The College of Wooster

Open Works

Senior Independent Study Theses

2022

Stroke Clustering And Fitting In Vector Art

Khandokar Shakib
The College of Wooster, kshakib22@wooster.edu

Follow this and additional works at: https://openworks.wooster.edu/independentstudy

 Part of the Discrete Mathematics and Combinatorics Commons, Geometry and Topology Commons, Graphics and Human Computer Interfaces Commons, Numerical Analysis and Computation Commons, Ordinary Differential Equations and Applied Dynamics Commons, and the Other Mathematics Commons

Recommended Citation

Shakib, Khandokar, "Stroke Clustering And Fitting In Vector Art" (2022). *Senior Independent Study Theses.* Paper 9702.

This Senior Independent Study Thesis Exemplar is brought to you by Open Works, a service of The College of Wooster Libraries. It has been accepted for inclusion in Senior Independent Study Theses by an authorized administrator of Open Works. For more information, please contact openworks@wooster.edu.

© Copyright 2022 Khandokar Shakib

Stroke Clustering and Fitting in Vector Art

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Bachelor of Arts in Computer Science and Mathematics in the Mathematical and Computational Sciences at The College of Wooster at The College of Wooster

by
Khandokar Shakib
The College of Wooster
2022

Advised by:

Professor Kowshik Bhowmik

THE COLLEGE OF
WOOSTER

© 2022 by Khandokar Shakib

Abstract

Vectorization of art involves turning free-hand drawings into vector graphics that can be further scaled and manipulated. In this paper, we explore the concept of vectorization of line drawings and study multiple approaches that attempt to achieve this in the most accurate way possible. We utilize a software called *StrokeStrip* to discuss the different mathematics behind the parameterization and fitting involved in the drawings.

Acknowledgments

I would like to express my gratitude to the Computer Science and Mathematics department at the College of Wooster for the amazing education throughout my undergraduate years. I would specifically like to thank Dr. Nathan Sommer for his invaluable lessons and directions towards my goals as a computer scientist. Additionally, I would like to thank Professor Kowshik Bhowmik for his remarkable patience and support throughout my senior IS. Thanks to my parents, who believed in and trusted me no matter the circumstances. I would also like to thank my friends, whom I owe an incredible amount of gratitude. Thanks to Eraj Sikandar, Riya Joshi and Sai Kwan Khal for never giving up on me. Special thanks to my roommates Arnav Bhatnagar and Shivam Bhasin for being through thick, thin, and the invisible. Finally, I would like to thank my host mom Sharon Delgadillo for always loving me without any limits!

Contents

Abstract v

Acknowledgments vii

Contents ix

CHAPTER PAGE

1 Introduction 1

2 Raster and Vector graphics 3

3 Arc Length Parameterization 5
- 3.1 Parametric Equations . . . 5
  - 3.1.1 Curves . . . 5
  - 3.1.2 Arc Length of parametric curves . . . 6
- 3.2 Vector Valued Functions . . . 7
  - 3.2.1 Curvature of curves . . . 8
- 3.3 Arc length parameterization . . . 8

4 Curves in Computer graphics 11
- 4.1 Basic Idea . . . 11
- 4.2 Types of curves . . . 12
  - 4.2.1 Explicit . . . 12
  - 4.2.2 Implicit Curves . . . 13
  - 4.2.3 Parametric . . . 14
- 4.3 Modeling smooth curves . . . 15
  - 4.3.1 General Principles . . . 15
  - 4.3.2 Continuity . . . 17
  - 4.3.3 Parametric Cubic curves . . . 20
  - 4.3.4 Hermite Curves . . . 24
  - 4.3.5 Bézier curves . . . 28

5 Human Perception for Over-sketched Art 33

6 Clustering 37
- 6.1 Overview . . . 37
- 6.2 Method Details . . . 37
  - 6.2.1 Coarse Clustering . . . 38
  - 6.2.2 Fine Clustering . . . 40

6.2.3 Final fitting . . . 41
7 Fitting Curves 43
7.1 Orbay and Kara fitting . . . 44
7.2 StrokeStrip Parameterization and Fitting . . . 46
7.2.1 Problem Statement and Goals . . . 47
7.2.2 Formulation . . . 50
7.2.3 Solution and Algorithm . . . 52
7.2.3.1 Pairwise Gradient Orientation . . . 53
7.2.3.2 Core Parameterization . . . 54
7.2.3.3 Curve Fitting . . . 54
8 Methods 57
8.1 Input using Adobe Illustrator . . . 57
8.2 StrokeAggregator . . . 58
8.2.1 Input Format . . . 58
8.2.2 Svg to Scap Conversion . . . 59
8.3 Aggregator Labeller . . . 60
8.3.1 Application UI . . . 61
8.3.2 Labeller Usage . . . 64
8.4 StrokeStrip Fit . . . 67
8.4.1 Prerequisites . . . 67
8.4.2 Running and Generating Files . . . 69
9 Results and Discussion 73
9.1 Special Case Study . . . 79
10 Conclusion 85
References 87

CHAPTER 1

Introduction

Free-form drawings are still the most common method artists use to give shapes to their ideas and images. Line drawings are usually rough sketches intended to provide a framework for the main drawing. While drawing these outline sketches, artists focus less on details and more on producing the general shapes. This means the drawings are often over-sketched, where multiple strokes are used to produce an overall, *aggregate* stroke that represents the intended curves. These rough drawings are typically retraced and the groups of strokes are carefully drawn as aggregate strokes. For digital media, these artworks are also often digitally 'cleaned' before production. This step is especially important for digital art, illustrations, and 2D animations.

Typically, an artist would require a considerable amount of time to consolidate all the drawings into a quality drawing that can be used in animations. To make this process time-efficient and effective, multiple algorithms have been developed to automate this process. The main purpose of this paper is to study such an algorithm through a project named StrokeStrip. StrokeStrip is a new powerful method that fits intended curves into stroke clusters [13]. The consolidation carried out by such software involves multiple steps, each performing a crucial part of the task. We explore each of these steps in moderate detail to learn more about how drawings can be automatically cleaned (or consolidated). We first study mathematical curves

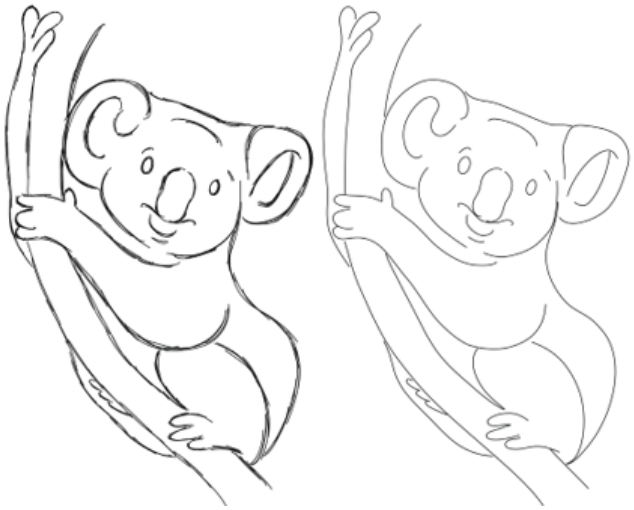


Figure 1.1: Raw sketch (left) and manual consolidation (right) [13]

and their application in approximating curves using points and their geometrical properties. To get a deeper understanding of the process as a whole, we also explore a few related projects. This includes the projects StrokeAggregator [7] and Beautification of Design Sketches by Orbay and Kara [9]. While we discuss individual steps involved in the StrokeStrip algorithm, we two similar steps that are involved in these papers to get a broader sense of these specific stages.

Before we explore the algorithmic implementation, we briefly discuss the two major types of digital graphics - Raster graphics and Vector graphics. We also explain the differences between these two formats and how they are related to the automatic generation of clean line art.

CHAPTER 2

Raster and Vector graphics

Traditionally, the most common format used for any sort of graphics is known as *Raster* graphics. Raster graphics work by storing information about the individual pixels of an image. It stores information such as how many total pixels are present in the image, their individual color, and their position. The number of pixels in the image determines its *resolution*. If we try to zoom in on (or practically stretch) an image beyond its resolution, computers have to make approximations for the extra pixels to be displayed. This 'approximation' of the larger image is often produced at the expense of quality. The result is a blurry image with poor details where individual pixel shapes can be spotted - a situation which is commonly referred to as 'pixelation'.

Compared to raster graphics, *Vector* graphics work in a fundamentally different way. Vectors graphics do not store information about pixels, rather it defines images with polygons, lines, shapes, and their position. Shapes like lines, boxes, and curves are used and their geometric position is saved in the vector file format. The positional data is relative, for which scaling is not an issue for vector images. For example, a 2 cm by 2 cm square in vector format will become a 4 cm by 4 cm square when zoomed to 200%. Information about the square is stored as mathematical coordinates, and hence can be freely scaled. This difference in image quality due to format difference is illustrated in Figure 2.1.

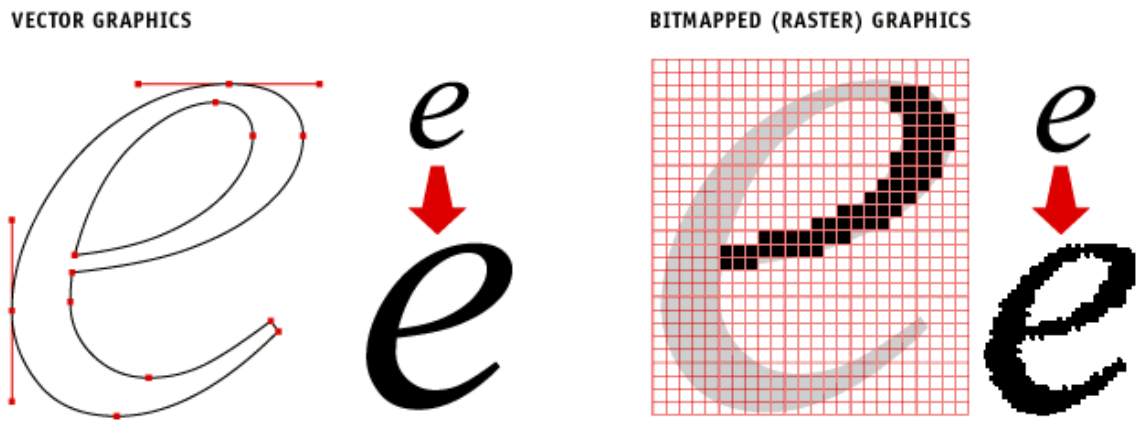


Figure 2.1: Raster vs Vector images [4]

Storing data about individual pixels requires significantly more memory than mathematically defined curves and shapes. Millions of pixels in raster graphics allow a very high level of granularity and details for the image. This makes raster graphics the preferred type for most graphics, especially photographs and art. In certain applications, however, vector graphics is irreplaceable. Images in vector format contain anchors and nodes within each part of the figure. This allows the artwork to be re-shaped or resized at each anchor point. For a shape in vector, we can even change thickness, length, and curvature for individual strokes. In applications such as logo drawings, posters, minimalist art, etc, this feature is leveraged along with its scalability. This makes vector graphics the perfect medium for creating schematic drawings. In 2D animation, the dynamic parts are usually drawn in a simplistic manner to ensure fluid and efficient animation. But the characters are re-drawn for multiple frames and scenes, making the whole process quite tedious. If a basic character design can be accurately vectorized, the drawing can then be simply edited for different scenes that use roughly the point of view.

CHAPTER 3

Arc Length Parameterization

In this chapter, we explore the concept of parameterizing arc length. This is a crucial concept for understanding the formation of strips from the required cluster of strokes. In chapter 7 we will discuss the use of, *arc length parameterization* and isolines for forming strips. Once the parameterization has been computed, we can produce the final fitted curve from it. The fitted curve reflects the overall shape of the cluster at individual parameter values [13].

3.1 Parametric Equations

3.1.1 Curves

A common way to describe functions is through Cartesian coordinates. But for a lot of curves, the functions for them do not show us the complete picture. We take the equation of circle to illustrate this. In Cartesian coordinates, the equation of a circle with center $(0, 0)$ and radius r can be written as

$$x^2 + y^2 = r^2$$

Now, if we want to investigate any individual variables, it will be very difficult

to do so using a single function. The x variable, for example, would require the two functions

$$x = \sqrt{r^2 - y^2}$$
$$x = -\sqrt{r^2 - y^2}$$

for describing two individual halves of the circle. This means without further information about the quadrants, we can only work with one half of the circle at a time. This is where *parametric equations* are particularly useful. We use a third parameter, which is an independent variable. The parameter is set in a way that x and y are functions of this third parameter. These functions are called parametric equations. Suppose our x and y values are given as functions of the parameter t by the following equations

$$x = f(t)$$
$$y = g(t).$$

As the value of the parameter t varies, the point $(x, y) = (f(t), g(t))$ varies and traces out a curve as shown in Figure 3.1 [12]. This curve is known as a **parametric curve** or a *plane curve*, commonly denoted by C.

3.1.2 Arc Length of parametric curves

For a curve C in Cartesian coordinates, we use the arc length formula to show its length L. If the curve is in form $y = F(x)$, where $a \leq x \leq b$ for continuous F', we get

$$L = \int_a^b \sqrt{1 + \left(\frac{dy}{dx}\right)^2}\, dx \tag{3.1}$$

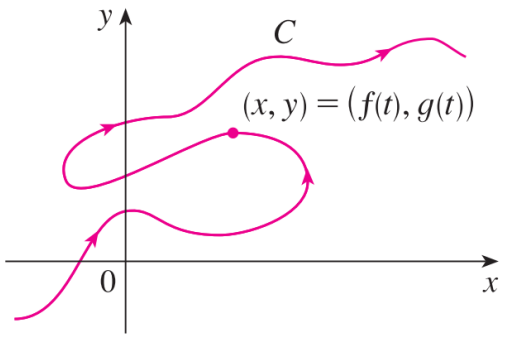


Figure 3.1: Parametric curve [12]

Let us set the boundary on our parameter t such that $\alpha \leq t \leq \beta$. For our parametric equations $x = f(t)$ and $y = g(t)$, we consider $dx/dt = f'(t) > 0$. This means we traverse the curve C once as t increases from α to β and $f(\alpha) = a, f(\beta) = b$. Using substitution on equation 3.1 we get

$$L = \int_a^b \sqrt{1 + \left(\frac{dy}{dx}\right)^2}\, dx = \int_\alpha^\beta \sqrt{1 + \left(\frac{dy/dt}{dx/dt}\right)^2} \frac{dx}{dt} dt.$$

Considering $dx/dt = f'(t) > 0$ we derive the formula for arc length of C as

$$L = \int_\alpha^\beta \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\, dt. \tag{3.2}$$

3.2 Vector Valued Functions

Before we discuss arc-length parameterization, we revisit the concept of *vector valued functions*. A **vector valued function** in two dimensions is a function of the form

$$\mathbf{r}(t) = f(t)\mathbf{i} + g(t)\mathbf{j}. \tag{3.3}$$

$$\text{Alternatively, } \mathbf{r}(t) = \langle f(t), g(t) \rangle. \tag{3.4}$$

The functions f and g are real valued functions of parameter t known as

component functions [8]. Restrictions on vector-valued functions can be placed through either the parameter t or the component functions themselves. For two dimensional curves with parametric equations $f(t)$ and $g(t)$, the arc length L can be calculated for the interval $a \leq t \leq b$ using the following formula :

$$L = \int_a^b \sqrt{f'(t)^2 + g'(t)^2}\, dx. \tag{3.5}$$

3.2.1 Curvature of curves

At a specific point on a curve, curvature can be understood as a measure of how quickly the curve changes direction at the point. Mathematically, curvature is the rate of change of the unit tangent vector with respect to arc length [12].

The unit tangent vector for parameter value t denoted by $T(t)$ is

$$\mathbf{T}(t) = \frac{\mathbf{r}'(t)}{|\mathbf{r}'(t)|} \tag{3.6}$$

The curvature in terms of the tangent vector can hence be written as:

$$\kappa = \left| \frac{d\mathbf{T}}{ds} \right| \tag{3.7}$$

This makes the final equation:

$$\kappa(t) = \frac{|\mathbf{T}'(t)|}{|\mathbf{r}'(t)|} \tag{3.8}$$

3.3 Arc length parameterization

A vector function $r(t)$ gives the exact position of a point through the parameter t. This parameter is often considered to be *time*, which might not always be the case. Suppose from some fixed point, we denote the distance along the curve

$r(t)$ as s. If s is used as a variable, we get $r(s)$ which describes the position in the space curve in terms of the distance along the curve from the starting point. The curve still represents the position of the object, but not at any specific value of the parameter. It gives the position as a function of the distance traveled by the object. The computation to determine arc length through the parameter t often poses to be quite difficult. But when the curve is arc-length parameterized, the computation becomes **simpler** as the arc-length (distance drawn) is in a one-to-one ratio with the parameter (the variable). For example, if r is arc length parameterized, the magnitude of $r(s)$ for $a \leq s \leq b$ is just $|b - a|$.

For a smooth curve $r(t)$ defined for $t \geq a$, the arc length function is given by

$$s(t) = \int_a^t \|\mathbf{r}'(u)\| \, du \tag{3.9}$$

where the integral is from the starting point a to point t. Using this arc length function, we can find a different parameterization of the curve which is known as **arc-length parameterization**. Given that our original function is vector-valued, we can reparameterize the integral in Equation 3.9 by changing the variables. By using the arc length function $s(t)$, we first need to solve for t as a function of $s(t)$. Using this expression for t, we can substitute it inside the original function $r(t)$ for the curve. This changes the parameter of the vector-values function to s, which is the arc-length of the curve.

To further understand the arc-length parameterization, we use a simple example using a circle of radius 2. That would give us the parameterization

$$r(t) = \langle 2 \sin t, 2 \cos t \rangle. \tag{3.10}$$

Using Equation 3.9 starting from parameter value $a = 0$ and using u as the variable of integration, we get the following arc-length function:

$$\begin{aligned} s(t) &= \int_a^t \|\mathbf{r}'(u)\|\, du \\ &= \int_0^t \|\langle 2\sin u, 2\cos u\rangle\| du \\ &= \int_0^t \sqrt{(2\sin u)^2 + (2\cos u)^2} du \\ &= \int_0^t \sqrt{4\sin^2 u + 4\cos^2 u} du \\ &= \int_0^t 2du = 2t, \end{aligned} \tag{3.11}$$

This gives us the relationship between arc length s and parameter t as $t = s/2$ [8]. This allows us to express $r(t)$ as a function of arc length s as follows:

$$r(s) = 2\sin\left(\frac{s}{2}\right)\mathbf{i} + 2\cos\left(\frac{s}{2}\right)\mathbf{j}. \tag{3.12}$$

Here, Equation 3.12 represents the arc-length parameterization of the curve $r(t)$ with the new boundary $s \geq 0$.

CHAPTER 4

Curves in Computer graphics

4.1 Basic Idea

Curves can be intuitively imagined as drawing with a pen on paper using lines. Using simple lines, we are unable to produce filled regions. But it allows us to give the outline of the shape or object being drawn. The concept of curves is crucial in understanding how graphics allows us to create remarkable shapes into flat objects. Although objects we observe are usually not flat, numerous objects can be drawn by making use of multiple curves. This is possible as modern graphics systems can render flat, 3D polygons at very impressive rates. This includes intermediate processes such as performing hidden-surface removal, shading, and texture mapping [3].

A curve can be imagined as a set containing infinite points. The beginning and the end of the curve are known as *endpoints*. Every point besides these endpoints should have two neighboring points. A simple line is also considered a curve, even though it might not look like it is 'curved'. Curves can also be *closed*, which means the curve meets itself and forms a connection in a loop. Mathematically, A curve can be defined as a continuous map from a one-dimensional space to a n-dimensional space.

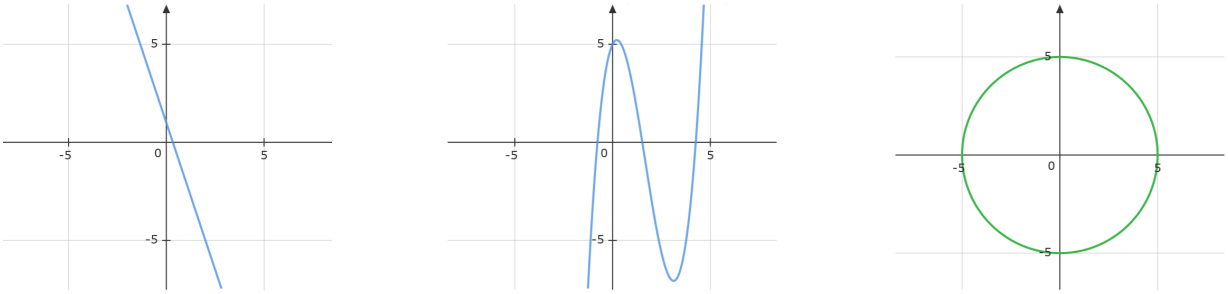


Figure 4.1: (From left to right) A line, cubic curve, and a circle

4.2 Types of curves

Due to the broad definition of a curve, it is not possible to give a name to every curve. Curves that are used commonly have their own names, such as lines, circles, ellipses, parabolas, etc. A curve can also be *free-form*, which is a curve that can take practically any shape but does not have a specific name for it. Using the types of object representation in mathematical form, curves can be classified into three categories - **explicit**, **implicit**, and **parametric** [3]. In the following portion, we discuss these three ways of modeling curves and surfaces.

4.2.1 Explicit

The explicit forms of a curve in two dimensions take the value of a single variable and produce the value of another variable. The input variable is known as the **independent variable** and the variable whose value is produced is known as the **dependent variable**.

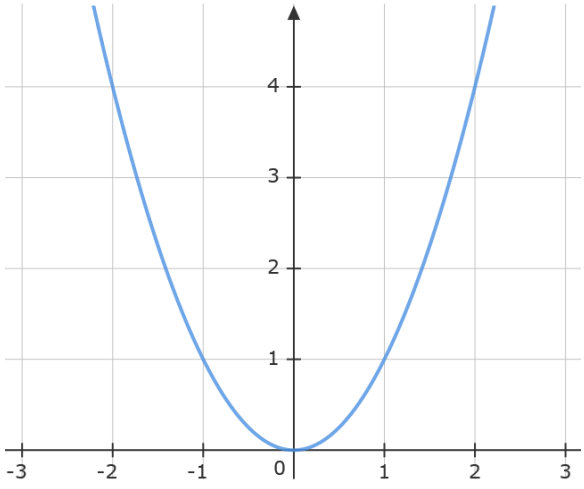


Figure 4.2: Graph of a parabola from explicit function $y = x^2$.

In the Cartesian xy-plane, we can write this as $y = f(x)$ where x and y are independent and dependent variables respectively. We can even try to find the inverse function to express x as a function of y as $x = g(y)$. But generally, there is no guarantee for explicit curves to have either of the forms. This means that a curve in the plane might not be the complete representation of the function. This is an inherent issue with explicit representation. This mainly occurs as the representation is coordinate-system dependent [3].

4.2.2 Implicit Curves

These curves are represented by a set of points on the curve using an implicit function that checks if a point lies on this curve. In other words, it divides the Cartesian space into points that are part of the curve, and those which do not.

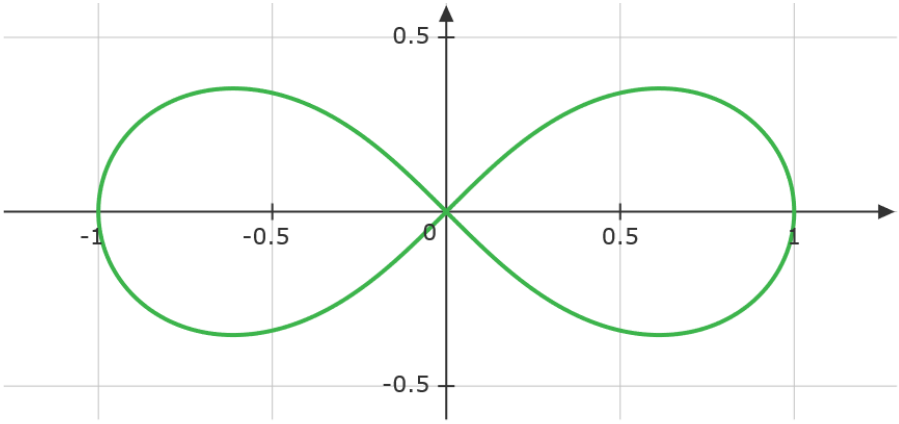


Figure 4.3: The lemniscate of Bernoulli given by the implicit form $(x^2 + y^2)^2 - (x^2 - y^2)$. [5]

The implicit form of curves is less coordinate-system dependent than the explicit form. It can be used to represent all lines and circles [3]. For example, the function for circle in implicit form is :

$$f(x, y) = 0 \tag{4.1}$$

However, this representation does not give a straightforward way to calculate a y value on the curve from a given x value. Implicit representations often arise in practical applications of curves and surfaces. But due to the difficulty of attaining the points on the curve, the use of implicit representations is limited.

4.2.3 Parametric

These curves are mappings from a free parameter to the set of points on the curve. An example would be drawing on paper with a pencil, where the free parameter is time. The time ranges from when we begin drawing to the time when we finish.

The parametric equation for the curve gives us the position at any instance in the time range.

$$x, y = f(t) \tag{4.2}$$

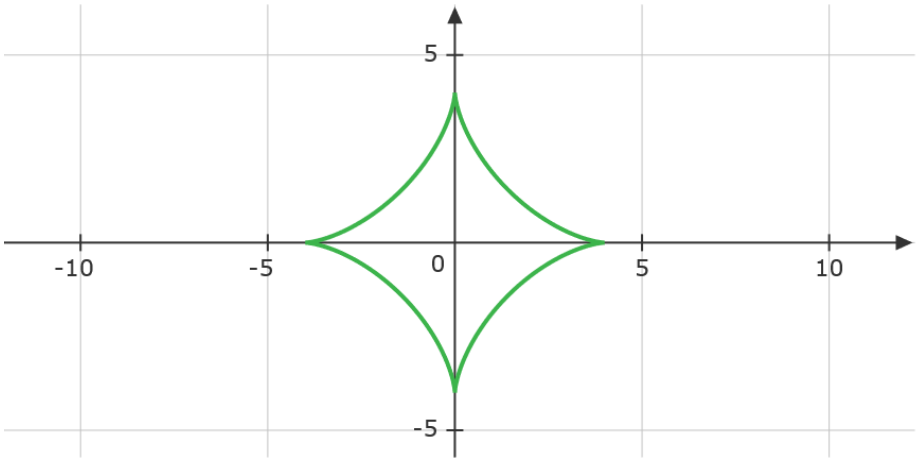


Figure 4.4: The curve given by parametric equations $x(t) = 3cos(t) + cos(3t)$, $y(t) = 3sin(t) - sin(3t)$.

A parameterized curve in $\mathbb{R}^n$ is a map $\gamma : (\alpha, \beta), \rightarrow \mathbb{R}^n$, for some α, β with $-\infty \leq \alpha < \beta \leq \infty$. We have discussed parameterized curve in more detail in Section 3.

4.3 Modeling smooth curves

4.3.1 General Principles

For the paper, one of our major goals is to study how we can approximate *smooth* curves from a given point set.

An obvious method for controlling curves would be by specifying points to be interpolated. This would form an approximation to the curve by forming lines

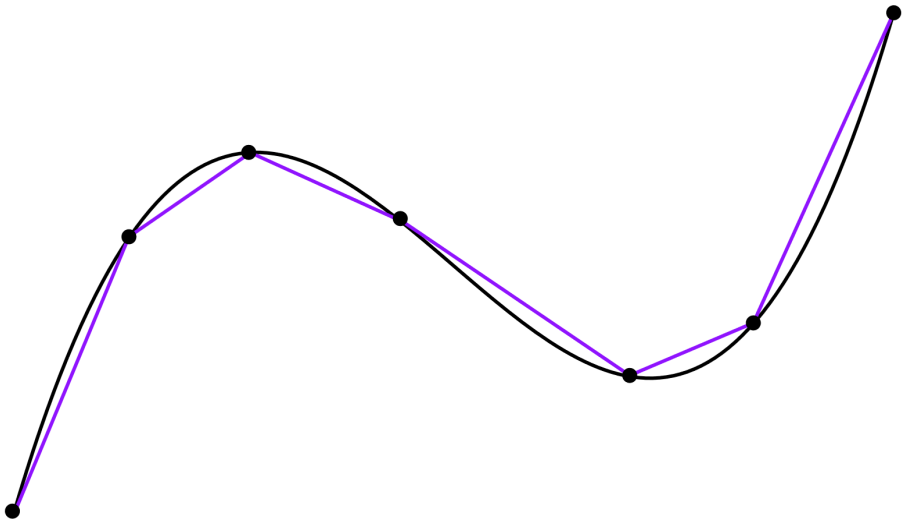


Figure 4.5: Polyline approximation of a curve. The approximation is shown with a blue line.

between adjacent points, as shown in Figure 4.5. This method is known as *polyline* conversion, and it is especially tempting to use as the computational approximations are easy to work with. Although this method is straightforward, this gives rise to a few undesirable properties. This method has less continuity with very less control of the actions between the points. If we need to move the curve in any direction, a polyline approximation gives us a static representation of the curve. To shift the curve in any direction, all the points are required to be shifted as well. Additionally, the number of points plays a major role in the accuracy of the approximation.

The two main demands that we expect from a curve approximating method are **expressiveness** and **simplicity** [11]. The representation should be able to approximate most shapes (within its degree of freedom). While a combination of multiple approximations might be useful in some cases, it is better to have a method that extends its application over most of the general uses. It is practically impossible to work in the space containing all possible curves, as that space would contain infinite curves. At the same time, we do not want a method that is overly complicated to

implement.

Compared to methods that involve specifying exact points, approximation methods using **control points** are preferred. Control or data points influence the overall shape of the curve, allowing us more local control and a better idea about the curve behavior.

4.3.2 Continuity

When two separate parametric pieces come together to form an aggregate curve, it is crucial to understand the local properties at this point. The functions must have a single value corresponding to the given parameter values. Having multiple values would indicate that the curve is 'broken', which means the curve cannot be drawn in a single, continuous pen stroke.

To better understand this local continuity, we introduce the notion of *parametric continuity* and *geometric continuity*. To express n^{th} order parametric continuity, we use the symbol C^n. Similarly, we use G^n to represent n^{th} order geometric continuity. Two curves are C^1 continuous if their tangent vectors are equal in terms of both magnitude and direction. When the tangent vectors have the same direction but different magnitudes, then the two curves have G^1 continuity [6].

Consider Figure 4.6, where the polynomial on the left is $g(t)$ and the other polynomial is $h(t)$. We consider the polynomials and their derivatives at parameter value $t = 1$ for $g(t)$. At the same time, we take the corresponding values of $h(t)$ at parameter value $t = 0$. For the function to be continuous, we need to enforce the following conditions:

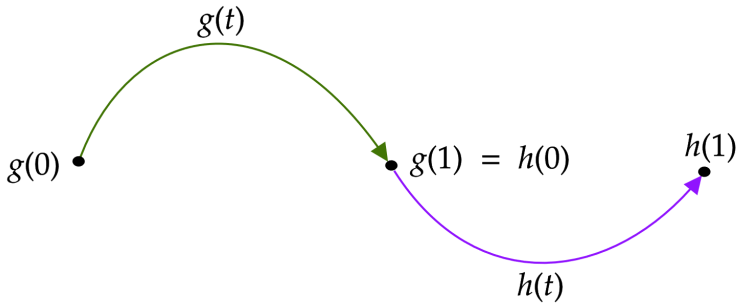


Figure 4.6: Continuity

$$\mathbf{g}(1) = \begin{bmatrix} g_x(1) \\ g_y(1) \\ g_z(1) \end{bmatrix} = \mathbf{h}(0) = \begin{bmatrix} h_x(0) \\ h_y(0) \\ h_z(0) \end{bmatrix}. \tag{4.3}$$

When all the individual parametric components are equal at the joining point, we can say that the curve has C^0 parametric continuity [3]. Taking the same idea, we can also consider the following conditions for the first derivatives:

$$\mathbf{g}(1) = \begin{bmatrix} g'_x(1) \\ g'_y(1) \\ g'_z(1) \end{bmatrix} = \mathbf{h}(0) = \begin{bmatrix} h'_x(0) \\ h'_y(0) \\ h'_z(0) \end{bmatrix}. \tag{4.4}$$

When all of the parametric and derivative conditions in Equation 4.3 and Equation 4.4 are satisfied, the curve has C^1 parametric continuity.

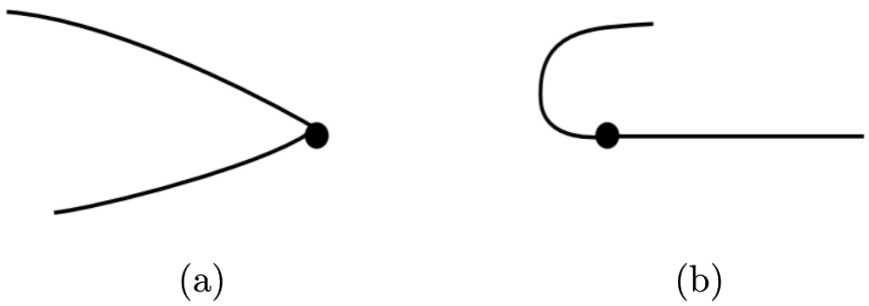


Figure 4.7: (a) C^0 Continuity. (b) C^1 Continuity [11].

In terms of geometry of the curve, we address the continuity in a different manner. Similar to the conditions in equation 4.4, we also consider the derivative for geometric continuity. But instead of enforcing the derivatives of two segments at the joint point to be equal, we impose the rule that they should be proportional [3]. This would mean

$$g'(1) = \kappa h'(0),$$

for some positive number κ. In simple terms, this condition implies that the two segments may have different magnitudes for tangent value. But as the tangents are proportional, their direction remains to be the same. This type of continuity is known as G^1 geometric continuity, as shown in Figure 4.8 [3].

From the above conditions, we notice that geometric continuity is less restrictive than parametric continuity as it does not require the magnitudes of the tangents to be equal. This means a C^n parametric continuous curve necessarily has G^n geometric continuity (if the derivatives are non-zero), but the converse is not always

Figure 4.8: (a) G^0 Continuity (sharp turn). (b) G^1 Continuity (smooth connection) [11].

true. Additionally, C^0 and G^0 continuity are analogous, and it means that the two segments share a common endpoint [6].

4.3.3 Parametric Cubic curves

In computer graphics, *piecewise parametric functions* are the most commonly used representation. Using this approach, the curve being approximated is broken down into multiple pieces where each piece is a polynomial of the parameter.

When choosing parametric polynomial curves for approximation, the degree of the curve must be chosen as well. Choosing a higher degree polynomial would give us more parameters that can be modified to give us the required shape. But the computation involving points on the curve will be demanding. Polynomials with high orders also tend to contain more bends, which might not be the optimum pick to model smooth curves. In contrast, a polynomial of a low order may not offer the required number of parameters needed to represent the desired shape. To choose a degree that balances out the pros and cons mentioned above, we can design each curve segment over a short interval and choose a polynomial of lower order [3]. Choosing a small region of the curve allows us to produce the required

shape even with fewer parameters. This tradeoff between flexibility and simplicity is best provided by parametric *cubic* curves. The fundamental form of a parametric cubic curve is as follows

$$F(t) = \mathbf{a} + \mathbf{b}t + \mathbf{c}t^2 + \mathbf{d}t^3 \tag{4.5}$$

where **a**, **b**, **c**, **d** are constant vectors [6]. Every value of $\mathbf{F}(t)$ is a point on the curve corresponding to the parameter value of t. In three dimensions, we can write the components of $F(t)$ separately in x, y, and z directions. These individual parametric equations are

$$\begin{aligned} F_x(t) &= a_x + b_x t + c_x t^2 + d_x t^3 \\ F_y(t) &= a_y + b_y t + c_y t^2 + d_y t^3 \\ F_z(t) &= a_z + b_z t + c_z t^2 + d_z t^3. \end{aligned} \tag{4.6}$$

In matrix form, we can write the previous equations as a matrix product of coefficients with the parameters.

$$\mathbf{F}(t) = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix} \tag{4.7}$$

We can condense the product to write

$$F(t) = CT(t), \tag{4.8}$$

where C is the matrix containing the coefficients, and $T(t) = \langle 1, t, t^2, t^3 \rangle$ [6]. To find the tangent direction of the curve at t, we need to calculate the derivative of

$F(t)$. The matrix C here contains constant values, which gives us a straightforward way to write the derivative of $F(t)$ as

$$\mathbf{F}'(t) = \mathbf{C}\frac{d}{dt}\mathbf{T}(t) = \mathbf{C}\begin{bmatrix} 0 \\ 1 \\ 2t \\ 3t^2 \end{bmatrix}. \tag{4.9}$$

While attempting to utilize cubic curves to approximate curves, we need to factor in the geometrical constraints given to us. For the purpose of this paper, we focus on the constraints on endpoint values (values of function $F(t)$ at $t = 0$ and $t = 1$) or tangent directions at endpoints ($F'(t)$ values at $t = 0$ and $t = 1$). We know that any arbitrary cubic function contains four coefficients, which means we require four constraints to define the specific curve. Suppose we denote these constraints as $g_1, g_2, g_3,$ and g_4. We can then use these constraints to formulate a weighted sum, and incorporate the sum to get the following expression for $F(t)$ [6]

$$\begin{aligned} \mathbf{F}(t) = &\left(a_1 + b_1 t + c_1 t^2 + d_1 t^3\right)\mathbf{g}_1 \\ &+ \left(a_2 + b_2 t + c_2 t^2 + d_2 t^3\right)\mathbf{g}_2 \\ &+ \left(a_3 + b_3 t + c_3 t^2 + d_3 t^3\right)\mathbf{g}_3 \\ &+ \left(a_4 + b_4 t + c_4 t^2 + d_4 t^3\right)\mathbf{g}_4. \end{aligned} \tag{4.10}$$

Here we observe that there is a polynomial function $a_i + b_i t + c_i t^2 + d_i t^3$ for each of the four constraints. These polynomials are known as **blending functions**, which allow the behavior of the curve to have a reasonable abstraction. Blending functions specify how much the parameter vector values 'blend' together by representing the curve as a weighted linear combination of its control points. Using matrix notation, we re-write equation 4.10 as

$$\mathbf{F}(t) = \begin{bmatrix} \mathbf{g}_1 & \mathbf{g}_2 & \mathbf{g}_3 & \mathbf{g}_4 \end{bmatrix} \begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & d_4 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}. \tag{4.11}$$

This can be condensed down to just

$$F(t) = GMT(t), \tag{4.12}$$

where the matrix G is

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_1 & \mathbf{g}_2 & \mathbf{g}_3 & \mathbf{g}_4 \end{bmatrix} = \begin{bmatrix} (\mathbf{g}_1)_x & (\mathbf{g}_2)_x & (\mathbf{g}_3)_x & (\mathbf{g}_4)_x \\ (\mathbf{g}_1)_y & (\mathbf{g}_2)_y & (\mathbf{g}_3)_y & (\mathbf{g}_4)_y \\ (\mathbf{g}_1)_z & (\mathbf{g}_2)_z & (\mathbf{g}_3)_z & (\mathbf{g}_4)_z \end{bmatrix}. \tag{4.13}$$

This matrix G is called the **geometry matrix**. The 4×4 matrix M is the **basis matrix** and is defined as

$$M = \begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & d_4 \end{bmatrix}. \tag{4.14}$$

These two matrices are very important in investigating the curves being approximated. For example, each class of cubic curves will have a constant basis matrix M. The formation of the shapes of specific curves in each of these classes are determined exclusively by the geometry matrix G [6].

4.3.4 Hermite Curves

The interpolation method or curves using cubic curves in the preceding chapter are completely dependent on the control points given. More precisely, it hinges on using the points to construct parametric segments that pass through them. A major downside of using a point-based approach is the lack of control over the curves.

Given a set of points to approximate a curve, we cannot move any points as the curve needs to pass through all of them. The only way to modify the outlook of the curve formed from such approximation is by adding more points to create a better interpolation. From a designer's perspective, it would be quite convenient to include an aspect to these approximations such that the curve can be adjusted interactively. This allows the designer to modify the shape of the curve more intuitively without the computational overhead of adding more points. The *Hermite interpolation* is a method designed to deal with such issues.

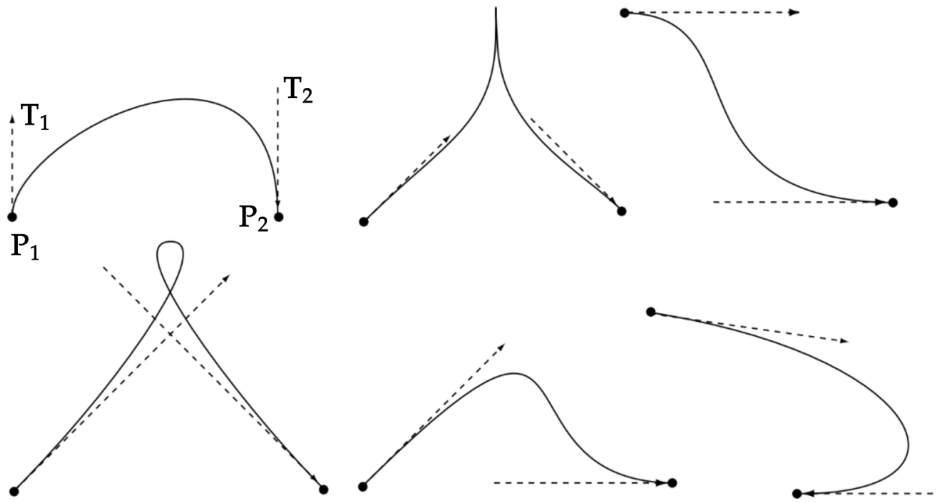


Figure 4.9: Different Hermite curve segments [11].

A cubic Hermite curve can be defined using two endpoints P_1 and P_2 and the tangent vectors T_1 and T_2 at these endpoints [6]. Here, we assume that the derivatives of the polynomial exist at the endpoints to calculate the tangent values.

All these four quantities are used to obtain the geometry matrix, and a Hermite curve $H(t)$ can be expressed as

$$\mathbf{H}(t) = \begin{bmatrix} \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{T}_1 & \mathbf{T}_2 \end{bmatrix} \mathbf{M}_H \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}, \tag{4.15}$$

where we need to determine the 4×4 basis matrix M_H [6]. The known four quantities $P_1, P_2, T_1,$ and T_2 can be used to find the imposed geometrical constraints. For parameter values $t = 0$ and $t = 1$, we first calculate the parameter matrix $\langle 1, t, t^2, t^3 \rangle$ in Equation 4.15. We also calculate the derivative for the parameter matrix $\langle 0, 1, 2t, 3t^2 \rangle$ at $t = 0$ and $t = 1$. This leads us to the following four equations:

$$\begin{aligned} \mathbf{H}(0) &= \begin{bmatrix} \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{T}_1 & \mathbf{T}_2 \end{bmatrix} \mathbf{M}_H \langle 1, 0, 0, 0 \rangle = \mathbf{P}_1 \\ \mathbf{H}(1) &= \begin{bmatrix} \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{T}_1 & \mathbf{T}_2 \end{bmatrix} \mathbf{M}_H \langle 1, 1, 1, 1 \rangle = \mathbf{P}_2 \\ \mathbf{H}'(0) &= \begin{bmatrix} \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{T}_1 & \mathbf{T}_2 \end{bmatrix} \mathbf{M}_H \langle 0, 1, 0, 0 \rangle = \mathbf{T}_1 \\ \mathbf{H}'(1) &= \begin{bmatrix} \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{T}_1 & \mathbf{T}_2 \end{bmatrix} \mathbf{M}_H \langle 0, 1, 2, 3 \rangle = \mathbf{T}_2. \end{aligned} \tag{4.16}$$

We can write this as a single equation in matrix notation as

$$\begin{bmatrix} \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{T}_1 & \mathbf{T}_2 \end{bmatrix} \mathbf{M}_H \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 0 & 3 \end{bmatrix} = \begin{bmatrix} \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{T}_1 & \mathbf{T}_2 \end{bmatrix} \tag{4.17}$$

Using Equation 4.17, we observe that simple matrix operations allow us to calculate the basis matrix M_H by

$$\mathbf{M}_H = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 0 & 3 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & -3 & 2 \\ 0 & 0 & 3 & -2 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \tag{4.18}$$

The values of the matrix M_H in Equation 4.18 give us the coefficients of the blending function for Hermite curves. We use the coefficients to represent the Hermite curve as the weighted sum of the constraints $P_1, P_2, T_1,$ and T_2 [6].

$$\mathbf{H}(t) = \left(1 - 3t^2 + 2t^3\right)\mathbf{P}_1 + t^2(3 - 2t)\mathbf{P}_2 + t(t-1)^2\mathbf{T}_1 + t^2(t-1)\mathbf{T}_2 \tag{4.19}$$

To thoroughly understand the behavior of the Hermite interpolation method, we can separate individual blending functions from Equation 4.19 and study how each function affects the interpolation. The four Hermite blending functions are

$$\begin{aligned} F_1(t) = \left(1 - 3t^2 + 2t^3\right), \quad F_2(t) = \left(-2t^3 + 3t^2\right) = 1 - F_1(t) \\ F_3(t) = \left(t^3 - 2t^2 + t\right), \quad F_4(t) = \left(t^3 - t^2\right). \end{aligned} \tag{4.20}$$

From equation 4.19 we observe that function $F_1(t)$ and $F_2(t)$ is the weight function associated with the start point P_1 and endpoint P_2 respectively. The function $F_1(t)$ starts from its maximum value $F_1(0) = 1$, and gradually descends to $F_1(1) = 0$. This is because for small values of t (closer to 0) the curve is close to P_1. But further away when the value of t increases, P_1 has minimal influence on the curve. The case for $F_2(t)$ is the exact opposite of $F_1(t)$.

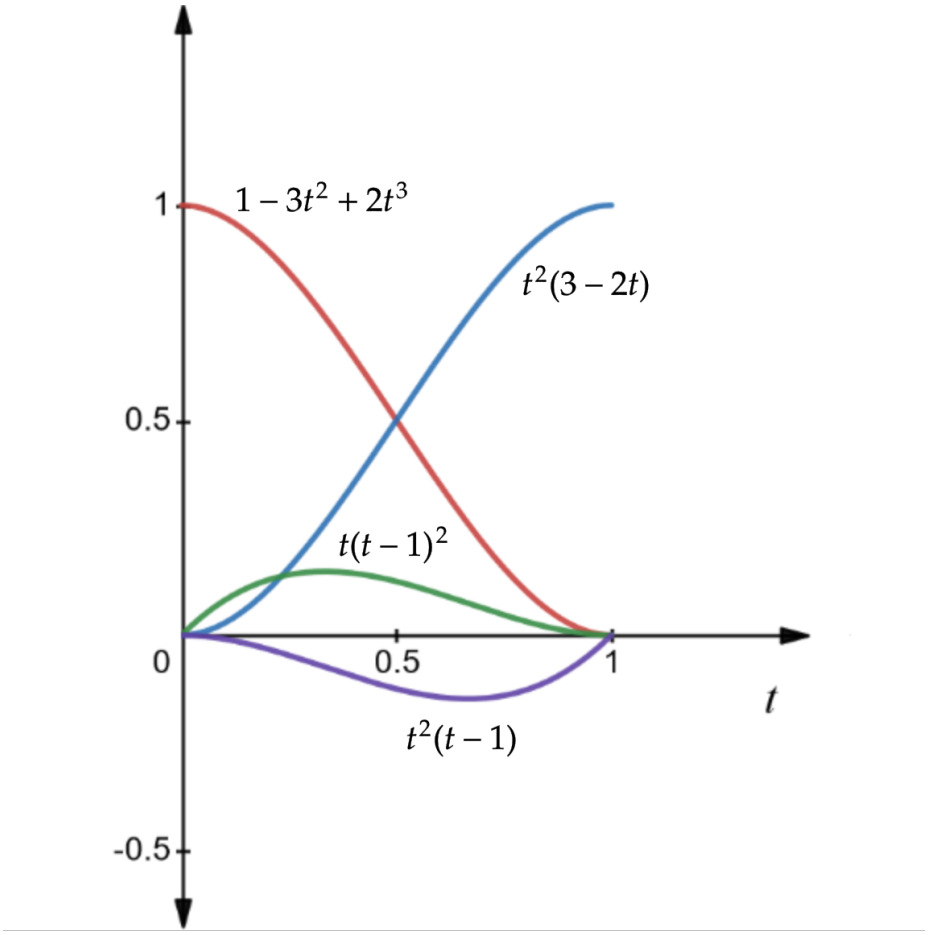


Figure 4.10: Blending functions for Hermite cubic curves.

The behavior of function $F_3(t)$ is a little more complicated compared to $F_1(t)$ and $F_2(t)$. The peak of both $F_3(t)$ and $F_4(t)$ is much smaller compared to either $F_1(t)$ or $F_2(t)$, which means the overall influence of the tangent weight functions is less than the endpoint weight functions. Function $F_3(t)$ begins as zero, gradually reaching a maximum point when $t = 1/3$. From here, the function slowly reaches zero at $t = 0$. At $t = 1/3$ the weight of $F_4(t)$ is very small, so using equation 4.19 we can write

$$\mathbf{H}(t) = \left(1 - 3t^2 + 2t^3\right)\mathbf{P}_1 + t^2(3 - 2t)\mathbf{P}_2 + t(t-1)^2\mathbf{T}_1.$$

Here the sum of the weights of $F_1(t)P_1$ and $F_2(t)P_2$ make the largest contribution.

But $F_3(t)T_1$ makes a small contribution, due to which the weight of $F_3(t)$ seems to 'pull' the curve in the direction of the tangent vector T_1. As t values increase, $F_3(t)$ once again influences the curve minimally. Function $F_4(t)$ can be interpreted in a similar manner as $F_3(t)$. At $t = 2/3$, function $F_3(t)$ has negligible effect while $F_4(t)$ has its maximum negative value. This means the curve approaches the endpoint P_2 while moving in the same direction as T_2 [11].

4.3.5 Bézier curves

The Hermite form and the cubic interpolation form have their own advantages and disadvantages. The major similarity between them is the fact that they are both cubic polynomials curves. However, it is difficult to compare the two methods as they fundamentally use different data to form an approximation[3]. During derivation of the interpolating curves, we used the data for control points. Using the same control points, we can approximate the derivative data required for Hermite curves. This results in the formation of *Bézier curves*. As it is using the same data as the interpolating curves, it is comparable to them on equal terms. Additionally, as they do not require calculating derivatives, Bézier curves are great for graphics and design purposes.

Similar to the other forms of approximation, a Bèzier curve segment is a single polynomial of order n, containing $n + 1$ control points. A Bèzier curve $B(t)$ of order n is given by the parametric function

$$\mathbf{B}(t) = \sum_{k=0}^{n} B_{n,k}(t)\mathbf{P}_k. \tag{4.21}$$

Here $B_{n,k}(t)$ are special blending functions known as *Bernstein polynomials* [6] using binomial coefficients, given by

$$B_{n,k} = \begin{pmatrix} n \\ k \end{pmatrix} t^k (1-t)^{n-k} \tag{4.22}$$

We consider the example where we have four control points p_0, p_1, p_2, and p_3 for a cubic polynomial $C(t)$. The Bèzier curve, therefore, has an order of 3, and it is given by the expression

$$\mathbf{B}(t) = (1-t)^3\mathbf{p}_0 + 3t(1-t)^2\mathbf{p}_1 + 3t^2(1-t)\mathbf{p}_2 + t^3\mathbf{p}_3 \tag{4.23}$$

The value of parameter t changes from 0 to 1 as we move along the endpoints of the curve. Assuming the two endpoints are p_0 and p_3, we have

$$p_0 = C(0)$$
$$p_3 = C(1)$$

For Bézier curves, we do not use the remaining control points p_2 and p_3 to perform interpolation. These two points are used to find the tangents at parameter values $t = 0$ and $t = 1$. Expanding the expression in Equation 4.23, we derive the basis matrix $\mathbf{M}_B$ for cubic Bèzier curve as:

$$\mathbf{M}_B = \begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.24}$$

Using the basis matrix, we can then write the Bèzier function $B(t)$ as follows:

$$\mathbf{B}(t) = \begin{bmatrix} \mathbf{p}_0 & \mathbf{p}_1 & \mathbf{p}_2 & \mathbf{p}_3 \end{bmatrix} \begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix} \tag{4.25}$$

Each of the four polynomials for each point in Equation 4.23 is an individual blending function. The blending functions for a cubic Bèzier curve are shown in Figure 4.11.

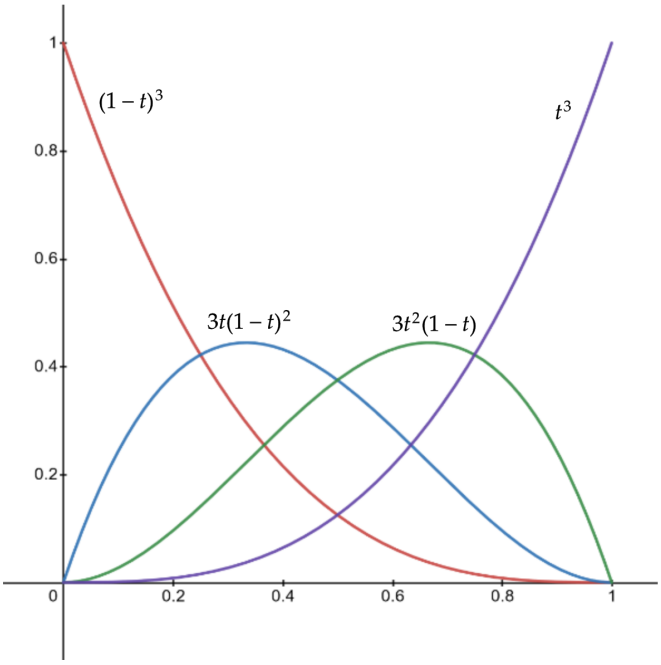


Figure 4.11: Blending functions for cubic Bèzier function

As the curve moves from its starting point to the end, it is contained within the convex hull of all the control points without actually passing through them. While Hermite curves contain controls that are vectors, controls of Bèzier curves are completely contained within the same space as the curve. This is illustrated in Figure 4.12.

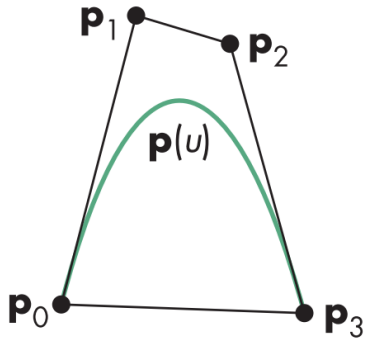


Figure 4.12: Convex Hull and Bézier polynomial [3].

CHAPTER 5

Human Perception for Over-sketched Art

The StrokeStrip algorithm uses different techniques for parameterization based on our natural perception on line drawings. There are a few factors that we consider and process when presented with a sketch. These factors have been derived from perception literature and sketching tutorials [7]. We utilize these special cases in our algorithm to fine-tune different parameters for accurate and robust digital consolidation.

Angular compatibility. Multiple studies conclude that whenever a drawing is viewed, the perspective relies on angular compatibility. This can be understood as the similarity between stroke slopes when they are viewed side-by-side(cite). If two adjacent strokes bend independently but at a similar slope, these two strokes can be considered 'compatible' . This allows the strokes to be seen as a continuation of each other, expressing the flow of the drawing. This means even when separate strokes are viewed, this helps us to see the strokes as an aggregate according to the flow.

Relative proximity. Relative distance or proximity between objects have shown to be the way humans visually group objects [14]. Given multiple shapes together, we group them together if the space between them is smaller than the space that exists between other objects in the same drawing. This form of grouping that uses relative distance is independent of scale. This means scaling distance changes the spacing throughout the image, but the relative spacing and hence the grouping stays the same. But this means to determine the 'relative' proximity, which implies

we need some context to compare the spacing with. Proximity also detects when multiple adjacent objects do **not** belong together. In the case that the drawing contains moderately even-spaced strokes, determining relative proximity does not provide us with any new information.

Connectedness. Perception research [14] suggests that inter-connected objects or strokes are also grouped together by viewers. Longer strokes could be at an angle in the middle, or curved. But if their ends are connected, we group them together as a single, thick stroke. To ensure accurate clustering, the connectedness is used to group intersecting strokes only if it does not contradict any other cues.

Narrowness/Width. Curves in general are understood as being 'narrow' by humans. Narrow here is defined as strokes with small width but a relatively larger length. Using this width-length ratio, this information becomes particularly useful in our clustering algorithm. A threshold is used to determine narrowness and can be estimated through the perception study. This is a major parameter for our algorithm.

Strength in numbers. After utilizing multiple factors for the artist's perception, comprehending some art styles would still depend too heavily on individual perception. For such situations, the artist's *intent* is leveraged over perception. The general intention for artists is to help us assemble a clear mental image of their drawings. They often rely on thicker, extended lines to separate certain strokes from others. In this case, tight multiple-strokes aid in additional clarity of drawings by putting extra emphasis on the strokes. Without the use of multiple strokes, a single stroke in such cases might produce even more confusing results. This problem is solved as a 'strength in numbers' problem where stroke numbers are used within a cluster. Numbering them allows us to use these numbers as factors for final fitting.

Combining all the information about drawings gathered till now, we can use it to judge whether groups of strokes are part of an aggregate curve. But transforming

this information into algorithmic form for the computer to execute is exceptionally difficult. There is no existing clustering framework that combines all of these artistic signals into a single algorithm. The clustering is therefore done in multiple steps, where each succeeding step carries out a more refined and local clustering than the previous. This clustering step is discussed in detail in Chapter 6. Once we have clustered our drawings into multiple aggregate strokes, we need to carry out fitting to finally produce the single line, 'clean' version of the original artwork.

CHAPTER 6

Clustering

6.1 Overview

For most vector joint consolidation processes, the first step is to cluster them into groups of aggregate strokes before fitting them. StrokeStrip uses a joint parameterization and fitting and does not include a clustering step in its algorithm. To understand the StrokeStrip method in comparison to more commonly implemented methods, we chose to study the clustering process involved in the tangent based fitting done by the project *StrokeAggregator* [7]. Clustering in StrokeAggregator is done in increasing levels of granularity. It also utilizes all the grouping cues as mentioned earlier in Chapter 5. First, a coarse clustering is carried out on the drawing based on the stroke-wise properties. This coarse clustering is further refined on smaller, local groups within the art. Finally, the method uses the final reliable clusters to finalize the stroke results.

6.2 Method Details

We have already discussed how the perception cues could be used to find an aggregate curve from a bunch of strokes. But these cues each demand a different

method to be implemented. Moreover, these visual perception cues are quite difficult to be translated into a computer program. Multiple clustering frameworks attempt to cluster in a similar fashion. But for the specific methods mentioned here, there is no existing standard framework. This means for this clustering method, a separate framework was created in accordance with these requirements. We now discuss the individual stages of this clustering framework.

6.2.1 Coarse Clustering

The first step in coarse clustering is to remove strokes that are highly unlikely to form an aggregate. This is done by sequentially carrying out the stroke cues discussed in Chapter 5. The two initial cues used are *angular compatibility* and *relative proximity* cues. The goal of using angular compatibility is to separate angle incompatible strokes while keeping almost parallel strokes together. The relative proximity cue helps us separate these roughly parallel strokes into distinct components. This is implemented using density-based clustering using the stroke-wise proximity.

The first portion of clustering is very general; the strokes are clustered in accordance with their average, or global compatibility. Firstly, coarse clustering makes use of the angular compatibility of every pair of strokes independently [7]. Strokes that are placed at a similar angle, would get grouped together. This assumes that strokes within a cluster are *roughly* parallel along side-by-side sections. For two nearby strokes S_i and S_j, an angular compatibility score $\text{ComA}(S_i, S_j)$ is calculated that attempts to deal with multiple scenarios. Positive scores imply that strokes are angle compatible, while negative score suggests incompatibility. Greater magnitude of the value means greater compatibility (or incompatibility).

$$\text{Com}\,A\left(S_i, S_j\right) = \begin{cases} 1, & \phi < 8^\circ \\ \exp\left(-\frac{\left(\phi - 8^\circ\right)^2}{2\sigma_1^2}\right), & 8^\circ \leq \phi < 17^\circ \\ 0, & 17^\circ \leq \phi < 23^\circ, \\ -1.5\exp\left(-\frac{\left(\phi - 23^\circ\right)^2}{2\sigma_2^2}\right), & 23^\circ \leq \phi < 30^\circ \\ -1.5, & 30^\circ \leq \phi \end{cases} \tag{6.1}$$

Here ϕ is the angular distance between the aggregate curve and each stroke. The values of ϕ have been derived from perception research. From perception literature, 20° is considered as the threshold value for viewers to differentiate between similar and not-so similar tangent values in strokes [14].

After the initial segmentation, we carry out a bunch of refinements to get to our final result. The first step of refinement is by using the *relative proximity* between stroke clusters that have been formed already. We want to separate the initial, roughly-parallel strokes into distinct components. This is implemented using **density-based** clustering using the stroke-wise proximity. Given that the strokes are already angle compatible, this step measures the average spacing between strokes in the same cluster. This creates sub-clusters with strokes that have approximately the same space between them. Following this step, the width of the resulting sub-clusters is assessed along with its local spacing uniformity [7]. These two cues are used in conjunction to identify clusters that are only weakly connected. This means strokes from one cluster have a few branches that travel into another cluster and vice versa. This provides us with an output that satisfies all the perceptual criteria we mentioned in the human perception of over-sketched art.

Till this stage, the edge cases were discussed as more of a nuance. But we have to consider clustering them as well to get the most accurate results. These outlying clusters did not present any solid evidence for compatibility till now. But we realize

that at this stage of clustering, the algorithm has already produced clusters that contain multiple strokes. We can now focus on these clusters and refine the strokes within them.

6.2.2 Fine Clustering

The 'relative' proximity calculation in the previous step requires context to be judged. This implies the algorithm would not perform well on any strokes or stroke-pairs that are relatively distant and do not have any adjacent stroke pairs. For any given stroke pair, the distance between them can vary quite a lot. There would be strokes that branch into other strokes but are originally from a different curve. To overcome such issues, the algorithm carries out a *targeted clustering framework* that improves cluster refinement gradually. This is done by integrating new and more confined perceptual cues into the clustering.

In any sketch, it is common for local parts of the sketch to contain multiple varying proximities. This is known as *branching*. Branching is separated by a recursive method based on **local contextualized relative proximity**. In the coarse clustering step, the proximity for strokes was determined in isolation between a pair of strokes. In this fine clustering step, we compute the proximity based on the local point-wise value relative to *all* the strokes within the cluster. This approach calculates the **inter-distance** between points on different strokes. The inter-distance is then compared against *inner-distance* between points on the same stroke. If the inter-distance is much larger compared to the inner distance, it indicates that the two strokes being compared should be separate. This process is repeated for multiple points at consecutive locations on the stroke for reliability.

6.2.3 Final fitting

After carrying out these steps, we are left with distinct clusters of strokes. Strokes in these clusters belong to the same group according to our algorithm and are called 'reliable' clusters. With our stroke grouping complete, these reliable clusters themselves are re-assessed with our earlier cues. If they satisfy all our cues, they are merged together to get the final result.

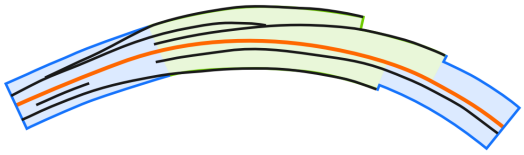


Figure 6.1: Different colors represent different envelopes

As most individual clusters contain multiple strokes, the re-assessment is more reliable for calculating the width of the aggregate curve. Every cluster is first fitted with an aggregate curve, and widths of these curves W_c are computed. We check if the distance from the aggregate curve to the outermost intersection with a cluster stroke is greater than $0.5 * W_c$ for that curve. If the condition is fulfilled, then this is considered a sub-section within the cluster. Within this sub-section, the point of intersection as mentioned in the condition is regarded as *envelope vertex* [7]. For sections of the curve that do not satisfy the condition, a 'ray' is passed through the half-width distance. These rays are considered 'boundary' for the envelope. The idea of the envelope is illustrated in Figure 6.1. Once we have two boundaries with an envelope vertex between them, we pass a ray from one boundary to the other, passing through the envelope point to create the aggregate curve for this section. The aggregate now roughly reflects the average width of the cluster, while being completely contained within it as well [7].

CHAPTER 7

Fitting Curves

In this chapter, we explore smoothing and curve fitting for sketch inputs. There are multiple methods to group strokes that together form our input images. The problem of combining these strokes to a proper fit has been relatively less explored. Fitting the strokes involves multiple factors to be considered that help us achieve an accurate depiction of the intended drawing. The older methods mentioned in the StrokeStrip paper are Orbay-Kara [9] and StrokeAggregator [7]. The approach by Orbay and Kara used spectral embedding to order the points, which were then fitted to a curve. StrokeAggregator utilizes a spanning tree for the moving least squares method to order the points. The algorithm then fits the curve to the longest path as seen. The major focus for StrokeStrip, however, was to produce a completely new method to fit these intended curves to vector stroke clusters.

In terms of fitting curves, the perception study helped us observe that humans view clusters of strokes as continuous, varying-width *strips* whose paths are described by the intended curves [13]. This is a fundamental observation in this paper, as all the fitting is done by turning each stroke cluster into strips. This helps to construct the geometry of the drawings provided, without any actual geometry data provided to us. Eventually, this method is what provides the robustness of StrokeStrip's performance.

For this paper, we first study the curve fitting process in the project "Beautification

of Design Sketches Using Trainable Stroke Clustering and Curve Fitting" by Gunay Orbay and Levent Burak Kara [9]. The StrokeStrip project uses a few principles fitting that has components and conclusions from multiple papers and studies, including this project. We explore the fitting method by Orbay and Kara to understand the contrasting difference between this fitting and StrokeStrip's own fitting method.

7.1 Orbay and Kara fitting

Suppose we begin the parameterization process with a single stroke. We want to assign a number to every point on the stroke that should smoothly go from 0 to the magnitude of length of the stroke. This denotes that 0 is the beginning of the stroke and we reach the other end of the stroke by traversing the total length of the stroke.

One approach would be to just calculate directly the fraction of the distance along the stroke for each point. The value of this fraction would be the desired parameter value. Although this would give us the most accurate answer, we explore another method more suited for our purpose. We can also think of the parameterization as an optimization problem: we want the *difference* in value between two adjacent points to be as close as possible to the distance between those two points. The constraint here is that we have set one point to be 0 and another point to be the total length. If the stroke is divided into a collection of discrete line segments between points, this could be thought of as a *least squares* sort of optimization. Each discrete segment are regarded to be "as close to the distance as possible". These segments now become the error terms that we want to minimize through our optimization. The mathematical explanation is further clarified in Figure 7.1.

For any point set, we start from one end and locally fit an approximate quadratic

curve to a fixed number of points [9]. The vertical projection of every point on the original point set (x_p, y_p) onto the curve is then calculated.

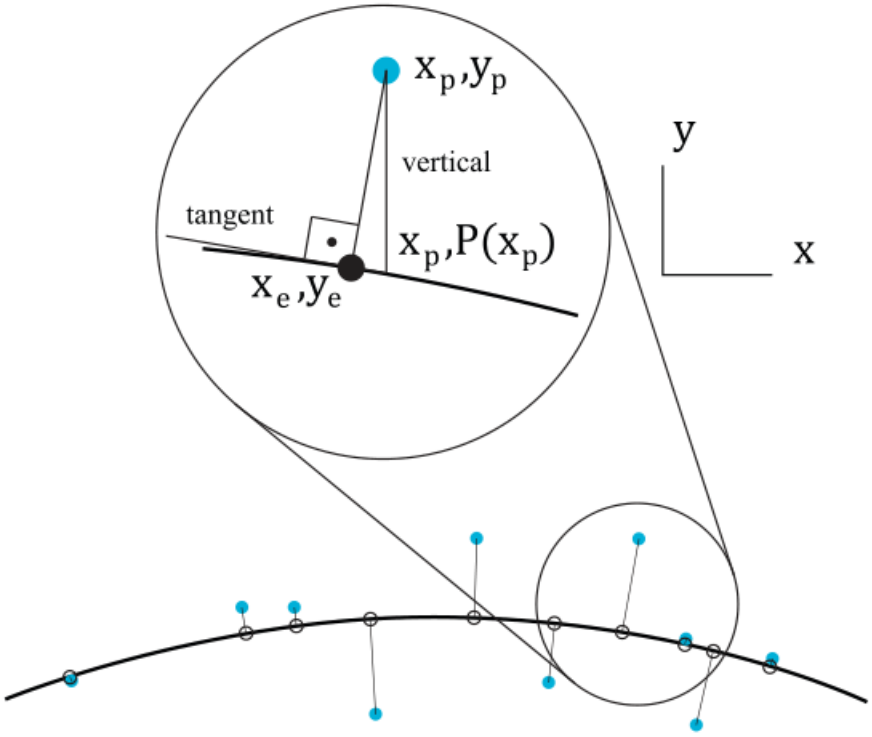


Figure 7.1: Approximation of normal projection on the curve [9]

This produces the projection point $(x_p, P(x_p))$. The projection point is then run along the local tangent to the curve at this point. This is done until we reach a point (x_e, y_e) where the distance to the original point (x_p, y_p) is minimized. When repeated for all the points, we obtain a first order approximation of the normal projection of (x_p, y_p) on the local quadratic curve [9]. Using the distances between these projected points, a final parameterization is computed. If the new parameterization produces a point ordering that is not equivalent to the original order, the point set is rectified accordingly. This process is repeated several times over the original coordinates until the endpoint of the stroke is reached. Using the final parameterization, the

entire point set can be approximated to a curve using cubic B-spline approximation.

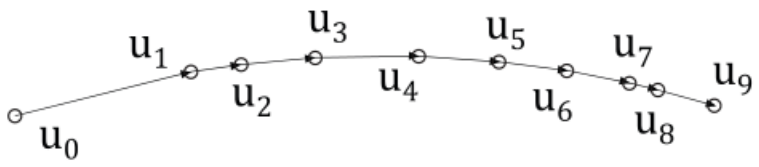


Figure 7.2: Ordered points for parameterization [9]

As we have seen in Chapter 4, cubic approximation for curves depends on control points. For B-spline fitting, in this case, the number of control points can be set by the user. If we ignore the variation in the fitting caused by pen or stylus pressure, the approach for the fitting involves minimizing an error-function defined between points on the curve and the original points. For K control points, this error function can be written as follows:

$$E = \frac{1}{2} \sum_{k=1}^{K} \|\mathbf{P}(u_k) - \mathbf{X}_k\|^2 . \tag{7.1}$$

Here, X_k is the actual position vector for kth point and $P(u_k)$ is the position vector of that point on the curve with a parameter value of u_k.

7.2 StrokeStrip Parameterization and Fitting

Overview

The major working assumption for the fitting method used by StrokeStrip is that users perceive stroke clusters as strips with varying widths. The perception tells us that we follow these hypothetical strips along a path that follows the artist-intended curves. Human observers portray the outline of these strips mentally by viewing

cross-sections along the paths that follow these aggregate curves [13]. This helps us determine the points in the drawing that are next to each other within the strip. This distinction is important as we can also judge whether the points are within the strip or simply nearby considering Euclidean space. A number of points lie neighboring in the Euclidean space but is part of a different section of the observed strip. These particular findings resulted in the formation of a 1D parameterization problem.

7.2.1 Problem Statement and Goals

Before we discuss the different properties of our parameterization, we first define in simple terms, what exactly 'strip' means for this project.

The three major aspects of a strip are :

- **Centerline** The centerline of a strip can be considered as the 'backbone' of the strip. This is a single curve that passes through the center of the strip from one end to the other.

- **Arclength** Centerline of the strip is directly related to the arc-length of the strip. This is the one-dimensional length of the strip if we imagine it to be flattened into a straight line.

- **Width** The width of a strip is the same as the width of the cross-section of the strip. This can also be imagined as twice the distance of the edge of a strip from the centerline.

A strip is essentially two separate arcs connected together. These arcs can be of varying width and length, but as long as it is closed, it forms a strip. This allows a strip to be parameterized naturally using its arc-length. Every stroke cluster would contain *isolines* that help us define the cross-sections of the strip. During the parameterization, the isolines are orthogonal to the path. If we assign smoothly

varying isovalues to all points on the strip, then we can parameterize the strip. This means the natural arclength 1-D strip parameterization can be defined by the arclength of the strip's path. This parameterization can then be extended to the strip using path-orthogonal isolines. The resultant parameterization isolines are straight and evenly spaced on average. This allows them to cross the strip from one side to the other while forming cross-sections, as shown in Figure 7.3(b).

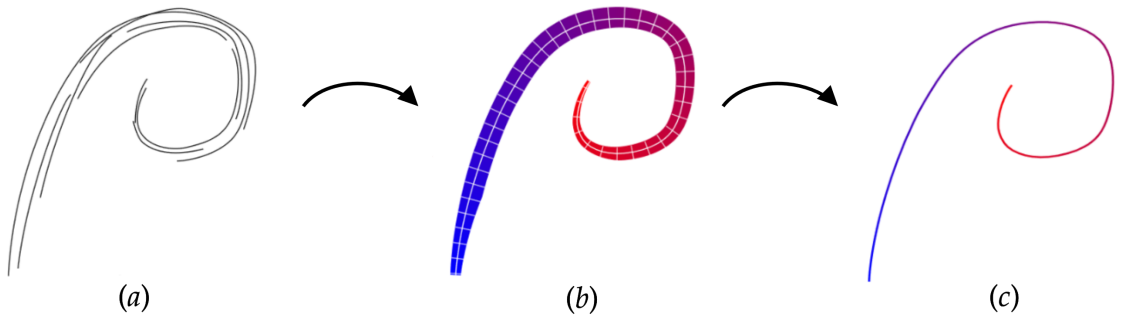


Figure 7.3: Path and Strip parameterization followed by Fitted curve

Once this joint parameterization of the strokes on the strip has been determined, we can directly compute the desired parametric fitted curve. The shape of this new curve - its position, tangent, and curvature - at each parameter value reflects the average shape of the input strokes. This means the fitting problem used by StrokeStrip is essentially a parameterization problem first, which then uses this same parameterization to fit the curve, shown in Figure 7.3.

To get a clear understanding of which methods to use to predict a-priori geometry, we first discuss *what* properties of the arclength parameterization are desired.

- **Tangent Alignment** For every input stroke clusters, we first check the direction of the strokes in this specific cluster. We seek strips whose path is aligned

with this direction. This can be achieved by comparing the gradient of the parameterization with the tangents of the strokes in the cluster. If these two values are aligned, the tangent alignment property is then satisfied.

- **Arc Length Preservation** The parameterization for our fitting must be arc-length preserving. We calculate the gradient of the arc length of the strip and average it over an isoline. These should always be perpendicular to each other and should have the same unit value over the strip. This ensures an even sampling of stroke points is achieved when the subsequent fitting is carried out.

- **Monotonicity** For a strip, we observe that the arc length parameterization would always be *monotonic* to its gradient direction. Recall that a monotonic function is a function that does not change direction. In other words, monotonic functions are either always increasing, or decreasing. For our case, we expect the parameter value to keep increasing or conversely, keep decreasing. As we have already established our property of tangent alignment, we expect the gradients of the parameter (u) for all the strokes along their isolines to be similar.

- **Isoline Span** According to the perception research, we consider the fact that viewers perceive nearby strokes with similar tangent values to be adjacent within the strip. Therefore, we want the parameter values on these different strokes (but perceived as adjacent) to be similar. This property is referred to as isoline span [13].

All these requirements mentioned are combined to form an objective function. The objective function is provided with a set of constraints. All the parameterization satisfying these requirements is considered as *joint* [13]. In the following section,

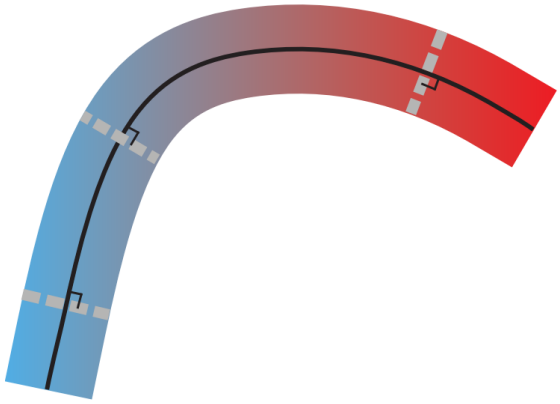


Figure 7.4: The notion of a strip from its arclength [13]

we will discuss the formulation of the objective function along with the different constraints.

7.2.2 Formulation

Mathematically, a strip can be described using its centerline. The centerline can be written as $\gamma(t) \in \mathbb{R}^2$ where t is the arclength $t \in [0, L]$ and width $W(t)$. The width $W(t)$ is always perpendicular to the centerline at any point. The width is essentially twice the distance from the sides, towards the direction of normal. The overall expression for the centerline then becomes $\gamma(t) \pm \frac{W(t)}{2} n(t)$, where $n(t)$ is the normal direction at arclength value t. We can visualize the strip similar to Figure 7.4. Here the thin black curve passing through the middle of the strip is its centerline which depends on the arclength of this curve. The dotted grey line represents the isoline at that point, and the strip is shaded from blue to red in accordance with its parameter values. The parameter values across the isolines should remain constant.

For a strip $\mathcal{S} \subset \mathbb{R}^2$ that is not self intersecting, the arc length parameterization $u : \mathcal{S} \to \mathbb{R}$ minimizes the following variational problem:

$$\min_u \int_0^L \left\| \frac{1}{W(t)} \int_{C(t)} \nabla u(x) dx - \tau(t) \right\|^2 dt. \tag{7.2}$$

Here the outer integration is done with respect to t which is the arclength for the centerline. The inner integral is over the cross-section $C(t)$ with length $W(t)$, which crosses the strip perpendicular to the strip [13]. However, this formulation cannot be directly applied to strokes, since the gradient of u - denoted as ∇u - is not clearly defined. Here we leverage the fact that the gradient for arc-length parameterization is always orthogonal to the path tangent. Moreover, the average over each of the gradient cross-sections has unit length. This allows us to write the parameterization in terms of projection of the expression inside the norm onto the centerline's tangent and normal [13].

Now, we seek the function u whose domain is the strip formed from the set of strokes in our strip. The range for this function is the arc length, and the functions minimize the functionals E_{length} and E_{align} while satisfying a monotonicity constraint.

The formulation in this manner is shown as follows:

$$\begin{aligned} E_{\text{length}} &= \int_0^L \left| \frac{1}{W(t)} \int_{C(t)} \nabla u(x) \cdot \bar{\tau}(t) dx - 1 \right|^2 dt \\ E_{\text{align}} &= \int_0^L \int_{C(t)} |\nabla u(x) \cdot \bar{n}(t)|^2 dx dt \\ \min_u E_{\text{arc}} &= \min_u \left(E_{\text{length}} + E_{\text{align}} \right), \end{aligned} \tag{7.3}$$

where $\bar{\tau}$ is the average of stroke tangents τ along the cross sections. The term $\bar{n}$ is the vector perpendicular to $\bar{\tau}$. This formulation satisfies two goals - arc length preservation (E_{length}) and tangent alignment (E_{align}). The tangent alignment here implies that the tangents of the aggregate curve and the gradient of the parameterization that defines it should align with stroke tangents.

As mentioned earlier, the formulation is also subjected to a monotonicity constraint. Suppose we describe a sample on an input stroke as $\{s, i\}$ where i is the

index of the sample. We can denote the position of the sample as $p_{s,i}$. The length of a segment along a specific polyline can therefore be written as $l_{s,i}$ where

$$l_{s,i} = \left\| p_{s,i} - p_{s,i-1} \right\|$$

between sample points $p_{s,i}$ and $p_{s,i-1}$. We can also define the gradient orientation as $o_s \in \{1, -1\}$. The orientation $o_s = 1$ means gradient at stroke s is ascending, and $o_s = -1$ implies the gradient is descending. Combining these notations, we can now write our constraint for the minimizer problem:

$$(u_{s,i} - u_{s,i-1})\, o_s \geq \frac{l_{s,i}}{2} \forall s, i. \tag{7.4}$$

Geometrically, each of the functionals in Equation 7.3 has a distinct interpretation. First, we look at the arc-length preserving functional E_{length} . The term $\int_{C(t)}$ means we are integrating along each cross-section of the strip and then $\int_0^L$ denotes that we are integrating along the length of the strip. The term $\nabla u(x) \cdot \bar{\tau}(t)dx - 1$ tells us that u should change at the rate of 1 in the direction of aggregate curve. For our tangent alignment functional E_{align} , we also integrate along cross-section and length of the strip similar to E_{length} . The term $\nabla u(x) \cdot \bar{n}(t)$ tells us that at each point, we want the gradient of u to be orthogonal to the normal to the aggregate curve. This is crucial as this would make the isolines formed during parameterization to be orthogonal to the aggregate curve, and would align the tangents.

7.2.3 Solution and Algorithm

With the functionals mentioned in Equation 7.3 and the constraint in Equation 7.4, we can attempt to formulate the parameterization. But the strip geometry is still unknown, which means we do not have a continuous $C(t)$ as formulated. Without the strip geometry itself, we lack a proper definition for the cross-section. To find

$\min_u \left(E_{\text{length}} + E_{\text{align}}\right)$ we need to solve for three sets of variables. These variables are :

1. The parameter values at all stroke points.
2. The gradient orientations of each stroke.
3. A set of isoline cross-sections defined for parameter values of u, including the set of stroke points that lie on each of these cross-sections.

The parameter values u are continuous, but we need to discretize the $C(t)$ to find the set of cross-sections. We have the monotonicity constraint as well, which requires discrete gradient orientations. The gradient orientation o_s can be reliably predicted for the parameterization gradients along individual strokes. This is therefore our first step for parameterization. Next, the parameterization cross-sections and parameter values along them are calculated simultaneously using a variational framework [13]. Lastly, this parameterization would allow us to define parametric fitted curves for the final aggregate result. We now look at these steps individually to get a better understanding of each step.

7.2.3.1 Pairwise Gradient Orientation

Our goal is to maximize isoline span and tangent alignment properties for our strokes. This means we aim to maximize the similarity of gradient direction for strokes that are roughly parallel. Taking each pair of adjacent strokes, the most compatible orientation is computed locally. For stroke pairs that might have multiple strip interpretations, orientations that are consistent with narrower strips are preferred. Additionally, any orientation that produces isolines with diverging gradient is avoided. Once the pairwise local compatibility has been determined, the impact of this orientation is checked against the global solution. We take the global orientation that overall maximizes compatibility between stroke pairs. The

formulation and solution to this optimization are done using integer programming [13].

7.2.3.2 Core Parameterization

For solving the parameterization problem, the continuous strip parameterization formula in Equation 7.3 is discretized and restricted to the strokes. This creates a discrete, energy functional that can be optimized for getting the parameterization. But we still face the challenge of finding which points adjacent in Euclidean space are also adjacent within the strip. This is solved as a multi-step problem. First, a set of close points are identified, that is likely to be strip-adjacent. This is done by generating a set of stroke-orthogonal cross-sections. Using these cross-sections we compute an approximate parameterization to check the likelihood of pairs of points on the current cross-sections of being adjacent. This computation is an iterative computation process that alternates between calculating parameter value and likelihood update steps. This approach is known as the *coordinate descent* method. It essentially means we only optimize one thing at a time, and go back and forth optimizing one property or the other until the results stop changing. During this step, we aim to filter highly unlikely pairings of strokes. This includes strokes with poor tangent similarity, outlier inter-stroke distances, large gaps between connections, etc.

7.2.3.3 Curve Fitting

Once we reach a point where our previous parameterization method converges, we use the final parameterization to compute a fitted curve. The shape at each parameter value u should reflect the shape of input strokes at this u value. During this computation, the fitting accounts for mainly three properties: positions, tangents,

and curvatures. A higher priority is given to curvature and tangent preservation over position approximation, as suggested by observations from prior work [13].

CHAPTER 8

Methods

One of the main goals of this project was to use StrokeStrip's algorithm on various types of input. This includes creating our own inputs and applying fitting to explore the artistic perspective of the project. The method for using custom input files involves primarily four steps:

1. Draw the desired graphical input in vector format.
2. Convert the vector image to the appropriate input format (.scap).
3. Break the image into different groups of aggregate strokes.
4. Perform fitting on the file produced in step 3.

Each step involves the use of different software and/or script to carry out our goals. In this chapter, we discuss each of these individual steps in detail and understand how they combine to give us clean line drawings from our drawings.

8.1 Input using Adobe Illustrator

To create our own input, we require a drawing tool that can support .svg format export. For this paper, the custom input files were created using Adobe Illustrator version 2.1. The device used was iPad Air (4th generation) running on iPadOS 15.3 alongside Apple Pencil with pressure sensitivity disabled. For drawing the input,

multiple drawing tools were tested to check compatibility with the StrokeStrip code. This includes tools such as the pen tool, pencil tool, blob brush, etc. The only tool that always produced compatible .svg files was the pencil tool. The pencil tool was selected with a 1pt, thin stroke setting with the fill setting turned off. Inputs were drawn on a letter-sized (612×792 pt) canvas which was exported as .svg files. During export to .svg format, the font was selected as SVG. Selecting 'outlines' as the font during extraction would cause the strokes to be hollow outlines which is not optimal for our purposes. Once exported, the .svg file is ready for conversion in the next step.

8.2 StrokeAggregator

For our next step, we convert valid .svg vector drawing files to .scap files. We use the `svg_to_scap.py` script from the StrokeAggregator project [7] to generate input files for the next step.

8.2.1 Input Format

The input files for the project are vector drawings with the custom format with `.scap` extension. In .scap files, drawings are stored as strokes that are encoded as *polylines*. The format of .scap files is shown below:

```
#[width] [height]
@[thickness]
{
#[stroke_id] [cluster_id]
[x1] [y1] [pressure1]
[x2] [y2] [pressure2]
[x3] [y3] [pressure3]
```

```
[...etc]
}
[...etc]
```

Individual stroke information is contained in pairs of braces `{ ... }`. This information contains a unique *stroke ID* for individual strokes in the drawing. It also contains a unique *cluster ID* which is the same for all strokes that collectively form a single aggregate curve. The width and the height values collectively determine the origin of the drawing relative to the dimensions of the canvas. The stroke polylines are simply x and y coordinates in the Cartesian plane, with respect to the determined origin. Each coordinate also contains a field for pressure value, which is taken under consideration if the drawing input was done using a pressure-sensitive stylus. For this paper, we do not take the pressure value into account and omit it by setting it to a default value of 0. The thickness value in the file lets the user incorporate the thickness of the stroke for fitting. By default, we consider the thickness of all strokes to be the same and set it to 1 for most cases.

As the .scap files are very specific to this specific project, it is quite difficult to generate them from scratch. Simple conversion from usual vector image formats is difficult too, as we need to extract the information from individual files and format them in the exact order of the .scap format. For this reason, we use .svg files as our preliminary file format and then convert these files into .scap files.

8.2.2 Svg to Scap Conversion

The **SVG** (Scalable Vector Graphics) file format is a popular two-dimensional vector graphics format for illustrations [1]. Image information is stored in text format written in **XML** (Extensible Markup Language) format, which gives it a considerable amount of flexibility, especially for web-based usage.

The python script `svg_to_scap.py` reads SVG files and converts the mathematical information contained into stroke information for the SCAP files. This is done entirely through the command line and requires the installation of Python and the `svgpathtools` package. The script also re-parameterizes the path from SVG during conversion. There is an option to control sampling step size during re-parameterization using the command line argument `-r SAMPLE_STEM_MM`. The resampling and step size only apply to parametric curves such as Bèzier curves, but not to piecewise linear curves. The step size is also useful as we can reduce its value if our input contains minute details.

8.3 Aggregator Labeller

Once we have converted our .svg file to .scap file, we move to the next step which involves labeling the strokes into their respective clusters. This is a required step as the newly converted .scap file contains the whole drawing as one single group. Our main script for fitting curve expects input files to have multiple aggregate strokes for different sections of the image, which means it will not accept .scap files with a single curve as valid input.

We use the labeler to only group strokes that are perceived as a single aggregate stroke. For this step, we use the JavaScript application from the StrokeAggregatorLabeller project [10]. The web application is run through a browser by hosting the application on our local server. This is done by using **SimpleHTTPServer** in python. A local server allows us to statically server files from our own directory. This lets the JavaScript app dynamically read files from this directory and perform the labelling task.

8.3.1 Application UI

The user interface (UI) of the application is shown in Figure **??**. Once a .scap file is loaded onto the browser, the sketch with overdrawn strokes is viewed. Some of these strokes can be grouped together to depict the clean version of the same curve. As a user, our task is to interactively group strokes in the drawing that we perceive to be a single clean curve.

The cursor acts as a small circular selection tool that helps the user to choose strokes or choose a specific point on the stroke. The input drawing is shown on the screen with all the buttons on the right side of the screen. Clicking on any of the buttons activate the function of the button. Once a change has been made to the drawing, the user needs to click on *Confirm Change* button if satisfied with the outcome.

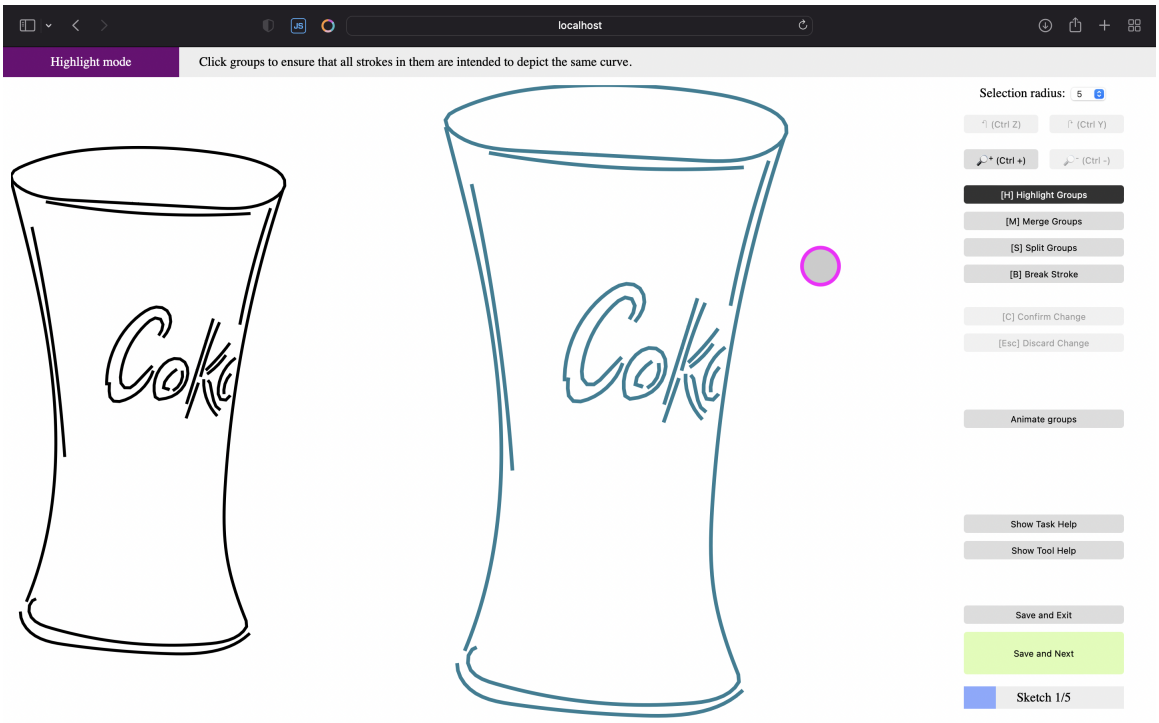


Figure 8.1: StrokeAggregator Labeller user interface

Otherwise, the user can also disregard the immediate changes made by clicking

on *Discard Change* button. Each button serves a different purpose, but there are mainly five buttons that serve the purpose of the labelling task. We now discuss the function of each of these five main buttons.

Highlight Groups When this is activated, the cursor highlights any selected group in pink while reducing the transparency of all other groups in the background. If a single stroke is selected using this tool, it shows a warning message notifying the user that a single stroke is not a part of any group.

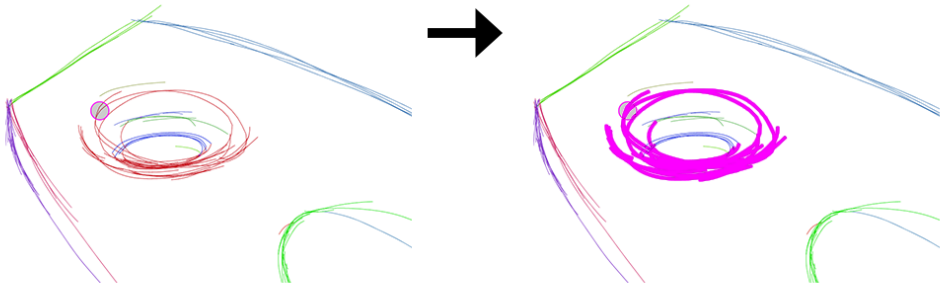


Figure 8.2: The group highlighted is shown in pink

Merge Groups This button allows the user to select two or more groups to be merged together. This mode also lets the user select a single stroke and then select the group it belongs to or another stroke, and merge them.

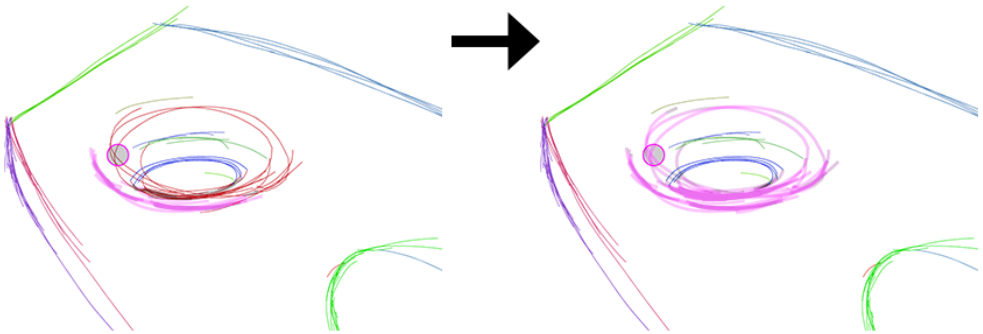


Figure 8.3: Merging the selected group (on the left) with the newly selected group (on the right). The selected group is given a light pink shade in both pictures.

Split Groups The split group button allows the user to select one or more strokes within an existing group to split the selected strokes into a new group. The desired strokes turn red when selected, indicating that these are the specific strokes that leave the group to form a new group on their own.

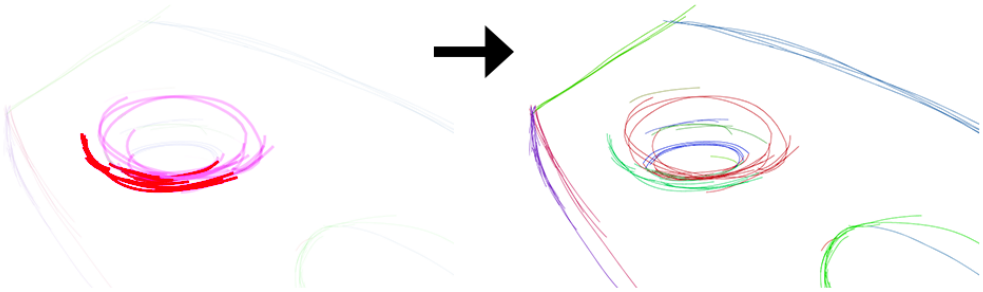


Figure 8.4: Splitting a group (marked red) from a larger group (marked light pink on the left).

Break Stroke The purpose of this button is to break a single, continuous stroke into two separate segments that belong to different groups. When this option is selected, the user is prompted to select a stroke. Once selected, a pointer appears on the stroke which can be moved to the point where the user wants to break the stroke.

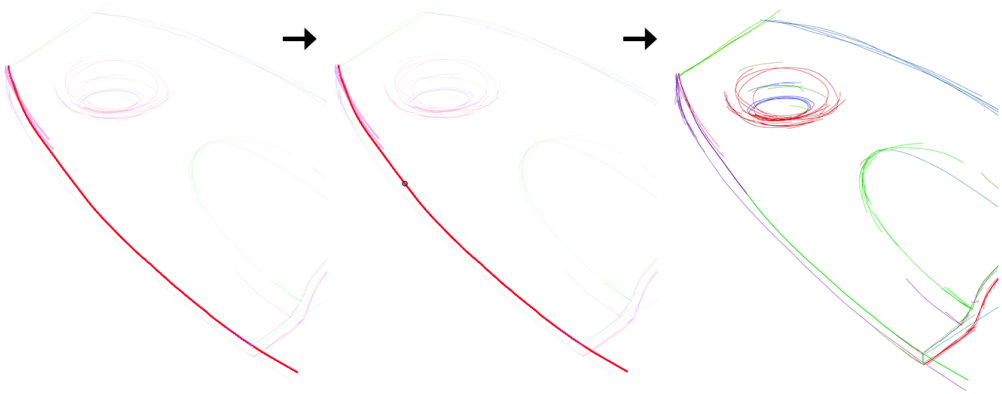


Figure 8.5: Breaking the stroke (marked red) at the selected point (red dot), shown in the center picture.

Animate Groups This button does not have any direct effect on the strokes, but is useful to visualize the different groups in a drawing after labelling has been carried out. Clicking this button highlights individual groups in the order they were created to create an animated representation of all the different groups present.

8.3.2 Labeller Usage

The main goal of the labeler is to form different groups of strokes within the original drawing, where each group represents a single aggregate curve. Without separating the individual groups, the fitting algorithm would group the entire drawing into a single curve and attempt to fit them together. This would fundamentally be impossible and would cause an error in the code, failing to generate any files.

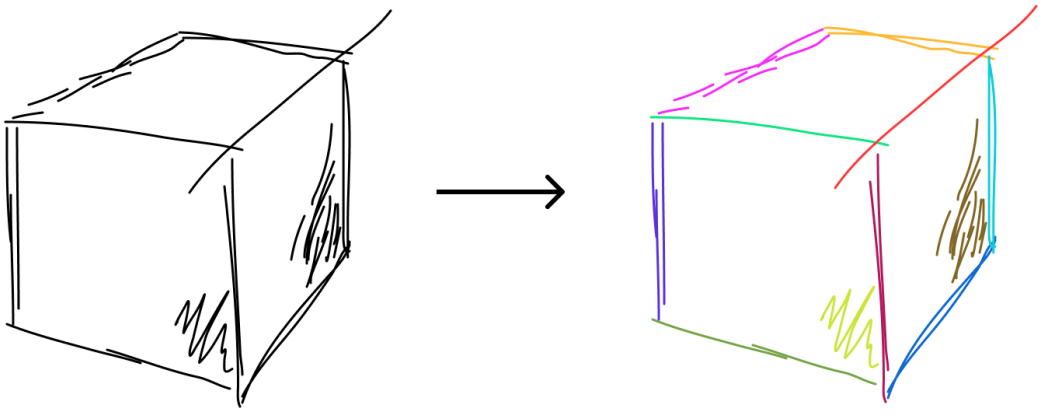


Figure 8.6: Input drawing grouped using labeler

As we observe and mentally group the strokes together, we try to incorporate the human perception factors as mentioned in Section 5. As shown in Figure 8.7, some strokes might be disconnected from the group as a whole but can be very similar in terms of distance and tangent directions. This would imply these strokes are in

the same group. For strokes connected in Euclidean space with different tangent directions has to be split into two different strokes. Finally, in the case where the strokes are very similar in terms of direction but contain a significant amount of space between them, they should be regarded as different strokes as well.

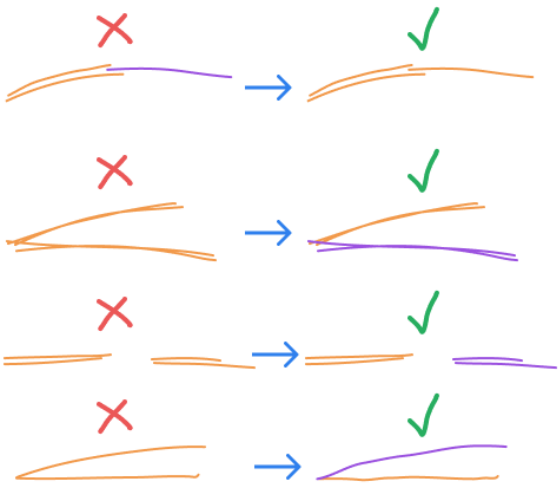


Figure 8.7: Examples to demonstrate appropriate stroke grouping. Here, the same color represents the same group.

Sketches with shadings are required to be separately grouped as well. In the example shown in Figure 8.8, we group the shading (brown) separately from the non-shading (blue).

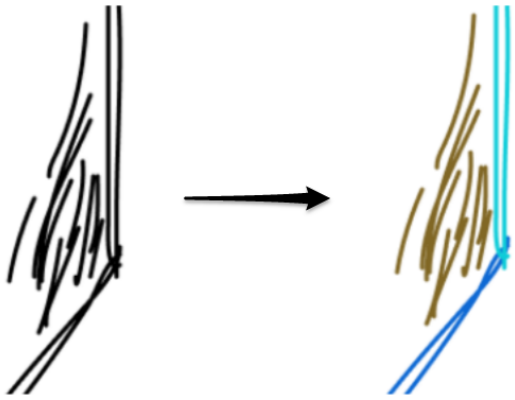


Figure 8.8: Shading

Sharp turns within stroke produce segments that move in different directions. Even with a direct connection between strokes, there is a significant change in the tangent direction between the two segments. This means these turns will always be grouped differently, as shown in Figure 8.9

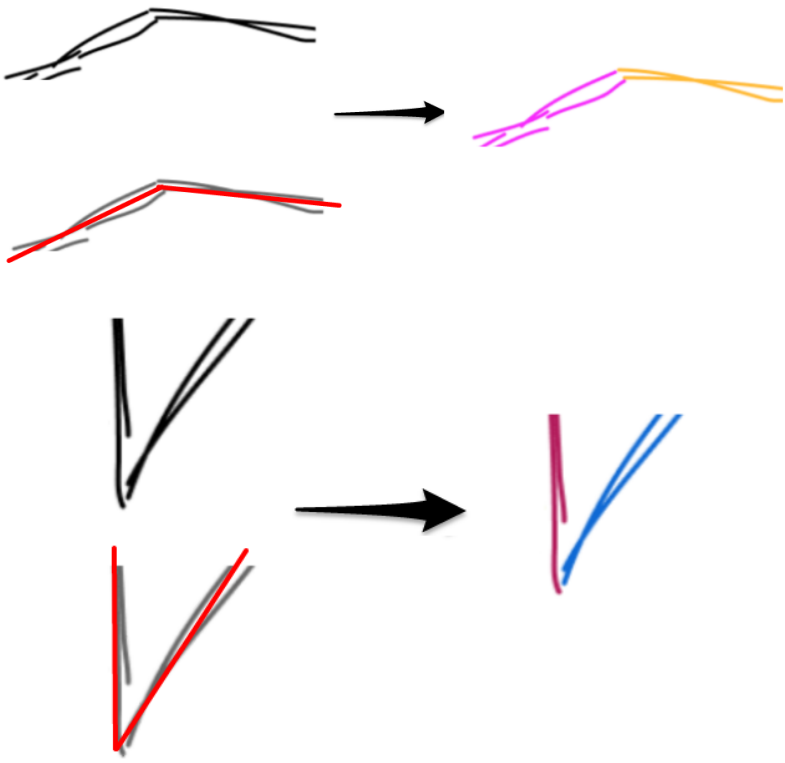


Figure 8.9: Grouping strokes with sharp turning points

Finally, strokes can form smooth overall curves from individual smooth strokes. Strokes that form curves through such smooth transition should be grouped together.

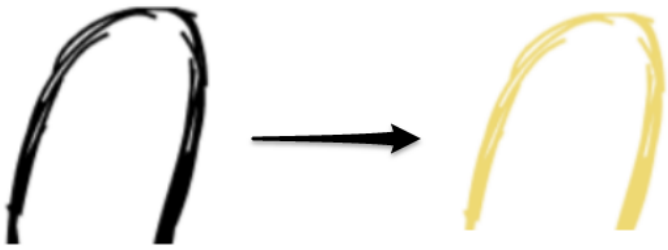


Figure 8.10: Smooth curves as a single group

Once the sketch has been successfully labelled into different groups, the user can either press the **Save and Exit** button or the **Save and Next** button on the UI. The **Save and Next** button prompt the user to save the new .scap file for the processed sketch and then moves to the next image to be labeled. The next images are determined by the order set in the `input.js` file in the data folder. Clicking the **Save and Exit** button first prompts the save option, then exits the labeler application from the browser. Using this labeled .scap file, we can run the main StrokeStrip script to fit the stroke clusters.

8.4 StrokeStrip Fit

For the final fitting, we use the main project discussed in this paper - StrokeStrip - to perform joint parameterization and fitting of the input drawing [13]. The .scap file to be processed is placed in the appropriate folder and the script is run. Once processed, it generates different .svg files during the fitting process, including the final fitted sketch.

8.4.1 Prerequisites

For our experiments, the code was run on a Mac M1 (2020) with macOS Monterey version 12.2.1. The project is written in C++ language, requiring the installation

of a few third party libraries. The first one is the **Gurobi Optimization** library which is a mathematical programming solver for different types of mathematical problems [2]. Although this library requires a license, a free academic license is available for college students, which was used to register and use for our code. It is crucial to make some changes to the `CmakeLists.txt` file and the `FindGUROBI.cmake` file to allow the system to detect and successfully use this library. The version of Gurobi used for the project is version 9.5.0. In the `FindGUROBI.cmake` file, we edited the `find_library` function to account for the version number. This was done by changing the gurobi version number to gurobi95 as shown below.

```
find_library(GUROBI_LIBRARY
    NAMES gurobi gurobi95
    HINTS ${GUROBI_DIR} $ENV{GUROBI_HOME}
    PATH_SUFFIXES lib)
```

Additionally, a few lines were added to the `CmakeLists.txt` file to include the path for finding the library folders including the `GUROIB_CXX_LIBRARY`. The additional lines are shown below. It is worth noting that the individual paths and version numbers will vary across different machines and Gurobi versions.

```
find_path(GUROBI_INCLUDE_DIR
      NAMES gurobi_c++.h
      PATHS "$ENV{GUROBI_HOME}/include"
            "/Library/gurobi950/macos_universal2/include"
      )


find_package( GUROBI_LIBRARY
              NAMES gurobi
    gurobi45
```

```
    gurobi46
            gurobi50
            gurobi51
            gurobi52
            gurobi55
            gurobi56
            gurobi60
            gurobi65
          PATHS "$ENV{GUROBI_HOME}/lib"
                "/Library/gurobi950/macos_universal2/lib"
          )


find_package( GUROBI_CXX_LIBRARY
              NAMES gurobi_c++
              PATHS "$ENV{GUROBI_HOME}/lib"
             "/Library/gurobi950/macos_universal2/lib"
              )
```

Once Gurobi has been manually installed, the project provides a script for installing other third party libraries. These include `Cmake`, `Eigen`, and `xcode` on a mac. With all the required dependencies fulfilled, we can build the StrokeStrip project using CMake, and start running experiments on our .scap files.

8.4.2 Running and Generating Files

The .scap inputs for the study are saved in the `examples/study_inputs` directory. To run the code, we use the strokestrip executable from the build folder using the following command: `build/strokestrip examples/study_inputs/filename.scap` where 'filename' is replaced by the name of the input file. Our script produces four

different .svg files as results, each representing different stages of the computation. The four types of result files generated are gradient orientation, parameterization, isolines, and the final fitted curve. We now discuss the purpose behind the generation of each of these four types of output images.

Gradient orientation We know that predicting the orientations of the gradients along each stroke is quite reliable. For this reason, the code first computes the optimal gradient orientations for all strokes in all clusters. The code generates files with ending '_orientation.svg' which reflects the gradient orientation in the input file.

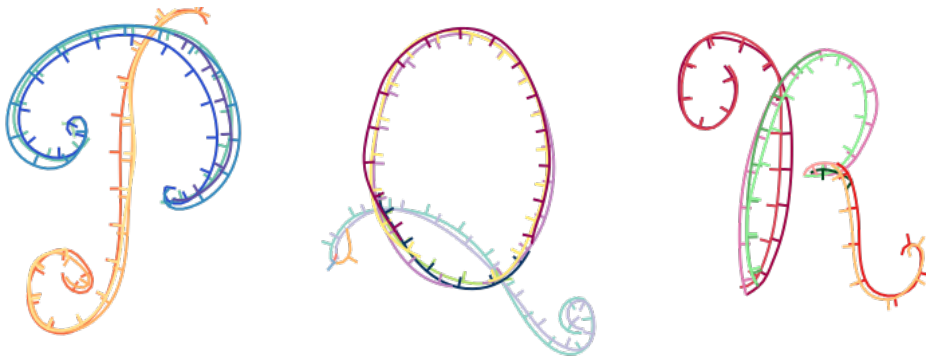


Figure 8.11: Example to show gradient orientation

Parameterization and Parameter Values The next step for the code is to compute both parameterization cross-sections and parameter values at the same time. The file generated for this step ends with '_params.svg'. An example is shown in Figure 8.12 where the input strokes are colored blue to red. The colors depend on parameter values, which in turn depend on the distance of each point on each stroke in the group. This means two opposite ends would have two different colors - red and blue.

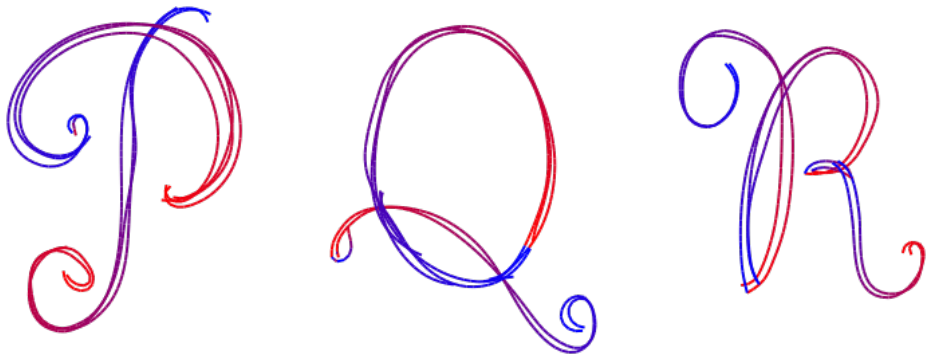


Figure 8.12: Example to show parameterization

Isolines After the parameterization step, the algorithm computer the arc length *cluster* parameterization by finding the cross-section isolines and isovalues [13]. As a final result of this step, we solve for the optimized cross-sections joining together points that are supposed to have similar isovalues. The isolines for within-the-strip adjacent points are then calculated and shown as an output similar to the ones shown in Figure 8.13.

Figure 8.13: Example to show isolines

Fitted Curve Using both the cross-sectional data and parameter values, the algorithm moves on to the final step - defining the parametric fitted curve. The fitted image generated ends in '_fit.svg' and its shape represents the geometric properties of the input curve that forms a clean line drawing. An example

is shown in Figure 8.14 where the parameterized drawing in Figure 8.12 has been fitted.

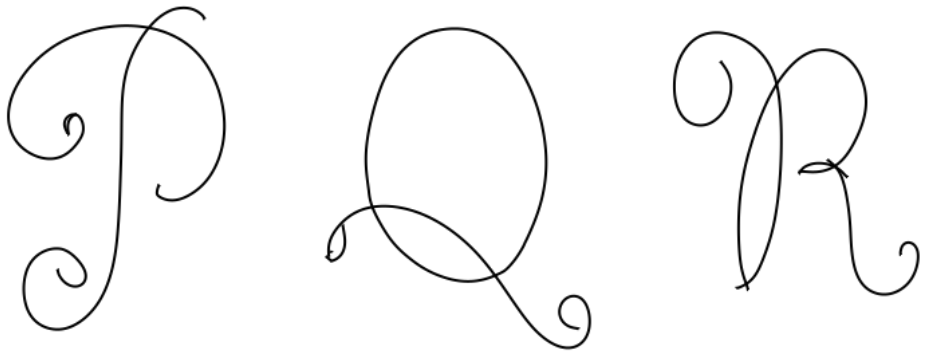


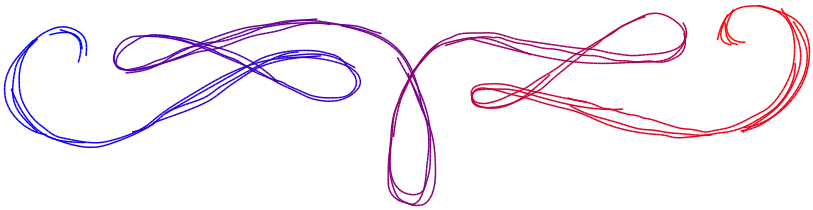
Figure 8.14: Example to show the final fitted result

CHAPTER 9

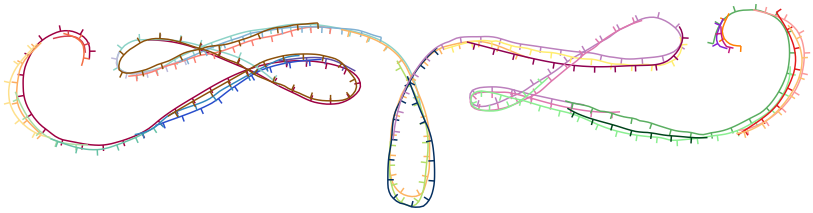
Results and Discussion

To study the results of the Strokestrip algorithm, we chose multiple inputs. These inputs include some of the .scap input files provided by the project itself. Additional inputs include our own sketches created through Adobe Illustrator. These inputs contain letters and words from different languages, and a few drawings for slightly more complex input.

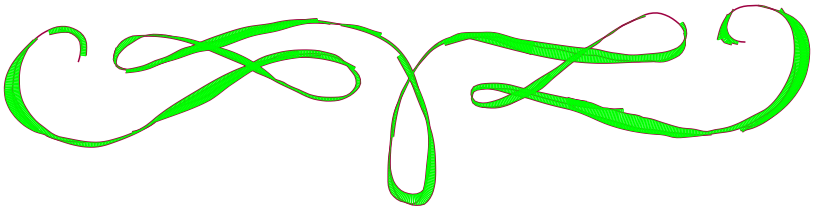
For every input, there will be four different types of output produced as mentioned in Section 8.4.2 and illustrated in Figure 9.1. For the purposes of this paper, we would mainly focus on the parameterization and final fit, as seen in Figure 9.1a and Figure 9.1d respectively. The input drawing is analogous to the parameterized picture. The only difference between the input and the parameterized output picture is the absence of shaded parameter values in the input. The two colors chosen for expressing the two ends of the stroke parameter values are red and blue. The shading ranges from blue at the start to purple and then red at the end, for every stroke being parameterized.



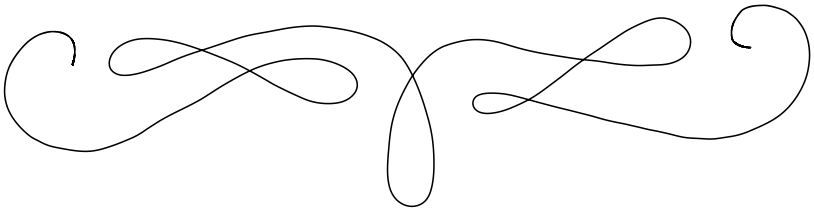
(a) Parameterized input



(b) Gradient Orientation



(c) Isolines



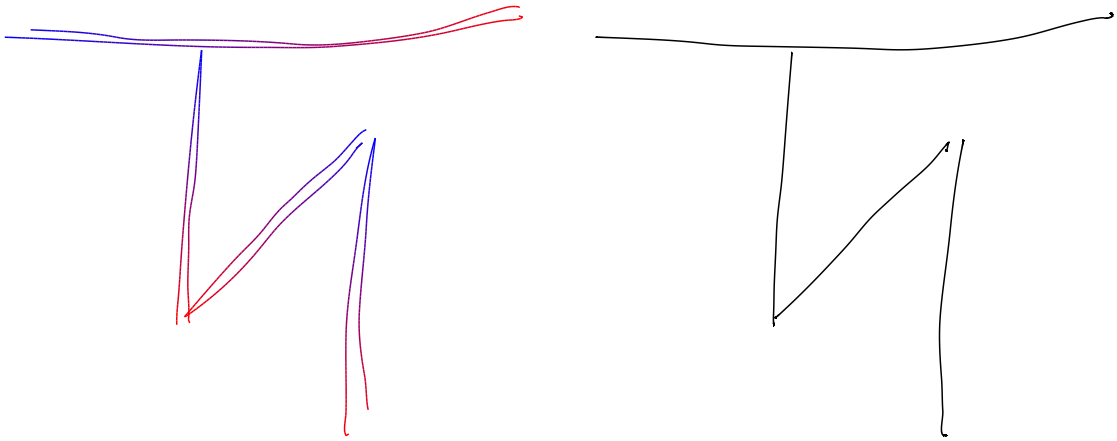
(d) Fitted output

Figure 9.1: Types of outputs produced

We recall from Chapter 7.2.1 that the StrokeStrip algorithm aims to satisfy the following properties during arc length parameterization: tangent alignment, arc length preservation, monotonicity, and isoline span. For our inputs, we tried to create drawings that face different challenges based on these requirements. The tangent alignment requirement can be challenged by multiple strokes that are adjacent but moving at slightly different angles. This can be illustrated by the example shown in Figure 9.2.

In Figure 9.2a, we observe that there multiple strokes seem to converge (or

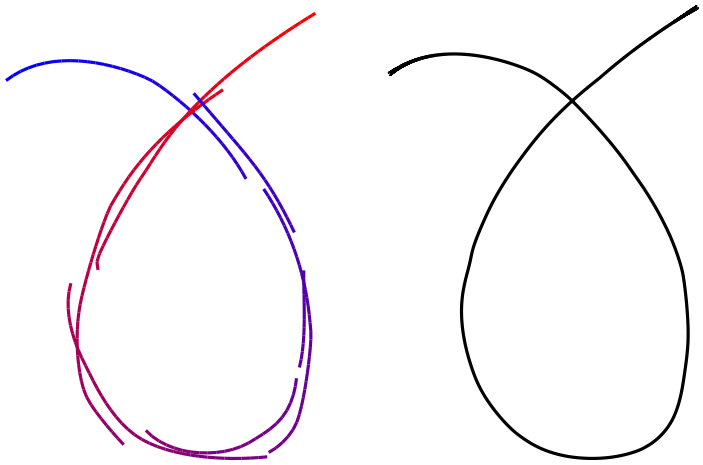
diverge) with respect to a point from different angles. But as they are within-the-strip adjacent, the tangents are aligned and parameterized accordingly. For such strokes, we see that the individual strokes forming the curve have very similar parameter values, as color coded in Figure 9.2a. In Figure 9.2b we see that the algorithm successfully fits the strokes in an aggregate curve representing the average geometric representation of the letter.



(a) Parameterized input **(b)** Fitted output

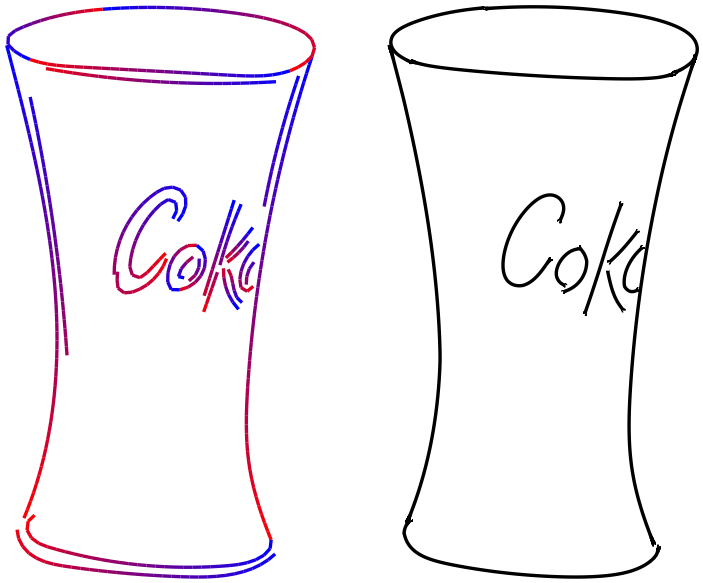
Figure 9.2: Fitted letter 'Do' from Bengali alphabets

These strokes could also form a loop instead of combining the two strokes that form an aggregate. In that case, the strokes would be considered a single stroke being parameterized from one end to the other. But instead, both the strokes have similar parameter colors side-by-side and the converging point is considered one of the ends. This satisfies both the monotonicity and isoline span properties. To test the same challenges on strokes actually crossing and turning back, we ran the StrokeStrip code on a drawing of a loop shown in Figure 9.3.



(a) Parameterized input **(b)** Fitted output

Figure 9.3: Fitted 'loop' drawing

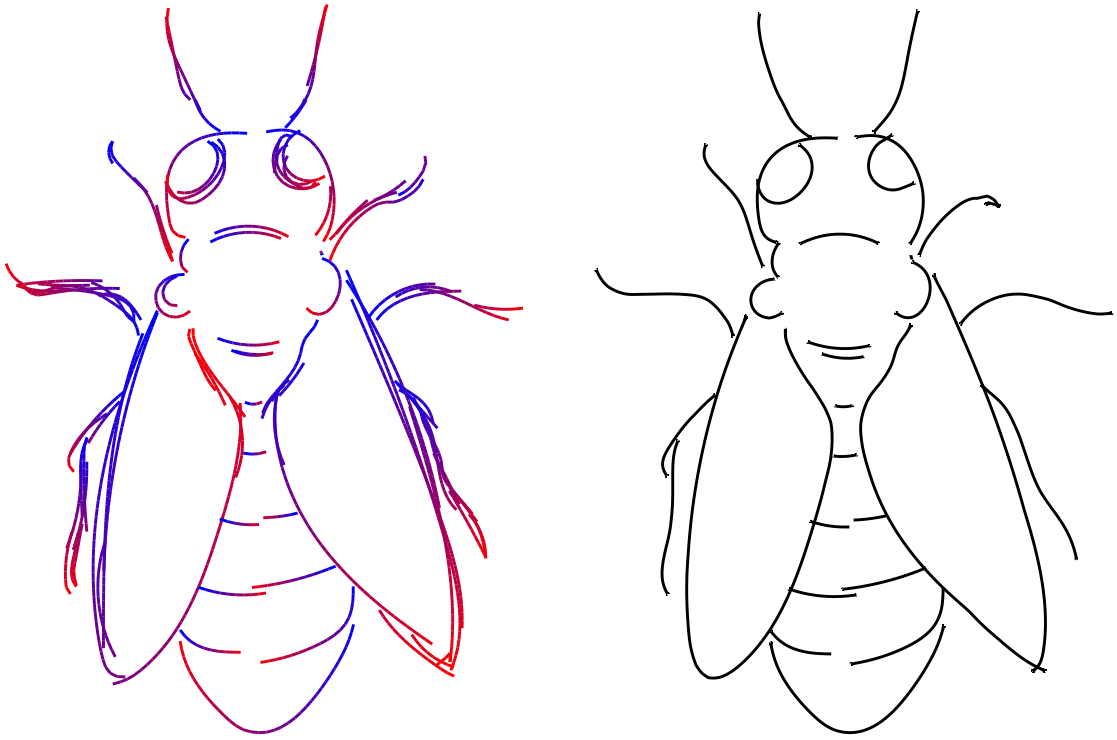


(a) Parameterized input **(b)** Fitted output

Figure 9.4: Fitted drawing of a glass with a label

As we can see in Figure 9.3a, the parameter values for side-by-side strokes are not the same (represented by different colors). This is because the algorithm used tangent alignment requirements and correctly detected the drawing as a loop going from one end to the other. Treating them as within the strip adjacent would also

violate the isoline span property. The fitted result in Figure 9.3b is a smooth curve, which has been formed from the parameterization and fit by preserving the arc length of the complete loop. Similar results are also shown in Figure 9.4 for a slightly more complex input. The two strokes on the bottom of Figure 9.4 seem to join together smoothly, while still retaining the overall intended shape. The fitted curve does not take the space in between the strokes, but rather fits it into the portion connected to the glass itself. This is a good example to illustrate how we prioritize curvature over position, as mentioned in Section 7.2.3.3.



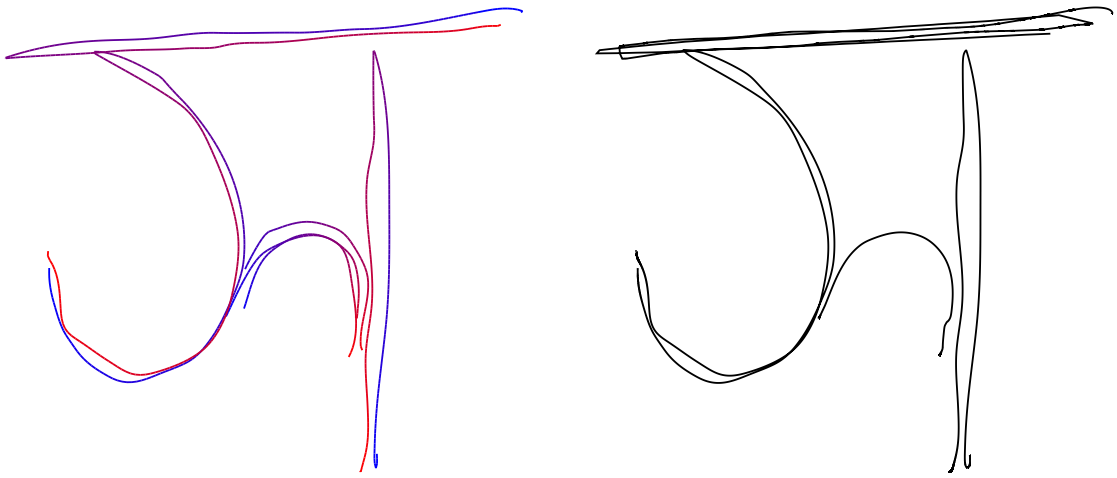
(a) Parameterized input **(b)** Fitted output

Figure 9.5: Fitted results for the drawing of an insect

Figure 9.5 shows the performance of the algorithm on a moderately complicated input. The fit on the wings and the central body of the bug are quite accurate to possible intended curves. The input on the top right 'leg' shows a different curvature of the input compared to the output. From Figure 9.5a, we see that there

is a small, disjoint stroke at the right edge that has been parameterized in a similar manner to the stroke underneath.

Using the labeler, we can enforce strokes to be in a specific group. In a similar manner, we can also take out strokes from groups into a separate group or just a standalone stroke. By overriding the usual expectations from strokes forming an aggregate curve, we can inspect the behavior of the algorithm in such situations. One such example is shown in Figure 9.6. In this example, vertical strokes on the right form an open loop, which connects the two strokes into a single stroke. This tells the code to treat it as a continuous, single stroke that does not get fit into an aggregate curve as seen in Figure 9.6b.



(a) Parameterized input **(b)** Fitted output

Figure 9.6: Fitted results for Bengali letter 'Sho'

From the parameterized input in Figure 9.6a, we can also see how the color changes from one end to the other without treating them as separate strokes. This fit does not satisfy our required isoline span expectations. Ideally, the whole stroke should be treated as two separate strokes due to its sharp turn, as suggested in Figure 8.9. In terms of the labeler, the stroke should be broken at the turning point,

into two separate strokes. Only then we can put these two strokes in the same group.

9.1 Special Case Study

To see how different labeling causes separate nuances in the fitted output, we move from a custom letter input to a custom word input. The drawing shown in Figure 9.7 is the word 'Riya' written in the Hindi language, but with some added extra strokes to convey the word more like a sketch. The extra added strokes also allow us to track how the algorithm deal with distinct stroke groupings.

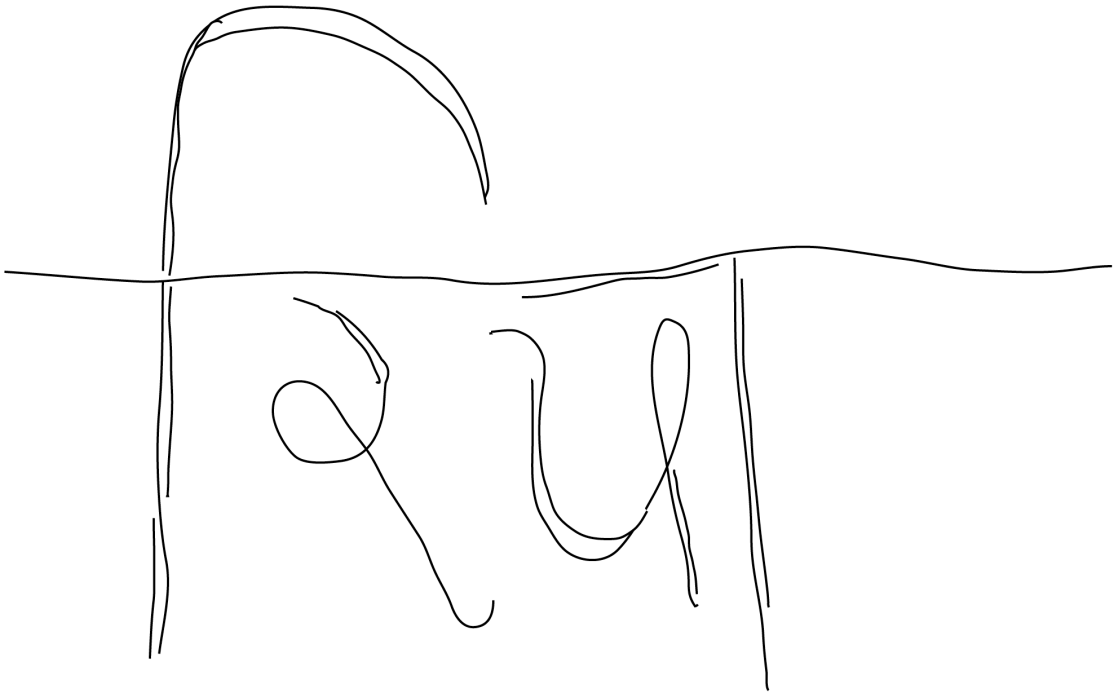
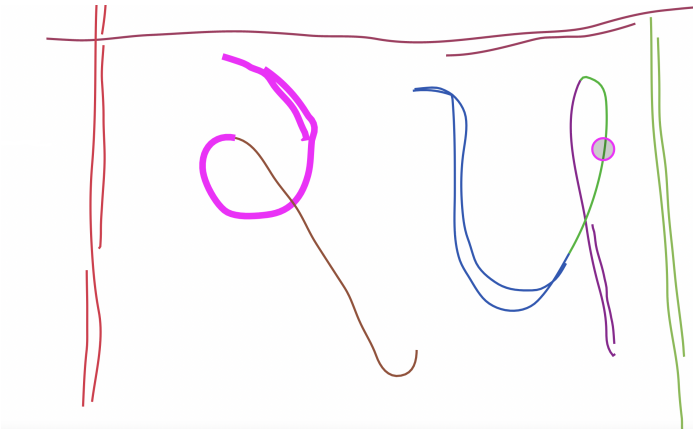
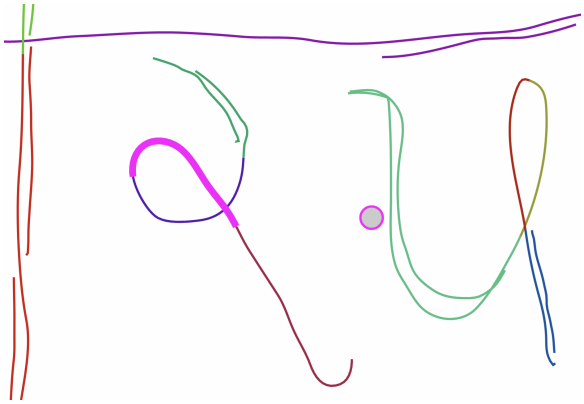


Figure 9.7: Raw sketch of the word 'Riya' in Hindi

As we can see in Figure 9.7, there are a few places where the extra strokes can be treated slightly differently in terms of grouping them. These different groupings for a specific letter (that spells 'R' in English) from the word are shown in Figure 9.8



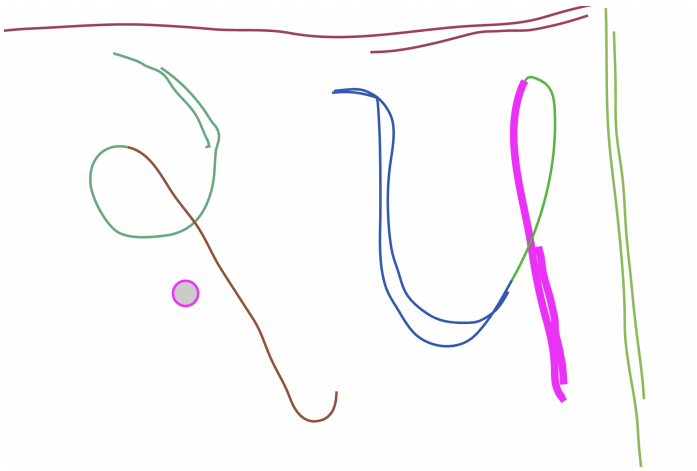
(a) Grouping A



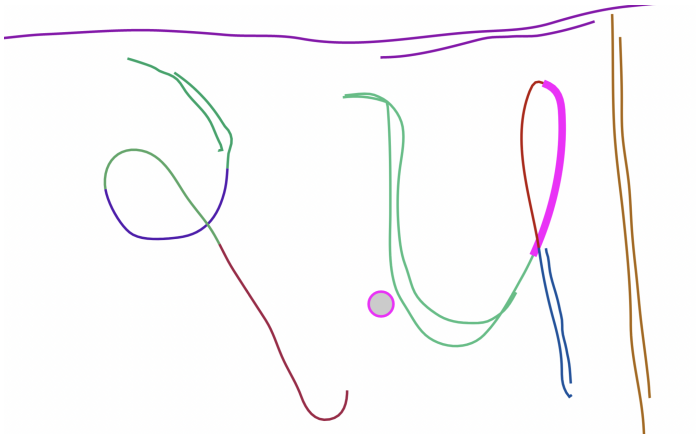
(b) Grouping B

Figure 9.8: Two possible groupings at Hindi letter 'R'

We observe that **group A** in Figure 9.8a contains only *two* strokes for this specific letter. One group is highlighted as pink, and the other group is the stroke that is colored brown. For **group B**, we divide the letter into *four* different groups shown in different colors. This is shown in Figure 9.8b, where we break the top two strokes in a single group, divide the loop into two groups, and keep the bottom end of the letter as a standalone stroke. In a similar fashion, we create two different stroke orientations for the other letter (that spells 'Y' in English) for both the groups. The comparative stroke grouping for this letter is shown in Figure 9.9



(a) Grouping A



(b) Grouping B

Figure 9.9: Two possible groupings at Hindi letter 'Y'

For group A, there are three groups including the highlighted group as shown in Figure 9.9a. In group B, there are four groups where the loop has been broken into two groups at the turning point. This signifies the different types of orientation that the StrokeStrip algorithm can deal with separately.

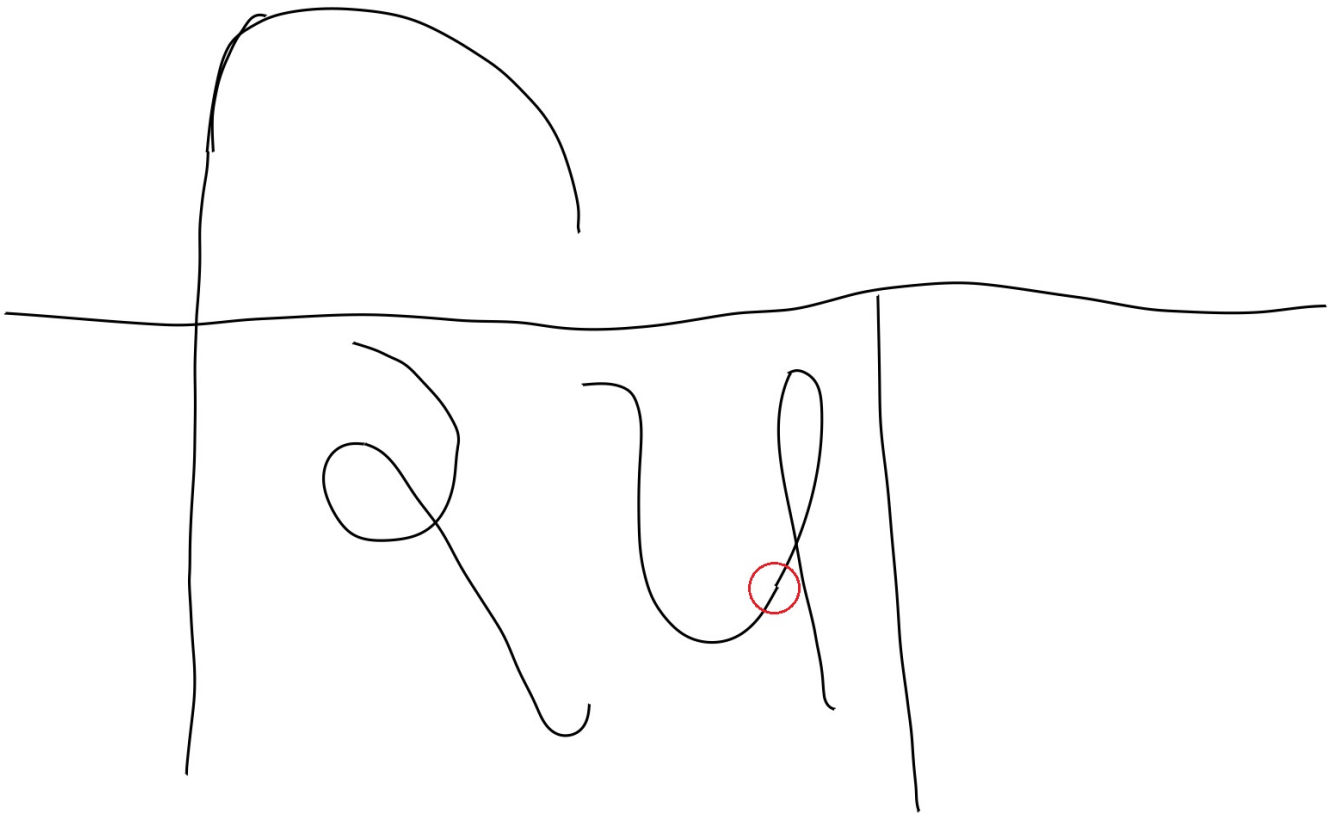


Figure 9.10: Fit for group A

The aim of this test was to see whether a further division of strokes into different types of shapes helped with the overall fit of the drawing. The final fit for both group A and group B is shown in Figure 9.10 and Figure 9.11 respectively. From Figure 9.10 we see that in group A the letter 'R' was fitted smoothly without any breaks. But for group B fit in Figure 9.11, there is a break on the top stroke fit, which does not keep the fit smooth. Similarly, group A fit for the letter 'Y' in Figure 9.10 has a small break at the point where the loop starts from the right. But the group B fit contains two breakpoints at the bottom of the loop in two different directions.

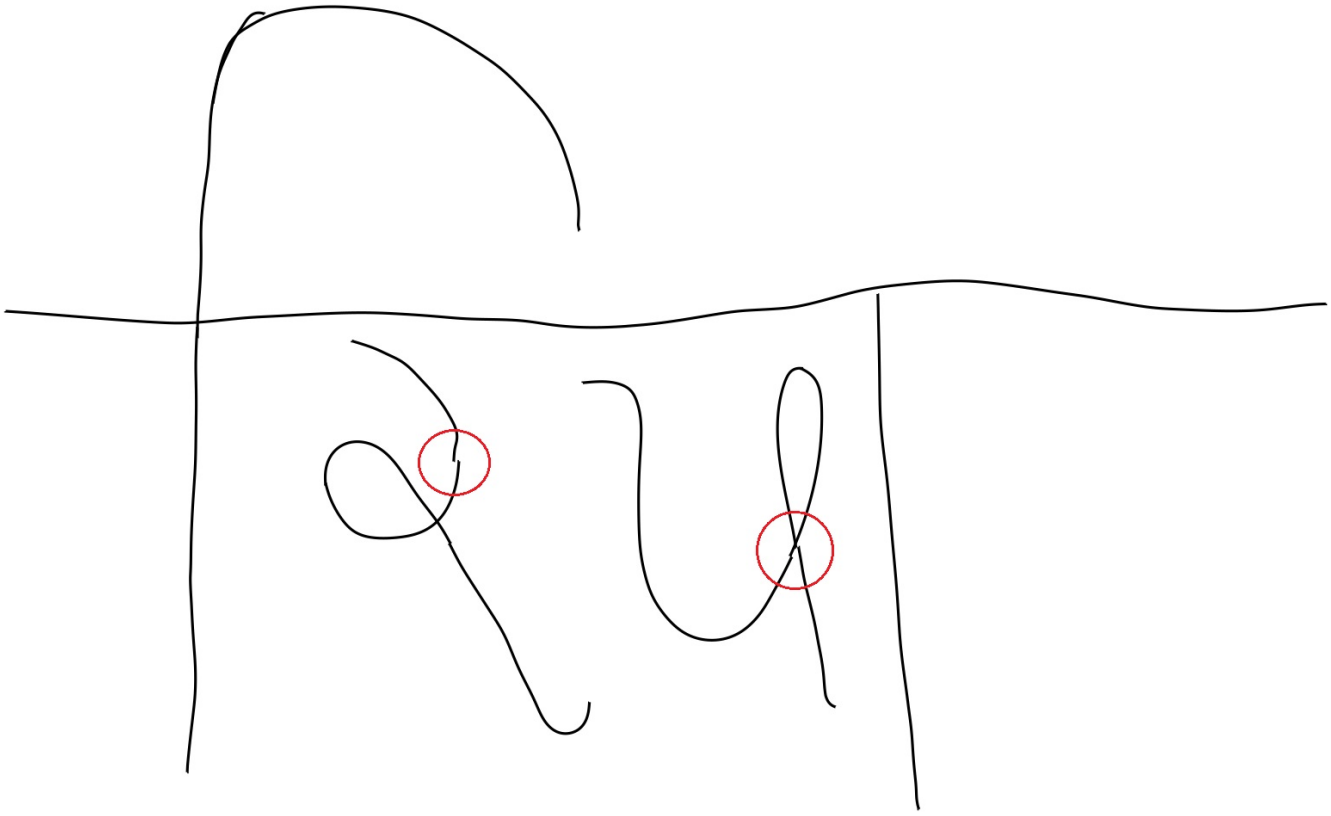


Figure 9.11: Fit for group B

Although both the fit for this letter are very close to the intended curve, the fit with a single breakpoint may be preferred over the one with two breakpoints residing close to each other. This illustrates that refining the strokes too much makes the algorithm fit them separately, which does not always guarantee the smooth, continuous curve that a user might expect. Grouping strokes that 'belong' to each other should be preferred rather than grouping strokes that individually 'should' produce the final aggregate curve.

CHAPTER 10

Conclusion

The main goal of this paper was to understand the basic principles involved in the automatic consolidation of freely drawn sketches. These basic principles involve steps such as point ordering, stroke clustering, stroke orientation, curve fitting, etc. Each of these steps performs certain tasks that allow the overall algorithm to systematically proceed towards the final result. There is quite a few underlying math involved in each of these steps, which we studied throughout this paper. The main subject of our study was the project *StrokeStrip*, which parameterizes clusters by computing the restriction of the strokes to their natural arclength parameterization [13]. This arc length parameterization is done by imagining stroke clusters as strips.

Besides numerous mathematical concepts, the project is heavily involved with artistic ideas. We saw how the software implementation was majorly dependent on the artist's insights on how humans perceive art. Specifically, the algorithm utilizes information on how humans visualize the overall intended art when there are multiple strokes being used to depict different curves and shapes. These help to make decisions such as finding adjacent strokes, differentiate between continuous and disjoint strokes, etc. We studied clustering and fitting steps that are generally the two main steps involved in such art consolidation frameworks. This helped us to view StrokeStrip in contrast to other methods, as StrokeStrip performs joint parameterization and fitting without any point ordering or clustering.

To explore the project in terms of performance, we the fitted output produced by StrokeStrip on multiple inputs. These inputs were both project-provided .scap files and our own sketch inputs, with varying levels of complexity. The outputs were studied and the performance on each image was recorded. The results were compared in accordance with the mathematical factors and assumptions involved in the joint parameterization process. This helped us understand the functionality and usage of the code, as well as some of its limitations on different types of output.

For future works, the software can make use of a graphical user interface that combines multiple steps of the whole process together. The .svg to .scap conversion can be a part of the GUI, allowing the user to then label the strokes from the newly generated .scap file. The fitting process can then use these labeled strokes to carry out parameterization and fitting to produce output, all through a simple interface. Additionally, the project can be extended beyond simply processing sketches. It can be used in conjunction with raster-to-vector conversion techniques to clean older images and manuscripts. In general, StrokeStrip would be very useful in applications where there are multiple paths involved as image, that requires to be fitted into curves. Finally, more research can be focused on the perceptual study of different factors that humans engage during the mental consolidation of sketches. The findings can then be used and applied to StrokeStrip's existing method to further refine the parameterization and consolidation process.

References

1. SVG files: How to create, edit and open them | adobe. URL https://www.adobe.com/creativecloud/file-types/image/vector/svg-file.html. 59

2. Gurobi optimizer. URL https://www.gurobi.com/products/gurobi-optimizer/. 68

3. Edward Angel and Dave Shreiner. *Interactive computer graphics: a top-down approach with WebGL*. Pearson, 7th edition edition. ISBN 978-0-13-357484-5. 11, 12, 13, 14, 18, 19, 20, 28, 31

4. Emily Brewer. Vector vs. raster. URL https://thinkeps.com/2020/08/11/vector-vs-raster/. 4

5. Ron Goldman. CHAPTER 1 - introduction: Foundations. In Ron Goldman, editor, *Pyramid Algorithms*, The Morgan Kaufmann Series in Computer Graphics, pages 1–43. Morgan Kaufmann. ISBN 978-1-55860-354-7. doi: 10.1016/B978-155860354-7/50002-7. URL https://www.sciencedirect.com/science/article/pii/B9781558603547500027. 14

6. Eric Lengyel. *Mathematics for 3D game programming and computer graphics*. Game development series. Charles River Media, 2nd ed., [repr.] edition. ISBN 978-1-58450-277-7. 17, 20, 21, 22, 23, 24, 25, 26, 28

7. Chenxi Liu, Enrique Rosales, and Alla Sheffer. StrokeAggregator: consolidating raw sketches into artist-intended curve drawings. *ACM Transactions on Graphics*, 37(4):97:1–97:15, July 2018. ISSN 0730-0301. doi: 10.1145/3197517.3201314. URL https://doi.org/10.1145/3197517.3201314. 2, 33, 37, 38, 39, 41, 43, 58

8. OpenStax, BCcampus, OpenStax College, and University of Minnesota. *Calculus Volume 3*. OpenStax College, Houston, 2016. URL http://VH7QX3XE2P.search.serialssolutions.com/?V=1.0&L=VH7QX3XE2P&S=AC_T_B&C=Calculus%20Volume%203&T=marc&tab=BOOKS. OCLC: 1066560292. 8, 10

9. Gunay Orbay and Levent Burak Kara. Beautification of Design Sketches Using Trainable Stroke Clustering and Curve Fitting. *IEEE Transactions on Visualization and Computer Graphics*, 17(5):694–708, May 2011. ISSN 1941-0506. doi: 10.1109/TVCG.2010.105. Conference Name: IEEE Transactions on Visualization and Computer Graphics. 2, 43, 44, 45, 46

10. Dave Pagurek. Labeller. URL https://github.com/davepagurek/StrokeAggregatorLabeller. original-date: 2019-10-08T22:03:12Z. 60

11. D. Salomon. *Curves and surfaces for computer graphics*. Springer. ISBN 978-0-387-24196-8 978-0-387-28452-1. 16, 19, 20, 24, 28

12. James Stewart. *Multivariable calculus*. Brooks/Cole Cengage Learning, Belmont, CA, 7th ed edition, 2012. ISBN 978-0-538-49787-9. 6, 7, 8

13. Dave Pagurek Van Mossel, Chenxi Liu, Nicholas Vining, Mikhail Bessmeltsev, and Alla Sheffer. StrokeStrip: joint parameterization and fitting of stroke clusters. *ACM Transactions on Graphics*, 40(4):50:1–50:18, July 2021. ISSN 0730-0301. doi: 10.1145/3450626.3459777. URL https://doi.org/10.1145/3450626.3459777. 1, 2, 5, 43, 47, 49, 50, 51, 53, 54, 55, 67, 71, 85

14. Johan Wagemans, James H. Elder, Michael Kubovy, Stephen E. Palmer, Mary A. Peterson, Manish Singh, and Rüdiger von der Heydt. A century of gestalt psychology in visual perception: I. perceptual grouping and figure–ground organization. 138(6):1172–1217. ISSN 1939-1455, 0033-2909. doi: 10.1037/a0029333. URL http://doi.apa.org/getdoi.cfm?doi=10.1037/a0029333. 33, 34, 39