University of California
Los Angeles

A Framework for Non-Photorealistic 2D Animation

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Jasleen Singh

2011

The thesis of Jasleen Singh is approved.

Petros Faloutsos

Song-Chun Zhu

Demetri Terzopoulos, Committee Chair

University of California, Los Angeles

2011

Table of Contents

List of Figures

Acknowledgments

I would like to take this opportunity to thank all the people who have helped me while working on this thesis.

First and foremost, my sincere gratitude to my advisor, Demetri Terzopoulos, who has been an inspiration throughout with this deep knowledge in so many different areas. He gave me the freedom to pursue my own interests, giving helpful suggestions for potential research topics, and was patient when I couldn't settle on one. This thesis would not have been possible without his guidance.

My thanks to fellow graduate students, who made my time here so much more interesting with their energy and passion. I'd like to especially thank Simardeep Uppal, Manu Jose, Kannan Parameswaran, Sharath Gopal, Ren Rong, Nadya Widjaja, Diana Ford, Shawn Singh and other friends who provided useful feedback and encouragement over the course of this thesis.

Thanks to all the faculty and staff at UCLA, who encouraged me to think independently and gave me the opportunity to explore a variety of topics.

Finally, a big thank you to my sister, for being a friend and confidante, and to my parents, who are the source and strength of my life, and to whom this thesis is dedicated.

Abstract of the Thesis

A Framework for Non-Photorealistic 2D Animation

by

Jasleen Singh

Master of Science in Computer Science

University of California, Los Angeles, 2011

Professor Demetri Terzopoulos, Chair

Non-photorealistic techniques are used in Computer Graphics to render scenes that have an abstract or painterly feel to them. However, most such techniques usually employ a source image as input in order to produce a stylized image as output and they use stochastic methods to distribute strokes or other primitives. This makes it difficult to extend such methods to video and animation, as simply applying the technique to each frame results in a noisy looking sequence, commonly known as the *shower-door* effect.

In this thesis, we present a framework for creating 2D animation and for rendering it using stroke-based non-photorealistic techniques. Objects in a scene to be animated are represented as a hierarchy of triangle meshes, each of which has additional information about color and tone. A generator then distributes particles over each mesh. An object is animated by deforming its component meshes, with the particles maintaining their relative position with respect to the mesh vertices. A renderer then draws the output image given the position of the particles.

We implement our framework in an application, called *LiveCanvas*, for creating stylized 2D animations. The application supports both interpolation (tweened) and key-frame based animation, or a mixture of the two. Since objects are represented as meshes and since the application automatically does color blending between two key-frames, we can also use it to create animated *morphs* rendered in various non-photorealistic styles. The generated frames maintain the temporal coherency of the brush strokes, thus avoiding the shower-door effect, which results in smooth-looking animations. We demonstrate the flexibility of our framework by implementing mesh deformation using three different techniques and providing a number of non-photorealistic styles.

CHAPTER 1

Introduction

From time immemorial, humans have expressed themselves through art. From cave paintings dating 32,000 years ago,[1] to random doodling by a student in a boring lecture, drawing and painting have always been natural activities for humans. In prehistoric times, primarily due to limitations of the tools at their disposal, humans drew abstract forms and images of natural landscapes, animals and other objects of interest, real or imaginary. With improvements in technology over time and a desire to create life-like depictions, artists and painters strove to achieve greater fidelity between the subject and its image. The art of photo-realistic painting was perfected by Renaissance era artists such as Michelangelo and Raphael. Their exquisite command over the paintbrush and their eye for detail yielded works of art that are highly prized and sought after. However, over time, artists started experimenting with other styles of painting, moving back towards abstraction and simplification. This is exemplified in the works of Van Gogh, Monet, and more recently Pablo Picasso. These artists developed their own unique style, which while being abstract and stylized, is able to communicate with and influence the viewer. Figurative imagery can be seen in other fields such as medical illustration, which conveys important aspects of a specimen by omitting extraneous details, or even caricature, which by enhancing certain fea-

[1] See, e.g., *The Cave Painters: Probing the Mysteries of the World's First Artists* by Gregory Curtis, Knoph, New York (2006).

tures and diminishing others, catches the attention of the viewer, often resulting in amusement.

We can see a similar trajectory in the field of computer graphics. With the introduction of early graphics terminals, researchers developed systems to create plain drawings and abstract two and three dimensional worlds. For a long time since, there has been a drive towards photorealism in graphics. This has been a largely successful venture, and one might be hard pressed today to find images around them that have not been touched or enhanced by computer-based tools, if not entirely created using such tools. The entertainment industry has benefited greatly from the advancement in graphics technology and ever increasing computing power, as is evident in the latest blockbuster movies and immersive games. Health is another area that utilizes graphics, in this instance to help doctors visualize internal anatomy and even to perform complex surgery through computer assisted tools. Computer graphics is indispensable in engineering and architecture, where renderings of buildings and equipment can help in visualizing and understanding the final product through virtual walkthroughs and physical simulations before it is physically constructed.

As in painting, so in the field of Computer Graphics, we see a collection of people experimenting with ways to convey information better by abstracting and removing details. Blending ideas from both science and art, this fledgling field, called *Non-Photorealistic Rendering (NPR)*, draws upon human psychology and perception to create imagery that is aesthetically pleasing, useful, and aids in creative expression.

NPR research can be classified according to the general approach to generating the final image:

1. Artistic media simulation,

2. User-assisted image creation, and

3. Automatic image creation.

Media simulation techniques attempt to model the physical properties of the medium being emulated. This involves simulating parts or all of the interacting systems—the medium, such as oil paint or water color, the brush or applicator used to transfer the media, and the substrate that received and distributes the media on a surface. User-assisted image creation attempts to incorporate the skills and techniques of human artists in order to enable non-experts to produce stylized images with reduced effort. Finally, automatic image creation attempts to fully automate the process of creating abstract images (usually using a photograph as input). Though NPR has attracted much interest recently, and there are impressive examples of the use of its techniques in the mainstream, it is still in its early days and the field is still somewhat ad hoc and disorganized.

If we look at the landscape of digital art creation tools, we will find a number of high-end tools for expert users that take lots of time and effort to master. 2D and 3D modeling and animation tools such as Adobe Flash, Maya, or 3D Studio Max are still out of reach of the casual user. There are not that many applications that enable a creative-minded but untrained individual to quickly express themselves through animation.

1.1 Thesis Objective

The goal of this thesis is to develop a framework that combines techniques in non-photorealistic rendering with the purpose of enabling users to create and render 2D stylized animations through an simple and intuitive user interface.

We demonstrate the utility of our framework through a desktop application,

called LiveCanvas, which enables the user to create animations and render them in different styles. Objects in a scene are represented as meshes that can then be manipulated to create a keyframe-based animation. To avoid the "shower-door" effect that would result if we were simply to apply a filter to each frame in the animation, a number of 'particles' are distributed on the mesh and tracked as the mesh is deformed. These particles are then used to attach strokes on the mesh in order to produce the final rendering.

1.2 Thesis Overview

Chapter 2 reviews relevant techniques developed in non-photorealistic rendering research, as well as 2D animation tools, some of which will be integrated within our framework.

Chapter 3 presents our non-photorealistic 2D animation framework, which can be used to create animations in 2D rendered in different artistic styles through the use of pluggable renderers.

Chapter 4 describes our demonstration application, called LiveCanvas, which is made up of three components—a Sketch Creator, a Mesh Editor, and an Animator—that can be employed by the user to move from a concept sketch, to modeling objects in the scene, to creating frames of the animation sequence, and finally to rendering them in different styles.

Chapter 5 present examples of some animations and morphs generated using LiveCanvas.

Chapter 6 concludes the thesis by summarizing its contributions and discussing future work.

CHAPTER 2

Previous Work

We first discuss some of the techniques developed in the area of non-photorealistic rendering (NPR). Specifically, we will focus on techniques that are applicable in 2D.[1] Subsequently, we will review tools for 2D animation and discuss how they may be combined with NPR into a novel framework for creating stylized 2D animation.

2.1 Non-Photorealistic Rendering

Non-photorealistic rendering began as a experimental science when researchers tried to mimic the process that an artist follows to create a new painting. Some methods are based on simulating the actual physics or an approximation of the underlying process by which pigment interacts with a substrate. Others tried to position strokes on an image either relying on the user for guidance or automatically, using a reference image.

[1] Object space methods, usually used in interactive applications, work with 3D geometry [Markosian et al. 1997, Northrup and Markosian 2000] and are beyond the scope of this review. The reader may refer to [Gooch and Gooch 2001] for a more comprehensive overview of non-photorealistic rendering research.

2.1.1 Simulating Artistic Media

Simulation based techniques work by modeling and simulating the applicator, such as pencil or brush, and/or the the movement of color particles through a substrate such as paper or canvas. Sousa and Buchanan [1999] simulated graphite pencil drawing by observing and modeling physical materials (pencil, paper, eraser, blender) and how they interact with each other. Strassman's *Hairy Brushes* simulated the effect of a brush by sweeping an integer array, representing color, along a spline with perturbations based on pressure values [Strassmann 1986]. Small [1991] used cellular automata running on parallel hardware (each pixel is a separate processor) to simulate the movement of pigment and water through paper, which is then rendered.

2.1.2 Painterly Rendering

Haeberli [1990] describes a simple interactive application to create abstract representations of input images. In his system, strokes are placed in the neighborhood of the path along which the mouse is dragged. Each stroke has a number of properties that determine the final outcome: 1) Location, 2) Color, 3) Size, 4) Direction, and 5) Shape. Location is determined by the cursor and color by point sampling the input image at the current location, while the size, direction, and shape can be controlled either manually or procedurally using additional information. Once the strokes are distributed throughout the image, a variety of techniques can be used to render the output image. Using a texture mapped image of a real brush stroke would give it a more painterly feel, or strokes could be aligned perpendicular to the gradient of the image, which accentuates the edges, or finer details and colors could be enhanced by extrapolating beyond the range formed by an image and its blurred version. If 3D geometry information

Figure 2.1: An example from "Paint by Numbers" (©1990 P. Haeberli, reproduced from [Haeberli 1990]).

of the scene is available, surface normals could be used to control the direction of the strokes. All of these allow a user to experiment with different interesting variations. Figure 2.1 shows some images created using this method.

Salisbury et al. [1994] presented an interactive image-based tool for creating pen-and-ink illustrations. Their system uses a grayscale reference image to define shape and tone in the illustration. As the user moves the 'brush' on the canvas, the system adds strokes from either a *stored texture* library or they are procedurally generated. The reference image can be used either as a purely visual aid for the artist or as a source for the system to capture information such as tone, edges, and texture orientation, which directs the placement of strokes. The paper [Salisbury et al. 1996] extended the technique to make it scale independent.

To automatically render video in a painterly style, we need more than simply to apply these techniques to each frame. This is because randomly placing strokes on each frame, or keeping the location fixed and changing the color of the strokes produces a sequence that looks as if it has been shot through a glass shower door, which came be referred to as the *shower-door* effect. Litwinowicz [1997] created a system that uses optical flow fields to maintain temporal coherency of brush

strokes across different frames in a video sequence. His system consisted of three parts:

1. An algorithm to render and clip strokes,
2. An algorithm to produce brush stroke orientations, and
3. An algorithm that uses a gradient-based multi-resolution technique to compute optical flow that is then used to alter the location of the brush strokes. Delaunay triangulation is used to remove strokes that bunch up together as a result of movement, as well as to add new strokes in areas that have a sparse distribution of strokes. Depth ordering of these strokes is maintained from frame to frame to avoid jitter.

One of the drawbacks of the methods discussed so far is that all the strokes that they use are of the same length, which gives the output images a mechanical appearance. However, artists and painters usually start with the bottom-most layer, painting large regions with constant color and incrementally adding details using smaller strokes as the painting progresses. Using this idea, Hertzmann [1998] created a method to "paint" an image with spline brush strokes on a series of layers, with the largest brush size used first, followed by progressively smaller brushes. Given an input image and a set of brush sizes $(r_1, \ldots, r_n)$, a *reference image* is created for each layer by blurring the source image (using a Gaussian kernel with standard deviation proportional to r_i for the i^{th} layer.) For each layer, new brush strokes are placed on areas that differ from the reference image by more than a threshold T. The pseudocode for this operation is given in Algorithm 1. The *paintStroke* method invoked in Line 13 of the Algorithm paints a long spline curve on the canvas beginning at p and along the normal to the gradient of the reference images, ending either when it reaches a maximum stroke length or if

Algorithm 1: Painterly rendering using multi-sized curved brush strokes.

Input: $I_s, I_p, r_1, \ldots, r_n$

Output: Image rendered in a painterly style

foreach *brush size* $r_1, \ldots, r_n$ *from largest to smallest* **do**
- $I_{ref_i} \leftarrow I_s * G_{f_\sigma r_i}$, where G_σ is a Gaussian kernel of standard deviation σ and f_σ is a constant
- $\mathbf{S} \leftarrow$ set of strokes, initially empty
- $grid \leftarrow f_g r_i$
- **foreach** *position* p *on the grid* **do**
  - $M \leftarrow region(p_x - grid/2, p_y - grid/2, p_x + grid/2, p_y + grid/2)$
  - $areaError \leftarrow \sum_{p \in M} \|I_p(p) - I_{ref_i}(p)\|$
  - **if** $areaError > T$ **then**
    - $p \leftarrow \text{argmax}_{p \in M} \|I_p(p) - I_{r_i}(p)\|$
    - $s \leftarrow makeStroke(p, I_p, r_i, I_{ref_i})$
    - add s to $\mathbf{S}$
- **foreach** *stroke* $s \in \mathbf{S}$ *in random order* **do**
  - $paintStroke(s)$

the image color differs from the stroke color more than the canvas color at the point.

The effect of this algorithm can be controlled through a number of parameters such as the *approximation threshold* T that controls whether to place a stroke or not, the brush sizes, the *curvature filter* that is used to limit or exaggerate stroke curvature, the blur factor f_σ, the minimum and maximum stroke lengths (very short strokes produce a pointillistic effect, while longer strokes produce a more expressionistic effect), the opacity, the grid size (f_g), and the color jitter, which

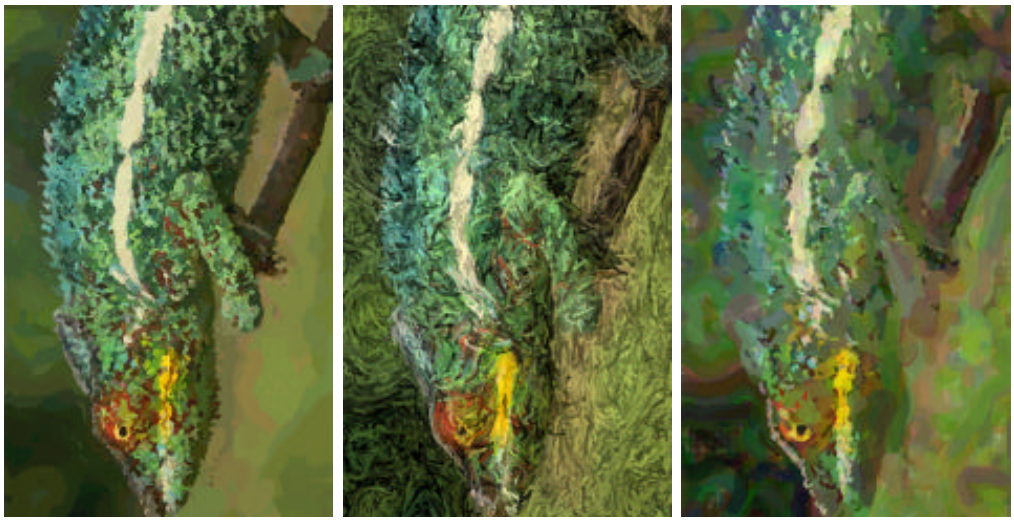


Figure 2.2: Painterly rendering with curved brush strokes of multiple size (©1998 A. Hertzmann, reproduced from [Hertzmann 1998]).

adds randomness to the sampled color at stroke placement sites. Figure 2.2 shows an image to which this method was applied, rendered with different parameter settings. Hertzmann [2003] discusses this and other stroke-based techniques for creating non-photorealistic imagery.

We discussed how Litwinowicz [1997] used optical flow to maintain temporal coherency across frames and eliminate the *shower-door* effect. Another method to avoid jitter was proposed by Meier [1996]; it uses a set of particles distributed over a 3D surface to track the movement of brush strokes as the objects in a scene move. Having the 3D geometry allows the scene to be rendered using different shaders to generate *reference pictures* that encode orientation, brush size, and color information. Each frame is then rendered by compositing brush stroke textures using parametric information obtained from reference pictures at each particle location. To give the frame a painterly look, random perturbations are applied to each particle changing its position, size, or orientation. Since these perturbations are stored with each particle itself, their coherence is maintained

across the entire rendered sequence.

2.2 Particle Generation and Stippling

In [Meier 1996], particles were distributed uniformly across each triangle of the surface mesh. However, this may not always be ideal, especially in cases where conveying continuously varying tone is important. The problem of discretely representing tone has been studied widely in the field of *halftoning*, which tries to reproduce photorealistic representations using a number of closely placed dots. To place points along a single dimension with a probability density function $p(x)$, we first compute the *cumulative density function*

$$C(x) = \int_0^x p(t)\, dt. \tag{2.1}$$

To redistribute a point x_i so that it is more likely to be located in a higher intensity region in the source image, we invert $C(x)$ and calculate the new position $x'_i = C^{-1}(x_i)$. This will cause more points to be located in segments where $p(x)$ is higher, assuming that the input set x is uniformly distributed.

Secord et al. [2002] extended the idea to images with a given 2D probability distribution function (PDF), taking into account the tone that would be generated by the redistribution process. This is necessary as two exactly overlapping primitives contribute the same to the output tone as a single primitive, and so the output tone is not a linear function of the number of primitives. They first calculate the number of primitives needed to convey a particular tone using the input image and then redistribute a uniformly spread set of primitives to match the tone in the input image.

Alternatively, points can be redistributed iteratively. Secord [2002] presented a method to generate *stipple* drawings by distributed points according to the

Algorithm 2: Lloyd's algorithm

Input: Initial distribution of generating points $x_1, \ldots, x_n$

Output: Even distribution of points $x_1, \ldots, x_n$

```
while generating points x_i not converged to centroids do
    Compute Voronoi diagram for x_i
    Compute the centroids C_i using (2.2)
    Move each generating point x_i to its centroid C_i
```

intensity of a source image. Stippling is a process where an image is represented as a number of small dots such that their density gives the impression of tone. In the early days of printing, stippling was especially useful as images could be scaled down without significantly affecting the perceptual quality of the resized image. According to this technique, a Voronoi diagram is generated for a distribution of points in a 2D plane. A Voronoi diagram is a partitioning of a plane into regions such that all points in each region are closest to a point s from a given set S, called the generating point for the cell. After computing the Voronoi diagram, the generating points are redistributed to the centroid of its Voronoi region. This process is repeated until all points converge to their region's centroid. The centroid of a region is defined as

$$\mathbf{C_i} = \frac{\int_A \mathbf{x} p(\mathbf{x}) dA}{\int_A p(\mathbf{x}) dA}, \tag{2.2}$$

where A is the region, $\mathbf{x}$ is the position, and $p(x)$ is the density function. A centroidal Voronoi diagram can be thought of as minimizing the energy function $\int_A p(\mathbf{x}) \|\mathbf{C_i} - \mathbf{x}\|^2$.

A method to iteratively relax a Voronoi partitioning so that the generating points lie on the centroids, known as Lloyd's algorithm is shown in Algorithm 2.

Figure 2.3 shows images generated using the Weighted Voronoi Stippling

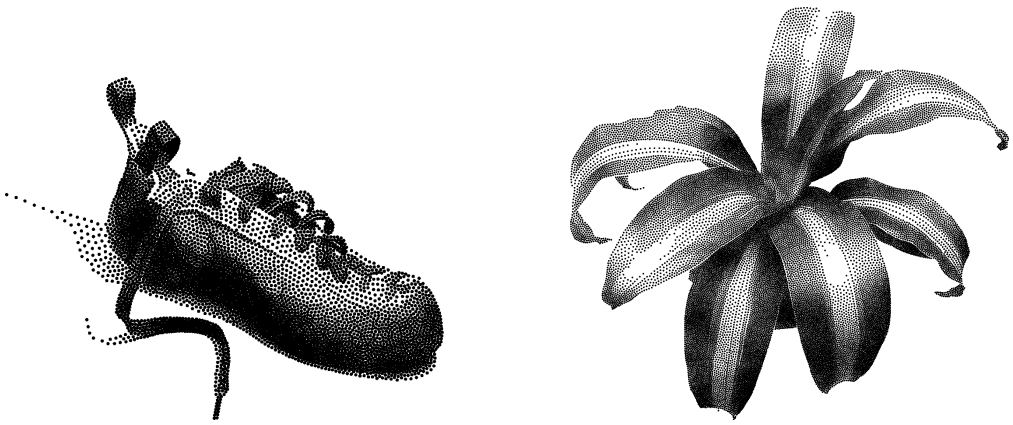


Figure 2.3: Examples of weighted Voronoi stippling (©2002 A. Secord, reproduced from [Secord 2002]).

method of Secord [2002].

2.3 2D Animation Tools

The traditional way of making animated cartoons was very time-consuming and required an innate talent for drawing, as each frame was drawn and painted by hand. This made cartoon animation accessible mostly to large studios with big budgets. Recently, however, computers are being increasingly used to reduce the tedium of inking, painting, and compositing each frame in an animation. Various approaches have been pursued to automate all or part of the cartoon animation production pipeline. We will briefly discuss some research prototypes and commercial applications for 2D animation to get an idea of the kind of tools available to users.

One of the earliest systems for $2\frac{1}{2}$D animation was *Inkwell* by Litwinowicz [1991]. The goal was to create an intuitive and and useful user interface for producing animations quickly with a focus on character animation. The system

provided a number of tools for creating and editing drawings, which could then be animated. In Inkwell, characters were specified using polygons, ellipses, and splines organized into an hierarchy of layers. This allowed changes in a higher layer in the hierarchy to be reflected in its children, providing a higher-level control for making large changes that may be needed. These basic shapes were animated by interpolating parameters using cubic splines. Shapes could be texture mapped using Coons patches and animated by warping the space within as it transformed over time. This allowed a shape to be mapped to different textures that can then be used to animate similar characters using the same underlying shape transformation. Since warping is done globally on a patch, however, it gives the resulting animation a flat look.

Smith and Sederberg [2004] described a user-guided solution to the problem of automatic inbetweening of keyframes using skeletal interpolation. In their system, the user specifies keygrames as colored regions bounded by B-splines. The user also specifies a skeleton for the first frame and attaches control points of the curves to the *bones* in the skeleton. The position of the skeleton in other keyframes is automatically inferred by minimizing the sum of squared error for each bone

$$E = \sum_{i=1}^{n} \|\mathbf{T}\mathbf{p}_i - \mathbf{p}_i^s\|^2, \tag{2.3}$$

where $\mathbf{p}_i^s$ are the control points in the skeletal coordinate system (in the first frame), $\mathbf{p}_i$ are the control points in the new keyframe, and $\mathbf{T}$ is the affine transformation matrix that produces skeletal coordinates for $\mathbf{p}_i$. Once the configuration of the skeleton is known for each keyframe, inbetween frames are generated by linearly interpolating the ends of each bone from which the position of the control points is eventually derived. The ends of the bones can be specified to move along an arc instead of a line to add more *life* to characters.

Another tool, called *CharToon*, that uses skeleton-based animation, but for faces, was described by Ruttkay and Noot [2000]. Their system allowed a user to model a face as a $2\frac{1}{2}$D drawing that could then be animated and rendered in different non-photorealistic styles. In CharToon, a user first creates a face by assembling it from a set of *basic components*. Each component is defined using control points on the skeleton and other points that define the shape of the component. These other points move in relation to the control points (maintaining the same relative distance and orientation, at a constant ratio from an edge or are even fixed). One can create different designs for each part of the face to create a *feature repertoire*, each of which can then be varied by adjusting the control points to form an *expression repertoire*. By interpolating the control points through time and then rendering the shapes in different styles, CharToon enabled a user to create non-photorealistic face animation.

This shape blending approach can also be seen in commercial tools such as Adobe Flash and Toon Boom Animate, which allow interpolation between shapes using either just the boundary stroke, or by specifying a skeleton and using inverse kinematics to manipulate the joints (and also deforming the shape attached to each bone). However, this often results in animations that look stiff and mechanical and are difficult to control precisely as the shapes move and bend in ways defined by the application.

2.4 Summary

We have reviewed the non-photorealistic rendering literature that is relevant to our work. Whereas some techniques try to model the artistic medium and tools and they achieve non-photorealistic effects by running a simulation, others place strokes on a canvas using some heuristic, as well as an input image to create a

stylized output image. Our framework is more suitable for the latter category of techniques, collectively known as stroke-based non-photorealistic rendering. We also described a technique for painterly rendering using multi-sized curved brush strokes. We then reviewed some methods for distributing particles in a region, which if guided by an input image can be used to generate a stippled version of the image. Finally, we reviewed relevant software applications for 2D animation and discussed why a new tool is needed that can render smooth-looking animations in a non-photorealistic style. Next, we will present our non-photorealistic 2D animation framework, which can be used to create animations in 2D rendered in different artistic styles using pluggable *renderers*.

CHAPTER 3

Non-Photorealistic 2D Animation Framework

We now present our framework for creating non-photorealistic 2D animation. We decouple the motion of an object within a scene from its rendering, so that the same underlying object animation can be rendered in different artistic styles. In our framework, objects are represented as polygonal meshes that have *particles* associated with them. An animation is created by deforming the regions covered by a mesh using *control points* on the mesh. Objects are then rendered using the particles as sites for placing strokes. By varying the behavior of the different components in the framework, we can achieve different styles of non-photorealistic 2D animation.

3.1 Framework Components

We will now describe the different components in the framework and how they interact with each other to produce the final rendered animation.

Figure 3.1 illustrates the different components in the NPR animation framework, which include the following:

1. **Background References**: Many non-photorealistic techniques use an image as input to create an artistic rendering as the output image. Images may also be used to convey tone information, which is employed in many

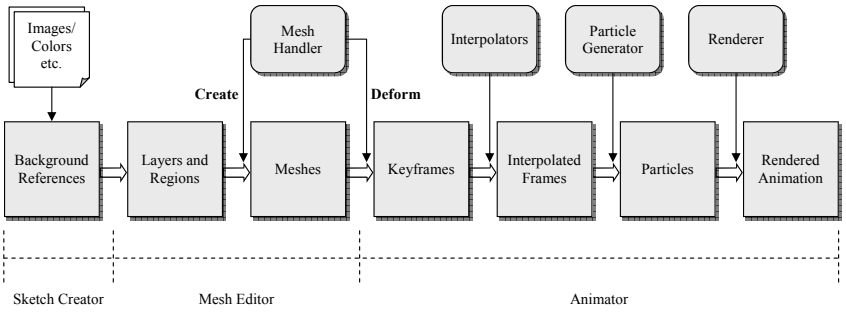


Figure 3.1: Framework overview

halftoning techniques, or to create a stippled version of the input image. In our framework, we provide the option to use images to provide color and tone, though other methods (such as assigning a fixed color or a gradient) may be used as well. An image or a solid color used in rendering is generically referred to as a *background reference*. Each layer can have multiple background references, and each key-frame can select the background reference to use. If two adjacent key-frames have difference background references, colors are automatically blended for intermediate interpolated frames. Layers and key-frames are explained later in this list.

2. **Layers and LayerGroups**: We represent a scene as a composition of stacked layers, a common approach for 2D drawing and animation tools. Each layer may have multiple sub-layers, thus forming a tree-like structure. The layers are rendered from bottom to top (ordering among sibling layers), so objects higher up in the stack are rendered on top of those below. A LayerGroup is a set of two or more layers that are transformed as a single unit. A binary operation (*join, intersect, or subtract*) may be applied to a LayerGroup containing two layers, which creates a new region from the regions contained in the grouped layers.

3. **Regions**: Each layer (including the background layer) may contain a *region*, which represents a boundary around an object feature. A region is a stored as a polygon with the length of each side fixed. Each region has one or more associated *background references*. A region queries its background reference to obtain the color of each individual pixel contained within it. Thus, a background reference backed by an image may serve to draw an arbitrary texture within a region. Alternatively, a background reference backed by a solid color or a gradient would paint the region with a fixed or linearly varying color.

4. **Meshes**: The area within each region is subdivided into a uniform mesh composed of triangles. Each mesh contains an array of vertices and an array of triangles, where each triangle contains pointers to 3 vertices. Any vertex may be set as a *control point*, which is used by the *Mesh Handler* to deform the mesh.

5. **Mesh Handler**: As soon as the user specifies a region around an object feature, the Mesh Handler subdivides it into a uniform mesh. It also provides methods to deform the mesh using its control points. By using different implementations of Mesh Handlers, we can change what the generated mesh looks like and how it deforms.

6. **Key-frames**: Once a scene has been specified as a set of layers containing regions subdivided into meshes with control points, we can proceed with the creation of key-frames for animation. To change the shape of a region, the control points are moved around, which deforms the mesh by adjusting the positions of the other unconstrained vertices. Alternatively, global transformations such as translation, rotation, or scaling can be applied to each vertex in the mesh. If a region has multiple associated background

references, the user can select which one to use. Colors are automatically blended between the background references of two adjacent key-frames. The user can create the entire animation using only key-frames, or can use interpolation between two adjacent key-frames to create intermediate frames using an Interpolator. Each key-frame has its own interpolator.

7. **Interpolator**: An Interpolator controls how the control-point positions or transformations related to a mesh should vary in the intermediate frames from one key-frame to the next. Using interpolators, we can create different types of tweened animation, such as "Ease-in", "Ease-out", etc., which is important to give the animation more life and character.

8. **Interpolated Frames**: Interpolated frames are generated by interpolating between two key-frames, using the interpolator attached to the key-frame on which the sequence ends.

9. **Particles**: After all the key-frames and intermediate key-frames have been created, we are ready to render the animation. To do this, we first generate a set of *particles* to be distributed on the surface of each mesh. Each particle contains all the information that is needed to generate a single stroke in the output image. A particle is essentially a 2D coordinate with pointers to its parent mesh and the containing triangle. In addition, each particle contains a generic pointer to any *packet* of information that a particular style may want to store. When a triangle within the mesh is stretched or moved, the particles on its surface change location as well.

   By separating the rendering process into stroke placement (via particles) and stroke drawing, we can avoid the "shower-door" effect that would be evident if we placed strokes randomly in each frame. This approach is

similar to that by Meier [1996] with the difference that we do not associate brush stroke attributes with each particle. So particles in our framework are more analogous to tracking feature points. This gives tremendous flexibility to someone implementing a particular style—they have complete control over how many particles to generate, where initially to place them on the mesh and how to render them. This allows the application of a number of techniques using randomly drawn primitives described in the literature to a sequence of frames without losing frame-to-frame coherency.

10. **Particle Generator**: A Particle Generator is responsible for creating a number of particles on the surface of a mesh. The generated particles are displaced relative to the deformation of the mesh and passed on to the Renderer for the final rendering of a frame. Particles are generated only once for the entire animation sequence and stored for use in all the frames. For multi-pass techniques such as the one by Hertzmann [1998], we generate and store particles once for each pass.

11. **Renderer**: A renderer is the final piece our framework's pipeline. At this stage, we have all the information we need to color the pixels on the render surface (usually an image buffer). The renderer is passed a pointer to the surface along with all the particles that need to be rendered. For multi-pass rendering techniques, we call the renderer until it signals that it is done.

We now discuss the mesh generation and deformation techniques, which are used to manipulate the shape of objects in each key-frame. This will conclude the description of the theoretical basis of the techniques used in the application of our framework, which will be described in the next chapter.

Algorithm 3: Generate random points within a polygon

Input: Points $x_1, \ldots, x_n$ on the polygon; g number of points to be generated

Output: Generated random points $y_1, \ldots, y_g$ within the polygon

1 Construct Delaunay Triangulation $(t_1, \ldots, t_T)$ for $(x_1, \ldots, x_n)$ where T is the total number of generated triangles

2 Create cumulative array $\mathbf{A} = (A_1, \ldots, A_i, \ldots, A_T)$ such that $A_i = A_{i-1} + Area(t_i)$ for $i \in (1, \ldots, T)$ and $A_0 = 0$

3 $totalArea = \sum_{i=1}^{T} Area(t_i)$

4 **for** $i = 1$ **to** g **do**

5 $\quad N \leftarrow Random(0, totalArea)$

6 $\quad$ find minimum j such that $A_j \geq N$

7 $\quad$ set y_i to a random point in t_j using (3.2)

3.2 Mesh Generation

To create the mesh, first a number of points proportional to the area of the region are created. Next, they are distributed uniformly within the region. If the region is a triangle, we can generate a random point within the triangle $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2)$ by generating two random numbers $u, v \in Random(0, 1)$, then computing

$$u', v' = \begin{cases} u, v & \text{if u + v} \leq 1; \\ 1 - u, 1 - v & \text{if u + v} > 1, \end{cases} \tag{3.1}$$

and finally

$$\mathbf{v}_{random} = \mathbf{v}_0 + u'.(\mathbf{v}_1 - \mathbf{v}_0) + v'.(\mathbf{v}_2 - \mathbf{v}_0) \tag{3.2}$$

For a general polygon shape, we use Algorithm 3.

Finally, Constrained Delaunay Triangulation followed by Lloyd's Voronoi relaxation (see Section 2.2) is used to distribute the vertices of the mesh evenly

Algorithm 4: Mesh generation

Input: Points $v_1^p, \ldots, v_P^p$ on the polygon $\mathbf{V}^p$ enclosing the region; Vertex density D_{mesh} of the mesh

Output: Generated mesh ($\mathbf{V} = (v_1, \ldots, v_V)$, $\mathbf{T} = (t_1, \ldots, t_T)$)

1 $G \leftarrow Area(\mathbf{V}^p) * D_{mesh}$

2 Generate G random points $(v_1^g, \ldots, v_G^g)$ using Algorithm 3

3 Use Lloyd's method (see Algorithm 2) to more evenly distribute generated points

4 $\mathbf{V} \leftarrow (v_1^p, \ldots, v_P^p, v_1^g, \ldots, v_G^g)$

5 Create Constrained Delaunay Triangulation $\mathbf{T}'$ with all edges on the input polygon as constraints

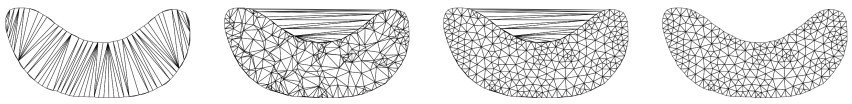
6 Remove triangles that fall outside the polygon in $\mathbf{T}'$ to create $\mathbf{T} = (t_1, \ldots, t_T)$ where T is the total number of triangles in the mesh

7 $Mesh = (\mathbf{V}, \mathbf{T})$

throughout the area of the region. A mesh is represented as a collection of V vertices $(v1, \ldots, v_V)$ and T triangles $(t_1, \ldots, t_T)$, and each triangle contains pointers to 3 vertices. Algorithm 4 outlines the steps.

Figure 3.2 shows the sequence of steps in the generation of a mesh from a given polygonal path: (a) Triangulation of the points on the polygon—points are randomly generated by first selecting a triangle with probability proportional to its area and then generating a random point in it; (b) triangulation of the generated points before relaxation; (c) triangulation after the application of Lloyd's algorithm; (d) generating the final mesh after removing triangles outside the input polygon.

The Constrained Delaunay Triangulation in Step 5 of Algorithm 4 may create triangles that fall outside the polygon (see Figure 3.2c). We remove those



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

Figure 3.2: Generating a mesh from a polygonal path

triangles in Step 6.

3.3 Mesh Deformation

Mesh deformation has been widely studied in the field of computer graphics, and it is used in a variety of modeling and animation tools. The input to this step is a triangle mesh with a number of control points (or pairs of control points representing the end points of line segments). The goal is to find the coordinates of the remaining free vertices as the user changes the location of the control points. Over time, a lot of methods have been proposed that strive to produce aesthetically pleasing or natural looking deformations as easily as possible. Most of these techniques have been developed for 3D meshes. A related technique that operates in 2D is called morphing, which is the visual transformation of one image into another, usually by warping the space containing the source image onto the target image so that important feature points correspond.

We discuss three techniques for deforming 2D meshes: 1) Thin-Plate Splines, 2) Feature-Based Mesh Deformation, and 3) As-Rigid-As-Possible Shape Manipulation. Each has its own advantages and disadvantages, depending on the type of animation. Thin-Plate Splines and Feature-Based Mesh Deformation work well when the outer shape is more or less preserved, thus making them useful

for morphing. Finally, the As-Rigid-As-Possible Shape Manipulation technique gives good results if we also want to change the bounding shape, and it is more useful for cartoon and character animation. Note that all three techniques have closed-form solutions, unlike some physically-based and iterative techniques (e.g., [Terzopoulos and Vasilescu 1991]).

3.3.1 Thin-Plate Splines

Thin-Plate Splines (TPS) [Chui 2001] is a warping technique that generates a smooth spatial mapping from one set of points to a second set of corresponding points, also called control points as they are used to control the deformation. We define an energy function for the mapping, given a set of corresponding points, which we then try to minimize. For TPS, the space integral of the square of the second order derivatives of the mapping function is used:

$$E = \int\int \left(\frac{\partial^2 f}{\partial x^2}\right)^2 + 2\left(\frac{\partial^2 f}{\partial xy}\right)^2 + \left(\frac{\partial^2 f}{\partial y^2}\right)^2 \mathrm{d}x\,\mathrm{d}y, \tag{3.3}$$

where $f(x, y)$ is the mapping function. For a 2D coordinate transformation, two splines ($f_x(x, y)$ and $f_y(x, y)$) are used, the first representing displacement in the x direction and the second representing displacement in the y direction. The solution to the energy minimization problem is of the form

$$f_x(x, y) = a_0 + a_x x^s + a_y y^s + \sum_{i=1}^{P} w_i U(|(x_i^s, y_i^s) - (x^s, y^s)|), \tag{3.4}$$

and similarly for $f_y(x, y)$, where $U(r) = r^2 \log(r)$ and $\sum_{i=1}^{P} \phi_i w_i = 0$ with $\phi_i = \{1, x_i^s, y_i^s\}$. Written in matrix form, we have

$$\begin{bmatrix} \mathbf{K} & \mathbf{P} \\ \mathbf{P}^T & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{w} \\ \mathbf{a} \end{bmatrix} = \begin{bmatrix} \mathbf{x}^d \\ \mathbf{0} \end{bmatrix}, \tag{3.5}$$

where $K_{ij} = U(|(x_i^s, y_i^s) - (x_j^s, y_j^s)|)$ and the i^{th} row of $\mathbf{P}$ is $[1, x_i^s, y_i^s]$, and vector $\mathbf{x}^d$ contains the x-coordinates of the transformed control points (in the destination

mesh). We invert the matrix on the left and multiply it with the right hand side to obtain the x components of $\mathbf{w}$ and $\mathbf{a}$. A similar computation is done to obtain their y components.

We then use these parameters in (3.4) to derive the transformation that maps the other points in the source mesh to the destination.

3.3.2 Feature-Based Mesh Deformation

The Feature-Based Mesh Deformation technique, described in [Beier and Neely 1992], is "based upon fields of influence surrounding two-dimensional control primitives". A set of lines in the source mesh and their corresponding positions in the destination mesh is input by the user. Lines are specified by pairs of vertex positions $\mathbf{p}$ and $\mathbf{q}$. Thus, $\overrightarrow{\mathbf{p}^s\mathbf{q}^s}$ specifies a line in the source mesh, while $\overrightarrow{\mathbf{p}^d\mathbf{q}^d}$ specifies a line in the destination mesh. Given a pair of corresponding lines, a vertex in the source mesh $\mathbf{x}^s$ is mapped onto the destination as follows:

$$\mathbf{x}^d = \mathbf{p}^d + u(\mathbf{q}^d - \mathbf{p}^d) + \frac{v \perp (\mathbf{q}^d - \mathbf{p}^d)}{\|\mathbf{q}^d - \mathbf{p}^d\|}, \tag{3.6}$$

where

$$u = \frac{(\mathbf{x}^s - \mathbf{p}^s) \cdot (\mathbf{q}^s - \mathbf{p}^s)}{\|\mathbf{q}^s - \mathbf{p}^s\|^2}, \qquad v = \frac{(\mathbf{x}^s - \mathbf{p}^s) \cdot \perp (\mathbf{q}^s - \mathbf{p}^s)}{\|\mathbf{q}^s - \mathbf{p}^s\|}, \tag{3.7}$$

and $\perp(\cdot)$ is the vector that is perpendicular to (i.e., rotated 90^{deg} anti-clockwise) and of the same length as the input vector. The value u is the position along the line and v is the distance from the line. The value of u is between 0 and 1 if the pixel $\mathbf{x}^s$ falls in the rectangular plane bounded by $\mathbf{p}^s$ and $\mathbf{q}^s$.

If we have multiple pairs of lines, the coordinate transformation for each pair is calculated and a weight is assigned to each. The final coordinates are calculated by adding the weighted average of displacements $\mathbf{d} = \mathbf{x}^d - \mathbf{x}^s$ to the original point

Algorithm 5: Feature-based mesh deformation

for *each vertex* $\mathbf{x}^s$ *in the source mesh* **do**
  $\mathbf{d}_{sum} \leftarrow (0, 0)$
  $weightsum \leftarrow 0$
  for *for each line* $(\mathbf{p}^s, \mathbf{q}^s)$ **do**
    calculate u, v per (3.7)
    calculate $\mathbf{x}_i^d$ using u, v and $\overrightarrow{\mathbf{p}_i^d\mathbf{q}_i^d}$ per (3.6)
    calculate displacement $\mathbf{d}_i \leftarrow \mathbf{x}_i^d - \mathbf{x}_i^s$
    $dist \leftarrow$ shortest distance from $\mathbf{x}_i^s$ to $\overrightarrow{\mathbf{p}_i^s\mathbf{q}_i^s}$
    $weight \leftarrow \left(\frac{length^p}{a+dist}\right)^b$
    $\mathbf{d}_{sum} \leftarrow \mathbf{d}_{sum} + \mathbf{d}_i * weight$
    $weightsum \leftarrow weightsum + weight$
  $\mathbf{x}^d \leftarrow \mathbf{x}^s + \frac{d_{sum}}{weightsum}$

in the source mesh $\mathbf{x}^s$. The weight is calculated as

$$weight = \left(\frac{length^p}{a + dist}\right)^b, \tag{3.8}$$

where *length* is the length of the line, *dist* is the distance from the pixel to the line, a controls the smoothness of the warping (a larger value of a decreases the effect of distance of the pixel from the line), b determines how the relative strength of different lines falls off with distance and p, typically 0 or 1, controls how the relative weight of each line changes with the length of the line—if it is 0, lines with different lengths all affect the weight equally. Algorithm 5 specifies the multiple line algorithm.

Thus, this method can be used to warp one mesh onto another by treating every consecutive pair of vertices as a line and providing a corresponding set of

lines in each mesh.

3.3.3 As-Rigid-As-Possible Shape Manipulation

As opposed to the space warp techniques discussed earlier, Igarashi et al. [2005] use a two-step process to minimize the deformation of each individual triangle in a mesh. This technique defines the error function as a quadratic so that it can be minimized using well known linear algebra methods. Instead of a single quadratic error function that represents overall distortion, this technique solves the problem in two steps:

1. Generate an intermediate result by minimizing an error function that allows only rotation and uniform scaling, and
2. Generate the final result by adjusting the scale of each triangle

We will now describe each step in detail.

3.3.3.1 Scale Free Construction

This step generates a set of intermediate triangles using only rotation and uniform scaling. The input is the initial and final configurations of a set of constrained vertices and the output is the final configuration of the remaining free vertices.

The error function for an intermediate deformed triangle (Step 1) $\mathbf{v}'_0$, $\mathbf{v}'_1$, $\mathbf{v}'_2$ is calculated as follows (see Figure 3.3): First, given the coordinates of the corresponding triangle in the initial configuration $\mathbf{v}^s_0$, $\mathbf{v}^s_1$, $\mathbf{v}^s_2$, compute the relative coordinates (x_{01}, y_{01}) of $\mathbf{v}^s_2$ in the local coordinate frame defined by $\mathbf{v}^s_0$ and $\mathbf{v}^s_1$, as follows:

$$\mathbf{v}^s_2 = \mathbf{v}^s_0 + x_{01}(\mathbf{v}^s_1 - \mathbf{v}^s_0) + y_{01} \perp (\mathbf{v}^s_1 - \mathbf{v}^s_0). \tag{3.9}$$

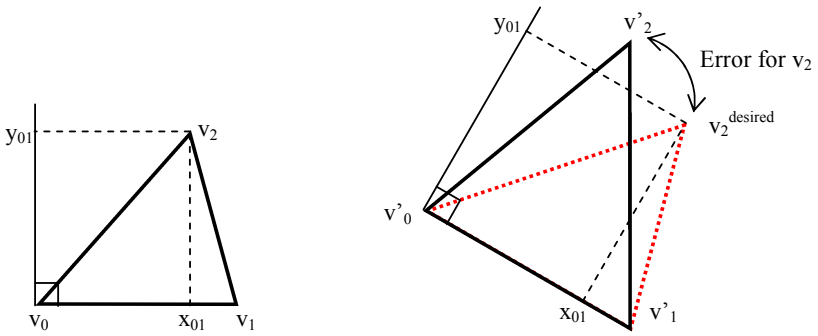


Figure 3.3: Deformed triangle obtained in Step 1, and the error with respect to the desired triangle, which is similar to the original triangle after matching v_0^s to v_0' and v_1^s to v_1'

The desired location of $\mathbf{v}_2^{desired}$ in terms of $\mathbf{v}_0'$ and $\mathbf{v}_1'$ is then

$$\mathbf{v}_2^{desired} = \mathbf{v}_0' + x_{01}(\mathbf{v}_1' - \mathbf{v}_0') + y_{01} \perp (\mathbf{v}_1' - \mathbf{v}_0'). \tag{3.10}$$

The error associated with $\mathbf{v}_2^{desired}$ is defined as the square of the distance from the actual position in the destination mesh:

$$E_{1v_2'} = \|\mathbf{v}_2^{desired} - \mathbf{v}_2'\|^2. \tag{3.11}$$

The total error for the mesh is then the sum of the errors of each individual triangle. We can express the total error in matrix form as

$$E_1 = \mathbf{v}'^T \mathbf{G} \mathbf{v}', \tag{3.12}$$

where $\mathbf{v}' = \{v_0'x,\, v_0'y,\, \ldots,\, v_n'x,\, v_n'y\}$.

To minimize this error, we take the partial derivative of E with respect to the free variables $\mathbf{f} = \{f_0x, f_0y, \ldots, f_mx, f_my\}$ and set it to zero. Putting the free

variables first, we can write

$$\mathbf{v}' = \begin{bmatrix} \mathbf{f} \\ \mathbf{c} \end{bmatrix}, \tag{3.13}$$

where $\mathbf{c}$ are the control points. Thus, (3.12) can be written as

$$E_1 = \mathbf{v}'^T \mathbf{G} \mathbf{v}' = \begin{bmatrix} \mathbf{f}^T & \mathbf{c}^T \end{bmatrix} \begin{bmatrix} \mathbf{G}_{00} & \mathbf{G}_{01} \\ \mathbf{G}_{10} & \mathbf{G}_{11} \end{bmatrix} \begin{bmatrix} \mathbf{f} \\ \mathbf{c} \end{bmatrix}, \tag{3.14}$$

and

$$\frac{\partial E_1}{\partial \mathbf{f}} = (\mathbf{G}_{00} + \mathbf{G}_{00}^T)\mathbf{f} + (\mathbf{G}_{01} + \mathbf{G}_{10})^T \mathbf{c} = \mathbf{0}, \tag{3.15}$$

giving us

$$\mathbf{G}'\mathbf{f} + \mathbf{B}\mathbf{c} = \mathbf{0}, \tag{3.16}$$

and finally

$$\mathbf{f} = \mathbf{G}'^{-1}\mathbf{B}\mathbf{c} \tag{3.17}$$

Thus, the free vertices $\mathbf{f}$ can be obtained for any set of control points $\mathbf{c}$ by simply multiplying with $\mathbf{G}'^{-1}\mathbf{B}$, which remains fixed. This step is very fast and runs at interactive rates; i.e., the user can see the deformation in real time as the location of the control points is changed. As this step may cause changes in scale that the error function in (3.12) does not take into account, a further scale adjustment step is required.

3.3.3.2 Scale Adjustment

The intermediate triangles obtained in the first step are now scaled to generate the final result. This is done by first fitting the original triangles within the intermediate triangles and then scaling the fitted triangles toward the intermediate triangles.

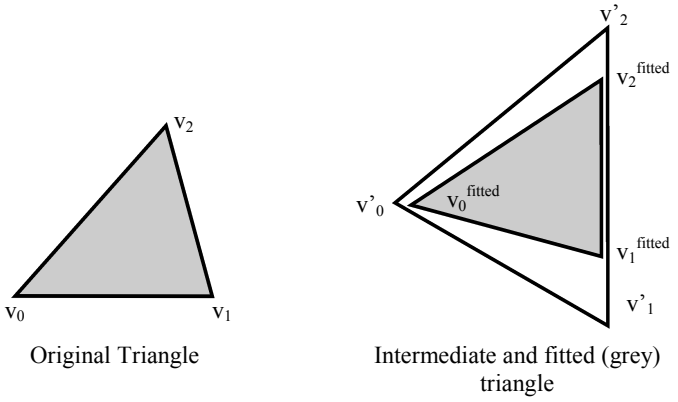


Figure 3.4: Original triangle fitted to the intermediate triangle

To fit an original triangle within its intermediate triangle $\mathbf{v}'_0$, $\mathbf{v}'_1$, $\mathbf{v}'_2$, we must find a triangle congruent to the original triangle $\mathbf{v}_0^{\text{fitted}}$, $\mathbf{v}_1^{\text{fitted}}$, $\mathbf{v}_2^{\text{fitted}}$ (see Figure 3.4), such that the error

$$E_{f(v_0^{\text{fitted}}, v_1^{\text{fitted}}, v_2^{\text{fitted}})} = \sum_{i=0,1,2} |\mathbf{v}_i^{\text{fitted}} - \mathbf{v}'_i|^2 \tag{3.18}$$

is minimal. Using relative coordinates x_{01} and y_{01} defined in Section 3.3.3.1, and expressing $\mathbf{v}_2^{\text{fitted}}$ using $\mathbf{v}_0^{\text{fitted}}$ and $\mathbf{v}_1^{\text{fitted}}$, we have

$$\mathbf{v}_2^{\text{fitted}} = \mathbf{v}_0^{\text{fitted}} + x_{01}(\mathbf{v}_1^{\text{fitted}} - \mathbf{v}_0^{\text{fitted}}) + y_{01} \perp (\mathbf{v}_1^{\text{fitted}} - \mathbf{v}_0^{\text{fitted}}). \tag{3.19}$$

The error in (3.18) is minimized by setting the derivative with respect to the free variables $\mathbf{w} = (v_{0x}^{\text{fitted}}, v_{0y}^{\text{fitted}}, v_{1x}^{\text{fitted}}, v_{0y}^{\text{fitted}})^T$ to zero:

$$\frac{\partial E_f}{\partial \mathbf{w}} = \mathbf{F}\mathbf{w} + \mathbf{C} = \mathbf{0}. \tag{3.20}$$

Thus,

$$\mathbf{w} = \mathbf{F}^{-1}\mathbf{C}. \tag{3.21}$$

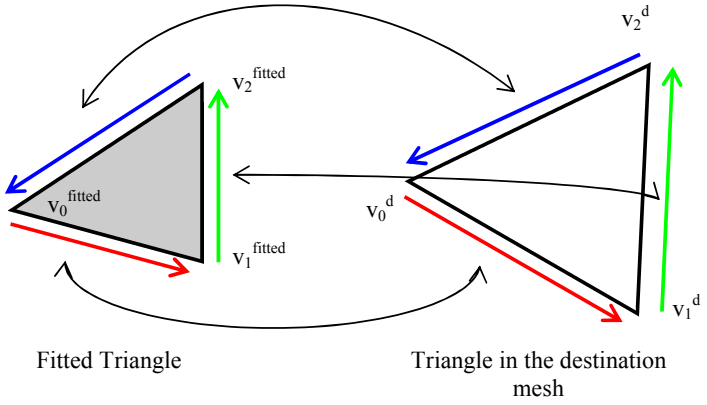


Figure 3.5: Corresponding edges in the fitted triangle and the final triangle in the destination mesh. The error metric used in Step 2 tries to minimize the edge vector difference with the constraint that vertices have the same connectivity as the original mesh.

Since $\mathbf{F}^{-1}$ is fixed for a mesh, we calculate it initially when the mesh is defined. Matrix $\mathbf{C}$ is obtained from Step 1. The fitted triangle is made congruent by multiplying by $|\mathbf{v}_1^s - \mathbf{v}_0^s|/|\mathbf{v}_1^{\text{fitted}} - \mathbf{v}_0^{\text{fitted}}|$.

Now we have individual fitted triangles, but we still need to adjust the vertices so that they form a connected mesh as was originally defined. This is done by adjusting the edges (see Figure 3.5) by minimizing the error

$$E_{2(v_0^d, v_1^d, v_2^d)} = \sum_{(i,j)\in\{(0,1),(1,2),(2,0)\}} |(\mathbf{v}_i^d - \mathbf{v}_j^d) - (\mathbf{v}_i^{\text{fitted}} - \mathbf{v}_j^{\text{fitted}})|^2. \tag{3.22}$$

Since a vertex v^d occurs in multiple triangles, the final position is an average of its position in the corresponding fitted triangles. Writing the error for the entire mesh as

$$E_2 \;=\; [\mathbf{v}^d]^T \mathbf{H} \mathbf{v}^d + \mathbf{b}\mathbf{v}^d + \mathbf{r} \tag{3.23}$$

$$= \begin{bmatrix} \mathbf{f}^T & \mathbf{c}^T \end{bmatrix} \begin{bmatrix} \mathbf{H}_{00} & \mathbf{H}_{01} \\ \mathbf{H}_{10} & \mathbf{H}_{11} \end{bmatrix} \begin{bmatrix} \mathbf{f} \\ \mathbf{c} \end{bmatrix} + \begin{bmatrix} \mathbf{b}_0^T & \mathbf{b}_1^T \end{bmatrix} \begin{bmatrix} \mathbf{f} \\ \mathbf{c} \end{bmatrix} + \mathbf{r} \quad (3.24)$$

and minimizing by setting its partial derivative with respect to the free vertices $\mathbf{f}$ to zero, we obtain

$$\frac{\partial E_2}{\partial \mathbf{f}} = (\mathbf{H}_{00} + \mathbf{H}_{00}^T)\mathbf{f} + (\mathbf{H}_{01} + \mathbf{H}_{10})^T\mathbf{c} + \mathbf{b}_0 = \mathbf{0}, \quad (3.25)$$

giving us

$$\mathbf{H}'\mathbf{f} + \mathbf{D}\mathbf{c} + \mathbf{b}_0 = \mathbf{0}, \quad (3.26)$$

where $\mathbf{H}'$ and $\mathbf{D}$ are fixed, but $\mathbf{q}$ and $\mathbf{f}_0$ change with the position of the control points. Note that since the x and y components are mutually independent in $\mathbf{H}'$, we can apply (3.26) to each component separately.

3.4 Summary

We described the design of our 2D non-photorealistic animation framework, the different components that make up the framework and how they interact with each other. We also discussed techniques used to generate and deform meshes that are used in the implementation of our framework. Our implementation provides a user-interface that animators can use to create different styles of animation by changing the components within the framework, as we will discuss next.

CHAPTER 4

LiveCanvas

Our demonstration application, called LiveCanvas, is made up of 3 components—the *Sketch Creator*, the *Mesh Editor*, and the *Animator*—integrated into a single application. These components can be employed by the user to move from a concept sketch, to modeling objects in the scene, to creating frames of the animation sequence, and finally to rendering them in different styles:

1. **Sketch Creator**: Includes tools to quickly create a sketch that serves as a reference image for the creation of meshes for objects in the scene. The sketch is also used to provide tone and color information for the non-photorealistic rendering of the animation sequence.

2. **Mesh Editor**: Imports the sketch created in the previous step to serve as reference for creating distinct regions for different moving parts of objects in the scene. For example, a man walking would be segmented into separate regions for the torso, head, and limbs. These paths are automatically converted into meshes with an approximately uniform distribution of vertices within each region. The mesh can then be saved to a file.

3. **Animator**: Imports the mesh created in the Mesh Editor so that it can be manipulated to create a sequence of keyframes for the animation. Once the sequence is ready, the user can select a rendering style and press a button to render the frames as a movie.

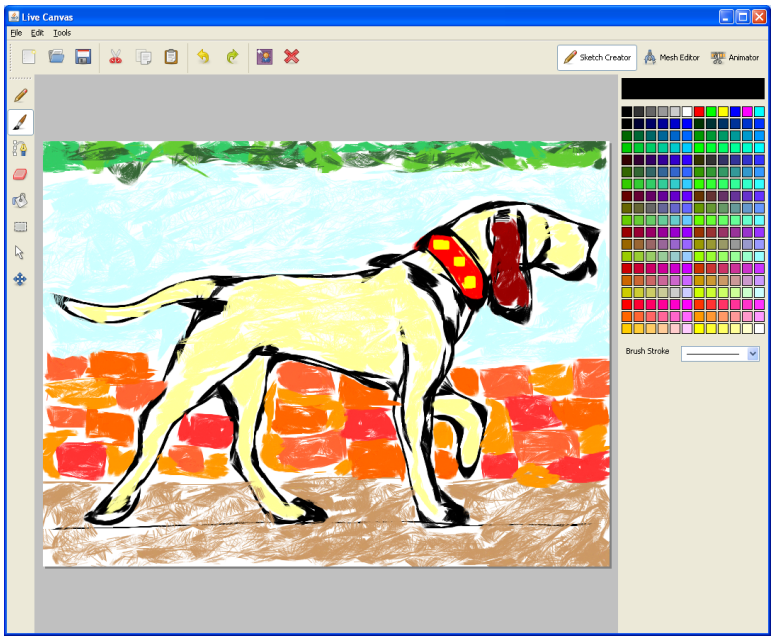


Figure 4.1: The Sketch Creator

Each component, which we will describe in detail in the following three sections, is presented to the user as a *perspective* (content within the main window), and is accessible through a toggle button in the application.

4.1 Sketch Creator

Figure 4.1 shows a screenshot of the Sketch Creator perspective within LiveCanvas. The Sketch Creator provides a simple interface to rapidly sketch out an idea of a scene, offering quick way to create a rough drawing of the scene that the user has in mind. It provides tools to draw on a canvas, such as a *Pencil*, which draws in a sketchy style, or a *Brush*, which draws thick lines, or a *Pen*, which draws cubic curves. Other standard editing tools, such as *Bucket Fill* and *Erase*, are also provided. Some of these tools also have associated parameters accessible through the user interface, such as Pencil *intensity* (higher intensity

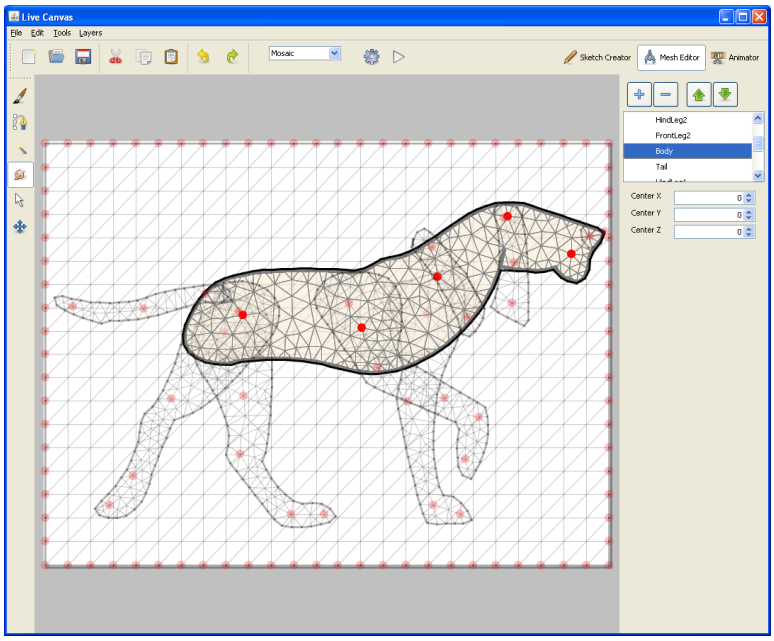


Figure 4.2: The Mesh Editor

creates denser strokes) or line thickness for the Pen tool. Additionally, the user can resize or pan the canvas to draw more comfortably. The sketch produced is also serves in providing tone and color information when rendering the output animation. Though more features could be added, we refrained from doing so as any of the more capable commercial and free painting applications (e.g., Adobe Photoshop or Gimp) may be used for the same purpose. The Sketch Creator enables a drawing to be exported as an image.

4.2 Mesh Editor

Figure 4.2 shows a screenshot of the Mesh Editor perspective within LiveCanvas. The Mesh Editor is used to create the *regions* that make up an object. The *Pointer* tool can be used to select and transform regions. The types of transformations currently supported include translation, rotation, and scaling. Much

like other $2\frac{1}{2}$D animation tools, LiveCanvas enables a user to build a character as a hierarchy of layers, with the bottom-most layer drawn first. A number of tools provided in the Mesh Editor can be used to create different parts of a character. Using the *Brush* tool, an experienced user can simply draw the regions by hand. The *Pen* tool can be used to create regions as piece-wise cubic splines. If required, a *reference image* can be imported, which is displayed over the canvas as a semi-transparent overlay. This can be used to trace over the drawing creating in the first step, and is analogous to the clean-up process used in traditional animation (see [Williams 2009]) where a rough drawing is traced onto a new sheet of paper.

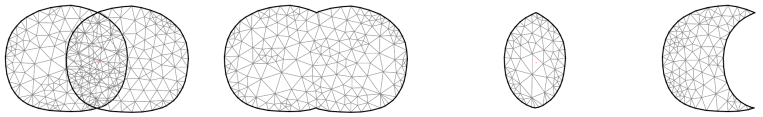
If the original sketch has strong, dark edges, the *Magic Wand* tool can be used to quickly create a region by clicking on a boundary and then moving the cursor around the path. The Magic Wand uses an algorithm based on the 'Intelligent Scissors' method by Mortensen and Barrett [1995]. First the reference image is converted to a grayscale intensity image, followed by convolution with a Sobel filter to enhance intensity edges. Each pixel in the image is then treated as a node in a graph, and the edges in the graph that link adjacent nodes are assigned weights according to a cost function that is proportional to the intensity edges in the image. When the user clicks on a dark boundary pixel, Djikstra's algorithm is used to calculate the cumulative cost from this initial pixel to every other pixel in the image. As the user moves the mouse, the shortest path from the pixel at the mouse position back to the initial pixel is calculated and displayed in real time, which enables the user to interactively see the path being traced as the mouse is moved around the image. The user can release the mouse button to select a satisfactory path.

As soon as a region is created (when the user releases the mouse), it is triangulated using the technique outlined in Section 3.2. Once the triangle mesh

is generated for a given region, the user specifies the *control points* by clicking within the mesh. The nearest mesh vertex is selected as the control point. To create the animation sequence, control points are used to deform the mesh according to key-frames. The control points can be thought of as being analogous to the joints in skeleton-based animation, but in contrast to skeleton-based animation, control points can be set anywhere (technically, on a vertex of the mesh) within a region, and thus they provide much more control over how the shape changes from one key-frame to the next.

Each object in a scene is made of multiple regions each of which resides in a *layer*. The layers are organized as a tree with transforms (translation, scaling or rotation) applied at a "higher" level and propagated to the entire subtree below it. This is useful as it enables both fine-grained control of individual regions that make up an object as well as a more coarse-grained control when we want to modify larger portions or all of an object. A special *root* layer, which is not editable, represents the entire scene and is the ancestor of all layers. Standard layer operations are provided, such as adding a new layer, deleting a layer, moving a layer up or down (re-ordering is allowed only among sibling layers), re-parenting a layer, which transplants the entire sub-tree (if one exists) below it, duplicating an existing layer, and renaming a layer.

The Mesh Editor also provides operations that work on two or more sibling layers. Multiple layers can be subjected to a *Group* operator that creates a *grouped layer*. A grouped layer can then be manipulated as a single unit. This is useful, for example, if we want to move the eyes and eyebrows in a face while not modifying nose and mouth. Constructive area geometry (CAG) binary operators such as Join, Intersect, and Subtract allow complex regions to be created out of basic ones. Figure 4.3 shows these operators in action.



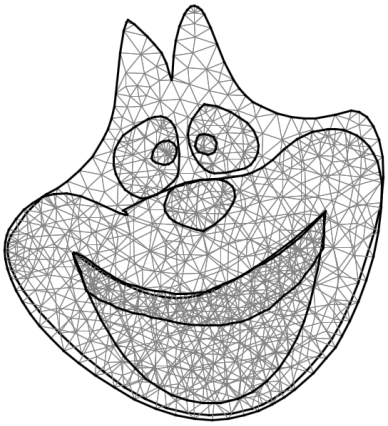
(a) Original shapes (b) Join (c) Intersect (d) Subtract

Figure 4.3: Constructing complex shapes using binary operators

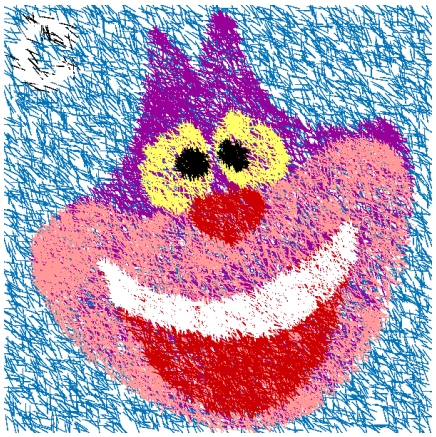
After a region has been defined, the next step is to set its *background references*, which provides color and tone information. A background reference can be an arbitrary image, a solid color, or a gradient. A region can have multiple associated background references, though only the currently selected one is used for sampling. Having multiple background references is useful if we want to blend colors between adjacent key-frames, for example, when creating a morph between two images. We can also specify the alignment of the background image (centered, top-left corner, etc.) with respect to the canvas. The background reference of a layer (or all its sub-layers) can be made invisible, in which case the layer will not be rendered.

Sometimes, we may want to create a mesh from an image that is imported as a background reference. The Mesh Editor provides an option to create a *mesh grid* from the background reference associated with a layer. This feature is especially handy when creating a mesh for an image used as background in a scene, or when we want to create a morph between two images. A regularly spaced grid of vertices and edges is added within a rectangular region around the image.

Once all the objects in the scene have been divided into regions (meshes), the user can click a button to see a preview of the non-photorealistic rendering. A drop-down widget enables the user to select the style in which to render the scene. Figure 4.4 shows an example of a (Cheshire Cat) character created and rendered

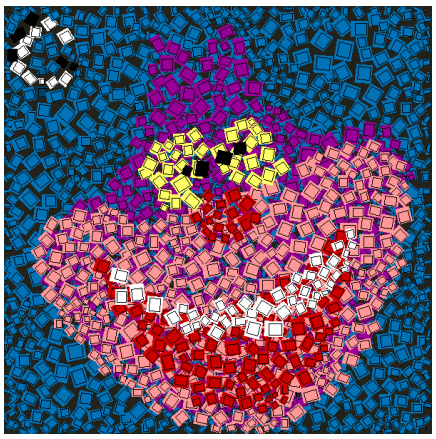


(a) Mesh



(b) Sketchy rendering

(c) Texture rendering



(d) Mosaic rendering

(e) Sand rendering

(f) Painterly rendering

Figure 4.4: Example of a character created and rendered in the Mesh Editor

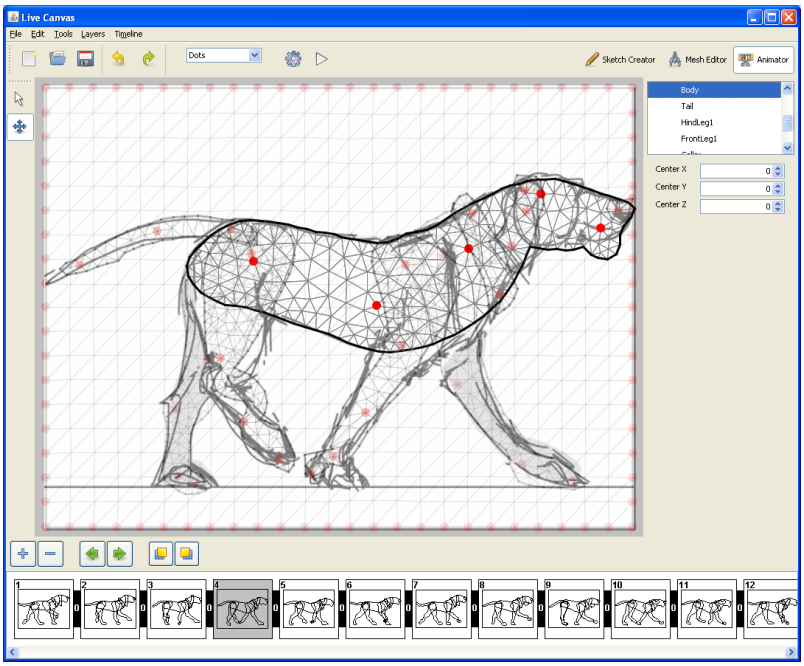


Figure 4.5: The Animator

in the Mesh Editor. The mesh around the teeth was created by intersecting two meshes.

4.3 Animator

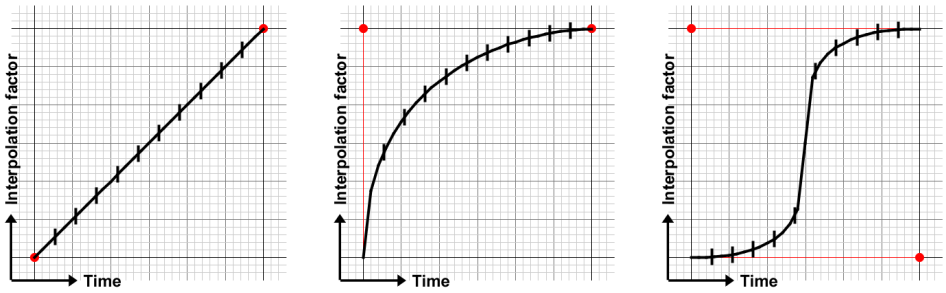
Figure 4.5 shows a screenshot of the Animator perspective within LiveCanvas. The Animator is used to create the sequence of key-frames from the scene created using the Mesh Editor and then render it as a movie. It provides a *timeline* of key-frames in the animation. The user first imports the meshes in a scene saved from the previous step, which creates the first key-frame of the sequence. A background reference image must be set for the first frame and optionally for each key-frame in the timeline. The reference image of the first frame is used to capture color and tone when the scene is rendered.

The timeline shows a thumbnail view of each key-frame in the animation. The user interface provides buttons to add a new key-frame, delete the selected key-frame, set and unset the reference image for the selected key-frame or enable and disable a semi-transparent drawing of the previous and/or next key-frame. This last feature is sometimes called *Onion Skinning* and it enables the user to see surrounding frames while changing the current frame. This is particularly useful for inexperienced animators as they can instantly see the current drawing in relation to preceding and subsequent drawings.

The Animator uses frames as opposed to time as the basic *pace* unit of the animation. The user can set the frame-rate, specified as number of frames per second. A higher frame-rate will make the animation go faster.

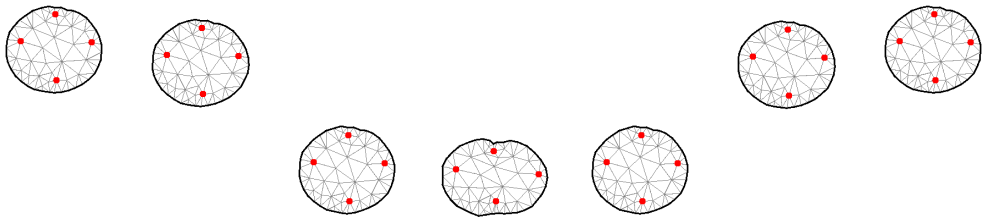
By default, each key-frame corresponds to one frame in the resulting animation. However, the Animator also allows *tweening* between two adjacent key-frames, which smoothly interpolates shapes over a user-specified number of intermediate frames, along a cubic Bézier curve. The curve can be edited using a tool called the *Interpolation Editor*, which enables the user to control the total count and timing of the intermediate, inbetween frames. The timing is specified by dragging control points on a Bézier curve plot with $(0, 0)$ and $(1, 1)$ as end points. The x-axis on the plot represents time and the y-axis represents the interpolation factor. The user can also specify the number of intermediate frames to be generated. Figure 4.6 shows examples of interpolation that can be obtained by adjusting the control points. A number of predefined templates are provided by the Interpolation Editor, such as linear, ease-in, ease-out, etc.

Figure 4.7 shows several frames from a bouncing ball animation. Of those shown, frame 6 and 30 are automatically generated through tweening, while the rest are key-frames. Frame 18 shows how we can *squash* an object through by



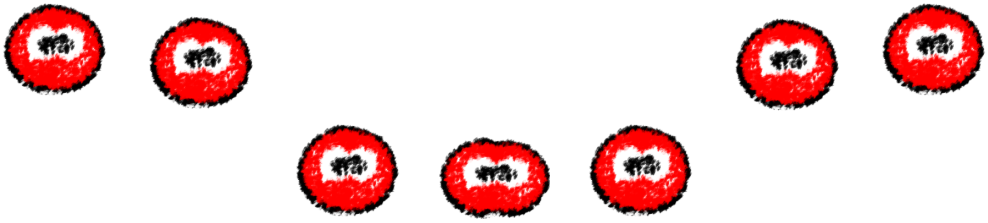
(a) $\{(0,0),(1,1)\}$ (b) $\{(0,1),(1,1)\}$ (c) $\{(1,0),(0,1)\}$

Figure 4.6: Timing control for 10 inbetween frames of a pair of key-frames. The coordinates of control points are shown in brace brackets.



(a) 0 (b) 6 (c) 12 (d) 18 (e) 24 (f) 30 (g) 36

Figure 4.7: Frames from a simple bouncing ball animation, showing the mesh and control points for the ball object.



(a) 0 (b) 6 (c) 12 (d) 18 (e) 24 (f) 30 (g) 36

Figure 4.8: Frames from a simple bouncing ball animation, rendered using textured brush strokes.

manipulating the control points on its mesh.

Figure 4.8 shows a bouncing ball animation rendered using textured brush strokes. The user provided a background reference image for only the first frame, which was then used to derive color information for the particles distributed on the mesh.

4.4 Implementation

LiveCanvas is written in Java and source code is available online at `http://code.google.com/p/livecanvas`.

4.5 Summary

We demonstrated our framework in an application prototype called LiveCanvas, which provides an integrated user-interface to create animation, from the initial concept sketch to the final rendered animation sequence. We described the components that make up LiveCanvas, highlighted their important features, and gave some examples of using LiveCanvas to create simple animations. The next chapter shows several more complex results and demonstrates the wide variety of animation, including key-frame based or tweened cartoon animation and image morphs, that may be created using LiveCanvas.

CHAPTER 5

Animation Results

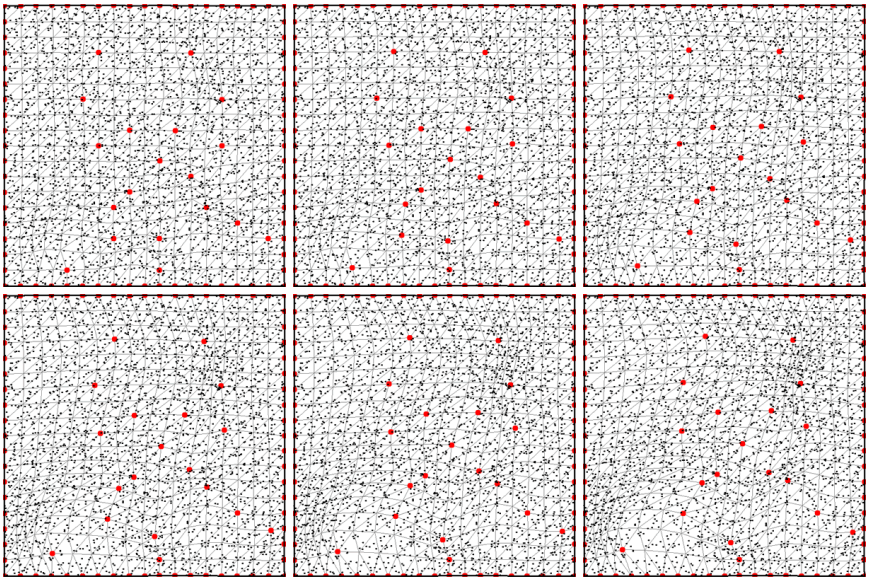
We now present frames from some of the animations and morphs generated using LiveCanvas. All results shown were obtained on a Windows Vista machine running a 32-bit JRE with 1GB maximum heap size. Please refer to the online project page for full videos and additional examples.[1]

Figure 5.1 shows a morphing animation generated by mesh deformation using Thin-plate splines (see Section 3.3.1 for details), with particle distribution on the mesh and rendering in the *sketchy* style. This style places a line stroke centered at each particle position and aligned with but jittered slightly from a particular angle.

Figure 5.2 shows the morphing animation rendered in the *mosaic* and *halftone* styles. The mosaic style, inspired by Hausner [2001], places square tiles randomly on a rectangular region. The halftone stipple pattern is generated by uniformly distributing particles on the mesh and then using a threshold to calculate the size of each particle.

Figure 5.3 shows a morphing animation generated by mesh deformation using Feature-Based Mesh Deformation (see Section 3.3.2 for details), with particle distribution on the mesh, rendered using textured brush strokes. This style *stamps* a colored brush image at each particle position aligned with but jittered slightly

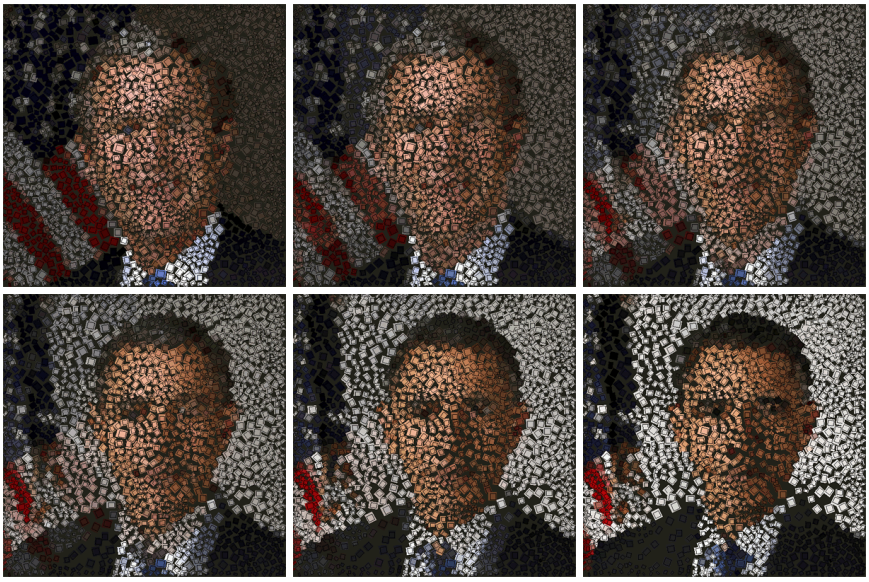
[1] http://www.cs.ucla.edu/~jasleen/livecanvas



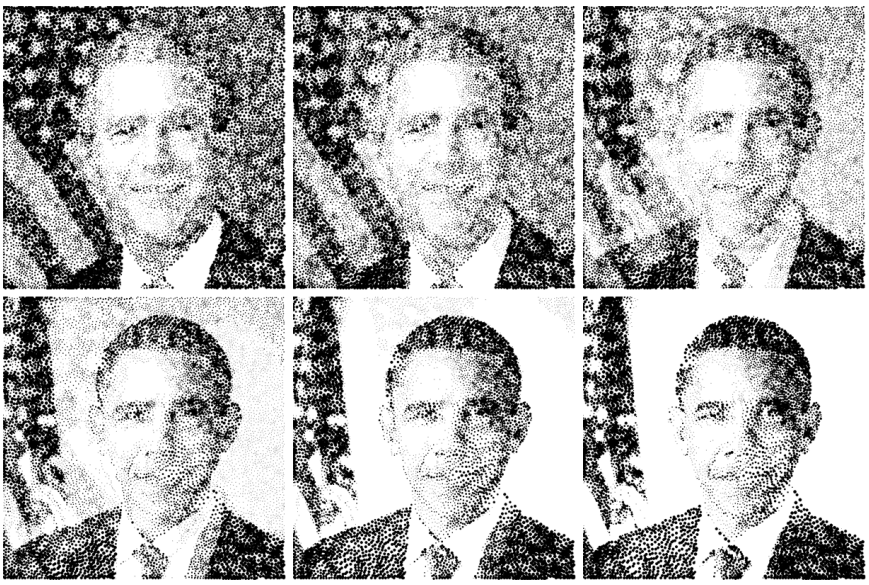
(a) Mesh deformation with particles

(b) Sketchy rendering

Figure 5.1: Thin-plate morphing showing particle distribution and sketchy style rendering

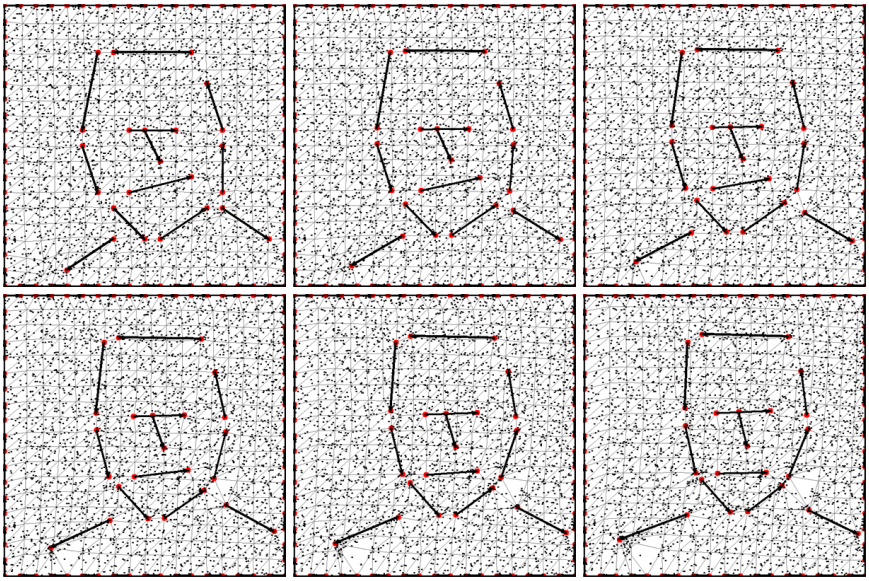


(a) Mosaic rendering



(b) Halftone rendering

Figure 5.2: Thin-plate morphing showing mosaic and halftone style rendering



(a) Mesh deformation with particles

(b) Texture strokes

Figure 5.3: Feature-based morphing showing particle distribution and rendering with texture strokes

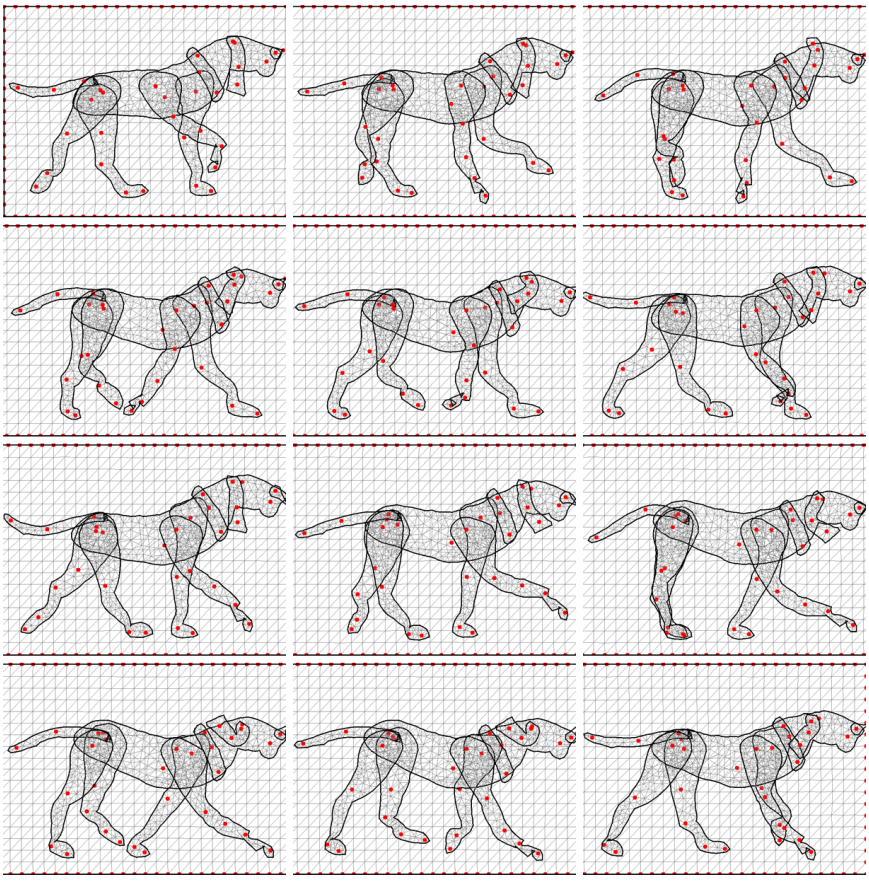


Figure 5.4: Key-frame based animation of a walking dog

from a particular angle.

Figure 5.4 shows a key-frame based animation of a walking dog. Each key-frame is generated by deforming the mesh representing each body part of the dog, using the control points (shown in red). The regular mesh-grid is associated with the background image.

Figure 5.5 shows the walking dog animation rendered with brush texture strokes. Note that the background translation is created by setting the position of the mesh-grid in the first and last frames, and then generating a motion tween in the intermediate key-frames. Thus, we can apply tweening to both frames

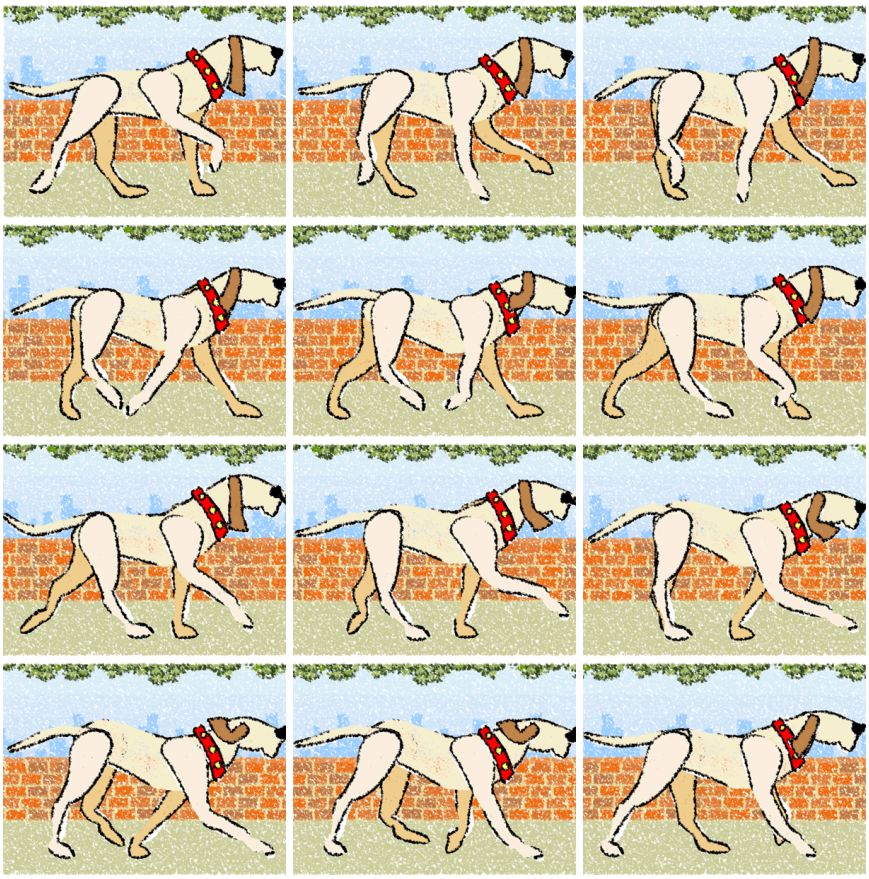


Figure 5.5: Rendering with brush texture strokes of the walking dog animation. The background has been animated using a translation tweening.

generated automatically (between two adjacent key-frames), as well as to the key-frames themselves. We can also freely mix tweening and manually-adjusted mesh deformations in any animation.

Figure 5.5 shows the walking dog animation rendered in the *sand* style. The sand style uses a threshold first to classify each particle as dark or light and then render each type using points sampled on corresponding dark-sand and light-sand textures.

Figure 5.6: Sand style rendering of the walking dog animation

CHAPTER 6

Conclusion

We will now conclude the thesis by reviewing our contributions and discussing interesting directions for future work.

6.1 Thesis Contributions

We have proposed a novel framework for 2D animation and rendering in non-photorealistic styles using stroke-based techniques. Creating an animation using tools such as Adobe's Flash and then simply applying a non-photorealistic *filter* results in a sequence that looks noisy and flickery, also called the *shower-door* effect. To avoid this problem, our framework represents objects in a scene as triangle meshes. Each mesh has an additional *background reference* that provides color and tone information. Particles are distributed over each mesh. These particles, each of which is essentially a 2D coordinate pair with an additional information *packet*, are then used as sites for placing strokes. An object is animated by deforming its component meshes, which causes the particles distributed on each mesh to move appropriately and coherently in relation to the motion of the three vertices of the triangle face containing each particle.

Each part of our framework is inter-changeable, thus affording flexibility and control to the animator. We have implemented and applied our framework in an application called *LiveCanvas*, which can be used to create stylized 2D anima-

tions. The application enables an animator (or even a casual user) to go from concept sketch to non-photorealistically rendered animations by employing an intuitive and user-friendly interface. LiveCanvas provides access to a number of features commonly found in 2D animation tools and utilizes the 2D animation framework to structure and ultimately render the frames in an animation. It allows both interpolation (tweened) and key-frame based animation, or a mixture of the two. Since objects are represented as meshes, and the application also supports color blending between two key-frames, one can also use it to create animated *morphs* and then render the frames in different styles. The synthesized frames maintain temporal coherency of the brush strokes thereby yielding smooth-looking animations that avoid undesirable shower-door artifacts. We demonstrated the flexible nature of our framework by implementing mesh deformation using three different techniques and providing several non-photorealistic styles.

6.2 Future Work

Our work was focused on designing a framework for 2D animation unifying different stoke-based non-photorealistic techniques, demonstrating its feasibility by implementing it in an application, and creating animations of reasonable complexity. In LiveCanvas, we have developed a usable prototype, which can nevertheless be extended in many ways.

Currently, the filters used to render scenes are fully specified by the initial parameters and once the rendering process starts, there is little the animator can do to control the final outcome. Extending the framework to include the *artist-in-the-loop* has interesting HCI and design challenges.

Moreover, the objects in a scene are represented as layered 2D regions, which often results in flat-looking animation. To give more character to the animation, we would like to look into ways to include more 3D information about an object, without specifying the entire underlying geometry in 3D. Rivers et al. [2010] presents an interesting technique to interpolate between different poses specified in 2D to generate plausible renderings in any view.

Finally, it would be beneficial to conduct a user study in order to obtain feedback from artists and animators that may guide the process of making LiveCanvas more useful and powerful.

Bibliography

Beier, T. and Neely, S. (1992). Feature-based image metamorphosis. In *Proc. ACM SIGGRAPH Conf.*, pages 35–42. 26

Chui, H. (2001). *Non-Rigid Point Matching: Algorithms, Extensions and Applications*. PhD thesis, Department of Electrical Engineering, Yale University. 25

Gooch, B. and Gooch, A. (2001). *Non-Photorealistic Rendering*. A K Peters, Natick, MA. 5

Haeberli, P. (1990). Paint by numbers: Abstract image representations. In *Proc. ACM SIGGRAPH Conf.*, pages 207–214. 6, 7

Hausner, A. (2001). Simulating decorative mosaics. In *Proc. ACM SIGGRAPH Conf.*, pages 573–580. 45

Hertzmann, A. (1998). Painterly rendering with curved brush strokes of multiple sizes. In *Proc. ACM SIGGRAPH Conf.*, pages 453–460. 8, 10, 21

Hertzmann, A. (2003). A survey of stroke-based rendering. *IEEE Computer Graphics and Applications*, 23(4):70–81. 10

Igarashi, T., Moscovich, T., and Hughes, J. F. (2005). As-rigid-as-possible shape manipulation. *ACM Transactions on Graphics*, 24(3):1134–1141. 28

Litwinowicz, P. (1991). Inkwell: A 2-D animation system. In *Proc. ACM SIGGRAPH Conf.*, pages 113–122. 13

Litwinowicz, P. (1997). Processing images and video for an impressionist effect. In *Proc. ACM SIGGRAPH Conf.*, pages 407–414. 7, 10

Markosian, L., Kowalski, M. A., Goldstein, D., Trychin, S. J., Hughes, J. F., and Bourdev, L. D. (1997). Real-time nonphotorealistic rendering. In *Proc. ACM SIGGRAPH Conf.*, pages 415–420. 5

Meier, B. J. (1996). Painterly rendering for animation. In *Proc. ACM SIGGRAPH Conf.*, pages 477–484. 10, 11, 21

Mortensen, E. N. and Barrett, W. A. (1995). Intelligent scissors for image composition. In *Proc. ACM SIGGRAPH Conf.*, pages 191–198. 37

Northrup, J. D. and Markosian, L. (2000). Artistic silhouettes: A hybrid approach. In *Proc. Int. Symp. on Non-Photorealistic Animation and Rendering (NPAR)*, pages 31–37. 5

Rivers, A. R., Igarashi, T., and Durand, F. (2010). 2.5D cartoon models. *ACM Transactions on Graphics*, 29(4). 54

Ruttkay, Z. and Noot, H. (2000). Animated CharToon faces. In *Proc. Int. Symp. on Non-Photorealistic Animation and Rendering (NPAR)*, pages 91–100. 15

Salisbury, M., Anderson, C. R., Lischinski, D., and Salesin, D. (1996). Scale-dependent reproduction of pen-and-ink illustrations. In *Proc. ACM SIGGRAPH Conf.*, pages 461–468. 7

Salisbury, M., Anderson, S. E., Barzel, R., and Salesin, D. (1994). Interactive pen-and-ink illustration. In *Proc. ACM SIGGRAPH Conf.*, pages 101–108. 7

Secord, A. (2002). Weighted Voronoi stippling. In *Proc. Int. Symp. on Non-Photorealistic Animation and Rendering (NPAR)*, pages 37–43. 11, 13

Secord, A., Heidrich, W., and Streit, L. (2002). Fast primitive distribution for illustration. In *Rendering Techniques*, pages 215–226. 11

Small, D. (1991). Modeling watercolor by simulating diffusion, pigment, and paper fibers. In *Proc. SPIE 1460: Image Handling and Reproduction Systems Integration*, pages 140–146. 6

Smith, M. D. and Sederberg, T. W. (2004). User-guided shape blending for cartoon animation. http://www.elecorn.com/tweenmaker. 14

Sousa, M. C. and Buchanan, J. W. (1999). Computer-generated graphite pencil rendering of 3D polygonal models. *Computer Graphics Forum*, 18(3):195–208. 6

Strassmann, S. (1986). Hairy brushes. In *Proc. ACM SIGGRAPH Conf.*, pages 225–232. 6

Terzopoulos, D. and Vasilescu, M. (1991). Sampling and reconstruction with adaptive meshes. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 70–75. IEEE. 25

Williams, R. (2009). *The Animator's Survival Kit, Expanded Edition*. Faber and Faber, Second Edition. 37