CPSC589 Final Project

Image Based Font Generator

Modan Han

April 2017

Contents

Chapter 1

Abstract

The process of font creation, especially digital typography is time consuming, the Image Based Font Generator aims to speed up the process.

The Image Based Font Generator explores ways to convert traditional typography digitally, provide easier and faster ways to create fonts, and enable the ability to create fonts for people with no digital typography experience.

The user only needs to provide enough samples of the font they would like to generate in an image format, the Image Based Font Generator will then generate a set of B-Splines for the character. The font provided by the user can very well be a scanned image of the user's handwriting, or from any other source, including existing fonts.

Chapter 2

Overview

I implemented the Image Based Font Generator, which can generate a set of quadratic B-Spline curves representing the glyph of a character, almost completely automatically, requiring only an image as input and very few parameters.

The input image first goes through several stages of processing until the image data is integrated to serve as target data points for the generated curve. Several input parameters are available at this point allowing for more accurate processing and style customization.

The main curve generation technique uses reverse Chaikin Subdivision, outputting a small number of control points relative to the image, and smoothing out noise the image may contain.

Finally the curve can be manually edited and output to a file, and can be re-opened for more editing later.

Since the curves are quadratic B-Splines, which is used by modern typeface standards, these curves can be converted into glyph datas for these typeface standards.

Chapter 3

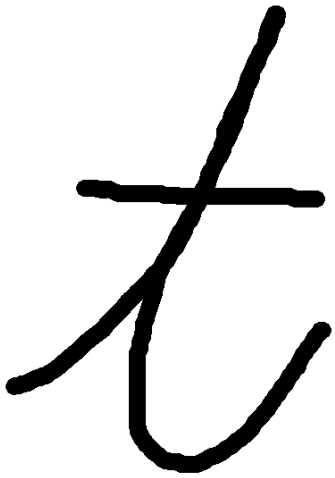
Implementatoin Details

3.1 Image Processing

The user needs only to provide an image of size 1024×1024 pixels, of any valid image format, e.g. .jpg, .png.

See below for an example of an image of the character t created in GIMP.

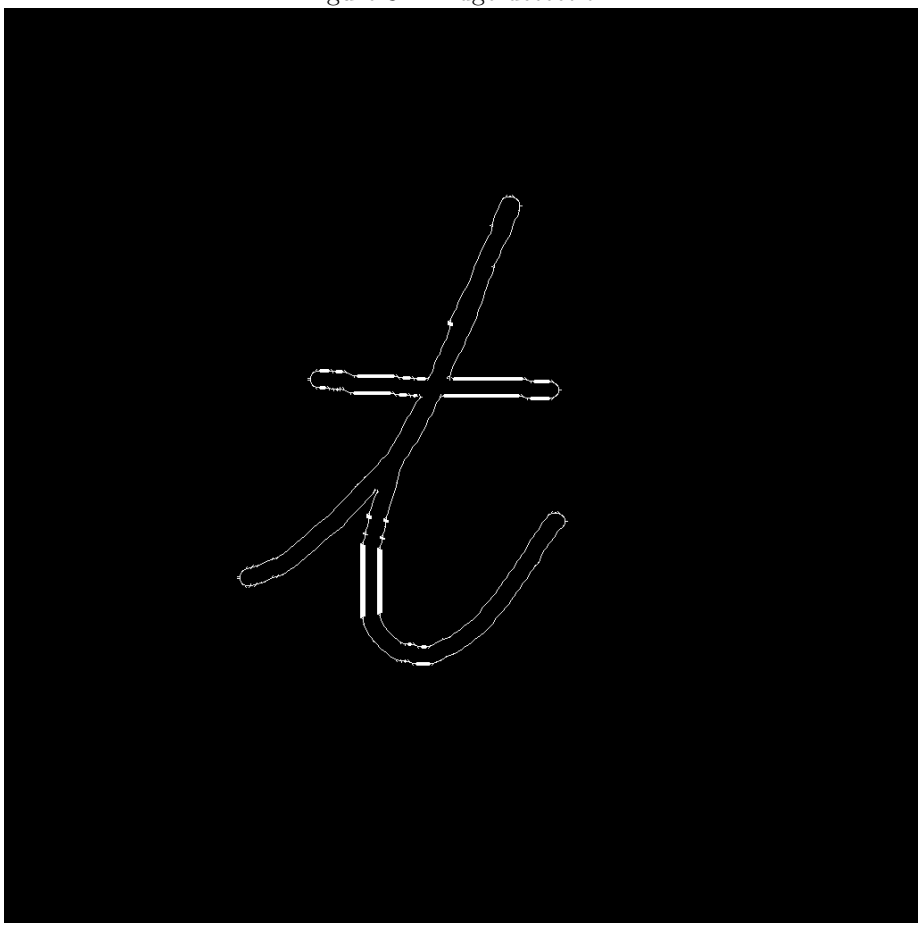
Figure 3.1: Example image of the character t.



3.1.1 Edge Detection

The first image processing pass runs an edge detection on the image. This pass of image processing uses Canny Edge Detector, and is implemented with a multi-directional Sobel filter, non max suppression and double threshold.

Figure 3.2: Edge detection.



The user can also adjust the edge detection threshold using the scroll wheel to get a more accurate result. As too much threshold results in some edges not being detected, too little threshold results in noisy edges, although usually noise is well dealt with in later stages of processing where noise reduction takes place.

Figure 3.3: Edge detection with adjusted threshold.

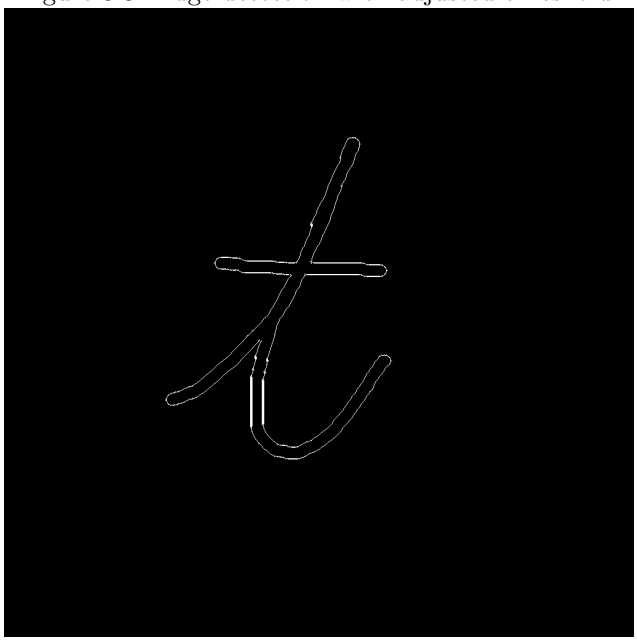
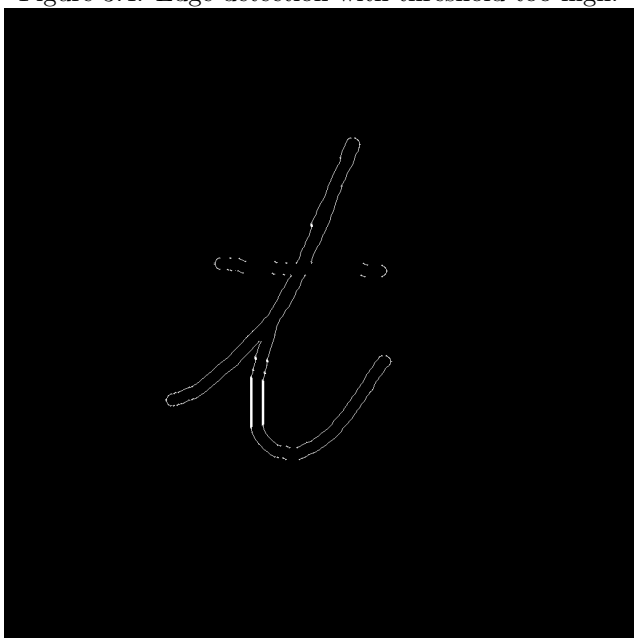


Figure 3.4: Edge detection with threshold too high.



3.1.2 Corner Detection

This is the second image processing run on the image and uses the Harris Corner Detector. The implementation uses the weighted sum of squared differences formula,

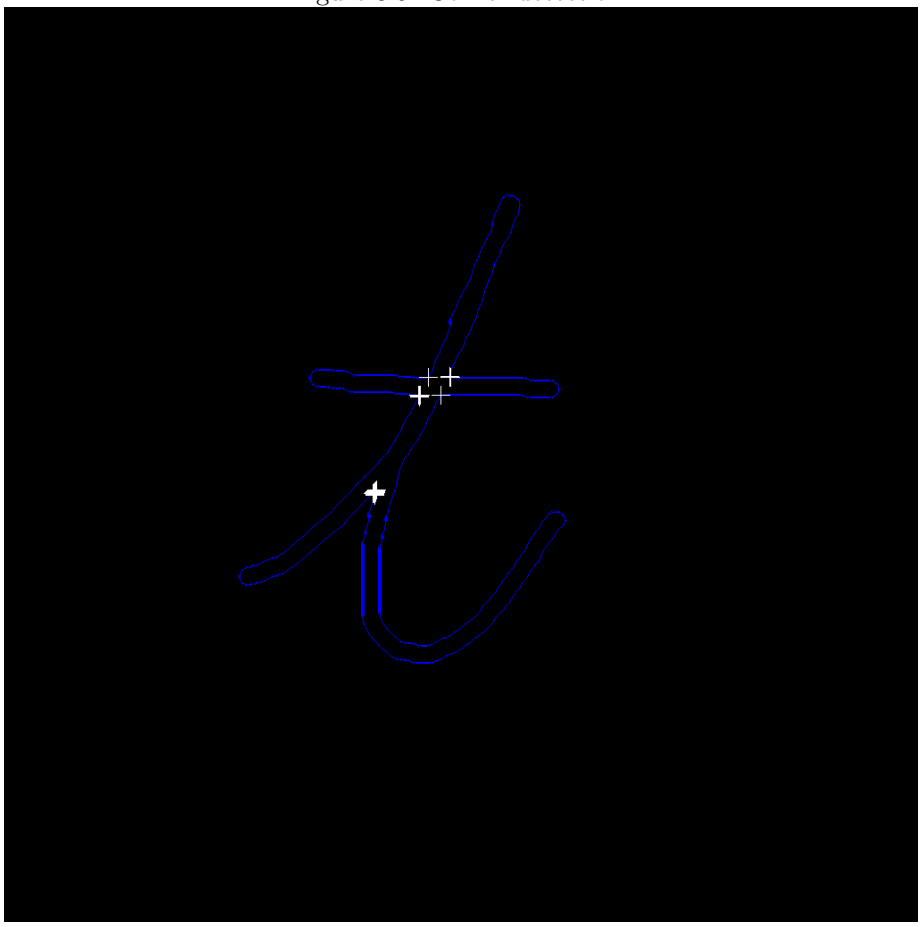
$$S(x,y) = \sum_u \sum_v w(u,v)(Image(u+x, v+y) - Image(u,v))^2$$

where

$$w(x) = G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

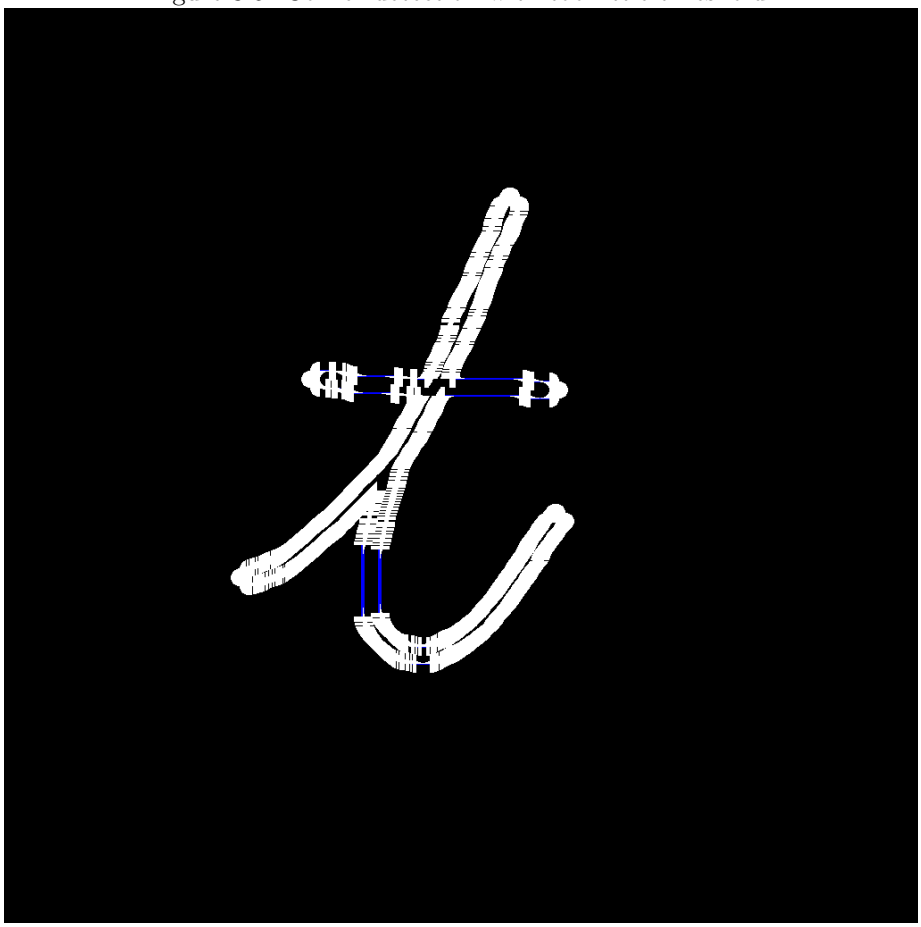
is the Gaussian function.

Figure 3.5: Corner detection.



Similar to edge detection, the threshold can be adjusted to achieve desirable results.

Figure 3.6: Corner detection with too little threshold.



3.2 Data Point Processing

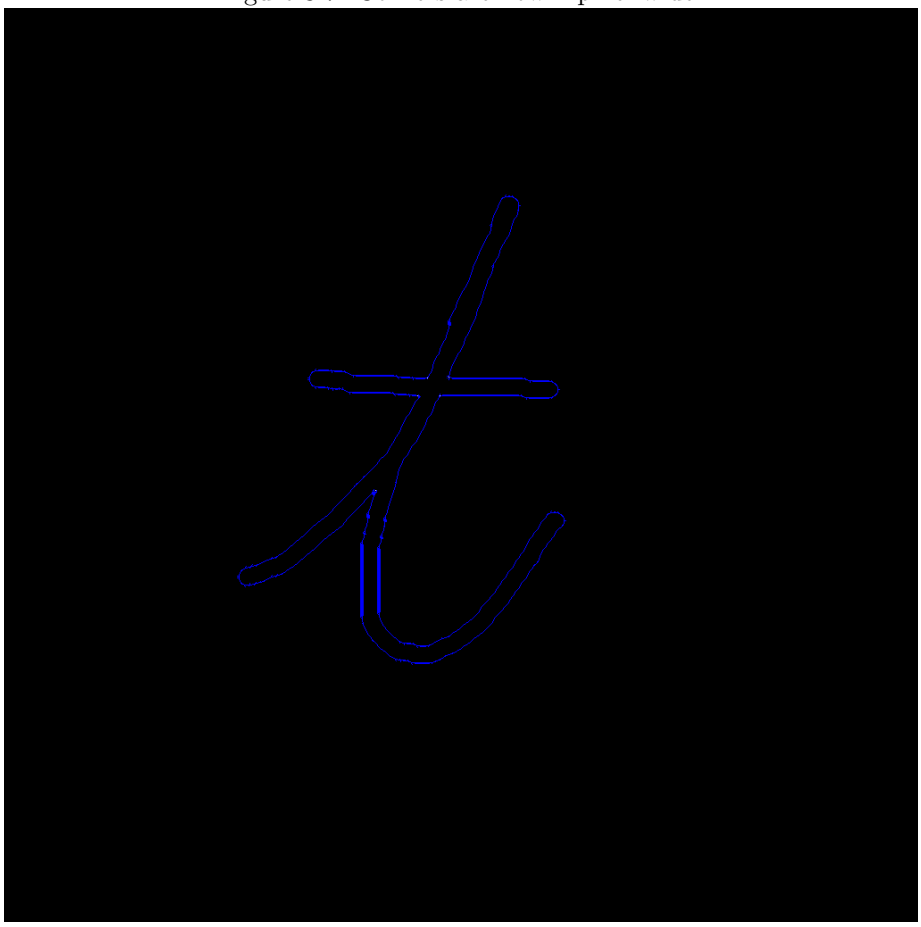
At this stage, many adjoint corners maybe detected, edges may not be connected to corners, edge noises maybe present and edges maybe not be perfectly 1 pixel wide.

The next few processing stages aim to solve these issues.

3.2.1 Corner Merging

A single pass of breadth first search is run on the image to merge all corners into one single pixel, giving constant runtime.

Figure 3.7: Corners are now 1 pixel wide.

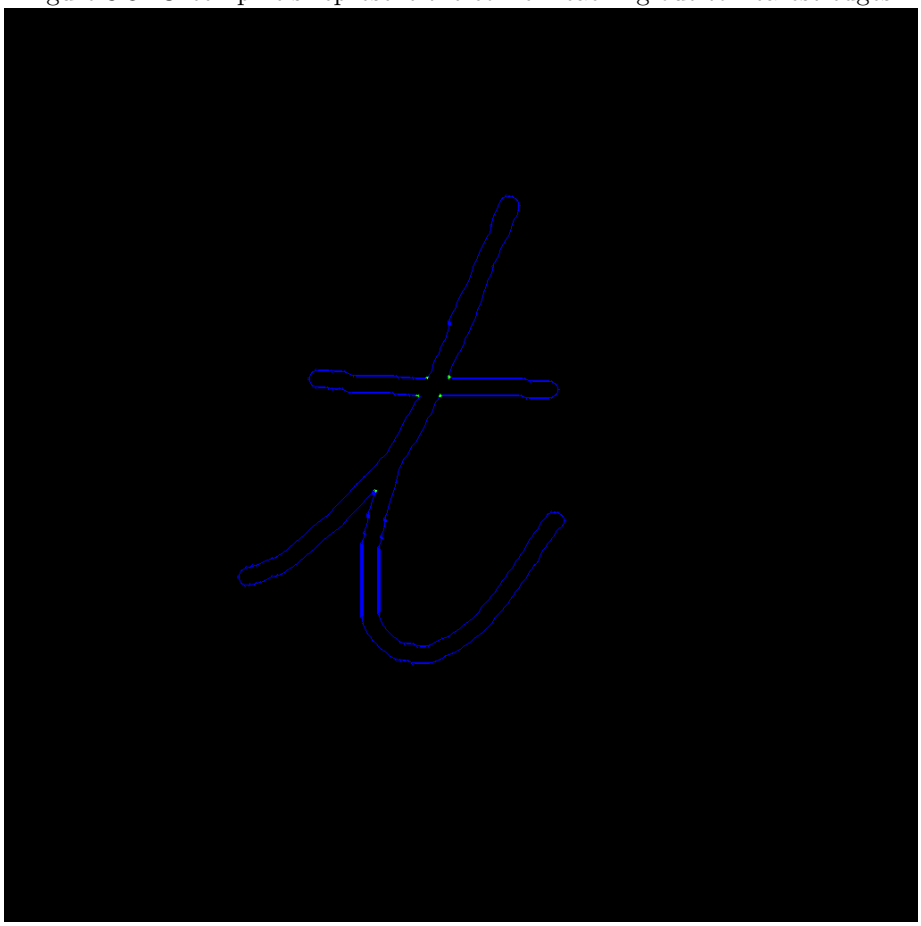


3.2.2 Corner Connection

A limited distance breadth first search is run on each corner to find its nearest non-connected edge, then makes the shortest path to connect to that edge.

The max distance can be adjusted to reach for more distance edges, however the error distance is expected to be very short, therefore in time linear to the number of corners, i.e. constant time in the size of the image, the program makes all the connections for all the corners.

Figure 3.8: Green pixels represent the corner reaching out to nearest edges.

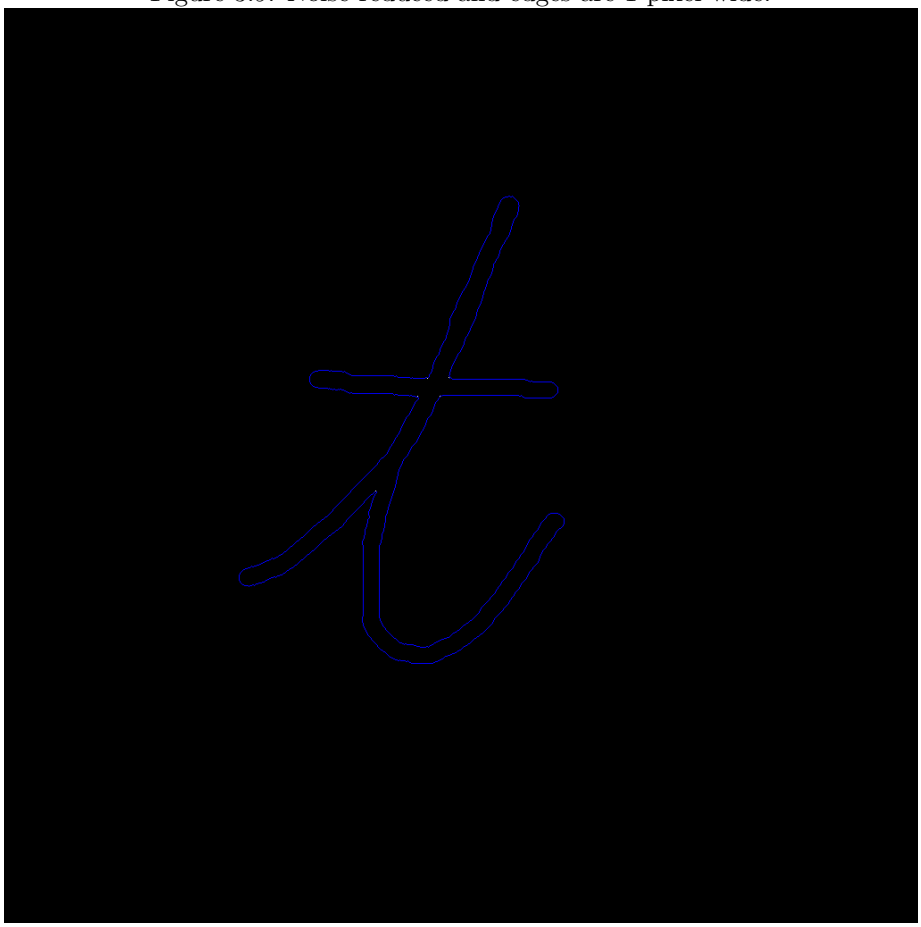


3.2.3 Noise Reduction

For each corner, an edge connected to it is found, then the program traverses through that edges until it finds the next corner. Any pixel not visited by the traversals from all the corners are removed, further reducing noise and guarantees that all edges are 1 pixel wide.

The traversals also give the orders of target data points for curve generation.

Figure 3.9: Noise reduced and edges are 1 pixel wide.

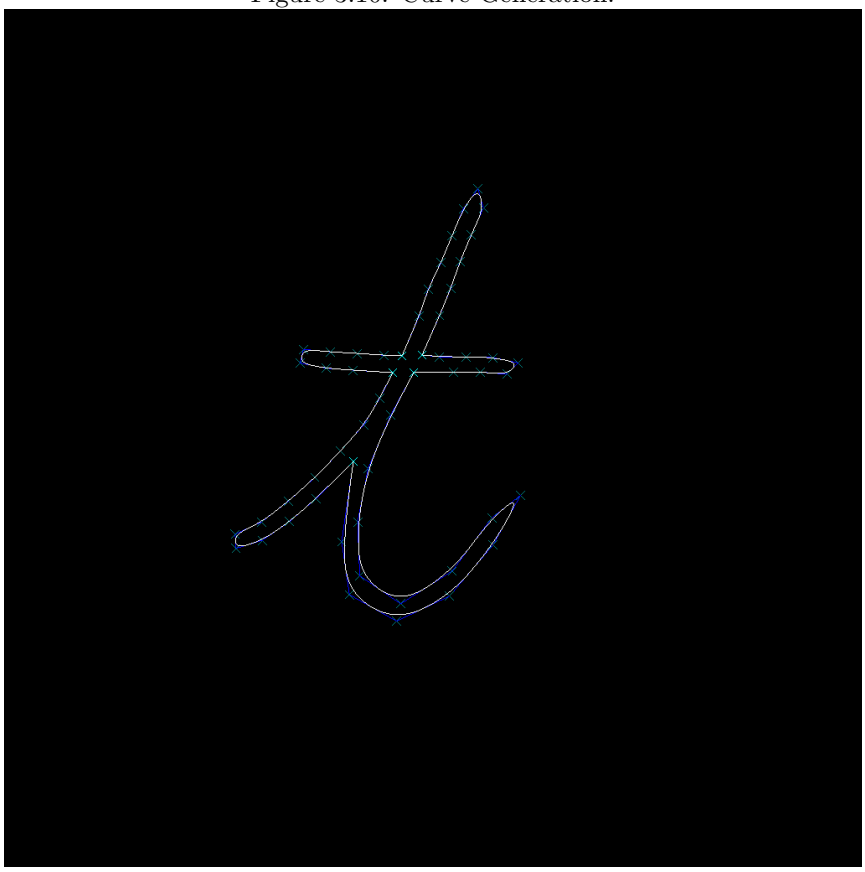


3.3 Curve Generation

Curve generation uses reverse Chaikin Subdivision with edge cases. For general cases, the program considers every second point the "generated" point by Chaikin Subdivision, and uses it to deduce the first point's position before subdivision. This results in pairs of points being very close to each other (their positions would be identical if the target data points were really generated from Chaikin Subdivision), and their average is considered a control point before subdivision.

By this algorithm every 2 target data points is merged into 1; this process is repeated for several times until a good number of control points is achieved. The number of control points is determined by the length of the target curve, i.e. the number of target points in it, and user preference.

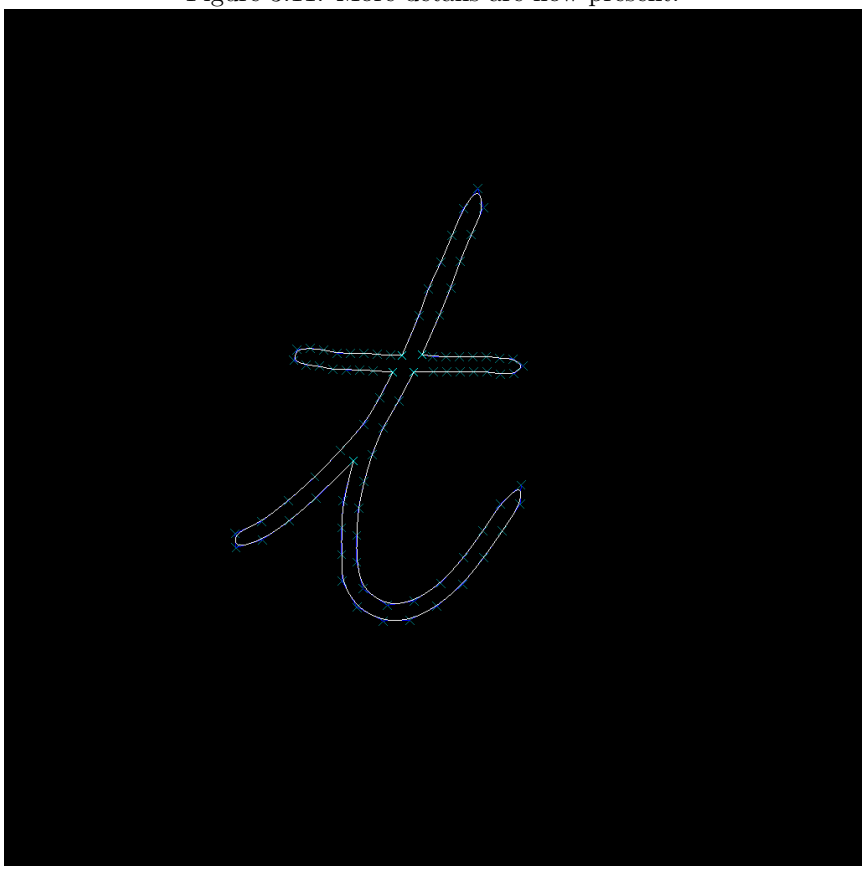
Figure 3.10: Curve Generation.



3.3.1 Details Adjustment

The amount of reverse subdivision can be adjust to include to exclude details from the target data points.

Figure 3.11: More details are now present.



3.3.2 Manual Editing

Finally the control points can be manually edited and deleted to achieve space efficient and desirable results.

Figure 3.12: Manually edited final result.

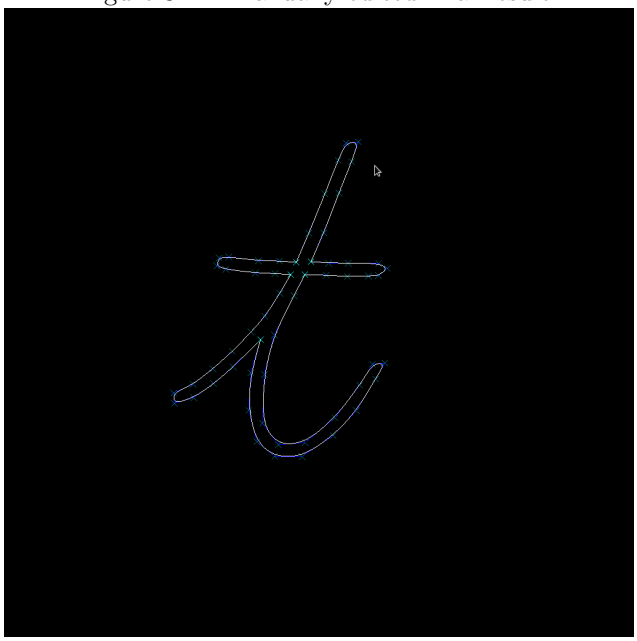
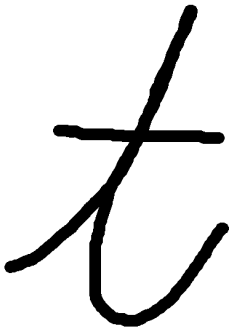


Figure 3.13: Comparison with the original image.



Chapter 4

Future Work

4.1 TrueType Font

The glyphs are saved as raw B-Spline curves data. An useful extension to the program would be to include functionality to group up glyphs, and interfaces to manage them so they can be output to a TrueType Font file.

4.2 Multi-Glyph Recognition

The program can detect multiple disconnected glyphs but will only output them as one set of B-Spline curves. An useful functionality would be to be able to separate glyphs by connectivity, and/or output the list of curves in a specific order such that they can be separated easily, e.g. the first $1, \ldots, n$ curves are of glyph g_1, and the next $n+1, \ldots, m$ curves are of glyph g_2, etc.

Chapter 5

Conclusion

The Image Based Font Generator achieves not perfect, but reasonable results; the generated glyphs are clearly recognizable from the input image, provided additional input parameters are valid.

Performance wise The Image Based Font Generator computes each processing stage efficiently, utilizing both efficient algorithms and shaders.

The Image Based Font Generator has a minimal user interface, with the intent to perform actions as automatically as possible.

Even though the functionality is somewhat lacking, I am happy about The Image Based Font Generator's accuracy, performance and simplicity.

References

[1] Faramarz Famil Samavati, Nezam Mahdavi Amiriy *A Filtered Bspline Model of Scanned Digital Images*, University of Calgary, 1999.