Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Master's thesis

Shape Decomposition for Multi-channel Distance Fields

Bc. Viktor Chlumský

Supervisor: Ing. Ivan Šimeček, Ph.D.

5th May 2015

Acknowledgements

I would like to thank my supervisor, Doctor Ivan Šimeček, for guidance, and my friend, Tomáš Báča, for providing general advice on writing a scientific thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for nonprofit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on 5th May 2015 .....

Czech Technical University in Prague
Faculty of Information Technology
© 2015 Viktor Chlumský. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Chlumský, Viktor. *Shape Decomposition for Multi-channel Distance Fields.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Tato práce zkoumá možnosti vylepšení populární techniky vykreslování textu, která je hojně používaná ve 3D aplikacích a počítačových hrách. Předkládá univerzální a efektivní metodu konstrukce vícekanálového pole vzdáleností pro vektorové obrazce, zejména znaky písem, a popisuje jeho použití při vykreslování se zvýšenou kvalitou.

Klíčová slova Pole vzdáleností, vykreslování textu, Bézierovy křivky.

Abstract

This work explores the possible improvements to a popular text rendering technique widely used in 3D applications and video games. It proposes a universal and efficient method of constructing a multi-channel distance field for vector-based shapes, such as font glyphs, and describes its usage in rendering with improved fidelity.

Keywords Distance field, text rendering, Bézier curves.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the important problems in computer graphics is text rendering. There are many different techniques, each specialized for a different scenario. When the text is static, it is not a problem to take time and pre-render it with high precision, but if its transformation, perspective, or the text itself changes rapidly, a specialized technique has to be employed. A popular one used in real-time graphics relies on storing the character shapes, or glyphs, in structures called signed distance fields. [11, 6] I will present an improved version of this technique that combines multiple distance fields in a way that significantly improves the quality of the image.

In this chapter, I will explain how the original technique works, demonstrate the potential of my improvement, and discuss alternate methods.

1.1 Signed distance fields

In order to understand the signed distance field rendering technique, we must first establish what a signed distance field, or SDF for short, is and how it works.

A signed distance field in general is the result of a signed distance transformation applied to a subset of N-dimensional space. It maps each point P of the space to a scalar signed distance value. A signed distance is defined as follows: If the point P belongs to the subset, it is the minimum distance to any point outside the subset. If it does not belong to the subset, it is minus the minimum distance to any point of the subset.

For our purposes, we will only use a specific type of signed distance field, one that uses Euclidean distance, and whose domain is the two-dimensional space only. The vector shape we wish to render will be the subset in question. Additionally, our signed distance fields will be represented as rectangular grids with the signed distances specified only at a finite number of discrete points, and therefore not exact.

The distance field in this form will serve as the simplified representation of the vector shape, from which it can be approximately reconstructed. The idea will be best shown on an example. Assume that we want to encode the shape in Figure 1.1.

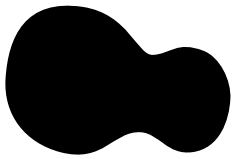


Figure 1.1: A two-dimensional shape.

The diagram in Figure 1.2 is a visual representation of the corresponding distance field in a 16×16 grid. Note that each value is the distance to the closest edge point as illustrated by the lines, and that outside values are negative (blue) while inside values are positive (red).

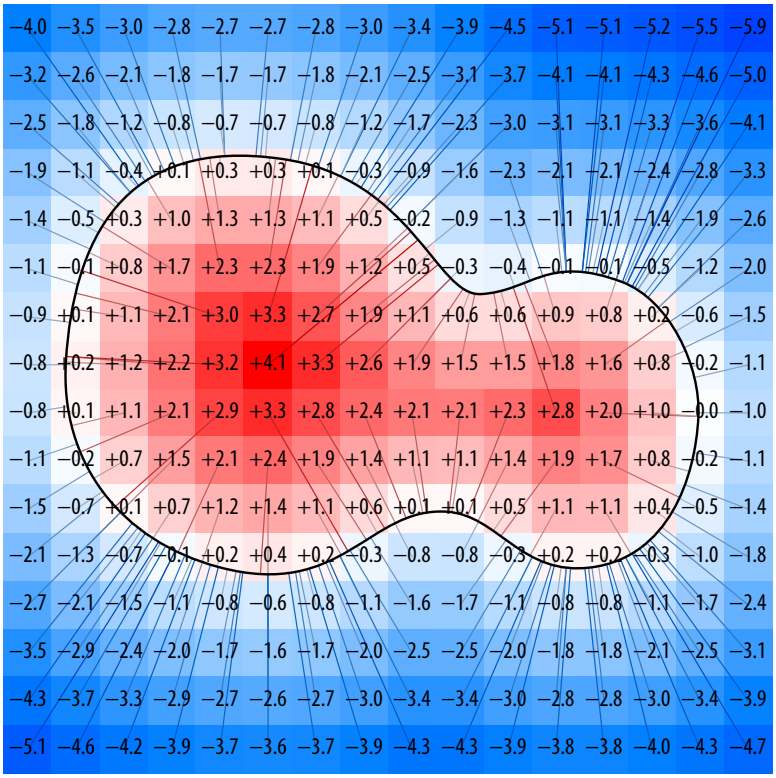


Figure 1.2: The shape's signed distance field.

1.1.1 Information carried by the distance field

Clearly, the conversion to a grid of 16×16 scalar values is lossy, so let's examine what information about the original shape this representation holds. Imagine plotting a circle for each point of the grid, and using the absolute value of the signed distance as its radius. The shape's edge must touch each of these circles at (at least) one point, but not intersect it. Therefore, the collective area of the negative circles lies strictly outside the shape and the area of the positive ones inside. Only the space that does not coincide with any circle is uncertain. That space is shown in red in Figure 1.3.

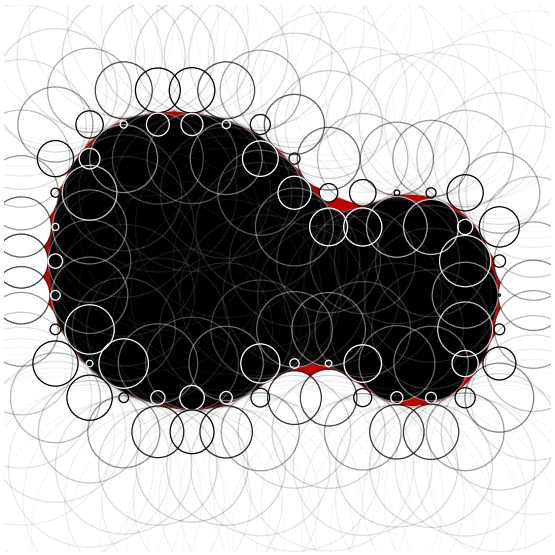
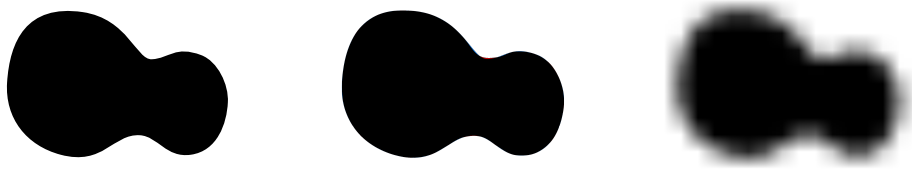


Figure 1.3: Analysis of the distance field's data.

Although the edge of the shape could lead anywhere through the red area, by taking the shortest or smoothest route, it can still be reconstructed without problems. That is, of course, assuming the shape is relatively smooth. The distance field representation would fail to capture any small details of the edge's winding at this resolution, which is however usually not a problem for text, since glyphs tend to consist of smooth strokes.

1.1.2 Benefits of distance field representation

The primary advantage of signed distance fields is the fact that the values change very smoothly and predictably, even in areas near the edges, where deciding which pixels belong inside the shape is hardest. Therefore, using simple interpolation, the signed distance field grid can be sampled at a much higher resolution, and still provide a good approximation of the actual signed distances throughout the plane. By only considering the sign of the sampled distances, the image can be reconstructed at any resolution (Figure 1.4b).



(a) (b) (c)

Figure 1.4: Demonstration of reconstructing the original shape (a) from a low resolution distance field (b) and from a low resolution image (c).

As Figure 1.4 shows, the reconstruction is almost perfect despite the distance field's low resolution. The way this works is demonstrated in Figure 1.5, which shows a 3D rendering of a height map generated from the distance field. Here, only the intersections of the white grid hold exact signed distances (heights). An interpolation has been used for the rest of the surface.

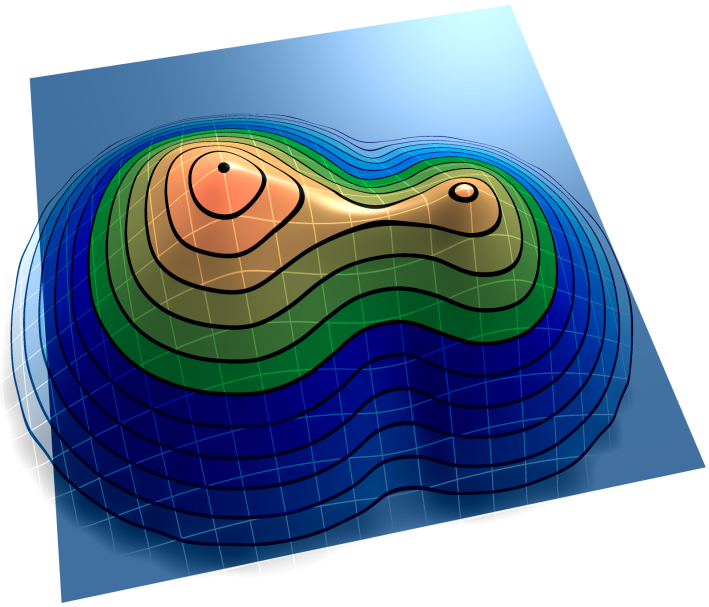


Figure 1.5: A 3D representation of the signed distance field.

Imagine the values in the signed distance field as height values above water level. The shape then forms an island or a group of islands sticking above water. The "coastline" however is perfectly smooth this way.

This means that shapes represented using a signed distance field are infinitely scalable without any pixelation.

1.1.3 The problem

The problem is that "smooth" isn't always what we're looking for. Shapes often have sharp corners, which cause irregularities in the distance field.

Figure 1.6: A shape with sharp corners (left) and its reconstruction from a low resolution distance field (right).

Because the technique relies on interpolation, which provides a good approximation only where the rate of change is more or less constant, the corners cannot be reconstructed correctly, and will instead look rounded or chipped in the resulting image. Figure 1.6 displays one such case.

1.1.4 Multi-channel distance fields

A possible solution to the case in Figure 1.6 is to divide it into two smooth shapes, as shown in Figure 1.7, and create two separate distance fields. When reconstructing the image, one can first reconstruct the two auxiliary subshapes, and afterwards fill only those pixels that belong in both.

Figure 1.7: A possible decomposition of the shape into a union of two round shapes.

The distance fields are usually stored as monochrome images. However, image files have the ability to hold at least 3, and sometimes 4 or more color channels, making it natural to encode the two separate distance fields as one image, where each channel holds one of them.

The goal of this thesis is to explore the possibilities of combining multiple distance fields to improve the quality of rendering corners.

1.2 Practical viability

To verify the potential of the effort, I have performed such decomposition manually on a string of three letters. This is shown in Figure 1.8.

(a) A string of letters. (b) An ad-hoc decomposition.

Figure 1.8: Multi-channel decomposition of several letters.

I have generated both single-channel and multi-channel signed distance fields using this decomposition, and reconstructed the image of the string from each. The results can be seen in Figure 1.9.

(a) Using single channel. (b) Using decomposition.

Figure 1.9: A reconstuction of the letters from low resolution distance fields.

Both versions are perfectly readable, but the appearance of the font is very different and much less sharp using the basic single channel distance field, while when using the improved method, the result is almost indistinguishable from the original (Figure 1.8a). This difference in quality of reconstruction shown on a real-life scenario should prove the practical viability of using a multi-channel distance field in place of a conventional one.

1.3 State of the art

There aren't many advanced and widely used technologies for dynamic text rendering in real-time graphical applications and games. In most cases, a simple raster representation of the glyphs is sufficient. Another common way is to draw the glyphs in the form of triangle meshes.

Where both high quality and performance is demanded, the signed distance field technology, which was first proposed in 2006 [11], is still considered the state-of-the-art technology. Some of its many advantages include:

- The rendering process is extremely efficient in a hardware-accelerated context, comparable to basic textured drawing. It therefore requires no pre-rasterization step and is ideal for dynamic text, whose transformation changes between frames.
- Rendering quality is high and scalable – if the source SDF is large enough in proportion to the target text size, the result will be almost indistinguishable from exact methods. Otherwise, a larger SDF can be used.
- It can be used to display text at an arbitrary resolution with no blurring, pixelation, or other disruptive artifacts.
- Its implementation (in OpenGL for example) is relatively easy in comparison with other methods that provide similar image quality.
- It is capable of inexpensive and easy to implement anti-aliasing.
- It holds information about edge distance, which can be used to easily implement plenty of inexpensive visual effects, such as outlines or shadows, or even animations.

Of course, it is an approximate method and therefore isn't perfect. It cannot capture features smaller than the resolution of the SDF's grid, and thin strokes are a problem too. Another issue is its tendency to round corners, as mentioned before.

Even then, no other known method can match all if its advantages. For this reason, combined with its simplicity, it is currently a very popular method for displaying all text in a 3D application, including static text, for which exact methods may have better results, but are also more complex to implement in a real-time graphics context.

1.3.1 Alternate methods

Apart from the simplest, yet widely used methods, of using raster glyphs or triangle meshes, there are very few alternatives to signed distance fields used in practice. Let's summarize the most significant ones and and compare them to the distance field technique.

1.3.1.1 Raster glyphs

This is the simplest method, where character glyphs are treated like regular images. When the glyphs are magnified, they will become blurry or pixelated.

Compare (b) and (c) in Figure 1.4 as a good example of the quality difference. It is only marginally faster than the distance field method, but this is negated by the need for larger image resolution.

1.3.1.2 Triangle meshes

The next possibility is converting the vector glyphs into triangle meshes. However, since modern font formats define their glyph shapes using parametric curves, this conversion has to be approximate, and depending on the original complexity of the glyph's shape and the desired quality, the amount of triangles can become unpractically high, especially considering that a large amount of characters may be needed to be displayed at the same time. Therefore, the quality to performance ratio is generally much worse than what distance fields have to offer.

1.3.1.3 Exact vector rendering

A more advanced version of the triangle mesh rendering method is the possibility of using programmable shaders to draw parametric curves exactly. This method is described in [9].

It is an exact method, and therefore provides higher quality than distance fields, but its performance cost is even higher than that of the triangle mesh method. The glyphs also have to be divided into a large amount of triangles, but additionally, complex shader calculations must be performed in the rendering process.

1.3.1.4 Advanced texture-based methods

There are other more advanced niche techniques which attempt to eliminate the need for large triangle meshes and encode the geometry of the shapes in textures. One such technology is [12], which reconstructs the shape using functions whose coefficients it looks up from texture data. Another one is the experimental library called GLyphy [4], which approximates the shape geometry using circular arcs, and also stores their definitions as texture data.

All of these methods offer higher quality than distance fields, however, they require complex shader computations, including multiple texture lookups, making them far less efficient. Because of this, they are rarely used in practice.

1.3.2 Pre-existing solution

Although Valve hinted at the possible improvement of adding more channels to the distance field in 2007 [6], a working solution for multi-channel distance field construction has not been published until November 2014 – after I started working on this thesis. Only limited information about this solution is available [10], but it is clear the author took an approach very different to my own.

The author himself mentions problems with artifacts and lower robustness, as well as having to treat several special cases. He also uses four color channels, while my method only needs three for the same task.

Based on this limited information about his method, I believe that my final solution is more effective and more reliable, as well as faster and simpler.

Chapter 2

Preliminaries

In this chapter, I will cover the basic mathematical concepts required to understand the problems addressed in this work. We will predominantly focus on the geometry of the vector shapes, and their conversion to distance fields.

2.1 Bézier curves

The font glyphs and other vector shapes will be described by outlines that consist of line segments and Bézier curves. A line segment is trivial, but it can also be defined as a sort of "Bézier curve" of order 1, which will be useful later on. The parametric definition of a line segment between points P_0 and P_1 is:

$$B_1(t) = (1-t)P_0 + tP_1 \tag{2.1}$$
$$= P_0 + t(P_1 - P_0) \tag{2.2}$$

The first formulation is a weighted average between the two endpoints, while the second adds a portion of the line's vector to its initial point. A line segment is simply the set of all points $B_1(t)$ for all $t \in \langle 0, 1 \rangle$.

Bézier curves are a class of parametric curves. A Bézier curve of order 2, also known as a quadratic Bézier curve, is the simplest one. Like the line segment, it also has two endpoints, but additionally a single control point, which lies outside the curve, but affects its path. This is illustrated in Figure 2.1.

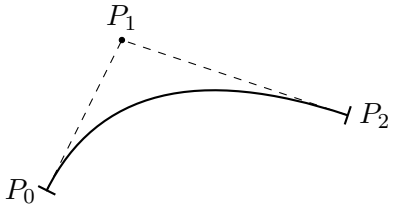


Figure 2.1: A quadratic Bézier curve.

The parametric function generating the points along the quadratic Bézier curve with endpoints P_0 and P_2 and control point P_1 is: [5]

$$B_2(t) = (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2 \tag{2.3}$$

$$= P_0 + 2t(P_1 - P_0) + t^2(P_2 - 2P_1 + P_0) \tag{2.4}$$

Same as with the line segment, all points $B_2(t)$ for any t in the range $\langle 0, 1 \rangle$ lie on the Bézier curve.

A cubic Bézier curve (of order 3) is similar, but has two control points, as illustrated in Figure 2.2.

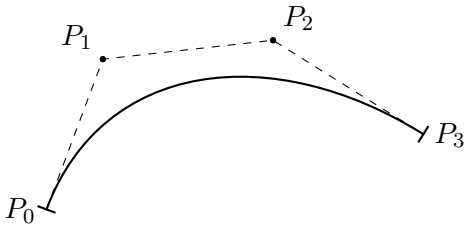


Figure 2.2: A cubic Bézier curve.

The parametric function generating the points along the cubic Bézier curve with endpoints P_0 and P_3 and control points P_1 and P_2 is: [5]

$$B_3(t) = (1-t)^3 P_0 + 3(1-t)^2 tP_1 + 3(1-t)t^2 P_2 + t^3 P_3 \tag{2.5}$$

$$= P_0 + 3t(P_1 - P_0) + 3t^2(P_2 - 2P_1 + P_0) + t^3(P_3 - 3P_2 + 3P_1 - P_0) \tag{2.6}$$

Bézier curves of higher order will not be used.

2.1.1 Limitations

Unfortunately not any curve can be represented exactly as a Bézier curve or spline. For example, it is not possible to describe a circle exactly, no matter the curve's order. [13] The problem is comparable to converting a sine function to a polynomial. It is not possible exactly, but an approximation, such as the Taylor polynomial can be used.

A higher order Bézier curve cannot be described exactly by curves of lower order, so it is not possible to convert cubic Bézier curves to quadratic ones without error. The opposite conversion is possible exactly, but since cubic curves are more complex, and working with them is harder, this conversion wouldn't be very practical. Therefore, we will work with both quadratic and cubic Bézier curves.

2.2 Locating sharp corners

The vector path of the shape's outline has many vertices, but only some of them may be in fact corners. In Figure 2.3, only point C is a sharp corner, while B and D smoothly connect two curves into a spline.

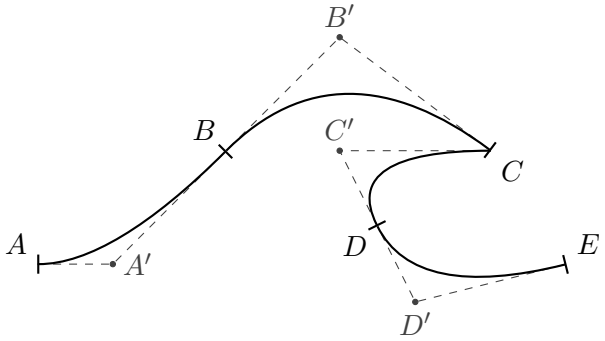


Figure 2.3: A path composed of quadratic Bézier curves.

To find the corners, it is necessary to detect whether the curve's direction changes abruptly at the endpoint. The direction vector can be obtained from the curve's derivative at any point:

$$\frac{\mathrm{d}B_1}{\mathrm{d}t}(t) = P_1 - P_0 \tag{2.7}$$
$$\frac{\mathrm{d}B_2}{\mathrm{d}t}(t) = 2t(P_2 - 2P_1 + P_0) + 2(P_1 - P_0) \tag{2.8}$$
$$\frac{\mathrm{d}B_3}{\mathrm{d}t}(t) = 3t^2(P_3 - 3P_2 + 3P_1 - P_0) + 6t(P_2 - 2P_1 + P_0) + 3(P_1 - P_0) \tag{2.9}$$

For a line segment (as defined by Equation 2.2), the direction is obviously always the vector $P_1 - P_0$. At the beginning of a curve, where $t = 0$, the direction is also $P_1 - P_0$. This may be intuitive from Figures 2.1 and 2.2, but let's verify this:

$$\frac{\mathrm{d}B_1}{\mathrm{d}t}(0) = \frac{\mathrm{d}B_1}{\mathrm{d}t}(1) = P_1 - P_0 \tag{2.10}$$
$$\frac{\mathrm{d}B_2}{\mathrm{d}t}(0) = 2(P_1 - P_0) \tag{2.11}$$
$$\frac{\mathrm{d}B_2}{\mathrm{d}t}(1) = 2(P_2 - 2P_1 + P_0) + 2(P_1 - P_0) = 2(P_2 - P_1) \tag{2.12}$$
$$\frac{\mathrm{d}B_3}{\mathrm{d}t}(0) = 3(P_1 - P_0) \tag{2.13}$$
$$\begin{aligned}\frac{\mathrm{d}B_3}{\mathrm{d}t}(1) &= 3(P_3 - 3P_2 + 3P_1 - P_0) + 6(P_2 - 2P_1 + P_0) + 3(P_1 - P_0) \\ &= 3(P_3 - P_2)\end{aligned} \tag{2.14}$$

Using the derivatives, we have confirmed that the vector to the adjacent control point in fact represents the curve's direction vector at the endpoint. Since we are only interested in the direction, the constant factors can be

ignored. Knowing the direction vector of both curves meeting at the endpoint, its "sharpness" can be computed using the cross product.

The cross product of two vectors is not normally defined in two-dimensional space, but since this operation will be needed on multiple occassions, I will define it as the following operation between two vectors, $\vec{a}$ and $\vec{b}$, that returns a scalar value:

$$\vec{a} \times \vec{b} \stackrel{\text{def}}{=} a_x b_y - a_y b_x \tag{2.15}$$

The core property of this operation is that the result is 0 for any two parallel vectors (which also implies Equation 2.16). It can be thought of as an indicator of the vectors' difference in direction. The sign of the result also tells us if the vectors are in clockwise order. Other properties of this operation include:

$$\vec{a} \times \vec{a} = 0 \tag{2.16}$$

$$\vec{a} \times \vec{b} = -(\vec{b} \times \vec{a}) \tag{2.17}$$

$$(k_1\vec{a}) \times (k_2\vec{b}) = k_1 k_2 (\vec{a} \times \vec{b}) : k_1, k_2 \in \mathbb{R} \tag{2.18}$$

So, to check if the connection between the curves $B_A(t)$ and $B_B(t)$ is a corner, one must find if

$$\tfrac{\mathrm{d}B_A}{\mathrm{d}t}(1) \times \tfrac{\mathrm{d}B_B}{\mathrm{d}t}(0) \stackrel{?}{=} 0. \tag{2.19}$$

For the vertices B, C, and D respectively from Figure 2.3, this would be:

$$(B - A') \times (B' - B) = 0 \tag{2.20}$$

$$(C - B') \times (C' - C) \neq 0 \tag{2.21}$$

$$(D - C') \times (D' - D) = 0 \tag{2.22}$$

Since we are going to use floating point arithmetics, the result will almost never be exactly zero, and sometimes it may be desirable to treat very obtuse corners as smooth too. Therefore, in practice, a maximum angle $\alpha \leq \pi$ should be chosen and used as a threshold to detect sufficiently sharp corners only. To achieve this, the direction vectors must be normalized first, and the resulting formula would be:

$$\left| \frac{\frac{\mathrm{d}B_A}{\mathrm{d}t}(1)}{\|\frac{\mathrm{d}B_A}{\mathrm{d}t}(1)\|} \times \frac{\frac{\mathrm{d}B_B}{\mathrm{d}t}(0)}{\|\frac{\mathrm{d}B_B}{\mathrm{d}t}(0)\|} \right| \stackrel{?}{\leq} \sin\alpha \tag{2.23}$$

To also account for the possibility that the two adjacent edges are in fact parallel, making the cross product zero, but form a 180 degree turn at the corner, which is valid for a shape composed of curves, additionally, we also have to check the dot product of the direction vectors, making sure it is positive, meaning the vectors don't have opposite directions:

$$\tfrac{\mathrm{d}B_A}{\mathrm{d}t}(1) \cdot \tfrac{\mathrm{d}B_B}{\mathrm{d}t}(0) \stackrel{?}{>} 0. \tag{2.24}$$

2.3 Point – edge segment distance

Because a signed distance field is an array of signed distances from points in the grid to the shape's outline, we must be able to compute these distances. The shape's outline consists of edge segments, which are one of a line segment, a quadratic Bézier curve, or a cubic Bézier curve. These will be collectively referred to as *segments*.

Let's start by establishing the mechanism of determining the minimum Euclidean distance between a point and the three types of edge segments.

2.3.1 Point – line segment

This is obviously the easiest problem of the three, and an analytic solution is well known. The algebraic formula for the distance between a point P and a line (P_0, P_1) is:

$$\text{distance}(P, (P_0, P_1)) = \frac{|P \times (P_1 - P_0) - P_0 \times P_1|}{\|P_1 - P_0\|} \tag{2.25}$$

For a line *segment* though, the computation is a little more complicated. The formula only yields the perpendicular distance, which is fine for point A in Figure 2.4, but for point B, the minimum distance is actually the distance from endpoint P_1, whereas the previous formula would output the distance from B's perpendicular projection onto the infinite line, which is closer, but does not belong to the line segment.

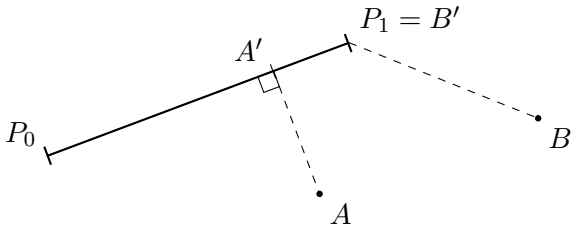


Figure 2.4: Finding the minimum distance between a point and a line segment.

What we need to do instead is find the parameter t from the parametric equation of the line segment (Equation 2.2), for which the distance is minimal. If it doesn't lie on the line segment, the closer endpoint will be used instead.

There are two possible ways to find the value of t. Either, we minimize the distance $\|B_1(t) - P\|$ by finding the root of its derivative (Equation 2.26), or, knowing that the direction to the point P is always perpendicular to the line at $B_1(t)$, we only need to find where the dot product is zero (Equation 2.27).

$$\tfrac{\mathrm{d}}{\mathrm{d}t}\|B_1(t) - P\| = 0 \tag{2.26}$$

$$(B_1(t) - P) \cdot \tfrac{\mathrm{d}B_1}{\mathrm{d}t}(t) = 0 \tag{2.27}$$

By simplifying either formula, we arrive at:

$$t = \frac{(P - P_0) \cdot (P_1 - P_0)}{(P_1 - P_0) \cdot (P_1 - P_0)} \tag{2.28}$$

After obtaining the parameter t, we can check that it lies in the range $\langle 0, 1 \rangle$ and therefore on the line segment. If it does not, we simply clamp the value to this range (set it to either 0 or 1, whichever is closer to the original result). Since $B_1(0) = P_0$ and $B_1(1) = P_1$, this will yield the closer endpoint as planned. The final distance can then be obtained simply as the distance between $B_1(t)$ and P.

2.3.2 Point – Bézier curve

The very same approach can be also applied to Bézier curves. Since they are continuous, it is also true that the minimum distance is perpendicular. The only difference is, that there could be multiple perpendiculars leading to P, as well as multiple extremes of the distance, including maxima.

The generalized equations for a Bézier curve of an arbitrary degree n are:

$$\tfrac{\mathrm{d}}{\mathrm{d}t} \| B_n(t) - P \| = 0 \tag{2.29}$$

$$(B_n(t) - P) \cdot \tfrac{\mathrm{d}B_n}{\mathrm{d}t}(t) = 0 \tag{2.30}$$

Before continuing, let's define some auxiliary vectors to simplify the formulae.

$$\vec{p} = P - P_0 \tag{2.31}$$

$$\vec{p_1} = P_1 - P_0 \tag{2.32}$$

$$\vec{p_2} = P_2 - 2P_1 + P_0 \tag{2.33}$$

$$\vec{p_3} = P_3 - 3P_2 + 3P_1 - P_0 \tag{2.34}$$

Now, the Bézier curves can be expressed as:

$$B_2(t) = t^2\vec{p_2} + 2t\vec{p_1} + P_0 \tag{2.35}$$

$$B_3(t) = t^3\vec{p_3} + 3t^2\vec{p_2} + 3t\vec{p_1} + P_0 \tag{2.36}$$

For a quadratic Bézier curve, the formula from Equation 2.30 is then:

$$(t^2\vec{p_2} + 2t\vec{p_1} - \vec{p}) \cdot (2t\vec{p_2} + 2\vec{p_1}) = 0 \tag{2.37}$$

$$(\vec{p_2} \cdot \vec{p_2})t^3 + 3(\vec{p_1} \cdot \vec{p_2})t^2 + (2\vec{p_1} \cdot \vec{p_1} - \vec{p_2} \cdot \vec{p})t - \vec{p_1} \cdot \vec{p} = 0 \tag{2.38}$$

Since all of the vectors are arbitrary, this is a general cubic equation. Such an equation can be solved analytically, using Cardano's formula [17].

It can only degenerate into a lower order polynomial if $\vec{p_2} = \vec{0}$. In that case, the quadratic term will also be zero, and it becomes a linear equation.

Using Equation 2.33, this only happens when $P_1 = \frac{1}{2}(P_0 + P_2)$, meaning the control point P_1 lies directly in the middle between the endpoints, resulting in a straight line. This is consistent with the line segment scenario.

For a cubic Bézier curve, the formula from Equation 2.30 is:

$$(t^3\vec{p_3} + 3t^2\vec{p_2} + 3t\vec{p_1} - \vec{p}) \cdot (3t^2\vec{p_3} + 6t\vec{p_2} + 3\vec{p_1}) = 0 \tag{2.39}$$

$$\begin{aligned}(\vec{p_3} \cdot \vec{p_3})t^5 + 5(\vec{p_2} \cdot \vec{p_3})t^4 + (4\vec{p_1} \cdot \vec{p_3} + 6\vec{p_2} \cdot \vec{p_2})t^3 \\ + (9\vec{p_1} \cdot \vec{p_2} - \vec{p_2} \cdot \vec{p})t^2 + (3\vec{p_1} \cdot \vec{p_1} - 2\vec{p_2} \cdot \vec{p})t - \vec{p_1} \cdot \vec{p} = 0\end{aligned} \tag{2.40}$$

In this case, we must find the roots of a general quintic equation. Unfortunatelly, it has been proven that an analytical solution is impossible by Abel [16], so a numerical approximation is necessary.

For both the quadratic and the cubic Bézier curve, all roots t in the range $\langle 0, 1\rangle$ must be inspected, as well as the endpoints $t = 0$ and $t = 1$. For each, the distance $\|B_n(t) - P\|$ must be computed, and the minimum shall be the result.

2.3.3 Signed distance

Now that we have found the minimum distance, we also have to decide whether it is positive or negative. By distinguishing the two endpoints of an edge segment, the line segment or curve can be considered oriented. That way, one side of the segment can be designated as the inside, and the other as the outside. Points on the segment's inside will then be in a positive distance, and the rest in a negative distance, as illustrated in Figure 2.5.

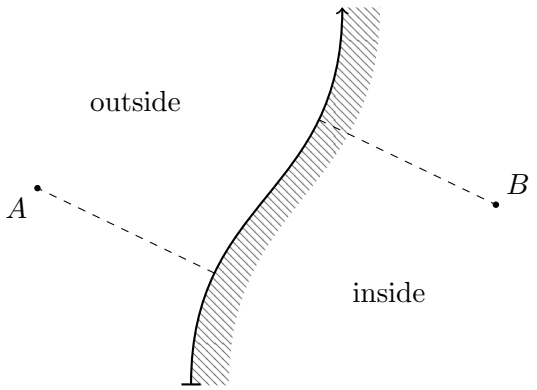


Figure 2.5: Negative (A) and positive (B) distance from a Bézier curve.

We already know that the direction of the curve at t is $\frac{\mathrm{d}B_n}{\mathrm{d}t}(t)$, and we also know the parameter t of the closest point on the curve $B_n(t)$ to P. All that's left to do to determine which side P is on, is to find the relative direction from $B_n(t)$ to P in respect to the curve's direction. That's what cross product is for.

The complete formula for the *signed* distance therefore is:

$$\text{sdistance}(B_n, P) = \text{sgn}\left(\frac{\mathrm{d}B_n}{\mathrm{d}t}(t) \times (B_n(t) - P)\right) \|B_n(t) - P\| \tag{2.41}$$

2.4 Point – shape distance

When finding the signed distance from the entire shape, the first idea might be to simply use the distance from the closest edge segment. This would work well for the absolute distance, but poses a problem with determining the sign.

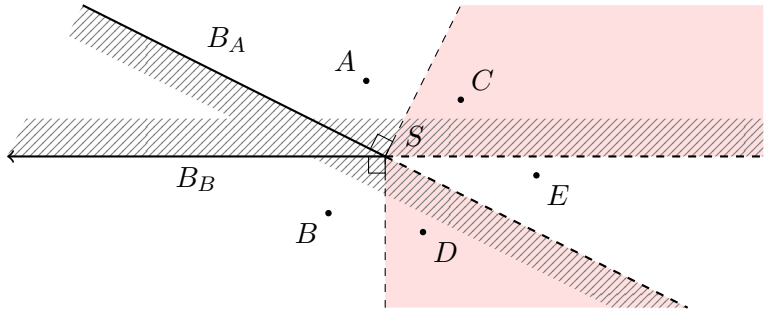


Figure 2.6: The problem with using the distance to the closest segment.

Consider the situation in Figure 2.6, where the segments B_A and B_B make up a convex corner S of the shape. For point A, segment B_A is the closest. For pont B, it is segment B_B. For points C, D, and E however, both segments are equally distant, and the minimum distance is that to the corner S.

Point A is on the outside of B_A and on the inside of B_B. Since it is closer to B_A, only that will be taken into consideration and A will be ruled outside. Point E is on the outside of both B_A and B_B, so it will also be correctly ruled outside. Point C is on the outside of B_A but on the inside of B_B. It is equally close to both segments, and therefore it is unclear which value should be used.

To solve this, we will divide the plane between the two segments along a ray leading from S in the direction of the corner, as shown in Figure 2.7.

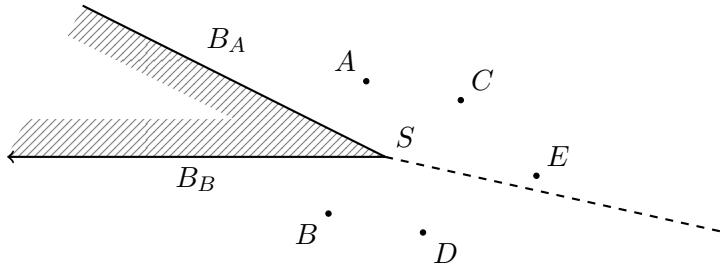


Figure 2.7: Dividing the plane between two adjacent segments.

To achieve this, the signed distance value from each edge is not enough. In Figures 2.6 and 2.7, although the shortest distance from C to segments B_A and B_B is equal, the signed distance to B_A must be used. That's because relative to the corner, it lies on B_A's half of the plane. This can be also described by how orthogonal to the segment's direction at the corner the point is. The rate of "orthogonality" between two normalized vectors can be measured by their cross product. In the case of C, it would hold that:

$$\left| \frac{\frac{\mathrm{d}B_A}{\mathrm{d}t}(1)}{\|\frac{\mathrm{d}B_A}{\mathrm{d}t}(1)\|} \times \frac{C - B_A(1)}{\|C - B_A(1)\|} \right| > \left| \frac{\frac{\mathrm{d}B_B}{\mathrm{d}t}(0)}{\|\frac{\mathrm{d}B_B}{\mathrm{d}t}(0)\|} \times \frac{C - B_B(0)}{\|C - B_B(0)\|} \right| \tag{2.42}$$

This also works in general. The dividing ray is equally non-perpendicular to both segments, but points above it would make a sharper angle with B_A than with B_B.

Therefore, if distances are equal, we must maximize the "orthogonality":

$$\text{orthogonality}(B_n, P) = \left| \frac{\frac{\mathrm{d}B_n}{\mathrm{d}t}(t)}{\|\frac{\mathrm{d}B_n}{\mathrm{d}t}(t)\|} \times \frac{P - B_n(t)}{\|P - B_n(t)\|} \right| \tag{2.43}$$

It is worth noting that partitioning the plane by which segment or edge is the closest according to these rules results in a generalized Voronoi tesselation. [2] An example of a generalized Voronoi diagram generated by dividing the plane by the closest edge is shown in Figure 2.8.

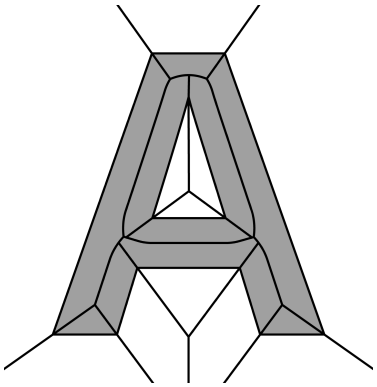


Figure 2.8: A generalized Voronoi diagram of the letter "A".

2.5 Pseudo-distance fields

A special variant of the signed distance field is the signed pseudo-distance field, which uses signed pseudo-distances instead. The pseudo-distance to an edge segment is the minimum perpendicular distance to any point lying on

the edge segment itself or anywhere on its infinite extension which leads from each endpoint. Unlike before, the distance to endpoints is not considered.

For a line segment, this is trivial, and Equation 2.25 can be used. For curves, there are two possible approaches with different outputs. Either the parameter t is computed as usual, but isn't clamped to the range $\langle 0, 1 \rangle$, resulting in a smooth continuation of the curve, or the curve is extended using straight rays, and the candidates for closest point on the segment will include solutions perpendicular to both the curve itself, and the rays. An example of pseudo-distance can be seen in Figure 2.9.

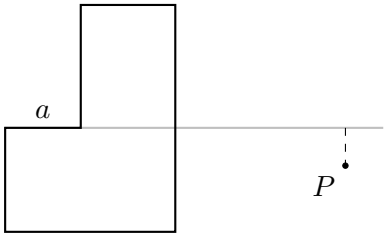
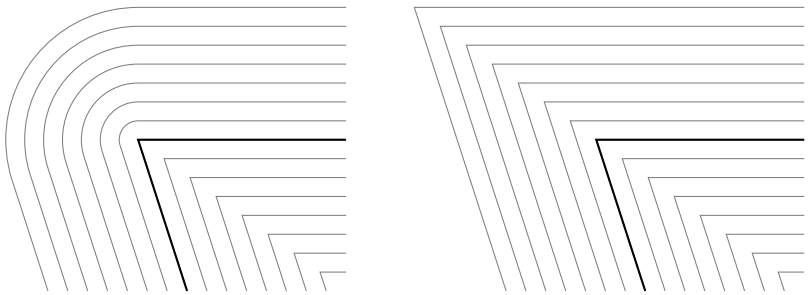


Figure 2.9: The signed pseudo-distance from point P to edge a.

Figure 2.9 also demonstrates however, that edge pseudo-distances won't be enough to find the shape pseudo-distance. Note that at point P, the minimum pseudo-distance is to the edge a and is positive. This would place P inside the shape. It is clear however, that edge a is completely irrelevant in this area and that P is in fact outside. The correct pseudo-distance to the shape is therefore the pseudo-distance to the edge segment that is closest in terms of true distance.



(a) A true distance field. (b) A pseudo-distance field.

Figure 2.10: Contour graph of a regular distance field and a pseudo-distance field around a corner.

The difference between a regular and a pseudo-distance field is best seen in Figure 2.10. At first glance, it may seem that this change of the distance metric facilitates the preservation of sharp corners, but that is not the case.

Chapter 3

Theoretical analysis

In this chapter, I will present the problems of multi-channel shape decomposition from a theoretical standpoint, and the solutions I have developed.

3.1 Shape simplification

The outline of the vector shape may include subtle details or even unintentional imperfections that may be only visible at a large magnification and that are smaller than the grid of the signed distance field. Such nuances may hinder the effectiveness of the decomposition as the edges in these areas will be too short to be encoded in the distance field, and attempting to preserve the sharpness of their corners will be counterproductive. Therefore, some preprocessing of the shape vector is in order.

3.1.1 Merging close vertices

If two or more adjacent vertices are too close together, a possible solution would be to somehow exclude them from the decomposition logic, and not attempting to preserve corners in these areas.

In some cases however, it may be better to treat the group of vertices as one. The Open Sans typeface for example, has a recurring design element, where a very short edge divides what would otherwise be a sharp concave corner. The edge is so short that it in fact looks like a sharp concave corner at reasonable sizes, and therefore treating it as such would be preferable to losing sharpness in the area.

To do this, we can prune the shape in pre-process of any edges shorter than a given limit. The outline then has to be reconnected by moving the endpoints of the remaining edges together.

3.1.2 Moving the endpoint of a curve

If the edge segment, whose endpoint we need to move is a line segment, the operation is trivial. If it is a curve though, we must take care not to alter its appearance any more than necessary.

An effective way to achieve this is to preserve the curve's direction at the endpoints, and therefore not creating (or destroying) new corners and preserving the adjoining corners' angles. For a cubic Bézier curve, this is also simple. Since its direction vectors at endpoints are $P_1 - P_0$ and $P_3 - P_2$ (Equations 2.13 and 2.14), by changing the position of P_1 along with P_0 and P_2 along with P_3, both of these vectors will remain unchanged.

The same cannot be said for the simpler quadratic curves however, where only one control point is shared between the endpoints.

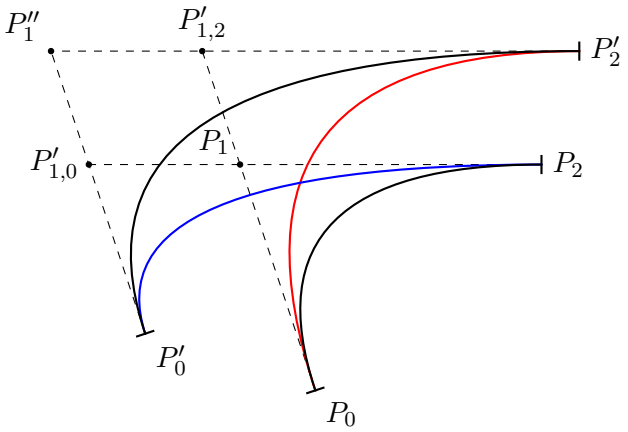


Figure 3.1: Independent translation of a quadratic curve's endpoints.

In Figure 3.1, we start with the curve (P_0, P_1, P_2). If we were to move the endpoint P_0 to P_0', the correct way to adjust the control point P_1 would be to move it along $P_2 - P_1$ using the following formula:

$$P_{1,0}' = P_1 + \frac{(P_0 - P_1) \times (P_0' - P_0)}{(P_0 - P_1) \times (P_2 - P_1)} \cdot (P_2 - P_1) \tag{3.1}$$

Adjusting the control point along $P_0 - P_1$ when moving P_2 to P_2' instead would look like this:

$$P_{1,2}' = P_1 + \frac{(P_2 - P_1) \times (P_2' - P_2)}{(P_2 - P_1) \times (P_0 - P_1)} \cdot (P_0 - P_1) \tag{3.2}$$

One can verify that the orientation of the curve at either endpoint hasn't changed by checking the cross products:

$$(P_0 - P_1) \times (P_0' - P_{1,0}') \stackrel{?}{=} 0 \tag{3.3}$$

$$(P_2 - P_1) \times (P_2 - P_{1,0}') \stackrel{?}{=} 0 \tag{3.4}$$

Both of the equations have been checked by a computer and confirmed to hold for any vectors P_0, P_1, P_2, and P'_0, and substituting $P'_{1,0}$ using Equation 3.1. However, this alone doesn't guarantee that the direction vectors weren't flipped, still making the cross products zero but changing the shape in a major way. Since this is the only way to alter a quadratic Bézier curve while preserving endpoint directions, if it fails, that means the direction cannot be preserved due to limitations of the quadratic curve.

Combining the two operations in any order places the control point at P''_1:

$$P''_1 = P'_{1,0} + \frac{(P_2 - P'_{1,0}) \times (P'_2 - P_2)}{(P_2 - P'_{1,0}) \times (P'_0 - P'_{1,0})} \cdot (P'_0 - P'_{1,0}) \tag{3.5}$$

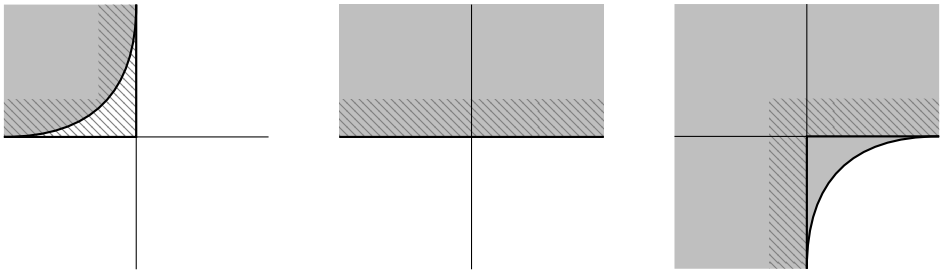
$$= P'_{1,2} + \frac{(P_0 - P'_{1,2}) \times (P'_0 - P_0)}{(P_0 - P'_{1,2}) \times (P'_2 - P'_{1,2})} \cdot (P'_2 - P'_{1,2}) \tag{3.6}$$

Again, it has been confirmed by a computer program that (3.5) and (3.6) are equivalent, and therefore moving the two endpoints in any order results in the same curve.

3.2 Corner analysis

In order to correctly adjust the distance field to accommodate for sharp corners, we must first understand what exactly happens around them.

Let's divide the plane into four quadrants, with the ones lying inside the shape filled. Figure 3.2 shows the average case behavior of the image reconstructed from a signed distance field. Figure 3.2a corresponds to a convex corner of the shape, Figure 3.2b to a straight edge, and Figure 3.2c to a concave corner. The correct border is hatched.



(a) Top-left only. (b) Top-left and top-right. (c) All but bottom-right.

Figure 3.2: The average resulting image of filled quadrants from an SDF.

There is however one other case. When only the opposite quadrants are filled, the result can be one of several possibilities depending on the alignment with the grid of the distance field. The possibilities include the cases illustrated in Figure 3.3 and anything inbetween.

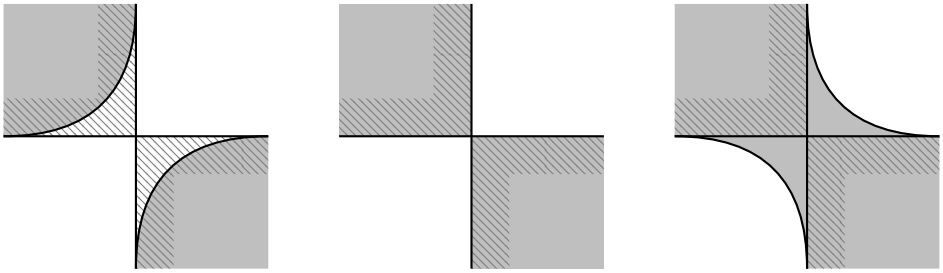


Figure 3.3: The possible results of filled opposing quadrants using an SDF.

The amount of rounding off at the corner also depends on the alignment with the distance field's grid and therefore is essentially random. A composite image of all possible outcomes is shown in Figure C.1. What is important, is that some quadrants are guaranteed to remain homogeneous.

Using the observations from Figures 3.2 and 3.3, we can divide any corner into eight roughly homogeneous areas, as illustrated in Figure 3.4.

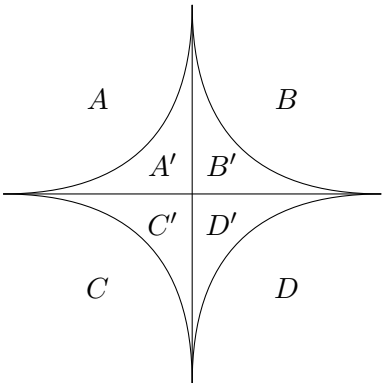


Figure 3.4: Dividing the plane into quadrants and subquadrants.

All of the previous illustrations only capture the case of an axis aligned and orthogonal corner. Of course, a corner can be oriented in any way and can form any angle. The quadrants would then also be aligned differently, always along the two edge segments at the corner, as illustrated in Figure 3.5.

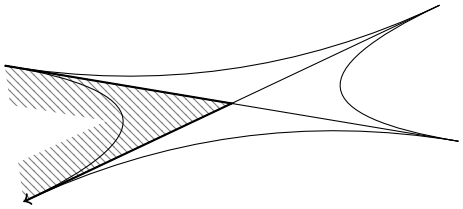


Figure 3.5: Example of quadrant alignment of a non-orthogonal corner.

Based on whether the shape is filled in a given quadrant A through D (1) or not (0), it can be determined if the areas A' through D' will appear filled in the resulting image. This is captured in Table 3.1.

A	B	C	D	A'	B'	C'	D'
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	1	1
0	1	0	0	0	0	0	0
0	1	0	1	0	1	0	1
0	1	1	0	**?**	**?**	**?**	**?**
0	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0
1	0	0	1	**?**	**?**	**?**	**?**
1	0	1	0	1	0	1	0
1	0	1	1	1	1	1	1
1	1	0	0	1	1	0	0
1	1	0	1	1	1	1	1
1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1

Table 3.1: Truth table of the filling of quadrant areas.

This can be expressed by a Boolean function f that returns whether the central subquadrant is filled depending on the intended filling of the quadrants. Its first parameter is the fill of the local quadrant, the following two are the neighboring quadrants, the forth one is the opposite quadrant, and the last one, r, is a random bit, which has a role in the two uncertain scenarios.

$$A' = f(A, B, C, D, r) \tag{3.7}$$
$$B' = f(B, A, D, C, r) \tag{3.8}$$
$$C' = f(C, A, D, B, r) \tag{3.9}$$
$$D' = f(D, B, C, A, r) \tag{3.10}$$

Based on Table 3.1, function f can be defined as follows:

$$f(A, B, C, D, r) \stackrel{\text{def}}{=} AB + AC + BCD + rA\overline{BC}D + \overline{r}\,\overline{A}BC\overline{D} \tag{3.11}$$

3.2.1 Switching to multiple channels

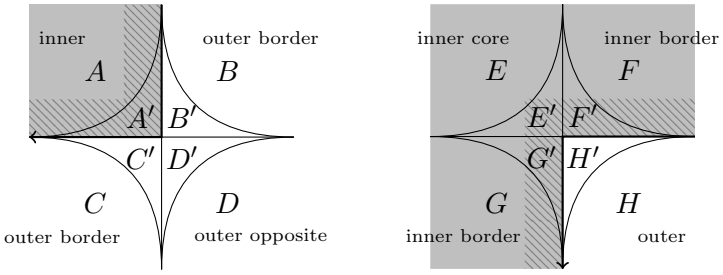
Using more than one channel in the distance field, the same rules apply for the values in each channel. Instead of using a binary value to denote the fill

of an area, we will use binary vectors, where each dimension corresponds to a certain channel. For any such vector A, its norm $\|A\| \in \mathbb{B}$ will determine if it marks an area inside the shape (1) or outside (0).

The function f can also be generalized to accept and return binary vectors, performing the original operation separately for each channel:

$$f_n(A, B, C, D, r) \stackrel{\text{def}}{=} (f(a_1, b_1, c_1, d_1, r), ..., f(a_n, b_n, c_n, d_n, r)) \\ : A, B, C, D \in \mathbb{B}^n, r \in \mathbb{B} \quad (3.12)$$

All that is left to do is to assign the correct color combinations for quadrants of convex and concave corners (Figure 3.6).



(a) A convex corner. (b) A concave corner.

Figure 3.6: The two possible corner types and their quadrants.

To ensure the corners are sharp, the channels must be chosen in such a way, that the areas that are supposed to be filled are distinguishable from those

Inside	Outside
A	B
$f_n(A, B, C, D, 0)$	$f_n(B, A, D, C, 0)$
$f_n(A, B, C, D, 1)$	$f_n(B, A, D, C, 1)$
E	C
$f_n(E, F, G, H, 0)$	$f_n(C, A, D, B, 0)$
$f_n(E, F, G, H, 1)$	$f_n(C, A, D, B, 1)$
F	D
$f_n(F, E, H, G, 0)$	$f_n(D, B, C, A, 0)$
$f_n(F, E, H, G, 1)$	$f_n(D, B, C, A, 1)$
G	H
$f_n(G, E, H, F, 0)$	$f_n(H, F, G, E, 0)$
$f_n(G, E, H, F, 1)$	$f_n(H, F, G, E, 1)$

Table 3.2: The differentiation of binary vectors that denote the inside and outside of the shape.

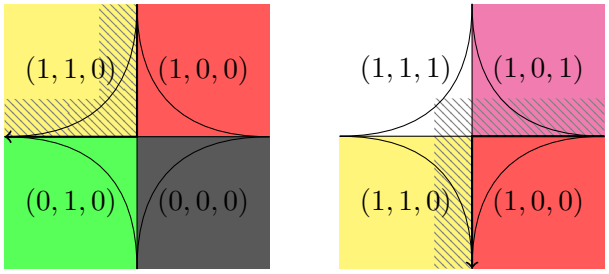
that are not. Therefore, the values of vectors $A, B, C, D, E, F, G, H \in \mathbb{B}^2$ must be chosen so that the norm of no vector in the left column of Table 3.2 is equal to the norm of any vector in the right column.

3.2.2 Median of three model

By having a computer program test all possible values of vectors A through H, I have found that the minimum dimension n for which the distinction can be satisfied is 3. The most intuitive working model I have discovered this way, is one where the median value of the three components of the vector dictates the result:

$$\|A\| \stackrel{\text{def}}{=} \text{median}(a_1, a_2, a_3) \tag{3.13}$$

An example of this is shown in Figure 3.7, where the first dimension of the binary vectors is encoded using the red color channel, the second dimension with green channel, and the third one with blue channel.



(a) A convex corner. (b) A concave corner.

Figure 3.7: Possible color encoding of a corner's quadrants using the median of three model.

The individual dimensions in this model are interchangable because median disregards the order of arguments. For a convex corner with inner quadrant A, outer border quadrants B and C, and outer opposite quadrant D, it must hold that:

$$B + C \leq A \tag{3.14}$$

$$BC = D = \vec{0} \tag{3.15}$$

$$\|B\| = \|C\| = 0 \tag{3.16}$$

For any concave corner with inner core quadrant E, inner border quadrants F and G, and outer quadrant H, it must hold that:

$$FG \geq H \tag{3.17}$$

$$F + G = E = \vec{1} \tag{3.18}$$

$$\|F\| = \|G\| = 1 \tag{3.19}$$

3.3 Plane partitioning

Having set the rules of channel assigment around corners, it is now necessary to enforce them throughout the plane. There are many corners in a a shape's outline, and the neighborhood of each of them must be assigned different coloring without interfering with the rest. To achieve this, the plane must be divided not only into quadrants around corners, but also between quadrants of different corners.

To divide the plane, I have utilized the generalized Voronoi tesselation introduced in Section 2.4, with edges split into two half-edges. The plane is this way divided into areas with a common nearest half-edge. Each half-edge coincides with one corner, and therefore the half-edge's area can be attributed to that corner. Each corner then has two areas in the Voronoi partitioning, and in those areas, the plane is divided according to that corner's quadrants.

Ideally, the touching quadrants of different corners should have the same color. This can be enforced for the quadrants of adjacent corners, which are guaranteed to meet at the midpoint of each edge, using a coloring strategy.

3.3.1 Coloring strategy

Using the median of three model from Section 3.2.2, there are several valid ways to assign colors to quadrants of a corner. In order to make sure the touching quadrants of adjacent corners are colored the same, we can use the following procedure:

- Set I to any vector with two set bits, set O to any vector with one set bit, such that $IO \neq \vec{0}$.
- Walk along the edge and for each corner:
  - If it is convex, use I for its inner quadrant, $\vec{0}$ for outer opposite, O for the first outer border quadrant, and $I - O$ for the second outer border quadrant (the one touching the area of the following corner). Set $O := I - O$.
  - If it is concave, use O for its outer quadrant, $\vec{1}$ for inner core quadrant, and I for the first inner border quadrant. Set $I := \vec{1} - (I - O)$ and use the new value of I for the remaining inner border quadrant.

Unfortunatelly, it is not always possible to apply this method in a way that the first and last corner's quadrants are matched, which means that some shapes must have at least one edge with mismatching colors.

Take the example in Figure 3.8. Here we start with $I_1 = (1, 1, 0)$ and $O_1 = (1, 0, 0)$. The first corner is a concave one, so its quadrants are colored according to these rules, and only I is changed:

$$I_2 = \vec{1} - (I_1 - O_1) = \vec{1} - ((1, 1, 0) - (1, 0, 0)) = (1, 1, 1) - (0, 1, 0) = (1, 0, 1) \tag{3.20}$$

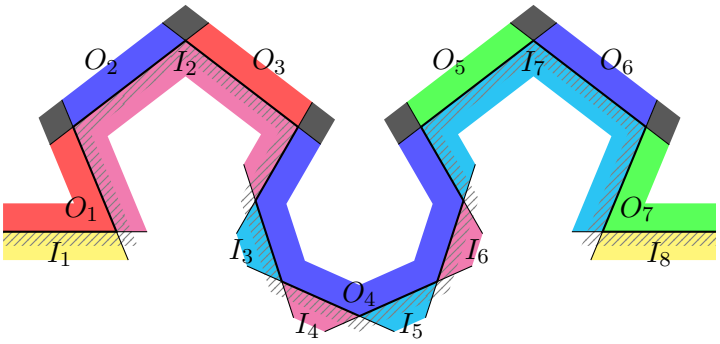


Figure 3.8: Quadrant coloring of a sequence of corners.

The next corner is convex, so only O is changed this time:

$$O_2 = I_2 - O_1 = (1, 0, 1) - (1, 0, 0) = (0, 0, 1) \tag{3.21}$$

And so on. You can check that the quadrants of each corner satisfy the condition established in Table 3.2, and also that each edge has a single solid color on each side.

If the ends of the sequence were to be connected though, this poses a problem, as $O_7 \neq O_1$. The outside quadrant color would have to change in the middle of the connecting edge.

3.3.2 Collisions of uncorrelated areas

Often, the quadrants of unrelated edges from completely different parts of the shape will meet too. Unfortunately, this happens in a very irregular and unpredictable fashion, and therefore the quadrant coloring in these cases cannot be reliably correlated.

Since the distance field representation isn't exact, it can be disturbed by nearby edges in such a way that the border between the uncorrelated quadrants will be shifted slightly differently for each color channel. This phenomenon is illustrated in Figure 3.9.

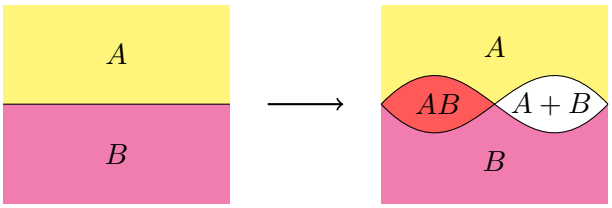


Figure 3.9: The border between areas colored A and B (left) and a possible result after reconstruction (right).

Note that if $A = (1, 1, 0)$ and $B = (1, 0, 1)$, both inside values, as is the case in the picture, then $AB = (1, 0, 0)$, which is an outside value. This means that an unintentional hole will appear in the resulting image at this spot, and that is a serious problem.

To prevent this, we will introduce padding. Between the separate areas in the plane partitioning, a neutral area of padding will be inserted and filled with a neutral color $p(A, B)$ derived from the colors on both sides, A and B. The effect is demonstrated in Figure 3.10.

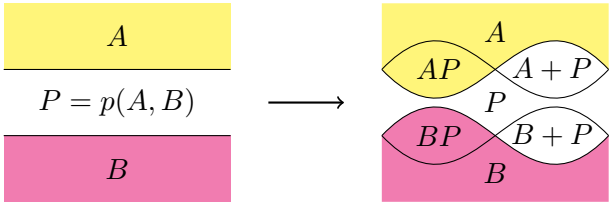


Figure 3.10: Padding P between areas colored A and B (left) and a possible result after reconstruction (right).

We can see that for this to work, it must hold that

$$\|A\| = \|B\| = \|p(A, B)\| = \|Ap(A, B)\| = \|A + p(A, B)\| \\ = \|Bp(A, B)\| = \|B + p(A, B)\|. \quad (3.22)$$

The neutral color depends on whether this is inside the shape, or outside. For the inside, we can use $p(A, B) = A+B$, and for the outside $p(A, B) = AB$. Note that if $A = B$, which is the ideal case when no padding is needed, then $p(A, B)$ will also be equal to A and B, making the padding invisible.

Chapter 4

Realization

This chapter details the workings of the proposed methods and algorithms.

4.1 Preparing the input

The first step will be to acquire the vector shape in the right format.

4.1.1 Loading shape from file

Since the primary purpose of this work is improving text rendering, it is important to be able to load character glyphs from widely used font file formats, specifically vector formats, such as TrueType and OpenType. To also allow for decomposition of other miscellaneous vector shapes, I have added basic support for SVG files.

4.1.1.1 Loading font files

The two commonly used formats for vector-based fonts are TrueType (TTF) and OpenType (OTF). [1] Both of these formats are based on storing the outlines of individual characters (glyphs) using line segments, and quadratic or cubic Bézier splines. To decode these two formats, I have used the FreeType library [15], which enables extraction of the individual curves.

4.1.1.2 Loading SVG files

SVG files have the form of XML documents. Although they have extensive capabilities for vector graphics storage, we must extract only a single colorless shape. For this simple task, I used the lightweight TinyXML 2 library [14] to parse the XML file and locate the first `<path>` element. The definition of the path is then parsed from the element's attributes according to [19].

4.1.2 Input representation

It might not be immediately clear how to represent a vector shape using outlines, since it might not form a simply connected space [18]. For this, I have adopted the approach used by the TTF and OTF file formats, which solves the exact same problem.

Each shape, or glyph, consists of a set of non-intersecting outlines, or *contours*, which are closed paths made up of edge segments. The edge segments in the contours are oriented, determining the contour's winding, so for example, if the segments of a contour are oriented clockwise, its winding is positive. When combining the contours into a single shape, a clockwise oriented contour will be additive, and a counter-clockwise one subtractive.

This means, that counter-clockwise oriented contours can be placed inside clockwise oriented ones to form cutouts. This system has an important property, which is that the inside of the shape is always on the same side of an oriented edge segment. This property will be very important when computing signed distances later on.

An edge segment is a part of the outline that smoothly connects two vertices. It can be either a line segment, a quadratic Bézier curve, or a cubic Bézier curve.

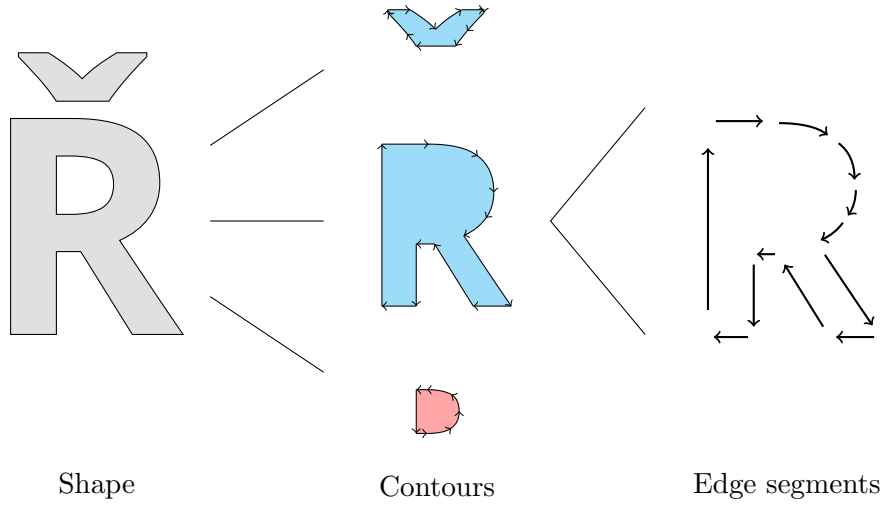


Figure 4.1: The initial structure of a shape prototype.

An example can be seen in Figure 4.1. The glyph of the letter "Ř" here consists of three contours. The basic "R" shape and the caron are oriented clockwise, while the central crescent cutout is oriented the opposite way.

This representation shall be called the shape prototype, and if the provided SVG and font file loading capabilities don't suffice, such prototype can be easily constructed by other means.

4.1.3 Shape preprocessing

After having acquired the shape in this format, the contour must be divided into edges at actual corners. The mechanism of identifying these is described in Section 2.2. This step basically groups several smoothly connected edge segments into a single logical edge, and alters the hierarchy of the shape contents, as shown in Figure 4.2.

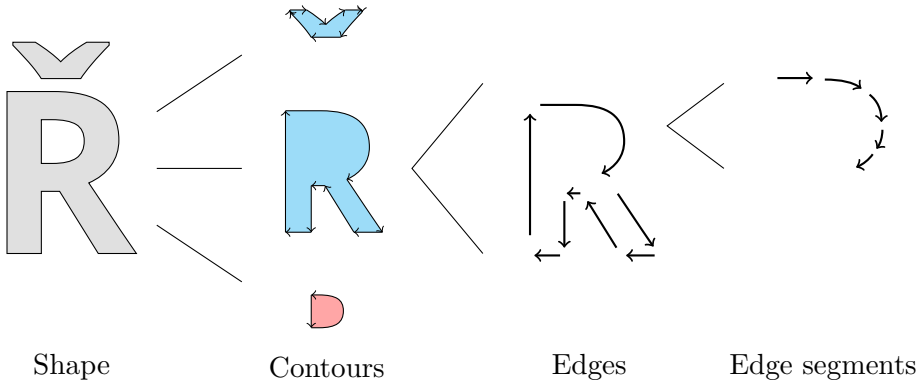


Figure 4.2: The structure of the shape after edge grouping.

4.1.3.1 Edge pruning

The next step, which is optional, is to prune the shape of edges that are too short to be properly represented in the low-resolution distance field and unfit for any attempts at corner preservation.

Here, edges, whose (estimated) length is below a given threshold will be removed. After that, the resulting gaps will be closed up by moving the endpoints of the disconnected neighboring edges to their midpoint using the technique from Section 3.1.

At this point, the shape is ready for processing, and what to do with it is up to the user. I have implemented two algorithms, which construct a multi-channel signed distance field of the shape that improves the rendering quality of sharp corners. Each takes a fundamentally different approach to the distance field's construction.

4.2 Single-channel distance field construction

The most basic functionality, which could not have been omitted is the construction of a regular single-channel distance field. This is necessary mainly for comparative testing, but some concepts described here will be useful later on.

4.2.1 General image construction

The distance fields will have the form of raster images. All algorithms in this thesis whose task is to generate an image will have roughly the following form, and only the GeneratePixel function will differ:

Algorithm 1 The general procedure for generating an image.

```
1: procedure ConstructImage(bitmap, width, height)
2:     for y ← 0 to height − 1 do
3:         for x ← 0 to width − 1 do
4:             P ← TransformCoordinate((x+1/2)/width, (y+1/2)/height)
5:             bitmap(x, y) ← GeneratePixel(P)
```

The entire bitmap of the image will be traversed once using a cycle, and the color of a single pixel determined in each step. Each pixel corresponds to a position in the coordinate system of the vector shape. That position is computed using a transformation function TransformCoordinate. This transformation will be specified by the user in order to frame the shape in the desired way. Note that half a pixel size is added to the pixel coordinates. This is so that the position of the pixel's center is used.

Another issue is converting the signed distance to a color value. Each color channel is typically stored as a single byte – an integer value between 0 to 255. For this, a maximum absolute distance d_{MAX} must be chosen by the user, and the range $\langle -d_{MAX}, +d_{MAX} \rangle$ converted to $\langle 0, 255 \rangle$ in the case of a byte bitmap. Finally, the value is rounded if necessary.

For this conversion, I will define the distanceColor map as

$$\text{distanceColor}(\mathit{distance}) \stackrel{\text{def}}{=} \left(\frac{\mathit{distance}}{\mathit{distanceRange}} + \frac{1}{2} \right) \cdot \mathit{maxColor}, \qquad (4.1)$$

where $\mathit{distanceRange} = 2d_{MAX}$ is the width of the distance range and $\mathit{maxColor}$ is the maximum color value, 255 for byte bitmaps.

4.2.2 Finding the signed distance to the closest edge

As described in Section 2.4, to determine which edge is the closest, the distance measure has to include two values – the actual distance, and a measure of orthogonality. All signed distance values in the following algorithms will hold both of these components, and the comparison function Cmp will correctly take both into account to determine which distance value is factually closer.

I have implemented the EdgeSignedDistance function, which returns the correct signed distance from an edge in this format, according to Section 2.3. For cubic curves, which require solving a fifth degree polynomial equation, it uses Henrik Vestermark's implementation of the Jenkins-Traub algorithm [7], which is a globally convergent approximate solution method.

Using this function and the comparison Cmp, the edge whose absolute distance is deemed lower than any other is the closest edge at position P:

Algorithm 2 Finding the closest edge of shape s to a point P.

```
1:  function ClosestEdge(P, s)
2:      dMin ← ∞
3:      eMin ← nil
4:      for each contour c of shape s do
5:          for each edge e of contour c do
6:              d ← EdgeSignedDistance(P, e)
7:              if Cmp(d, dMin) < 0 then
8:                  dMin ← d
9:                  eMin ← e
10:     return eMin
```

The signed distance from the shape can be now determined simply as the signed distance from the closest edge:

Algorithm 3 Pixels of a regular signed distance field.

```
1: function GeneratePixel(P)
2:     e ← ClosestEdge(P, s)
3:     d ← EdgeSignedDistance(P, e)
4:     return distanceColor(d)
```

A more sophisticated lookup of the closest edge could be employed here to increase the algorithm's performance.

4.2.3 Pseudo-distance field

The pseudo-distance field can be constructed just as simply, assuming an EdgeSignedPseudoDistance function that returns the correct shape pseudo-distance in accordance with Section 2.5.

Algorithm 4 Pixels of a pseudo-distance field.

```
1: function GeneratePixel(P)
2:     e ← ClosestEdge(P, s)
3:     d ← EdgeSignedPseudoDistance(P, e)
4:     return distanceColor(d)
```

4.3 Corner preserving shape decomposition

There are several possible ways to construct the signed distance field of a shape's multi-channel decomposition:

- Constructing the exact vector representation of the decomposition and using it to generate the SDF,
- constructing a raster representation of the decomposition and using it to generate the SDF,
- or, constructing the distance field directly.

I have not attempted the first option, because it would include a problem very similar to the construction of the vector representation of an exact generalized Voronoi diagram of Bézier curves and line segments. That problem alone would be extremely difficult, and would probably exceed the scope of this whole thesis.

Therefore, I started with the second option, which is the easiest. It has two disadvantages however. Firstly, the SDF will be generated from an image of finite resolution, and therefore won't be exact. This can however be negligible if the resolution is high enough. The second problem is that constructing a possibly very large image of the intermediate decomposition will have a major impact on the algorithm's performance and memory consumption.

4.3.1 Edge channel assigment

Although coloring pertains to corner quadrants, it should be apparent from Figure 3.8, that the colors can be assigned to edges instead. Since it might be necessary that the two endpoints of an edge have different coloring, each edge will be assigned four colors in total – two inner and two outer, for the first half and the second half of the edge.

The edge colors will be assigned using the coloring strategy from Section 3.3.1.

4.3.2 Construction of the intermediate decomposition

The next step is to render the high resolution representation of the decomposition. At each pixel of the decomposition, it must be determined which corner it belongs to by finding the closest half-edge. Then, it must be decided which of the corner's quadrants the pixel occupies.

The closest half-edge can be determined using ClosestEdge, and then finding at what portion $t \in \langle 0, 1\rangle$ this edge is closest to P. A value $t < \frac{1}{2}$ implies the first half of the edge and its first endpoint A as the parent corner. A value $t \geq \frac{1}{2}$ implies the second half of the edge and its second endpoint B as the parent corner.

After acquiring the closest edge with endpoints A and B, its neighboring edges n_A and n_B, its signed distance d, and the portion t at which the distance is minimal, the following algorithm can be used to determine the quadrant in which the point P lies.

Algorithm 5 Determining the pixel's quadrant.

if $t < \frac{1}{2}$ **then**
    core $\leftarrow \left((P - A) \times \frac{\mathrm{d}n_A}{\mathrm{d}t}(1) > 0\right) \oplus$ (is A convex?)
else
    core $\leftarrow \left((P - B) \times \frac{\mathrm{d}n_B}{\mathrm{d}t}(0) > 0\right) \oplus$ (is B convex?)
if $d \geq 0$ **then**
    if *core* **then** *quadrant* $\leftarrow$ inner core **else** *quadrant* $\leftarrow$ inner border
else
    if *core* **then** *quadrant* $\leftarrow$ outer opposite **else** *quadrant* $\leftarrow$ outer border

Now, applying the median of three model from Section 3.2.2, the inner core pixels will be colored white (all channels on), and outer opposite pixels black (all channels off). For the border quadrants, the half-edge's inner or outer color will be used, depending on the quadrant, as assigned in the previous step.

The only remaining issue to be solved is when two inner or two outer areas of different colors touch, which has to be resolved using padding (described in Section 3.3.2). I have developed two methods of inserting padding into the decomposition.

4.3.2.1 Padding from distance ratios

This approach isn't perfect, but it works very well in most situations and can be performed in one pass along with the previous color computation.

Once the closest edge for a pixel has been found, additionally all edges, which are less than x times farther, and on the same side of the shape, must be found as well. The coloring of each of these vertices will be then combined (using OR if inside, AND if outside).

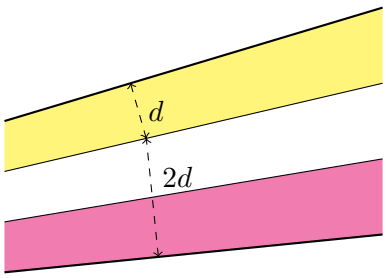


Figure 4.3: Padding (white) derived from distance ratios.

This mechanism, in most cases, results in a stripe proportional to the shape's thickness in that area. The example in Figure 4.3 uses $x = 2$ and because of that, divides the stroke into thirds.

There is however a small problem with this approach. In Figure 4.4, the reddish area also satisfies this condition for edges a and b, marking it as padding between the edges, which it clearly is not. Although this behavior is not intentional, it is harmless, as all it does is add unnecessary areas of padding very far from any edges (a has to be the closest one for this to happen), which have no impact on image quality, but may make the decomposition look strange. I have tried to fix this cosmetic issue, but failed to reach a solution without side effects.

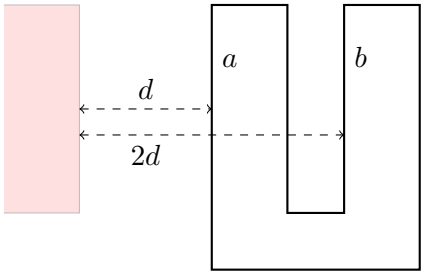


Figure 4.4: Unnecessary false padding resulting from this method.

4.3.2.2 Padding in post-process

The other option is to generate the image without addressing this problem, and solving it separetely afterwards, by locating pixels closer than x to pixels of incompatible color, and marking them. In this case, the appropriate value of x can be derived from the original and target SDF resolution, making the padding only exactly as thick as needed to prevent artifacts. In another pass, the inner marked pixels would be recolored white, and the outer ones black.

4.3.3 Distance field construction from raster decomposition

The last step after acquiring the raster representation of the decomposition is constructing the multi-channel signed distance field. Of course, the procedure is exactly the same as constructing a regular signed distance field separately for each color channel.

Exact linear-time algorithms exist for this task [3], but I chose to take a simpler approach, iterating over all of the border pixels in the decomposition, and incorporating the signed distance of each into the entire distance field, thus eventually finding the minimum for each cell.

Interestingly, even computing the correct signed distance between one such border pixel and a distance field cell position poses a problem. Since the actual border of the shape isn't at the border pixel but right next to it, the Euclidean distance has to be adjusted by about half a pixel to account for this.

4.4 Direct multi-channel distance field construction

After carefully analyzing the properties of the various decompositions and observing the behavior of the distance fields, I have managed to develop a simplified version of the algorithm, which is able to construct the multi-channel distance field directly from the vector representation of the shape.

This algorithm is superior to the indirect construction for the following reasons:

- It is very fast. It's time complexity is equivalent to the construction of a regular distance field.
- It requires no additional memory apart from a constant number of temporary variables.
- It doesn't lose the signed distance information in any part of the image, which can be used for visual effects. It even preserves sharp corners at other distances from the outline.
- The algorithm itself is simple and elegant.

Because of this, I have omitted some optimizations in the implementation of the previous algorithm, and only mentioned the possibilities. They can however be added easily.

The direct algorithm is also based on the median of three model, and therefore works with 3 channels. It has three steps.

4.4.1 Edge channel assignment

This time, each edge will be only assigned one color. The only exception to this is the case when a contour has a teardrop shape – when it only consists of a single edge, but forms a sharp corner where it connects to itself. In this case, the edge has to be divided into half-edges with different colors.

There are only two rules for the coloring. First, every edge must have at least two channels on (leaving white, yellow, magenta, or cyan colors only), and second, in order for a corner to remain sharp, its two adjacent edges must have exactly one channel in common.

In practice, this means that we need to cycle between the three two-channel colors – yellow, magenta, and cyan – so that no two adjacent edges have the same color. This is always possible without dividing the edges any further (except in the teardrop case). The following is a simple method to achieve this:

Algorithm 6 Simple edge color assignment assuring that no two adjacent edges of shape s share the same color.

```
1: procedure EdgeColoring(s)
2:     for each contour c of shape s do
3:         if c has only 1 edge then
4:             current ← (1, 1, 1)
5:         else
6:             current ← (1, 0, 1)
7:         for each edge e of contour c do
8:             color(e) ← current
9:             if current = (1, 1, 0) then
10:                current ← (0, 1, 1)
11:            else
12:                current ← (1, 1, 0)
```

Additionally, the white color can be used in some special cases, as a sort of neutral color. As applied in Algorithm 6, it can be used for fully smooth contours (with only one edge). For this case it is the safest option, because it won't cause clashes with neighboring contours that are too close. It could also be used as padding, or to exclude edges from the corner preserving scheme. This could be used for the very short edges, where coloring won't pay off, instead of pruning them in the pre-process stage.

4.4.2 Distance field construction

Now that each edge has a color, the distance field can be constructed in exactly the same way as the pseudo-distance field (Section 4.2.3), except that for each channel of the distance field, only the edges with that channel set will be considered. This is facilitated by Algorithm 7.

Another perspective to look at this is that we are constructing three pseudo-distance fields, each with only a subset of the original shape's edges. However, since the subsets of edges are generally disconnected, these are not correctly formed distance fields that could be constructed from an actual decomposition. The values in the distance fields won't be continuous either, there will be hard jumps between large negative and large positive distances, referred to as *false edges*. It has been confirmed by testing though, that these cases can only arise either far from any edges, or at edges that do not belong to the discontinuous distance field channel. If an edge does not belong to one channel, it must belong to the other two according to the coloring rules, and therefore the distance values for the remaining two channels will be the same. Because of

$$\text{median}(a, a, b) = a, \tag{4.2}$$

the value in the discontinuous channel won't affect the result.

Algorithm 7 Pixels of a corner preserving multi-channel distance field.

```
1:  function GeneratePixel(P)
2:      dRed ← ∞, dGreen ← ∞, dBlue ← ∞
3:      for each contour c of shape s do
4:          for each edge e of contour c do
5:              d ← EdgeSignedDistance(P, e)
6:              if color(e) · (1, 0, 0) ≠ 0 and Cmp(d, dRed) < 0 then
7:                  dRed ← d, eRed ← e
8:              if color(e) · (0, 1, 0) ≠ 0 and Cmp(d, dGreen) < 0 then
9:                  dGreen ← d, eGreen ← e
10:             if color(e) · (0, 0, 1) ≠ 0 and Cmp(d, dBlue) < 0 then
11:                 dBlue ← d, eBlue ← e
12:     dRed ← EdgeSignedPseudoDistance(P, eRed)
13:     dGreen ← EdgeSignedPseudoDistance(P, eGreen)
14:     dBlue ← EdgeSignedPseudoDistance(P, eBlue)
15:     return distanceColor((dRed, dGreen, dBlue))
```

Another property to note is that by taking only the median channel at each cell, we would get the exact pseudo-distance field from Section 4.2.3. This has been confirmed by tests. Because of this, the approximately correct signed distance can be sampled from anywhere in the field.

4.4.3 Collision correction

Just like the indirect method, it has been shown that this method causes tiny artifacts in certain situations. These only appear in areas where two false edges are too close together, and fortunatelly can be quite easily corrected.

For this, an additional pass of the output distance field is needed. False edges can be detected by the hard jump from a large negative to a large positive distance value between two adjacent cells. If this is detected for at least two color channels in a single cell, this area might cause artifacts. Such a cell shall be flagged for collision correction.

The correction itself is even simpler. We have established that the median component of each cell is guaranteed to be the exact pseudo-distance, so we can simply use this median value for all three components in the flagged cells. This means that the critical cells will work as in a single-channel distance field, losing their quality enhancing properties, but avoiding to cause artifacts.

This correction routine has shown to work well in practice, but the threshold difference between adjacent signed distance values indicating a false edge had to be adjusted carefully in order for it to detect all artifacts but also minimize false positives, which would cause the distance field to degenerate into the original single-channel variant.

Chapter 5

Application

In this chapter, I will cover how the signed distance fields can be used to draw text or other vector shapes. Since the technique is intended for use in real-time graphics, I will focus on its realization using the programmable real-time graphics pipeline and shaders.

5.1 Shape reconstruction

First of all, we will look at the complete procedure of reconstructing a raster image of the original shape from a signed distance field. This is done by determining at each pixel of the raster image, whether or not it lies inside the shape. Let's assume that using the correct transformation, we have computed that the current pixel lies at point P in the coordinate system of the distance field's grid. Now, we need to sample a value from that position in the distance field. The most common sampling methods are bilinear and bicubic. We will use bilinear sampling as it is the preferred option for this application of distance fields. [6]

Algorithm 8 Bilinear sampling of the distance field.

1: **function** SampleBilinear(sdf, P)
2: $x_1 \leftarrow \lfloor P_x - \frac{1}{2} \rfloor$
3: $y_1 \leftarrow \lfloor P_y - \frac{1}{2} \rfloor$
4: $x_2 \leftarrow x_1 + 1$
5: $y_2 \leftarrow y_1 + 1$
6: $w_x \leftarrow P_x - x_1 - \frac{1}{2}$
7: $w_y \leftarrow P_y - y_1 - \frac{1}{2}$
8: **return** $(1 - w_x)(1 - w_y)sdf(x_1, y_1) + w_x(1 - w_y)sdf(x_2, y_1)$
$+ (1 - w_x)w_y sdf(x_1, y_2) + w_x w_y sdf(x_2, y_2)$

Now that we have sampled a value from the distance field, we can convert

it back to a signed distance, using the inverse of the distanceColor function:

$$\text{colorDistance}(\text{distanceColor}(x)) = x \tag{5.1}$$

$$\text{colorDistance}(color) \overset{\text{def}}{=} \left(\frac{color}{maxColor} - \frac{1}{2} \right) \cdot distanceRange \tag{5.2}$$

The following is the basic procedure that generates pixels of the image reconstruction, using *insideColor* for pixels inside the shape and *outsideColor* for pixels outside.

Algorithm 9 Pixels of the image reconstructed using a single-channel SDF.

```
1: function GeneratePixel(P)
2:     sample ← SampleBilinear(sdf, P)
3:     d ← colorDistance(sample)
4:     if d ≥ 0 then
5:         return insideColor
6:     else
7:         return outsideColor
```

5.1.1 Reconstruction from a multi-channel distance field

With the median of three model, the reconstruction process is almost equally simple, with only one additional step. In this case, the result of the sampling won't be a scalar value, but a vector, where each component is the sample of one color channel. Immediately after acquiring this vector, its median component can be extracted and used as the scalar value from before. Its conversion to signed distance and the rest of the procedure will be the same:

Algorithm 10 Pixels of the image reconstructed using multi-channel SDF.

```
1: function GeneratePixel(P)
2:     s⃗ ← SampleBilinear(sdf, P)
3:     sample ← median(s_R, s_G, s_B)
4:     d ← colorDistance(sample)
5:     if d ≥ 0 then
6:         return insideColor
7:     else
8:         return outsideColor
```

5.1.2 Anti-aliasing

Instead of always using either the inside color or the outside color, we could smoothly blend from one to the other at the edge, thereby eliminating hard pixelated edges and achieving anti-aliasing. For this, we need to choose

a threshold value t, and for distance values in the interval $\langle -t, t \rangle$, use the weighted average of the two colors:

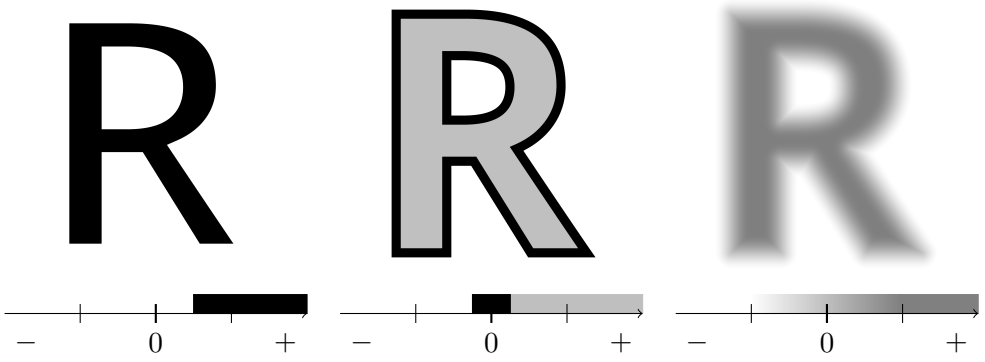
Algorithm 11 Pixels of the reconstructed image with anti-aliasing

1: **function** GeneratePixel(P)
2: $\vec{s} \leftarrow$ SampleBilinear(sdf, P)
3: $sample \leftarrow \text{median}(s_R, s_G, s_B)$
4: $d \leftarrow \text{colorDistance}(sample)$
5: $w \leftarrow \text{clamp}\left(\frac{d}{t}, -1, 1\right)$
6: **return** $\frac{1-w}{2} outsideColor + \frac{1+w}{2} insideColor$

The value of t should be chosen so that the interval $\langle -t, t \rangle$ in signed distance units is about as wide as a single pixel in the target bitmap.

5.1.3 Special effects

By using a different transformation from the signed distance d to the pixel color, a number of different visual effects can be achieved. Some examples of such transformations are shown in Figure 5.1. The used transformation from signed distance to color can be seen at the bottom.



(a) Thickness adjustment (b) Outline (c) Soft shadow

Figure 5.1: Examples of distance based visual effects.

5.1.4 Usage in OpenGL

The intended usage of this rendering method is in real-time graphics, where a graphics library such as OpenGL [8] is commonly used. With these libraries, the signed distance field has to be loaded into a 2D texture, and its evaluation happens in a pixel (fragment) shader. The graphics library is capable of performing the bilinear sampling routine by itself, since it is a very common

task in 3D graphics. This is one of the reasons why distance fields are a natural choice in this context.

The pixel or fragment shader basically has to perform something very similar to our GeneratePixel methods. The main difference is that in a 3D scene, the scale of the distance field in the resulting image might not be constant throughout, and therefore the threshold t cannot be determined globally before rendering. Instead, it has to be computed separately at each pixel or fragment. Fortunatelly, the OpenGL Shading Language (GLSL) offers the `fwidth` function for this task, which returns the amount of change of a variable between adjacent pixels.

Assuming that the correct position P has been computed in the vertex shader, the following is the complete OpenGL fragment shader that renders a shape encoded by a multi-channel distance field stored in a texture, and applies anti-aliasing. It is roughly equivalent to Algorithm 11.

```
varying vec2 P;
uniform sampler2D sdf;
uniform vec4 outsideColor;
uniform vec4 insideColor;

// Computation of the median value using minima and maxima
float median(float a, float b, float c) {
    return max(min(a, b), min(max(a, b), c));
}

void main() {
    // Bilinear sampling of the distance field
    vec3 s = texture2D(sdf, P).rgb;
    // Acquiring the signed distance
    float d = median(s.r, s.g, s.b) - 0.5;
    // The anti-aliased measure of how "inside" the fragment lies
    float w = clamp(d/fwidth(d) + 0.5, 0.0, 1.0);
    // Combining the two colors
    gl_FragColor = mix(outsideColor, insideColor, w);
}
```

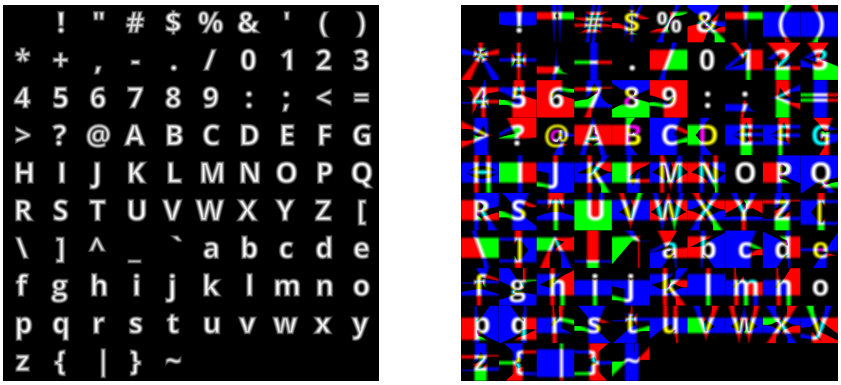
The shader can be easily modified or expanded in many ways, for example to make some parameters ajustable, or to support the visual effects mentioned before.

The shader is also backwards compatible with single-channel distance fields, where all three components of `s` would be equal, and although the call to the median function would be unnecessary, the result would be the same. This again shows that one simple median calculation is the only additional operation required in my multi-channel distance field rendering method.

5.2 Text rendering

So far, we have covered how to reconstruct the image of a single shape from its distance field representation. To display text however, the images of individual glyphs have to be composed together.

Typically in real-time graphics, the distance fields of glyphs of the entire character set will be compiled into a single texture. Each glyph will be located at a known position in this texture. Examples of such textures are shown in Figure 5.2.



(a) Single-channel. (b) Multi-channel.

Figure 5.2: SDF textures, from which individual glyphs can be extracted.

Note that much of the space in these textures is empty due to a regular but inefficient distribution of the glyphs. To utilize the maximum possible available space of the texture, a 2D bin packing algorithm is often used for this task.

A text can be rendered as a sequence of rectangles whose vertices are correctly positioned and correctly mapped to the glyph positions inside the texture. All of the necessary metrics can be retrieved from the definition of the font. Figure 5.3 shows an example application of this principle in the form of a triangle mesh.

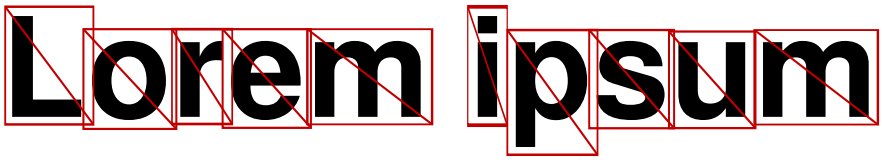


Figure 5.3: A string of text as a textured triangle mesh.

Chapter 6

Results

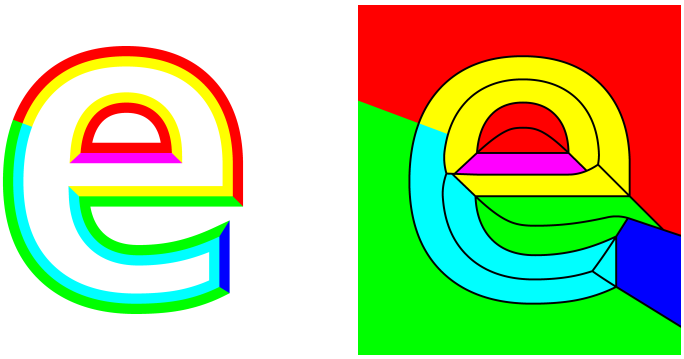
In this chapter, I will evaluate the outputs of the devised algorithms and measure the changes in rendering quality and performance.

6.1 Outputs

First of all, we will examine the actual outputs of both the indirect decomposition algorithm and the direct distance field construction algorithm.

6.1.1 Outputs of the indirect decomposition method

Before jumping to the finished outputs, we will look at the steps that lead there. Let's take the letter lowercase "e" from the Open Sans bold typeface for example. In Figure 6.1a, we can see the inner and outer edge colors assigned according to the rules of the median of three model, using the procedure described in Section 3.3.1. Notice that the glyph's longest edge on the very



(a) Edge coloring (b) Plane partitioning

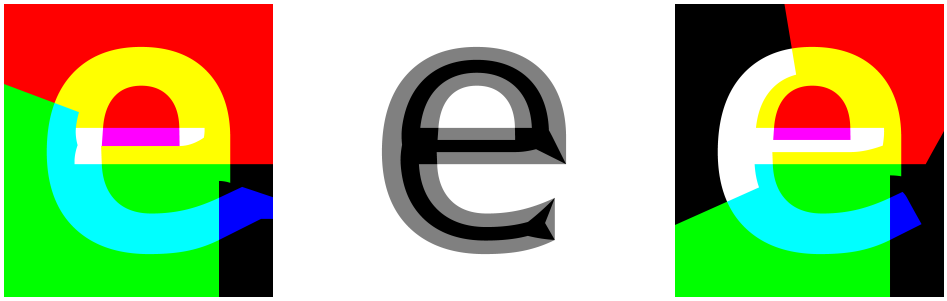
Figure 6.1: The edge coloring (a) and plane partitioning (b) of the letter "e".

left had to be divided in the middle in order to connect the beginning and the end of the contour.

In Figure 6.1b, we can see how the plane is divided between the edges. This is basically the Voronoi plane partitioning. Pixels that are closest to a certain edge have that edge's (or half-edge's) inner or outer color in this image, depending on whether the pixel lies inside the shape or outside.

Using this partitioning, the corners can now be divided into the four quadrants, which shall be colored according to the incident edges' colors. Figure 6.2a shows this finished decomposition.

The next problem is resolving the possible clashes described in Section 3.3.2 using padding. Figure 6.2b marks the areas inside the glyph designated as padding in black. This is using the distance ratio method, so all areas, where another edge is less than twice as far as the closest edge, will be shared by all such edges. If they happen to have the same color, it will not have any effect. Figure 6.2c shows the decomposition including this inter-edge padding, which is mainly noticeable in the central horizontal stroke. Padding has also been added in the middle of the divided leftomst edge, separating the colors of the half-edges.



(a) Decomposition (b) Inner padding (c) Padded decomposition

Figure 6.2: The decomposition of the letter "e".

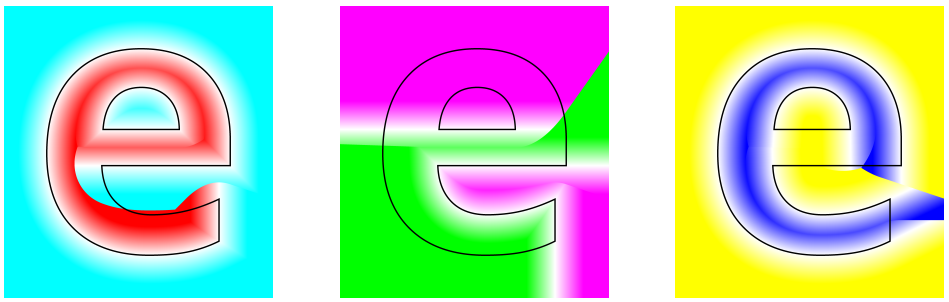
Some other examples of glyph decompositions can be found in Figure C.2.

6.1.2 Outputs of the direct decomposition method

Now let's look at the outputs of the direct multi-channel distance field construction method. Again, the first step is edge coloring, shown in Figure 6.3 on the lowercase letter "e" as an example. This time however, each edge has only one color.

Using this edge coloring, the multi-channel signed distance field can be constructed directly. Figure 6.4 shows the three individual color channels of this distance field along with the glyph's real outline in black. White color denotes zero, red, green, and blue, respectively, are positive distance values, and cyan, magenta, and yellow, respectively, are negative distance values.

Figure 6.3: The edge coloring of the letter “e”.



(a) Red channel (b) Green channel (c) Blue channel

Figure 6.4: The individual channels of the resulting distance field.

Notice that each component of the distance field behaves like a regular distance field near the edges whose color includes the channel of that component, and completely ignores the rest. The red component, for example, uses the pseudo-distances to the yellow and magenta edges, but ignores the cyan edge.

Visualizing the combined components of the distance field meaningfully would be problematic, but let’s look at the combined reconstruction of the three distance fields instead in Figure 6.5.

Figure 6.5: The combined reconstruction of the distance field’s components.

This image is comparable to the indirect decomposition (Figure 6.2c), except this time, the distance field doesn't properly encode some of the false edges, which aren't part of the shape's outline. You can see the corner quadrants have been correctly filled according to the median of three model (Section 3.2.2), and therefore applying the median function will yield an image of the letter "e" with sharp corners.

6.2 Rendering quality

In this section, I will measure the factual improvement in quality of rendering using my multi-channel distance field technique, and compare the results with the original single-channel version.

6.2.1 Methodology of quality measurement

We will need a method of quantifying the image quality in order to measure it and compare the results. Using a distance field with given dimensions and an exact representation of the original shape, we will sample the distance field at a very large number of points (tens of millions), which is equivalent to producing a high resolution reconstruction of the shape, and at each point, we will observe several indicators.

6.2.1.1 Pixel mismatch (PM)

This is the simplest metric. At each point, it will be determined whether it lies inside the shape according to the exact vector representation and the distance field. The resulting value will be the portion of the points where the two values do not match. For example, the value 0.1 signifies that in 90% of the area the pixels will be filled correctly, and in 10% not.

6.2.1.2 Weighted pixel mismatch (WPM)

This metric is designed to distinguish more serious artifacts. When pixels don't match exactly around edges, it is usually due to the limited precision of the distance field format. However, when chunks of incorrect pixels start to appear relatively far away from edges, or when and edge becomes severely misplaced, it may result in a noticeable distortion of the image. In this metric, the absolute distances at which the mismatches occurred are summed, so that mismatching pixels farther from the outline have greater weight.

6.2.1.3 Weighted distance difference (WDD)

To evaluate the distance field's precision in other areas than just around the outline, which is important for visual effects that use signed distance values, I have also added a metric that measures the difference in reported and actual

signed distance. Since the distance values are less likely needed further away from the outline, the difference will be given a greater weight the closer to the outline it is. The value is computed as

$$\sum_{P} |d - d_S| \mathrm{e}^{-\frac{|d|}{k}}, \tag{6.1}$$

where P are the sampled points, d is the correct signed distance, d_S is the signed distance sampled from the distance field, and k is an adjustable weight distribution parameter, which I set to 60.

6.2.2 Intermediate decomposition resolution

Before comparing the quality of the original and improved methods, we must determine the optimal configuration of the algorithms. The most significant parameter here is the resolution of the intermediate decomposition for the indirect method. Obviously, increasing this resolution will increase precision, but also the computation time. Therefore, we need to find a compromise between the two, a point where increasing the resolution any further only causes a marginal change in output quality.

The first quality measurement is focused on determining this optimal ratio between the size of the intermediate decomposition and the target distance field. I used the ASCII character set of the Open Sans bold typeface, and a fixed size of the distance field, large enough to represent the glyphs with only minor imprecisions. *Reference* is the quality measurement for the original single distance field method. The other rows are the results of the indirect decomposition algorithm with different resolution ratios.

Ratio	Average error		Time (s)
	PM	WPM	
Reference	0.000721 ± 0.00029	0.00158 ± 0.0010	0.376
2	0.00745 ± 0.0017	0.0188 ± 0.0062	9.349
4	0.00322 ± 0.00091	0.00371 ± 0.0018	21.727
8	0.00158 ± 0.00045	0.000911 ± 0.00039	58.040
16	0.000665 ± 0.00028	0.000216 ± 0.00012	176.996
32	0.000282 ± 0.00019	0.0000522 ± 0.000053	575.024
64	0.000222 ± 0.00018	0.0000420 ± 0.000051	2132.878
128	0.000204 ± 0.00019	0.0000397 ± 0.000050	7773.227
131	0.000283 ± 0.00019	0.0000452 ± 0.000049	8128.431

Table 6.1: Rendering quality for different intermediate resolutions.

As you can see in Table 6.1, the error doesn't significantly decrease beyond about 64 times the dimensions of the target distance field, while the total

computation time (last column) starts to become impractical. Therefore, I will use this value as the default for the indirect method in the following tests.

The measurement in the last row (ratio 131) tests whether using a power of two has any impact on the precision, and since the error is much larger than both 2^7 and 2^6 (lower ratios), it is clear that using a power of two as the resolution ratio is in fact advantageous.

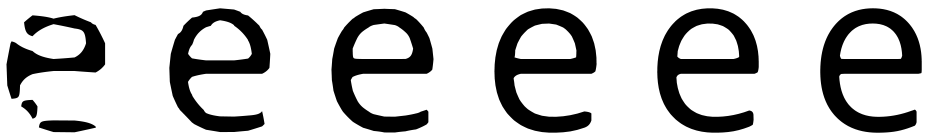
6.2.3 Comparison of reconstruction precision

Now, we finally get to the most important test, the measurement of the difference in quality of the resulting image, reconstructed from signed distance fields using the original and my two improved methods.

Original refers to the original single-channel pseudo-distance method, **indirect** is my multi-channel method with an intermediate raster decomposition (of resolution 64 times higher in each dimension than the target SDF resolution – unless stated otherwise), and **direct** is my direct multi-channel distance field construction method.

In the following tests, I will use the average error of the 94 printable ASCII characters from the Open Sans typeface, using the regular (R), bold (**B**), and light (L) variants.

Figure 6.6 shows the range of the distance field resolutions I will be using. The shape in the leftmost image (resolution 7×8 pixels) is already disintegrating, and therefore going any lower would be pointless. The rightmost image (43×50 pixels) on the other hand seems to possess a satisfying precision, apart from the sharpness of corners.



(a) Size 8 (b) Size 12 (c) Size 16 (d) Size 24 (e) Size 32 (f) Size 48

Figure 6.6: Reconstruction of the letter “e” from a single-channel pseudo-distance field of varying resolutions.

The results for the absolute pixel mismatch are in Table 6.2. We can see that at smaller sizes of the distance field, and especially with the light variant of the typeface, all of the methods have problems encoding the glyphs correctly. This is because the strokes of the glyphs are too thin relatively to the density of the distance field grid. The indirect method suffers from this the most, as when the two parallel edges of a stroke have different colors, the stroke will be divided into even thinner sub-strokes. Because of this “thinning”, the indirect method performs even worse than the original in one instance.

SDF size	Style	Average pixel mismatch (PM)		
		Original	Indirect	Direct
8	L	0.0842 ± 0.028	0.0818 ± 0.029	0.0706 ± 0.034
8	R	0.0864 ± 0.031	0.0849 ± 0.031	0.0713 ± 0.038
8	**B**	0.0390 ± 0.022	0.0289 ± 0.025	0.0265 ± 0.021
12	L	0.0689 ± 0.018	0.0657 ± 0.021	0.0512 ± 0.021
12	R	0.0217 ± 0.0082	0.0189 ± 0.011	0.0130 ± 0.0079
12	**B**	0.0124 ± 0.0051	0.00514 ± 0.0049	0.00419 ± 0.0042
16	L	0.0284 ± 0.016	0.0303 ± 0.017	0.0203 ± 0.016
16	R	0.00755 ± 0.0031	0.00501 ± 0.0035	0.00325 ± 0.0026
16	**B**	0.00637 ± 0.0025	0.00236 ± 0.0019	0.00203 ± 0.0020
24	L	0.00454 ± 0.0018	0.00411 ± 0.0031	0.00206 ± 0.0013
24	R	0.00322 ± 0.0012	0.00104 ± 0.00086	0.000857 ± 0.00088
24	**B**	0.00303 ± 0.0011	0.000857 ± 0.00078	0.000792 ± 0.00082
32	L	0.00180 ± 0.00067	0.000944 ± 0.00063	0.000553 ± 0.00053
32	R	0.00171 ± 0.00064	0.000642 ± 0.00045	0.000465 ± 0.00048
32	**B**	0.00163 ± 0.00062	0.000589 ± 0.00042	0.000433 ± 0.00044
48	L	0.000797 ± 0.00032	0.000270 ± 0.00021	0.000213 ± 0.00022
48	R	0.000741 ± 0.00030	0.000247 ± 0.00019	0.000204 ± 0.00021
48	**B**	0.000717 ± 0.00029	0.000222 ± 0.00018	0.000190 ± 0.00020

Table 6.2: Comparison of absolute rendering quality of the distance field techniques.

For the lower SDF resolutions, we can generally only see a small improvement. Looking at the absolute values, it is obvious that some of these resolutions are simply too low to encode the glyph properly. A mismatch value of 0.5 is the equivalent of filling the pixels randomly. The mismatch of two completely different letters, A and Z for example, is about 0.25.

Let's agree that the SDF resolution is acceptable if at least 99% of the pixels if the original method match (PM < 0.01). In these cases, the indirect method reduces the amount of mismatched pixels on average 2.59 times, and the direct method 3.38 times. The improvement is most significant for characters that contain little to no curves – for some of those, the direct method even reaches zero error.

As we are now able to properly reconstruct corners, reconstruction of curves becomes the next biggest issue, which isn't addressed by my decomposition technique. In Table 6.3, we can see the results of the same measurement for the subset of glyphs that consist of straight line segments only. Notice that the average for the original method is almost the same as before. This shows that corners are a much bigger issue than imprecise curvature. The multi-channel methods however exhibit significantly lower error values.

SDF size	Average pixel mismatch (PM)				
	Original	Indirect		Direct	
16	0.00724	0.00352	± 0.0025	0.00145	± 0.0020
24	0.00322	0.000246	± 0.00025	0.0000243	± 0.000056
32	0.00171	0.000243	± 0.00015	0.00000763	± 0.000028
48	0.000709	0.0000607	± 0.000063	0.00000038	± 0.0000013

Table 6.3: Comparison of rendering quality for non-curved glyphs only.

In conclusion, my decomposition methods decrease the error in rendering straight lines and sharp corners by several orders of magnitude, but do not improve the rendering of curves. Figures 6.7 and 6.8 offer a comparison of the reconstruction of several glyphs using the original and my improved technique.

Figure 6.7: Comparison of the reconstruction of several glyphs of varying thickness using the original (top) and my direct (bottom) method.

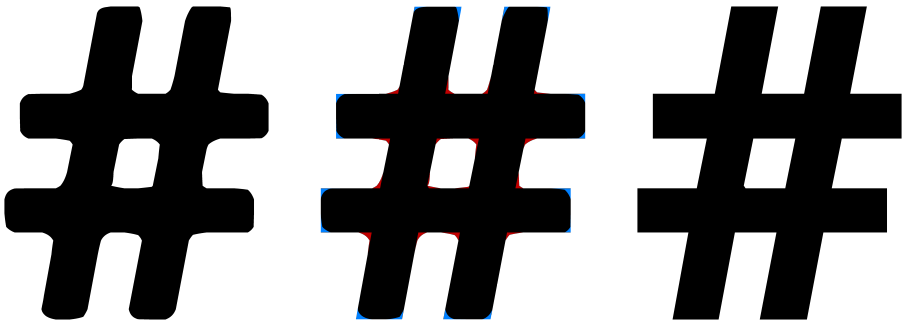


Figure 6.8: Detail of the hash symbol reconstructed using the original (left) and my direct (right) method, and the difference between the two (center).

Next, we will evaluate the weighted pixel mismatch for the three methods, which should be a better indicator of the apparent image quality. Any artifacts in the form of isolated islands of mismatching pixels will be punished more severely, along with significant extrusions or indents of the outline. These imprecisions have the biggest impact on the resulting image quality.

SDF size	Style	Average weighted pixel mismatch (WPM)		
		Original	Indirect	Direct
8	R	2.52 ± 1.2	2.60 ± 1.2	2.12 ± 1.4
12	R	0.229 ± 0.13	0.269 ± 0.31	0.123 ± 0.13
16	R	0.0462 ± 0.025	0.0262 ± 0.041	0.0126 ± 0.018
24	L	0.0181 ± 0.011	0.0210 ± 0.028	0.00522 ± 0.0082
24	R	0.0139 ± 0.0080	0.000945 ± 0.0014	0.000652 ± 0.00097
24	**B**	0.0135 ± 0.0074	0.000756 ± 0.0012	0.000743 ± 0.0012
32	L	0.00524 ± 0.0030	0.000852 ± 0.00099	0.000396 ± 0.00072
32	R	0.00522 ± 0.0032	0.000243 ± 0.00032	0.000196 ± 0.00029
32	**B**	0.00525 ± 0.0034	0.000234 ± 0.00029	0.000209 ± 0.00027
48	L	0.00176 ± 0.0011	0.000055 ± 0.00008	0.000040 ± 0.00008
48	R	0.00155 ± 0.0010	0.000041 ± 0.00006	0.000037 ± 0.00006
48	**B**	0.00156 ± 0.0010	0.000042 ± 0.00005	0.000039 ± 0.00005

Table 6.4: Comparison of apparent rendering quality of the distance field techniques.

We can see the results, this time for the entire ASCII character set again, in Table 6.4. Again, almost no improvement can be seen at the lower sizes, where the distance field resolution is insufficient. At higher sizes however, the errors are up to about 40 times lower using one of the multi-channel methods, which is a very significant improvement.

We can also see that the indirect method performs much worse than the direct one for the light font variant up until the highest resolution. At size 24, the error is actually higher than the original. This is caused by the method's tendency to divide the already very thin strokes into even thinner sub-strokes.

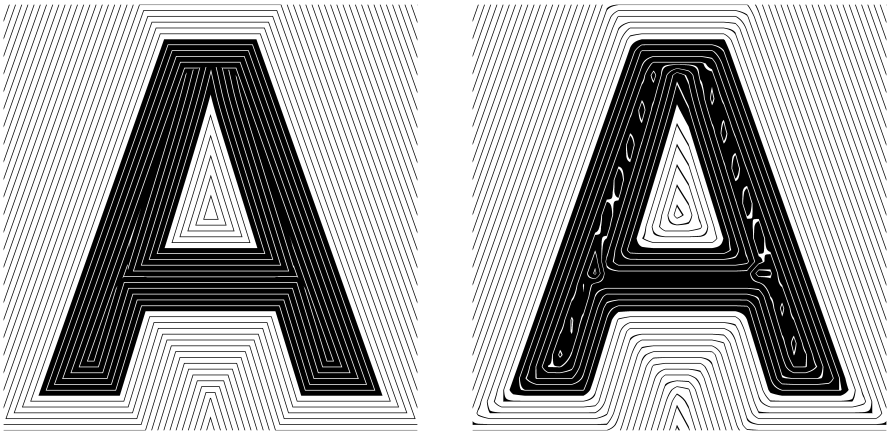
6.2.4 Preservation of distance information

In some cases, the correct signed distance may be needed further away from the outline. This capability will be tested next using the weighted distance difference metric.

Table 6.5 shows that the direct decomposition method encodes the signed distance with approximately half the original error. The indirect method unfortunately isn't designed for this functionality, and therefore performs worse than the original.

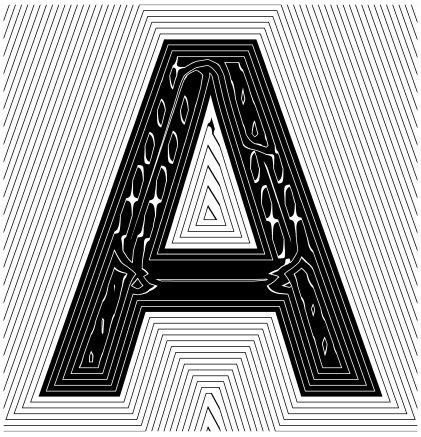
SDF size	Average weighted distance difference (WDD)		
	Original	Indirect	Direct
12	2.180 ± 0.607	1.877 ± 0.670	1.434 ± 0.654
16	1.281 ± 0.320	1.167 ± 0.402	0.7575 ± 0.337
24	0.6016 ± 0.163	0.6449 ± 0.225	0.3290 ± 0.157
32	0.3198 ± 0.0864	0.4629 ± 0.171	0.1699 ± 0.0840
48	0.1447 ± 0.0400	0.3401 ± 0.146	0.07638 ± 0.0374

Table 6.5: Comparison of the error in sampled distance values throughout the entire plane.

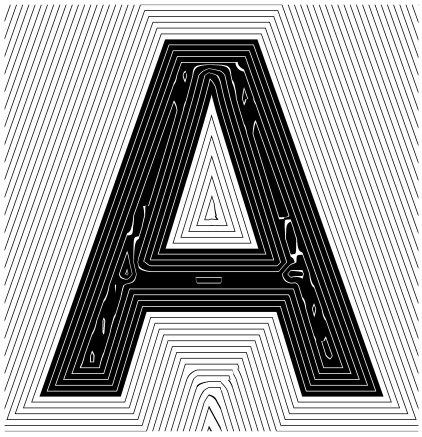


(a) Reference

(b) Original



(c) Indirect



(d) Direct

Figure 6.9: Contour diagram of the letter "A" constructed exactly (a), and reconstructed from distance fields (b, c, d).

Figure 6.9 shows the signed pseudo-distance reconstruction of a single glyph in the form of equidistant contour lines. In this example, we can see the precision is lowest at the center of the glyph's strokes, where the distance field forms a "ridge". At the outline, and in its vicinity, the distance is represented accurately for both of the multi-channel techniques. The direct method preserves the sharp angles even further outside the shape, but gradually loses precision. The indirect method is the most precise in the portion below the glyph, but behaves completely incorrectly inbetween edges. Notice that in some parts, most noticeably in the upper left, it forms a "double-ridge", which is a common occurence for this method.

6.3 Performance

The only remaining aspect to evaluate is the performance of the algorithms. This includes the cost of running the algorithms themselves, and the cost of using their outputs to render images.

Since the distance fields can be pre-generated once, and come packaged with the target program, their construction time is less important. In some cases however, it might be required to construct the distance fields for individual characters dynamically at runtime, for example when the character set is too large, and only an unknown small portion is expected to be used. In these cases, the generation of the distance fields has to be fast as well.

6.3.1 Distance field construction time

Table 6.6 shows the total time it took to construct distance fields of varying resolutions for all ASCII characters of the Open Sans typeface, with the use of the three different methods. Please note that the implementation of the indirect algorithm isn't optimal, because it has been deprecated by the direct algorithm. Also, since these values have been measured on one specific

	Total time (seconds)		
SDF size	Original	Direct	Indirect
8	0.010	0.011	45.013
12	0.022	0.026	105.28
16	0.039	0.045	192.87
24	0.088	0.099	448.78
32	0.158	0.176	827.47
48	0.355	0.394	1942.1

Table 6.6: Total construction time of distance fields for all ASCII characters of a font using different methods.

machine (Intel i5 3570K), the absolute values aren't objective by themselves, only their relative differences.

All of the algorithms have linear time complexity in respect to the number of pixels of the distance field (quadratic in respect to the SDF size units). The indirect algorithm however has a severe overhead of having to construct a high resolution image of the decomposition, which has (by default) $64 \times 64 = 4096$ times more pixels than the distance field, and then it has to compute the distance field from it. Although this computation could be implemented more efficiently, it will always be a significant overhead, and therefore the running time of the indirect method is several orders higher than the rest, making it not suitable for dynamic runtime distance field generation.

The computation time of the direct method on the other hand, is the lowest it could possibly be, only marginally higher than the construction of a regular single channel pseudo-distance field, which is the theoretical minimum. Although it factually has to construct three distance fields, it does so simultaneously, and therefore requires much less than three times the original time.

6.3.2 Real-time rendering performance

Since rendering performance is the original distance field technique's primary advantage, it is extremely important that this advantage isn't lost. To test this, I have created an OpenGL program, where an enormous amount of text is drawn on the screen from a distance field. I have designed it so that as little extra time as possible is spent with other tasks than the rendering itself. The text is drawn in batches of about 32 characters per draw call, and no uniform variables or other states are changed in between, only the vertex array object. The vertex shader only performs a single matrix transformation of the coordinates, and for the fragment shader, variants of the implementation from Section 5.1.4 have been used.

The measured framerates (frames per second) are shown in Table 6.7.

Frames per second	Distance field dimensions			
Distance field type	256	512	1024	2048
Single-channel (R)	30.812	28.604	26.435	21.020
Single-channel (RGB)	29.084	26.197	20.150	9.175
Multi-channel	27.183	24.762	19.559	9.161
Single-channel (R) + AA	28.427	26.333	24.221	18.977
Single-channel (RGB) + AA	26.647	23.979	18.425	8.418
Multi-channel + AA	25.047	22.863	17.957	8.406

Table 6.7: Comparison of text rendering framerates when using single-channel and multi-channel distance field textures.

Again, since the measurements have been performed on a specific machine (Nvidia GTX760), only the differences are of importance. The test has been performed with both single-channel and multi-channel distance fields in several resolutions, and with and without anti-aliasing (AA). Furthermore, the single-channel variant has been tested storing the texture both as only one channel (R), and as a regular 3-channel image (RGB).

As you can see, the performance impact of using a multi-channel distance field varies by the size of the distance field texture. For small resolutions, the decrease in framerate is only about 12%. For higher resolutions however, it seems that the sampling of the texture is a significantly slower operation, and interestingly, a very noticeable difference can be seen just in sampling a monochrome (R) versus a color (RGB) texture. The cost of the additional median computation is negligible in these cases, but the increase in the number of texture channels brings up to a 56% drop in performance.

The anti-aliasing routine seems like a relatively inexpensive addition to the rendering process, causing a slow of 8 to 10%.

In conclusion, the performance of the multi-channel distance field rendering method is 12 to 56 or possibly more percent worse than that of single-channel distance field rendering, depending on the distance field resolution. This result is slightly less optimistic than I anticipated, but hopefully still worth the increase in image quality.

An interesting observation is that since texture sampling is the most expensive operation, the original method is almost just as slow as the improved one when the single-channel distance field is needlessly stored as and sampled from a 3-channel texture.

Chapter 7

Conclusion

I have successfully created a solution for generating multi-channel signed distance fields of vector-based shapes, which serve as an approximate representation of said shapes that can be used for very efficient high quality real-time rendering.

Unlike the original single-channel distance field rendering technique, my improved method is capable of near-perfect reconstruction of the shape's corners, and because of this, the average error of the reconstructed image of the shape is lower by several orders of magnitude. The real-time rendering performace is only slightly worse for low distance field resolutions, but up to 56% worse for higher resolutions. However, since in some cases it can provide higher quality at a lower distance field resolution, it can allow for reduction of this resolution, bringing an improvement in both quality and performance.

I have developed two ways of constructing the multi-channel distance field, the indirect and the direct method. The indirect method was developed first, inspired by the original idea of decomposing the general shape into a combination of smooth shapes. With the findings and experience I have acquired during the development of this method, I was able to come up with a much more sophisticated, direct method of construction of the multi-channel distance field. The direct method has proven superior to the indirect one in all aspects, especially in the cost of its computation, which is only marginally higher than the computation of a single-channel distance field.

Because of my solution's quality improvement over the single-channel distance field technique, it could be used in place of the original technique in most scenarios, improving rendering quality without any major drawbacks. I see the biggest potential of my solution in rendering text in real-time 3D scenes, where performance is vital, and where the perspective can cause the text to appear very large, exposing any imperfections.

7.1 Future work

During the development of this work, I have come up with several ideas that could be explored further. There are also some details in my method of multi-channel decomposition that could be improved.

7.1.1 Improved coloring strategy

The first step of both the indirect and direct construction methods is assigning colors to the shape's edges. There are usually multiple possibilities of how the colors could be assigned. Depending on which one is used, the effectiveness of the decomposition can slightly vary.

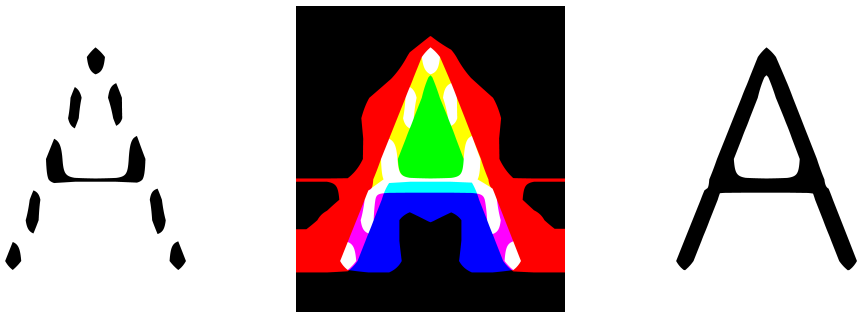
For the indirect method, we could for example choose a different starting edge that will be divided into halves of different colors, or maybe even dividing more than one would pay off in other areas. For the direct method, the only rule is that two consecutive edges cannot have the same color. This allows for many possible color sequences.

Since the impact is much less significant for the direct method, which essentially replaced the original indirect one, I haven't pursued this possibility any further.

7.1.2 Thickening decomposition

Although I have focused solely on improving the rendering quality of sharp corners, there are other disadvantages of the distance field rendering technique that could be solved by a multi-channel decomposition. One of them is the representation of thin features.

If a stroke is thinner than about two cells of the distance field grid across, it cannot be encoded properly and will probably be heavily distorted if at all visible in the reconstructed image (see Figure 7.1a). However, if the thin



(a) Single-channel (b) Multi-channel (c) Multi-channel (flat)

Figure 7.1: Image reconstruction of the glyph "A" with thin strokes.

feature were to be represented as a union of two much thicker strokes (one on each side) encoded in multiple channels, the issue could be resolved. This would reduce the minimum required resolution of the distance field to properly encode thinner fonts.

Figure 7.1 shows an example of this. On the left, you can see the result of using a low resolution single-channel distance field on a thin typeface. The center image demonstrates the reconstruction of the image using a thickening multi-channel decomposition. In addition to the white parts, which represent areas where all three color components are evaluated as inside, we can also use the yellow, cyan, and magenta areas, where only two components are. The final result after applying the median function is seen in Figure 7.1c. Although it isn't perfect, it certainly makes the character readable.

7.1.3 Models for higher number of channels

In this work, I have primarily used the median of three coloring model, which uses three channels – the normal amount for RGB images. However, I theorize that a higher number of channels could be utilized for an even higher reconstruction precision. One potential possibility would be the combination of corner preservation and thickening.

Bibliography

[1] Adobe Systems Incorporated. Font formats. `https://www.adobe.com/type/browser/info/formats.html`, 2006.

[2] Franz Aurenhammer and Rolf Klein. Voronoi diagrams. *Handbook of computational geometry*, 5:201–290, 2000.

[3] Heinz Breu, Joseph Gil, David Kirkpatrick, and Michael Werman. Linear time Euclidean distance transform algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(5):529–533, 1995.

[4] Behdad Esfahbod. GLyphy – high-quality glyph rendering using OpenGL ES2 shading language. `http://glyphy.org`, 2014.

[5] Gerald Farin. *Curves and surfaces for computer-aided geometric design: a practical guide*. Elsevier, 2002.

[6] Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 courses*, pages 9–18. ACM, 2007.

[7] Michael A. Jenkins and Joseph F. Traub. A three-stage algorithm for real polynomials using quadratic iteration. *SIAM Journal on Numerical Analysis*, 7(4):545–566, 1970.

[8] Khronos Group. OpenGL overview. `http://www.opengl.org/about/`, May 2015.

[9] Charles Loop and Jim Blinn. Resolution independent curve rendering using programmable graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 1000–1009. ACM, 2005.

[10] Gergely Patai. Playing around with distance field font rendering. `https://lambdacube3d.wordpress.com/2014/11/12/playing-around-with-font-rendering/`, November 2014.

[11] Zheng Qin, Michael D. McCool, and Craig S. Kaplan. Real-time texture-mapped vector glyphs. In *Proceedings of the 2006 symposium on interactive 3D graphics and games*, pages 125–132. ACM, 2006.

[12] Nicolas Ray, Xavier Cavin, and Bruno Lévy. Vector texture maps on the GPU. Technical report, Technical Report ALICE-TR-05-003, 2005.

[13] Aleksas Riškus. Approximation of a cubic Bézier curve by circular arcs and vice versa. *Information technology and control*, 35(4):371–378, 2006.

[14] Lee Thomason. TinyXML-2. `http://www.grinninglizard.com/tinyxml2/`, March 2015.

[15] David Turner, Robert Wilhelm, and Werner Lemberg. The FreeType project. `http://www.freetype.org/`, December 2014.

[16] Eric W. Weisstein. Abel's impossibility theorem. `http://mathworld.wolfram.com/AbelsImpossibilityTheorem.html`, 2002.

[17] Eric W. Weisstein. Cubic formula. `http://mathworld.wolfram.com/CubicFormula.html`, 2002.

[18] Eric W. Weisstein. Simply connected. `http://mathworld.wolfram.com/SimplyConnected.html`, 2002.

[19] World Wide Web Consortium. Paths – SVG 1.1 (second edition). `http://www.w3.org/TR/SVG/paths.html`, August 2011.

Appendix A

Glossary

anti-aliasing
A technique that attempts to reduce distortion arising from the discrete nature of pixels in a raster.

clamp
A function that yields the closest value to x in a given range $\langle a, b \rangle$:

$$\text{clamp}(x, a, b) = \min(\max(x, a), b) \tag{A.1}$$

contour
A cyclic sequence of edges forming an outline. A vector shape is composed of one or more contours.

glyph
The visual representation of a character in a specific typeface.

pseudo-distance
See Section 2.5.

segment
The building block of edges and contours. An edge segment can be either a line segment, or a Bézier curve.

shader
A program for graphics hardware that computes the color of each pixel of the rendered image.

spline
A smooth curved line that is composed of one or more primitive curves in a sequence.

texture
Representation of a raster image from which the color at a given point can be efficiently sampled by graphics hardware.

Appendix B

List of abbreviations

3D Three-dimensional

AA Anti-aliasing

ASCII American Standard Code for Information Interchange

GLSL OpenGL Shading Language

OpenGL Open Graphics Library

OTF OpenType font

PM Pixel mismatch

RGB Red, green, blue

SDF Signed distance field

SVG Scalable Vector Graphics

TTF TrueType font

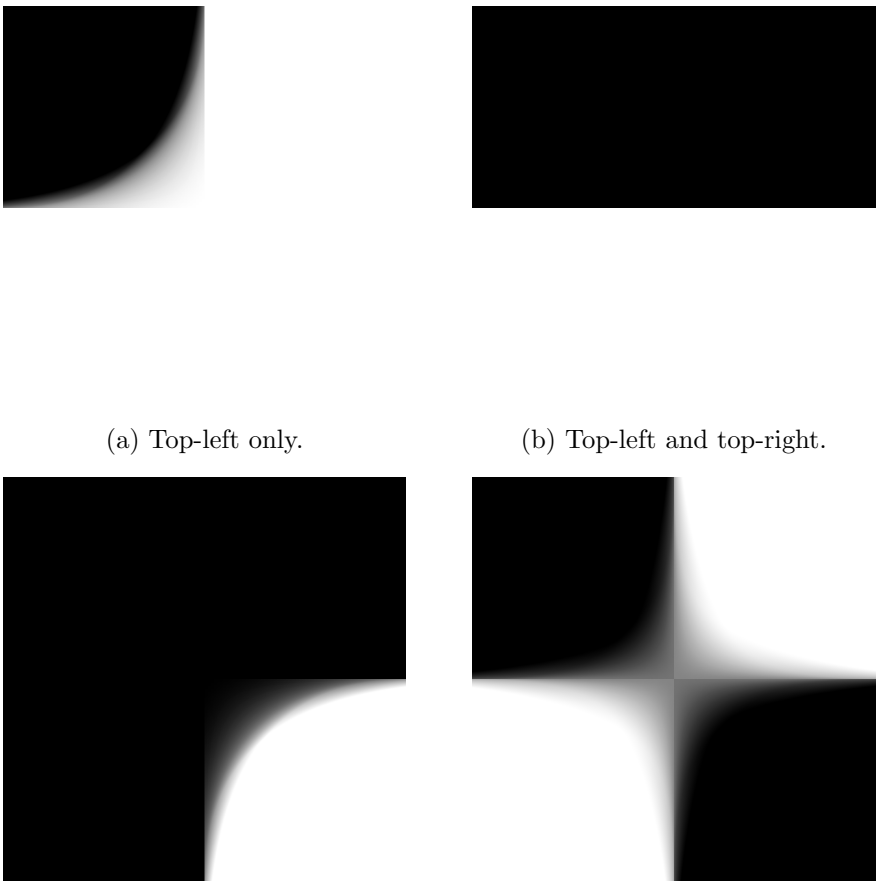
WDD Weighted distance difference

WPM Weighted pixel mismatch

XML Extensible Markup Language

Appendix C

Gallery



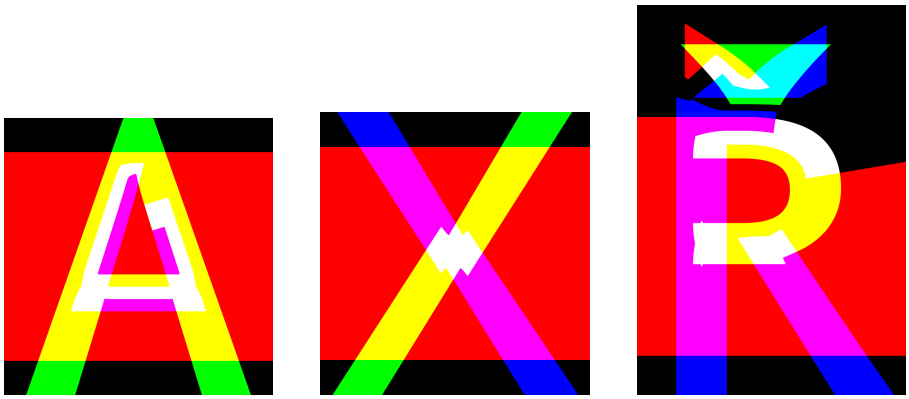
(a) Top-left only.

(b) Top-left and top-right.

(c) All but bottom-right.

(d) Top-left and bottom-right.

Figure C.1: The average of all possible results of corner quadrant reconstruction with varying distance field grid alignment.



(a) Uppercase “A” (b) Uppercase “X” (c) Uppercase “Ř”

(d) Hash (e) Ampersand (f) At sign

Figure C.2: Some examples of multi-channel decomposition outputs.

Appendix D

Contents of the enclosed CD

- `thesis.pdf` ..... the thesis in PDF format
- `readme.html` ..... description of contents and user manual
- `bin` ..... Windows binaries
- `freetype` ..... the prerequisite FreeType library for loading fonts
- `gallery` ..... additional outputs and visualizations in high definition
- `src` ..... source code
  - `decomposition.h` ..... main header file of the library
  - `main.cpp` ..... a console program wrapping the library's functionality
  - `core` ..... the core algorithms with no external dependencies
  - `io` ..... additional code for loading input and saving image output