not found a good candidate. However, although such stroke color interference is one visual drawback of our implicit renderer, they do not stand out by themselves and they happen under

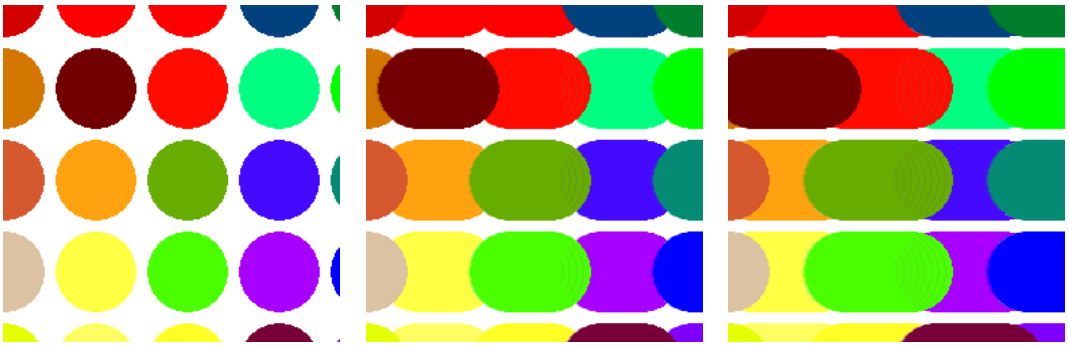


Figure 7.2: Banding effect occurs during composition stages of an implicit renderer. Images above show a process of stroke rendering where an image is warped to the left.

only some color combinations.

7.4.4 Stroke Ordering

As mentioned in the previous chapter, it is difficult to maintain visual effect of color-based ordering as well as temporal coherency. If we prefer to stroke color coherency at each frame, the ordering must apply to a whole stroke set, which leads to stroke scintillation. If we want to avoid such an effect, then strokes are not getting locally color-coherent. Another thing to note is that the color-based ordering is not controlled per region, which might be argued for more flexibilities on my multi-style framework or might be not that important since there are already a number of controllable style parameters to design. Additional controls would give users more freedom to seek for better painting designs; on the other hand, it could introduce more user interaction that could lead to a more time-consuming task.

Chapter 8 – Conclusion

We have described our multi-style interactive framework. Images and videos are processed to manufacture still or moving paintings with multiple different styles. We have shown how design elements are defined and how they are useful to fix a given edge field. Parameterizing stroke attributes and other properties makes it possible to have a wide variety of different styles. They can be keyframed per region so that they change over time within a region. Styles and fixed edge field are propagated to fill values for regions with no keyframe assigned.

It has been shown that stroke generation in our implicit renderer is based on image warping. The renderer can produce paintings as many varieties of different styles as an explicit renderer. It also adds randomness on stroke shapes so that they do not always have constant width, as opposed to the other.

The color-based stroke ordering can be made in various ways; it can be comparing intensities followed by each color channel, or a comparison can be made in HSV space in any favorite order. The resulting paintings give different impressions from the same input and style settings, and strokes with similar colors coherently appear.

There are, however, several issues to be addressed. Style propagation fails if no defined style is available. In the implicit renderer, color bleeding causes a visual artifact such as color bands within brush strokes. Since stroke computation is image-based that involves warping, depths cannot be encoded into colors; this means that the implicit renderer is dependent on the color-based stroke ordering. Bringing the ordering to videos still remains as a challenging problem. If strokes

are partially sorted, the effect of color-based ordering fades away. If an entire stroke set is sorted to enforce the order, scintillation is inevitable. Other issues should be resolved; fast diffusion method for non-square region, better reference image computation, better computation for edge field smoother in both space and time domains, and so on. They will be left for any improved version of a multi-style painting framework in future development.

Bibliography

[1] Aseem Agarwala, Aaron Hertzmann, David H. Salesin, and Steven M. Seitz. Keyframe-based tracking for rotoscoping and animation. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 584–591, New York, NY, USA, 2004. ACM.

[2] Teresa W. Bleser, John L. Sibert, and J. Patrick McGee. Charcoal sketching: returning control to the artist. *ACM Trans. Graph.*, 7(1):76–81, 1988.

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

[4] Adrien Bousseau, Fabrice Neyret, Joëlle Thollot, and David Salesin. Video watercolorization using bidirectional texture advection. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 104, New York, NY, USA, 2007. ACM.

[5] Brian Cabral and Leith Casey Leedom. Imaging vector fields using line integral convolution. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 263–270, New York, NY, USA, 1993. ACM.

[6] Ming-Te Chi. Stylized and abstract painterly rendering system using a multiscale segmented sphere hierarchy. *IEEE Transactions on Visualization and Computer Graphics*, 12(1):61–72, 2006. Member-Lee,, Tong-Yee.

[7] Harold Cohen. The further exploits of aaron, painter. *Stanford Hum. Rev.*, 4(2):141–158, 1995.

[8] J. P. Collomosse and P. M. Hall. Painterly rendering using image salience. In *EGUK '02: Proceedings of the 20th UK conference on Eurographics*, page 122, Washington, DC, USA, 2002. IEEE Computer Society.

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

[10] Doug DeCarlo and Anthony Santella. Stylization and abstraction of photographs. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 769–776, New York, NY, USA, 2002. ACM.

[11] Oliver Deussen, Stefan Hiller, Cornelius Van Overveld, and Thomas Strothotte. Floating points: A method for computing stipple drawings. *Computer Graphics Forum*, 19:40–51, 2000.

[12] Greg Esch, Peter Wonka, Pascal Müller, and Eugene Zhang. Interactive procedural street modeling. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, page 35, New York, NY, USA, 2007. ACM.

[13] Adam Finkelstein and Marisa Range. Image mosaics. In *EP '98/RIDT '98: Proceedings of the 7th International Conference on Electronic Publishing, Held Jointly with the 4th International Conference on Raster Imaging and Digital Typography*, pages 11–22, London, UK, 1998. Springer-Verlag.

[14] Bruce Gooch, Greg Coombe, and Peter Shirley. Artistic vision: painterly rendering using computer vision techniques. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 83–ff, New York, NY, USA, 2002. ACM.

[15] Paul Haeberli. Paint by numbers: abstract image representations. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 207–214, New York, NY, USA, 1990. ACM.

[16] Alejo Hausner. Simulating decorative mosaics. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 573–580, New York, NY, USA, 2001. ACM.

[17] James H. Hays and Irfan Essa. Image and video based painterly animation. *NPAR 2004: Third International Symposium on Non-Photorealistic Animation and Rendering*, pages 113–120, June 2004.

[18] Christopher G. Healey, Laura Tateosian, James T. Enns, and Mark Remple. Perceptually based brush strokes for nonphotorealistic visualization. *ACM Trans. Graph.*, 23(1):64–96, 2004.

[19] Aaron Hertzmann. Painterly rendering with curved brush strokes of multiple sizes. *Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH 1998)*, pages 453–460, 1998.

[20] Aaron Hertzmann. Paint by relaxation. In *CGI '01: Proceedings of the International Conference on Computer Graphics*, page 47, Washington, DC, USA, 2001. IEEE Computer Society.

[21] Aaron Hertzmann. Fast paint texture. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 91–ff, New York, NY, USA, 2002. ACM.

[22] Aaron Hertzmann. A survey of stroke-based rendering. *IEEE Computer Graphics and Applications*, 23(4):70–81, 2003.

[23] Aaron Hertzmann, Charles E. Jacobs, Nuria Oliver, Brian Curless, and David H. Salesin. Image analogies. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 327–340, New York, NY, USA, 2001. ACM.

[24] Aaron Hertzmann and Ken Perlin. Painterly rendering for video and interaction. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 7–12, New York, NY, USA, 2000. ACM.

[25] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 517–526, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[26] Victoria Interrante. Harnessing natural textures for multivariate visualization. *IEEE Comput. Graph. Appl.*, 20(6):6–11, 2000.

[27] Matthew Kaplan, Bruce Gooch, and Elaine Cohen. Interactive artistic rendering. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 67–74, New York, NY, USA, 2000. ACM.

[28] Hiroaki Kawata, Alexandre Gouaillard, and Takashi Kanai. Interactive point-based painterly rendering. *Cyberworlds, International Conference on*, 0:293–299, 2004.

[29] Junhwan Kim and Fabio Pellacini. Jigsaw image mosaics. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 657–664, New York, NY, USA, 2002. ACM.

[30] Gordon Kindlmann and Carl-Fredrik Westin. Diffusion tensor visualization with glyph packing. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1329–1336, 2006.

[31] Allison W. Klein, Peter-Pike J. Sloan, Adam Finkelstein, and Michael F. Cohen. Stylized video cubes. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 15–22, New York, NY, USA, 2002. ACM.

[32] Allison W. Klein, Peter-Pike J. Sloan, Adam Finkelstein, and Michael F. Cohen. Stylized video cubes. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 15–22, New York, NY, USA, 2002. ACM.

[33] Alexander Kolliopoulos, Jack M. Wang, and Aaron Hertzmann. Segmentation-based 3d artistic rendering. In *Eurographics Symposium on Rendering (EGSR'06)*, pages 361–370, 2006.

[34] Levente Kovács and Tamás Szirányi. Painterly rendering controlled by multiscale image features. In *SCCG '04: Proceedings of the 20th spring conference on Computer graphics*, pages 177–184, New York, NY, USA, 2004. ACM.

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

[36] David H. Laidlaw. Loose, artistic "textures" for visualization. *IEEE Comput. Graph. Appl.*, 21(2):6–9, 2001.

[37] Gregory Lecot and Bruno Le'vy. Ardeco: Automatic region detection and conversion. In *Eurographics Symposium on Rendering*, 2006.

[38] Peter Litwinowicz. Processing images and video for an impressionist effect. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer*

graphics and interactive techniques, pages 407–414, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[39] Zhanping Liu. An advanced evenly-spaced streamline placement algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):965–972, 2006. Senior Member-Moorhead,, Robert and Student Member-Groner,, Joe.

[40] Aditi Majumder and M. Gopi. Hardware accelerated real time charcoal rendering. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 59–66, New York, NY, USA, 2002. ACM.

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

[42] Luke Olszewski. A timing comparison of the conjugate gradient and gauss-seidel parallel algorithms in a one-dimensional flow equation using pvm. In *ACM-SE 33: Proceedings of the 33rd annual on Southeast regional conference*, pages 205–212, New York, NY, USA, 1995. ACM.

[43] Wai-Man Pang, Yingge Qu, Tien-Tsin Wong, Daniel Cohen-Or, and Pheng-Ann Heng. Structure-aware halftoning. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–8, New York, NY, USA, 2008. ACM.

[44] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, page 581, New York, NY, USA, 2001. ACM.

[45] Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. Orientable textures for image-based pen-and-ink illustration. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 401–406, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[46] Anthony Santella and Doug Decarlo. Abstracted painterly renderings using eye-tracking data. In *in Second International Symposium on Non-Photorealistic Animation and Rendering (NPAR*, pages 769–776, 2002.

[47] Yann Semet and Fre'do Dur. An interactive artificial ant approach to non-photorealistic rendering. In *Springer-Verlag Lecture Notes in Computer Science*, pages 188–200. Springer Verlag, 2004.

[48] Michio Shiraishi and Yasushi 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*, pages 53–58, New York, NY, USA, 2000. ACM.

[49] Noah Snavely, C. Lawrence Zitnick, Sing Bing Kang, and Michael Cohen. Stylizing 2.5-d video. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pages 63–69, New York, NY, USA, 2006. ACM.

[50] J. Stam. Real-time fluid dynamics for games, 2003.

[51] Scott N. Steketee and Norman I. Badler. Parametric keyframe interpolation incorporating kinetic adjustment and phrasing control. *SIGGRAPH Comput. Graph.*, 19(3):255–262, 1985.

[52] M. Strengert, M. Kraus, and T. Ertl. Pyramid Methods in GPU-Based Image Processing. In *Workshop on Vision, Modelling, and Visualization VMV '06*, pages 169–176, 2006.

[53] Tama's Szira'nyi and Zolta'n To'th. Random paintbrush transformation. *Pattern Recognition, International Conference on*, 3:3155, 2000.

[54] Chih-Hsuan Tzeng, Zhi-Fang Yang, and Wen-Hsiang Tsai. Adaptive data hiding in palette images by color ordering and mapping with security protection. *IEEE Transactions on Communications*, 52(5):791–800, 2004.

[55] Jarke J. van Wijk. Image Based Flow Visualization. *ACM Transactions on Graphics (SIGGRAPH 2002)*, 21(3):745–754, July 2002.

[56] Jarke J. van Wijk. Image based flow visualization for curved surfaces. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 17, Washington, DC, USA, 2003. IEEE Computer Society.

[57] Jue Wang, Yingqing Xu, Heung-Yeung Shum, and Michael F. Cohen. Video tooning. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 574–583, New York, NY, USA, 2004. ACM.

[58] Georges Winkenbach and David H. Salesin. Rendering parametric surfaces in pen and ink. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 469–476, New York, NY, USA, 1996. ACM.

[59] Eugene Zhang, James Hays, and Greg Turk. Interactive tensor field design and visualization on surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):94–107, 2007.

[60] Eugene Zhang, Konstantin Mischaikow, and Greg Turk. Vector field design on surfaces. *ACM Trans. Graph.*, 25(4):1294–1326, 2006.