Novel animation and rendering approaches to cartoon

Lu, Yixiang

2008

Lu, Y. (2008). Novel animation and rendering approaches to cartoon. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2599>

<https://doi.org/10.32657/10356/2599>

Nanyang Technological University

Downloaded on 26 Apr 2023 20:24:01 SGT

Nanyang Technological University

**NOVEL ANIMATION AND RENDERING
APPROACHES TO CARTOON**

A Thesis

Submitted to School of Computer Engineering

Nanyang Technological University

By

LU YIXIANG

Master of Engineering (M. Eng)

March 6, 2008

Abstract

2D animation production has always been time-consuming and labor-intensive. This thesis investigates several processes in the animation production pipeline that are especially laborious. A 2D animation system has been created to provide more efficient method of performing those tasks. A tremendous amount of research work has been done in the creation of the system, including user-controllable automatic inbetween generation, automatic coloring, smart stroke input, high quality stroke rendering to provide accurate visual feedback and distributed framework to reduce computational time.

Inbetweening is one of the most time-consuming processes of 2D animation production. Automatic inbetween generation addresses this issue, but it remains a tough issue in research. Most existing methods apply interpolation based on either pixel intensity of raster images or mathematical functions of vector representation, but neglect information in the drawing itself, e.g. shapes or motion of characters. That often results in distorted lines and shapes as well as unsmooth motion. This thesis contributes significant improvements to automatic inbetween generation algorithms. They include parameterized non-linear interpolation, feature point preservation and junction preservation.

Coloring is another time-consuming process in 2D animation production. Great amount of research has been done intending to achieve automatic frame generation. This thesis proposes vector-based region recognition and region matching algorithms.

Regions on a frame can be found, and then matched with regions in another frame; if one frame is colored, the other is automatically colored.

Many animators prefer the traditional way of creating animation – drawing on paper. This has become a significant barrier in switching animation production to digital animation systems. This thesis proposes algorithms and flows to convert user input to digital representations more efficiently and accurately. Rendering algorithms with enhanced rendering quality and performance are proposed as well to provide visual feedback more accurately and yet faster.

To make the system even more efficient, this thesis introduces a distributed framework, which all the computation intensive processes can run on. As a result, time wasted on waiting for previews is significantly reduced.

Acknowledgement

I would like to dedicate my most sincere thanks to Dr. Tian Feng, my thesis adviser, for his guidance, encouragement and support throughout the research. Special thanks go to Professor Seah Hock Soon for his generous advice and continuous support. I appreciate the valuable suggestions from Dr. Wu Zhongke at the initial phase of my research.

Deep thanks go to Mr. Qiu Jie, Mr. Chen Quan and Miss Xiao Xian, who offered valuable suggestions and great moral support when I needed most.

I would like to express my gratitude to the artists in IERC lab, Lu Peng, Eddie Ching, Anil Damodaran and Cao Youfang, for providing their great demonstration examples and valuable feedbacks about the project.

I must also thank my beloved family and friends for their indispensable source of motivation and encouragement. I deeply appreciate my lab mates for making the working life an enjoyable one. Lastly, I am grateful for all the help I have received in one way or another.

Table of Contents

Abstract	i
Acknowledgement	iii
Table of Contents	iv
List of Figures	vi
List of Tables	vii
Chapter 1 Introduction.....	1
1.1 Motivation	1
1.2 Approaches	2
Chapter 2 Survey	6
2.1 Animation Production Process	6
2.1.1 Traditional Animation.....	6
2.1.2 Computer-Assisted Animation.....	11
2.2 Existing 2D Animation Systems.....	12
2.2.1 Review of 2D Animation Systems.....	13
2.2.2 Key Frame Matching and Curve Interpolation	19
Chapter 3 Interpolation.....	22
3.1 Linear Interpolation	22
3.2 Nonlinear Interpolation.....	24
3.3 Feature-Based Stroke Interpolation	26
3.3.1 Feature Point Extraction	27
3.3.2 Feature Point Correspondence	30
3.3.3 Stroke Interpolation Based on Feature Points.....	33
3.4 Preserving Junction in Interpolation.....	35
3.4.1 Intersection Detection	36
3.4.2 Junction Preservation	40
3.5 Summary.....	42
Chapter 4 Curve Fitting and Rendering.....	43
4.1 Curve Fitting.....	43
4.2 Curve Rendering.....	46
4.2.1 Border Estimation Rendering	47
4.2.2 Approximating Border	48

4.2.3	Curvature Weighted Sampling.....	50
4.3	Examples and Analysis.....	51
4.4	Summary.....	53
Chapter 5	Regions and Colors	54
5.1	Region Definition	54
5.2	Region Detection	55
5.2.1	Converting to graph	55
5.2.2	Finding Regions	58
5.2.3	Finding Primary Regions	60
5.3	Region Matching	62
5.4	Region Matching Examples.....	64
5.5	Summary.....	65
Chapter 6	Distributed Interpolation and Rendering	67
6.1	Frame Generation Framework.....	68
6.2	Rendering Algorithm and Complexity	70
6.3	Distributed Rendering Algorithm.....	73
6.4	Experimental Results.....	75
6.4.1	Sample Output	75
6.5	Summary.....	81
Chapter 7	Conclusion and Future Work.....	82
7.1	Conclusion	82
7.2	Future Work.....	83
Appendix		84
Reference		90

List of Figures

Figure 1.1: Examples of Disk B-spline Curve.....	4
Figure 1.2: Example of Inbetweening with DBSC	5
Figure 2.1: Traditional 2D Animation Production Process	7
Figure 2.2: Master Frames.....	8
Figure 2.3: Model Sheet	8
Figure 3.1: Disk B-spline curve.....	23
Figure 3.2: Inbetweens Generation Using DBSC.....	24
Figure 3.3: Slow In Fast Out Example	26
Figure 3.4: Different Shape and Position Change Rate	26
Figure 3.5: Linear Interpolation – Facial Silhouette.....	27
Figure 3.6: Calculating Curvature	29
Figure 3.7: Feature Point Extraction	30
Figure 3.8: Feature Points Mismatching Example	32
Figure 3.9: Feature-Based Stroke Interpolation – Facial Silhouette.....	34
Figure 3.10: Inbetween Generation Results – Chicken	35
Figure 3.11: Example of Regions	36
Figure 3.12: Stroke Intersection	39
Figure 3.13: Bezier Curve and Convex Hull	40
Figure 3.14: Types of Artifacts in Interpolation Result.....	41
Figure 3.15: Junction Preservation Example	42
Figure 4.1: Bezier Curve and Its Control Points.....	44
Figure 4.2: B-Spline and Its Control Points	45
Figure 4.3: Control Point Relationship.....	46
Figure 4.4: Stroke Illustration	47
Figure 4.5: Stroke & Control Points	48
Figure 4.6: Comparing Results of Three Rendering Algorithms.....	52
Figure 4.7: Results of Three Rendering Algorithms with Knot Points.....	53
Figure 5.1: Self-Intersecting Stroke Converted To Graph.....	56
Figure 5.2: Strokes Converted To Graph	57
Figure 5.3: Pruning.....	58
Figure 5.4: Searching Tree	59
Figure 5.5: Tangent Line	62
Figure 5.6: Region Matching Examples	63
Figure 5.7: Region Not Matched	64
Figure 5.8: Region Matching Example Labeled.....	64
Figure 5.9: Region Matching Example Colored.....	65
Figure 6.1: Distributed Rendering Framework.....	69
Figure 6.2: Tukasa Rendering Output	76
Figure 6.3: Sayuri Rendering Output	78
Figure 6.4: Tukasa 2 Rendering Output	79
Figure 6.5: Kusaga Rendering Output.....	81

List of Tables

Table 4.1: Border Approximation Pseudo Code	49
Table 5.1: Find Cycle Pseudo Code	59
Table 5.2: Searching Primary Regions Pseudo Code	61
Table 6.1: Rendering Process Pseudo Code	70
Table 6.2: Rendering Process Decomposed	71
Table 6.3: Distributed Rendering Algorithm.....	73
Table 6.4: Tukasa Rendering Details and Performance	77
Table 6.5: Sayuri Rendering Details and Performance	78
Table 6.6: Tukasa 2 Rendering Details and Performance	79
Table 6.7: Kusaga Rendering Details and Performance	81

Chapter 1

Introduction

This chapter introduces animation production and discusses briefly why digital 2D animation systems are desirable. Existing approaches as well as our approach to digital 2D animations are then introduced.

1.1 Motivation

Traditionally, 2D animation production has been a labor-intensive process of drawing and painting animation sequences entirely by hand. Under the guide of script, story-board and character design and exposure sheets, key animators draw key frames showing the major features of each character's action and assistant animators draw inbetween frames to make animations smoother. The outlines will then be filled with different colors in various areas and become colored frames. Frames of different layers are combined into composite frames which appear in the final animation according to the exposure sheets; special effects may be applied during the composition process. Among all the procedures mentioned above, inbetween frame drawing and coloring are the most tedious and most time-consuming tasks. To save time and labor, an automatic inbetween generation system is desirable. Researchers have devoted tremendous time and effort on the techniques of interpolating animation key frames. For example, [1] and [23] work on image morphing using triangulation. However that is not suitable for line drawings. A pre-defined correspondence between components, e.g. points, curves or shapes, [1], [3], [27], in two key frames is usually needed to deliver smoother and better looking intermediate images. However, this demands much user-

interaction beforehand. Some automatic correspondence methods [14] work well for simple cases where the number of components to be matched is small but usually user correction is inevitable. J. D. Fekete et al. [8] presented an approach to a complete vectorized drawing process by providing an interface for a "paperless" system. He noticed that the automation changes the nature of inbetweening and limits its complexity.

1.2 Approaches

Animation production techniques develop rapidly. Vector-based animation is becoming more popular in industry. It has several advantages over traditional paper-based hand-drawn animation. Vector-based drawing facilitates easy modification and correction. It supports free scaling without quality loss, which makes the production independent of output resolution, thus suitable for multi-format output, including normal TV, high definition TV or even movies. Vector-based drawing requires much less storage space, compared with huge piles of drawings on paper and large amount of space for raster images. Transportation is therefore easier and more efficient, which is very important to the industry in current world of collaboration.

Two major approaches exist for vector-based animation systems, curves as strokes and regions as stroke. The former uses curves to represent strokes of constant width. This limits the drawing styles of animators and the drawings may look non-artistic, mechanical or even dull. The latter approach represents strokes using regions. Region representation is general and can represent a wider range of strokes, including those with variable width (within the same stroke). However it is difficult to handle inbetween generation using the representation, because

interpolating two regions requires significant amount of knowledge on how various parts of the two regions correspond.

We proposed a novel approach to computer assisted 2D animation, which is based on a novel representation of strokes, DBSC proposed in [32], [33], [34]. It defines a region as well as its center curve. It is a generalization of Disk Bezier curve and can be used as an effective representation of freeform 2D shapes. With rigid mathematical fundamentals, DBSCs are flexible for manipulation and deformation. When user draws with a digital tablet, a stroke action is converted to a DBSC in real-time. To create animations, two key frames, each containing a set of DBSCs, are given and inbetweens are automatically generated by interpolating corresponding DBSCs.

In previous method [33], [34], linear interpolation is applied of two DBSCs in parametric domain. Properties of the drawing of the character are not taken into consideration, which may result in distortion and unrealistic motion in animation.

For a complete presentation, we briefly describe the basic definitions and more details can be found in [32], [33] and [34].

Defining $N_{i,p}(t)$ as the i th B-spline basis [23] of degree p with knot vector:

$$[u_0, \dots, u_m] = \{a, \dots, a, u_{p+1}, \dots, u_{m-p-1}, b, \dots, b\} \quad 1.1$$

where a and b both repeat $p + 1$ times;

A disk [24] in the plane can be represented as

$$\langle P; c \rangle = \{x \in R^2 \mid |x - c| \leq r, c \in R^2, r \in R^+\} \quad 1.2$$

A disk B-spline curve is defined as follows:

$$\langle D \rangle(t) = \sum_{i=0}^n N_{i,p}(t) \langle P_i; r_i \rangle \quad 1.3$$

where P_i is the control points, r_i is the control radius. Some examples of DBSC are shown in Figure 1.1: .

Figure 1.1: Examples of Disk B-spline Curve

A disk B-spline curve has two components: a center curve (a B-spline curve) and a radius function (a B-spline scalar function). Therefore most algorithms designed for B-spline curves can be extended and applied on DBSCs.

When a set of points $\{Q_i\}, i = 0, \dots, m$ on the central curve and their corresponding maximum distances $\{c_i\}$ are given, a DBSC whose center curve passes through all data points of $\{Q_i\}$ and maximum radii are $\{c_i\}$ can be obtained by interpolation. The curve can be either closed or open. B-spline curve interpolation and scalar function method can be used to obtain DBSC interpolation.

An example of inbetweening is shown below in Figure 1.2.

Figure 1.2: Example of Inbetweening with DBSC

With the power of DBSC, it is possible to implement an animation creation system that allows users to draw directly from their tablets and create animation based those drawings.

In Chapter 2, existing animation systems are reviewed. Chapter 3 discusses automatic inbetween generation based on DBSC in detail. How user inputs are converted into strokes as well as how strokes are displayed is explained in Chapter 4. In Chapter 5, regions and automatic coloring of regions are covered. Chapter 6 then proposes a distributed framework for high performance interpolation and rendering. This thesis is concluded in Chapter 7.

Chapter 2

Survey

In this chapter, animation production processes will be reviewed in detail. Some existing digital animation systems are then introduced one by one. Their features, such as inbetween frame generation, will be compared and their limitations will be discussed.

2.1 Animation Production Process

To develop a 2D animation system which can save time and labor of animation production, it is important to understand the traditional and current animation production processes and which parts of the work can be assisted by computers. These are presented in the following sections of this chapter.

The flow of animation production process is illustrated in Figure 2.1 and as delineated in [22], [30]. For higher production efficiency, different steps are often overlapped or even parallel in practice [22].

2.1.1 Traditional Animation

The diagram shown in Figure 2.1 illustrates a typical production workflow. It includes *Story Development*, *Sound Track*, *Model Sheet*, *Storyboard*, *Leica Test*, *Scene Staging*, *Layout*, *Sound Detection*, *Exposure Sheet*, *Key Frame Drawing*, *Inbetweening*, *Line Test*, *Cleanup*, *Inking/Painting*, *Background Painting*, *Check*, *Rostrum Camera*, and *Editing*. Each of these procedures will be explained in the rest of this section.

Figure 2.1: Traditional 2D Animation Production Process

2.1.1.1 Story development

In this first stage, the story of the animation is developed and the following stages serve for visualizing this story to the viewer in the form of 2D animation.

2.1.1.2 Sound track

Traditionally, sound track is produced after the story is developed by recording dialogues and music in the prototype form, which is used to drive the animation timing later. Alternatively, sound track can also be added in the final editing stage.

Chapter 2. Survey

2.1.1.3 Model sheet

Based on the story and the chosen animation style, model sheets are prepared to guide animators to draw and color characters consistently.

Master frames as shown in Figure 2.2 are a part of model sheets. Additionally, characters' faces are also provided with various expressions and/or view angles, and colors used and shadows applied may also be shown in model sheets.

Figure 2.2: Master Frames

Figure 2.3 shows a typical model sheet from Walt Disney Animation Canada Inc.

Figure 2.3: Model Sheet

(Courtesy of Walt Disney Animation Canada Inc.)

2.1.1.4 Storyboard

After all characters are designed and illustrated on model sheets, the story is transcribed into a storyboard, which is basically made up of a stack of papers with sketchy drawings of the envisaged movie shown. A storyboard may also include shooting scripts, timing charts and dialogues.

2.1.1.5 Leica test

Leica test is the first attempt to put together a film of the required length. The test is composed from stills corresponding to the storyboard with a soundtrack added to test the timing of the sound, especially for dialogues. The aim of this test is to give animators an impression of the movie so that they can spot the weaknesses and rectify them.

2.1.1.6 Scene staging

Scene staging is the mapping out of each scene, in particular the relationship between the characters and the background elements within the same scene. These relations are clear presentation of ideas, including avoiding anything that might distract the viewer's attention, placing characters, viewpoints, expressions etc., and even the timing of a scene, so that the viewer's attention is engaged most directly.

2.1.1.7 Exposure sheet

The information from the storyboard, scene staging and soundtrack is transcribed onto an exposure sheet. An exposure sheet is basically a table whose rows represent frames, giving animators detailed instructions for drawing, compositing and shooting of each frame.

2.1.1.8 Drawing

Key frames and inbetween frames are drawn in succession. The process of drawing inbetween frames takes a lot of time and labor in the whole animation production, and it is what computer-assisted animation systems [4] need to address.

2.1.1.9 Line test

Line test is produced to reveal and correct problems in line drawings. The line drawings, without background, are shot and played back to verify that the movement is correct and characters interact accurately.

2.1.1.10 Cleanup

The sketch of key drawings obtained after the prior stages are cleaned up by tracing. Cleanup animators superimpose a piece of blank paper on top of each rough drawing on the light box and then draw clean strokes on the upper blank paper with reference to the underlying key drawing.

2.1.1.11 Coloring

The conventional way for coloring is transferring clean line drawings onto sheets of celluloid (cels) by xerography. Usually several layers of cels, containing animated elements and the background, are produced separately. And then coloring is performed on the cels, each with the right degree of opacity. With the assistance of a computer, coloring is usually completed with animation systems or image processing software like Adobe Photoshop or Retas! Pro after all layers are scanned into the computer. Due to high demand on manpower during coloring, automating coloring process is a significant goal for us to achieve.

2.1.1.12 Rostrum camera

To get the animation film, all layers of each frame are now composited together on the rostrum camera based on the exposure sheets, illuminated and shot at certain rate, e.g. 24 frames per second. Quite often, photographic special effects are incorporated in this stage as well.

2.1.1.13 Editing

To transform the film into a final product, post processing and editing is applied. In this phase, the film is assembled, sorted and joined when necessary.

2.1.2 Computer-Assisted Animation

As computer technologies advance, the process of traditional 2D animation production gradually starts to benefit from computer assistance. Many animation production procedures have been improved with advanced technology to achieve better efficiency. For example, inking and painting are often done digitally after the outline drawings are scanned into the computer instead of being transferred to cels and then colored by hand, which cost significantly more in human resource and materials. Compositing and Editing are mainly done on computer as well, considering the convenience of applying special effects. A good integrated computer animation system also facilitates exchanging of artwork and collaboration between departments, studios and even countries.

In frame drawing, it has become possible for animators to draw frame outlines directly into a computer using a tablet device. It is similar to drawing on paper but it is easier for modification and less demanding on skills. This paperless animation drawing technique is likely to supersede the traditional pencil and paper drawing in the future. Among different approaches for paperless drawing, some store

drawings as raster images while others represent them in vector form. Both will be discussed in the following paragraphs.

Raster images capture the exact styles and characteristics as desired and are more intuitive to control. Artists can easily produce higher-quality drawings using their paper drawing skills, comparing with vector drawing tools which might impose limitations on animators' drawing styles. However, the higher quality we require, which yields larger resolution, the more storage spaces are required.

On the other hand, vector drawings have their advantages, e.g. scalability and compactness. They can be scaled and rotated without losing fidelity. It makes reusing easier, e.g. by simply changing output resolution. Vector form is concise in representation hence small in storage size. The two advantages above are sufficiently convincing to support the view that vector drawings are more suitable for animation production. It is especially true for those animations made for high definition display devices, which are getting increasingly popular and will soon become dominant. Besides, there are also techniques developed to convert raster drawings to vector form which will benefit from both raster and vector techniques.

With that said, it is difficult to generate inbetweens automatically, for both vector and raster drawings. As of today, most inbetweens are still drawn frame by frame. Therefore it is one of our major goals to make automatic inbetween generation easier and more accurate.

2.2 Existing 2D Animation Systems

There are many computer-assisted 2D animation systems available in the market, both professional and amateur. This section reviews some of those 2D systems. Since our research concentrates on the inbetweening and coloring of animation

production, the review will focus on the inbetweening (if any) and coloring modules/features of these animation systems. Irrelevant attributes, like user-friendliness, are neglected in this case.

2.2.1 Review of 2D Animation Systems

2.2.1.1 TicTacToon [8]

TicTacToon is a computer-assisted system for professional 2D animation production produced by Toon Boom Technologies Inc. It is the first animation system to use vector-based sketching and painting. The Ink and Paint system of TicTacToon uses an area flooding algorithm [17]. To color a region after all gaps are closed, the user must select a color and point to a pixel in the region. Consequently all pixels within the region can be painted with the same color by flooding. Although animators need not paint all frames using brushes and pigments by hand, painting in this manner is still tedious because the system keeps animators busy clicking the mouse to paint and using the keyboard to change the drawings. The future work as described in [8] is to speed up the painting process by automatically finding the region color either from a template or from a previously painted cel, similar to our objective.

2.2.1.2 US Animation [49]

US Animation is another product of Toon Boom Technologies Inc., and it is also a vector-based 2D animation system. With the aid of the Paint Module of this system, users can select the color to paint from a palette, a color model, or a drawing and paint one or several regions in the same color by drawing a selection mask through the regions to be painted. One great feature of the Paint Module is to auto-paint the "same" regions on all drawings, that is, after the user clicks and

drags his cursor through the regions he wants to paint, the Paint Module flips through all drawings and colors all regions the cursor touched in the same color. This auto-painting function works in some cases where the translation and variation of the corresponding regions in sequential frames are very small. Some errors may be unavoidable and user-intervention is needed to correct these errors. Besides flat colors, this system can also paint regions with textures and gradients.

2.2.1.3 Toon Boom Studio [49]

Toon Boom Studio is the latest product of Toon Boom Technologies Inc. It acts like an integration of TicTacToon and US Animation with updated UI design. For the technology used in Ink and Paint module, new features like automatic update of painted zones with palette changes are added to make coloring process faster and easier. However, auto coloring based on region correspondence, stated as future research work of TicTacToon in [8], is still unavailable.

2.2.1.4 Animo [39]

Animo is a software produced by Cambridge Animation Systems for professional 2D/3D animation production. The Color Styling/InkPaint Module of Animo can derive a region map of the areas to be painted, allowing for very fast painting. Auto-fill function, locking tools and specialized ink-and-paint tools make painting fast and simple. In details, Auto-fill fills similar regions over a range of frames after the first frame is colored, working best if the number of regions remains same and the distance between the regions from frame to frame is not "huge". The main problem of Auto-fill is that it can only handle small changes between regions and is greatly influenced by the numbers of regions in different frames. If the frame to be colored has more regions (because of occlusion or other reasons) than the first colored frame, some regions are definitely left uncolored.

2.2.1.5 Retas! Pro [47]

RETAS! Pro is a digital animation production package produced by CELSYS, Inc., which replaces all labor-intensive processes of conventional animation production as mentioned in its official website. RETAS! Pro follows the conventional process of animation production with a combination of five tools which replace work such as drawing, pencil testing, tracing, painting, effects and shooting (camera). The tool for painting is PaintMan. Besides the common filling method by selecting color and clicking on a pixel in the region to be colored, the function of AutoFill automatically paints an entire sequence instead of manually painting frame by frame. The user just needs to click on one point in the first frame to determine the coordinate and then regions containing pixels at the designated coordinate will be painted with the same color in all frames. This function of AutoFill is in fact a batch process and errors do frequently occur because the selected points in the first frame may not be included in the corresponding regions in subsequent frames, thus user-intervention is inevitable.

2.2.1.6 A color animation system based on the multi-plane technique [16]

Cornell University develops a color animation system based on the multi-plane technique simulating the function of multi-plane camera created by Walt Disney. Each plane represents a layer. Coloring is also done by simple area-filling algorithm in this system with color manually selected from a palette by the user. However, if a set of seeds to drive the area-filler is interpolated along with the profile lines, intermediate frames can be automatically generated with full coloring. The seed means a pixel within a region to be colored. The auto coloring function here is simply a batch process with the user clicking on "seeds" first. Color is not automatically selected by the computer.

2.2.1.7 Inkwell [19]

Inkwell is an experimental 2D key frame animation system. 2D means maintaining a drawing order for 2D objects causing the objects to appear to be layered on top of each other. One unique feature of Inkwell is that it can perform inverse texture mapping to warp the portion of the picture underneath the Coons patch as its texture map and deform the texture corresponding to the metamorphosis of Coons patch, this is very useful to make the texture of regions more realistic. More details about Coons patch and the texture mapping can be found in [5], [7].

2.2.1.8 Adobe Flash [43]

Adobe Flash is widely known software for web animation design. The latest version is Flash Professional 9. Similar to traditional 2D animation, a web animation created by Flash is composed of one or more scenes. Successive frames constitute a scene and each frame is a composite result of one or more layers. All colors used in Flash are manually selected from a color palette by the user. No auto coloring function is included in this software.

2.2.1.9 Mirage Studio [38]

Mirage Studio is a paperless digital animation system with real-time paint, animation, and special effects functionalities designed by Bauhaus Software, Inc. Mirage Studio supports real-time paint on 16bit color depth, up to 32bit per layer. It provides powerful filling tools, like freehand fill, polygon fill, spline fill, flood fill, interactive flood fill, gradient fill, custom brush fill etc.. Similar to other animation software, it can facilitate the painting process of animation production, but cannot automatically color all characters.

2.2.1.10 Plastic Animation Paper [44]

Plastic Animation Paper (PAP) is a paperless 2D animation software package produced by Krogh Mortensen Animation. PAP enables the animator to draw directly to the screen in video or film resolution using a drawing-tablet. It specializes in doing the actual animation but not coloring or compositing. As stated on their website, other kinds of programs perform coloring or compositing tasks better. The focus of PAP is on getting the most effective software and workflow for the animator.

2.2.1.11 Toonz [50]

Toonz is 2D animation software produced by Digital Video S.p.A. (DV), and the latest version is 5.1. It supports paperless 2D animation, and can also be used for making animation from line-drawings on papers. Automatic Painting Tools provided by Toonz allows users to quickly and effectively paint sequences of drawings and animated levels, which is similar with the Paint Module in US Animation. However, the software also does not provide tools to color characters in a sequence of frames automatically.

2.2.1.12 CelAction2D [40]

CelAction2D is a suite of integrated programs that allows animators to design 2D character models on computer as resolution-independent vector graphics or scanned bitmaps, and animate them according to exposure sheets that are fully user-adjustable. As indicated in its workflow [41], no drawing is done using CelAction2D itself - artwork is created by drawing or scanning images into applications such as Photoshop, Illustrator, Flash or Freehand, in order to produce vector or bitmap-based files in either .AI, .SVG or .PSD format.

2.2.1.13 FlipBook Painter [42]

FlipBook Painter is the painting module in 2D animation software FlipBook produced by DigiCel Inc. In FlipBook Painter, numerous painting tools are developed to help save animators' time, to improve quality and to prevent mistakes, like Custom Color Palettes, Drag and Fill, Paint-By-Numbers, Stop Leak, AutoFinish and Multi-Level Painting for Limited Animation etc. The Paint-By-Numbers function is similar to US Animation's auto-painting function, painting the regions by flood filling regions from the cursor position in current and the following N frames, where number N is from user input. The AutoFinish function paints all of the remaining regions with nearest color at high speed. However the software does not include any auto coloring function.

2.2.1.14 Animation Stand [36]

As stated in its website, Animation Stand is the leader in cross-platform tools for professional, cost effective, easy to use computer-assisted 2D character animation, compositing and film quality special effects, which is produced by Linker System. Animation Stand provides automatic cel painting function, which is a batch process similar to the AutoFill function in Retas! Pro PaintMan. The color painted and mode used at the position mouse clicked on the first cel is recorded and displayed as a paint chip in the subsequent cels, and the user can adjust the position of each paint chip or delete it. Finally each cel will be painted based on these paint chips.

All the above systems, except Animo, do not have real auto coloring function or module. In fact, even the auto coloring provided by Animo is rather limited. Hence, coloring remains tedious and should be investigated further to find a

suitable auto coloring algorithm, which makes coloring more time and labor effective in traditional 2D animation production.

2.2.2 Key Frame Matching and Curve Interpolation

In 2D animation, characters are not simply the projection of 3D character models. Animators sometimes deliberately break the rules of geometry and physics to exaggerate reality [22]. Consequently, matching of animation frames is more difficult than camera-acquired images. The existing techniques for matching animation frames can be classified into two groups: vector-based and raster-based. Vector-based techniques first vectorize the strokes in a cartoon drawing with various curve fitting approaches. A collection of vertices (sample points) and vectors (short line segments) or curves obtained by vectorizing forms a high-level description of the drawing and is used by various kinds of matching processes. These techniques usually need users to indicate the control points, thus divide contours of regions into strokes.

Most inbetweening techniques are based on a curve-to-curve transition or an object-to-object transition [8]. Interpolation is performed on the corresponding elements in key frames based on fitting a set of interpolating splines through the key positions.

Some early attempts are made to achieve smoother and more natural transition between the target and the source images, avoiding mechanical look, discontinuity and distortion of the intermediate curves. Reeves [25] presents an approach that gives animators more flexible control over interpolation utilizing movement trajectory and dynamics of certain points in the key frames. Kochanek and Bartel [13] propose a method of using cubic interpolating splines. In this method

animators have the option to change tension, continuity, and bias of the splines. Each of these three parameters can be used for either local or global control. Bartel and Hardock [3] studied the relationships between spline curves within a frame that hold throughout inbetweens, and proposed a method that uses the control points of the curves as key points of interpolation to generate inbetweens.

All the methods above assume one-to-one correspondence between components in the key frames and interpolation is performed to the corresponding elements. Besides, most of these techniques require the animator to manually specify explicit correspondences between the curves in the key frames. Therefore, it requires additional work aside from drawing. Moreover, the methods focus on interpolating one curve at a time without considering the correlations between elements within a key frame, which are sometimes critical to be preserved.

Only when correspondence between two key frames is set up, interpolation can be performed in regard to the transformation of the corresponding vertices, curves or shapes. In some of these methods, when key frame drawings are traced into the computer, strokes of the image must be traced in the order across all key frames. This ordering of strokes between related images controls the form of the intermediate image and the number of strokes in each key frame must be same to satisfy a one-to-one correspondence. These restrictions are hardly acceptable in real animation production.

Kort [14] presents an algorithm for automatically mapping the strokes in key frames. Each key frame is analyzed and classified into strokes, strokes chains and relations that hold among them. Correspondence is obtained based on the rules that restrict changes allowed in stroke relationships within a frame. The correct

Chapter 2. Survey

matching of strokes is determined by a cost-function-based approach. Like some earlier attempts [28] for automatic stroke mapping, this approach often produces incorrect matching and hence requires indispensable manual correction.

With handful of knowledge about existing animation software and their features, I will move on to the next chapter to explain our approach to inbetween generation (also known as interpolation) in detail.

Chapter 3

Interpolation

As mentioned at the end of Chapter 2, no existing digital 2D animation system is perfect in inbetween generation. It is our goal to invent a better solution for inbetween generation. Many research efforts have been dedicated to making interpolation more accurate and realistic. Besides accuracy, flexibility is also a major concern; in case the result is not accurate or realistic enough, animators can still achieve what they intend to do by providing more manual control.

The most basic type of interpolation is linear interpolation of DBSC strokes. A parameterized non-linear interpolation method is invented to support slow in and slow out principle of animation. Simple interpolation may result in loss of stroke features, such as sharp angles, therefore a feature point-based interpolation is proposed to correctly identify feature points and take them into consideration during the interpolation process. Interpolation may also cause problem in stroke connectivity, so a junction preservation algorithm is applied to remove defects and closing gaps.

3.1 Linear Interpolation

To review the representation of a stroke, a DBSC, Figure 3.1 shows two examples, one open and the other closed.

Figure 3.1: Disk B-spline curve

- (a) Data points (the radii of the two end points are 0)
- (b) An open DBSC by interpolating the data points in (a)
- (c) Data points (the top point is both the starting and the ending point)
- (d) A closed DBSC by interpolating the data points in (c)

To illustrate the interpolation of those two DBSCs in Figure 6 more clearly, we assume that they have the same degree, number of control points and knot vector.

The two DBSCs are mathematically defined as following:

$$\langle D_1 \rangle(t) = \sum_{i=0}^n N_{i,p}(t) \langle P_i; r_i \rangle \quad 3.1$$

$$\langle D_2 \rangle(t) = \sum_{i=0}^n N_{i,p}(t) \langle Q_i; s_i \rangle \quad 3.2$$

The inbetween DBSCs can be generated by interpolating $\langle D_1 \rangle$ and $\langle D_2 \rangle$'s corresponding control points and radii. Given the control points P_j^i and the control radii r_j^i , to get m in-between DBSCs, (supposing linear interpolation is applied) the result is

$$\begin{cases} P_j^i = \left(1 - \frac{i+1}{m+1}\right) \cdot P_j + \frac{i+1}{m+1} \cdot Q_j \\ r_j^i = \left(1 - \frac{i+1}{m+1}\right) \cdot r_j + \frac{i+1}{m+1} \cdot s_j \end{cases} \quad 3.3$$

where $j = 0, 1, \dots, m-1$.

When two disk B-spline curves have different degrees, the curve with the lower degree is converted into higher degree curves through an elevation algorithm. For two curves with different numbers of control points and knot vectors, the one with more control points remains unchanged. Supposing the higher degree curve's number of control points is n and its knot vector is $T = [t_0, \dots, t_{n+p}] = \{a, \dots, a, t_{p+1}, \dots, t_{n-1}, b, \dots, b\}$, we compute these points G_i and radii e_i on the other curve at $t_i, i = p, p + 1, \dots, n$. A new curve of n control points and knot vector T is obtained.

When creating animations, given two key frame drawings composed of DBSCs, inbetweens are generated by interpolating the corresponding pairs of DBSCs.

The correspondence is established by the order of drawing on a tablet or specified by the user. An example is shown in Figure 3.2, in which the middle three inbetween frames are automatically generated by a DBSC-based animation system [34].

Figure 3.2: Inbetweens Generation Using DBSC

3.2 Nonlinear Interpolation

In the previous interpolation case Figure 3.2, linear interpolation was applied to generate the middle three inbetween frames. According to one typical animation

principle "slow in & slow out", linear interpolation is far from adequate to simulate all real life motions. The result is usually dull and lifeless.

Suppose n inbetweens are to be generated; when interpolating point P_i in key frame 1 and Q_i in key frame 2, the corresponding intermediate point T_i is computed as follows:

$$T_i^k = \left(1 - \frac{k}{n+1}\right) \cdot P_i + \frac{k}{n+1} \cdot Q_i \quad 3.4$$

When the resulting animation sequence is played, the character moves with a constant speed, which looks very mechanical. In animation, characters do not usually move at constant speed. Each character can even have its own motion styles, e.g. children move quickly while elderly move slowly and carefully. To achieve uneven changing of shape and motion, the linear interpolation can be modified as follows:

$$T_i^k = (1 - m) \cdot P_i + m \cdot Q_i \quad 3.5$$

where $0 \leq m \leq 1$.

m controls both the shape and the position of the inbetweens. For example, defining m with a cosine curve ($m = \frac{1+\cos t}{2}$, where $t \in [0, \pi]$) achieves the effect of slow in fast out, such as shown in Figure 3.3. In this case, shape and position of the object change at the same rate. There are situations where movements are more complicated. For example, in Figure 3.4, the change of object shape goes from slow to fast while the change of position goes in a fast in slow out manner.

Figure 3.3: Slow In Fast Out Example

Figure 3.4: Different Shape and Position Change Rate

To give the user separate controls over shape and position, we first apply Equation 3.5 to generate the intermediate frames. Subsequently, the elements are translated based on the position parameters specified by the animator.

Therefore it is better to provide users a way to precisely control the interpolation process, in order to get exactly what they want as inbetween frames. Several novel interpolation techniques were proposed to address this issue, including feature based interpolation, user specified trajectory etc., utilizing information such as feature points on strokes and motion of characters.

3.3 Feature-Based Stroke Interpolation

The above modification provides more control over shapes and positions of the inbetweens, but it does not solve all the problems. Figure 3.5 illustrates an example where the interpolation algorithm fails to generate accurate result. The contour of the face is distorted in the inbetweens. The reason is that points are evenly taken in the parametric domain of two DBSCs as corresponding points for interpolation thus some "critical" points do not correspond correctly, e.g. the tip of the nose. This defect causes incorrect inbetween frames from time to time, and may not be tolerated.

Figure 3.5: Linear Interpolation – Facial Silhouette

In animation production, points with high curvature are usually important features of a stroke and should be preserved during inbetween drawing. The rest of this section will discuss in detail how feature points are extracted and how correspondence between feature points is established, as well as the improved interpolation algorithm.

3.3.1 Feature Point Extraction

When drawing a stroke, a sequence of points is sampled by the input device, such as a tablet. To calculate the curvature at a point, the k-cosine technique [29] is applied where the left and the right k^{th} neighbors of a point are used to calculate the angle at the point.

Defining the k-vectors at a point p_i as:

Chapter 3. Interpolation

$\vec{a}_{ik} = (x_i - x_{i-k}, y_i - y_{i-k})$	3.6
$\vec{b}_{ik} = (x_i - x_{i+k}, y_i - y_{i+k})$	3.7

The k -cosine at p_i is:

$\rho_{ik} = \frac{\vec{a}_{ik} \cdot \vec{b}_{ik}}{ \vec{a}_{ik} \vec{b}_{ik} }$	3.8
$\theta = \cos^{-1}(\rho_{ik})$	3.9

where $\vec{a}_{ik}$ and $\vec{b}_{ik}$ are the vectors from p_i to its left k^{th} point and to its right k^{th} point respectively, and θ is the angle between $\vec{a}_{ik}$ and $\vec{b}_{ik}$. The sign of the angle is determined by $\vec{a}_{ik} \times \vec{b}_{ik}$. Feature points are defined as points with a small $|\theta|$.

In [12], a method of morphing using curves is proposed; the two immediate neighboring points are used to calculate the angle at a point i.e. $k = 1$. It is sensitive to noise of small perturbation, which is very common in free hand drawing hence the angle calculated is not accurate. To overcome the problem, we integrate the algorithm for determining region of support in [29] and corner detection algorithm in [2] to choose an appropriate k in order to calculate a more accurate angle value.

For a selected point p_i , we define the length of the chord joining the points p_{i-k} and p_{i+k} as $L(k)$. Let $C_l(k)$ and $C_r(k)$ be the contour lengths from p_{i-k} to p_i and p_i to p_{i+k} respectively. Let $D_l(k)$ and $D_r(k)$ be the lengths of the chords from p_{i-k} to p_i and p_i to p_{i+k} respectively, see Figure 11. Three conditions constrain the value of k , each defined as an equation below.

$\frac{C_l(k) + C_r(k)}{L(k)} > \frac{C_l(k+1) + C_r(k+1)}{L(k+1)}$	3.10
---	------

$$L(k) < L(k + 1)$$

3.11

$$\begin{cases} \frac{C_l(k) - D_l(k)}{C_l(k)} < 0.1 \\ \frac{C_r(k) - D_r(k)}{C_r(k)} < 0.1 \end{cases}$$

3.12

Start with $k = 1$, repeat $k = k + 1$ until neither Equation 3.10 nor Equation 3.11 is satisfied or Equation 3.12 is not satisfied. Equation 3.10 and Equation 3.11 are conditions from [29], which ensure the value of k is in an appropriate range. Equation 3.12 is proposed in [2], which ensures the difference between the arc and the chord is small enough.

Figure 3.6: Calculating Curvature

Empirically, θ is selected to be 140. In Figure 3.7, detected feature points are highlighted in red; negative ones are colored in green with circled numbers. The two ends of a stroke are also treated as feature points. The numbers of feature point from two strokes are equal in this case, but they may differ.

Figure 3.7: Feature Point Extraction

3.3.2 Feature Point Correspondence

Given two strokes S_1 and S_2 , and the feature points on them $\{p_0, p_1, p_2 \dots, p_m\}$ and $\{q_0, q_1, q_2 \dots, q_n\}$, the best match $\{(p_0, q_0), (p_1, q_{c1}), \dots, (p_m, q_n)\}$ can be found where

- $m \geq n$. If the condition does not hold for two strokes, they are switched;
- q_{ci} can be null, i.e. no matching feature point to p_i is found on S_2 ;
- $q_{ci} \neq q_{cj}$ if $i \neq j$ and either q_{ci} or q_{cj} has value;
- $q_{ci} < q_{cj}$, if $i \neq j$ and both q_{ci} and q_{cj} are not null.

The conditions above assure that at most one point on S_2 is matched to one feature point in S_1 and the order of feature points is preserved. The two fixed pairs (p_0, q_0) and (p_m, q_n) mean that the starting points of two strokes are always matched and so are the ending points.

3.3.2.1 Feature normalization

In [12], angle and parameter costs are used to evaluate a match. Given two strokes, among several feature points, each stroke has a point with a sharp turning of

opposite directions e.g. $\theta_1 > 0, \theta_2 < 0$. Both θ_1 and θ_2 are small angles. The two points look very different and should not be matched. However, since the angles are small, $|\theta_1 - \theta_2|$ results in a low cost and the two points are likely to be matched. To solve the problem, normalized curvature $C(p_i)$ and parameter cost $P(p_i)$ are used.

Similar to the method in [12]:

$$C(p_i) = \begin{cases} \frac{1}{2\pi} \cdot (\pi - \text{angle}), & \text{if } p_i \geq 0 \\ \frac{1}{2\pi} \cdot (-\pi - \text{angle}), & \text{if } p_i < 0 \end{cases} \quad 3.13$$

$C(p_i)$ is within $[-0.5, 0.5]$ and any $|C(p_i) - C(q_{ci})|$ is within $[0, 1]$. The parameter cost $P(p_i)$ is within $[0, 1]$ and $|P(p_i) - P(q_{ci})|$ is always in the range $[0, 1]$. Thus the curvature and parameter costs have equal influence in the evaluation of matching.

For a single match (p_i, q_{ci}) between two feature points p_i in S_1 and q_{ci} in S_2 , the cost function is defined as

$$\text{cost}(p_i, q_{ci}) = \begin{cases} |C(p_i) - C(q_{ci})| + |P(p_i) - P(q_{ci})|, & \text{if } q_{ci} \neq \text{null} \\ \lambda_{\text{null}}, & \text{if } q_{ci} = \text{null} \end{cases} \quad 3.14$$

where λ_{null} is the cost of no-match; it is set to the maximum cost, 1.0 in this case, such that feature points have high probability to be matched.

The best match is one that minimizes the total cost. Therefore, a recursive algorithm is applied to solve the optimization problem.

Feature point matching algorithm is defined as follows, matching two sequences of feature points $P_k: \{p_k, p_{k+1}, \dots, p_{m-1}\}$ and $Q_l: \{q_l, q_{l+1}, \dots, q_{l-1}\}$:

Match ($\{p_i | i \in [k, m-1]\}, \{q_j | j \in [l, n-1]\}$)

Chapter 3. Interpolation

```

If  $k = m - 1$  or  $l = n - 1$  (which means a solution is found)
  If  $l = n - 1$ 
    Set all remaining elements in list  $P_j$  to non-match
    For  $a = k$  to  $m - 1$  (inclusive)
      Set  $p_a$  to null
    Compute  $\sum_{i=0}^{m-1} cost(p_i, q_{c_i})$ 
    If  $\sum_{i=0}^{m-1} cost(p_i, q_{c_i}) < COST_{BestMatch}$ 
      Set  $COST_{BestMatch} = \sum_{i=0}^{m-1} cost(p_i, q_{c_i})$ 
  Otherwise find a match for  $p_k$ 
    Set  $p_k$  to null
    Call Match ( $\{p_i | i \in [k + 1, m - 1]\}, \{q_j | j \in [l, n - 1]\}$ )
    For  $a = l$  to  $n - 1$  (inclusive)
      Set  $p_k$  to  $q_a$ 
      Compute  $\sum_{i=0}^k cost(p_i, q_{c_i})$ 
      If  $\sum_{i=0}^k cost(p_i, q_{c_i}) < COST_{BestMatch}$ 
        Set  $COST_{BestMatch} = \sum cost(p_i, q_c)$ 
        Match ( $\{p_i | i \in [k + 1, m - 1]\}, \{q_j | j \in [a + 1, n - 1]\}$ )

```

Take Figure 3.7 for example. The best match is:

$$BestMatch = \{(p_0, q_0), (p_1, q_2), (p_2, null), (p_3, q_3), (p_4, q_4), (p_5, q_5), (p_6, q_6)\}$$

The minimum cost $Cost(S_1, S_2)$, is 1.48.

3.3.2.2 Ambiguity of matching

The matching algorithm is completely automatic and yields satisfying results in most cases. However, ambiguity may occur occasionally such as in Figure 3.8.

Figure 3.8: Feature Points Mismatching Example

Naturally, there can be two matches:

$$BestMatch_1 = \{(p_0, q_0), (p_1, q_1), (p_2, q_2), (p_3, q_3), (p_4, q_4)\}$$

$$BestMatch_2 = (p_0, q_0), (p_1, null), (p_2, null), (p_3, q_1), (p_4, q_4)$$

Both results can be obtained with different λ_{null} in the algorithm and both can be considered correct mathematically. We believe that it is more appropriate to let users choose according to what they want, in case of ambiguity.

3.3.3 Stroke Interpolation Based on Feature Points

Now the interpolation function is accordingly modified. Suppose two DBSCs and their feature points are given, i.e. the control points: $P_i (i = 1, 2, \dots, m)$ and $Q_j (j = 1, 2, \dots, n)$ and the feature points P_{i_k} and $Q_{i_k} (k = 1, 2, \dots, l)$. For the data points of the first DBSC, during interpolation, we use chord length for the knot vector, i.e.

$\begin{cases} u_0 = 0 \\ u_i = u_{i-1} + P_{i+1} - P_i \end{cases}$	3.15
--	------

For data points on the second DBSC, the knot vector is determined by adjusting the chord length through scaling so that these feature points are corresponded in parametric domain.

$\begin{cases} v_0 = 0 \\ v_j = v_{j-1} + Q_{j+1} - Q_j \cdot s_j \end{cases}$	3.16
--	------

$s_j = \frac{\sum_{t=i_k}^{i_{k+1}} P_{t+1} - P_t }{\sum_{t=j_k}^{j_{k+1}} Q_{t+1} - Q_t }$	3.17
---	------

where $j = j_k + 1, j_k + 2, \dots, j_{k+1}$ and $k = 1, 2, \dots, l$.

By applying the method above, inbetween result of Figure 3.7 is illustrated in Figure 3.9.

Chapter 3. Interpolation

Figure 3.9: Feature-Based Stroke Interpolation – Facial Silhouette

Another example is shown in Figure 3.10, together with result of the previous method.

(a) Linear interpolation (distortion highlighted in red circles)

(b) Feature-based stroke interpolation

Figure 3.10: Inbetween Generation Results – Chicken

It is obvious that feature-based stroke interpolation reduces distortion and stroke features are better preserved.

3.4 Preserving Junction in Interpolation

In key frame drawings, strokes intersect each other. More often than not, they form regions that usually represent parts of an object or a character. In Figure 3.11 the head is composed of about 60 strokes. The strokes intersect and form over 30 closed regions. The stroke of the ear contour connects to that of the face contour. (These intersection points are called "junctions".) In such cases, a junction point must fall on one end of either stroke. Therefore junctions should be preserved, otherwise the corresponding region in inbetween frames will be open and cause problem in the coloring process in a later stage.

Figure 3.11: Example of Regions

One property of our interpolation generation is that, if the intersection point of two strokes falls on ends of both strokes, the interpolated stroke will always connect at those ends. If one point is an end point and the other is not as shown in Figure 3.11, they may not remain connected in inbetween frames, because a stroke can reform in a very flexible way in a sequence of frames. It is very important to preserve this connection property, to maintain correctness of the drawing and to facilitate subsequent coloring process. A region must be closed to apply coloring in its interior; any gaps on the boundary will result in color leaking. Though small gaps can be fixed by some gap closing algorithms, it is much more effective and more efficient to avoid it at generation time than fixing it afterwards.

3.4.1 Intersection Detection

Detecting a junction point is mathematically to find the intersection of two DBSCs. The first step of detecting intersections of two DBSCs is finding their center

curves' intersections. A DBSC's center curve is a non-uniform rational B-splines (NURBS), so intersection of two DBSCs can be rephrased as:

Two DBSCs intersect if there are positions where control disks on the two DBSCs overlap.

Therefore the problem is converted into detection of intersection of two B-splines. We have proposed an efficient method to compute the intersections. Given two DBSCs with a small number of control points, we first convert them to piecewise Bezier curves.

For a DBSC with control disks $\{(P_0, r_0), (P_1, r_1), \dots, (P_{n-1}, r_{n-1})\}$, the control points of the converted piecewise cubic Bezier curve are:

$$\{Q_{0,0}, Q_{0,1}, Q_{0,2}, Q_{1,0}, Q_{1,1}, Q_{1,2}, \dots, Q_{n-2,0}, Q_{n-2,1}, Q_{n-2,2}, Q_{n-1,0}\}$$

$\{Q_{i,0}, Q_{i,1}, Q_{i,2}, Q_{i+1,0}\} (i = 0, 1, \dots, n-2)$ are the four control points of a cubic Bezier curve.

A sampling process in parametric domain is applied to compute the disks on the DBSC with a dense rate according to the length of the piecewise Bezier curve, where dense rate is defined as average number of points on a unit length segment of stroke. By linking consecutive control points of the piecewise Bezier curve, we calculate the total length of all segments, l , as the approximate length of the piecewise Bezier curve. Assuming sampling step is d , the number of disks n will be:

$n = \text{floor}\left(\frac{\text{length}}{\text{step}}\right) + 1$	3.18
--	------

Since the total length of all segments are longer than the real length of the Bezier curve, the number of sampling rate is higher than desired, which is adequate. The sampled knots in the parametric domain are calculated as follows:

$$knot_i = \frac{i}{n-1} (i = 0, 1, \dots, n-1) \quad 3.19$$

For animation drawings, a sampling step can be adjusted for different styles of animations or even for each individual animation. For this case, it is empirically preset to 0.5. The disks on the DBSC $knot_i$ ($i = 0, 1, \dots, n-1$) can be represented mathematically as $\{(C_0, r_0), (C_1, r_1), \dots, (C_{n-1}, r_{n-1})\}$, where C_i and r_i are the center and the radius of the i^{th} disk.

Now we compute the distance of every pair of disks on the two input DBSCs, S_1 and S_2 .

$$d_{ij} = |C_{1,i} - C_{2,j}| - r_{1,i} - r_{2,j} \quad 3.20$$

where $(C_{1,i}, r_{1,i})$ and $(C_{2,i}, r_{2,i})$ are two disks on S_1 and S_2 respectively.

If $d_{i,j} \leq 0$, S_1 and S_2 are intersected at these two disks.

Figure 3.12 shows the intersections of two DBSCs computed by the method above.

Figure 3.12: Stroke Intersection

The sampling step should be set smaller if more accurate result is desired, which will unfortunately result in high computational cost. Points on the entire range of the curve need to be computed and the distance between every two successive points needs to be calculated. For higher speed, the Bezier subdivision method [15] is applied. The Bezier subdivision method relies on the convex hull property of Bezier curves: a rational Bezier curve, with non-negative weights, lies within the convex hull of the control points. After a DBSC is converted to piecewise Bezier curve, we can get the convex hull for each Bezier curve using Graham's Scan algorithm [9], as shown in Figure 3.13.

Figure 3.13: Bezier Curve and Convex Hull

Given two piecewise Bezier curves, we compare their convex hulls. Any pair of strokes with non-overlapping convex hulls is definitely not intersecting and hence can be omitted. In contrast, overlapping convex hulls indicate possible intersection, meaning that the intersection detection algorithm described previously should be performed. Non-overlapping convex hulls are quite common for non-intersecting stroke pairs; therefore the computation time is greatly reduced. (Note that the same algorithm can be applied to detect self-intersection of a single DBSC as well.)

The parametric value and coordinates of every intersection point can be obtained by applying the method. They will be used in the next step.

3.4.2 Junction Preservation

Once junction points in key frames are detected, it is then possible to preserve them in inbetween frames. Assume strokes S_1 and S_2 form junctions in both key frames. One point is an end point P of S_1 and the other is a non-end point Q of S_2 . Either of the following two situations is possible.

1. S_1 and S_2 do not intersect, which means a gap occurs between P and Q (where Q is the a point on S_2 that is closest to P) in inbetweens, as shown on the left of Figure 3.14 labeled with number 1. In this case, we change the position of P .
2. S_1 and S_2 intersect at point R and a small artifact occurs. In this case, we remove the artifact i.e. the segment of S_1 from S to R , as illustrated on the right of Figure 3.14 labeled with number 2.

Figure 3.14: Types of Artifacts in Interpolation Result

In both situations, S_2 remains unchanged so that its shape is preserved.

Now for case 2, the correct position of P in inbetweens needs to be calculated. From the intersection detection method, the parametric value of the intersected point Q on S_2 is obtained, which is t_1 in key frame 1 and t_2 in key frame 2. Based on the interpolation function, the position of P , P_k , in inbetween I^k ($k = 0, 1, \dots, n - 1$) can be computed, assuming n inbtweens are generated. The parametric values of Q in I^k can also be computed as follows:

$$t_k = t_1 \cdot \left(1 - \frac{k+1}{n+1}\right) + t_2 \cdot \frac{k+1}{n+1} \quad 3.21$$

From the parametric value, we can compute the point positions of Q, Q_k , in inbetween I^k . To preserve the junction, P_k must be equal to Q_k . Therefore, P_k is set to Q_k ($k = 0, 1, \dots, n - 1$) so that the two points always join in the inbetweens. The result (right) is shown in Figure 3.15, in comparison with the result obtained without junction preservation (left).

Figure 3.15: Junction Preservation Example

3.5 Summary

In this chapter, linear and non-linear interpolation was explained. Feature points and junction points were preserved using several algorithms invented. However, an animation system cannot generate inbetween strokes without key frames, which are input by users. Moreover it is necessary to display the generated stroke, so that users can see whether they are correct and then adjust when necessary. The next chapter will explain in detail how user inputs are converted to strokes, and how strokes are displayed to user.

Chapter 4

Curve Fitting and Rendering

When animators draw a stroke with their tablets or mice, they see a nicely shaped, fine rendered curve on the screen. Between the moments they draw a stroke and they see the stroke on the screen, two major operations have taken place – input handling and stroke rendering. Both operations should be carefully handled in order to obtain high quality output.

User input with both tablets and mice are discrete points. The challenge here is to convert discrete points into smooth curves. A curve fitting algorithm is developed to generate curves that are smooth and yet not too distinct from user input points.

Displaying a NURBS curve is basically not very difficult to achieve. However, displaying a DBSC, which has non-uniform thickness, is not as easy. A border estimation rendering algorithm is proposed to render stroke with high quality and at high speed.

4.1 Curve Fitting

Handling user input is critical in an animation system. Our objective is to make the drawing interface as simple and intuitive as possible. Both the drawing tool and tracing tool are provided to animators. For tracing, there are no technical difficulties except some intensive GUI implementation. The drawing tool, however, need to be very advanced, and that requires tremendous amount of research.

Curves are represented by DBSC in this case, which are in turn represented with a sequence of control points and knots. However, usually animators are neither interested in nor good at manipulating control points. They do not care how the strokes are modeled, instead they are concerned with whether the strokes shown on the screen are exactly what they intend. Therefore two issues should be addressed in this case: accuracy and simplicity of control. Accuracy means strokes shown should be at least close to the animators' expectation if not exactly the same. Besides that, animator will not like to spend unreasonably long time for just getting a single stroke perfectly right. So we introduced a drawing mechanism without any explicit control points. Animators just need to draw a stroke, and it is nicely fitted with a DBSC. No control points are involved, from an animator's perspective. Control points are generated automatically by the program right after an animator draws a stroke and before the stroke is actually displayed.

The following diagram shows a curve created from 6 discrete points. Note that for simpler illustration, Figure 4.1 shows only the central lines of the DBSC, meaning that the radius parameter is not reflected.

Figure 4.1: Bezier Curve and Its Control Points

In the above example, the 6 black points (P0 to P5) are the sampled user input, and the 8 red circles (CP0 to CP7) are the control points automatically generated to fit the 6 points with a smooth DBSC. Note that the curve perfectly passes through all 6 points obtained through user input.

Figure 4.2: B-Spline and Its Control Points

The curve can then be converted to a piecewise cubic B-spline curve shown in Figure 4.2, with a degree reduction algorithm [23]. The piecewise B-spline is generated to facilitate rendering, because it is much easier to render the curve segment by segment than as one single curve. Figure 4.3 shows the relation between control points of DBSC and those of the piecewise B-spline [23].

Figure 4.3: Control Point Relationship

4.2 Curve Rendering

Now that the input has been converted to DBSCs, it comes to the stage of rendering. Stroke rendering is extremely important in our animation system, because it directly affects the visual appeal of characters and sets in the animation.

Two rendering engines have been developed. The first one is OpenGL-based; it tessellates the vector data into a large number of small triangles and then uses standard OpenGL API's to render them. The triangles have to be small enough so that the final output does not expose its polygonal look. Due to the large number of triangles required during tessellation, the rendering is rather inefficient in terms of time. It is critical since the strokes are displayed in real-time. The second method, which uses Graphics Device Interface (GDI), is much simpler in design. It uses ovals as atomic elements; a stroke is graphically deformed into large number of filled ovals. The centers of the ovals align with the center curve of the stroke; and variable thickness of a stroke is simulated by varying the radii of the ovals. This method preserves the smoothness of strokes well, but is again very slow and hence causes glitches during run-time.

We introduced a new rendering engine to resolve the issue. It has many advantages over its predecessors, including better time efficiency, anti-alias support, higher curve smoothness, and possible styling.

4.2.1 Border Estimation Rendering

A stroke, defined by a DBSC, can have variable thickness. Therefore, a stroke is an area rather than a curve. The conventional way of rendering an area is by obtaining its boarder first and then filling the inner portion surrounded by the border. The area of a stroke is defined as the closed region formed by two border curves that are tangent to all the center circles. Figure 4.4 illustrates a stroke, whose border is formed by two curves connected at end points.

Figure 4.4: Stroke Illustration

Here the term "center circles" refers to the set of circles whose centers are on the center curve of the stroke with interpolated radii. It is too costly to calculate the exact boundaries of a stroke, as the number of center circles is infinite. Therefore an approximation algorithm is desirable.

4.2.2 Approximating Border

Given a stroke defined by a set of skeleton points $\{P_i, i = 0 \dots n - 1\}$, where n is the number of knot points. The central curve is the piece-wise B-spline curve fitted based on the skeleton points mentioned in the previous section. It passes through every knot point of the stroke and is defined by a set of control points $\{CP_j, j = 0 \dots 3n - 1\}$, where n is the number of skeleton points. An example of a stroke's central curve together with skeleton points and control points is shown in Figure 4.5.

Figure 4.5: Stroke & Control Points

Note that P_i has the same value as CP_{3i} for any i in the valid range.

The border can be approximated with the following algorithm. In the following pseudo code, r_i is the radius of the disk at point P_i .

Chapter 4. Curve Fitting and Rendering

```

For i = 0 to n-1

    Find the tangent to central curve at Pi

    Find the line LI perpendicular to central curve at Pi

    Find the two points Ri and Li on LI ri away from Pi,
    (where ri is the radius of DISKi)

Use the ordered set {Ri, i=0...n-1} to form the right border

Use the ordered set {Li, i=0...n-1} to form the left border

Combine the left and right borders to get a closed stroke region
    
```

Table 4.1: Border Approximation Pseudo Code

The tangent vector m_i at point P_i can be calculated as

$$m_i = \frac{CP_{3i+1} - CP_{3i-1}}{\|CP_{3i+1} - CP_{3i-1}\|} \quad 4.1$$

L_i can then be calculate as

$$L_i = P_i + \frac{r}{\|m_i\|} \text{Rotate}_{\frac{\pi}{2}}(m_i) = P_i + \frac{r}{\|m_i\|} \begin{bmatrix} -y_{m_i} \\ x_{m_i} \end{bmatrix} \quad 4.2$$

R_i can be calculated in a similarly way.

$$R_i = P_i + \frac{r}{\|m_i\|} \text{Rotate}_{-\frac{\pi}{2}}(m_i) = P_i + \frac{r}{\|m_i\|} \begin{bmatrix} y_{m_i} \\ -x_{m_i} \end{bmatrix} \quad 4.3$$

The set of $\{L_i, i = 0 \dots n - 1\}$ will be used to create a Bezier curve and $\{R_i, i = 0 \dots n - 1\}$ to create the other. In most cases, these two curves will not form a closed region because the first and last skeleton points of the stroke are unlikely to have zero radii. Two extra points are added into each list, L_{-1} and R_{-1} in the front and L_n and R_n at the end. Their values can be calculated as follows.

$$L_{-1} = R_{-1} = \frac{L_0 + R_0}{2} \quad 4.4$$

$$L_n = R_n = \frac{L_{n-1} + R_{n-1}}{2} \quad 4.5$$

Now that the head points of the two sides are the same and so are the tail points, the two curves $\{L_i, i = -1 \dots n\}$ and $\{R_i, i = -1 \dots n\}$ guarantee a closed region.

The approximation algorithm has a low complexity of $O(n)$, where n is the number of skeleton points. Test result shows that the new method is efficient and sufficiently fast for real time rendering. The approximation results are fine in most cases, but there are cases where it fails or yields very poor result. Therefore a more advanced sampling algorithm is desirable.

4.2.3 Curvature Weighted Sampling

The rendering quality is closely related to the density of skeleton points; the more skeleton points there are, the higher the rendering quality is. It is observed that rendered strokes tend to distort around high curvature points. One solution to this is to increase the sampling density, e.g. double or triple the number of skeleton points or even more. Unfortunately, the stroke rendering performance declines significantly when sampling density increases.

In order to keep rendering quality high yet preserve rendering performance, a sampling method that takes curvature information as a parameter is proposed. In this method, sampling density is proportional to curvature.

Since the stroke has already been converted to a piecewise B-spline curve, its curvature can be obtained by calculating the curvature of each segment, which is a cubic Bezier curve.

For any 2D cubic Bezier curve $P(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}, t \in [0,1]$ in a plane, its curvature at t is defined as:

$$k(t) = \frac{x'(t)y''(t) - y'(t)x''(t)}{\sqrt{(x'^2(t) + y'^2(t))^3}} \quad 4.6$$

where $P(t) \equiv \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$, $P'(t) \equiv \begin{pmatrix} x'(t) \\ y'(t) \end{pmatrix}$ and $P''(t) \equiv \begin{pmatrix} x''(t) \\ y''(t) \end{pmatrix}$

$P(t)$ is a cubic Bezier curve, so it can be represented by the following parametric equation, assuming P_0, P_1, P_2 and P_3 are its four control points.

$$P(t) = P_0(1-t)^3 + 3P_1(1-t)^2t + 3P_2(1-t)t^2 + P_3t^3 \quad 4.7$$

Then the first and second order derivatives can be calculated.

$$P'(t) = -3P_0(1-t)^2 + 3P_1(1-t)^2t + 3P_2(1-t)t^2 + 3P_3t^2 \quad 4.8$$

$$P''(t) = P_0(6-6t) + P_1(-12+3t) + P_2(12-3t) + P_3 \cdot 6t \quad 4.9$$

Therefore equation of $k(t)$ can be obtained, once the values of P_0, P_1, P_2 and P_3 are given. The density at t can also be obtained by multiplying a curvature-to-density factor, which can be either preset or from user inputs.

$$density(t) = k(t)C_{curvature \rightarrow density} \quad 4.10$$

The value of $C_{curvature \rightarrow density}$ is based on experiments. The density is then used to guide the sampling process, for higher density parts, more points are sampled.

4.3 Examples and Analysis

This section shows experiment results of three different rendering algorithms. In Figure 4.6 and Figure 4.7, (a) is the old rendering result; (b) is the result of newly proposed algorithm with anti-alias and curve fitting and (c) is the result of (b) further improved by applying curvature weighted sampling.

Figure 4.6: Comparing Results of Three Rendering Algorithms

The rendered strokes with their knot points highlighted are shown in Figure 4.7.

The knot points are the control points that locate on the strokes. They dominantly determine the shape of a stroke. The points in (a) are not only knot points, but also the exact points drawn by the animator using a mouse or a tablet.

Figure 4.7: Results of Three Rendering Algorithms with Knot Points

It is obvious that the old algorithm without filtering or anti-aliasing is must less appealing to look at, although it preserves every point the uses have input. The result in (b) looks smooth and refined, but it is too far from the user input and may result in undesired strokes. (c) is smooth and it preservers features such as the sharp angle in the middle of the stroke, while still maintaining the conciseness of data.

4.4 Summary

Chapter 3 and this chapter have covered these topics, converting user input to strokes in mathematical forms, generating inbetween strokes from those key strokes, and then display both key and inbetween strokes to provide users visual feedback. To obtain more appealing animation, one major process is missing and that is coloring, which will be discussed in detail in the following chapter.

Chapter 5

Regions and Colors

The coloring process is one of the most laborious and time-consuming steps in animation production, as mentioned in Chapter 2. Therefore computer assisted coloring is desirable. In our vector based 2D animation system, inbetween frames can be colored automatically, once the key frames are manually colored.

There were two major challenges in automatic coloring, identifying regions and matching regions in different frames. A process has been developed to identify the regions, which includes converting a frame into graph, finding cycles in that graph, mapping cycles to regions and finally finding primary regions. To match regions in different frame, an algorithm utilizing geometric and topological information was created.

5.1 Region Definition

A frame mainly consists of strokes. Strokes may intersect with themselves and/or with one another. Strokes are divided into segments by intersection points, each of which can be the intersection of two or more strokes. A stroke segment can be defined by a stroke and two adjacent intersection points on the stroke. Note that intersection of strokes refers to the intersection of central curves of strokes and regions are defined based on them too. Otherwise, the intersection of two strokes would be a region instead of a single point, due to the variable thickness of DBSC.

$Seg_{ij} = (S_i, j, j + 1), i = 1, \dots, m, j = 1, \dots, n_i$	5.1
--	-----

where Seg_{ij} is the j^{th} segment on S_i , S_i is the i^{th} stroke in a frame, m is the number of strokes in the frame and n_i is the number of segments in S_i . Note that only strokes within the same frame are considered for possible intersection. Frame index subscription is removed from the notations for simplicity.

In our animation system, a region is defined as an ordered set of stroke segments $\{Seg_{ij}, i = 1 \dots m, j = 1 \dots n_i\}$. Segments in the sequence appear in the same order as travelling along the border of a region. The order of travelling can be either clockwise or counter-clockwise.

5.2 Region Detection

It is not easy to do region detection based on the stroke segments and intersection points directly. A better way is to convert the structure into a graph that is suitable for traversal and search. Regions are then equivalent to cycles in the graph.

5.2.1 Converting to graph

A set of intersecting strokes contains stroke segments and intersection points. An undirected graph is defined by nodes and edges. It seems quite straight forward to simply map intersection points to nodes and stroke segments to edges.

However, that conversion technique is not capable of handling some cases. Take a self-intersecting stroke for example; the converted graph has one node and one undirected edge looping back to the node. Looping-back makes searching of cycles difficult, because it is hard to uniquely identify a path. It will be even worse if self-intersection points of two strokes coincide. With the conversion technique to be

introduced, every cycle in the converted graph is formed and uniquely identified by at least 3 nodes, presenting no ambiguity at all.

Figure 5.1: Self-Intersecting Stroke Converted To Graph

We proposed a more robust graph conversion method. In this method, intersections and stroke ends become nodes in the graph, and every adjacent intersection and segment pairs are also converted to nodes. As shown on the left side of Figure 5.1, a self-intersecting stroke is converted into a graph with 3 nodes and 3 edges show on the right. P1 alone is converted to node N1; and the combination of P1 and S1 (with direction) is converted to N2; while the combination of P1 and S2 is mapped to node N3.

A more realistic example is illustrated in Figure 5.2. In this example, 3 strokes intersect with each other, generating 4 intersection points and 10 stroke segments. Node N1.0 is converted from intersection point IP1, node N1.1 corresponds to the intersection and segment pair (IP1, S2.2) and N2.2 to (IP2, S3.2).

Figure 5.2: Strokes Converted To Graph

As long as regions are concerned, we can leave out the terminal segments (those formed by one intersection point and one end of a stroke). The strokes that do not intersect any other can also be removed. In the converted graph, that means the nodes with one edge can be removed. This pruning is done in as many passes as

necessary till no such nodes are present, because one round of pruning may render other nodes qualified for removal.

The removable nodes are illustrated with a lighter color and connected to the rest of the graph by dotted lines in Figure 5.2, i.e. N1.4, N2.4 and N4.4 etc. A simplified graph is shown in Figure 5.3.

Figure 5.3: Pruning

The converted graph has the following properties:

1. There is at most ONE edge between any arbitrary pair of nodes.
2. Every node has at least TWO edges connected to it, which implies the graph is always connected with no isolated nodes.
3. Every cycle is formed and uniquely identified by at least THREE nodes.

5.2.2 Finding Regions

Once the graph is obtained, we can apply depth first search (DFS) to find all the cycles in the graph. The reason for opting for DFS over breadth first search (BFS) is that it has the same time complexity as BFS but consumes less memory.

Beginning with an arbitrary node N_0 from the graph, all the cycles can be found by invoking *FindCycle*(N_0). The pseudo code of *FindCycle* is defined below. The path here has first-in-last-out (FILO) behavior just as a stack does.

```

Global: path = zero-length path, cycle collection = empty

FindCycle(node  $N$ )

    For each adjacent node of  $N$ ,  $N_i$ 

        Append  $N_i$  to the global path

        If path contains cycle, copy path to the cycle collection

        Otherwise, FindCycle( $N_i$ )

    Remove  $N_i$  from the end of path

```

Table 5.1: Find Cycle Pseudo Code

This algorithm ensures all the possible cycles in the graph are found and put into the cycle collection after successful execution of *FindCycle*(...). Given the graph in the upper right corner of Figure 5.4, the DFS process is tree-structured as shown in the figure. Note that the order of child nodes being processed can be arbitrary.

Figure 5.4: Searching Tree

In this case, exhaustive depth first search results in the following cycle collection (each cycle mapped to one branch in the above tree from left to right):

$$\langle 1231 \rangle, \langle 2342 \rangle, \langle 12431 \rangle, \langle 2432 \rangle, \langle 1321 \rangle, \langle 3243 \rangle, \langle 13421 \rangle, \langle 3423 \rangle$$

Cycles in the collection are not unique; we need to run a duplication removal process to ensure uniqueness. Take the previous case for example, the 8 cycles can be separated into 3 groups; each containing a set of equivalent cycles.

$$\langle 1231 \rangle \equiv \langle 1321 \rangle$$

$$\langle 2342 \rangle \equiv \langle 2432 \rangle \equiv \langle 3243 \rangle \equiv \langle 3423 \rangle$$

$$\langle 12431 \rangle \equiv \langle 13421 \rangle$$

Therefore there are only 3 unique cycles $\langle 1231 \rangle$, $\langle 2342 \rangle$ and $\langle 12431 \rangle$, within the graph, and those three cycles correspond to 3 unique regions.

5.2.3 Finding Primary Regions

In the previous example, 3 unique regions were identified. However the region $\langle 12431 \rangle$ contains both $\langle 1231 \rangle$ and $\langle 2342 \rangle$. In the coloring process, complex regions like $\langle 12431 \rangle$ are not considered. Regions that do not contain any smaller regions and do not overlap with any other regions are more important. Those regions are named *primary regions*. The goal of this section is to find all primary regions.

In the case of Figure 5.2 on page 57, region 1 $\langle \text{IP1}, \text{S2.2}, \text{IP3}, \text{S3.2}, \text{IP2}, \text{S1.2}, \text{IP1} \rangle$, region 2 $\langle \text{IP3}, \text{S2.3}, \text{IP4}, \text{S1.3}, \text{IP2}, \text{S3.2}, \text{IP3} \rangle$ are primary regions while region 3 $\langle \text{IP1}, \text{S2.2}, \text{IP3}, \text{S2.3}, \text{IP4}, \text{S1.3}, \text{IP2}, \text{S1.2}, \text{IP1} \rangle$ is not.

Chapter 5. Regions and Colors

To find primary regions, the topological relationship between any two regions is utilized. Let $\{R_i\}$ be the set of all regions, primary and non-primary, and $\{PR_j\} \subset \{R_i\}$ be the set of all primary regions. The process of primary region searching is described by the following pseudo code.

```

Copy all the elements of  $\{R_i\}$  to  $\{PR_j\}$ 

For every ordered pair of regions  $(R_1, R_2)$  in  $\{R_i\} \times \{R_i\}$  where  $R_1 \neq R_2$ 

    Take an arbitrary point P within region  $R_1$ 

    Draw an arbitrary straight line L crossing P

    Calculate the intersection points between L and  $R_2$ 

    Count intersection points on each side of P, resulting number  $C_1$  on one side and  $C_2$  on the other side

    (For cases where the line is tangent line,
        one tangent point is counted as two intersection points)

    If  $C_1$  and  $C_2$  are both even,

        P is not in  $R_2$ , indicating  $R_2$  doesn't contain  $R_1$ 

        (Although overlapping is still possible)

    If  $C_1$  and  $C_2$  are both odd,

        P is in  $R_2$ , indicating  $R_1$  and  $R_2$  share common area

        Which means at least one of them is NOT primary

        Calculate  $D_j$  ( $j=1, 2$ ), distance between the two
            outer most intersection points of  $R_j$  and L

        Remove  $R_j$  ( $j=1, 2$ ), with larger  $D_j$  from  $\{PR_j\}$ 

For any closed region  $R_2$ , other cases are not possible

```

Table 5.2: Searching Primary Regions Pseudo Code

At the end the set $\{PR_j\}$ will be an exhaustive set of all primary regions in $\{PR_i\}$.

For cases where the line is tangent line, one tangent point is counted as two intersection points, just as shown in case A of Figure 5.5.

Figure 5.5: Tangent Line

Scenario A: S1 is smooth (at the tangent point), as the figure labeled **A**. A straight line L intersects with a stroke S1 at point P. If the stroke has C^1 or higher continuity at point P, the derivative of stroke S1 can be calculated and hence it can be used to determine if L is a tangent line at P. As described in the algorithm, this point P is counted as 2.

Scenario B, S1 is not smooth (at the intersection point) as shown in **B**. Because the nature of our stroke representation, a segment is always smooth. If S1 has C^0 or lower continuity at point P, the stroke must consist of two segments. When we calculate the intersections, both intersection between L and S1.1 and intersection between L and S1.2 will be found. Now, I was not accurate to put such a point as “tangent point”. However in such cases, number of intersection point will result in 2 instead of 1.

5.3 Region Matching

Once all the regions inside key frames are detected, they should be matched for automatically coloring. Our current region matching technique is based the stroke segments that form the regions. Since strokes are already matched for

interpolation, mentioned in Chapter 3, this region matching algorithm does not incur much extra time.

For any two frames with the same number of strokes, each of them contains a set of regions, defined as $\{R_i\}$ and $\{Q_j\}$ respectively. For an arbitrary region r in $\{R_i\}$, if r has exactly the same stroke segments as a region q in $\{Q_j\}$ (ignoring the order of stroke segments), r is matched to q . Figure 5.6 shows two of examples region matching.

Figure 5.6: Region Matching Examples

In some cases, there are just no straight forward matching of regions between two frames. For example, Figure 5.7 below shows a case that no possible matching is absolutely logical. Correctness of matching is purely dependent on the interpretation of its creator.

Figure 5.7: Region Not Matched

The regions will be matched correctly only if the topological relationship of the strokes does not change across frames. Since the animation system requires the users to input strokes and regions with correct topological relationship, those exceptional cases are not discussed in this thesis.

5.4 Region Matching Examples

An example of region detection and region matching will be shown in this section.

Figure 5.8 shows the octopus example, with all the strokes, intersections and regions labeled.

Figure 5.8: Region Matching Example Labeled

Strokes are labeled from A to J in red; X and Y are intersection points. Note that all the self-intersection points are not labeled because in the case of self-intersection, a stroke can uniquely identify a region.

Regions can then be detected using the algorithms in section 5.2. They are labeled in green from O to S, where, P is formed by stroke G; Q is formed by H; R by I and S by J; O is formed by region of stroke A and B subtracting those containing region P, Q, R and S;

Figure 5.9: Region Matching Example Colored

The first octopus in the first frame (the left most one of Figure 5.9) was colored manually, and the following region were colored automatically by the system according to the region matching result. As this example does not contain any to changes in topology, the regions were accurately matched and colored.

5.5 Summary

Algorithms proposed in this chapter do not cover all cases. However, during our testing, it is found that many cases that failed during the region matching are ambiguous cases, which a machine without artistic feeling is not able to determine. In those cases, human interactions are required.

Now that the regions are all matched, only one region in a corresponded region set needs to be colored and coloring of the rest regions are automatic. The time saved will be $\frac{1}{n} - overhead$, where n is the size of the sequence. Therefore given a large n , the overhead induced by manual matching work become insignificant, and it truly saves a lot of time and effort.

However, the coloring process is very computation intensive. It usually takes a long time for a workstation to generate a color frame. It affects the animators' working efficiency, since they have to wait while their workstation is busy computing coloring result. That is why a distributed framework is introduced in the next chapter to improve performance.

Chapter 6

Distributed Interpolation and Rendering

Automatic inbetween generation cuts down the time and cost of animation production. However, given the fact that a 90-minute animation feature may easily consist of more than dozens of thousands of key frames, generation of inbetween remains a time consuming and computation-intensive process. In addition, each frame of an animation may consist of several layers for characters, backgrounds, and other elements. Each layer has to be rendered separately and then composited together. Therefore many 3D animation companies build their own render farms, which are clusters of interconnected computers. There are some commercial render farms available online, such as ResPower [46] and RenderIt [45]. Going beyond the cluster and reducing the rendering time further, the Grid Computing, or simply called Grid, is introduced and realized by linking up clusters across different locations or areas. Grid computing may help smaller companies compete with bigger players in major markets. One such example is AXYZ Animation Inc. [37], a Toronto-based firm with 35 employees, creating special effects for TV ads. The Sun's Grid Engine [48] gives AXYZ smoother workflow and better control of projects.

While most Grid applications focus on 3D CGI and rendering, we propose a distributed 2D animation rendering framework by leveraging on our inbetweening technology. First, animators vectorize and decompose key frames into a number of layers to build correspondences of strokes when occlusion occurs. The framework then allows animators to submit layers of key frames to a cluster from their

workstations. The cluster will handle two tasks, inbetween generation and composition of various layers. Subsequently, the final rendering results, i.e. a sequence of frames (including composite key frames and inbetweens), are sent back to the animators for them to view at their workstations and touch-up if necessary. Our experimental results show that the framework is able to improve the speed of frame generation and rendering hence the productivity of animation, without degrading the interaction between the animator and the application. Moving forward, we will extend this framework from cluster to the Grid, which we believe will impact the whole 2D animation industry.

The following part of this chapter will cover details on the rendering framework, including both algorithm and implementation. Experiment results are presented at the end of the chapter.

6.1 Frame Generation Framework

To cut down the rendering time for 3D CGI, many companies either build their render farms, or use commercial online farms, or adopt the Grid computing. Similarly, we propose a frame generation framework for 2D animation production, aiming to increase animator's productivity and allow them to focus on creativity work rather than labor-intensive drawing task. The framework is based on the concept of a cluster for distributed computing and its structure is illustrated below in Figure 6.1.

Chapter 6. Distributed Interpolation and Rendering

Figure 6.1: Distributed Rendering Framework

First, an animator works on the *client program* running in the *Workstation* to decompose key frames into layers (if necessary). The program vectorizes and corresponds strokes of the key frames. After the "*render by server*" command is triggered in the *client program*, all strokes together with other data (layer, DBSC, etc.) are serialized, compressed and streamed over local network to the distributor. In the mean time, the *client program* is locked so that no further change can be made to the animation sequence until the rendering result is received. Upon receiving the rendering request, the *distributing service* on the *distributor* analyzes the work load, divides the task into subtasks and distributes them to the *cluster* nodes. Each node has already been pre-loaded with our 2D animation *rendering engine*, which consists of two main modules, inbetweening and merging. The *distributor* waits till every node completes its tasks and sends back the results, then streams them back to the animator who has submitted the rendering request.

When the rendering job (inbetweening and merging) is completed in the *cluster*, a sequence of frames (in the form of raster images) is sent back to the *workstation*. Here, images are compressed and represented in the PNG (portable network graphics) format. It is the best format for this application, because 2D animation

outlines usually have very limited number of colors and contain large mono-color areas, which can be highly compressed by the PNG encoder.

6.2 Rendering Algorithm and Complexity

An animation sequence is represented as a collection of layers. A layer contains a set of frames; and a frame in turn is comprised of strokes, which can be combined into groups if necessary. So a sequence can be considered as a matrix of strokes, as shown in Equation 6.1, where N_s and N_F are the numbers of strokes and composite frames respectively. S_{ij} denotes the i^{th} stroke in the j^{th} composite frame.

$$\begin{bmatrix} S_{11} & S_{12} & \cdots & S_{1N_F} \\ S_{21} & S_{22} & \cdots & S_{2N_F} \\ \cdots & \cdots & \cdots & \cdots \\ S_{N_s1} & S_{N_s2} & \cdots & S_{N_sN_F} \end{bmatrix} \quad 6.1$$

Each column of the matrix constitutes a composite frame, and each row represents a set of corresponded strokes, which is defined as CS_i . The rendering process of 2D animation consists of two main tasks: inbetweening (or interpolation) and merging, which are shown in Table 6.1,

Render (...)	
For each set of corresponded strokes CS_i in the animation For each pair of successive strokes in CS_i Generate given number of inbetween strokes	Inbetweening
For each composite frame F_j in the animation Create a buffer image buf with specified output dimension For each frame f in F_j Remove occluded strokes or stroke segments Merge f with buf (performance is proportional to output dimension)	Merging

Table 6.1: Rendering Process Pseudo Code

Chapter 6. Distributed Interpolation and Rendering

where CS_i denotes the i^{th} set of corresponded strokes $\{S_{ij}, j = 1, 2, \dots, N_F\}$, and F_j is the j^{th} composite frame. An example of the interpolation and merging process is illustrated below.

	Frame 1 (Key)	Frame 2	Frame 3	Frame 4	Frame 5 (Key)
Composite Frame Before Merging	(0, 1)	(0, 2)	(0, 3)	(0, 4)	(0, 5)
Layer 1	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)
Layer 2	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)
Layer 3	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)
Layer 4	(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)
Rendering Result	(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)

Table 6.2: Rendering Process Decomposed

As shown in Table 6.2, each composite frame $(0, m)$ is composed of four layers, and the frame at each layer n is represented as frame (n, m) , where n and m are indices of layers and frames respectively. For example, frame $(0, 1)$ is the

composition of frames $(1,1), (2,1), (3,1), (4,1)$. Frames $(n, 1)$ and $(n, 5)$ are the input key frames, which are interpolated to obtain inbetween frames at each layer. After inbetweening, each frame $(5, m)$ is obtained by merging frames $(1, m), (2, m), (3, m), (4, m)$. The rendering results are illustrated as frames $(5,1)$ to $(5,5)$.

Obviously the total number of composite frames is computed based on the key and inbetween frames:

$$N_F = N_{KF} + (N_{KF} - 1) \times N_{IB} \quad 6.2$$

where N_F , N_{KF} and N_{IB} are total numbers of composite frames, key frames, and inbetween frames between each pair of consecutive key frames respectively.

According to the pseudo code in Table 6.1, the time complexity of the inbetweening algorithm is computed as:

$$C_{interpolation} = O(N_S \times (N_{KF} - 1) \times N_{IB}) \quad 6.3$$

where N_S is the number of strokes in each composite frame

Equation 6.3 can be further deduced as:

$$\begin{aligned} C_{interpolate} &= O\left(N_S \times (N_{KF} - 1) \times \frac{N_F - N_{KF}}{N_{KF} - 1}\right) = O(N_S \times (N_F - N_{KF})) \\ &= O(N_S \times N_F) \end{aligned} \quad 6.4$$

The merging algorithm processes each composite frame as an individual entity. The processing time of each composite frame is determined by the output dimension and number of composite frames and their resolution. Its time complexity can be computed as follows:

$$C_{merge} = O(N_L \times N_F \times W \times H) \quad 6.5$$

where N_L , N_F are the numbers of layers and composite frames, W and H are the width and height of the frames in pixel, respectively.

According to the above analysis, the overall time complexity of the rendering algorithm is:

$$C_{non-distrib} = C_{interpolate} + C_{merge} = O(N_S \times N_F + N_L \times N_F \times W \times H) \quad 6.6$$

6.3 Distributed Rendering Algorithm

The distributed rendering algorithm, introduced in the previous section, distribute different tasks into distributor and nodes, as shown in Table 6.3.

Render (...)	
Distributor distributes corresponded stroke sets to available nodes evenly On each node , For each set of corresponded strokes CS_i assigned to this node For each pair of successive strokes in CS_i Generate given number of inbetween strokes	Inbetweening
For each composite frame F_j in the animation Distributor picks a cluster node to merge F_j On each node , Create a buffer image buf with specified output dimension For each frame f in F_j Remove occluded strokes or stroke segments Merge f with buf (performance is proportional to output dimension)	Merging

Table 6.3: Distributed Rendering Algorithm

The time complexity of the distributed rendering algorithm is determined by the computation and communication time. Given the large amount of images typically

Chapter 6. Distributed Interpolation and Rendering

sent, network setup time and latency are relatively quite small. So the main communication time is spent on transferring frame between client workstation and cluster. Accordingly, the overall time complexity for the distributed rendering algorithm is:

$$C_{distribute} = C_{compute} + C_{transfer} \quad 6.7$$

Similar to the non-distributed, the overall computational complexity is composed of interpolation complexity and merging complexity.

$$C_{compute} = C_{distrib_interpolate} + C_{distrib_merge} \quad 6.8$$

Assume N_N denotes number of nodes in the cluster, each node will be assigned with $\frac{N_S}{N_N}$ sets of corresponding strokes. Therefore, the time complexity for interpolation is computed as:

$$C_{distrib_interpolate} = O\left(\frac{N_S \times N_F}{N_N}\right) \quad 6.9$$

Similarly, the time complexity for merging the entire sequence is:

$$C_{distrib_merge} = O\left(\frac{N_F \times N_L \times W \times H}{N_N}\right) \quad 6.10$$

The overall time complexity of the distributed rendering algorithm is hence calculated as:

$$\begin{aligned} C_{compute} &= C_{distrib_interpolate} + C_{distrib_merge} \\ &= O\left(\frac{N_S \times N_F + N_F \times N_L \times W \times H}{N_N}\right) \end{aligned} \quad 6.11$$

Time spent on sending vectorized strokes is determined by the numbers of key frames (N_{KF}) and strokes in each composite frames (N_S) in the sequence.

$$C_{send} = O(N_{KF} \times N_S) \quad 6.12$$

The frames sent back are raster images, so the receiving time is determined by the total number of frames and the size of each frame which is related to the output resolution.

$$C_{receive} = O(N_F \times W \times H) \quad 6.13$$

Based on Equation 6.12 and 6.13, the time complexity for transferring is:

$$\begin{aligned} C_{transfer} &= C_{send} + C_{receive} \\ &= O_{send}(N_{KF} \times N_S) + O_{receive}(N_F \times W \times H) \end{aligned} \quad 6.14$$

The overall time complexity for the distributed rendering algorithm is thus:

$$\begin{aligned} C_{distribute} &= O_{compute} \left(\frac{N_S \times N_F + N_F \times N_L \times W \times H}{N_N} \right) + O_{send}(N_F \times N_S) \\ &\quad + O_{receive}(N_F \times W \times H) \end{aligned} \quad 6.15$$

6.4 Experimental Results

The experiment is carried out with an internal cluster of 7 nodes. Each node is equipped with Intel Xeon 3.2 Processor and 3G of RAM. A distributor server, with 2 Intel Core Duo 3.0G Processors and 2G of RAM, is also setup, by installing an ad hoc distributing service for interactive rendering on it. The client workstation is configured with Intel Core Duo 3.2G processor and 2G of RAM.

6.4.1 Sample Output

Distributed rendering service is tested with four examples: Tukasa, Sayuri, Tukasa 2 and Kusaga. The following are the sample outputs, where key frames are marked with a dark surrounding box.

6.4.1.1 Tukasa Rendering Case

Chapter 6. Distributed Interpolation and Rendering

In Figure 6.2, the first and last images are the rendering result of the two input key frames while the rest are generated from the two key frames and then merged.

Figure 6.2: Tukasa Rendering Output

(Courtesy of Animation International Co. Inc.)

Details of this animation sequence and rendering time are shown in the table below. It can be seen that, with the cluster the rendering time is reduced to 13.85% of time that will be taken in a single workstation.

Layers	Key Frames	Total Frames	Strokes	Output Dimension (Pixels)	Client Rendering Time (Seconds)	Distributed Rendering Time (Seconds)
11	2	12	348	2000×488	115.382	15.985

Table 6.4: Tukasa Rendering Details and Performance

6.4.1.2 Sayuri Rendering Case

The following is the rendering result of Sayuri case. There are 5 key frames and 10 inbetweens for each pair of key frames. Instead of showing all 45 frames, we sample some inbetween frames for illustration purposes. The frame, IB12_5 denotes the fifth inbetween generated for the key frames Key1 and Key2; IB23_3 denotes the third inbetween generated for the key frames Key1 and Key2; IB23_3 represents the third inbetween for the key frame Key 2 and Key3, etc.

Figure 6.3: Sayuri Rendering Output

(Courtesy of Animation International Co. Inc.)

Details of this sequence and rendering time are shown below.

Layers	Key Frames	Total Frames	Strokes	Output Dimension (Pixels)	Client Rendering Time (Seconds)	Distributed Rendering Time (Seconds)
12	5	45	381	1000×812	184.232	27.283

Table 6.5: Sayuri Rendering Details and Performance

Given total 45 output frames, distributed rendering takes less than half a minute, which is acceptable as an interactive function.

6.4.1.3 Tukasa 2 Rendering Case

The following are selected from the rendering result of Tukasa rendering case.

Figure 6.4: Tukasa 2 Rendering Output

(Courtesy of Animation International Co. Inc.)

Rendering details are shown below.

Layers	Key Frames	Total Frames	Strokes	Output Dimension (Pixels)	Client Rendering Time (Seconds)	Distributed Rendering Time (Seconds)
8	11	26	283	1404×1146	162.981	22.927

Table 6.6: Tukasa 2 Rendering Details and Performance

The distributed rendering algorithm cuts 85.93% of the rendering time required on a single workstation.

6.4.1.4 Kusaga Rendering Case

The following images are a part of the Kusaga rendering result. The inbetween frames are selectively displayed below in Figure 6.5.

Chapter 6. Distributed Interpolation and Rendering

Figure 6.5: Kusaga Rendering Output

(Courtesy of Animation International Co. Inc.)

The following table shows details of this rendering case.

Layers	Key Frames	Total Frames	Strokes	Output Dimension (Pixels)	Client Rendering Time (Seconds)	Distributed Rendering Time (Seconds)
9	7	13	245	3516×2144	243.322	33.134

Table 6.7: Kusaga Rendering Details and Performance

At extreme high resolution of 3516 pixels by 2144 pixels, the time is still as low as $33.134/243.322 = 13.62\%$.

6.5 Summary

It is easy to see from the numbers that interpolation and rendering performance is indeed improved. It will save more time to integrate the coloring process into the framework as well. Moreover, this distributed framework can also be extended to a much larger scale, in order to further boost rendering performance. This can be achieved by deploying a more scalable and reliable distributor service and then interfacing it to the existing National Grid Office (NGO) nodes.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This project accomplishes the following: user controlled inbetween frame interpolation, smart curve fitting, high quality animation rendering, automatic coloring and high performance distributed rendering. The features are successfully incorporated into our 2D animation system. They are well accepted by animators after over twelve months of intensive testing.

Parameterized non-linear interpolation enables animators to have more control over how inbetweens are generated. Feature point preservation and junction preservation make computer generated inbetweens closer to manual inbetweens. Automatic coloring of inbetween frames can save a large amount of coloring work. All those achievements make digital 2D animation production more efficient.

Accurate interpretation of user input and high quality and fast stroke rendering lowers the barrier of switching from traditional 2D animation to digital 2D animation. Distributed interpolation and rendering make the system even more interactive and responsive, with much shorter waiting time and therefore higher efficiency.

Overall, the novel approaches proposed in this thesis have well contributed to the existing cartoon animation system, in various aspects including stroke input and representation, inbetweening, painting and rendering.

7.2 Future Work

Till now, strokes are required to be one-to-one matched before interpolation is done. This is a tedious task hence prone to mistakes. Further research can be done to improve the automatic matching of strokes between two pools (each corresponds to a frame) of strokes that are freely drawn. Artists can focus on the content of animation itself rather than spending time on taking care of stroke matching. Automatic matching may utilize stroke properties like length, thickness and curvature, etc. It can also make use of relations between strokes, like positions, closeness, intersections and movements.

Region matching algorithm needs further improvement, since the current algorithm has limitations when the topology of strokes changes from one key frame to another. The accuracy of matching should be improved as well.

Appendix

Software Design

Significant amount time and effort have been put into implementing the 2D animation system, which involve research work as well as design and implementation considerations. A rich set of fundamental data structures were defined to facilitate higher level manipulations.

Data Structures

The data structures in this project were designed in an object oriented manner. All elements of animation are treated as objects. For example, an animation is an object containing set of layer objects, frame objects, stroke objects, etc. The following figure shows the how an animation is structured.

Animation Structure

Appendix

An animation contains layers; a layer contains frames and a frame contains strokes or groups of strokes. If the animation is taken as a matrix of frames, a layer corresponds to a row in the matrix and a column of the matrix represents an ordered set of frames. This ordered set of frames is combined logically and named as a composite frame. A frame contains a part of a scene and a composite frame contains the entire shot. Animation is a running sequence of the entire shot.

Classes can contain others, as discussed above. They can also inherit from other classes, as show in the following inheritance tree.

Stroke Inheritance Tree

A *Stroke* inherits *Element*, which in turn inherits *CFCObject*. A stroke is inherited by various other classes. An interesting point here is that a *Primitive* (which represents a primitive shape) inherits *Stroke* too. The reason is that this inheritance

allows a primitive shape to be interpolated with a stroke seamlessly, just as the way two strokes interpolate.

Observers and Events

To facilitate higher level manipulation, for example invocation from user interface, the observer design pattern is applied.

The observer pattern (sometimes known as publish/subscribe) is a design pattern used in computer programming to observe the state of an object in a program. The essence of this pattern is that one or more objects (called observers or listeners) are registered (or register themselves) to observe an event which may be raised by the observed object (the subject). This definition is illustrated in the following diagram. The object which may generate an event generally maintains a collection of the observers.

Observer Pattern

With that pattern properly implemented, all the objects in the library are capable of capturing changes of their own internal states. If an object is associated with an observer, it will expose the changes to the observer. The observer can then react to those changes properly, to keep itself synchronized.

The internal state of an animation can be represented as the states of its own attributes and the states of all the layers contained inside. The state of a layer can be represented by its own attributes plus the states of all its frames and so on and so forth. Every class is only responsible for its own internal states. The state of a complex object, such as an animation, can be captured without much extra effort. This is a great benefit from the object oriented design, which also keeps every class simple and comprehensive, without limiting their modeling power and functions.

Undo-Redo Based on States and Actions

An application often has "undo" or "undo-redo" feature. However, it is not easy to implement them. Our approach is by utilizing a design pattern called *memento*, which is a common way of storing and restoring application states. A memento stores internal state of the *originator* object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.

We slightly narrow this definition to: "a memento is an object capable of restoring a target object to one of its previous states". It defines behavior of a memento instead of data as in the original definition. This is more useful in a practical sense, because it can be easily translated into implementations. An object diagram is shown in the following figure.

Memento Pattern

In the above diagram, it is shown that *Memento* defines only one method *Restore*. The purpose of this method is to restore it to a previous state and then return a *Memento* representing the state before restoring. 'Undo' can be achieved by invoking the *Restore* method and the returned *Memento* can be used to support 'redo'.

The *UndoRedoHistory* class will maintain an undo stack of mementos and a redo stack of mementos. Whenever the states of a target object change, a memento is pushed on to the undo stack. Undo action is as simple as this: pop the top memento from undo stack; call its *Restore* function; and then push the result on to the redo stack. Redo can be achieved similarly: pop from redo stack, call *Restore*, and push result on to undo stack.

Appendix

Another major class in the diagram is *CompoundMemento*. An undo action may consist of a sequence of atomic actions, or even hierarchically structured actions (atomic or complex). To support those complex undo actions to the maximum extent, a compound memento is introduced as a primary participant into this design pattern. A compound memento is a memento that contains more than one memento. Since it is a memento itself, it can be included by another compound memento; and this enables modeling of sequential and hierarchical memento structures.

The concrete implementations of *Memento* can be divergent. For example a *Memento* may store as little information as possible (as long as it is sufficient to uniquely identify a state) to reduce memory consumption. This will increase the difficulty of implement the *Restore* method. A memento may as well store a clone of the entire object. In this case the restoring process is as simple as placing the stored object back to where it was. The tradeoff is that memory consumption can be very high, since a clone of the entire object (which can be rather complex) need to be made for every change. In our application, both techniques are used in situations where suitable, such that runtime memory consumption and development complexity are balanced.

Reference

- [1] Alexa M, Cohen-Or D, Levin D. As-rigid-as-possible shape interpolation. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 157–164. ACM Press/Addison-Wesley Publishing Co., 2000.
- [2] Arrebola F, Bandera A, Camacho P, and Sandoval F. Corner detection by means of contour local vectors. *Electronics Letters*, (38(14)):699C701, 2002.
- [3] Bartel RH and Hardock RT. Curve-to-Curve Associations in Spline-Based Inbetweening and Sweeping. In *Proceedings of the 1989 ACM SIGGRAPH conference*, pages 167–174, 1989.
- [4] Catmull E. The problems of computer-assisted animation. In *Proceedings of ACM SIGGRAPH*, pages 564-569, 1978.
- [5] Coons S. Surfaces for computer-aided design of space forms. In M.I.T. MAC-TR-41, June 1967.
- [6] Di Fiore F, Schaeken P, Elens K, and Van Reeth F. Automatic inbetweening in computer assisted animation by exploiting 2.5D modeling techniques. In *Proceedings of Computer Animation*, pages 192-200, 2001.
- [7] Forrest A. On coons and other methods for the representation of curved surfaces. In *Computer Graphics and Image Processing*, volume 1, pages 341-369, 1972.
- [8] Fekete JD, Bizouarn E, Cournarie E, Galas T, and Taillefer F. TicTacToon: a paperless system for professional 2D animation. In *Computer Graphics Proceedings, Annual Conference Series*, pages 79-90, 1995.

Reference

- [9] Franco P. Preparata, Michael Ian Shamos, Computational Geometry: An Introduction. Convex Hull Algorithms in a Plane, Springer, pages 104-131, 1985.
- [10] Fritsch DS, Eberly D, Pizer SM, and McAuliffe MJ. Stimulated cores and their applications in medical imaging. In Proceedings of Information Processing in Medical Imaging, pages 365-368, 1995.
- [11] Hiransakolwong N, Vu K, Hua KA, and Lang SD. Shape recognition based on the medial axis approach. In IEEE International Conference on Multimedia and Expo, pages 257-260, 2004.
- [12] Johan H, Koiso Y, and Nishita T. Morphing using curves and shape interpolation techniques. In Proceedings of Pacific Graphics 2000, The Eighth Pacific Conference on Computer Graphics and Applications, pages 348-358, 2000.
- [13] Kochanek D H U and Bartel R H. Interpolation splines with local tension, continuity, and bias control. In Proceedings of ACM SIGGRAPH, pages 33-41, 1984.
- [14] Kort A. Computer Aided Inbetweening. In *ACM Press*, 2002.
- [15] Lane J M and Riesenfeld R. A theoretical development for the computer generation and display of piecewise polynomial surfaces. In IEEE Trans. RAMI, volume 2, pages 35-46, 1980.
- [16] Levoy M. A color animation system based on the multiplane technique. *ACM Computer Graphics*, 11:65-71, 1977.
- [17] Levoy M. Area flooding algorithms. New York, NY 10036, USA, July 1982. *ACM Press*.

Reference

- [18] Litwinowicz P and Williams L. Animating images with drawings. *Comput Graph*, pages 409-412, 1994.
- [19] Litwinowicz PC. Inkwell: a $2\frac{1}{2}$ D animation system. In *Computer Graphics / Proc. SIGGRAPH*, pages 113-122, 1991
- [20] Miura KT, Sato R, and Mori S. A method of extracting curvature features and its application to handwritten character recognition. In the *Fourth International Conference on Document Analysis and Recognition*, volume 2, pages 450-454, August 1997.
- [21] Orange CM and Groen FCA. Model based corner detection. In *IEEE Conference on Computer Vision and Pattern Recognition*, 1993.
- [22] Patterson JW and Willis PJ. Computer assisted animation: 2D or not 2D? *The Computer Journal*, 37(10):829-839, 1994.
- [23] Piegl L and Tiller W. *The NURBS book 2nd Edition*. Fundamental Geometric Algorithms, Springer-Verlag, pages 141-226, 1997.
- [24] Qun L and Jon G. Rokne: disk Bezier curves. In *Computer Aided Geometric Design*, 15(7), pages 721-737, 1998.
- [25] Reeves W T. Inbetweening for computer animation utilizing moving point constraints. In *Computer Graphics*, volume 15(3), pages 263-269, 1981.
- [26] Seah HS and Tian F. Computer-assisted coloring by matching line drawings. *Visual Computer*, 16(5):289-304, 2000.
- [27] Sederberg T W and Greenwood E. Shape Blending of 2-D Piecewise Curves. In *Mathematical methods for curves and surfaces*, Vanderbilt University Press, pages 497–506, 1999.
- [28] Stern and Garland. GAS - a system for computer aided keyframe animation. PhD dissertation, Univ. of Utah, 1978.

Reference

- [29] Teh C H and Chin R T. A scale-independent dominant point detection algorithm. In IEEE trans Computer Vision & Pattern Recognition, pages 229-234, 1988.
- [30] Thalmann N M and Thalmann D. Computer Animation: Theory and Practice. Springer-Verlag, 1985.
- [31] Van den Bergh J, Di Fiore F, Claes J, Van Reeth F. Interactively morphing irregularly shaped images employing subdivision techniques. In *1st Ibero-American Symposium on Computer Graphics 2002 (SIACG 2002)*. XX: XYQ2, 2002; 315–321.
- [32] Wu Z, Seah H S, Tian F, Xiao X, and Xie X. Simulating artistic brushstrokes using disk B-spline curves. In The Conference of Multimedia Arts Asia Pacific, 2004.
- [33] Wu Z K, Seah H S, Tian F, Xiao X, Xie B Y, "Artistic Brushstroke Representation and Animation with Disk B-spline Curve", *ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, ACE 2005, Jun 2005, Spain.
- [34] Xiao X, Seah H S, Wu Z, Tian F, and Xie X. Interactive free-hand drawing and in-between generation with disk B-spline curves. In The Conference of Multimedia Arts Asia Pacific, 2004.
- [35] Yu MP and LO KC. Object recognition by combining viewpoint invariant Fourier descriptor and convex hull. In Proceedings of Intelligent Multimedia, Video and Speech Processing 2001, pages 401-404, 2001.
- [36] <http://www.animationstand.com>
- [37] <http://www.axyzfx.com>
- [38] [http://www.bauhaussoftware.com/products mirage studio.php](http://www.bauhaussoftware.com/products/mirage_studio.php)

Reference

- [39] <http://www.cambridgeanimation.com/products/animo.htm>
- [40] <http://www.celaction.com>
- [41] <http://www.celaction.com/c2dworkflow.html>
- [42] http://www.digicelinc.com/FlipBook_Painter.htm
- [43] <http://www.adobe.com/products/flash>
- [44] <http://www.plasticanimationpaper.dk>
- [45] <http://www.render-it.co.uk>
- [46] <http://www.respower.com>
- [47] <http://www.retas.com>
- [48] <http://www.sun.com/software/Gridware/index.xml>
- [49] <http://www.toonboom.com/products>
- [50] <http://www.toonz.com>
- [51] <http://www.usanimation.com>