Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Fast and approximate textrendering using distance fields

Examensarbete utfört i Informationskodning
vid Tekniska högskolan vid Linköpings universitet
av

Gustav Adamsson

LiTH-ISY-EX--15/4868--SE

Linköping 2015

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Fast and approximate textrendering using distance fields

Examensarbete utfört i Informationskodning
vid Tekniska högskolan vid Linköpings universitet
av

Gustav Adamsson

LiTH-ISY-EX--15/4868--SE

Handledare: **Jens Ogniewski**
ISY, Linköpings universitet
Mattias Wingstedt
VISIARC

Examinator: **Ingemar Ragnemalm**
ISY, Linköpings universitet

Linköping, 9 april 2015

Avdelning, Institution
Division, Department

informationskodning
Department of Electrical Engineering
SE-581 83 Linköping

Datum
Date

2015-04-09

Språk
Language

□ Svenska/Swedish
☒ Engelska/English
□ _____

Rapporttyp
Report category

□ Licentiatavhandling
☒ Examensarbete
□ C-uppsats
□ D-uppsats
□ Övrig rapport
□ _____

ISBN
__

ISRN
LiTH-ISY-EX--15/4868--SE

Serietitel och serienummer
Title of series, numbering

ISSN
__

URL för elektronisk version

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-XXXXX

Titel
Title

Snabb och approximativ textrendrering med distance fields

Fast and approximate textrendering using distance fields

Författare
Author

Gustav Adamsson

Sammanfattning
Abstract

Distance field text rendering has many advantages compared to most other text rendering solutions. Two of the advantages are the possibility to scale the glyphs without losing the crisp edge and less memory consumption. A drawback with distance field text rendering can be high distance field generation time. The solution for fast distance field text rendering in this thesis generates the distance fields by drawing distance gradients locally over the outlines of the glyphs. This method is much faster than the old exact methods for generating distance fields that often includes multiple passes over the whole image.

Using the solution for text rendering proposed in this thesis results in good looking text that is generated on the fly. The distance fields are generated on a mobile device in less than 10 ms for most of the glyphs in good quality which is less than the time between two frames.

Nyckelord
Keywords

Tex trendering, distance field, distance map, distance transform

Abstract

Distance field text rendering has many advantages compared to most other text rendering solutions. Two of the advantages are the possibility to scale the glyphs without losing the crisp edge and less memory consumption. A drawback with distance field text rendering can be high distance field generation time. The solution for fast distance field text rendering in this thesis generates the distance fields by drawing distance gradients locally over the outlines of the glyphs. This method is much faster than the old exact methods for generating distance fields that often includes multiple passes over the whole image.

Using the solution for text rendering proposed in this thesis results in good looking text that is generated on the fly. The distance fields are generated on a mobile device in less than 10 ms for most of the glyphs in good quality which is less than the time between two frames.

Acknowledgments

I would like to thank VISIARC and the employees at VISIARC for letting me do my thesis work at the company and for helping me when needed. I would also like to thank Ingemar Ragnemalm for supporting me through the implementation of this thesis.

Contents

1 Introduction **1**
1.1 Problem formulation 1
1.2 Structure 1

2 Background **3**
2.1 Computer graphics 3
2.2 Basic text rendering 4
2.3 Distance fields 6
2.4 Beziér curves 7
2.5 Polygon filling 10

3 Related work **13**
3.1 Early EDT algorithms 13
3.2 Exact EDT algorithms 14
3.3 Improved distance measure for EDT algorithms 15

4 Methods for implementation **17**
4.1 Distance field generation initial attempt 17
4.2 Fast and approximate distance field generation 18
4.2.1 Beziér to line segment approximations 18
4.2.2 Drawing the glyph 20
4.2.3 Drawing distance gradients 21
4.3 Distance field rendering 24

5 Methods for evaluation of the implementation **27**
5.1 Visual evaluation 1 27
5.2 Visual evaluation 2 28
5.3 Performance evaluation 28
5.4 Distance tolerance evaluation 28

6 Result **29**
6.1 Visual evaluation 1 29
6.2 Visual evaluation 2 31

6.3 Performance evaluation 31
6.4 Distance tolerance evaluation 32

7 Discussion and conclusions **35**
7.1 Result 35
7.2 Method 38
7.3 Memory 38

8 Future work **41**

A Result from visual evaluation #1 **45**

B Result from performance evaluation **49**

C Result from distance tolerance evaluation **55**

Bibliography **57**

1 Introduction

It is hard to create a text rendering solution that is fast, dynamic and outputs good looking text. Distance field text rendering was introduced for the public in an article from Valve 2007 and is used in many applications today. The goal with this thesis was to evaluate the distance field text rendering and implement it in a cross-plattform development tool used by VISIARC.

1.1 Problem formulation

This section covers the problem formulations which were set up at the beginning of this thesis. The problem formulation has functioned as guidelines throughout the work of the thesis.

- How can distance fields be used when rendering text and how does the technique work?
- What distance field size is necessary to use for the rendered text to get equally good appearance as the text rendered from VISIARCs current text rendering implementation?
- Is it possible to generate a distance field fast enough for the user not to be able to determine if the distance field was pre generated or not?

1.2 Structure

This chapter described the goal and the problem formulation of this thesis. The next chapter gives basic knowledge to facilitate further reading of the report.

Chapter 3 includes information about related work that has helped the implementation of this thesis. Chapter 4 and 5 describes the methods used for implementing respectively evaluating the text rendering solution of this thesis. The methods for evaluation was used to produce the result presented in chapter 6 and discussed in chapter 7. Chapter 8 which is the last chapter of this report proposes some ideas for future work within the scope of this thesis.

2

Background

This chapter includes relevant background information about the main topics of this report.

2.1 Computer graphics

Distance fields and text renderering are subjects closely related to computer graphics. Therefore it is important to understand some basic concepts about computer graphics when reading this report. Two important concept in the context of this thesis are polygons and textures. A polygon is a figure bound together by a finite chain of straight line segments. The corners of the polygon are called vertices and are defined as coordinates in space. A usual representation of a polygon is a list of vertices where the vertices are ordered in a way that every vertice is connected to the next vertice in the list by a line segment. A texture is a representation of an image. The texture is usually represented as a one or two dimensional array with a 8 or 32 bit value per pixel.

When drawing an image or texture to the screen it has to be mapped to a polygon first. When working in two dimensions a useful polygon for mapping textures to is the quad which is the polygon with four corners. The process of drawing a texture to a quad on the screen follows. The quad is created by creating a list of vertices. A list of texture coordinates is created. Texture coordinates defines which part of the texture should be drawn on the quad. The texture is uploaded to the GPU together with the list of vertices and the list of texture coordinates. Shaders written by the programmer runs and the texture is drawn on the quad. There are two shaders of importance within the scope of this thesis, the vertex shader and the fragment shader. The vertex shader is run one time for every vertex. In the case with a texture mapped to the quad, the vertex shader would just pass through the input values. The next shader run is the fragment

shader. The fragment shader is run once for every candidate pixel on the screen. The output data from the vertex shader is interpolated before sent as input to the fragment shader. The interpolation is needed because it is very rare that every vertice maps exactly to one pixel on the screen. The interpolation is done for every variable that is sent to the fragment shader. An example often used in computer graphics is the interpolated colored triangle. The three vertices of the triangle get three different colors sent to the vertex shader. The vertex shader pass through the colors and the GPU interpolates the colors to a separate color for each pixel before sent to the fragment shader. This will create a color gradient between the three corners were each pixel has a unique color.

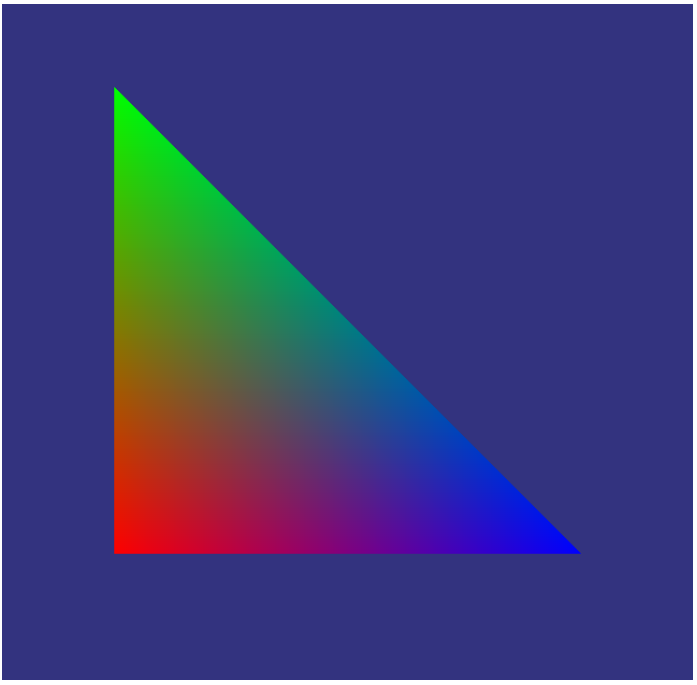


Figure 2.1: *An example of the interpolation between the vertex and fragment shader on the GPU*

2.2 Basic text rendering

Text rendering is a non trivial subject in computer graphics and has been for a long time. There are many different ways to render text to a screen. A common way is to pre render all the glyphs of a font to an image. The part of the texture corresponding to a specific glyph can then be mapped as a texture to a quad and rendered on the screen. Another way is to use some library to render the text to an image and use it to texture a quad [FreeType, 2014a]. Both of the above mentioned methods have some drawback, for example the text can not be scaled

without losing the smooth edges and the images allocates a lot of memory if you want to have high resolution on the text.

When rendering a text it is really important that every glyph is positioned relative other glyphs as specified in the font file. Every glyph in a font has visual information stored in the font file. For example the OpenType and the TrueType™ font format have information about advance, kerning, height, width etc[Microsoft Corporation, 2015, Apple Inc., 2015].

Without kerning it might be hard to get a good looking text if the font is not built in a way that kerning is not a factor. Kerning is displacement along the axis of the advancing direction of the text. A kerning value can be positive or negative and is a function of two parameters; the previous letter and the current letter. A negative kerning value is the most common and it means that the current glyph will be moved backwards relative to the advancing direction of the text and a positive value means that it will be moved forward. To get a better understanding of what kerning is, take the strings "AV" and "AA" as an example. It is obvious that the left part of "V" is hanging above the "A" but that is not the case in the second string where "A" is followed by an "A". This is because the kerning table in the font has a negative entry for the combination "AV" but not for the combimation "AA". [FreeType, 2014b]

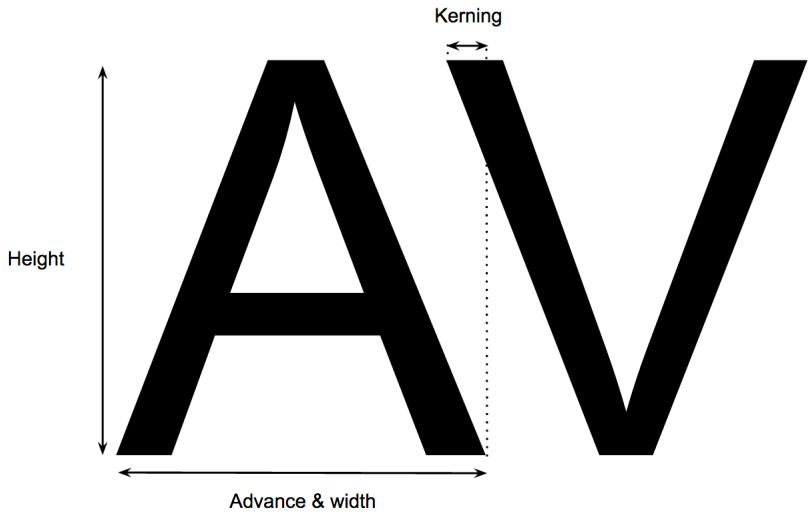


Figure 2.2: *Illustration of how kerning works*

Another important concepts when rendering text is the baseline. A text can be placed on different baselines located on different heights. The baselines used in this thesis work is top, hanging, middle, alphabetic, ideographic and bottom. The most commonly used baseline is the alphabetic baseline which is used for example when writing english text on a piece of paper.

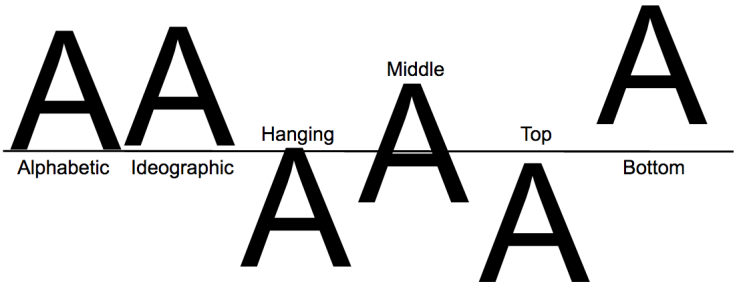


Figure 2.3: *The different baselines*

2.3 Distance fields

Binary images are frequently used in image processing [Ragnemalm, 1993]. A binary image is an image with only one binary value per pixel. This will limit the image to only contain two values or colors. One example of an image that could be represented as a binary image is a picture of a white letter on a black background. In this case white would be represented as 0 and black as 1 or vice versa.

A distance transform is the transform of an input binary image to an output image called distance field or distance map [Rosenfeld and Pfaltz, 1966]. The values of the distance field pixels represents the closest distance by some distance metric to an arbitrary shape in the input image. The distance metric can differ between different applications but the most commonly used distance metric is the euclidean distance metric which is also called real distance. If euclidean distance metric is used when doing the distance transform it is called euclidean distance transform(EDT). Other distance metrics that can be used in distance transforms are the city-block distance metric and the chessboard distance metric. When using city-block distances it is only allowed to travel horizontally or vertically in a grid with the cost of one. Chessboard distance metric is an extension of city-block distance metric where it is also allowed to travel diagonally with the cost of one.

A distance field can be signed or unsigned. An unsigned distance field only maps the distances either inside the shape or outside the shape while a signed distance field maps the pixels inside the shape with negative distances and the outside of the shape with positive distances or vice versa.

Figure 2.4: *Distance fields for the all the letters in the word lorem*

Distance fields have some advantages compared to storing the shape by drawing it in an image. One of the most obvious advantage is that a distance field represents more than just the boundary. For example a signed distance field also represents the environment around the boundary, both the interior and the exterior. This makes it easy to check if a point is inside or outside a shape by checking the value of that point in the distance field. It is also easy to move the boundary of the object by just changing the threshold value determining where the boundary is located. [Jones et al., 2006]

2.4 Beziér curves

A beziér curve is defined in space or the plane as two endpoints and a number of control points which are blended together with one blending function per point. The number of control points depends on the degree of the beziér curve. The most commonly used beziér curves are the cubic beziérs with four points and the quadratic beziérs with three points. The quadratic and the cubic beziér are defined as follows.

$$B(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2, t \in [0,1]$$

$$B(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3, t \in [0,1]$$

The quadratic beziér has P_0 and P_2 as endpoints and P_1 as a control point. The cubic beziér has P_0 and P_3 as endpoints and P_1 and P_2 as control points. The sum of the blending functions for both quadratic beziérs and cubic beziérs is always equal to 1 for any t in the function domain.[Ragnemalm, 2008]

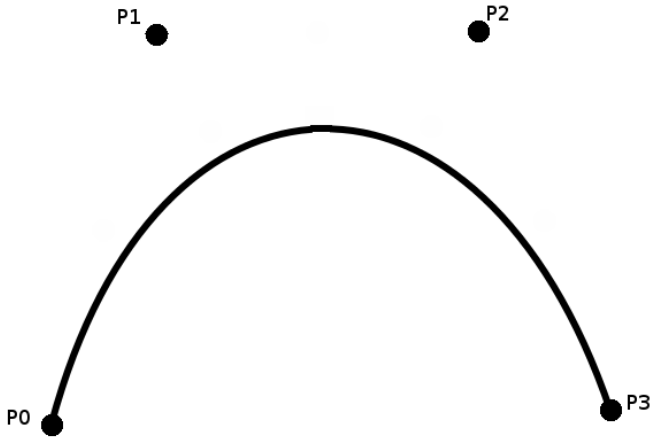


Figure 2.5: *A cubic beziér curve*

One beziér curve can not represent arbitrary curves, therefore it is important to be able to connect several beziérs to a path to be able to describe more complex shapes. When connecting curves, continuity between the curves is an important concept to consider. Parametric continuity is a measure that will have importance of the appearance of the curve. There are three levels of parametric continuity.

C^0 = The curves meet
C^1 = The derivative for the both curves are equal where the curves meet
C^2 = The second derivative for the both curves are equal where the curves meet

Another measure that can be used for continuity is geometric continuity which is almost the same as parametric continuity. Geometric continuity also has three levels of continuity G^0, G^1 and G^2 which are very similar to the corresponding parametric continuity levels. The only difference is that geometric continuity only require the derivatives of G^1 and G^2 to be proportional and not equal. Given two cubic beziér curves p defined by p_1, p_2, p_3 and p_4 and q defined by q_1, q_2, q_3 and q_4. Assume that p and q are placed in a way that $p_4 = q_1$. This will trivially fulfill the requirement for C^0 because they share one endpoint and $p(1) = q(0)$. C^1 continuity, $p'(1) = q'(0)$ will be fulfilled if p_3, p_4 and q_1 are located on the same line. C2 continuity $p''(1) = q''(0)$ will be fulfilled if p_3, p_4 and q_1 are located on the same line and $|p_4 - p_3| = |q_2 - q_1|$ which means that the distance from p_3 to p_4 must be equal to the distance from p_4 to q_2.[Ragnemalm, 2008]

An example of an application of beziérs are fonts. Every glyph in a font are stored as a number of straight lines and beziér curves, either cubic or quadratic.[Phinney, 2001] Even though beziérs are used in fonts it is not a trivial problem to draw a beziér curve to the screen. Beziér curves are hard to draw because the lowest level

graphics hardware can only draw polygons and line segments. This is solved by approximating smooth curves to line segments before drawing[Shreiner et al., 2009]. Another problem with beziér curves is that it is time consuming to find the closest point on a beziér curve from an arbitrary point. To solve this problem a solution for the equation $(p - q(t))q'(t) = 0$, where $t \in [0, 1]$, p is a point in space and q is a beziér curve has to be found[Chen et al., 2007]. This implies that a quintic polynomial needs to be solved to find the closest point on a cubic beziér curve given a point in space.

De Casteljau's algorithm can be used to subdivide a beziér curve recursively by using the properties of beziér curves to calculate new beziér points for both the left and the right part of the old beziér.[Fischer, 2000] A special case of the algorithm is to divide the curve at $t = 0.5$ which gives a first curve $t \in [0, 0.5]$ and a second curve $t \in [0.5, 1]$. An example of this subdivision for a cubic beziér is presented in figure 2.6 below. The first curve is described by the points p_0, p_{01}, p_{012} and p_{0123} and the second curve is described by p_{0123}, p_{123}, p_{23} and p_3. The points for the new curves are derived from the initial curve described by p_0, p_1, p_2 and p_3. The following calculations shows how the new beziér points are computed.

$$P_{01} = \frac{P_0+P_1}{2}$$
$$P_{23} = \frac{P_2+P_3}{2}$$
$$P_{12} = \frac{P_1+P_2}{2}$$
$$P_{012} = \frac{P_{01}+P_{12}}{2}$$
$$P_{123} = \frac{P_{12}+P_{23}}{2}$$
$$P_{0123} = \frac{P_{012}+P_{123}}{2}$$

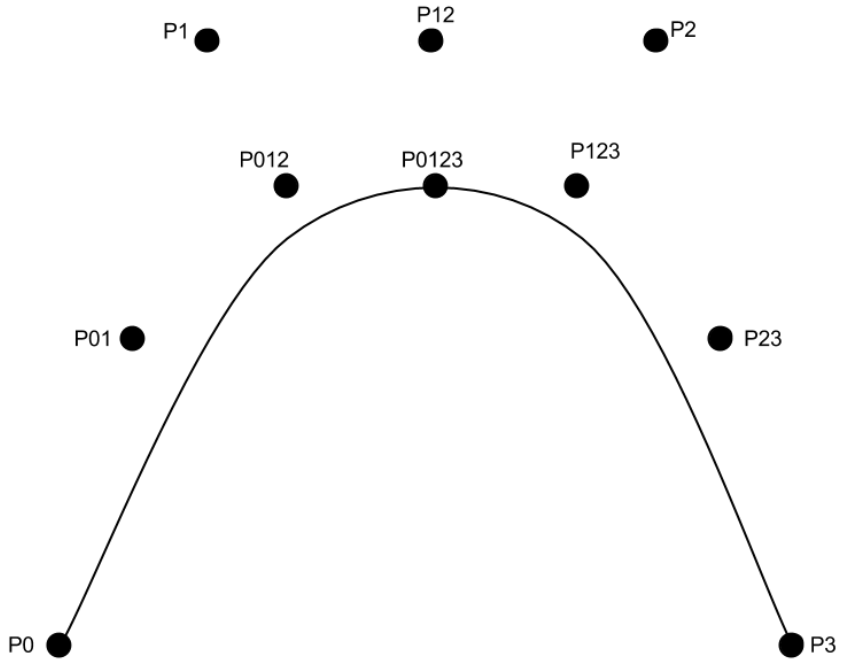


Figure 2.6: *An illustration of the casteljau's algorithm splitting a cubic beziér at* $t = 0.5$

2.5 Polygon filling

Computer graphics is partly about drawing polygons to the screen but this does not make it a trivial problem. There are several algorithms for drawing a polygon to the screen where each one of them have their advantages and drawbacks. One algorithm that is used a lot for this purpose is the scan-line polygon fill algorithm. The algorithm contains two main components. The first component is a sorted edge table which is an array of linked lists where each linked list corresponds to a row of pixels in the image. The second component is an active edge list which is an initially empty linked list of edge references. The x value for the lowest y value of each line segment is inserted into the edge table. The next step in the algorithm is to iterate through the edge table row by row. All edges in the active edge list that ended on the previous line are removed. All remaining values of the active edge list are updated to the intersection between the line segment and the current row. All values of the current row in the edge table are then inserted

into the active edge list and the active edge list is then sorted by x value. The image can now be filled for the current row by using some fill rule. The odd even fill rule is easy to use by just iterating through the active edge list and filling the pixels between every pair of edges starting with an odd edge.

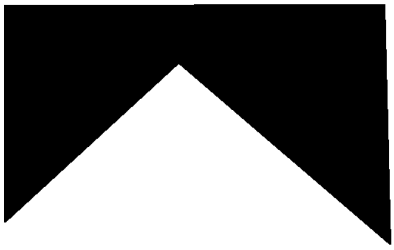


Figure 2.7: *A case managed by odd even fill*

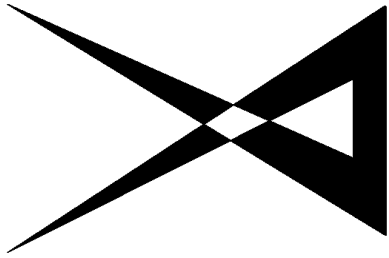


Figure 2.8: *A case not managed by odd even fill*

The odd even fill rule only work if edges can not cross eachother. If edges cross eachother there might be filled where it should not be filled and vice versa. An example of this can be seen in figure 2.8 where the odd even fill rule fails to fill the area in the middle. To handle this another fill rule has to be used, for example the non-zero winding number rule. This rule handle case of edges crossing eachother because the rule also takes into account in which direction the edges of the polygon are moving and increments a variable with different sign depending on that direction when crossing an edge.

3

Related work

3.1 Early EDT algorithms

In a very often mentioned article, Danielsson [1980] proposed an improved way to generate distance maps by representing the output of the distance transform as a vector, separating the distance of the different dimensions. Danielsson also proposed two sequential algorithms 4SED (4-point Sequential Euclidean Distance mapping) and 8SED (8-point Sequential Euclidean Distance mapping) for calculating the EDT using his representation of distance. Both the 4SED algorithm and the 8SED algorithm consists of two consecutive picture scans where they incrementally update pixel values depending on nearby pixels. The 8SED algorithm is described in the following pseudocode.

Algorithm 1 First scan of the 8SED algorithm

```
for j = 1 to N − 1 step 1 do
    for i = 0 to M − 1 step 1 do
        L(i, j) = min(L(i, j), L(i-1, j-1)+(1, 1), L(i, j-1)+(0, 1), L(i+1, j-1)+(1,1))
    end for
    for i = 1 to M − 1 step 1 do
        L(i, j) = min(L(i, j), L(i-1, j)+(1, 0))
    end for
    for i = M − 2 to 0 step 1 do
        L(i, j) = min(L(i, j), L(i+1, j)+(1, 0))
    end for
end for
```

Algorithm 2 Second scan of the 8SED algorithm

```
for j = N − 2 to 0 step 1 do
    for i = 0 to M − 1 step 1 do
        L(i, j) = min(L(i, j), L(i-1, j+1)+(1, 1), L(i, j+1)+(0, 1), L(i+1, j+1)+(1,
1))
    end for
    for i = 1 to M − 1 step 1 do
        L(i, j) = min(L(i, j), L(i-1, j)+(1, 0))
    end for
    for i = M − 2 to 0 step 1 do
        L(i, j) = min(L(i, j), L(i+1, j)+(1, 0))
    end for
end for
```

The two scans are very similar. The only difference is that the first scan is done top down evaluating the pixels below and on the sides of the current pixel and the second scan is done bottom up evaluating the pixels above and on the sides of the current pixel[Ragnemalm, 1993]. The 4SED algorithm and the 8SED algorithm are not error free as Danielsson [1980] proves in his article but he also claims that the errors are rare and small and are negligible for practical purposes. In 8SED and 4SED every pixel are visited a constant number of times. This makes the time complexity of 8SED and 4SED trivially $\mathcal{O}(mn)$, if m and n are the width respectively the height of the input image.

3.2 Exact EDT algorithms

Distance transforms has been used in different applications in many years. The first article on the subject was Rosenfeld and Pfaltz [1966] introducing an algorithm for distance transform using simple metrics. The article from Danielsson [1980] was not the first on the subject but the fact that the algorithms he proposed in his article are some of the most widely used[Fabbri et al., 2008] makes it easy to call Danielsson one of the pioneers on the subject. As mentioned earlier in this report, 8SED and 4SED are not exact. Exact algorithms for EDT has only been around since the 1980s. An example of an exact EDT from the 1980s is Ragnemalm [1989]. In an article Fabbri et al. [2008] compares the execution time of exact EDT algorithms. The six algorithms compared in the test are Meijster et al. [2000], Maurer Jr et al. [2001], Eggers [1998], Lotufo and Zampirolli [2001], Cuisenaire and Macq [1999] and Saito and Toriwaki [1994]. The conclusion of the comparison is that the algorithms from Meijster and Maurer are the fastest but Meijster's algorithm is preferred due to slightly better performance and the fact that it is easier to implement than Maurer's algorithm.

Meijster's algorithm consist of two separate phases. The first phase iterates through each column performing a distance propagation in both directions. This will create a new image $g(i, j)$ which is used in the second phase. The second

phase iterates through each row left to right and right to left applying a function $DT(x, y)$ to calculate the output value of each pixel. The function depends on what distance metric is used. The function used for euclidean distance transform follows.

$$DT(x, y) = \min_{0...m}((x - i)^2 + g(i)^2)$$

With the same motivation for 4SED and 8SED the time complexity of Meijster's algorithm is $\mathcal{O}(nm)$, if m and n are the width respectively the height of the image.

3.3 Improved distance measure for EDT algorithms

Calculating a distance transform of large size images can be very time consuming. There is a significant difference in computation time between creating a distance map from a 4096x4096 image and creating a distance map from a 64x64 image. For example, the 8SED and the 4SED algorithms both have time complexity O(n) where n is the number of pixels. Time complexity O(n) means that every pixel is visited a constant number of times when transforming the image. Assuming every pixel is visited once and one calculation is done per visit the larger image would require 4096 times more calculations than the smaller image. In an article from Green [2007], distance fields are generated by using a 4096x4096 binary image of a glyph as input to a distance transform. The output from the distance transform is then downsampled to a 64x64 texture. Generating a distance field using a large input image makes the discrete set of possible boundary pixels more dense compared to if a smaller input image was used. The increased density of the possible pixel set helps decrease the calculation error of the distance field assuming subpixel distance measures is not used.

In an article, Gustavson and Strand [2011] shows that the calculation errors of distance transforms can be decreased by using information about the subpixel boundary between the foreground and the background pixels. In the article they use an anti-aliased greyscale input image to be able to locate the boundary in the pixels to get more precise distance measures between the pixel and the boundary.

They show that they get approximately the same amount of errors using their method on a 16x16 times smaller input image than by using the method Chris Green proposed in his article. The smaller input image increases the the execution time and decreases the memory consumption by a factor of approximately 30 times according to Gustavson and Strand.

4

Methods for implementation

This chapter gives a detailed description of the methods used for solving the problem formulations of this thesis. The chapter is divided into two different parts. The first part gives a detailed description about the implementation of the distance transform module. This part has been reimplemented several times due to poor performance of the used methods. The second part gives a detailed description about the implementation of the distance field rendering module.

4.1 Distance field generation initial attempt

There are many EDT algorithms for generating distance fields. Most of the EDT algorithms used today runs in $\mathcal{O}(nm)$, where n and m are the width respectively the height of the image. Example of algorithms running in $\mathcal{O}(nm)$ are Danielsson [1980] and Meijster et al. [2000].

The initial implementation of distance field generation in this thesis was built around Gustavson and Strand [2011] article on the subject. A signed version of the 8SED algorithm was used along with the anti-aliased sub-pixel distance measure proposed in the article. A signed distance field has several advantages compared to an unsigned distance field. The most obvious advantages are the increased flexibility it gives and that it facilitates the implementation of proper anti-aliasing around the border of the shape[Gustavson, 2012]. To generate a signed distance field the distance transform was run twice with inverted input representation. The first transforms creates a distance field on the inside of the glyph and the second transform on the outside of the glyph. The resulting distance field was then calculated with the following formula.

$$result(i,j) = inside(i,j) - outside(i,j), \forall (i,j), i \in \{0,\ldots,m-1\}, j \in \{0,\ldots,n-1\}$$

4.2 Fast and approximate distance field generation

Fast distance field generation on mobile devices requires some approximations and optimizations to decrease the transformation time. This can be seen in many of the distance field generation implementations done lately. Two examples of this is GLyphy by Behdad Esfahbod and the implementation of distance field generation in a cross-platform application framework called Qt. Both implementations approximate the beziér curves from the font file to some different representation of a glyph that is easier to work with. GLyphy approximates the beziér curves to circular arcs and Qt approximates the beziér curves to line segments. The easier representation of the outline makes it possible to draw the distance field locally over the outlines of the glyph and a distance transform of the whole image is not needed.

Because the initial implementation of the distance field generation was too slow to meet the requirement another, faster implementation was necessary. A variant of the implementation done by Qt was implemented. The implementation can be divided into four steps in the following order; extracting beziérs from the given font file; approximating the beziérs to line segments; drawing the glyph in an image and drawing distance gradients over the line segments in the image to create a distance field locally over the outlines of the glyph. Extraction of beziér curves from font files can easily be done with different libraries for example freetype and a description of how this is done is not relevant for the proceeding of this report.

4.2.1 Beziér to line segment approximations

Beziér to line segment approximation is done a lot in computer graphics. A naive way to solve the beziér to line segment problem is to split the Beziér curve into n smaller curves with a constant length and approximate them to line segments. This would make the line density constant over the whole curve meaning that both the strongly bent parts and the almost straight parts will be represented by an equal amount of line segments. A smarter way to solve this is proposed in an article by Fischer [2000]. The solution includes using a recursive function dividing the beziér into two parts using the de Casteljau's algorithm until a stop condition is met. This will make the line segment density higher where the curve is strongly bent and lower where the curve is almost straight. The following pseudocode is taken from the article by Fischer.

Algorithm 3 Function for approximating beziér to line segment

```
procedure flattenCurve(Curve c)
    if isSufficientlyFlat(c) then
        output(c);
    else
        Curve l,r
        subdivide(c,l,r)
        flattenCurve(l)
        flattenCurve(r)
    end if
end procedure
```

In the above code Curve is a cubic beziér curve represented by four points in the plane. The function isSufficientlyFlat checks if the difference between the curve and the line segment from the start point to the end point of the curve is small enough to meet the stop condition. This function can be implemented in many different ways. In his article Fischer propose a stop condition initially developed by Roger Willcocks. The pseudocode for this stop condition follows.

Algorithm 4 Stop condition for cubic beziér subdivision

```
function isSufficientlyFlat(Curve c)
    double ux = 3.0*c.b1.x - 2.0*c.b0.x - c.b3.x; ux *= ux
    double uy = 3.0*c.b1.y - 2.0*c.b0.y - c.b3.y; uy *= uy
    double vx = 3.0*c.b2.x - 2.0*c.b3.x - c.b0.x; vx *= vx
    double vy = 3.0*c.b2.y - 2.0*c.b3.y - c.b0.y; vy *= vy
    if ux<vx then
        ux = vx
    end if
    if uy<vy then
        uy = vy
    end if
    return (ux+uy ≤ tolerance)
end function
```

There are other ways to check if a curve is flat enough to approximate it to a line segment. One solution is proposed by Shemanarev [2005]. The stop condition in his article only depends on the sum of the distances d1 and d2 illustrated in figure 7.1. If $d1 + d2 > distance\_tolerance$ the beziér curve is approximated to the line segment between P_0 and P_3. The constant distance_tolerance is set depending on the required quality of the approximation. A low distance_tolerance value will result in a smooth curve with only small artifacts between the different line segments while a high distance_tolerance value will result in a jagged curve with clearly visible artifacts between the different line segments.

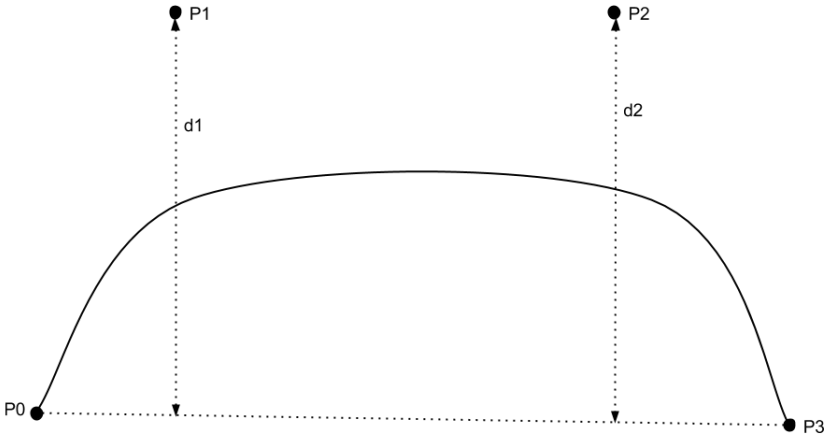


Figure 4.1: *Illustration of the stop condition used when approximating a bezier curve to line segments.*

The method used for approximating beziér curves to line segments in this thesis is de Casteljau's algorithm splitting the beziér curves at $t = 0.5$ together with the stop condition proposed by Fischer.

4.2.2 Drawing the glyph

The distance gradients in the distance field will only be drawn locally within a predefined distance from each outline. This creates a need to draw the glyph in the image before generating the distance field. If the glyph is not drawn before the distance gradients are drawn, there will most certainly be parts of the glyph that has the outside color but is located on the inside of the glyph or vice versa. For example the middle of the 'I' might have the same color as pixels outside the glyph. This would lead to a hole in the middle of the 'I' when rendering it which is not desired. The input to this step of the distance field generation is a glyph represented as connected line segments in the shape of a polygon. As mentioned in the background the scan-line algorithm together with the odd even fill rule or the non-zero winding rule can be used to fill a polygon. Because this is a very common problem in computer graphics and there are many open source libraries doing this very fast and optimized there were no need to implement this. The function drawPath in the open source 2D graphics library Skia was used instead. The Skia library was already installed and used in other places within VISIARC's

products.

4.2.3 Drawing distance gradients

When drawing gradients it is convenient to draw them in primitive shapes like rectangles or triangles. To be able to draw gradients contiuously over the outline of the whole glyph two different distance gradients has to be drawn for each line segment. The first gradient is a rectangular gradient and the second gradient is a triangular gradient. The gradients are illustrated in figure 4.2.

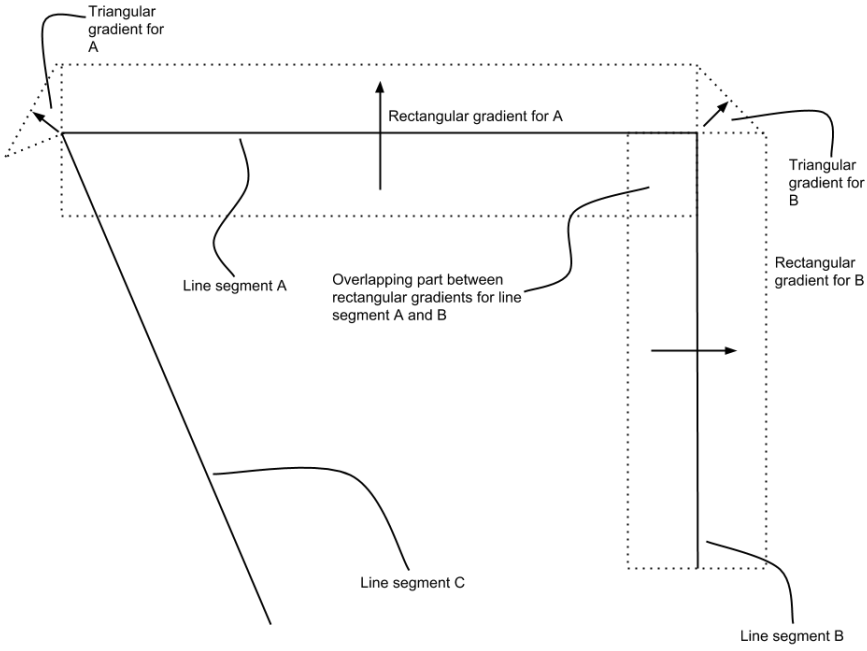


Figure 4.2: *Line segment to pixel distance*

The gradients are drawn in the direction of the arrows in the range from 0 to 255. This will make the inside of the glyph black/grey and the outside of the glyph white/grey. Because the glyph is drawn in the image using the Skia library before drawing the gradients there are no need for the gradients to cover the whole glyph. The triangular gradients are used to repair the distance field when two line segments are not parallel which is almost always the case. As illustrated in the figure it might occur problems in some areas when drawing the gradients. One of the problem areas is the overlap between different gradients. Another problem that might occur is that parts of gradients that should be on the inside of the glyph ends up on the outside or vice versa. This will happen if the angle between two line segments is less than 90 degrees. To solve both the mentioned problems a check is done before drawing in a pixel. A description of

the check done follows. If the new and the old value are not both inside values or outside values, the pixel should not be drawn. If both the new and the old value are inside values, draw if and only if the old value is lower than the new value. If both the new and the old value are outside values, draw if and only if the old value is higher than the new value.

Drawing gradients can be done in many different ways. The initial implementation of this step, in this thesis, had real measured distances as values in each pixel in the gradient. The pixel to line segment distance was calculated by projecting the pixel coordinate onto the line. If the projection hit the line segment the distance was equal to the projection distance. Otherwise the distance was equal to the shortest distance to one of the endpoints of the line segment.

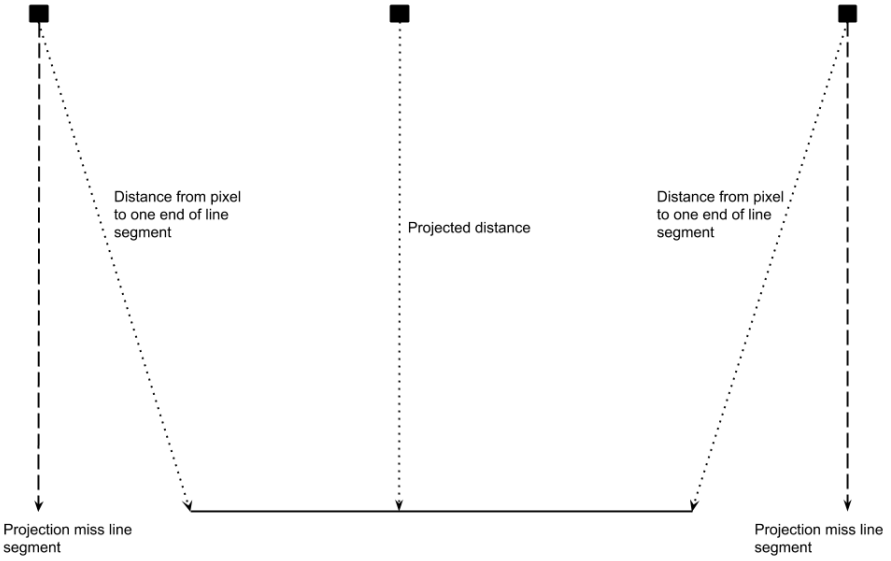


Figure 4.3: *Line segment to pixel distance*

The code for projecting a pixel to a line segment made the distance fields look good because there were few sources of errors drawing the gradients. Every pixel was calculated independently from eachother and if the new distance was not better than the distance already in the pixel the pixel was not updated. The only problem with this approach was the execution time. The code for pixel to line segment distance calculation was located in the innermost loop which means that the code was executed very many times and performance improvements were neccessary.

In the distance field generation made by Qt no pixel to line projection is done. The rectangular gradients are drawn directly using trigonometric calculations

stepping through the pixels row by row. This influenced how the rectangular gradients are drawn in the final result of this thesis. The problem by using this solution is to find the distance d in figure 4.4.

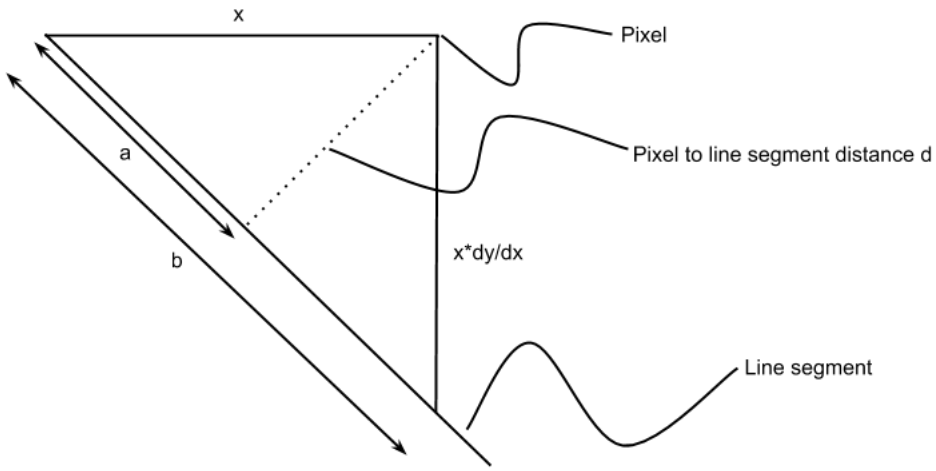


Figure 4.4: *Illustration of the calculations done when drawing the rectangular gradients*

The line segment is defined between two points in the plane p_1 and p_2. The derivative of the line can trivially be calculated $\frac{dy}{dx} = \frac{p_{2y}-p_{1y}}{p_{2x}-p_{1x}}$. The projection of the pixel to the line segment will always be perpendicular to the line segment which implies that the triangle with sides x, $x\frac{dy}{dx}$ and b and the triangle with sides x, a and d are similar triangles. Similarity between two triangles means that the sides of the first triangle are proportional with the same constant to the sides of the second triangle. In mathematic terms it implies that $\frac{x}{b} = \frac{a}{x\frac{dy}{dx}} = \frac{d}{x}$ is true. The similarity of the triangles gives two ways to calculate d, either $d = \frac{x^2}{b}$ or $d = \frac{a}{\frac{dy}{dx}}$. The first solution is used in this thesis and b is calculated with the following formula $b = \sqrt{x^2 + (x * \frac{dy}{dx})^2}$. Having b the final formula for calculating d is $d = \frac{x^2}{\sqrt{x^2+(x*\frac{dy}{dx})^2}} = \frac{x}{\sqrt{1+\frac{dy}{dx}^2}}$. The shortest distance from the line segment to the pixel is equal to $|d|$.

The rectangular gradients are drawn by stepping through all the pixels covered by the rectangle row by row and calculating the distance d. The control mentioned previous in this section is used before drawing to remove the overlapping problems. Drawing the triangular gradients are done in the same way with one exception, that is the calculation of the distance d. The closest point on the

line segment to the pixel in the triangular gradient will always be the corresponding endpoint p of the line segment closest to the triangular gradient. This makes it possible to calculate d as the distance from the pixel to the point p.

4.3 Distance field rendering

Distance field rendering is not as easy as usual image rendering. When rendering images to the screen with OpenGL the image is loaded into a texture and uploaded to the GPU. On the GPU the image is resized and interpolated to fit the pixels on the screen before it is drawn. The interpolation is a built in operation on the GPU. An image can be drawn to the screen with a basic vertex and fragment shader where the vertex shader pass through the coordinates of the polygon that the texture is mapped to and the fragment shader fetch the color corresponding to its pixel from the texture and output it to the screen. When rendering distance field glyphs a more complex fragment shader has to be used.

The basic idea with distance field rendering is that the interpolation of a greyscale image does not decrease quality of the glyph border as much as if the image was not a greyscale image. This will let the GPU interpolate the image to the screen preserving the information about where the outlines of the glyph are located before the fragment shader takes the decision whether the corresponding pixel is inside or outside the glyph. The most basic fragment shader for rendering distance field will set the output pixel transparent if the corresponding lookup in the distance field has a value greater or equal to 0.5. On the GPU the color values are floats between 0 and 1 which means that outlines of the glyph are located at 0.5. This solution is really easy to implement and can be written as one line of code but it will create aliasing artifacts around the border of the glyph. To render good looking text using distance fields some kind of anti-aliasing is needed.

A commonly used function when rendering distance field glyphs is the smoothstep function. The smoothstep function is a built in function in GLSL and it performs a smooth hermite interpolation between 0 and 1 in a range for an input value. The range and the value are the parameters to the function. Using this function over an interval around the outline of the glyph will create a smoother edge and decrease the aliasing. This is a pretty cheap way to create descent anti-aliasing which can be modified when changing the interval. A bigger interval for example $[0.4, 0.5]$ will create a blured edge while a smaller interval for example $[0.49, 0.51]$ will create a sharper edge with the cost of possible aliasing effects. The use of the smoothstep function is not enough aliasing reduction in many cases. There are methods to improve the anti-aliasing even more. One way used by Gustavson [2012] is to use the GLSL built in functions dFdx and dFdy which returns the derivatives of the distance field for the current pixel in the x and y axis. The function used in the fragment shader used in the article follows.

Algorithm 5 Anti-aliasing function proposed by Gustavson

```
procedure aastep(float threshold, float distance)
    float afwidth = 0.7 * length(vec2(dFdx(distance), dFdy(distance)));
    return smoothstep(threshold - afwidth, threshold + afwidth, distance);
end procedure
```

All three of the above described fragment shaders were implemented in this thesis. Implementing all three makes it possible to choose shader depending on the computational power of the GPU. The first mentioned implementation with a simple threshold check is the fastest one and the last mentioned shader containing the aastep function is the slowest.

5

Methods for evaluation of the implementation

The goal with this thesis was to solve the problem formulation described in the introduction. To be able to evaluate if the goals were achieved four different methods of evaluation were needed. The methods of evaluation are described in this chapter.

5.1 Visual evaluation 1

Evaluating the visual appearance of rendered text is hard. The evaluation should be done in a way that it is justified to say that the text rendered with the implementation from this thesis is at least as good looking as the text rendered by the old implementation. One way to evaluate this is letting one or several persons judge if the text rendered with the new implementation is good enough. This method could be used in this thesis and the judge in this case would have been VISIARC which is the company this thesis is written for. The results from using this evaluation method would not be facts, it would be one or several personal oppinions. Because people have different oppinions this test could not be replicated by other people which is desirable. This is the reason this method was not used in this thesis.

To get a test that could be replicated a more technical test was needed. The method used for the first visual evaluation of glyphs rendered with the implementation from this thesis follows. Two equally big glyphs were rendered from both the old text implementation and from the new implementation from this thesis. This made it possible to compare pixel values between the two images and output a result. The result from the test was a greyscale image with the same size as the input images where white symbolizes difference and black no difference between the input images. The value for pixel n in the output image was calculated by taking the positive difference between pixel n in the first and the

second image. In the output image it is easy to see where the errors are located and how big the errors are.

5.2 Visual evaluation 2

The second visual evaluation uses the same method as the first but output a different result. The output from this evaluation is a percentage of how many of the pixels are equal between the images.

5.3 Performance evaluation

There are many ways to measure performance. Because a goal with this thesis was to be able to render text in real time a good measure for performance is the distance field generation speed. The distance field generation speed was measured as the time between the function call and the function return of the function generating distance field. The impact of different scheduling in the mobile device CPU was neglected.

5.4 Distance tolerance evaluation

The constant distance_tolerance used when approximating a bezier curve to line segments can be set to any number. To be able to determine which distance_tolerance value gave the best result a test was necessary. To measure which value of the contant gave the best result the distance field generation time was measured for different values of the distance_tolerance. The impact of different scheduling in the mobile device CPU was neglected performing this test.

6

Result

This chapter present the results of this thesis. To get a result four tests were run comparing the new implementation of text rendering with distance field with the old implementation. The four tests are described in the previous chapter. The font used for all the tests was Source Sans Pro which is an open source font originally developed by Adobe. All the distance fields generated for the tests visual evaluation 1, visual evaluation 2 and performance evaluation were generated with the constant distance_tolerance described in the method chapter set to 0.1.

6.1 Visual evaluation 1

The method described in 5.1 involves comparing pixel values of rendered glyphs from the old text rendering implementation with the text rendering solution implemented in this thesis. The test was performed on ten different glyphs in two different distance field sizes.

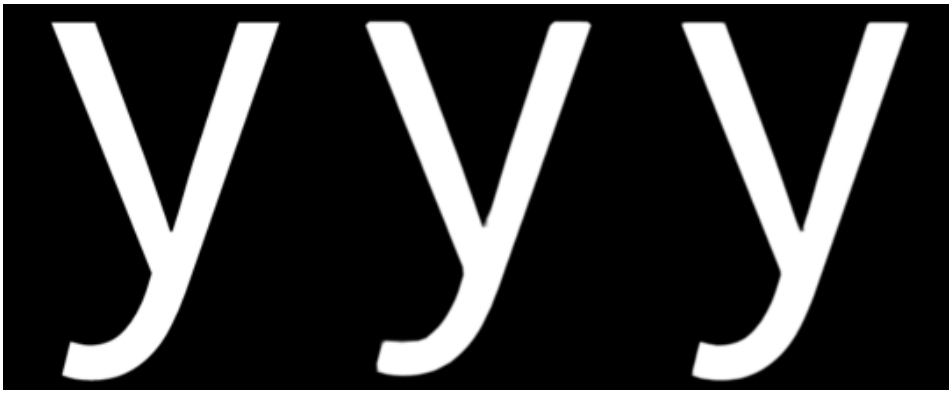


Figure 6.1: *One set of input images to the visual evaluation of the rendered glyphs. From the left the pictures are rendered with the old implementation, the new implementation with size 64x64 and the new implementation with size 128x128.*

The difference between the images with different distance field is significant. The quality of the glyph rendered from the 128x128 distance field is higher which can easily be recognized comparing the corners of the two distance field glyphs. The following images are the result of the test on one of the glyphs.

Figure 6.2: *The result of the test Visual evaluation 1. The picture to the left is a comparison between renderering with the old implementation and the new implementation with distance field size 64x64 and the picture to the right is a comparison between rendering with the old implementation and the new implementation with distance field size 128x128.*

There is a significant difference of the amount of error between glyphs rendered with 64x64 and 128x128 distance fields. It is well known that glyphs ren-

dered with distance fields get rounded corners which this picture shows aswell, especially in the upper right corner of the right error image. The anti aliasing implementation of the old text rendering and the new is not the same. This might give possible errors around the edges of the glyph in the output image. The rest of the result images can be found in appendix A.

6.2 Visual evaluation 2

As described in 5.2 the test Visual evaluation 2 was done in the same way as Visual evaluation 1 but with a different output. The result is a comparison between the equality of pixels between the both input images. The following table gives the result of the tests.

Table 6.1: *Results of visual evaluation 2*

Glyph	64x64	128x128
Å	0.65%	1.69%
B	3.24%	1.35%
G	4.67%	0.32%
J	3.46%	0.27%
y	2.75%	0.44%
ö	2.66%	0.87%
x	2.47%	2.25%
a	3.19%	0.75%
3	2.87%	1.96%
7	2.09%	0.49%

The table shows that almost every glyph gets better at the higher distance field size. The average pixel error is 2.81% for the 64x64 distance field size test and 1.04% for the 128x128 distance field size test. The only glyph that had less error at the smaller distance field size was the swedish letter Å. The reason for this special case might be different sub pixel alignment of the glyphs rendered by the the two implementations.

6.3 Performance evaluation

As mentioned in 5.3 the method for testing the performance of the distance field text rendering implementation was measuring the time for generating the distance field. The test was run with three different mobile device iphone4, samsung galaxy tab 1 and samsung S6. The glyphs included in the test were a-ö, A-Ö, 0-9 and some other commonly used glyphs. The test was run one time each on the samsung galaxy tab 1 and the samsung s6. The test was run five times on the iphone4 because the time measure on the iphone was not as stable and reliable

as the time measure on the samsung devices. In table 6.2 the average generation time for all the tests on the different devices are presented.

Table 6.2: *Results of performance evaluation*

Device	64x64	128x128
Samsung galaxy tab 1	0.0069 s	0.0159 s
Samsung S6	0.0020 s	0.0050 s
Iphone4 #1	0.0024 s	0.0049 s
Iphone4 #2	0.0027 s	0.0045 s
Iphone4 #3	0.0026 s	0.0047 s
Iphone4 #4	0.0022 s	0.0047 s
Iphone4 #5	0.0025 s	0.0057 s

The Samsung S6 was the device generating the distance fields fastest with an average of 0.002 seconds with 64x64 distance field and 0.0069 seconds with 128x128 distance fields. All the results from the test including generation times for each glyph on every device are presented in appendix B.

6.4 Distance tolerance evaluation

The test distance tolerance evaluation was run on a Samsung S6 rendering the glyph O. The distance field size for this test was contant and set to 128x128. Figure 6.3 present some of the output images of the test.

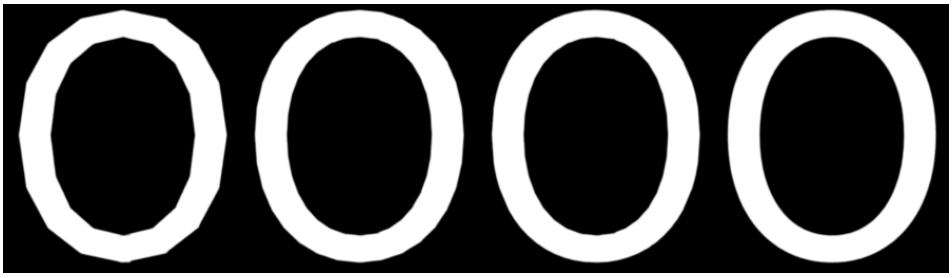


Figure 6.3: *Some of the output images from the test distance tolerance evaluation. From the left the value of the distance_tolerance are* 128.0*,* 1.0*,* 0.5 *and* 0.1*.*

An important result of this test is how the distance field generation time is connected with the value of the distance_tolerance. Table 6.3 present the distance field generation times for the test.

Table 6.3: *Results of the distance tolerance evaluation*

Value	Time
128.0	0.009545 s
1.0	0.009559 s
0.9	0.009113 s
0.8	0.009112 s
0.7	0.009132 s
0.6	0.009296 s
0.5	0.010477 s
0.4	0.011286 s
0.3	0.010185 s
0.2	0.011947 s
0.1	0.016554 s
0.05	0.017990 s
0.01	0.033086 s

As the table show the distance field generation time is stable around 10 ms for most of the value of distance_tolerance. For low values of distance tolerance the distance field generation time increases fast.

7

Discussion and conclusions

This chapter evaluates the result and the methods used to reach the results of this thesis.

7.1 Result

Four different tests were run to evaluate the performance of the text rendering solution implemented in this thesis. The four tests were developed with the goal to evaluate if the implementation solved all the problems set up in the problem formulation. The problems formulation described in chapter one follows.

- How can distance fields be used when rendering text and how does the technique work?
- What distance field size is it necessary to use for the rendered text to get equally good appearance as the text rendered from VISIARCs current text rendering implementation?
- Is it possible to generate a distance field fast enough for the user not to be able to determine if the distance field was pre generated or not?

The first problem of the problem formulation is answered by chapter two of this report. Therefore this problem was not covered by the tests in the result chapter. The two visual evaluation tests were developed to give an answer to the second problem of the problem formulation. The visual evaluation tests only evaluated the distance field sizes 64x64 and 128x128. The reason for only evaluating two distance field sizes was that other distance field size gave too bad results. Smaller distance field sizes resulted in too rounded corners of the glyphs and bigger distance field sizes did not improve the quality enough to be prefered over the distance field size 128x128.

There are many sources of error in the visual evaluations. One of the possible errors is that the in this thesis developed text rendering solution uses a different type of anti aliasing than the old implementation. This might be the reason for some of the errors in the visual evaluations. Another possible error in the visual evaluations is that the distance fields rendered by the distance field generating module implemented in this thesis generates smaller distance fields than the textures generated by the old text rendering implementation. A drawback of this is that the renderings of the old text rendering solution and the text rendering solution in this thesis might have different sub-pixel positioning of glyphs due to rounding. This can be seen in the results of visual evaluation #1 presented in appendix A where some glyphs have more errors on for example vertical lines or horizontal lines depending on vertical and horizontal sub-pixel positioning.

The test performance evaluation was developed to give an answer to the third problem formulation. The test consisted of measuring the execution time of the function for generating distance fields implemented in this thesis on three different mobile devices. The three devices used in the test were very different to eachother. The computational power of the CPUs and the GPUs differ between all the devices aswell as the release year. The performance of the devices in the test was reflected on the age of the devices which is natural considering they get better and better for every year.

The result of the test distance tolerance evaluation showed that the distance field generation time increased for small values on the constant distance_tolerance. The test was only run on one mobile device which was a Samsung S6. There is a small possibility that the result of this test differ between mobile devices because of different hardware. This is not covered by the test. The test shows that a value of the constant distance_tolerance higher than 0.2 for distance field size 128x128 gives about the same distance field generation time. The fact that 0.2 gives about the same distance field generation time as higher values gives no reason to use a higher value than 0.2.

Figure 7.1: *The word "Lorem" rendered by the implementation done in this thesis with distance field size 128x128 and* $distance\_tolerance = 0.01$.

The value of the constant distance_tolerance used for beziér to line segment approximations was 0.1 in all the runs of the performance evaluation. This value is pretty low and gives almost no visual artifacts between the line segments of the outline of the glyphs. A higher value of this constant would decrease the

distance field generation time for all the glyphs containing beziér curves. The result of this test can therefore be seen as the maximum generation time for the devices in many cases.

As the visual evaluation shows bigger distance fields result in less errors when rendering the glyph. The test visual evaluation 2 resulted in an average of 2.81 % and 1.04 % using 64x64 respectively 128x128 distance fields. Comparing the average numbers of the test gives that rendering with a 64x64 distance fields result in almost 3 times more errors than rendering with an 128x128 distance fields. This is an expected result considering an 128x128 distance fields use 4 times the memory of a 64x64 distance field. Because memory usage is important on mobile device this is a strong argument for the 64x64 distance fields. When choosing distance field size the factors that matters are visual appearance, memory consumption and distance field generation speed. Because mobile devices usually have small screens and text rendered on a mobile device usually do not cover the whole screen with one glyph the distance field size 64x64 is recommended for usage.

The device performing the worst of the three devices in the performance test was the Samsung galaxy tab 1. The average distance field generation time of this device in the performance test was 0.0069 s for the 64x64 distance field and 0.0159 s for the 128x128 distance fields. One of the requirements set up by VISIARC on the performance of this thesis was that the implementation should be able to generate a distance field between two frames. Assuming the mobile devices renders at 60 FPS the maximum distance field generation time is $\frac{1}{60} = 0.01666\ldots$. Because the average generation time for all devices in the performance test are lower than $\frac{1}{60}$, this requirement is fulfilled and the user should not be able to notice any difference between typing in a character that has a distance field pre generated and a character that does not. There is always possible to find arguments against this. One way to find arguments against the performance of the distance field generation module is to find bad cases. One bad case for the distance field rendering module would be if the app wants to render the whole alphabet in a font with no pre rendered distance fields. This could happen for example when scrolling through a pdf file that contains a line with a different font than the rest of the file listing the whole alphabet. Using 128x128 distance field size and a Samsung galaxy tab 1 this would freeze the app for a couple of seconds if the distance field generation is not done in a separate thread.

As mentioned in this section the recommended values for the constant distance_tolerance and the distance field size are 0.2 and 64x64. Figure 7.2 shows how a rendering with the recommended values can look like.

Figure 7.2: *The word "Lorem" rendered by the implementation done in this thesis with the in this thesis recommended distance field size and distance_tolerance.*

7.2 Method

When generating distance fields in real time there is no time to iterate through all pixels. Example of such algorithms are described by Meijster et al. [2000] and Danielsson [1980]. Optimizations and approximations has to be done to be able to skip some of the pixels. The method used for generating the distance fields in this thesis is not very well known. The only two similar implementations used as reference for the method are as mentioned in the method chapter the implementation in GLyphy and Qt. The fact that Qt has a similar method to the one used in this thesis implemented in their products proves that the method works and is used in the industry.

The fact that the method used for generating distance fields in this thesis uses two steps of approximations makes it not exact. The first approximation done is the beziér to line segment approximation. This approximation depends on the constant distance_tolerance. The approximations gets better the lower the value of the constant distance_tolerance is but it will never become exact. Because decreasing the value of the constant distance_tolerance increases the distance field generation time it is not desirable to use an infinite low distance_tolerance. The second approximation done is the gradient drawing. The distances in the gradients are not real distances. The distances are scaled so that the range $[0, 255]$ covers the whole gradient in the increasing direction. The fact that the distance field might not contain the real distances to the shape makes it lose usability in applications where real distances are important. Luckily this is not the case when rendering distance field text because the boundary of the shape will still be interpolated to the correct position.

7.3 Memory

The text rendering implementation of this thesis is fast and produces good looking text. Another aspect that was not evaluated in the tests is the memory usage. As mentioned in the background an usual way to render text is to render the text in an image and draw it on the GPU as a texture. Interpolating an image of a

glyph with a sharp boundary between the foreground and the background will result in decreased sharpness of the rendered glyph. Therefore it is often desired to render an image to the screen with a one to one mapping between the pixels in the image and the pixels on the screen.

Distance fields can be scaled up and down without losing much quality at the border of the glyph. Assume that a distance field with size 128x128 is created. The distance field only contains the alpha value of the RGBA. This means that the distance field will use $128 * 128 * 8 = 131072$ bits if uncompiled. An 128x128 distance field can easily be used to render a good looking glyph at 500x500 pixels. If the glyph was rendered to an image at the CPU and sent to the GPU a 500x500 image would be optimal to use. A 500x500 pixel RGBA image of a glyph will use $500 * 500 * 4 * 8 = 8000000$ bits of memory if uncompiled which is more than 61 times the memory of the distance field rendering.

8

Future work

The focus of this thesis has been around generating distance fields fast that produces good looking text. There are other factors that matters when rendering text to the screen. For example how the glyphs are stored in textures for the GPU to be able to render the text fast. The implementation of this thesis is far from optimal when it comes to the rendering. This is something that needs more work. The implementation of text rendering done in this thesis renders every character with a separate draw call to the GPU on a separate quad. For example, a text containing 10000 characters will perform 10000 draw calls per frame and 40000 vertices for the GPU to work with.

A good thing with distance fields are the possibility to do text effects. For example a red border can easily be added to a glyph by setting the color red in the fragment shader if the interpolated distance value in that pixel is in the range [0.5,0.6]. Another possible effect is to use some kind of function over the boundry between the inside and the outisde of the glyph. This could create cool effects with for example sinus outline of the glyph. To create even better and more living effects the current time could be sent to the fragment shader and used in the fragment shader calculations creating for example moving waves over the outlines of a glyph or a glyph with pulsating size.

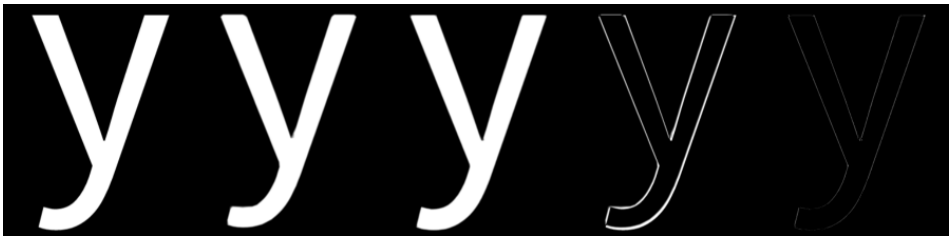
Appendix

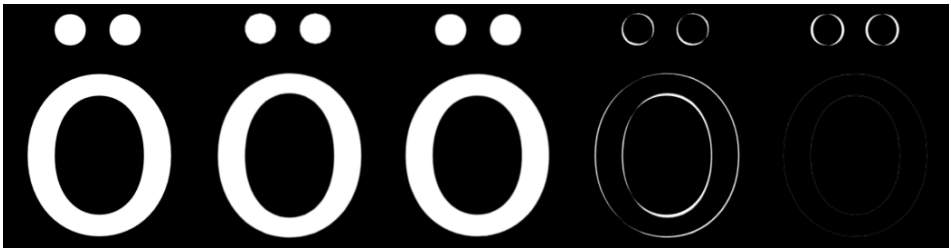
A

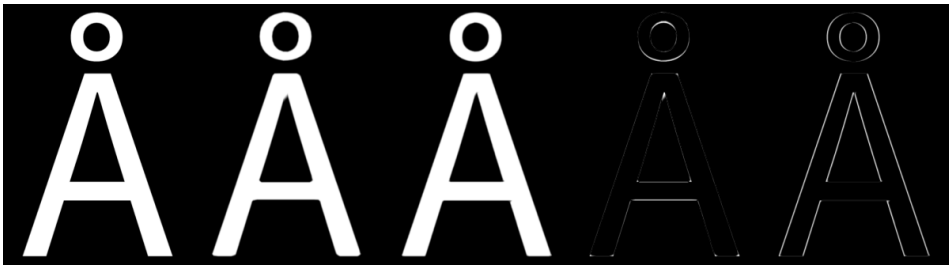
Result from visual evaluation #1

This appendix contain the complete test result from visual evaluation #1. Every sequence of images has the following order.

1. Glyph rendered with the old text rendering implementation
2. Glyph rendered with the new text rendering implementation with distance field size 64x64.
3. Glyph rendered with the new text rendering implementation with distance field size 128x128.
4. Comparison between first and second image.
5. Comparison between first and third image.







B

Result from performance evaluation

This appendix gives the complete result of the performance evaluation. A total of 14 tests were run on three different devices in two different distance field sizes. The values of the result tables are measured in seconds. The following devices and distance field sizes were used running the tests.

Test1 Samsung galaxy tab 1, 64x64 distance field

Test2 Samsung S6, 64x64 distance field

Test3 Iphone 4, 64x64 distance field

Test4 Iphone 4, 64x64 distance field

Test5 Iphone 4, 64x64 distance field

Test6 Iphone 4, 64x64 distance field

Test7 Iphone 4, 64x64 distance field

Test8 Samsung galaxy tab 1, 128x128 distance field

Test9 Samsung S6, 128x128 distance field

Test10 Iphone 4, 128x128 distance field

Test11 Iphone 4, 128x128 distance field

Test12 Iphone 4, 128x128 distance field

Test13 Iphone 4, 128x128 distance field

Test14 Iphone 4, 128x128 distance field

	Test1	Test2	Test3	Test4	Test5	Test6	Test7
A	0.00529	0.00227	0.00599	0.00703	0.00943	0.00253	0.00000
B	0.01198	0.00305	0.00000	0.00485	0.00012	0.00124	0.01349
C	0.00961	0.00378	0.01039	0.00028	0.00061	0.00303	0.00223
D	0.00706	0.00353	0.00000	0.00000	0.00127	0.00000	0.00291
E	0.00161	0.00082	0.00000	0.00000	0.00000	0.00000	0.00000
F	0.00148	0.00074	0.00000	0.00087	0.00094	0.00000	0.00000
G	0.00984	0.00436	0.00000	0.00000	0.00058	0.00063	0.00350
H	0.00181	0.00062	0.00076	0.00038	0.00159	0.00148	0.00000
I	0.00111	0.00035	0.00000	0.00000	0.00000	0.00000	0.00096
J	0.00493	0.00153	0.00043	0.00000	0.00105	0.00000	0.00000
K	0.00345	0.00102	0.00094	0.00045	0.00000	0.00154	0.00052
L	0.00133	0.00039	0.00000	0.00000	0.00000	0.00000	0.00109
M	0.00495	0.00153	0.00000	0.00000	0.00194	0.00000	0.00000
N	0.00377	0.00120	0.00170	0.00000	0.00193	0.00000	0.00000
O	0.01189	0.00377	0.00000	0.00364	0.00000	0.00357	0.00831
P	0.00642	0.00204	0.00000	0.00000	0.00319	0.00328	0.00000
Q	0.01597	0.00457	0.00448	0.00552	0.00295	0.00520	0.00831
R	0.00698	0.00207	0.00000	0.00000	0.00233	0.00000	0.00000
S	0.01200	0.00341	0.01000	0.00439	0.00326	0.01000	0.01000
T	0.00146	0.00039	0.00000	0.00068	0.00000	0.00000	0.00000
U	0.00671	0.00189	0.00363	0.00000	0.00289	0.00000	0.00000
V	0.00399	0.00107	0.00000	0.00000	0.00175	0.00000	0.00509
W	0.00702	0.00185	0.00000	0.00000	0.00000	0.01001	0.00154
X	0.00454	0.00126	0.00714	0.00194	0.00050	0.00000	0.00338
Y	0.00312	0.00082	0.00000	0.00042	0.00000	0.00416	0.00000
Z	0.00267	0.00069	0.00000	0.00000	0.00107	0.00000	0.00000
Å	0.00943	0.00261	0.00000	0.00304	0.00333	0.00750	0.00000
Ä	0.00885	0.00248	0.00073	0.00339	0.00024	0.00252	0.00000
Ö	0.01662	0.00477	0.00656	0.00032	0.00355	0.01003	0.01221
a	0.01021	0.00294	0.00129	0.00448	0.00352	0.00000	0.00000
b	0.00915	0.00265	0.00364	0.00312	0.00198	0.01000	0.00000
c	0.00865	0.00244	0.00333	0.00144	0.00000	0.00214	0.00000
d	0.00920	0.00262	0.00276	0.00380	0.00393	0.00000	0.00662
e	0.00993	0.00282	0.00340	0.00328	0.00449	0.01000	0.00000
f	0.00402	0.00119	0.00218	0.00168	0.00049	0.00000	0.00000
g	0.01747	0.00528	0.00350	0.00362	0.00751	0.00662	0.00661
h	0.00479	0.00140	0.00102	0.00275	0.00024	0.00000	0.00000
i	0.00360	0.00102	0.00000	0.00208	0.00000	0.01000	0.00000
j	0.00608	0.00170	0.00262	0.00000	0.00285	0.00000	0.01000
k	0.00295	0.00082	0.00065	0.00598	0.00000	0.00661	0.00000
l	0.00267	0.00077	0.00068	0.00000	0.00178	0.00000	0.00000
m	0.00851	0.00238	0.00000	0.00762	0.00000	0.00000	0.00381

	Test1	Test2	Test3	Test4	Test5	Test6	Test7
n	0.00476	0.00138	0.00112	0.00000	0.00270	0.00000	0.00000
o	0.01110	0.00316	0.00000	0.00882	0.00000	0.00000	0.01033
p	0.00913	0.00255	0.00351	0.00000	0.00805	0.00662	0.00000
q	0.00942	0.00270	0.00104	0.00782	0.00518	0.00000	0.00604
r	0.00332	0.00094	0.00144	0.00000	0.00423	0.00000	0.00134
s	0.01045	0.00293	0.00261	0.00877	0.00000	0.01000	0.00420
t	0.00392	0.00113	0.00000	0.00129	0.00751	0.00284	0.00162
u	0.00523	0.00145	0.00000	0.00000	0.00256	0.00000	0.00000
v	0.00323	0.00085	0.00064	0.00000	0.00000	0.00000	0.00000
w	0.00573	0.00151	0.00000	0.00454	0.00520	0.00720	0.00116
x	0.00399	0.00105	0.00000	0.00000	0.00000	0.00000	0.00000
y	0.00604	0.00162	0.00246	0.00329	0.00329	0.00000	0.00063
z	0.00223	0.00057	0.00000	0.00386	0.00000	0.00076	0.00000
å	0.01524	0.00441	0.00176	0.00363	0.00587	0.00587	0.00232
ä	0.01484	0.00426	0.00551	0.01187	0.00381	0.00000	0.00427
ö	0.01542	0.00458	0.00453	0.00756	0.00057	0.00568	0.00667
0	0.01093	0.00315	0.00735	0.00575	0.00286	0.00000	0.00162
1	0.00181	0.00054	0.00000	0.00000	0.00437	0.00040	0.00000
2	0.00725	0.00205	0.00000	0.00678	0.00000	0.00000	0.00059
3	0.01223	0.00358	0.00825	0.00393	0.00755	0.00434	0.00409
4	0.00279	0.00080	0.00000	0.00373	0.00000	0.00000	0.00000
5	0.01006	0.00264	0.00863	0.00000	0.00763	0.00379	0.00249
6	0.01641	0.00384	0.00000	0.01008	0.00824	0.00223	0.00426
7	0.00375	0.00102	0.00764	0.00000	0.00176	0.00035	0.00193
8	0.01537	0.00442	0.00761	0.01006	0.00301	0.00661	0.00292
9	0.01346	0.00390	0.00488	0.00562	0.00892	0.00010	0.00791
,	0.00411	0.00117	0.00396	0.00444	0.00451	0.00163	0.00404
.	0.00309	0.00086	0.00000	0.00000	0.00307	0.00000	0.00000
-	0.00085	0.00023	0.00515	0.00000	0.00128	0.00000	0.00471
+	0.00139	0.00039	0.00000	0.00000	0.00000	0.00000	0.00130
/	0.00270	0.00070	0.00235	0.00000	0.00000	0.00000	0.00000
|	0.00123	0.00034	0.00000	0.00564	0.00000	0.00000	0.00000
*	0.00283	0.00072	0.00000	0.00000	0.00000	0.00104	0.00000
^	0.00270	0.00069	0.00000	0.00326	0.00884	0.00000	0.00765
>	0.00269	0.00068	0.00771	0.00000	0.00000	0.00000	0.00240
<	0.00261	0.00067	0.00098	0.00101	0.00314	0.00000	0.00000
(	0.00466	0.00126	0.00373	0.00271	0.00000	0.00183	0.00000
)	0.00463	0.00125	0.00000	0.00000	0.00574	0.00000	0.00631
{	0.00811	0.00223	0.00536	0.00784	0.00240	0.00200	0.00387
}	0.00800	0.00227	0.00000	0.00247	0.00305	0.00354	0.00587
&	0.01513	0.00437	0.01062	0.00983	0.00244	0.00490	0.00419
’	0.00138	0.00035	0.00000	0.00000	0.00644	0.00000	0.00000
?	0.00899	0.00250	0.00781	0.00564	0.00362	0.00000	0.00000
!	0.00418	0.00120	0.00000	0.00000	0.00000	0.00226	0.00876
_	0.00093	0.00024	0.00453	0.00000	0.00561	0.00000	0.00000
@	0.02234	0.00670	0.01005	0.01007	0.01035	0.00960	0.00889

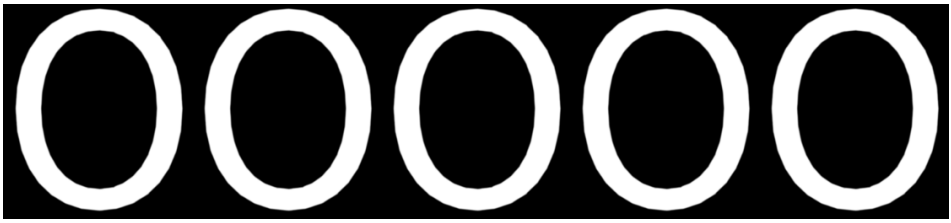
	Test8	Test9	Test10	Test11	Test12	Test13	Test14
A	0.01253	0.00738	0.00000	0.00563	0.00000	0.01012	0.00363
B	0.02252	0.01143	0.01387	0.00696	0.00962	0.00892	0.00647
C	0.02494	0.01263	0.01003	0.00702	0.01000	0.00429	0.00934
D	0.01894	0.00974	0.01000	0.01000	0.00000	0.00281	0.00728
E	0.00408	0.00217	0.00398	0.00000	0.01000	0.00065	0.00053
F	0.00369	0.00194	0.00000	0.00000	0.00000	0.00108	0.00000
G	0.02592	0.01351	0.01271	0.01312	0.01000	0.00592	0.00903
H	0.00478	0.00264	0.00065	0.00000	0.00000	0.00000	0.00000
I	0.00260	0.00141	0.00049	0.01000	0.00000	0.00000	0.00604
J	0.01122	0.00561	0.00419	0.00000	0.01135	0.00240	0.00358
K	0.00960	0.00399	0.00358	0.00000	0.00000	0.00183	0.00000
L	0.00308	0.00124	0.00197	0.00000	0.00000	0.00102	0.00000
M	0.01342	0.00488	0.00473	0.01000	0.00072	0.00342	0.00442
N	0.01011	0.00370	0.00031	0.00000	0.00575	0.00132	0.00000
O	0.03122	0.01171	0.00930	0.01181	0.01428	0.00967	0.01490
P	0.01432	0.00532	0.00000	0.00473	0.00675	0.00369	0.00910
Q	0.03684	0.01421	0.01361	0.01527	0.01521	0.00934	0.00752
R	0.01526	0.00604	0.00550	0.00624	0.00306	0.00014	0.01000
S	0.02733	0.00791	0.00221	0.01074	0.00253	0.00793	0.00661
T	0.00338	0.00103	0.00000	0.00000	0.00116	0.00000	0.01000
U	0.01510	0.00429	0.00455	0.00106	0.00000	0.00602	0.00656
V	0.01109	0.00291	0.00008	0.00000	0.00239	0.00326	0.00000
W	0.01984	0.00520	0.00502	0.00086	0.00072	0.00212	0.00987
X	0.01243	0.00332	0.00301	0.00000	0.00265	0.00000	0.00536
Y	0.00842	0.00224	0.00000	0.00000	0.00101	0.00169	0.00427
Z	0.00715	0.00193	0.00226	0.00633	0.00180	0.00060	0.00000
Å	0.02302	0.00672	0.00389	0.00830	0.00383	0.00810	0.01219
Ä	0.01981	0.00538	0.00326	0.00322	0.00301	0.00662	0.00000
Ö	0.03911	0.01153	0.01311	0.00959	0.00887	0.00782	0.02015
a	0.02288	0.00669	0.00724	0.00824	0.00000	0.00777	0.00868
b	0.01947	0.00560	0.00341	0.00495	0.01020	0.00604	0.00736
c	0.01879	0.00536	0.00235	0.00191	0.01000	0.00547	0.00581
d	0.01958	0.00560	0.01007	0.00148	0.00662	0.00561	0.00562
e	0.01988	0.00570	0.01006	0.00656	0.01000	0.00565	0.00684
f	0.00924	0.00268	0.00000	0.00386	0.00657	0.00314	0.00338
g	0.03898	0.01122	0.01734	0.00870	0.01582	0.01200	0.00948
h	0.01178	0.00349	0.00814	0.00259	0.00272	0.00607	0.00212
i	0.00683	0.00201	0.00000	0.00000	0.00000	0.00000	0.00178
j	0.01202	0.00351	0.00757	0.00000	0.01376	0.01000	0.00482
k	0.00829	0.00229	0.00474	0.00199	0.00000	0.00000	0.00000
l	0.00655	0.00191	0.00532	0.00360	0.00000	0.00000	0.00163
m	0.01908	0.00561	0.00622	0.00696	0.01000	0.01000	0.00711

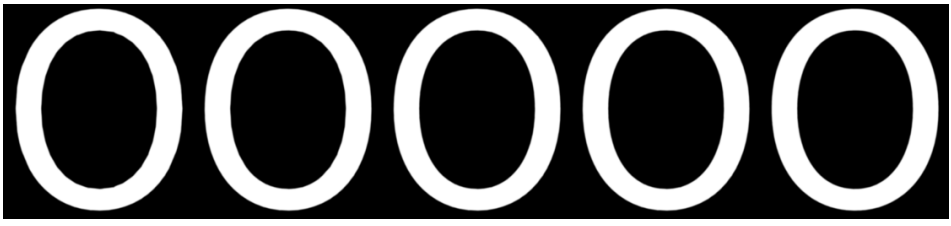
	Test8	Test9	Test10	Test11	Test12	Test13	Test14
n	0.01144	0.00337	0.00426	0.00000	0.00000	0.00660	0.00427
o	0.02197	0.00632	0.00810	0.00814	0.01000	0.00000	0.00609
p	0.01979	0.00579	0.00552	0.00625	0.00661	0.01576	0.00461
q	0.01939	0.00601	0.01006	0.00563	0.01008	0.00000	0.00723
r	0.00858	0.00246	0.00535	0.00000	0.00160	0.01179	0.00000
s	0.02291	0.00664	0.00537	0.00852	0.00875	0.00965	0.00706
t	0.00947	0.00276	0.00100	0.00094	0.00244	0.00000	0.00270
u	0.01189	0.00337	0.00189	0.00390	0.00493	0.01083	0.00433
v	0.00869	0.00227	0.00323	0.00291	0.00111	0.00000	0.00178
w	0.01707	0.00414	0.00223	0.00000	0.00345	0.00822	0.00069
x	0.01050	0.00270	0.00235	0.00038	0.00311	0.00000	0.00253
y	0.01463	0.00398	0.00381	0.00000	0.00000	0.00656	0.00501
z	0.00596	0.00155	0.00000	0.00000	0.00056	0.00000	0.00302
å	0.03451	0.01023	0.01209	0.01001	0.01307	0.01666	0.01310
ä	0.03108	0.00904	0.00820	0.01122	0.00684	0.00966	0.00977
ö	0.03034	0.00868	0.00931	0.00924	0.01062	0.01044	0.01565
0	0.02569	0.00732	0.00932	0.00963	0.00813	0.00978	0.00673
1	0.00469	0.00141	0.00038	0.00201	0.00090	0.00219	0.00000
2	0.01687	0.00482	0.00501	0.00633	0.00509	0.00079	0.00639
3	0.02755	0.00811	0.00769	0.00708	0.00655	0.00966	0.01058
4	0.00737	0.00208	0.00096	0.00000	0.00111	0.00122	0.00000
5	0.02024	0.00578	0.00449	0.00511	0.00509	0.00442	0.01239
6	0.03037	0.00919	0.00414	0.00881	0.00645	0.00711	0.01200
7	0.00854	0.00232	0.00367	0.00281	0.00034	0.00383	0.00531
8	0.03245	0.00940	0.01068	0.01028	0.00997	0.00921	0.01302
9	0.03049	0.00877	0.00558	0.00940	0.00939	0.00628	0.00714
,	0.01015	0.00291	0.00371	0.00386	0.00439	0.00390	0.01006
.	0.00681	0.00193	0.00000	0.00207	0.00262	0.00000	0.00000
-	0.00166	0.00043	0.00000	0.00000	0.00000	0.00138	0.00584
+	0.00320	0.00095	0.00221	0.00000	0.00017	0.00061	0.00000
/	0.00787	0.00200	0.00163	0.00259	0.00134	0.00000	0.00425
|	0.00292	0.00094	0.00000	0.00000	0.00090	0.00000	0.00000
*	0.00710	0.00182	0.00461	0.00000	0.00000	0.00269	0.00000
^	0.00713	0.00186	0.00000	0.00267	0.00201	0.00000	0.00767
>	0.00744	0.00185	0.00000	0.00000	0.00086	0.00030	0.00000
<	0.00743	0.00184	0.00122	0.00000	0.00201	0.00180	0.00688
(	0.01211	0.00349	0.00000	0.00176	0.00255	0.00466	0.00059
)	0.01208	0.00331	0.01007	0.00103	0.00221	0.00362	0.00721
{	0.02134	0.00558	0.01006	0.00282	0.00178	0.00328	0.00341
}	0.01950	0.00554	0.00666	0.00310	0.00272	0.00696	0.01188
&	0.03502	0.01005	0.01345	0.01720	0.01143	0.00853	0.01448
’	0.00330	0.00086	0.00305	0.00247	0.00090	0.00801	0.00122
?	0.02040	0.00593	0.00886	0.01025	0.00826	0.00198	0.00884
!	0.01042	0.00293	0.00000	0.00396	0.00000	0.00945	0.00459
_	0.00189	0.00052	0.00000	0.00000	0.00000	0.00000	0.00000
@	0.05605	0.01626	0.01829	0.02411	0.01819	0.01661	0.02567

C

Result from distance tolerance evaluation

The distance tolerance evaluation ended up with two types of result. The first result was the images of the renderings which are presented below. From left to right, top to bottom the renderings use the distance_tolerance 128.0, 1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2 and 0.1.





The second result from the distance tolerance evaluation was the distance field generation time for the different values of the constant. The second result is presented below.

Table C.1: *Results of performance evaluation*

Value	Time
128.0	0.009545 s
1.0	0.009559 s
0.9	0.009113 s
0.8	0.009112 s
0.7	0.009132 s
0.6	0.009296 s
0.5	0.010477 s
0.4	0.011286 s
0.3	0.010185 s
0.2	0.011947 s
0.1	0.016554 s
0.05	0.017990 s
0.01	0.033086 s

Bibliography

Apple Inc. Truetype™ reference manual, 2015. URL https://developer.apple.com/fonts/TrueType-Reference-Manual/. Cited on page 5.

Xiao-Diao Chen, Yin Zhou, Zhenyu Shu, Hua Su, and J.-C. Paul. Improved algebraic algorithm on point projection for beziércurves. In *Computer and Computational Sciences, 2007. IMSCCS 2007. Second International Multi-Symposiums on*, pages 158–163, Aug 2007. doi: 10.1109/IMSCCS.2007.17. Cited on page 9.

Olivier Cuisenaire and Benoît Macq. Fast euclidean distance transformation by propagation using multiple neighborhoods. *Computer Vision and Image Understanding*, 76(2):163–172, 1999. Cited on page 14.

Per-Erik Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14(3):227 – 248, 1980. ISSN 0146-664X. doi: http://dx.doi.org/10.1016/0146-664X(80)90054-4. URL http://www.sciencedirect.com/science/article/pii/0146664X80900544. Cited on pages 13, 14, 17, and 38.

Hinnik Eggers. Two fast euclidean distance transformations in z 2 based on sufficient propagation. *Computer Vision and Image Understanding*, 69(1):106–116, 1998. Cited on page 14.

Ricardo Fabbri, Luciano Da F. Costa, Julio C. Torelli, and Odemir M. Bruno. 2d euclidean distance transform algorithms: A comparative survey. *ACM Comput. Surv.*, 40(1):2:1–2:44, February 2008. ISSN 0360-0300. doi: 10.1145/1322432.1322434. URL http://doi.acm.org/10.1145/1322432.1322434. Cited on page 14.

Kaspar Fischer. Piecewise linear approximation of bezier curves. In *HTTP://HCKLBRRFNN. WORDPRESS. COM/2012/08/20/PIECEWISE-LINEAR-APPROXIMATION-OF-BEZIER-CURVES/*. Citeseer, 2000. Cited on pages 9 and 18.

FreeType. Freetype outlines, 2014a. URL http://www.freetype.org/freetype2/docs/glyphs/glyphs-6.html. Cited on page 4.

FreeType. Kerning, 2014b. URL http://www.freetype.org/freetype2/docs/glyphs/glyphs-4.html. Cited on page 5.

Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, pages 9–18, New York, NY, USA, 2007. ACM. ISBN 978-1-4503-1823-5. doi: 10.1145/1281500.1281665. URL http://doi.acm.org/10.1145/1281500.1281665. Cited on page 15.

Stefan Gustavson. 2d shape rendering by distance fields. 2012. Cited on pages 17 and 24.

Stefan Gustavson and Robin Strand. Anti-aliased euclidean distance transform. *Pattern Recognition Letters*, 32(2):252 – 257, 2011. ISSN 0167-8655. doi: http://dx.doi.org/10.1016/j.patrec.2010.08.010. URL http://www.sciencedirect.com/science/article/pii/S0167865510002953. Cited on pages 15 and 17.

M. Jones, J.A. Baerentzen, and M. Sramek. 3d distance fields: a survey of techniques and applications. *Visualization and Computer Graphics, IEEE Transactions on*, 12(4):581–599, July 2006. ISSN 1077-2626. doi: 10.1109/TVCG.2006.56. Cited on page 7.

Roberto A Lotufo and Francisco A Zampirolli. Fast multidimensional parallel euclidean distance transform based on mathematical morphology. In *Computer Graphics and Image Processing, 2001 Proceedings of XIV Brazilian Symposium on*, pages 100–105. IEEE, 2001. Cited on page 14.

Calvin R Maurer Jr, Vijay Raghavan, and Rensheng Qi. A linear time algorithm for computing the euclidean distance transform in arbitrary dimensions. In *Information Processing in Medical Imaging*, pages 358–364. Springer, 2001. Cited on page 14.

Arnold Meijster, Jos BTM Roerdink, and Wim H Hesselink. A general algorithm for computing distance transforms in linear time. In *Mathematical Morphology and its applications to image and signal processing*, pages 331–340. Springer, 2000. Cited on pages 14, 17, and 38.

Microsoft Corporation. Opentype specification, 2015. URL https://www.microsoft.com/typography/otspec/default.htm. Cited on page 5.

Thomas W Phinney. Truetype, postscript type 1, & opentype: What's the difference, 2001. Cited on page 8.

Ingemar Ragnemalm. The euclidean distance transform and its implementation on simd architectures. 1989. Cited on page 14.

Ingemar Ragnemalm. *The Euclidean Distance Transform*. Linköping Studies in Science and Technology, first edition, 1993. ISBN 91-7871-083-9. Cited on pages 6 and 14.

Ingemar Ragnemalm. *Polygons feel no pain*. First edition, 2008. Cited on pages 7 and 8.

Azriel Rosenfeld and John L Pfaltz. Sequential operations in digital picture processing. *Journal of the ACM (JACM)*, 13(4):471–494, 1966. Cited on pages 6 and 14.

Toyofumi Saito and Jun-Ichiro Toriwaki. New algorithms for euclidean distance transformation of an n-dimensional digitized picture with applications. *Pattern recognition*, 27(11):1551–1565, 1994. Cited on page 14.

Maxim Shemanarev. Adaptive subdivision of bezier curves. 2005. URL `http://antigrain.com/research/adaptive_bezier/`. Cited on page 19.

Dave Shreiner, Bill The Khronos OpenGL ARB Working Group, et al. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education, 2009. Cited on page 9.