FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Automatic Gradient-Preserving Stencilization of Raster Images

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

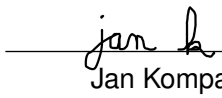
Medieninformatik und Visual Computing

eingereicht von

Jan Kompatscher
Matrikelnummer 01527584

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: : Projektass. Hsiang-Yun Wu, PhD

Wien, 8. Februar 2021



Jan Kompatscher

Hsiang-Yun Wu

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Automatic Gradient-Preserving Stencilization of Raster Images

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Jan Kompatscher
Registration Number 01527584

to the Faculty of Informatics
at the TU Wien
Advisor: : Projektass. Hsiang-Yun Wu, PhD

Vienna, 8th February, 2021

Jan Kompatscher

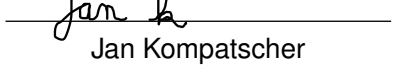
Hsiang-Yun Wu

Erklärung zur Verfassung der Arbeit

Jan Kompatscher
Völs am Schlern

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 8. Februar 2021



Jan Kompatscher

Danksagung

Zum Ersten möchte ich meiner Betreuerin Hsiang-Yun Wu danken, die mich den ganzen Weg vom Finden des Themas meiner Arbeit bis hin zu ihrem Abschluss begleitet hat. Sie hatte viele der Ideen, die wichtig für das Realisieren meiner Arbeit waren und wies mich immer darauf hin, wenn ich den falschen Lösungsansatz hatte. Durch ihre freundliche Umgangsweise waren unser wöchentlichen Meetings sehr angenehm und ich denke nicht, dass ich diese Arbeit ohne ihre Hilfe zustande gebracht hätte.

Zum Zweiten möchte ich meinen Eltern und meiner Familie danken, dass sie mich auf so viele Arten unterstützt haben und mein Studium überhaupt erst ermöglicht haben.

First of all, I would like to thank my supervisor Hsiang-Yun Wu, who accompanied me from finding the topic of my thesis to completing it. She had many of the ideas that were important in making my work a reality and she always pointed out to me when I came up with the wrong approach. Due to her friendly manner, our weekly meetings were very pleasant and I don't think that I would have been able to complete this work without her help.

Secondly, I would like to thank my parents and family for supporting me in so many ways and for making my education possible in the first place.

Kurzfassung

Schablonen sind Zwischenobjekte mit entworfenen Lücken, durch die Muster auf Oberflächen erzeugt werden, indem Farbe durch die Schablone aufgetragen wird, wobei es die Lücken der Farbe ermöglichen, auf die Oberfläche dahinter zu kommen und dadurch das gewünschte Muster auf der Oberfläche zu erzeugen. Für die Herstellung von Schablonen aus einem beliebigen Binärbild reicht es nicht aus, die Hintergrundfarbe als die Teile des Materials, die ausgeschnitten werden sollen, und die andere Farbe als das in der Schablone verbleibende Material anzunehmen. Es muss ein Zusammenhalt zwischen allen unabhängigen Teilen bestehen, die vom Material verbleiben, damit sie nicht einzeln an Ort und Stelle gehalten werden müssen. Die daher benötigten Verbindungen zwischen den Bauteilen könnten deutlich abgehoben und leicht von den beabsichtigten Formen unterscheidbar gemacht werden, um sie später mit einem Pinsel zu überzeichnen. Das Ziel dieser Arbeit wird hingegen ein Algorithmus sein, der Verbindungen zwischen den Formen herstellt, die im Bild verbleiben können, das die Schablone erzeugt, ohne das Erscheinungsbild der vorhandenen Formen (zu stark) zu stören. Dazu werden die Richtungen der Formkonturen auf einer vektorisierten Version des Originalbilds ermittelt, um mit den Verbindungen zwischen verschiedenen Formen in dieselben Richtungen fortfahren zu können. Dann werden aus allen möglichen Verbindungen diejenigen gefunden, die verwendet werden, indem eine Graph-Datenstruktur erstellt und ein maximales Matching dieses Graphs gefunden wird. Am Ende wird es möglich sein, ein Binärbild einzugeben und eine zusammenhängende Schablonenform zurückzugewinnen, die unverändert verwendet werden kann.

Abstract

Stencils are used as intermediate objects with designed gaps in them, to create patterns on surfaces by applying pigments on the surface through the stencil, which allows the pigment to reach the surface through the gaps and thereby to create the pattern on the surface. For the production of a stencil out of any raster image, it is not enough to assume the background color as the parts of the material that will be cut out and the other color as the material remaining in the stencil. There has to be cohesion between all the independent parts that are left in so that they do not have to be held in place individually. The needed connections between the components could be made very obvious and easy to distinguish from the intended shapes in order to draw over them later on with a paintbrush. The goal of this work however, will be an algorithm that produces connections between the shapes that can be left in the image the stencil produces, without disturbing the appearance of the shapes present (too much). This is done by finding the directions of the shapes' contours on a vectorized version of the original image, to be able to continue in the same direction with the connections between different shapes. Then from all the possible connections the ones that will be used are found by creating a graph data structure and finding a maximum matching of that graph. In the end, it will be possible to input a binary image and get back a continuous stencil form that can be used as-is.

Contents

CHAPTER 1

Introduction

1.1 Background

Stencils are a widely used artistic tool. In graffiti and street-art they are used to paint wanted shapes onto walls and floors for example. Also, they can be used to project shadows onto a surface by shining light through them (so-called light painting stencils). If some artist is designing a certain shape without specifically wanting to create a stencil, the physical requirements of a stencil (connectedness of the whole piece) could not be met. Creating a stencil out of some piece of art afterward is not a particularly creative task. In fact, with a predefined set of rules, this task can be automated. The challenge is to automate the creation of stencils but to make the needed connections "fit" into the original image. Meaning, they follow the existing curvature present in the image when possible.

1.2 Goal

The goal is to produce stencils from any raster image by vectorizing it and trying to connect unconnected components, without gravely altering the appearance with bridges that do not follow the curvatures of the original image.

1.3 Contribution

The contribution of this thesis is an algorithmic pipeline that automatically makes a stencil out of a raster image (before it only has been done for meshes or in direct cooperation with a human designer). The way the connections that connect the yet unconnected components for the stencil are found (by following existing gradients in the image) is also new.

1.4 Approach

The white part of our images will be the parts that will be left of the material (e.g. a piece of paper) the stencil will be made of and the black parts will be the parts that will be cut out from the material when producing the stencil. Therefore when the stencil is used with a spray-can those black parts will be the colored parts of the wall.

The approach will be to determine the curvatures in the image by vectorizing the image and to continue those curves in order to connect the unconnected components within the image.

The first step is to identify all the shapes (connected components) present in the raster image. Then all those shapes are vectorized independently from one another. With these vector images, the points where the connections of the components will be made are determined. The data structure of a graph, where the determined points are the vertices, will help us. Between all pairs of vertices that can have a connection, an edge will be added to the graph. With the *cardinality-matching algorithm*, a matching of edges that connects the most points is found. The edges of this matching will be drawn. The last step of the process is to draw those connections with a predetermined thickness.

1.5 Structure

The structure of this thesis is as follows:

1. **Chapter 2**: overview over papers that tackle similar/related problems.
2. **Chapter 3**: explains the steps of the image processing pipeline to get the result.
3. **Chapter 4**: shows the implementation details (including platform, libraries,programs used and the realization of the algorithms in more detail).
4. **Chapter 5**: talks about representative examples and what the results for them look like.
5. **Chapter 6**: ends the thesis with a conclusion and possibilities for future work on the problem.

CHAPTER 2

Related Work

2.1 Chinese Paper-cuts

Chinese Paper-Cutting is an ancient folk art coming from China where a single piece of paper is made into an artwork by cutting. These *papercuts* however are not used to apply paint to a surface like stencils but the (in most cases red) paper is hung up for example in windows or doors. Gut they do have the physical requirement of stencils, meaning that the whole piece of paper (or other material) needs to be connected.

Large databases of Chinese Paper-cuts were created to preserve this tradition for the future, as it is being done by fewer and fewer people. For example, there is a database of the most basic element symbols used in Chinese Paper-cutting.[PSP07] With the help of such databases science tries to preserve this craft for future generations.

For example tools aiding the production of such art are made. Liu and colleagues [LCW+18] created a tool that uses the most basic element symbols there are and other often used symbols in Chinese Paper-cutting to create new paper-cuts from scratch. It is a design tool that helps the artists to produce paper-cuts with a professional look to them, even if they do not have a lot of experience in the craft.

A method created by Meng and colleagues [MZZ10] can be used to make paper cuts from human portraits. Therefore representative binary templates are used to portray the facial features. Based on directions from the artist a few pre-defined curves are proposed to enforce connectivity. Since the templates are known and they assume similar positions in the stencils, the same few predefined curves work for all the stencils.

2.2 Articles on automatic Stencil Creation

Stencils are used a bit differently from Chinese Paper-cuts. Not they themselves are the piece of art, but the painting on the wall that is done with them as a tool. That is why

they can often look somewhat different. But the physical requirement stays the same: The whole material needs to be connected. Yuki and Takeo Igarashi created a tool called "Holly" that helps users to draw stencils that meet those physical constraints. [II10] With that tool stencils can be created with various settings for ensuring the connectedness: either bridges to connect yet unconnected components are drawn automatically, or the user is informed by a green highlighting of the components that are still unconnected. If the user draws a self-intersecting line (which leads to an unconnected part), one of the two sectors of the line that intersect is broken up so the other line can pass through without creating an isolated white region. The user can decide if he wants to have one line to be broken up or the other one. More on this subject in Section 6.

Bronson and colleagues proposed a method for generating expressive stencils from polygonal meshes [BRO08]. They describe the method as follows: "In this system, users provide input geometry and can adjust desired view, lighting conditions, line thickness, and bridge preferences to achieve their final desired stencil. The stencil creation algorithm makes use of multiple metrics to measure the appropriateness of connections between unstable stencil regions. These metrics describe local features to help minimize the distortion of the abstracted image caused by stabilizing bridges. The algorithm also uses local statistics to choose a best fit connection that maintains both structural integrity and local shape information. The goal is to produce a stencil form 3D polygonal meshes. The connections that are considered are bridges orthogonal to the two shapes gradients they connect." Every shape needs only one connection so the method of connecting the shapes is a (slightly altered) minimum spanning tree problem. In our paper on the other hand the stencils are created from any binary images. So the user does not have to create 3D geometry or create the whole stencil with the help of the tool, but can just input any image and get a stencil out of it.

Arjun Jain and colleagues tackled a similar problem [JCT+15] in 2015. They created an algorithm that for a given user design generates a set of stencil layers satisfying all required properties. Images can be produced by applying multiple stencils in a certain order on the same spot with different colors. In this case, also a raster image is input.

In all of the above approaches, stencils have connections that minimize the added material to the stencil but do not try to follow the curvature of the input design. The input design was specifically created with the goal of having a stencil in the end in mind. In our work on the other hand any design can be the input, even if it was created for other reasons (e.g. mandala). The stencil also has to work with a material as frail as paper. So more added connections are the goal while still maintaining the curvature of the original image. That is tried to achieve by continuing the curvature of the shapes over sharp corners and finding the suitable connections by creating a graph and finding the maximum-cardinality matching.

2.3 Consequences of Raster-to-Vector Conversion

The study "Exploring and Evaluating the Consequences of Vector-to-Raster and Raster-to-Vector Conversion" deals with the ramifications of the conversing from raster to vector format and vice versa [Con97]. This is something that has a lot of impact on our results as we will see later in Section 5.

CHAPTER 3

Algorithmic Pipeline of the Stencil Creation

The design goal is a stencil (meaning all components are connected) out of the input image where the needed bridges follow the curvature of the input image where possible. Some steps must be executed to get there from a raster image input. In this Section these steps are described.

3.1 Input image

The input image is a preferably binary raster image. If the image is not binary it is converted with a user-defined threshold. The first white component can start from the borders of the image or it can start from inside the image. The resolution should be high in relation to the complexity of the image (more on this in Section 5).

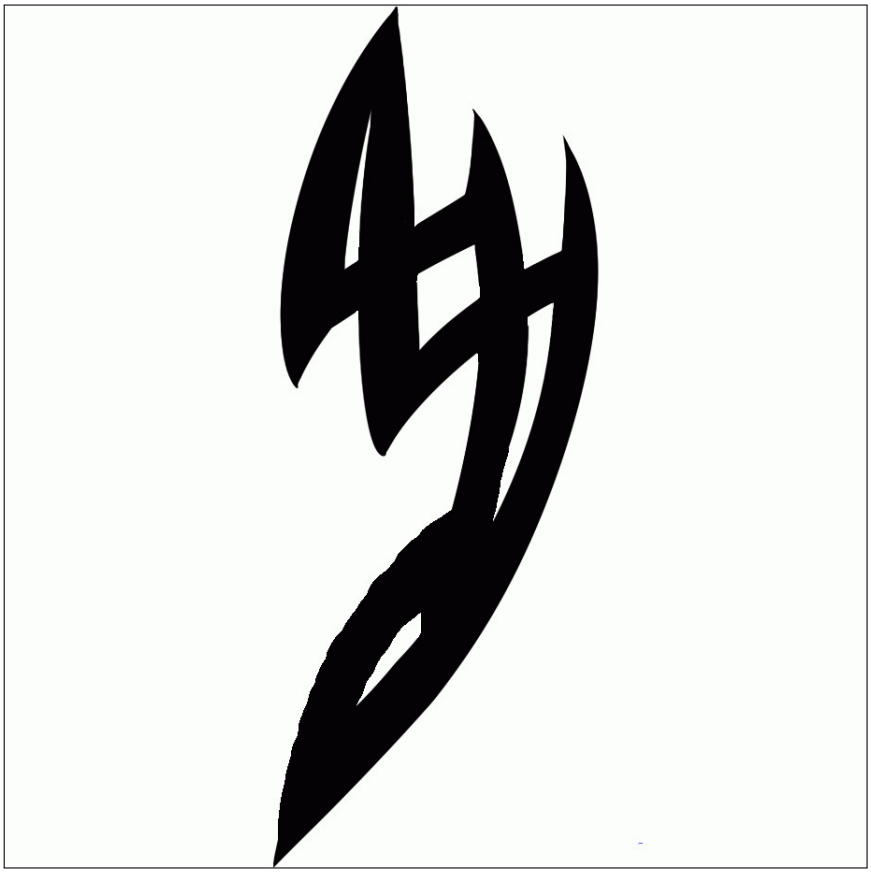


Figure 3.1: an example of an input image

3.2 Finding the connected components

The very first step to connect all the components of an image is to identify those components. This was done by using the floodfill algorithm. This algorithm starts from a seed point and gives the connected component it is part of a new color. To do that it looks for every surrounding pixel if it has the same color as the seed pixel. If it does, it gets the new color (or label) and the same thing is done for that pixel (looking at all the surrounding pixels). Therefore in the end all the pixels that are part of the same connected shape as the seed pixel have the new color [Smi79a].

This floodfill algorithm [Smi79b] is used alternating between black and white since the shapes could be nested (like concentric circles). The seed points are found like this: we start with the very outer surrounding pixels of the image (in most cases a rectangle). We look if the first of those pixels has the color we are looking for and if it does, we perform a floodfill operation. The pixels of the color we are looking for that are found by this floodfill operation are labeled as "reached". If one of the surrounding pixels was already reached by a previous floodfill operation, the operation will not be performed again with

it as a seed point (since it is labeled as "reached"). The pixels where the floodfilling stops, because they have the wrong color, but they were not yet labeled, are saved as the new surrounding pixels. Those will be used for the next floodfill operations that are done for the other color. This set of floodfill operations is performed after all surrounding pixels for the previous operations were labeled as "reached". The surrounding pixels are saved in a queue (first in - first out). Like that it is ensured that every single component in the image is reached eventually. The white components are saved as the components of the image.

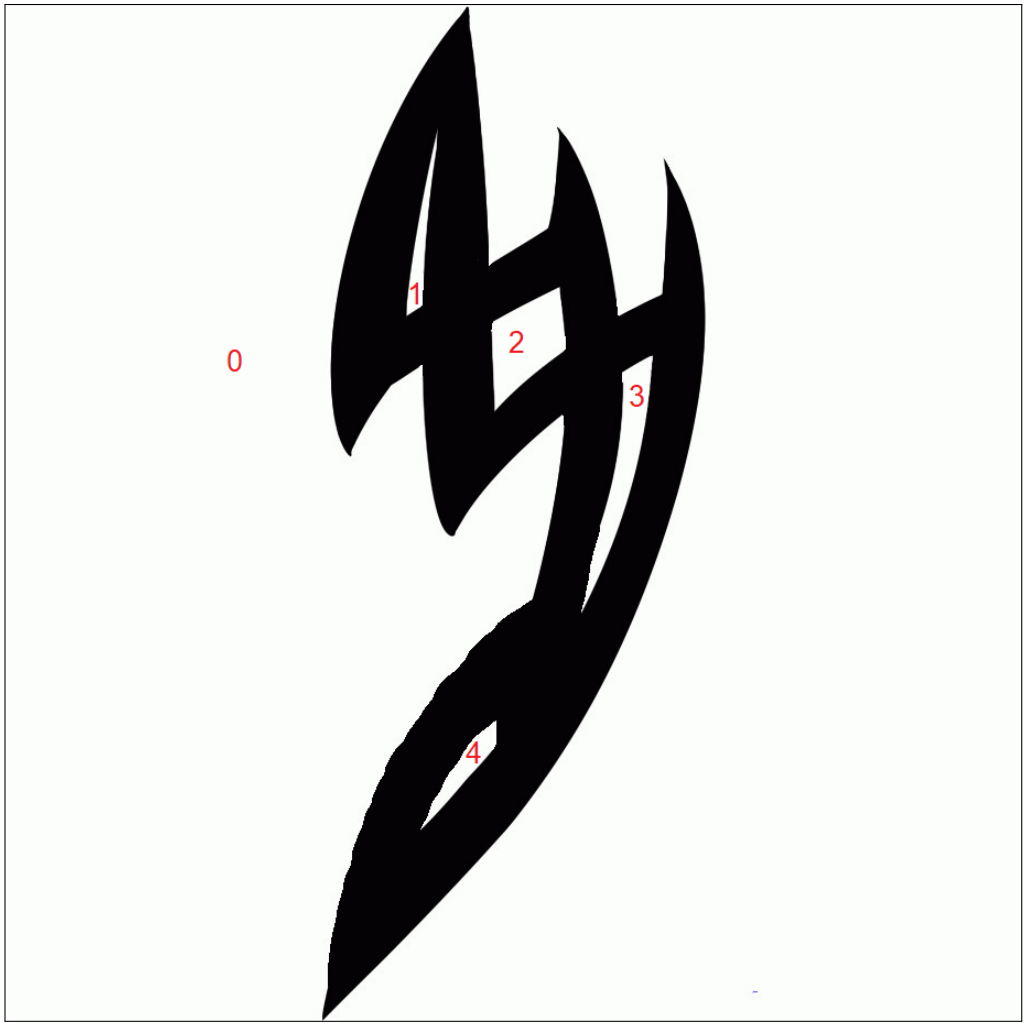


Figure 3.2: the components are labeled in the order they are found in

For the floodfill algorithm, there is a need to decide between the use of 8-connectivity or 4-connectivity (Figure 3.16). 8-connectivity means two components are considered as connected even if their pixels only touch on the corners (diagonally). 4-connectivity

means that no two components are determined as connected if their pixels only touch on the corners (diagonally). This produced some few-pixel artifacts in some of the data images. By using 8-connectivity those artifacts can be avoided.

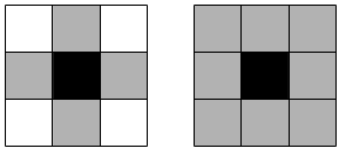


Figure 3.3: left: 4-connectivity vs. right: 8-connectivity
black: pixel in question, grey: pixels considered as its neighbors

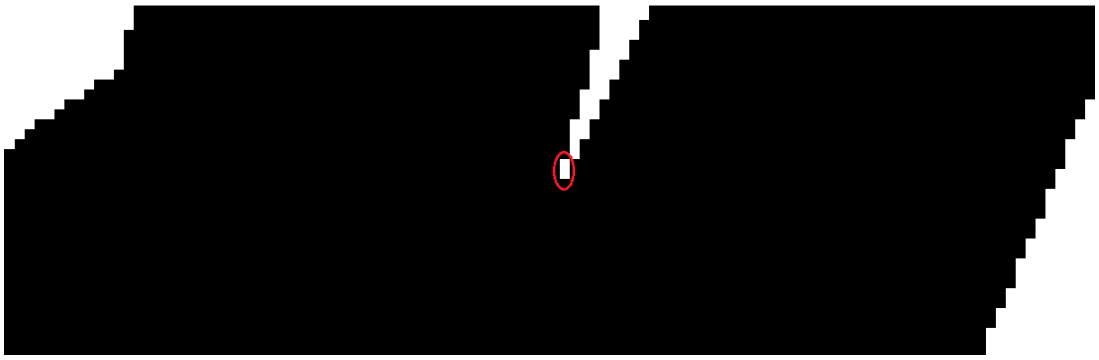


Figure 3.4: few-pixel artifacts
circled in red: an example of a connected part that only is found with 8-connectivity

3.2.1 The label matrix

The label matrix is a data structure where each field of the matrix represents a pixel in the original image. The value of each field is a label. A pixel in the background (black part) of the image is labeled with the value -1. Any pixel of a component is labeled with the order the component was found in. For example, the pixels of the outermost white component are labeled with 0, the index the component also has in the vector of components (since it was found first).

This is done using the binary matrix, where only the pixels of the component found at that moment have the value true, as a mask on the label matrix. All the values of the masked pixels are set to the component's index in the label matrix. This matrix later is used to immediately identify which component a given position in the image belongs to.

3.3 Detecting the contour points and their tangents

Now that the components are detected, we need to trace the lines and get the vector information to accurately draw bridges that follow the curvature of the shapes of the image. The process that is used for this is called vectorization. It is the inverse operation of rasterization. But the two operations are not bijective, since the vectorization operation is not an exact operation and it follows human set parameters that lead to different results.

In the process of vectorization, curves are fitted onto the raster data with the method of least squares. [HF98] For this task open source software solutions exist, like *autotrace* [WH04].

The traced data is returned in an SVG-file [Qui03]. SVG-files (short for Scalable Vector Graphics) are XML-style files that describe vector data. The XML-nodes of the SVG-file are of type path, line, circle, rect etc. (<path ...> ... </path>). But every shape is representable by the type path (which *autotrace* does so we only need to handle that one).

One parameter of the path node is the color. In our case, it is either black or white. Since the data is always represented in the manner of filling a closed path to get a shape of a certain color we know that if the color is white we know the path is surrounding the component and if it is black we know the path marks the inside of a component (the components in question are only the white parts). That is important to determine if a corner is convex or concave. To find a connection continuing over one side of a corner to another component, we need to only look at a shape's convex corners. Because continuing a concave corner in the direction of one of its tangents only leads into the shape itself and not out of it. Therefore concave corners of the components are not suitable to draw our connections.

The data the path node contains is a string that possesses the following form. All points are expressed by an x coordinate and a y coordinate that are separated by a space. The coordinates are real numbers (with or without decimal places). The path in the used type of tracing has two possible types of curves: cubic Bézier curves and lines.

If after a point follows the character "C" [1] for cubic Bézier curves, the next two points (the next four space-separated numbers) are the control points of the curve and the third point that follows is the endpoint of the curve. For example: "10 8C15 12 20 12 25 8" is the cubic Bezier Curve between the points (10, 8) and (25, 8) with the control points (15, 12) and (20, 12). If after a point follows the character "L" for line the next point after the character is the endpoint of that line. For example: "10 8L15 12" is the line between the points (10, 8) and (15, 12). With the character "Z" a path is closed. The first point and the last point of a path mark the same point in the used method of vectorization.

[1] uppercase because our method of tracing outputs the SVGs only in absolute coordinates and uppercase characters are the indication of absolute coordinates as opposed to lowercase characters which indicate relative coordinates

With this knowledge, the data can be parsed. Every start- and endpoint of a path is a point on the curve around the shape, a contour point. The directions of the two tangents in such a contour point are found by computing the direction from the point before the contour point to the contour point itself and the direction from the point after the contour point to the contour point itself.

A tangent of a cubic Bézier curve is the derivative of the equation of that Bézier curve. The curve is described by

$$P(t) = (1-t)^3 * P0 + 3t(1-t)^2 * P1 + 3t^2(1-t) * P2 + t^3 * P3$$

so its derivative is described by

$$dP(t)/dt = -3(1-t)^2 * P0 + 3(1-t)^2 * P1 - 6t(1-t) * P1 - 3t^2 * P2 + 6t(1-t) * P2 + 3t^2 * P3$$

The tangents we need are the one at the starting point and the one at the ending point. So the tangent on $t = 0$ and the tangent on $t = 1$. If we plug those values into the equation we get for the tangent at the starting point t_1:

$$t_1 = -3 * P0 + 3 * P1$$

and for the tangent at the ending point t_2:

$$t_2 = 3 * P2 + 3 * P3$$

Ergo the tangent at the starting point is the line from the starting point to the first control point and the tangent at the ending point is the line from the ending point to the last (second) control point.

A tangent of a line is always the line itself at every point of the line, therefore also in the start point and endpoint of the line.

Now, every contour point has two tangents, going into the two directions the path follows before and after this contour point. With these two tangents, the angle in the contour points can be computed for finding corner points. Every point that describes an angle of a degree smaller than 135° is considered a corner point and will be eligible for a connection. The points with an angle greater than 135° are left aside for now and considered later for additional connections that will be needed to connect shapes with few corners and round parts.

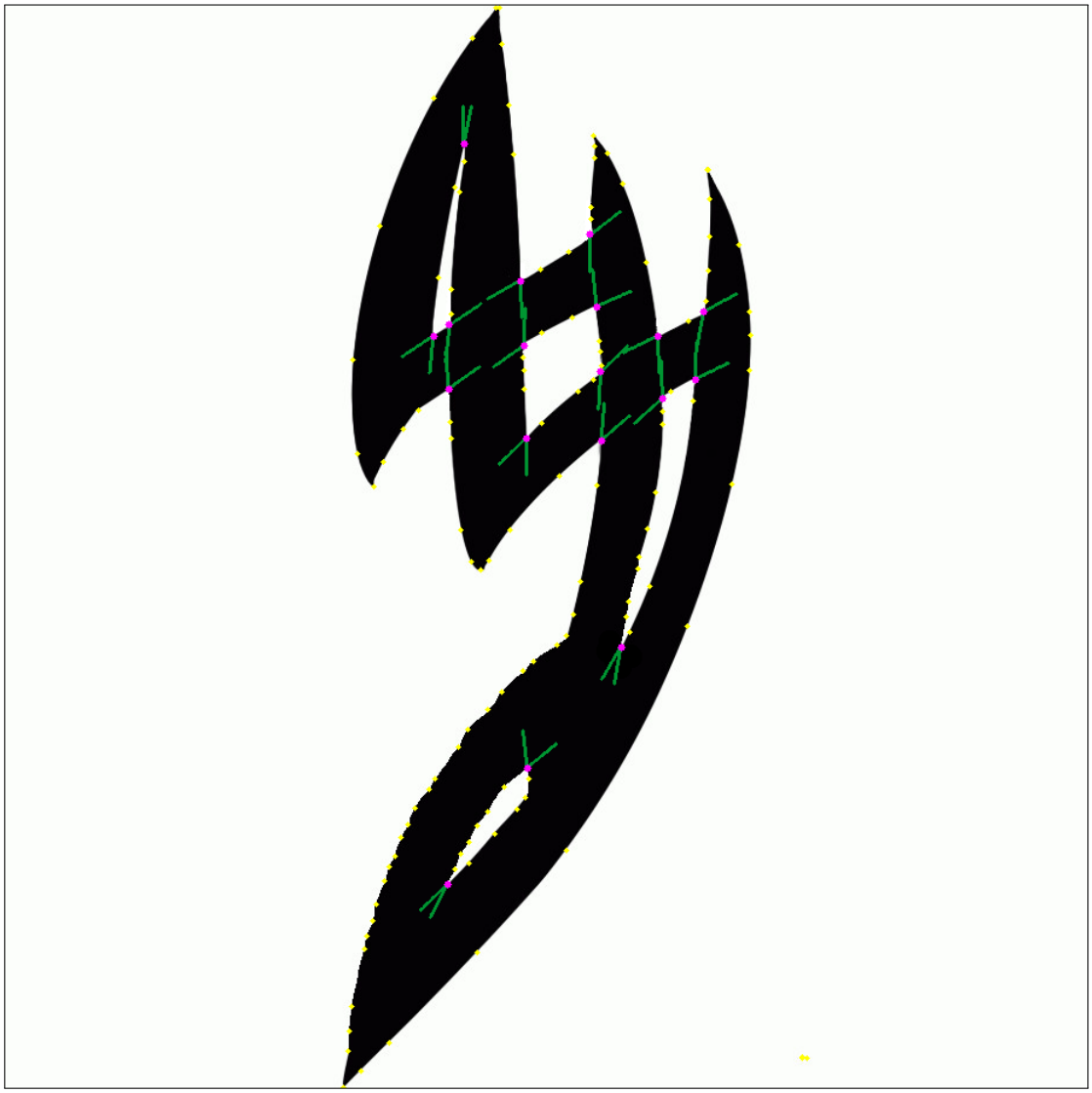


Figure 3.5: the information retrieved from the vector representation of the image
yellow: contour point, pink: corner point, green: short section of a tangent at a corner point

The reason for this angle is: if the two tangents make an angle of a degree less than 135°, the tangent that describes the direction of the connection drawn and the next side of the component form an angle between 45° and 135°. The connection is wanted to be short, because less amount of paper is left in the stencil yields a greater similarity to the original input. If the connection leaves the component at an angle between 45 and 135 degrees, the distance from the component covered by the connection (a in Figure 3.9) is greater than the distance that is parallel to the other side of the component we are departing from (b in Figure 3.9).

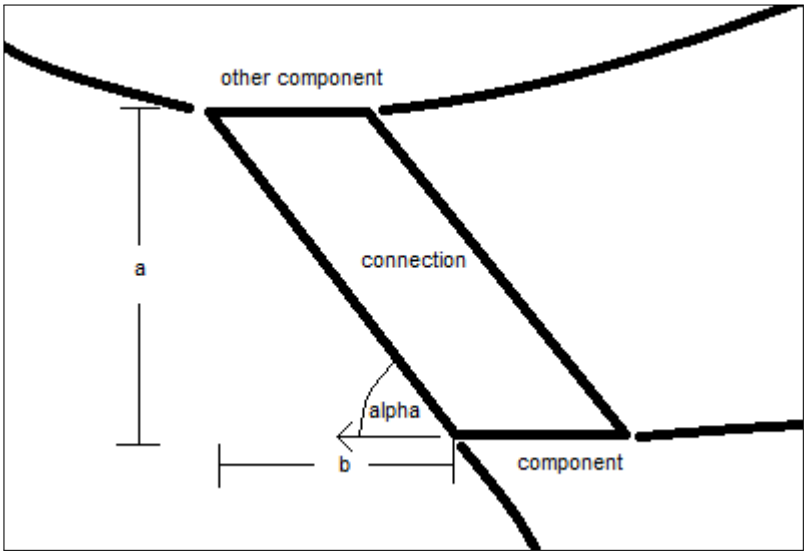


Figure 3.6: explanation of the chosen angle
if alpha is greater than 45 degrees and smaller than 135 degrees, a will increase at a higher rate than b when increasing the length of the connection

One exception for making a point a corner point is if two corner points are less than 2 pixels apart. Because that is not two pronounced corners in the image but more like one corner that was made into two contour points during the vectorization. Therefore one of the two corners has to be discarded and for the tangent pair of the corner that is kept the tangent of the side which leads to the discarded corner point (which is less than 2 pixels long) needs to be swapped with the other tangent of the discarded corner point. Then we have a more exact representation of the corner and no tangents that are part of a negligible (because it is shorter than two pixels) side.

3.4 Types of Points in this Paper

When speaking of points there are four types of points in this paper:

- **Real corner points:** are all the contour points where the two tangents describe an angle of less than 135°. They are all considered to have connections.
- **Contour points:** are points on the contour that are beginning and endpoints of lines and curves on the path of a component. Only some of them are considered for a connection if there are not enough corner points to make connections.
- **Added points for straight connections:** are points not present in the original SVG-file but are found when a corner point (or contour point when needed) has no possible connections, so a straight connection is made. The first point that is on

another component when going step by step in the direction of a tangent of the initial point is our "added point"

- **Drawing points:** points needed to draw the connections in the end. The other types of points will always be two of the drawing points in a connection. The other points are found with additional calculations.

The first three types of points are represented with a vertex in the graph explained in the next section.

3.5 Generating a graph structure with the corners as vertices and the connections as edges

A simple way to connect the components of an image would be to draw short, straight connections between different components that are close and perpendicular to the contours of those components (which is also the way the problem was approached in "Semi-Automatic Stencil Creation through Error Minimization" [BRO08]). But this could be considered as disturbing the flow of the image, so we tried to avoid that. In some cases (for example if a component is a circle) it will still be necessary to draw perpendicularly from the contour of the component, but in cases where there are sharp corners in the shape a different method of finding connections is possible. We draw the connection by continuing into the direction the contour was going before making the sharp turn on the corner. This will result in smoother transitions between the component and the connection. With the other edge of the corner, the connection will make a sharp angle but that is acceptable, since there was already a sharp angle before (at the corner, into the opposite direction).

This is why the components are connected by drawing connections from the corners of the image and then drawing perpendicular connections where needed. A graph is created to later use the "weighted-matching-algorithm" on it to choose the connections that will be drawn from all possible connections. The corners of the components will be the vertices of this graph. An edge is introduced between every pair of vertices that could be connected.

As Siek and colleagues explain in the manual of the *Boost Graph Library* [SLL02] a matching is a subset of the edges of a graph where the edges do not border each other (meaning no vertex is contained in more than one of the matched edges). A maximum-cardinality matching is a matching of maximum size over all possible matchings in a graph. A maximum weighted matching is a matching over an edge-weighted graph, not necessarily a maximum-cardinality matching, but the sum of the weights of the matched edges is maximum over all other possible matchings.

The maximum cardinality matching works as follows: First, an initial maximal matching (maximal means that it is a matching that cannot be increased by adding edges because no additional edges meet the condition not to share any vertex of the matching) is

found[1]. Now, between any two vertices that are "free", meaning that they do not belong to any edge of the matching, an augmenting path can be found. An augmenting path between the two points alternates between a matched edge and a non-matched edge. All matched edges and non-matched edges can be swapped and that gives a matching with one more edge in it. From the initial maximal matching, augmenting paths are being found until no further augmenting paths can be found. In the end, it is verified that the matching found is maximum. The weighted-matching problem is the generalization where edges are assigned a weight and the matching with the maximum weight is to be found. The weighted algorithm works as a combinatorial algorithm that uses the unweighted-matching algorithm as a subroutine.

We use the implementation of the *weighted-matching algorithm* the *Boost Graph Library* provides. The weight of an edge is determined by the distance the connection spans. Since the *weighted-matching algorithm* tries to maximize the weight of the matching and the connections with a minimum distance between the connected points are the goal, the distance value has to be inverted for the weight.

For this, the weight of an edge is calculated by subtracting the distance of its two vertices from a value absolute for one component. That value is the maximum distance of two points of that shape. Since two shapes are being connected the value is the smaller one of the two distances calculated. This is done with the following algorithm:

- ps ... set of all contour points of the shape
- distance(Point a, Point b) ... function that returns the distance between point a and point b by calculating it with the Pythagorean theorem

Algorithm 3.1: Find max distance between two contour points of a shape

Result: maxD ... maximum distance

```
1 maxD = 0;
2 for i = 0; i < ps.size; i++ do
3 |   for j = i + 1; j < ps.size; j++ do
4 |   |   d = distance(ps[i], ps[j]) ;
5 |   |   if d > maxD then
6 |   |   |   maxD = d;
7 |   |   end
8 |   end
9 end
```

[1]The initial maximal matching is found greedily by just adding one edge to the matching after the other until no further edges can be added. The edges can be sorted first in increasing order by the degree of the two vertices in the edge.

This leads to the effect that bridges between larger components are favored over connections between a large and a small component. Another effect is that the edge of a bridge which is longer than the shapes it connects, has a negative weight, discarding it for the matching. This is done because adding bridges that span longer than the size of the smaller one of the components it is there to connect, would make a significant change to the original image. That is not the desired result.

If a component has not enough corners in it, we randomly take some of its contour points to look for connections. We just take the first point of the component if there are none yet. If there is a point already, we take the point with the greatest distance from that point. Since the tangents depart from the shape in an angle smaller than 45° they are not eligible for sensible connections. That is why for the selected contour point a new "tangent" is introduced, which points into the direction between the two other tangents, so it is orthogonal to the contour of the shape.

There exist three cases that are treated differently for the corners.

- The first case is if one of the two tangent rays of a corner intersects with one of the two tangent rays of another corner in an intersection point that is not in any other component but in the black part in between the two components the corner points are from. In this case, it is possible to draw a curved bridge from one of the corners to the other so that in both corners the slope will be preserved onto the bridge.

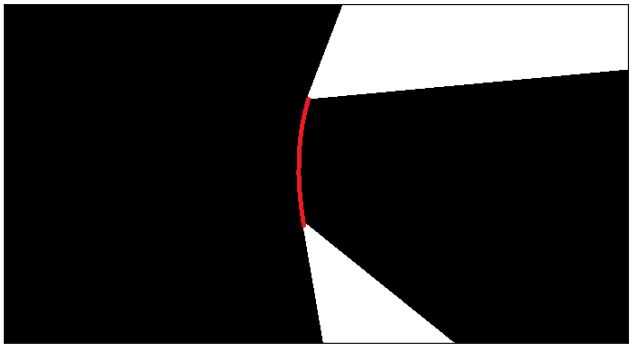


Figure 3.7: a curved connection between two corner points
the slopes at the two corner points can both be preserved into the connection
white: component, red: connection

- The second case it is possible to connect the corner with a corner of another component but there is no intersection point of their tangents since their tangents lie on the same line. Other cases where this type of connection could be made between two points are: the intersection point of the tangent lines is either before the start point of one of the rays or in another component (we do not want to draw connections through other components), or it is too far out (if the distance from one of the corners to the intersection point is greater than the distance between

the two corners we also do not draw the curve. Because that either means that the bridge would go behind one of the corner points and then turn back. Or it means that the bridge would turn out/away from a direct connection between the two corners farther than the direct connection itself would be. And In this case, preserving the slope is not worth it drawing such a big turn. In this case, we will connect the two corners but with a straight bridge from one point to the other.

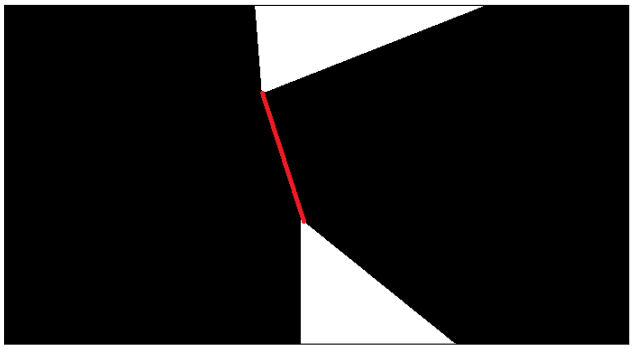


Figure 3.8: a straight connection between two corner points
two corner points are being connected but it is not possible to preserve the slopes with a quadric Bézier curve into the connection
white: component, red: connection

- The third case is none of the above two cases are found for a corner. In that case, we continue both tangent rays of the corner straight until we find another component. We take the one that has the shorter path to another component (because shorter connections have more stability and are less change to the original image). The pixel where we found the other component is added to the graph as an additional vertex so that we can put an edge between the two vertices. This connection will for sure be chosen by the matching algorithm since it is the only edge both those vertices have.

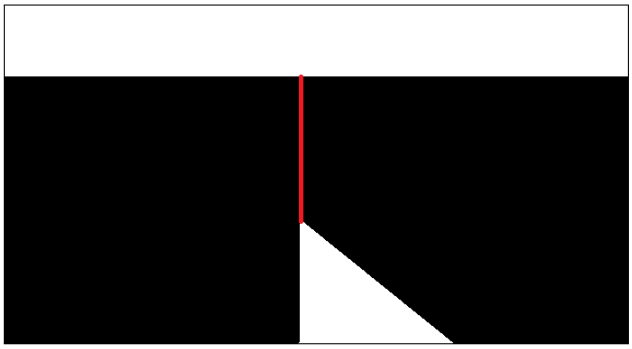


Figure 3.9: a straight connection from a corner point
there is no second corner point found we can connect the first corner point with, so we just make a straight connection to the next component
white: component, red: connection

Now that we have initialized the graph we can use the matching algorithm on it to choose the connections we want to draw. If an edge was added twice in the filling of the graph[2] it does not matter, since a vertex pair can at most have one edge in an undirected graph. The edge will be added just once. We use the matching algorithm to find a matching of our graph. As defined before a matching of a graph is a subset of the edges where no two edges share a vertex. This is exactly the behavior we want for our algorithm since no corner needs to be connected more than once and multiple connections per corner would disturb the flow of the original image more than necessary. But also it takes the most possible number of candidates for bridges, which leads to a very stable stencil with connections at all the very convex points of the gradient.

The graph G is an undirected graph. The set of its vertices V is the set of corner (and contour) points that are eligible for a connection. The set of edges E consists of edges between every two points that can be connected. If they can be connected is decided via criteria in the input image (Points that are on the same component in the image do not need to be connected and points must be visible to each other to be eligible for a connection). The edges have a weight that is calculated by subtracting the distance the connection spans from the maximum size of the smaller of the two components that will be connected.

3.6 Adding connections for yet unconnected components

Now there is one step left to mathematically make sure that the stencil will be connected. Every component is checked for being connected (directly or indirectly) to

[2]That can happen for a lot of edges since if a possible connection is found for one corner it will be found again when looking at the other corner. So it only does not happen for connections with the outside component, since we do not explicitly search for connections of its corners. And an edge is also only found once for the third case of connections, where a point was specifically introduced for the connection.

the outside component. This is done by stepping through each edge of the matching. If a component is not connected additional connections that connect it to either the outside or to components connected to the outside are searched. In a loop it is checked which components are connected (directly or indirectly) to the outside and they are saved as connected. If we do not know yet if a component c is connected, every component that is connected to c is saved in a small heap of c. Now if c later will be saved as connected, all the components in its heap are saved as connected too.

Algorithm 3.2: Check for connectedness of all the islands

Result: unconnectedComponents, dependingComponents

```
1  edges ... matched edges;
2  c0 ... outside component;
3  connectedComponents ... heap of all the connected components;
4  unconnectedComponent ... heap of all the unconnected components;
5  dependingComponents ... array of empty heaps (one heap for each component
     except c0);
6  connectedComponents.add(c0);
7  unconnectedComponents = all components except c0;
8  foreach edge in edges do
9      c1 ... first component the edge connects;
10     c2 ... second component the edge connects;
11     bool c1connected = c1 in connectedComponents;
12     bool c2connected = c2 in connectedComponents;
13     if c1connected AND !c2connected then
14         connectedComponents.add(c2);
15         unconnectedComponents.remove(c2);
16         foreach dependingComponent in dependingComponents[c2] do
17             connectedComponents.add(dependingComponent);
18             unconnectedComponents.remove(dependingComponent);
19         end
20         dependingComponents[c2].removeAll();
21     end
22     else if c2connected AND !c1connected then
23         connectedComponents.add(c1);
24         unconnectedComponents.remove(c1);
25         foreach dependingComponent in dependingComponents[c1] do
26             connectedComponents.add(dependingComponent);
27             unconnectedComponents.remove(dependingComponent);
28         end
29         dependingComponents[c1].removeAll();
30     end
31     else if !c1connected AND !c2connected then
32         dependingComponents[c1].add(c2);
33         dependingComponents[c2].add(c1);
34     end
35 end
```

All components will be connected to other components. But it is possible that this algorithm finds out a group of components is only connected with each other and not with the outside. Then additional connections have to be found. Every contour point of it is looked at and a line from it perpendicular to the contour of the component is drawn. Every line that ends up at a connected component is saved and of those the shortest one is taken. This is done for all of the unconnected components. Therefore every component will be connected (directly or indirectly) to the outside component. See also Section 3.8.

3.7 Drawing the connections between the components

Now between every two points of a matched edge, a connection needs to be drawn. If the connection will be straight, just drawing a line between them is sufficient. But where possible, a curve between two corner points can be drawn. This is to better preserve the slopes of the gradient in the original image. Like this as few new corner points as possible are introduced to ensure smooth connections and not disturb the flow of the original image more than necessary. The curves that will be drawn are quadric Bézier curves. Quadric Bézier curves need a starting point, an endpoint and one control point to be drawn. The curve does not go through the control point, it just approaches it. The start- and endpoints are already determined. The control point is found by taking the intersection point of one tangent ray of the start point and a second tangent ray of the endpoint. Because if the control point lies on the tangent of the corner, the tangent of the quadric Bézier curve at the start point is the same (this is an inherent attribute of Bézier curves). So the transition from component to connection is smooth and does not introduce a new angle at the corner. The same is true for the endpoint and its tangent. That is why this approach of finding the intersection point and taking it as the control point for the Bézier can be used.

Each matched edge of the graph can be represented by either a straight line between the two points or by a curve. But those geometric figures have no width (or in the computer graphics case: width of one pixel) which is not suitable for a stencil. The components of the stencil need to be connected by connections of some thickness, that can be materially created in the real world. Therefore the thickness needs to be calculated. The thickness can have some uniform maximum value computed with the size of the image, which will create a more uniform look. But not for all connections this maximum thickness can be achieved. The connection has the same thickness throughout but it can not be broader than the components it connects allow. For the algorithm that gives the maximum thickness of a connection the label matrix computed in Section 3.2.1 is used, in case one of the vertices represents a newly introduced point, where the component it is from needs to be found out.

The algorithm goes as follows:

Algorithm 3.3: Find maximum possible thickness of the connection

Result: d

```
1  p1...first point of the connection;
2  p1p...first point projected;
3  t1...tangent in the corner p1 that marks the direction of the connection in p1;
4  t2...other tangent in p1;
5  labelMat...the label matrix;
6  label1 = label of the component p1 is from;
7  d...distance (integer value);
8  maxD...maximum distance allowed;
9  for d = 0; d < maxD; d++ do
10     p1p = d steps into the inverse direction of t2 (from p1);
11     x = 0;
12     while p3 is in image bounds AND labelMat.at(p3) != label1 do
13         p3 = x steps into the inverse direction of t1 (from p1p);
14         x++;
15     end
16     if p3 is out of image bounds then
17         d -= 1;
18         break; //terminate for-loop
19     end
20 end
21 do the same for the other corner point (with its tangents respectively)(if it is not
     a corner point, but just some additional point on the contour of the second
     component because the connection will be straight, we use the tangent
     perpendicular to t1 to take d steps and end up on p2p, we use t1 departing from
     p2p we take steps trying to find a pixel with the label of the second component);
22 smaller one of the two distances found is taken;
```

With the corner points, their tangents and the maximum thickness of the connection we can draw a connection with some width. We differentiate between curved and straight connections.

The connections are drawn by writing new path nodes into the SVG-file. For this the start points, control points (for curved connections) and endpoints need to be found and written in the correct order in an XML-node of type path. The parameter *style* of this XML node is set to "fill: white" and the parameter *stroke* is set to "none".

3.7.1 Straight connection

A straight connection departs in a straight line from a corner point $p1$ (or a contour point when it is an additional connection introduced for round shapes) to a point $p2$ which is found by finding the first point that lies on a different component than $p1$ if you start from one of its tangents. The line is not allowed to intersect its own component, otherwise the connection will not be used. From $p2$ a line has to be drawn to $p3$, which is a point that lies in the opposite direction of the not used tangent ray of $p1$ departing from $p2$ and has the perpendicular distance d (computed before with the Algorithm 3) from the line from $p1p2$. From $p3$ a straight line to $p4$, which is a point that lies in the opposite direction of the not used tangent ray of $p1$ departing from $p1$ and also has the same perpendicular distance d from the line $p1p2$. The shape is closed with a line from $p4$ to $p1$.

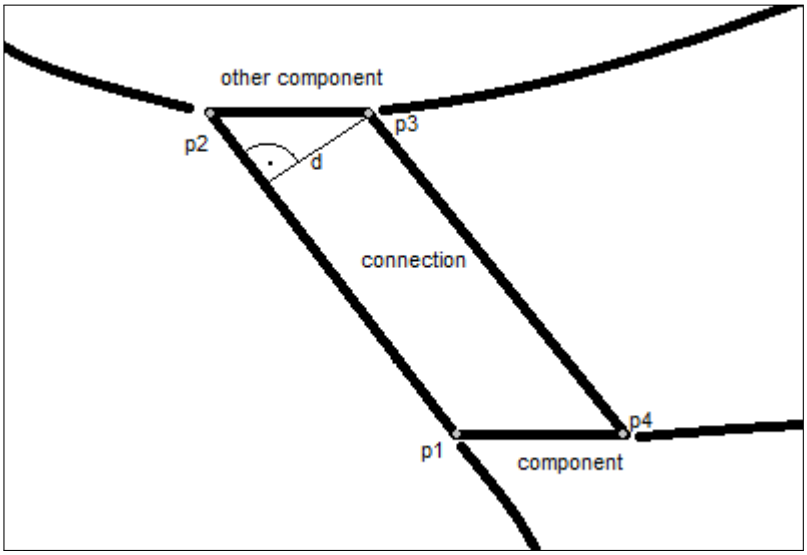


Figure 3.10: straight connection
points of a straight connection

3.7.2 Curved connection

Two corner points are found and they are visible to each other through a clear path (not occluded by any component) and one of their tangents each intersect at a point that lies in between them. In that case, it is possible to draw a curved connection such that at both corner points the slope of one side of the corner is preserved, by making one

continuous curve. This can be done by drawing a quadric Bézier curve with the two corner points as start and endpoint and the intersection point as control point. Because the connections need to have a certain thickness it is necessary to project two other points and one more control point to draw a second curve, between which and the original curve the image is filled out. Two cases are differentiated: if the corners turn to the same side and if the two corners turn to opposite sides.

Both tangents to the same side

Directionality of a corner: the direction the unused tangent points to when looking in the direction of the used tangent.

Figure 3.11: left directionality

Figure 3.12: right directionality

Figure 3.13: *black: corner point, red: used tangent, blue: unused tangent*

With this definition for directionality, we can determine if a tangent pair of a corner has right or left directionality. Both corners go to the same side if the two tangent pairs have opposite directionalities (one turns left and one turns right), because the two used tangents are facing each other (otherwise they would not have an intersection point ip in between the points). For this case the two corner points $p1$ and $p2$ can be projected perpendicular to the line between them for the distance d calculated earlier. The point ip is projected in the same direction.

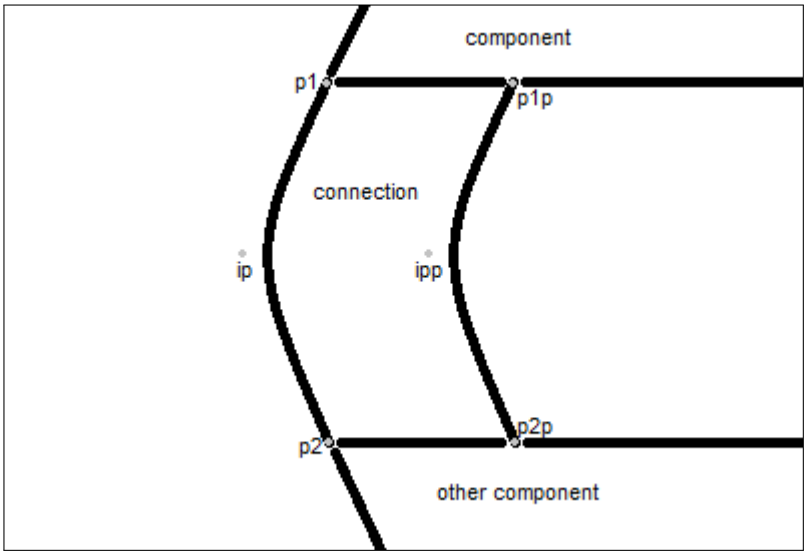


Figure 3.14: curved connection
when both corners go to the same side

Directionalities to opposite sides

If the two corners have the same directionality, the corners go to different sides. This is a special case because the thickness of the curve will be given by stepping to different sides. In this case, the curves are each made by one of the original points and one projected point, so the intersection points of the corners' tangents need to be found again. No such intersection point may be found between the two points, in which case the third type of connection (Section 3.7.3) is made. The projection of the two original points will also be done differently since the points are lying diagonally from each other in the rectangle made out of the original and projected points. The projected points are found by rotating the line between $p1$ and $p2$ around its midpoint so much that the distance d is reached by each of the projection distances.

$$\alpha = 2 \arcsin(d/2a) \tag{3.1}$$

The formula 3.1 is used to find the angle α by which a corner point is rotated around the midpoint of the line between p1 and p2 to find the projection of distance d from the original point. a marks the distance from the midpoint to the corner point. If d is greater than $2a$ the projections can not be found using this way, since there is no arc sine of values greater than 1. In this case, again the fallback to 3.7.3 is used.

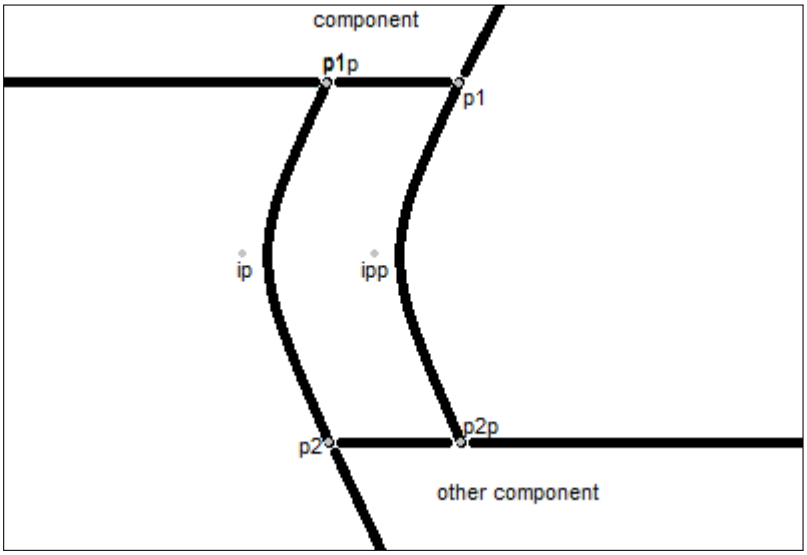


Figure 3.15: curved connection
when the corners go to opposite sides

3.7.3 Straight connection between two corners

A straight connection between two corner points is used when the two corner points are visible to each other but none of their tangent rays have an intersection point that can be used (it does not lie in between the two points). The intersection point is not allowed to be inside a third component (because that makes the curve traject too close by the component or even intersect it). Other cases where this type of connection is fallen back to were already mentioned earlier in "Curved connection" 3.7.2. The straight connection between two points is drawn just like the curved connection, with the difference that not two Bézier curves are drawn, but straight lines between the points and their projections respectively.

3.7.4 Example of the usage of the three different types

The following figure shows the three different types of connections colored in a different way.

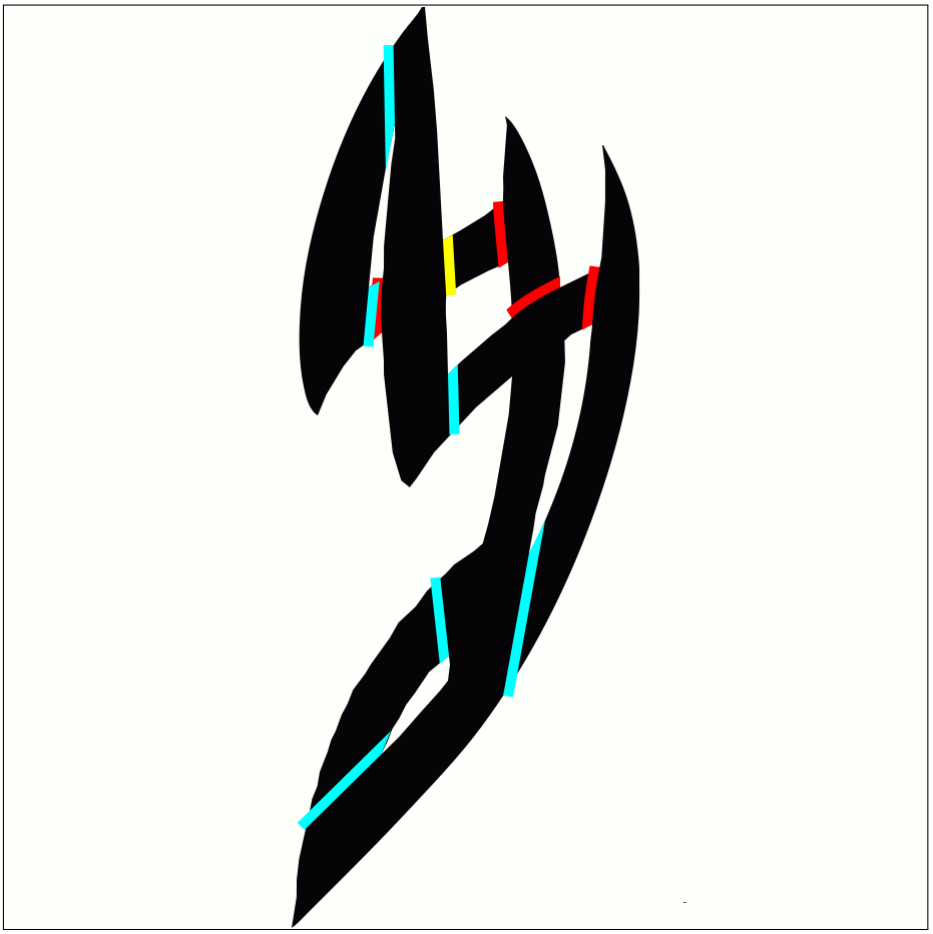


Figure 3.16: the output with colored connections.
cyan: straight, red: curved, yellow: straight between two points

3.8 Confirmation of connectedness

We have an ordered, finite set of components in our image. The end goal is that all of them are connected with the outside component. The set of components is ordered from the outside in (Section 3.2), so the outside component is the first component we look at. Subsequently, after using the matching algorithm on the graph we generated (Section 3.5) we make sure, that every component with at least one connection with a vertex of the outside component and the outside component itself is added to the set of connected components. If a component is connected to any component of that set it is also added to the set. This is done until all components are in the set. If not we will add a connection for the component that is not connected (Section 3.6).

Connectedness is transitive in the sense that a component A being connected to a component B, which is connected to a third component C, is connected indirectly to C.

If every component is connected to at least one component that is connected, they are also connected. Because of this, all components will be connected directly or indirectly to the outside component. Ergo the whole stencil will be in one piece.

CHAPTER 4

Implementation

4.1 Development Environment

The project was set up as a *C++* project with *cmake*. The integrated development environment that was used was *visual studio 2019*, which allows for step by step debugging, but other IDEs could also be used since there is no particular feature needed that only visual studio provides.

By using *cmake* the computer graphics library *openGl* could be linked to the project. This library was used mainly for displaying and saving the images in processing steps of the algorithm, having the images in matrix form and accessing pixels of those matrices and lastly for applying functions on the images that are provided by this library (like thresholding for example).

For the graph data structure and algorithms the graph library *Boost Graph Library 1.62* was used.

For the vectorization process, the open-source project *autotrace* [WH04] was used. It was called by the program from the command line.

4.2 Connected Components

The algorithm for finding the connected components works by using the floodfill already explained in "Finding the connected components" 3.2. [Smi79b]. The floodfill algorithm is modified so that it not only returns the floodfilled image, but also the points where the floodfilling stops. All the contour points of new components (where the last floodfilling stopped) are saved in a vector called *seedpoints*. At the beginning the frame of outermost pixels in the image is saved to *seedpoints*. The color to label the floodfilled black sections with is *color_0* and the color to label the floodfilled white sections with is *color_1*. Also

the hierarchy of the components can be saved we start on hierarchy 0 and every time we step inside a component enclosing other components the hierarchy is increased by 1. The algorithm is as follows:

- seedpoints ... vector of seedpoints
- seedpoints2 ... second vector of seedpoints
- hierarchy ... integer value of the hierarchy
- floodfill(seedpoint, oldcolor, newcolor) ... floodfill function from seedpoint, returns list of new seedpoints, side-effect: changes color of pixels and labels them as "reached"

Algorithm 4.1: Modified floodfill algorithm

Result: list of components

```
1  hierarchy = 0;
2  seedpoints = outermost pixels of the image;
3  while !seedpoints.empty() do
4      seedpoints2.removeAll();
5      foreach seedpoint in seedpoints do
6          seedpoints2 = floodfill(seedpoint, black, color_0);
7          //No pixels with label "reached" are put into seedpoints2 by floodfill(...),
             only white pixels where the floodfilling stopped.;
8      end
9      seedpoints.removeAll();
10     foreach seedpoint in seedpoints2 do
11         seedpoints = floodfill(seedpoint, white, color_1);
12         //No pixels with label "reached" are put into seedpoints by floodfill(...),
             only black pixels where the floodfilling stopped.;
13         if the component is newly found, save it to the list of components then
             give it the color color_0;
14         its hierarchy set to the current value of hierarchy;
15     end
16     hierarchy++;
17 end
```

During this algorithm also the label matrix is filled. Every black component is filled with the label -1 while every white component is filled with the value of the order it was found at (the first component that is found is labeled with 0, the next with 1 and so on). The label matrix is used to know in O(1) which component a pixel in the image belongs to.

4.3 Vectorization

The vectorization is done by saving the images of the isolated components to the hard drive with the openGL command *imwrite(inputpath, matrix)* where *inputpath* is a string that describes the path the image will be saved to and *matrix* is the Mat object (openGl data object) that contains the image data. Then the command line is called to use *autotrace* [WH04] (which needs to be installed before) to vectorize the raster images. This is done with the code line:

system("autotrace -color-count 2 -error-threshold 0.5 -output-file *outputpath* -output-format SVG *inputpath*");

where *outputpath* is a string that describes the path the image in vector form will be saved to and *inputpath* is a string that describes the path the raster image can be found at.

The parameter "error threshold" needs to be set. It determines which fitted curves are subdivided. If a curve is off (from the original data) by more pixels than the threshold the curve is subdivided into more curves. The default is 2.0 which is too inexact for our purposes. We set it to 0.5 since in the next step it is better to have more exact curves and it does not matter if it takes more curves to represent the image gradients (apart from a negligible increase in computation time).

4.4 Parsing of the SVG-file

The previously saved vector image is in the SVG format (scalable vector graphic) [Qui03]. This is an XML file where every node is one shape in the image. They are of type path, line, circle, rect etc. But every shape is presentable with just the type path so it is enough to work with path-type nodes. *Autotrace* does this, so only paths are output by it.

A path node contains the fill color as an attribute (for example *fill: #ffffff*), which in the case of this work is one of two things: black or white. If it is white, the path describes the outer contour of one of the shapes in the image. If it is black the path describes an inner contour of a white shape.

The attribute "d" or "data" contains the data of the points on the path. Its value is a string that looks the following way:

The character "M" denotes the start of a shape. After it comes a point.

The first number of a point is the x-value of the point and the next number is the y-value of the point. They are separated by a space. The point (0 0) is in the left upper corner of the image and x- and y-values increase from there.

In the case of *autotrace* all paths are described via cubic paths (denoted by the character "C") and lines (denoted by the character "L").

A point (two numeric values separated by a space) followed by an "L" followed by another point, means that the line between these two points is part of the path.

A point (the starting point) followed by a "C" followed by two points (the control points) separated from each other by a space and a third point (the ending point) also coming after a space, means that a cubic Bézier curve between the first and the last point with the two control points is part of the path.

When the description of one shape is complete another "M" follows. Unless the path itself is complete, then the data string is ended with a "z" (or an uppercase "Z"). If the last point of a shape equals the first point of that shape, the path is closed already. If the last point does not equal the first point the path of that shape is closed via a straight line between those two points.

With this information, the SVG can be parsed and all the starting and ending points of lines and cubic Bézier curves can be extracted and assigned to the shape it belongs to. Also, the two tangents of each of those points can be determined. A tangent of a line on each point of the line is always the line itself. A tangent to a Bézier curve on the start point is the line from that point to the first control point. A tangent to a Bézier curve on the end point is the line from that point to the last control point.

4.5 Corner points

We will call the starting and ending points of cubic Bézier curves and lines contour points from now on. The tangents found from the parsing of the SVG-files are reordered. So that each tangent is associated with the point it departs from. For each contour point now the tangent to the point coming before it t_1 and the tangent to the point coming after it t_2 are known. The points are ordered counterclockwise in the SVG-file (this is done so by *Autotrace*. Additional to the position of the point, also which shape it belongs to and if the path it comes from was black or white. That is important to not only find points with an angle sharper than some certain angle but also only the convex corners of the (white) shapes in the image. The angle between the two tangents is calculated with the following equation:

b ... contour point we want to calculate angle from
$\vec{ab}$ ... unit vector of tangent coming before
$\vec{cb}$ ... unit vector of tangent coming after

$$\theta = \arccos\left(\frac{\vec{ab}\cdot\vec{cb}}{\|\vec{ab}\|\,\|\vec{cb}\|}\right).$$

Now that the angle at the corner of the contour point is known, the points where there are sharp corners can be found. If we only want convex corners of the white components of an angle less than or equal to 135°, we take the corners of less than or equal to 135° for white shapes (corners on the outside) and corners of more than or equal to 225° for black shapes (corners on the inside). That is why we needed to save the fill color of the SVG paths before.

4.6 Adding connection points to shapes with no corners

Not all shapes have enough corner points to be connected stably. For example, a round shape would not have any corners. In the case of no previous corners, an arbitrary contour point is taken and one of its two tangents is redirected. Its new direction is the middle of the two tangents before, away from the component. For the finding of the connection later on only the redirected tangent is considered, since otherwise there would be unwanted effects when drawing the connection. The new connection of course will always be at an orthogonal angle to the contour of the shape, but that is the most elegant way we found to deal with round and curved shapes.

If there already is one or more corners present in the shape, the index that is farthest away from the other corners indexes (if you count through continuously, from the last index to index 0) is taken as a new corner index.

This is repeated for every shape until it has at least two connection points.

4.7 Graph library

The graph library used is the *Boost Graph Library 1.62*. Using it, a graph is created. The vertices have the attributes *componentIndex* (the index of the component) and *cornerIndex* (the index where the point is at in that component). The edges have the attributes *weight* (it will be explained later how the weight is calculated and what it is used for), *p1t1* (a bool, true if for the first point the connection is drawn in the direction of its first tangent and false if the connection is drawn in the direction of its second tangent) and *p2t1* (same bool but for the second point).

Now the graph is set up so that every contour point that is eligible for a connection has a vertex. Between every two contour points that can have a connection, an edge is added. The connection has to be in empty (black) image space. The weighted matching algorithm will be used so the weight of the edges is maximized (meaning more suitable connections have to have a greater weight than less suitable ones). The weight is calculated by subtracting the length of the connection from the greatest distance between two contour points of the one shape where this distance is smallest of the two shapes that are being connected. Edges with negative weight are not added to the graph.

Now the maximum-cardinality weighted matching algorithm of the *boost graph library* is used on the graph and results in the matching of the edges of the graph with the

most weight. Since for each edge, its vertices are known as well as in which tangents direction the connection has to be drawn in and for each vertex, the points position and its tangents are known, the connections can be drawn.

4.8 Adding additional points eligible for connections

This is only quickly reiterated in more detail than in Section 3.6 just to give a complete manual on how to implement the whole algorithmic pipeline in this Section:

After using Algorithm 2 to find eventually unconnected components, we have the unconnected components and we know how they are connected with each other (if they are connected with each other). For every unconnected island, we look at connections of each of its contour points. One tangent of every point is redirected as in "Adding connection points" 4.6 and they are taken through the whole pipeline of finding connections again. Only points with connections (direct or indirect) to the outside are now considered. The one with the highest calculated weight is taken. Now the components are connected to the outside. Empirically this was almost never needed since components are connected into multiple directions and therefore are very likely to get connected to the outside.

4.9 Rendering scene objects

There are three different types of connections that will be drawn. They are all added as new lines to the SVG-file.

4.9.1 Straight connection departing from corner point

This connection is between a corner point and an additional point on the other shape that was added to be able to add this connection (this bridge) to the graph. The additional point is not necessarily a corner point or even a contour point, since it is found by following the straight path in the direction of one of the starting points tangents until another shape is reached. This type of connection in most images will be the most common one because in most cases no other types of connections were possible.

The connection consists of four points that will be connected only by lines in the following order:

1. The starting point *p1*, which either is a corner point of the component we departed from or a contour point that was added as an additional point, since there were not enough corners on the shape.

2. The point *p2* found on the other shape by going in a straight line in the direction of one of the tangents of the starting point.

3. The point *p2p* which is found with the Algorithm 3 starting from *p2*.

4. The point *p1p* which is found with the Algorithm 3 starting from *p1*.

The last point is connected to the first point with a straight line and the whole quadrilateral is filled white.

4.9.2 Curved connection between two points

This connection is between two contour points (*p1* and *p2*) of different components in the image. We know the tangents at the contour points and two of them (one of each point) have an intersection point *ip* that lies between them. The distance for the weight is calculated by going from *p1* in a straight line to *ip* and from *ip* in a straight line to *p2*. This ensures that the connection will not be favored over a straight connection that would require less ink (which could happen if just the distance between the two points would be used to calculate the weight).

There are two cases for such a connection as explained already in "Curved connection" 3.7.2:

Both tangents to the same side (opposite directionalities)

The connection consists of four points that will be connected with two lines and two quadric Bézier curves, therefor two control points are needed:

1. The starting point *p1* of the first curve
2. The control point *ip1* of the first curve
3. The ending point *p2* of the first curve
4. The starting point *p2p* of the second curve. It is found with the Algorithm 3 starting from *p2*. It is connected to *p2* by a line
5. The control point *ip2* of the second curve.
6. The ending point *p1p* of the second curve. It is found with the Algorithm 3 starting from *p1*.

The last point is connected to the first point with a straight line and the whole quadrilateral is filled white.

Both tangents to opposite sides (same directionalities)

The connection consists of four points that will be connected with two lines and two quadric Bézier curves, therefor two control points are needed:

1. The starting point *p1* of the first curve

2. The control point *ip1* of the first curve
3. The ending point *p2p* of the first curve. It is found with the Algorithm 3 starting from *p2*
4. The start point *p2* of the second curve. It is connected to *p2p* by a line
5. The control point *ip2* of the second curve.
6. The ending point *p1p* of the second curve. It is found with the Algorithm 3 starting from *p1*.

The last point is connected to the first point with a straight line and the whole quadrilateral is filled white.

4.9.3 Straight connection between two points

This has the same two cases to consider as the curved connection, but there are no control points since there are no curves.

Both tangents to the same side (opposite directionalities)

The connection consists of four points that will be connected only by lines in the following order:

1. The first point *p1*, which either is a corner point of the component we departed from or a contour point that was added as an additional point, since there were not enough corners on the shape.
2. The second point *p2*, which either is a corner point of the component we departed from or a contour point that was added as an additional point, since there were not enough corners on the shape.
3. The point *p2p* which is found with the Algorithm 3 starting from *p2*.
4. The point *p1p* which is found with the Algorithm 3 starting from *p1*.

The last point is connected to the first point with a straight line and the whole quadrilateral is filled white.

Both tangents to opposite sides (same directionalities)

The connection consists of four points that will be connected only by lines in the following order:

1. The first point *p1*, which either is a corner point of the component we departed from or a contour point that was added as an additional point, since there were not enough corners on the shape.
2. The point *p2p* which is found with the Algorithm 3 starting from *p2*.
3. The second point *p2*, which either is a corner point of the component we departed from or a contour point that was added as an additional point, since there were not enough corners on the shape.
4. The point *p1p* which is found with the Algorithm 3 starting from *p1*.

The last point is connected to the first point with a straight line and the whole quadrilateral is filled white.

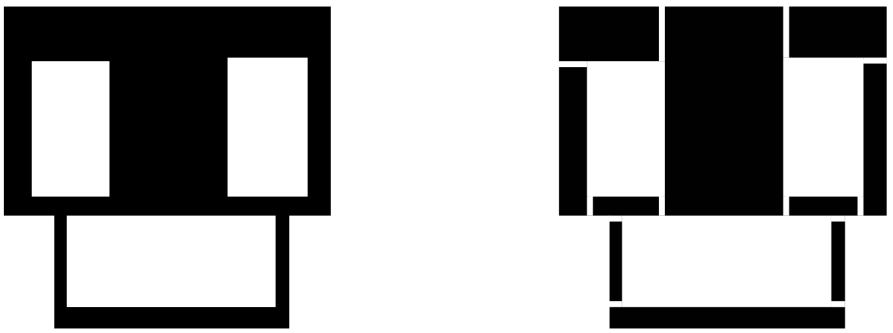
CHAPTER 5

Data examples and results

5.1 Data

In this section, some representative examples and the resulting stencil will be shown. The resolution is given by "width-height" and we try to approximate the complexity with the size of the SVG-file in kilobytes.

With straight and rectangular shapes the algorithmic pipeline works well and produces stencil as a human operator would probably produce them too. For all components, connections are found that fit the original design and also the symmetry is respected (since rectangular shapes are exactly portrayable in a raster image).

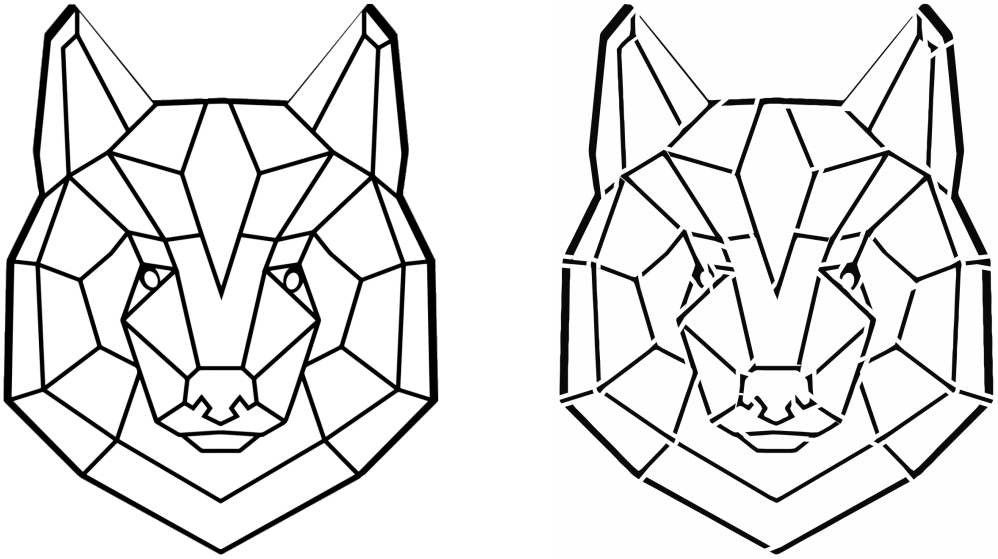


(a) The original image (b) The resulting stencil

Figure 5.1: Example of input with a lot of rectangular shapes
resolution: 750-660, size of SVG: 4,03kb

Also for polygonal shapes, the method works well. The connections do fit the original

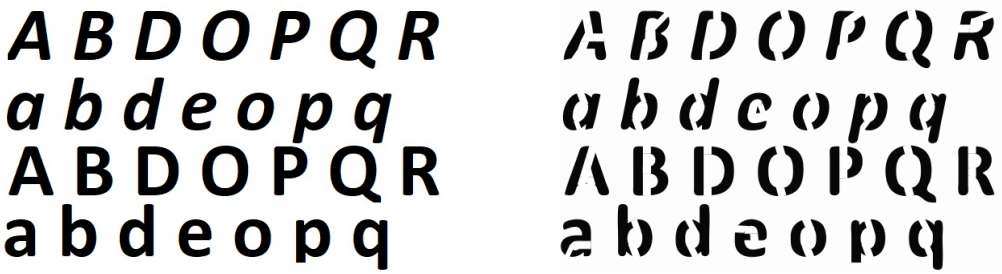
design very well, because they follow the curvature of it. The symmetry is not respected since the effect that we will mention in Section 5.2 is already in place. But In this case, the achieved degree of symmetry is enough.



(a) The original image (b) The resulting stencil

Figure 5.2: Example of input with only straight lines
resolution: 1000-1280, size of SVG: 42,6kb

Also with block letters (roman and italic), the algorithm works quite well. Only for letters with isolated regions in them, it is needed, so we omitted the other letters from the image. For some letters, it could be discussed if more than one connection is needed per isolated region. But all the letters are recognizable as the letter they represent and most of them look like the standard stencil version of the letters.



(a) The original image (b) The resulting stencil

Figure 5.3: Example of input with letters
resolution: 557-357, size of SVG: 32,5kb

The algorithmic pipeline also works well with curved lines (provided that the resolution of the image is high enough in relation to its complexity).

For this butterfly tattoo, all the bridges are in an extremely good place, except for the most top bridge, which is caused by a very small straight line in the contour that runs horizontally which is continued to make the bridge. It is longer than 2 pixels though, otherwise the tangents coming from it would be neglected. This is not a mistake caused by our stencil-making method but by a fault in the design.

(a) The original image (b) The resulting stencil

Figure 5.4: Example of input with curved lines
resolution: 398-515, size of SVG: 6,87kb

The bridges added for the dragon also are a satisfying result. Worthy to point out is how the small spirals at the dragons' legs were connected.

(a) The original image (b) The resulting stencil

Figure 5.5: Example of input with curved lines
resolution: 680-340, size of SVG: 41,6kb

5.1.1 Limitations

As suggested already from the data samples presented until now, the method has some limitations. We will explain them now in more detail.

For curved and/or complex shapes there is an important difficulty: If the resolution is relatively high, but the image has symmetry (symmetry axes in it) the connections drawn by the algorithmic pipeline will disturb that symmetry frequently since it does not consider it. That disturbance is easily detected by the human eye.

For this butterfly, the upper three outside components were connected okay. But when looking at the other white components, the connections drawn are not satisfying. This is due to symmetry that would be expected since the original design is symmetric. The added bridges disturb it.

(a) The original image (b) The resulting stencil

Figure 5.6: Example of input with one symmetry axis (down the middle) resolution: 280-190, size of SVG: 13,7kb

In this example of a mandala, we have three problems that lead to a very messy stencil. First of all the resolution is not high enough in comparison to the complexity of the image. Therefore in the vectorization very inexact positions and tangents were found. Secondly, there is a total of eight symmetry axes that are not respected by the bridges drawn to create a stencil. And lastly, the long wedges in the middle are not suitable to be connected by continuing a tangent over their sharpest corner with a bridge. Instead, it would be better to find the main axis of the wedge and continue in that direction. More on this last point in Section 6.

(a) The original image (b) The resulting stencil

Figure 5.7: Example of input with lots of symmetry axes and low resolution relative to its complexity
resolution: 1024-768, size of SVG: 114kb

In conclusion, there is also a third case where the results produced by the algorithm may vary in appeal. Cases where also a human operator would not be so sure where to put the connections.

5.2 Discussion of results

The approach taken in this paper works for some cases, but it has many limitations and there are cases where there exist some problems (missing symmetry, inexact vectorization). Also to produce acceptable results a high resolution of the input image is crucial, so that the vectorization has high accuracy. But in that case the algorithmic pipeline slows down a lot, since the complexity of the algorithms used (specifically floodfill, and the search for possible connections) is relatively high.

5.2.1 Low-resolution input image

The lower the resolution, the less exact the image can be vectorized, in respect to the shapes its artist intended. As mentioned earlier the inverse operation of vectorization, rasterization is not without loss. This means that the original vector graphic can not be perfectly recuperated from its raster image (in most cases). So naturally, the corner points and tangents found on the vectorized graphic are not exactly right and the whole process from there can have effects that are not wanted.

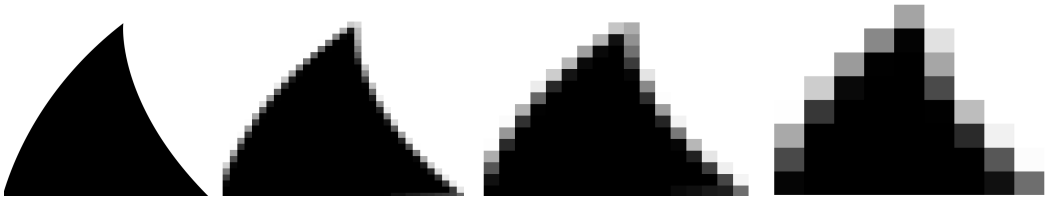


Figure 5.8: corner in different resolutions
from left to right: "vector graphic", raster graphic, raster graphic with half resolution, raster graphic with quarter resolution

A higher resolution produces better stencils, but in turn, costs a lot more computation time.

5.2.2 Symmetry axes that are not aligned with the raster grid

As seen in the data examples stencils made out of images with symmetry axes (e.g. mandalas) in various directions do not look very appealing. This is because one would expect a certain type of bridge that is made to close a certain gap would also be made on a mirrored, translated and/or rotated version of that gap. This is only the case if it is mirrored on a horizontal or vertical symmetry axis, translated by an integer multiple of the width/height of one pixel and rotated by an integer multiple of 90 degrees. Because otherwise it will be sliced up differently by the rectangular raster grid and the vectorization will give slightly different positions and directions. The different solutions for similar connections are very apparent with input data where there are a lot of symmetry axes.

CHAPTER 6

Conclusion and future work

6.1 Conclusion

While pipeline does work fine for some data examples, it still has a lot of limitations, that are not easily solved without adding a lot of algorithmic complexity. The question is worth asking if it would not be more sensible starting with a different approach that is not so reliant on an exact vectorization and is not bottom-up with many different solutions for different cases to produce a stencil where all components are connected.

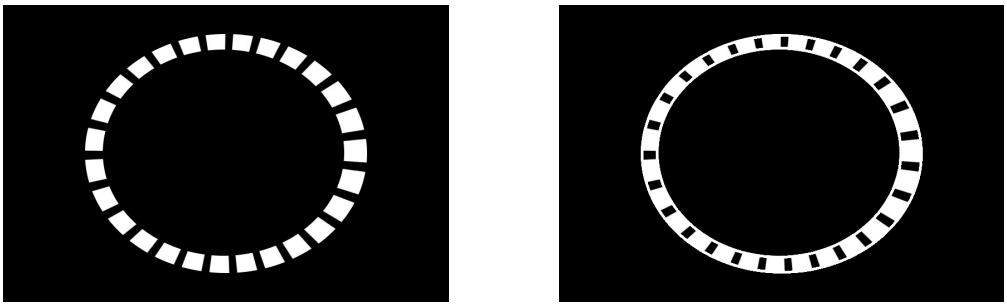
One other way to solve the problem would be to replicate the way of creating a stencil used in the earlier mentioned paper "Semi-Automatic Stencil Creation through Error Minimization" [BRO08] which does not try to make bridges in the directions of the gradients of contours but makes them orthogonally in a manner the minimizes the added ink. In the paper, they depart from 3D-meshes and play with lighting to get appealing stencils. The way of drawing the bridges could be replicated, but starting from a binary input, not from a 3D-mesh.

If on the other hand, we want to keep the idea of drawing the bridges in a way that preserves the slopes of the gradient of the shapes we could try out other approaches:

- **similar approach**: but start from vector images. The finding of the shapes does not have to be done with floodfilling. An SVG-file also describes the positions of the components in the image. Of course, the input is more restricted as only vector images are allowed. Also using skeletonization for certain shapes to follow their main directions with the bridges drawn could be contemplated (That would make sense for example for Figure 5.7).
- **Resizing, rotating, translating of components**: Connectedness of an isolated region in the image could in many cases be achieved by resizing (also only in one direction), rotating or translating it. This would avoid having to make additional

bridges to ensure connectivity, but it would also alter the original intention of the design.

- **shape fitting**: fit a set of basic geometric shapes into the black parts (the parts that will be cut out) of the binary image. The contours of the fitted shapes are extended into the white components to get a certain thickness.



(a) The original image (b) The resulting stencil

Figure 6.1: What shape fitting could look like for a simple example: two circles are fitted on the black parts (around the white shapes and inside)

- **curve tracing** As explained by Yuki and Takeo Igarashi [II10] a continuous line that intersects itself results in an isolated region that needs to be connected for a stencil. If at the intersection one part of the line is drawn as "going over" the other part of the line (by giving it white edges) the isolated region is connected. In the work of the paper, the drawn curves are known since it is a drawing tool. If this idea is tried to replicate for any binary image the curves need to be traced and then decided which part goes over the other at an intersection. A way to trace lines on images is shown by Everts and colleagues [EBRI09].

Figure 6.2: Celtic design
a stencil where the curve goes over itself at intersections

- **A combination of the two**: shape fitting and curve tracing could both be used if parts of the image are more suitable for one or the other.

List of Figures

List of Algorithms

Bibliography

[BRO08] Jonathan Bronson, Penny Rheingans, and Marc Olano. Semi-automatic stencil creation through error minimization. In *Proceedings of the 6th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '08, page 31–37, New York, NY, USA, 2008. Association for Computing Machinery.

[Con97] Russell G Congalton. Exploring and evaluating the consequences of vector-to-raster and raster-to-vector conversion. *Photogrammetric Engineering and Remote Sensing*, 63(4):425–434, 1997.

[EBRI09] Maarten H Everts, Henk Bekker, Jos BTM Roerdink, and Tobias Isenberg. Depth-dependent halos: Illustrative rendering of dense line data. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1299–1306, 2009.

[HF98] Radim Halır and Jan Flusser. Numerically stable direct least squares fitting of ellipses. In *Proc. 6th International Conference in Central Europe on Computer Graphics and Visualization. WSCG*, volume 98, pages 125–132. Citeseer, 1998.

[II10] Y. Igarashi and T. Igarashi. Holly: A drawing editor for designing stencils. *IEEE Computer Graphics and Applications*, 30(4):8–14, 2010.

[JCT+15] Arjun Jain, Chao Chen, Thorsten Thormählen, Dimitris Metaxas, and Hans-Peter Seidel. Multi-layer stencil creation from images. *Computers Graphics*, 48:11–22, 2015.

[LCW+18] Lijuan Liu, Yang Chen, Pinhao Wang, Yizhou Liu, Caowei Zhang, Xuan Li, Cheng Yao, and Fangtian Ying. Papercut: Digital fabrication and design for paper cutting. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI EA '18, page 1–6, New York, NY, USA, 2018. Association for Computing Machinery.

[MZZ10] Meng Meng, Mingtian Zhao, and Song-Chun Zhu. Artistic paper-cut of human portraits. In *Proceedings of the 18th ACM International Conference on Multimedia*, MM '10, page 931–934, New York, NY, USA, 2010. Association for Computing Machinery.

[PSP07] D. Peng, S. Sun, and L. Pan. Research on chinese paper-cut cad system. In *Fourth International Conference on Image and Graphics (ICIG 2007)*, pages 892–896, 2007.

[Qui03] A. Quint. Scalable vector graphics. *IEEE MultiMedia*, 10(3):99–102, 2003.

[SLL02] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. *The boost graph library: user guide and reference manual*. Addison-Wesley, 2002.

[Smi79a] Alvy Ray Smith. Tint fill. *SIGGRAPH Comput. Graph.*, 13(2):276–283, August 1979.

[Smi79b] Alvy Ray Smith. Tint fill. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '79, page 276–283, New York, NY, USA, 1979. Association for Computing Machinery.

[WH04] Martin Weber and B Herzog. Autotrace, 2004.