INSTITUT FÜR INFORMATIK
Algorithmen und Datenstrukturen

Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

BACHELOR THESIS

Rendering of Vector Graphics on the GPU

Author:
Thomas GERMER

Supervisors:
Prof. Dr. Egon WANKE
PD Dr. Dr.-Ing. Wilfried LINDER

September 18, 2016

Declaration of Authorship

Hiermit versichere ich, Thomas GERMER, dass ich diese Bachelorarbeit selbstständig verfasst und dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Signed:

Date:

Acknowledgements

I would like to thank my parents for their patience and my supervisors and everyone in our study group for their valuable feedback and ideas.

Abstract

We describe several methods to render vector graphics comprised of line segments and Bézier curves on the GPU at the cost of some preprocessing time.

Two methods entail approximating curves with circular arcs. The first method renders those arcs using a stencil buffer while the second method further decomposes the arrangement of line segments and circular arcs into parts that are directly consumable by common graphics APIs like OpenGL.

Two other methods approximate curves with line segments, thereby defining a polygon. The polygon is drawn by either using a stencil buffer or by decomposing the polygon into triangles.

Finally, the rendering times of all methods are measured.

Contents

List of Abbreviations

API	**A**plication **P**rogramming Interface
BMP	**B**it**M**a**P** image file
GIF	**G**raphics **I**nterchange **F**ormat
JPG or **JPEG**	**J**oint **P**hotographic **E**xperts **G**roup
PNG	**P**ortable **N**etwork **G**raphics
CPU	**C**entral **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
OpenGL	Open **G**raphics **L**ibrary
GLSL	Open**GL** **S**hading **L**anguage
SDF	**S**igned **D**distance **F**ield
SVG	**S**calable **V**ector **G**raphics
DCEL	**D**oubly **C**onnected **E**dge **L**ist

Chapter 1

Introduction

1.1 Motivation

Common raster graphics such as BMP, GIF, JPG and PNG, store the color values of pixels in a regularly spaced grid. This is efficient if the image does not contain easily describable patterns, but has the disadvantage that the underlying grid structure will become apparent at higher resolutions.

On the other hand, vector graphics define shapes like circles and polygons with additional information on how their interior and outline is to be colored. This allows them to be drawn at any desired resolution so they can be displayed on a computer screen without block artifacts. With the popularization of high resolution displays, for example in mobile phones, tablets and TVs, this feature is becoming more important, since transmitting and storing raster graphics at a comparable quality level would require excessive amounts of memory.

However, to rasterize, render or draw larger vector graphics on the fly, i.e. converting vector graphics to raster graphics for displaying, at higher screen resolution, more processing power is required.

CPU-based rendering has been popular in the past, but it has been shown that GPU-accelerated renderers can potentially have superior performance (Kilgard and Bolz, 2012).

1.2 General terminology

The *outlines* of shapes in vector graphics are stored as *paths*, which are connected curves like circular arcs, elliptic arcs, line segments and Bézier curves. The outline might also be called a *general polygon*, as opposed to a common polygon which only has straight edges. The outline encloses a region, which is a subset of points in $\mathbb{R}^2$. Points are defined to be inside the region based on a *fill rule*.

A collection of curves in no particular order can be referred to as an *arrangement*. A curve endpoint can also be described as a *vertex*, which might also store additional information.

Vector quantities, for example the vector $\mathbf{v}$, are printed in bold, while scalar values like x are printed in italic.

1.3 OpenGL-specific terminology

The description of the following OpenGL terms are simplified and of minimal length to understand the described algorithms. For more in-depth

information, see the official documentation (*OpenGL 4.5 API Specification* 2016).

OpenGL is a library to interface with the GPU to draw certain primitives like triangles, lines and points. Each point, or vertex, of a primitive can be accompanied by a small number of arbitrary values, for example red, green and blue color values, or space coordinates.

1.3.1 Rasterization

After the vertices with their per-vertex data have been defined, they are sent to the GPU for rasterization, i.e. finding the pixels that are covered by a primitive.

A tie-breaking rule exists to ensure that pixels which are located exactly on the edge between two triangles are only rasterized once.

1.3.2 Fragment shader

Once a pixel is rasterized, a user-defined fragment shader program is executed to calculate the final pixel color. The input to such a program are the per-vertex values, for example color values (Figure 1.1), which are interpolated across the triangle by the graphics hardware.

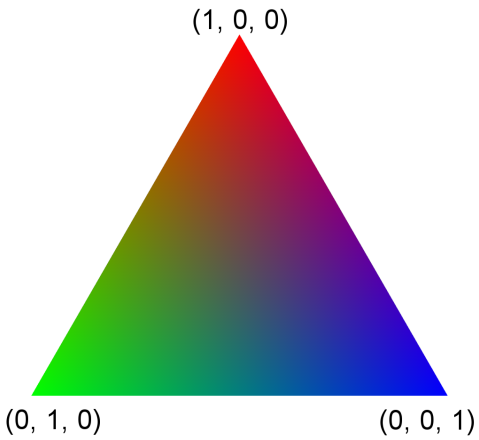


FIGURE 1.1: A triangle with green (left), red (top) and blue (right) vertex RGB triplets.

1.3.3 GLSL

The language in which shader programs are written is the GL shading language (GLSL) which looks very much like C with the addition of vector types for convenient operations on color values and coordinates. For example see the following fragment shader which colors pixels white if they are inside a circle centered at $(256, 256)$ with a radius of 128 pixels, or black otherwise.

```
void main(){
        // Position of the pixel on the screen
        vec2 position = gl_FragCoord.xy;

        float radius = 128.0;
        vec2 center = vec2(256.0, 256.0);

        // If the distance between the pixel position
        // and the center is less than the radius
        // color it
        if (distance(position, center) < radius){
                // white
                gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
        }else{
                // black
                gl_FragColor = vec4(0.0, 0.0, 0.0, 0.0);
        }
}
```

1.3.4 Textures

Textures can be understood as a collection of raster images stored in GPU memory. Their pixels can be accessed in GLSL at their texture coordinates. If a texture access is made at a position that does not correspond exactly to a texture coordinate, the color of the nearest pixel or an interpolated color based on the distance to the surrounding pixels is returned, depending on which configuration was requested.

1.3.5 Framebuffer objects

Instead of drawing objects on the screen directly, they can be rendered into a framebuffer object (FBO) which can have several images attached to it as storage for various buffers. This has the advantage that the screen content is preserved. FBOs can be used to compose graphics and to store them for later use.

1.3.6 Stencil buffer

In addition to the color buffer, which holds the pixel colors produced by the fragment shader to be displayed on the screen, there is also the stencil buffer, which holds a discrete value for each pixel. Commonly the stencil buffer is used to define regions that can be rendered to while not disturbing other pixels.

1.4 Problem statement and Related Work

The main task for rendering vector graphics is to determine which pixels lie inside shapes that are concave, have holes, or are possibly self-intersecting. This matter is further complicated by the fact that GPUs can usually only cover convex shapes natively. We review some of the most common solutions for this problem.

1.4.1 Line segment approximation with subsequent triangulation

A straightforward method to render concave polygons is to first approximate curves with line segments. The resulting polygon with straight edges is then decomposed into triangles (triangulated) which can be rendered by GPUs.

One disadvantage of this method is that a high number of line segments can be required to make approximated curves appear smooth at high resolution (Figure 1.2).

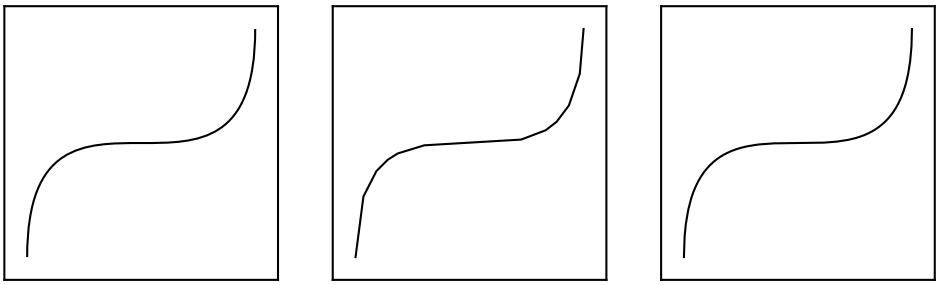


FIGURE 1.2: Bézier curve (left), subdivided into 11 and 44 line segments with a difference of less than 20 (middle) respectively 5 degrees (right) in orientation.

1.4.2 Signed Distance Fields

Another method is the signed distance field (SDF) (Green, 2007). The signed distance from a point to a path is the shortest distance to the path. If the point is inside the path, the signed distance is defined to be negative.

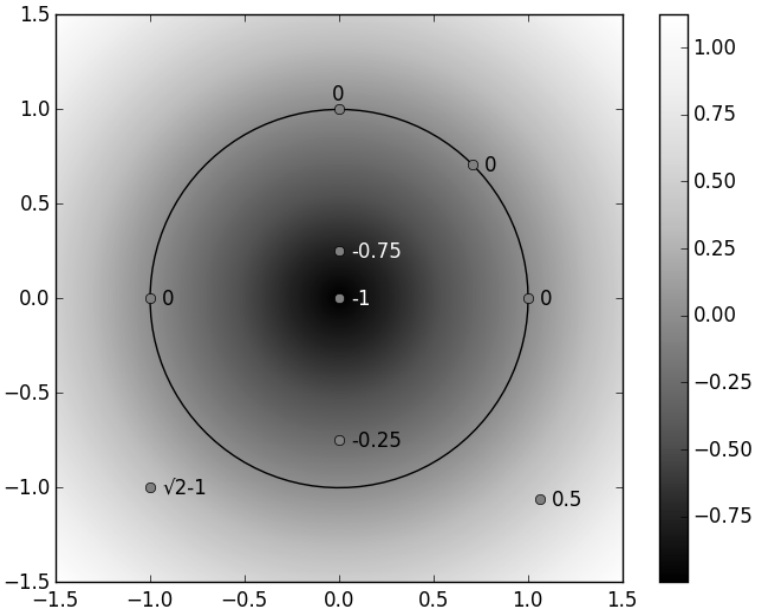


FIGURE 1.3: Signed distance field of a circle.

For example, the signed distance field of a circle (Figure 1.3) at a point $\mathbf{p}$ is the Euclidean distance to its center minus its radius:

$$\text{sdfCircle}(\mathbf{p}, \mathbf{center}, \mathit{radius}) = \|\mathbf{p} - \mathbf{center}\|_2 - \mathit{radius} \tag{1.1}$$

A signed distance field function is evaluated on the CPU for every point in a raster image. The image is then uploaded to the GPU to be used as a texture.

To extract the original shape from the distance field texture it is sufficient to test whether the SDF at a given location has a negative value which corresponds to being inside the shape. More involved pixel-coverage calculations for antialiasing are also possible.

The advantage of using signed distance field textures over regular textures is that the linear interpolation of the texture done by the GPU approximates the signed distance field quite well (Figure 1.4), even at low resolution, which saves texture memory.

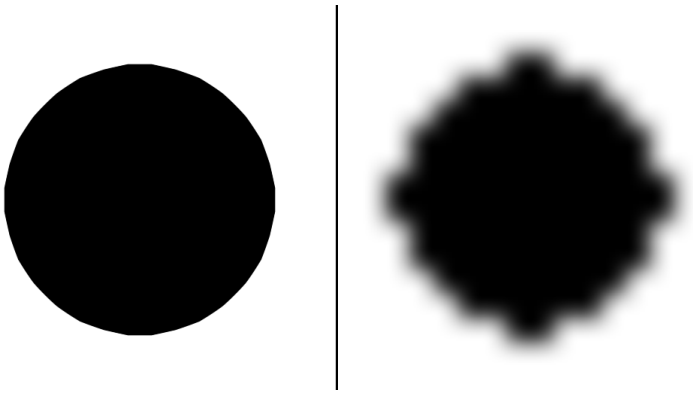


FIGURE 1.4: Circle recovered from a SDF (left) and regular image (right) upscaled from 16x16 to 512x512.

An open problem with SDFs was that sharp features of vector graphics were smoothed at low resolutions, but a solution has recently been found with multi-channel SDFs (Chlumský, 2015).

However, SDF textures still require a high resolution to resolve thin features such as lines, even when the graphics are otherwise sparse. Multiple edges that meet at the same vertex are a problem, too.

1.4.3 Texture encoded shapes

Some methods encode the shape information directly in the pixels of a texture but use computationally expensive indirect texture lookups (Nehab and Hoppe, 2008; Ray et al., 2005) and complicated fragment shaders.

Another method of this category is (Esfahbod, 2011), but is limited to simple shapes. However, in this method Bézier curves are approximated with circular arcs, which we found interesting and pursued further.

1.4.4 Stencil buffer

A long-known method to render concave polygons is making use of the stencil buffer (Neider, Davis, and Woo, 1997). For each consecutive pair of

vertices of the polygon, a triangle formed by the pair and a third point is drawn into a stencil buffer. The stencil buffer is set to toggle the inside-outside state of a pixel every time it is accessed. This way it is possible to determine if a pixel has been drawn an even or an odd number of times. Cover geometry is then drawn to color every pixel where the corresponding stencil buffer value was flipped an odd number of times.

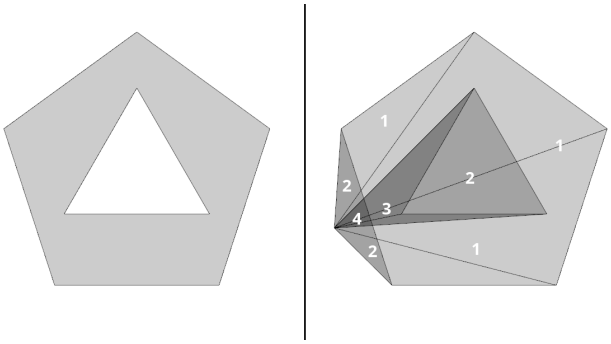


FIGURE 1.5: Polygon with hole (left) and number of times a pixel was overlapped by a triangle connected to a pivot point (right). The pivot point could also be chosen as a vertex of the polygon, at the cost of a less picturesque illustration.

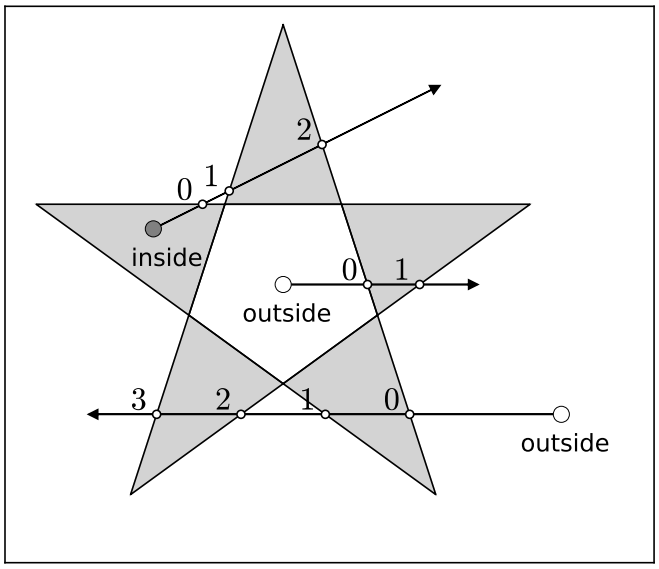


FIGURE 1.6: Example of even-odd rule on three points. The two white points are classified as being outside of the shape. The dark gray point is inside the shape.

The workings of this stencil buffer-based method are related to the following definition. Consider a polygon and a point. One way to define whether the point is inside or outside the polygon is the even-odd rule. A ray with origin at that point is traced towards any direction until there is no chance that it will ever intersect the geometry again, for example because it

escaped the polygon's bounding box. If the ray intersected the polygon an odd number of times, the point is inside.

Concave shapes with curves can be drawn in a similar manner. A polygon given by the control points is drawn into the stencil buffer as before. The curves are also drawn into the stencil buffer, but since OpenGL does not support curve primitives directly, they are packed into enclosing triangles first and a special fragment shader discards the pixels that are on the side of the curve that should not be filled.

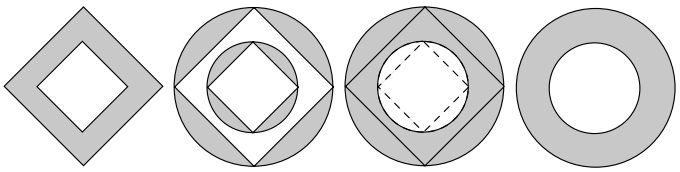


FIGURE 1.7: The polygon (left) given by the control points of the shape (right) is combined with filled curves (second) in the stencil buffer. Convex filled curves are added while concave regions are carved out (third) to form the final shape (right).

This technique is utilized in the `NV_path_rendering` OpenGL extension which as of now is only available with NVIDIA hardware.

A disadvantage of this method is that, depending on the shape, the number of pixels that have to be rasterized could far exceed the pixels that will be filled in the end.

Additionally, several related patents exist (Kilgard, 2014, 2016; Yhann, 2011; Yhann and Choudhury, 2011a,b).

1.4.5 Triangulation of curves

(Loop and Blinn, 2005, 2007) propose to triangulate the polygon created by the control points of the Bézier curves and to evaluate the curves in texture coordinate space. Curves with overlapping convex hulls have to be subdivided to avoid artifacts (Figure 1.8). It was noted that intersecting or osculating curves require future work.

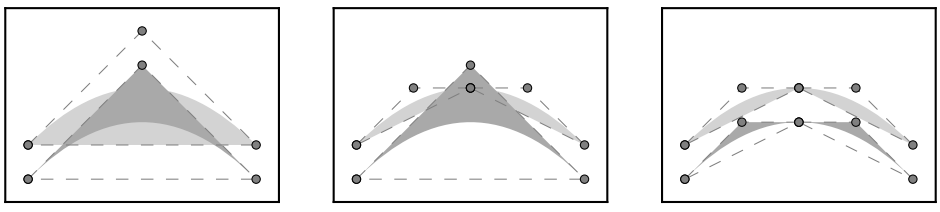


FIGURE 1.8: Two overlapping Bézier curves (left) after a first subdivision step (middle) and a second subdivision step (right).

A possible solution is described in Chapter 4 by inserting additional line segments and decomposing the arrangement of curves into trapezoids bounded by circular arcs and line segments. We later found a patent (Michail,

Teitlebaum, and Furtwangler, 2014) which proposes a triangulation and to combine multiple curves in a single triangle.

1.4.6 Miscellaneous

(Ganacim et al., 2014) built a tree acceleration structure which can be queried in parallel for pixel colors. A graphics API with general compute capability (CUDA) is required, which is not available on some platforms.

Chapter 2

Approximating curves with arcs

2.1 Bézier curves

Bézier curves are a popular tool in graphics software to create smooth curves given some points or tangency information. Let $\mathbf{P}_k \in \mathbb{R}^d$ with $k = 0, \ldots, n$ be points in d-dimensional space. The Bézier curve to those control points is defined as:

$$\mathbf{B}_n \colon [0,1] \to \mathbb{R}^d$$
$$t \mapsto \sum_{k=0}^{n} \binom{n}{k} (1-t)^{n-k} t^k \mathbf{P}_k$$

More specifically, we are only interested in quadratic and cubic Bézier curves in $\mathbb{R}^2$ (Figure 2.1) as used in the scalable vector graphics (SVG) specification:

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

The last and first point of a Bézier curve are the first and last control points:

$$\mathbf{B}_n(0) = \mathbf{P}_0$$
$$\mathbf{B}_n(1) = \mathbf{P}_1$$

Further, the line segment from the first to the second control point $\overline{P_0P_1}$ is tangent to the Bézier curve at $t = 0$. The same is true for the line segment from the second-last to the last control point $\overline{P_{n-1}P_n}$ at $t = 1$.

A cubic Bézier curve can in general not be represented exactly as a collection of quadratic Bézier curves, although a quadratic Bézier curve can exactly be represented as a cubic Bézier curve. Therefore we will only discuss cubic Bézier curves.

Let $\mathbf{Q}_0, \mathbf{Q}_1, \mathbf{Q}_2$ be the control points of the quadratic Bézier curve. The control points of the equivalent cubic Bézier curve then are

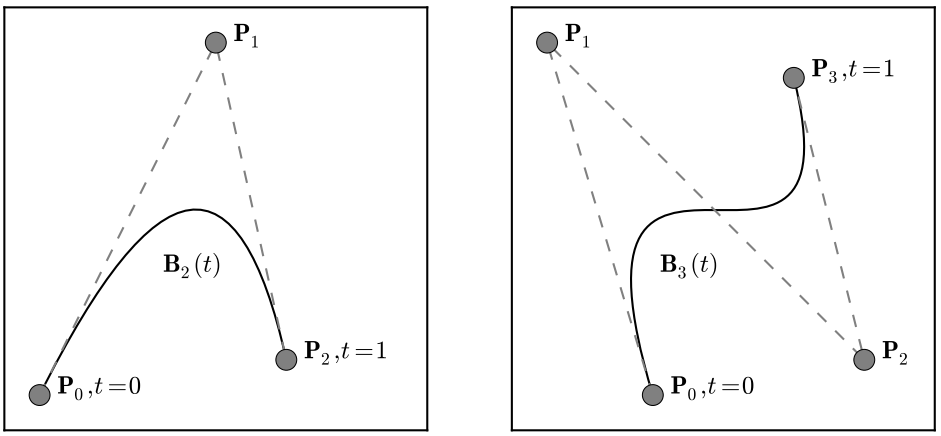


FIGURE 2.1: Quadratic (left) and cubic (right) Bézier curves with control points.

$$
\begin{aligned}
\mathbf{P}_0 &= \mathbf{Q}_0 \\
\mathbf{P}_1 &= \frac{1}{3}\mathbf{Q}_0 + \frac{2}{3}\mathbf{Q}_1 \\
\mathbf{P}_1 &= \frac{1}{3}\mathbf{Q}_2 + \frac{2}{3}\mathbf{Q}_1 \\
\mathbf{P}_3 &= \mathbf{Q}_2
\end{aligned}
$$

Some methods to render vector graphics might require the smallest distance to a curve, i.e. the euclidean distance between a point and the corresponding closest point on the curve. The distance is useful to calculate the pixel coverage for antialiased curve rendering or generating signed distance fields.

To calculate the minimal distance, we calculate the square root of the minimal squared distance, which is equivalent because the square root is strictly monotone for positive parameters such as squared distance.

The squared euclidean distance between a point $\mathbf{Q} = (\mathbf{Q}_x, \mathbf{Q}_y)$ and a Bézier curve $\mathbf{B} = (\mathbf{B}_x, \mathbf{B}_y)$ is

$$\|\mathbf{Q} - \mathbf{B}(t)\|_2^2 = (\mathbf{Q}_x - \mathbf{B}_x(t))^2 + (\mathbf{Q}_y - \mathbf{B}_y(t))^2$$

which for a cubic Bézier curve is a polynomial in t of degree 6 since $\mathbf{B}_{3,x}$ and $\mathbf{B}_{3,x}$ are of degree 3.

To get the smallest squared distance, we calculate the derivative with respect to t and set it equal to zero. The result is a polynomial of degree 5. It is known due to (Abel, 1824) that in general polynomials of degree five or higher have no algebraic solution in radicals. Therefore, numerical methods have to be used to calculate an approximation.

In contrast, calculating the smallest distance to a circular arc is trivial.

2.2 Circular arcs

A circular arc (Figure 2.2) can be described as a closed subset A of points on a circle or a parametric curve $\mathbf{A}$ given by a center point $\mathbf{c}$, a radius r, a start point $\mathbf{a}$, an end point $\mathbf{b}$ and a Boolean flag which indicates whether the arc direction is clockwise or counterclockwise. The representation can be chosen depending on which one is more convenient at the time.

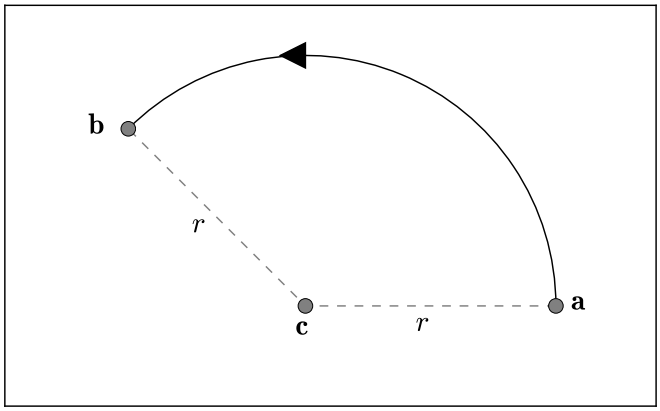


FIGURE 2.2: A counterclockwise circular arc.

The angles between the x-axis and the vectors from the center to the endpoints are

$$\alpha_a = \operatorname{atan2}(\mathbf{a}_y - \mathbf{c}_y, \mathbf{a}_x - \mathbf{c}_x)$$
$$\alpha_b = \operatorname{atan2}(\mathbf{b}_y - \mathbf{c}_y, \mathbf{b}_x - \mathbf{c}_x)$$

which are interpolated between with the function $\operatorname{arcAngle}$

$$\operatorname{arcAngle} : [0,1] \to [0, 2\pi] t \mapsto \left\{ \begin{array}{ll} \operatorname{lerp}(\alpha_a, \alpha_b + 2\pi[\alpha_a \leq \alpha_b], t), & \text{if clockwise} \\ \operatorname{lerp}(\alpha_a, \alpha_b - 2\pi[\alpha_b \leq \alpha_a], t), & \text{otherwise} \end{array} \right\}$$

where

$$[\text{condition}] = \left\{ \begin{array}{ll} 1, & \text{if condition} \\ 0, & \text{otherwise} \end{array} \right\}$$

denote Iverson brackets and

$$\operatorname{lerp} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$$
$$(\mathbf{a}, \mathbf{b}, t) \mapsto \mathbf{a}(1-t) + \mathbf{b}t$$

denotes linear interpolation. The above formula has the advantage over the alternative definition

$$(\mathbf{a}, \mathbf{b}, t) \mapsto \mathbf{a} + (\mathbf{b} - \mathbf{a})t$$

that $\operatorname{lerp}(\mathbf{a}, \mathbf{b}, 0) = \mathbf{a}$ and $\operatorname{lerp}(\mathbf{a}, \mathbf{b}, 1) = \mathbf{b}$ exactly, even if $\mathbf{b} - \mathbf{a}$ is not representable with the underlying number type.

Finally, the parametric equation $\mathbf{A}(t)$ for the circular arc curve is given by:

$$\mathbf{A} : [0,1] \to \mathbb{R}^2$$
$$t \mapsto \mathbf{c} + r \begin{bmatrix} \cos \text{arcAngle}(t) \\ \sin \text{arcAngle}(t) \end{bmatrix}$$

2.3 Biarcs

A biarc (Figure 2.3) is a composition of two circular arcs. The end of the first arc joins the beginning of the second arc in point **j**. Both arcs are tangential to each other at the joining point and do not necessarily have to have the same radii.

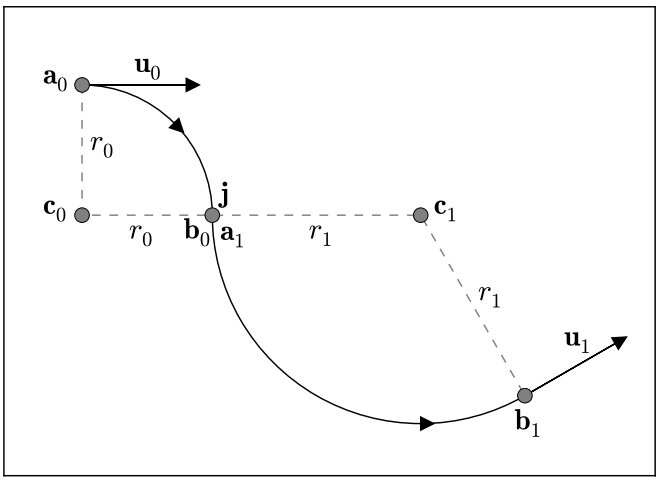


FIGURE 2.3: A biarc composed of a clockwise and a counterclockwise arc.

2.4 Biarc from points and tangents

Given two unit tangents tangents $\mathbf{u}_0$ and $\mathbf{u}_1$ at two points $\mathbf{a}_0$ and $\mathbf{b}_1$, it is possible to construct a family of biarcs. Their joining points lie on a circle (Šír et al., 2006).

The circle with center $\mathbf{c}_2$ and radius r_2 can be constructed by first finding the rotation matrix R

$$R = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$$

that rotates the unit tangent $\mathbf{u}_0$ on the unit tangent $\mathbf{u}_1$.

$$\mathbf{u}_1 = R\mathbf{u}_0$$

which are two equations with two unknowns, $\cos\alpha$ and $\sin\alpha$, that can be solved for. Note that it is not necessary to solve for α.

R is the same rotation matrix that rotates the point $\mathbf{a}_0$ around $\mathbf{c}_2$ on $\mathbf{b}_1$:

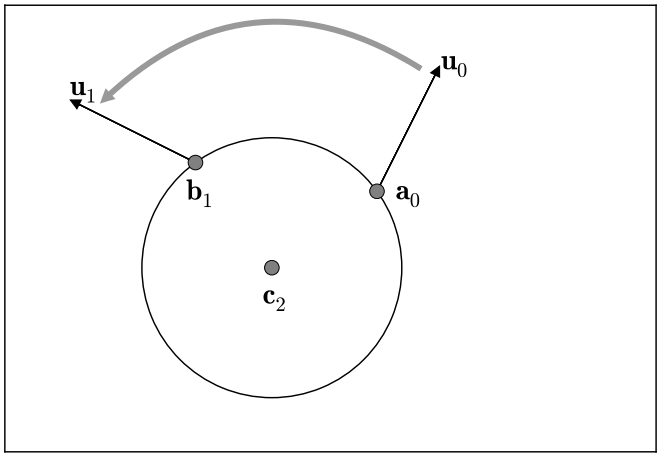


FIGURE 2.4: Rotating the tangent $\mathbf{u}_0$ to $\mathbf{u}_1$ on the circle of possible joining points.

$$\mathbf{b}_1 = R(\mathbf{a}_0 - \mathbf{c}_2) + \mathbf{c}_2$$

which can be solved for $\mathbf{c}_2$ which in turn can be used to find the radius

$$\mathbf{r}_2 = \|\mathbf{c}_2 - \mathbf{a}_0\|_2$$

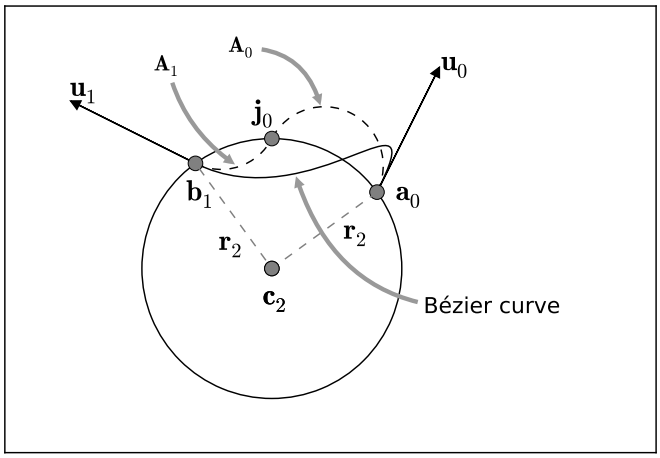


FIGURE 2.5: A biarc (dashed) approximating a Bézier curve with circle of possible joining points.

2.4.1 Error measure

To determine if a biarc is a "good enough" approximation to a curve, it makes sense to define an error measure. While several are discussed in (Safonova and Rossignac, 2003), we found the one-sided Hausdorff distance to be sufficient for our application:

$$d_H(\mathbf{B}(t), (A_0, A_1)) = \max_{t\in[0,1]}(\min_{p\in A_0\cup A_1} \|\mathbf{B}(t) - p\|_2)$$

A solver for the exact distance was written, which includes deriving a fifth-degree polynomial and finding its roots, but the one-sided Hausdorff distance estimated from a small number of curve points, also seen in (Šír et al., 2006), was found to work just as well.

2.4.2 Choice of joining point

While all joining points on the circle lead to correct biarcs, some have different curvature than the Bézier curve we want to approximate (Figure 2.6), so we exclude them.

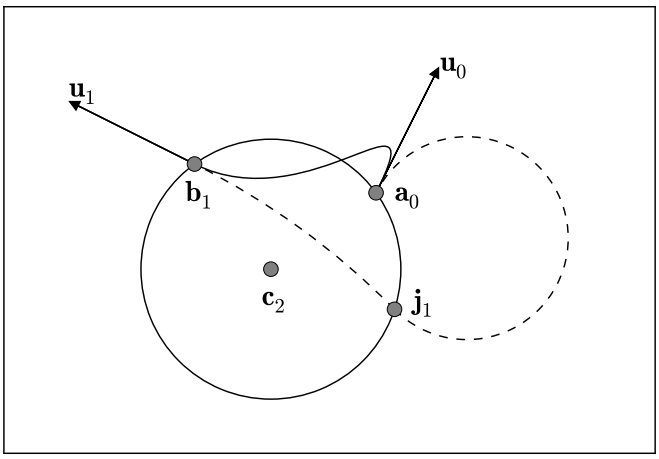


FIGURE 2.6: A biarc (dashed) with curvature different from the Bézier curve it should approximate.

(Šír et al., 2006) proposed to choose the joining point as the intersection of the Bézier curve with the arc of joining points where it compared favorably to two other methods. Care must be taken when the intersections should happen to lie on the endpoints (Figure 2.7).

Another option is to choose the point on the circle that is closest to $\mathbf{B}\left(\frac{1}{2}\right)$.

$$\mathbf{j} = \mathbf{c}_2 + r_2 \,\mathrm{normalized}\left(\mathbf{B}\left(\frac{1}{2}\right) - \mathbf{c}_2\right)$$

The $\mathrm{normalized}$ function obtains a unit vector $\mathbf{u}$ from a vector $\mathbf{v}$ so that its length is $\|\mathbf{u}\|_2 = 1$.

$$\mathrm{normalized} : \mathbb{R}^n \to \mathbb{R}^n, \mathbf{v} \mapsto \frac{\mathbf{v}}{\|\mathbf{v}\|_2}$$

2.5 Approximation

A recursive subdivision scheme is employed where we approximate a Bézier curve with a biarc. If the approximation error is above a user-defined threshold, we split the Bézier curve at the point where the approximation error was greatest and recursively approximate the two new Bézier curves.

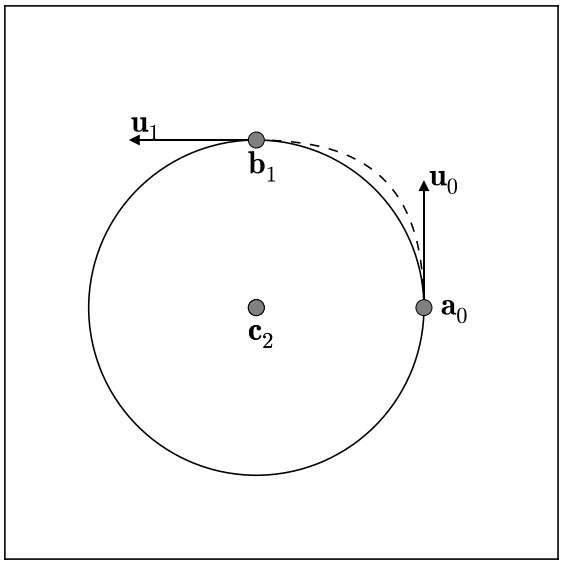


FIGURE 2.7: A Bézier curve that intersects the circle of joining points only in its endpoints.

2.6 Splitting cubic Bézier curves

To split a cubic Bézier curve $\mathbf{B}$ with control points $\mathbf{P}_i$ at point $\mathbf{B}(t)$ into two cubic Bézier curves $\mathbf{G}$ and $\mathbf{H}$ so that
$\mathbf{B}(r) = \mathbf{G}\left(\frac{r}{t}\right)$ for $r \in [0, t]$ and $\mathbf{B}(s) = \mathbf{H}\left(\frac{s-t}{1-t}\right)$ for $s \in [t, 1]$, evaluate

$$\begin{aligned}
\mathbf{P}_{01} &= \operatorname{lerp}(\mathbf{P}_0, \mathbf{P}_1, t) \\
\mathbf{P}_{12} &= \operatorname{lerp}(\mathbf{P}_1, \mathbf{P}_2, t) \\
\mathbf{P}_{23} &= \operatorname{lerp}(\mathbf{P}_2, \mathbf{P}_3, t)
\end{aligned}$$

$$\begin{aligned}
\mathbf{P}_{012} &= \operatorname{lerp}(\mathbf{P}_{01}, \mathbf{P}_{12}, t) \\
\mathbf{P}_{123} &= \operatorname{lerp}(\mathbf{P}_{12}, \mathbf{P}_{23}, t)
\end{aligned}$$

$$\mathbf{P}_{0123} = \operatorname{lerp}(\mathbf{P}_{012}, \mathbf{P}_{123}, t)$$

and choose the control points as

$$\begin{aligned}
&\mathbf{P}_0, \mathbf{P}_{01}, \mathbf{P}_{012}, \mathbf{P}_{0123} \\
&\mathbf{P}_{0123}, \mathbf{P}_{123}, \mathbf{P}_{23}, \mathbf{P}_3
\end{aligned}$$

for $\mathbf{G}$ and $\mathbf{H}$ respectively.

If the first control point is identical to the second control point, the tangent vector is not defined, in which case we replace the second control point with the third control point for tangent calculations, but keep the Bézier

curve constant. Likewise, we replace the third control point with the second control point if the third control point is identical to the last control point.

If both $\mathbf{P}_0 = \mathbf{P}_1$ and $\mathbf{P}_2 = \mathbf{P}_3$ then the Bézier curve degenerates to a line segment and does not need to be approximated with circular arcs.

If the tangents are parallel and three control points are on the same line, the fourth control point will be, too. In that case, we approximate with a line segment from the first to the last control point with the result that if the second and third control points are between the first and last control points, some zero-area region will be lost, which is not problematic, because in the end it would not have been drawn anyway.

The one-sided Hausdorff distance between $\mathbf{B}$ and $\mathbf{A}_0 \cup \mathbf{A}_1$ in line 17 of Algorithm 1 is not required to be exact. A guess obtained by testing a few values of t is usually sufficient, but might produce slightly more arcs.

We define the dot product, also called scalar product, of two vectors $\mathbf{u} = (\mathbf{u}_x, \mathbf{u}_y) \in \mathbb{R}^2$ and $\mathbf{v} = (\mathbf{v}_x, \mathbf{v}_y) \in \mathbb{R}^2$. Similarly, we also define the determinant of two vectors.

$$\begin{aligned}\operatorname{dot}(\mathbf{u}, \mathbf{v}) &= \mathbf{u}_x \mathbf{v}_x + \mathbf{u}_y \mathbf{v}_y \\ \det(\mathbf{u}, \mathbf{v}) &= \mathbf{u}_x \mathbf{v}_y - \mathbf{u}_y \mathbf{v}_x\end{aligned}$$

The determinant can be used to determine whether a point $\mathbf{p}$ lies to the left or to the right of a line through two points $\mathbf{a}$ and $\mathbf{b}$.

$$\begin{aligned}\operatorname{isLeftOf}(\mathbf{p}, \mathbf{a}, \mathbf{b}) &= \det(\mathbf{p} - \mathbf{a}, \mathbf{b} - \mathbf{a}) < 0 \\ \operatorname{isRightOf}(\mathbf{p}, \mathbf{a}, \mathbf{b}) &= \det(\mathbf{p} - \mathbf{a}, \mathbf{b} - \mathbf{a}) > 0\end{aligned}$$

Intuitively, this can be understood as testing the sign of the z-component of the cross product of the vectors $\mathbf{p} - \mathbf{a}$ and $\mathbf{b} - \mathbf{a}$ extended with a z-component of 0.

We also define the vector perpendicular to a vector $\mathbf{v}$ to the left:

$$\operatorname{leftPerp}(\mathbf{v}) = \begin{pmatrix} -\mathbf{v}_y \\ \mathbf{v}_x \end{pmatrix}$$

To create an arc from its endpoints $\mathbf{a}, \mathbf{b}$ and a normal vector $\mathbf{n_a}$ at $\mathbf{a}$, the radius is chosen as

$$r = \frac{\|\mathbf{b} - \mathbf{a}\|_2^2}{2 \operatorname{dot}(\mathbf{b} - \mathbf{a}, \mathbf{n_a})}$$

which places the center at $\mathbf{c} = \mathbf{a} + r\mathbf{n_a}$.

Algorithm 1 Approximate a cubic Bézier curve with biarcs

Require: A cubic Bézier curve with control points $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$

if $\mathbf{P}_1 - \mathbf{P}_0$ and $\mathbf{P}_3 - \mathbf{P}_2$ are short **then**
    return a line segment from $\mathbf{P}_0$ to $\mathbf{P}_3$
end if
if $\mathbf{P}_1 - \mathbf{P}_0$ is short **then**
    $\mathbf{P}_1 \leftarrow \mathbf{P}_2$
end if
if $\mathbf{P}_3 - \mathbf{P}_2$ is short **then**
    $\mathbf{P}_2 \leftarrow \mathbf{P}_1$
end if
$\mathbf{u}_0 \leftarrow \text{normalized}(\mathbf{P}_1 - \mathbf{P}_0)$
$\mathbf{u}_1 \leftarrow \text{normalized}(\mathbf{P}_3 - \mathbf{P}_2)$

if $\mathbf{u}_0$ is parallel to $\mathbf{u}_1$ **then**
    if $\mathbf{P}_2$ is on the line defined by $\mathbf{P}_0$ and $\mathbf{P}_1$ **then**
        return a line segment from $\mathbf{P}_0$ to $\mathbf{P}_3$
    else
        Split curve $\mathbf{B}$ at $t = \frac{1}{2}$ into curves $\mathbf{G}$ and $\mathbf{H}$
        return $\mathbf{G}$ and $\mathbf{H}$ approximated recursively
    end if
end if
$\mathbf{v} \leftarrow \frac{\det(\mathbf{u}_0, \mathbf{u}_1)}{\text{dot}(\mathbf{u}_0, \mathbf{u}_1) - 1}(\mathbf{P}_0 - \mathbf{P}_3)$

$\mathbf{c}_{2,x} = \frac{\mathbf{P}_{0,x} + \mathbf{P}_{3,x} - \mathbf{v}_y}{2}$
$\mathbf{c}_{2,y} = \frac{\mathbf{P}_{0,y} + \mathbf{P}_{3,y} - \mathbf{v}_x}{2}$

$r_2 = ||\mathbf{P}_0 - \mathbf{c}_2||_2$
$\mathbf{j} = \mathbf{c}_2 + r_2 \,\text{normalized}(\mathbf{B}(\frac{1}{2}) - \mathbf{c}_2)$
create arc $\mathbf{A}_0$ from points $\mathbf{P}_0, \mathbf{j}$ and normal $\text{leftPerp}(\mathbf{u}_0)$, clockwise if $\text{isLeftOf}(\mathbf{j}, \mathbf{P}_1, \mathbf{P}_0)$
create arc $\mathbf{A}_1$ from points $\mathbf{P}_3, \mathbf{j}$ and normal $\text{leftPerp}(\mathbf{u}_1)$, clockwise if $\text{isRightOf}(\mathbf{j}, \mathbf{P}_3, \mathbf{P}_2)$
reverse $\mathbf{A}_1$, i.e. swap start and end points and reverse clock direction
$t \leftarrow \arg\max_{t \in [0,1]}(\min_{\mathbf{q} \in \mathbf{A}_0 \cup \mathbf{A}_1} ||\mathbf{B}(t) - \mathbf{q}||_2)$

if $\min_{\mathbf{q} \in \mathbf{A}_0 \cup \mathbf{A}_1} ||\mathbf{B}(t) - \mathbf{q}||_2$ is small **then**
    return arcs $\mathbf{A}_0$ and $\mathbf{A}_1$
end if
Split curve $\mathbf{B}$ at t into curves $\mathbf{G}$ and $\mathbf{H}$
return $\mathbf{G}$ and $\mathbf{H}$ approximated recursively

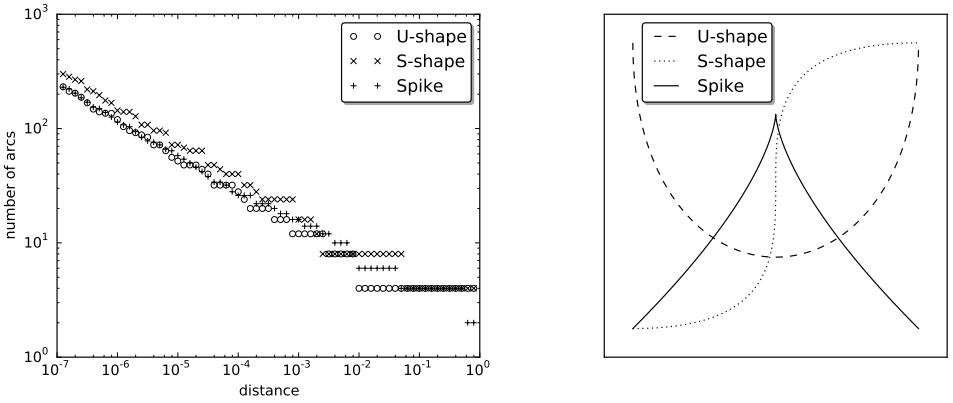


FIGURE 2.8: Log-log plot of number of arcs for different allowed distances (left) produced by three cubic Bézier curves (right).

Numerical experiments (Figure 2.8) show that this algorithm usually produces less than ten arcs with an allowed distance of $\frac{1}{100}$ for arcs with control points on the unit grid. The number of arcs roughly follows $\mathcal{O}(d^{-0.36})$ where d is the estimated deviation distance to the underlying cubic Bézier curve.

Chapter 3

Drawing general polygons with the stencil buffer

3.1 Cover geometry

Now that the vector graphics paths have been decomposed into line segments and circular arcs, we can prepare to send them to GPU memory.

We choose an arbitrary point, for example the starting point of the first curve, and for each curve, we create a triangle formed by the arbitrary point and the start and endpoints of the curve. Drawing those triangles into the stencil buffer will create a rough approximation of the desired shape with straight edges.

To fill in and carve out the missing circular segments, cover geometry for the circular arc segment has to be generated to enable a fragment shader to fill it.

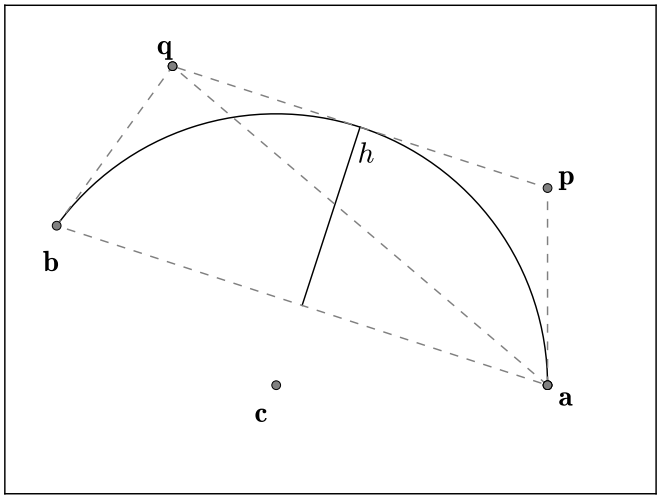


FIGURE 3.1: An arc segment covered by two triangles (dashed, gray).

A possible choice of cover geometry are two triangles forming a trapezoid with base edge length $\|\mathbf{a} - \mathbf{b}\|_2$ and height h of the circular segment with radius r where $\mathbf{a}$ and $\mathbf{b}$ are the points at the base edge and $\mathbf{p}$ and $\mathbf{q}$ the points at the top edge (Figure 3.1).

$$h = r - \left\| \frac{\mathbf{a} + \mathbf{b}}{2} - \mathbf{c} \right\|_2$$

The points $\mathbf{p}$ and $\mathbf{q}$ can be chosen as the intersection of the top edge with the circle tangents $\mathbf{v_a}$ and $\mathbf{v_b}$ at $\mathbf{a}$ and $\mathbf{b}$. A counterclockwise arc from $\mathbf{a}$ to $\mathbf{b}$ is assumed. For clockwise arcs, the signs of the tangents $\mathbf{v_a}$ and $\mathbf{v_b}$ have to be flipped.

$$\begin{aligned}
\mathbf{v_a} &= \mathrm{leftPerp}(\mathbf{a} - \mathbf{c}) \\
\mathbf{v_b} &= \mathrm{leftPerp}(\mathbf{c} - \mathbf{b}) \\
\mathbf{p} &= \mathbf{a} + \frac{h}{\|\mathbf{v_a}\|_2^2 - \frac{\mathrm{dot}(\mathbf{v_a}, \mathbf{b}-\mathbf{a})^2}{\|\mathbf{b}-\mathbf{a}\|_2^2}} \mathbf{v_a} \\
\mathbf{q} &= \mathbf{b} + \frac{h}{\|\mathbf{v_b}\|_2^2 - \frac{\mathrm{dot}(\mathbf{v_b}, \mathbf{b}-\mathbf{a})^2}{\|\mathbf{b}-\mathbf{a}\|_2^2}} \mathbf{v_b}
\end{aligned}$$

3.2 Filling circular segments

A fragment shader which discards pixels outside of a unit circle is given. The formulation in terms of the unit circle saves one variable compared to providing circle center and radius. It also allows for uniform handling of line segments and filled regions by transforming the coordinate system appropriately. This is also possible with the circle/radius formulation but requires large radii to create the appearance of line segments which might create problems for low-end devices with limited floating point precision.

```
void main(){
    // p is of type vec2 and describes the position
    // in the coordinate system of a unit circle
    float radius_squared = dot(p, p);

    if (radius_squared > 1.0) discard;

    gl_FragColor = color;
}
```

3.3 Approximating Bézier curves with line segments

We also implemented a simpler method to compare the stencil buffer arcs against. We approximated the Bézier curves with line segments until the approximation was visually indistinguishable from a smooth curve, which appears to be the case when consecutive segments have a difference in angle of less than 2 degrees.

For S-shaped cubic Bézier curves, there is a possibility that only two segments could be created from the first control point to the inflection point to the last control point, regardless of the rest of the curve. Such segments could have a noticeable deviation from the approximating line segments. To avoid this problem, the curve could be subdivided at the inflection point to create new curvature-monotone curves.

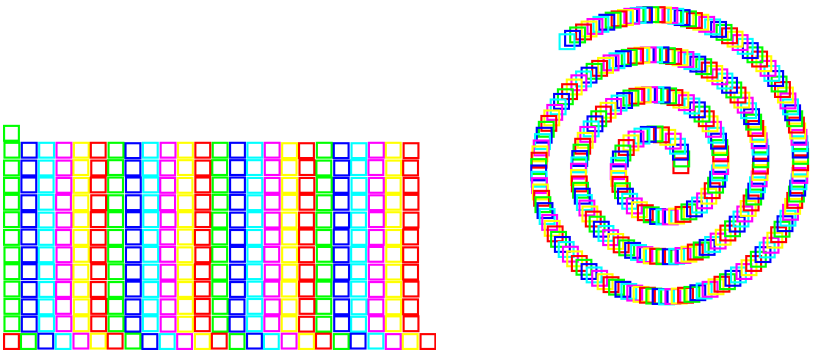
The resulting polygon was then rendered using the stencil buffer.

Other possible subdivision methods include path flattening (Hain et al., 2005), where the distance of a line segments to a Bézier curve is minimized, or by adaptive forward differencing (Lien, Shantz, and Pratt, 1987), which is an efficient method to calculate Bézier curve points.

3.4 Reducing API overhead

In OpenGL, it is preferable for performance reasons to draw as many triangles as possible at once without making any changes to internal OpenGL state. For drawing multiple overlapping polygons, one has to change the stencil state twice for every polygon; once to draw the polygon shape into the stencil buffer and then again to convert the polygon-shaped Boolean inside-outside information from the stencil buffer into color values.

To avoid state changes, the shapes can first be drawn all at once without overlap into an intermediate buffer (Figure 3.2a). In a second step, they can be read from the intermediate buffer to be colored and composed into the final color buffer (Figure 3.2b).



(A) Shapes in intermediate buffer without overlap.

(B) Shapes composed on screen.

FIGURE 3.2: Composing overlapping rectangles with holes.

However, using an intermediate buffer might also be slower since the buffer has to be prepared. Another problem might be that, depending on the graphics API or version, the stencil buffer can not be read directly within a fragment shader. Instead, stencil buffer values can only be used indirectly via stenciling operations. While it is possible to not just stencil, but also color the shapes in the intermediate buffer, composing them on the screen will require a third drawing operation, compared to just two drawing operations when not using an intermediate buffer.

Nevertheless, our tests (Figure 3.3) show an order of magnitude improvement in rendering time when using an intermediate buffer with three draw passes to render a spiral pattern of hollow rectangles with a size of 10 by 10 pixels. It was also observed that the rendering times were more predictable.

The problem of packing the polygon bounding boxes efficiently is related to the two-dimensional bin packing which in general is NP-hard, but

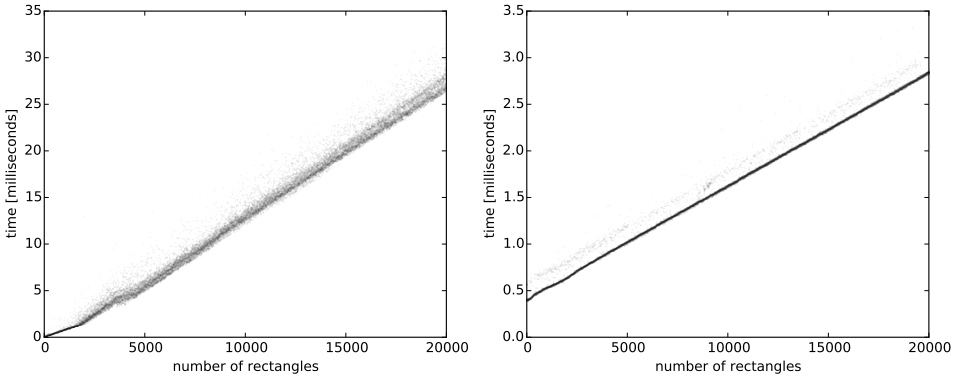


FIGURE 3.3: Time for drawing shapes with stencil state change between consecutive squares (left) and with an intermediate buffer (right).

fortunately many heuristics exist (Jylänki, 2010) to make a solution attainable in reasonable time.

Instead of packing the bounding boxes, the polygons could be packed directly while also allowing for rotation, at the cost of more preprocessing time.

Chapter 4

Drawing general polygons by decomposition

4.1 Overview

A general polygon is x-monotone if any vertical line intersects it in at most two points. Similarly, it is y-monotone if any horizontal line intersects it in at most two points.

Our goal is to decompose an arrangement of circular arcs and line segments so that the decomposed pieces can be rendered efficiently by graphics APIs like OpenGL.

Therefore, we split the polygon horizontally so that all resulting polygons are x-monotone.

Next, we split the x-monotone polygons into pieces which are bounded by vertical line segments to the left and right and by an x-monotone curve at the top and bottom each (Figure 4.1).

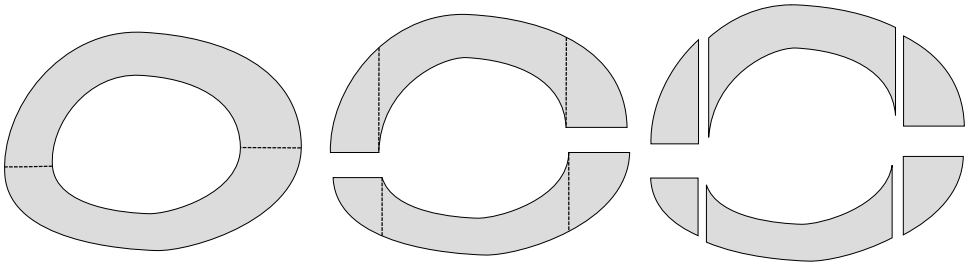


FIGURE 4.1: General polygon with hole (left), split into x-monotone (middle) and x- and y-monotone pieces (right).

A fragment shader can then fill the area between the two curves.

4.2 Preprocessing

First, we split all curves to make them x-monotone (Figure 4.2).

This will at worst triple the number of circular arcs since they can only be split at the leftmost and rightmost point of their circle.

Vertical line segments are excluded from this procedure and have to be handled separately.

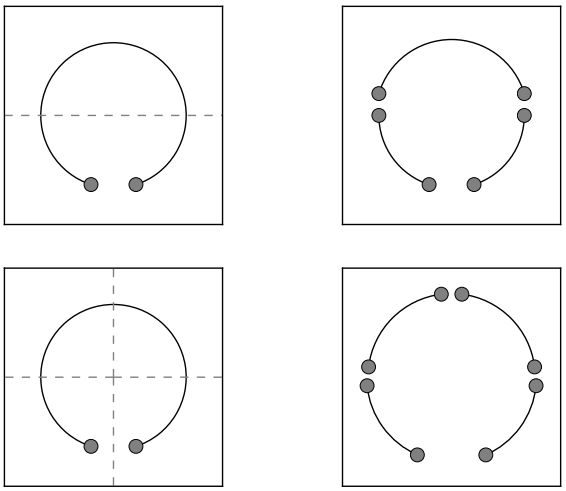


FIGURE 4.2: A circular arc split into x-monotone (top) and x- and y-monotone arcs (bottom).

4.3 Intersections

Second, we find all intersections between each pair of curves and split the curves at their intersections which will give us at most $2k$ additional curves, two for each intersection.

To split a curve at its intersections, the order of the intersection points along the curve has to be found to create new curves with correct start and end points, for example by sorting the points by their x-coordinate, which works because the curves are x-monotone.

Now all curves intersect only in their endpoints.

4.4 Efficient intersection of curves

While we focus on rendering performance instead of preprocessing time, for completeness' sake we still want to point out some notable improvements that can be made for finding intersections between curves.

Two circular arcs that do not have the same center and radius have at most two intersections, which follows because the underlying circles have at most two intersections.

A circle and a line segment can not have more than two intersections either.

As a result, an arrangement of n line segments and circular arcs can have at most $k \in \mathcal{O}(n^2)$ intersections.

A naive algorithm that tests each pair of curves for intersections has an algorithmic complexity of $\mathcal{O}(n^2)$ which is optimal in the worst case, but most of the time, the number of intersections k is much smaller than n^2, in which case we can do better with an output-sensitive algorithm.

(Bentley and Ottmann, 1979) proposed an algorithm for finding line segment intersections with time complexity of $\mathcal{O}((n + k) \log n)$ which is often used in practice, for example in CGAL (Fogel, Halperin, and Wein, 2012), where it is also used for x-monotone curves. A detailed description

of the algorithm, which includes handling of degenerate cases, is contained in (Berg et al., 2008).

(Chazelle and Edelsbrunner, 1988) described an asymptotically faster algorithm with $\mathcal{O}(n + k)$ space complexity and optimal $\mathcal{O}(n \log n + k)$ time complexity, later surpassed by (Balaban, 1995) with optimal $\mathcal{O}(n)$ space complexity, who also stated that the algorithm works with curves, not just line segments.

4.5 Making polygons x-monotone

Third, we want to find horizontal segments to split the polygon into x-monotone ones. For that, we extend a ray horizontally to the left or the right from each vertex $\mathbf{p}$ if they are "pointy" towards the ray direction (Figure 4.3). Alternatively, the implementation can be simplified by extending rays from all vertices at the cost of additional segments in the final result.

We assume that the curves are sorted counterclockwise around the ray origin $\mathbf{p}$.

The tangent end points of the curve before and after the ray, according to counterclockwise sorting order, should both lie to the left or the right of the ray origin $\mathbf{p}$.

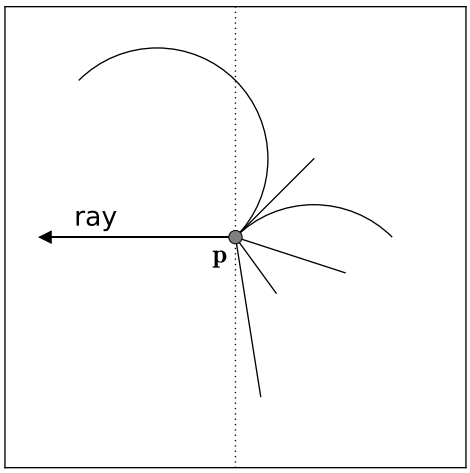


FIGURE 4.3: A point which is considered "pointy" and is therefore origin of a ray.

If the number of intersections of the ray with all curves is even, including intersections at the ray origin, a segment is inserted from the ray origin to the first intersection to indicate where the polygon is to be split to make it x-monotone. The same segment is inserted a second time at the same points, thereby nullifying any potential point-in-shape modifications due to the even-odd fill-rule.

To ensure that shared endpoints between consecutive curves are not counted twice, a tie-breaking rule has to be applied. In general, such degenerate cases can be handled through Simulation of Simplicity (Edelsbrunner and Mücke, 1990), which is a form of symbolic perturbation.

As a result, one might consider the lower endpoint to be on the line segment while the upper endpoint is not. Horizontal segments should be ignored completely.

Circular arcs can be split into x- and y-monotone pieces and be thought of as line segments with respect to endpoint inclusion.

Depending on the method with which the curve intersections in the previous step are determined, it is not necessary to cast rays explicitly, since the information of which curve is to the left and the right of an endpoint has already been found.

4.6 Finding faces

Fourth, we extract the x-monotone polygons from the arrangement of curves.

We assume that at each vertex the incident curves are ordered counterclockwise by angle, starting at the x-axis.

Algorithm 2 Face extraction algorithm

Require: Curves $\mathcal{C}$ for which each pair only intersects in their endpoints or in all points
repeat
  $v_0 \leftarrow$ rightmost endpoint
  $v \leftarrow v_0$
  $C \leftarrow$ first (by sorting order) curve incident in v
  Start new face F
  repeat
    $w \leftarrow$ endpoint of curve C that is not v
    $D \leftarrow$ previous (by sorting order) curve before C incident in w
    append curve C to face F
    remove curve C from curves $\mathcal{C}$
    $C \leftarrow D$
    $v \leftarrow w$
  until $w = v_0$
  Add face F to finished faces
until $|\mathcal{C}| = 0$
Return finished faces

Note the unfilled triangle-shaped region in Figure 4.4, as would be expected from applying the even-odd rule.

In practice, one often builds a doubly connected edge list (DCEL), which can be found in $\mathcal{O}((n+k)\log n)$ time (Fogel, Halperin, and Wein, 2012).

4.7 Generating pseudo-trapezoids

Fifth, we want to split the x-monotone polygons into regions bounded by a vertical line segment to the left and right each and by an x-monotone curve at the top and bottom. We denote these regions as pseudo-trapezoids.

This is a simple task of starting at the leftmost or rightmost vertex and simultaneously walking along the lower and upper curves while casting rays from the endpoints towards the other curve. If a ray intersects, the

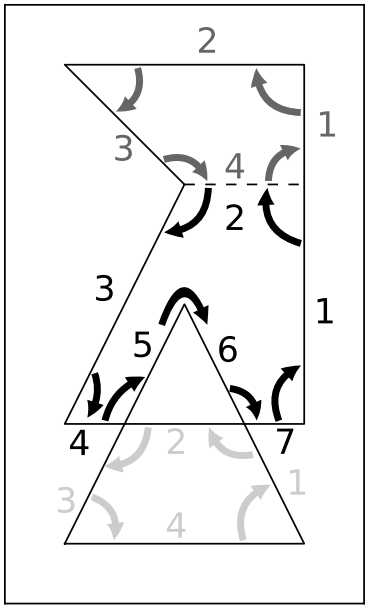


FIGURE 4.4: Order in which curves are traversed to form three faces. The dashed line indicates two separating segments.

intersected curve is split at the intersection and the other curve is stepped along further.

Since we previously started at the rightmost endpoint, we do so again and also assume that a curve **C** starts at its right endpoint **a** and ends at its left endpoint **b** in Algorithm 3.

4.8 Filling between curves

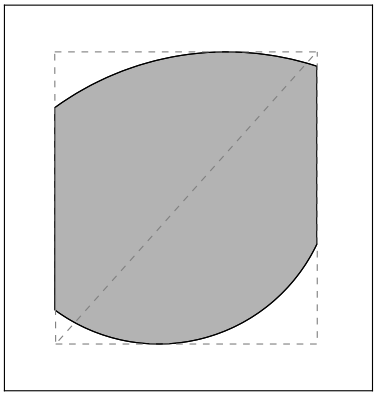


FIGURE 4.5: A pseudo-trapezoid with filled area between upper and lower boundary. The outline of the covering triangles is dashed in gray.

Sixth and last, we cover pseudo-trapezoids with two triangles. The triangle vertices have the information about the upper and lower bounding

Algorithm 3 Pseudo-trapezoid algorithm

Require: X-monotone polygon
$V \leftarrow$ rightmost vertices
$W \leftarrow$ leftmost vertices
$\mathbf{v}_0 \leftarrow \arg\min_{\mathbf{v} \in V} \mathbf{v}_y$
$\mathbf{v}_1 \leftarrow \arg\max_{\mathbf{v} \in V} \mathbf{v}_y$
$\mathbf{C}_0 \leftarrow$ last curve in counterclockwise order with starting point v_0
$\mathbf{C}_1 \leftarrow$ first curve in counterclockwise order with starting point v_1
repeat
  $\mathbf{a}_0$ and $\mathbf{b}_0$ are the start and endpoints of curve $\mathbf{C}_0$
  $\mathbf{a}_1$ and $\mathbf{b}_1$ are the start and endpoints of curve $\mathbf{C}_1$
  if $\mathbf{b}_{0,x} = \mathbf{b}_{1,x}$ **then**
    Create pseudo-trapezoid with curves $\mathbf{C}_0$ and $\mathbf{C}_1$
    $\mathbf{C}_0 \leftarrow$ other curve (that is not $\mathbf{C}_0$) connected to $\mathbf{b}_0$
    $\mathbf{C}_1 \leftarrow$ other curve connected to $\mathbf{b}_1$
  else
    if vertical ray upwards starting at $\mathbf{b}_0$ intersects $\mathbf{C}_1$ in $\mathbf{s}$ **then**
      Create pseudo-trapezoid with subcurve of $\mathbf{C}_1$ from $\mathbf{a}_1$ to $\mathbf{s}$ and curve $\mathbf{C}_0$
      $\mathbf{C}_1 \leftarrow$ subcurve of $\mathbf{C}_1$ from $\mathbf{s}$ to $\mathbf{b}_1$
      $\mathbf{C}_0 \leftarrow$ other curve connected to $\mathbf{b}_0$
    else
      Create pseudo-trapezoid with subcurve of $\mathbf{C}_0$ from $\mathbf{a}_0$ to $\mathbf{s}$ and curve $\mathbf{C}_1$
      vertical ray downwards starting at $\mathbf{b}_1$ intersects $\mathbf{C}_0$ in $\mathbf{s}$
      $\mathbf{C}_0 \leftarrow$ subcurve of $\mathbf{C}_0$ from $\mathbf{s}$ to $\mathbf{b}_0$
      $\mathbf{C}_1 \leftarrow$ other curve connected to $\mathbf{b}_1$
    end if
  end if
until $\mathbf{b}_0 \in W$ and $\mathbf{b}_1 =\in W$
return created pseudo-trapezoids

curve as well as their position in space attached which is sufficient for rendering them on the GPU. For each pixel, we compute the y-position of the lower and upper bounding curve. If the pixel y-position is between them, it is rendered, otherwise it is left transparent.

Note that the triangles may extend further than the curve endpoints (Figure 4.5) because they should fully cover the curves. This will account for some wasted rasterization effort which could be reduced by splitting the pseudo-trapezoids further.

A GLSL fragment shader is given to fill the area between two circular arcs or line segments. The `type0` and `type1` variables have a value of 0 in case of line segments, 1 in case of clockwise circular arcs and 2 in case of counterclockwise circular arcs.

```
float transparency(
    vec2 c0, float r0, int type0,
    vec2 c1, float r1, int type1, vec2 p
){
    vec2 p0 = p - c0, p1 = p - c1;
    // Bounds in case of clockwise arc
    float y0 = sqrt(r0*r0 - p0.x*p0.x);
    float y1 = sqrt(r1*r1 - p1.x*p1.x);
    // Bounds in case of lines and counterclockwise arcs
    if (type0 == 0) y0 = -y0;
    if (type1 == 0) y1 = -y1;
    if (type0 == 2) y0 = p0.y;
    if (type1 == 2) y1 = p1.y;
    // Visible if within bounds
    return y0 <= p0.y && p1.y <= y1 ? 1.0 : 0.0;
}
```

Chapter 5

Results and future work

5.1 Results

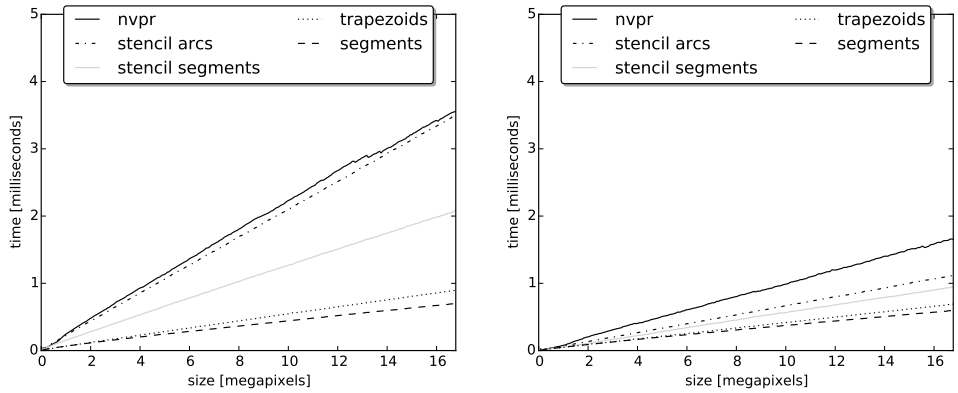


FIGURE 5.1: Milliseconds per framebuffer size for vector graphics with high (left) and low (right) winding. Benchmarks were performed on a Lenovo z50-70 with NVIDIA GeForce 840M GPU.

In our benchmark (Figure 5.1), we considered five methods.

NV Path Rendering, which uses the stencil buffer and evaluates Bézier curves on the GPU, ranks fifth, closely followed by the method described in Chapter 3, which used the stencil buffer to render polygons with circular arc boundaries. Arcs were approximated to have a distance of one tenth of a pixel to the approximating Bézier curve.

On the third place by some margin is the method which approximates Bézier curves with line segments and fills the resulting polygon using the stencil buffer, also described in Chapter 3.

Second was our method described in Chapter 4 which decomposed paths into pseudo-trapezoids that can be rendered directly without a stencil buffer. There is a noticeable gap to the stencil buffer-based renderers. A second benchmark was performed for a path with a lower number of windings than the first tested vector graphics (Figure 5.2), with the result that the gaps between different methods became smaller.

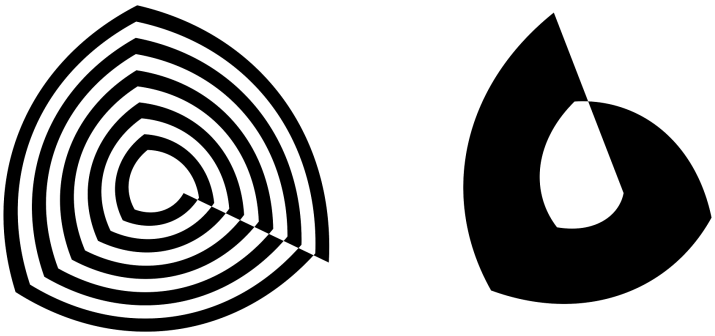
On first place, we triangulated the polygon from the method on third place with the GLUtesselator library and rendered the triangles again without a stencil buffer.

All rendering times scale almost linearly with the pixel size of the rendered vector graphics which indicates that we are indeed limited by the rate at which the GPU can process pixels.

In the first benchmark, the graphics with high winding covers roughly 22.5 % of the framebuffer compared to 20.5 % for the graphics in the second benchmark. The stencil buffer-based methods produce triangles with an area of 8 times and 1.4 times as many pixels respectively, excluding cover geometry.

Only pure rendering time was measured after geometry was uploaded to GPU memory.

Benchmark source code is available at: `https://github.com/99991/arcproximator`



(A) High winding (B) Low winding

FIGURE 5.2: Vector graphics with different winding.

5.2 Conclusion

The gap between the stencil buffer-based and direct renderers can be explained by the number of pixels that are drawn multiple times into the stencil buffer and is confirmed by the second benchmark where the difference is less pronounced due to lower overdraw.

Similarly, the difference between the method which approximates Bézier curves with line segments and the method which approximates with circular arcs can be explained by the fact that slightly more pixels are drawn for the cover geometry of the pseudo-trapezoids which are then discarded, although the number of vertices is slightly lower, which does not appear to be a limiting factor.

To ensure that the different fragment shader complexity was not the reason we also tested fragment shaders that write color values regardless of shape-inclusion information. We found that there appears to be a fast path for shaders that do not require the `discard` instruction like the method on third place. Otherwise, the performance differences were barely measurable.

Concluding one can say that methods which decompose the curves into segments are superior in performance and simplicity, both for stencil buffer-based methods and direct methods.

5.3 Future work

In this work we did not consider preprocessing time since usually vector graphics are displayed for multiple seconds, over which the costs will be amortized.

However, in low latency environments, for example in a drawing application, response times should be low. NV Path Rendering requires virtually no preprocessing time, but is slightly slower than direct methods once the geometry is computed.

A possible solution could be hybrid methods that use a stencil buffer-based approach until the triangulation becomes available, but would require the combined complexity of both methods. Another interesting research topic might be interactive modifications of the triangulation as the generating bounding segments change while they are edited.

Bibliography

Abel, Niels Henrik (1824). "Mémoire sur les équations algébriques, ou l'on démontre l'impossibilité de la résolution de l'équation générale du cinquième degré". In: 28–33. URL: http://www.abelprisen.no/nedlastning/verker/oeuvres_1881_del1/oeuvres_completes_de_abel_nouv_ed_1_kap03_opt.pdf.

Balaban, Ivan J. (1995). "An optimal algorithm for finding segments intersections". In: *Proceedings of the eleventh annual symposium on Computational geometry*, pp. 211–219.

Bentley, Jon L. and Thomas A. Ottmann (1979). "Algorithms for Reporting and Counting Geometric Intersections". In: *IEEE Transactions on computers* c-28.9, pp. 643–647.

Berg, Mark de et al. (2008). *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, pp. 20–29. ISBN: 3540779736, 9783540779735.

Chazelle, Bernard and Herbert Edelsbrunner (1988). "An Optimal Algorithm for Intersecting Line Segments in the Plane". In: *29th Annual Symposium on Foundations of Computer Science*, pp. 590–600.

Chlumský, Viktor (2015). "Shape decomposition for multi-channel distance fields". In: URL: https://dspace.cvut.cz/handle/10467/62770.

Edelsbrunner, Herbert and Ernst Peter Mücke (1990). "Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms". In: *ACM Transactions on Graphics* 9, pp. 66–104. URL: http://arxiv.org/pdf/math/9410209.pdf.

Esfahbod, Behdad (2011). *GLyphy: high-quality glyph rendering using OpenGL ES2 shaders*. URL: http://glyphy.org/.

Fogel, Efi, Dan Halperin, and Ron Wein (2012). *CGAL Arrangements and Their Applications*. Springer-Verlag, pp. 48–49, 55. ISBN: 18666795, 9783642172823.

Ganacim, F. et al. (2014). "Massively-Parallel Vector Graphics". In: *ACM Transactions on Graphics (Proceedings of the ACM SIGGRAPH Asia 2014)* 36.6, p. 229. URL: http://w3.impa.br/~diego/projects/GanEtAl14/.

Green, Chris (2007). "Improved Alpha-Tested Magnification for Vector Textures and Special Effect". In: *SIGGRAPH Course on Advanced Real-Time Rendering in 3D Graphics and Games*. URL: http://www.valvesoftware.com/publications/2007/SIGGRAPH2007_AlphaTestedMagnification.pdf.

Hain, Thomas F. et al. (2005).
"Fast, Precise Flattening of Cubic BéZier Path and Offset Curves".
In: *Comput. Graph.* 29.5, pp. 656–666. ISSN: 0097-8493.
DOI: 10.1016/j.cag.2005.08.002.
URL: http://dx.doi.org/10.1016/j.cag.2005.08.002.

Jylänki, Jukka (2010). *A thousand ways to pack the bin - a practical approach to two-dimensional rectangle bin packing*.
URL: http://clb.demon.fi/files/RectangleBinPack.pdf.

Kilgard, Mark and Jeff Bolz (2012). "GPU-accelerated Path Rendering".
In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2012)* 31.6, to appear.

Kilgard, M.J. (2014). "Path rendering with path clipping". Pat.
US Patent 8,698,837.
URL: https://www.google.com/patents/US8698837.

— (2016). "Path rendering by covering the path based on a generated stencil buffer". Pat. US Patent 9,311,738.
URL: https://www.google.com/patents/US9311738.

Lien, Sheue-Ling, Michael Shantz, and Vaughan Pratt (1987).
"Adaptive Forward Differencing for Rendering Curves and Surfaces".
In: *SIGGRAPH Comput. Graph.* 21.4, pp. 111–118. ISSN: 0097-8930.
DOI: 10.1145/37402.37416.
URL: http://doi.acm.org/10.1145/37402.37416.

Loop, Charles and Jim Blinn (2005). "Resolution Independent Curve Rendering using Programmable Graphics Hardware".
In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24, pp. 1000–1008. URL: http://research.microsoft.com/en-us/um/people/cloop/LoopBlinn05.pdf.

— (2007). "Rendering Vector Art on the GPU (GPU Gems 3)". In:
URL: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch25.html.

Michail, A.A., D.B. Teitlebaum, and B.C. Furtwangler (2014).
"Arc spline GPU rasterization for cubic Bezier drawing". Pat.
US Patent 8,624,899.
URL: https://www.google.com/patents/US8624899.

Nehab, Diego and Hugues Hoppe (2008).
"Random-Access Rendering of General Vector Graphics".
In: *ACM transactions on graphics* 27.
URL: http://hhoppe.com/ravg.pdf.

Neider, Jackie, Tom Davis, and Mason Woo (1997). *OpenGL programming guide: the official guide to learning OpenGL, Version 1.1*. 2nd.
Addison-Wesley. ISBN: 0201632748, 9780201632743.

OpenGL 4.5 API Specification (2016). URL: https://www.opengl.org/registry/doc/glspec45.core.pdf.

Ray, Nicolas et al. (2005). "Vector Texture Maps on the GPU".
In: *Technical Report ALICE-TR-05-003*. URL: http://alice.loria.fr/publications/papers/2005/VTM/vtm.pdf.

Safonova, Alla and Jarek Rossignac (2003).
"Compressed Piecewise-Circular Approximations of 3D Curves".
In: *Computer-Aided Design* 35.6, pp. 533–547.

Yhann, S.R. (2011).
"Rendering rational quadratic Bézier curves on a GPU". Pat.

US Patent 7,868,887.
URL: https://www.google.com/patents/US7868887.

Yhann, S.R. and P. Choudhury (2011a).
"Efficient data packaging for rendering bézier curves on a GPU". Pat.
US Patent 7,928,984.
URL: https://www.google.com/patents/US7928984.

— (2011b).
"Rendering cubic Bézier curves as quadratic curves using a GPU". Pat.
US Patent 8,068,106.
URL: https://www.google.com/patents/US8068106.

Šír, Zbyněk et al. (2006). "Approximating curves and their offsets using biarcs and Pythagorean hodograph quintics".
In: *Computer-Aided Design* 38.6.