Texture-based Rendering of Vector-based Shapes

using Graphics Hardware and the Web

J.B. van Velzen

Department of Computer Graphics and Visualization
Delft University of Technology, Faculty EEMSC
Master Thesis

TU Delft

Texture-based Rendering of Vector-based Shapes

using Graphics Hardware and the Web

by

J.B. van Velzen

to obtain the degree of Master of Science
at the Delft University of Technology,
after completing EEMCS's *Computer Science - Data Science & Technology* track.
To be defended publicly on Thursday March 15, 2018 at 10:00 AM.

"The single biggest problem we face is that of visualization."
— Richard P. Feynman (1918–1988) [3]

Student number:	1509411	
Project duration:	November 1, 2016 – March 15, 2018	
Thesis committee:	Prof. dr. E. Eisemann	TU Delft, supervisor
	Dr. F. Baart	Deltares, supervisor
	Prof. dr. ir. M. Verlaan	TU Delft, Deltares

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft Deltares

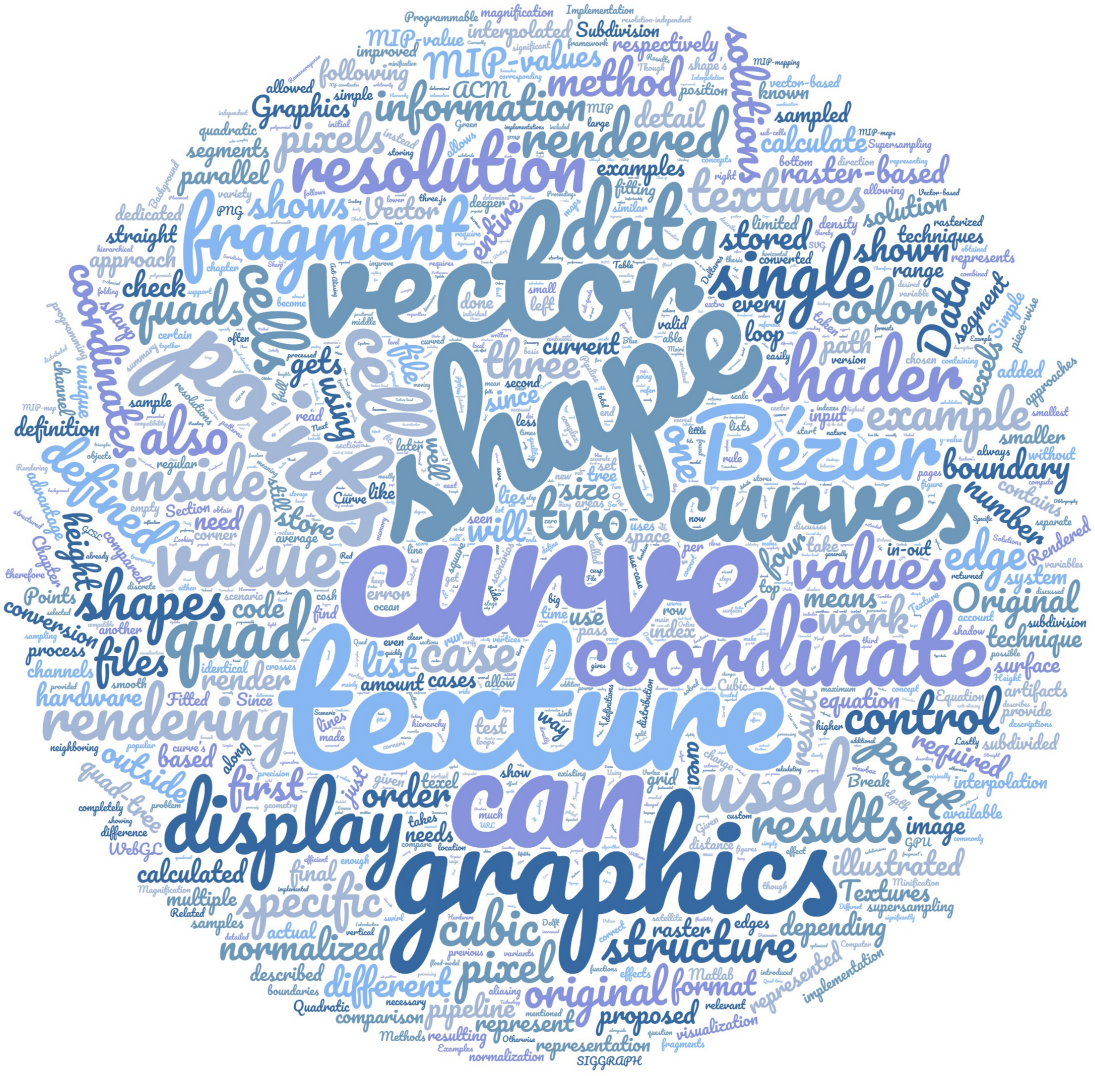
Abstract

Vector-based graphics are not directly compatible with the raster-based structure of dedicated graphics hardware. They suffer from a locality problem where, in order to check if a specific display pixel lies inside or outside the shape, the entire shape needs to be taken into account. This negates the advantage that the graphics pipeline's parallel rendering structure offers. Current solutions rasterize vector graphics into a discrete raster-based texture in an extra render pass, every time the required texture resolution changes.

This work presents an approach to store vector-based shapes into discrete raster-based textures, optimized for parallel rendering. This is done by encoding the shape as piece-wise cubic Bézier curves, structured in a quad-tree. Additional data is added to remove aliasing effects in case of minification. These textures are then rendered by a custom fragment shader.

This work discusses the implementation of both the serialization and deserialization steps, as well as rendering regular vector shapes with a comparable quality.

Contents



1

Introduction

Devices these days come in an ever-increasing variety of display sizes and resolutions. This means that when providing graphics for a targeted group of devices, a large number of different resolutions need to be dealt with. Solutions for this problem require compatibility with dedicated graphics hardware to keep the rendering process fast, as well as resolution-independence to handle all variations in displays. Graphics that are compatible with the graphics pipeline have a discrete, raster-based structure. These are in fundamental contrast with resolution-independent graphics, which are structured as continuous vector-based shape descriptions and can, therefore, be rendered at any desired resolution. Though, these are not raster-based, therefore having very limited compatibility with the graphics pipeline.

Raster-based graphics

When raster-based graphics are used as textures for virtual objects, these texture pixels are described as texels. The color information in these textures is uniformly distributed over its surface. This has the advantage that calculating a color for a specific coordinate on that surface only involves a quick interpolation between the four nearest texels. This fits very well in the graphics pipeline of most dedicated graphics hardware, which performs parallel rendering of raster-based graphics to significantly speed up the rendering process.

A disadvantage of having all color information stored in a discretized grid of samples is that the precision of these textures is limited by their resolution. To be able to render a texture on a display without having to scale it up, its resolution needs to be at least as big as that of the display. When the texture's resolution supersedes that of the display, it can be easily sampled down to the required size, albeit with potential under-sampling effects. When the texture's resolution is smaller than that of the display, the texture needs to be magnified. This often results in blurry and/or imprecise rendered graphics, depending on the magnification interpolation. Therefore, in order to get precise results on all targeted display resolutions, raster-based graphics need to be provided for each of those resolutions, matching its dimensions. This solution is clearly not sustainable for all the variety in display resolution that the future might bring.

Vector-based graphics

In contrast with raster graphics, vector graphics are defined as continuous and resolution-independent shape descriptions. These shapes are often defined as piece-wise path segments that represent the shape's boundary, where each segment can potentially affect any part of the texture space that makes up the final graphic. This makes it difficult to render the entire shape in parallel because when rendering a color for a specific display pixel, the entire shape needs to be taken into account.

In order for vector-based graphics to take advantage of the graphics pipeline, they first need to be rendered to a raster-based representation. Even though this pre-process provides a precise representation for any required resolution, it takes up valuable render time that could have been spent elsewhere.

Use-case

A use-case where extensive vector-based data is used in the context of a 3D visualization is in overland flood simulations. The bodies of water are defined as vector shapes, allowing detailed rendering from both close-up and from a distance. Furthermore, these visualizations need to be able to run in a modern web-browser, to make it easily shareable, while still being able to quickly update the current flood data for all its viewers.

This work uses this use-case at certain points in the implementation phase, as it provided a real-world example of the proposed technique. The extra requirements that it brings have a minimal side-effect on the final implementation, while also aiding in the selection of specific frameworks and languages to implement a working prototype in.

Main focus

By taking advantage of the strengths of both raster- and vector-graphics, we are able to encode vector-based shapes into raster textures that are then read and rendered by dedicated graphics hardware. This answers the following research question:

"How can we store vector shape descriptions in such a way, that dedicated graphics hardware can efficiently render them in a single, parallel pass at any given pixel resolution?"

Structure

Chapter 2 will introduce background information on the techniques that are used to achieve the desired result. Chapter 3 goes into detail about the methods that were used and their mathematical origins. After this, in Chapter 4, the implementation of this method and some precautions are discussed. The final results of this method and implementation are then shown in Chapter 5, along with a specific use-case. Finally, this work is respectively discussed and concluded in Chapter 6 and 7.

2

Background and Related work

This work assumes a certain amount of background knowledge. First, the Programmable Graphics Pipeline will be discussed, as well as the concepts of Shaders and Textures. Next, we will present concepts that are relevant to rendering graphics on multiple resolutions. Examples of these rendering concepts are Minification and Magnification. Lastly, we discuss related approaches to define and render curve- and vector information to a display.

2.1. Programmable Graphics Pipeline

Dedicated Graphics Hardware, or the GPU (Graphics Processing Unit), is used to achieve a significant increase in render performance. It uses a structure called the Programmable Graphics Pipeline, in which data is transformed in one step at the time. The advantage is that most steps can run on all data in parallel, resulting in the significant increase in performance. Some of these steps can run custom code, hence the term *Programmable* Graphics Pipeline. These pieces of custom code are called *Shaders* and their location in the pipeline can be seen in **bold** on the bottom of Figure 2.1. It shows the path that data takes, as well as the places where it gets transformed, before it finally ends up as colors in the frame buffer, ready to be put on a display.

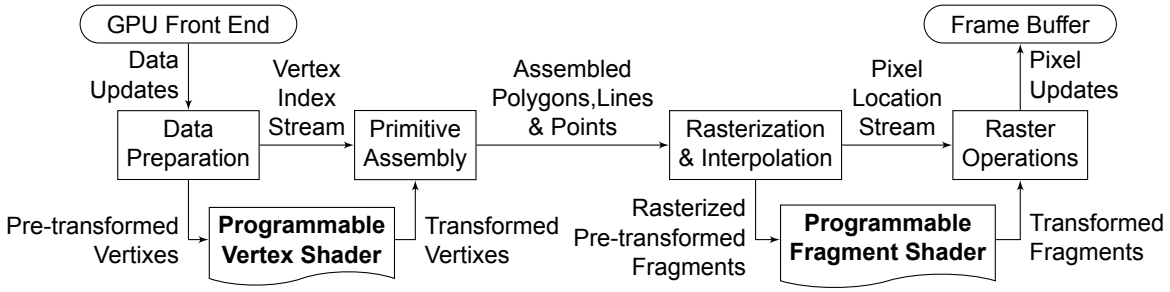


Figure 2.1: The Programmable Graphics Pipeline

2.1.1. Textures

Textures are the main data source in shader programming. A texture can be seen as an image that is stretched over the geometry of an object, like a table-cloth over a table. Since images are based on a regular grid, interpolation between texture-pixels (texels) is relatively straightforward, since the texel data is distributed as a regular grid over the entirety of the texture. All that is needed to obtain a color value for a specific texture coordinate, is a bi-linear interpolation between the nearest texels. Current-day graphics hardware has a number of optimizations for these raster-based textures, like pre-loading the values for a group of neighboring texels (for example 16x16 or 32x32), so their values can be quickly looked up and interpolated between neighbors.

2.1.2. Vertex shader

The vertex shader runs early in the process and is used to transform the objects that need to be rendered to the display. These objects are defined by vertices, points in 2D or 3D space, that can all be processed in parallel by the GPU. The vertex shader generally only knows about the vertex it is run for, although information on neighboring vertices can be obtained as well. This allows the storage of information on these vertices, which is later interpolated on the surfaces that are also defined by the objects that need to be rendered. The vertex shader ends with defining a position in the 2D or 3D space that is projected by the virtual camera. This position can be modified, allowing for dynamic or moving surfaces.

2.1.3. Fragment shader

Once all vertices have been processed, the scene gets rasterized into so-called *fragments*. A display pixel can have multiple fragments if there are overlapping surfaces. These are later combined based on their distance to the camera and their opacity. These fragments are also processed in parallel, with the information that is interpolated between the data that was set for each of the triangle's vertices. The main information that is used in this work's fragment shader is the interpolated UV- or texture-coordinate. This is the location on the texture that lies exactly in the middle of the display pixel. The fragment shader takes this interpolated information and calculates a color value, representing the specific UV-coordinate that it is run for.

2.2. Vector graphics definition

Compared to the de facto raster-based graphics that are commonly used for textures in the graphics pipeline, vector graphics differ in a fundamental way. Instead of a regular grid, or raster, of color information, Vector graphics are often stored in human-readable plain-text files, defining coordinates and boundary path segments. This has the advantage that vector formats can be searched, indexed, scripted and compressed. They are generally smaller in size, compared to visually identical raster-based variants, because only the shape's boundary-information is stored as well as some color information. This does bring the downside that there is limited support for color gradients and highly detailed graphics inside the shapes themselves.

2.2.1. Curve definition

Current-day solutions, nearly always define shapes by describing their outer edge in a series of piece-wise splines or curves. While moving along this curve, the area left of the curve is most often defined to be outside of the shape, where the right area is on the inside of the shape. These curves can be described as continuous polynomials that, given a certain input value t between 0 and 1, return a 2D or 3D coordinate of the position that the curve is at that fraction t. Important for piecewise curves is that, if you want to preserve smooth transitions between separate pieces, some continuity levels needs to be preserved. These levels guarantee that both connecting polynomials end and start at the same control point, and have the same derivative at these points, thereby ensuring a smooth transition between adjacent curves.

Curve polynomials can be of different degrees, depending on their required flexibility. The most useful polynomial for this work is a third power polynomial, also known as a cubic. This spline-type closely resembles a natural flat spline, a flexible curve with fixed endpoints, that can be shaped with great flexibility.

The most commonly used cubic curve description is known as the cubic Bézier curve. Bézier curves were made popular by Pierre Bézier, who was working for the automotive company Renault at the time. By basing his idea on the Bernstein polynomial [1] and his study in curves lines for aerodynamic shapes, he laid the groundwork for the shape definitions of today. Other options would have been the extension of Bézier curves known as Nonuniform rational B-splines (NURBS), but they lack the property of having fixed endpoints. Furthermore, the 2^{nd} degree quadratic Bézier curve definition is widely used in Truetype fonts and 3^{rd} degree cubic Bézier curves are used for the popular Adobe PostScript and SVG standards.

Bézier curve definition

The concept of Bézier curves is fairly straightforward and can be defined recursively to accommodate for as many control points as are required. The number of control points thereby also defines the order of a specific curve, offset by one. Examples of this are the 1^{st}, 2^{nd} and 3^{rd} order curves that have 2, 3 and 4 control points respectively. Where the 1^{st} order curve is a simple straight line, the 2^{nd} and 3^{rd} order are more commonly known as *Quadratic*- and *Cubic*-Bézier curves, as shown in Table 2.1.

Order	Control Points	Common Name
1	2	Straight line
2	3	Quadratic Bézier curve
3	4	Cubic Bézier curve
4	...	
...		

Table 2.1: Bézier curve orders

A Bézier curve always starts at the first control point and ends up in the last control point. Along the curve, it's position $B(t)$ can be calculated from a single parameter, most commonly called t, with values between 0 and 1. For first order curves, with the two control points P_0 and P_1, this is a simple linear interpolation, shown in Equation 2.1.

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

From there on, every n^{th}-order Bézier curve is simply defined as the recursive combination of two $(n-1)^{th}$ order Bézier curves with the first n and last n control points. (Note here that the n^{th}-order curve has $(n+1)$ control points.) This is shown in Equation 2.2 and expanded for the Quadratic and Cubic Bézier variants in Equation 2.3 and 2.4, respectively. Note that t is still a value between 0 and 1.

General
$$B_{P_0 \to P_{n+1}}(t) = (1-t)B_{P_0 \dots P_n}(t) + tB_{P_1 \dots P_{n+1}}(t) \tag{2.2}$$

Quadratic
$$\begin{aligned} B_{P_0P_1P_2}(t) &= (1-t)B_{P_0P_1}(t) + tB_{P_1P_2}(t) \\ &= (1-t)^2P_0 + 2(1-t)tP_1 + t^2P_2 \end{aligned} \tag{2.3}$$

Cubic
$$\begin{aligned} B_{P_0P_1P_2P_3}(t) &= (1-t)B_{P_0P_1P_2} + tB_{P_1P_2P_3} \\ &= (1-t)^3P_0 + 3(1-t)^2tP_1 + 3(1-t)t^2P_2 + t^3P_3 \end{aligned} \tag{2.4}$$

Figure 2.2 shows this recursive structure for Bézier curves of the first four orders. The bold line is the final curve at its current interpolation step. The thinner lines represent the interpolation of the recursive combinations. The control points are the big light-gray circles, annotated with P_0 to P_4.

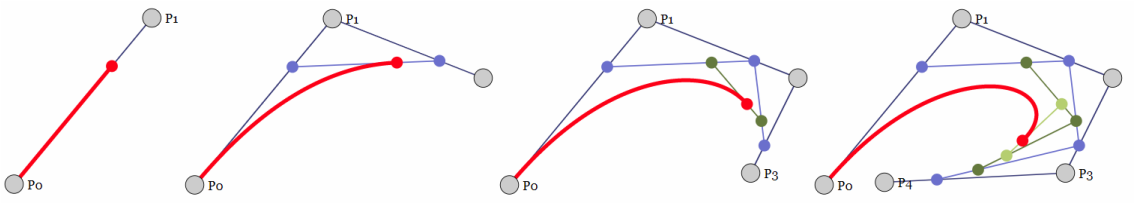
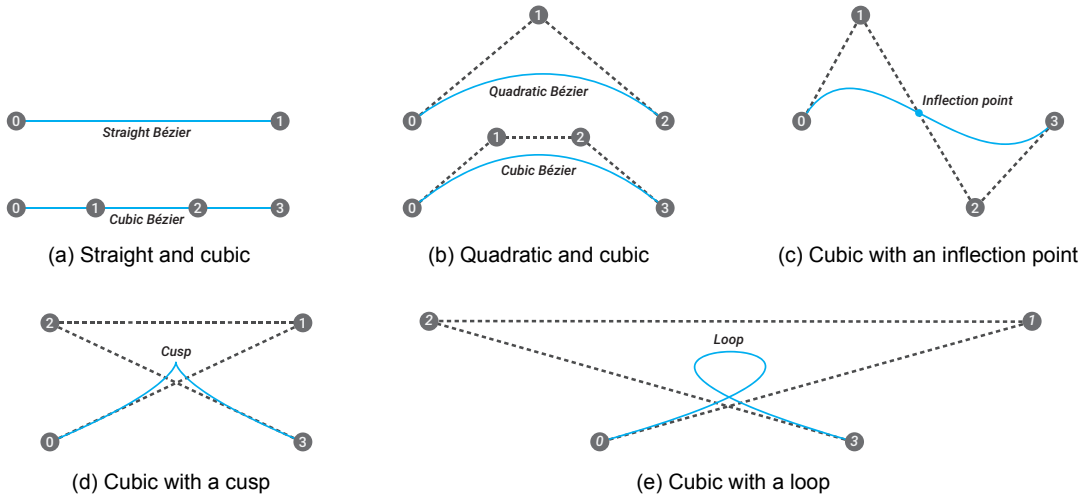


Figure 2.2: Interpolation of different Bézier curves of order 1, 2, 3 and 4 (l.t.r.), with $t = 0.71$.
Original (animated) version by Davies [5].

Given the range of available Bézier orders, the 3-rd order cubic Bézier curve is a clear winner. With its four control points, P_0 to P_3, it is able to represent a wide variety of different curves, from 1-st and 2-nd order Bézier curves to more complex curves with inflection points, cusps, and loops. Inflection points are points in the curve where the shape goes from concave to convex. If this method would have limited itself to just quadratic Bézier curves, there would have been two separate curves required to describe this shape. Cusps are folds in the curve's shape where a loop is just about to form. All of these cases are illustrated in Figure 2.3. The first two figures also illustrate how the control points of a cubic Bézier curve can be set to mimic those lower-order Bézier curves.



(a) Straight and cubic (b) Quadratic and cubic (c) Cubic with an inflection point

(d) Cubic with a cusp (e) Cubic with a loop

Figure 2.3: Cubic Bézier curves representing different order Bézier curves

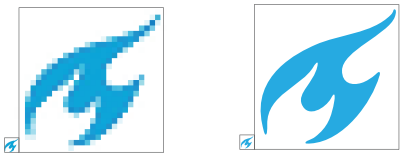
2.3. Scaling raster- and vector-based graphics

Raster-based graphics are limited to a discrete resolution. This means that it can only be perfectly rendered to a display when the resolution in display pixels is the same as the resolution of the texels in the graphic itself. In this case, the texels can be mapped one-to-one to their corresponding display pixel. This is not always the case. Textures can be required to stretch or shrink, depending on their desired size on a display, relative to their own size. The concepts of stretching and shrinking of textures are known as *Magnification* and *Minification*, respectively. Both of these come with visual artifacts that can distort the final rendering, and thereby distract the viewer from the actual visualization.

2.3.1. Magnification

Magnification occurs when a provided raster-texture does not have enough resolution to provide accurate information for all display pixels. In these cases, new data is often interpolated by performing a bi-linear interpolation between the nearest texels. This does have the side effect that these graphics become blurry when enlarged too much. A solution for this would be to define these sharp changes as shapes in a vector format. *Vector graphics* allow for resolution-independent rendering of graphics, meaning that sharp edges will remain sharp, no matter how close the camera gets, since the boundary of a shape is precisely known. This is possible because the edge is defined by a parametric curve, where the shape of the curve can be calculated with almost-infinite precision. Figure 2.4 shows the difference between the magnification of raster- vs. vector graphics. The raster image had an original resolution of 30 by 30 pixels.

.

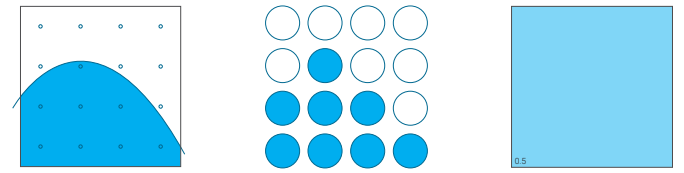


(a) Raster graphics (b) Vector graphics

Figure 2.4: 10x Magnification of both raster- and vector graphics

Supersampling

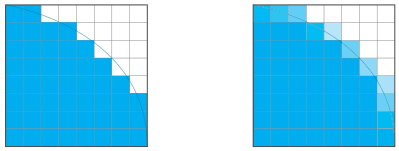
The in-out test results in a binary value for each of the tested texture coordinates. This means that the edge is a hard transition between display pixels that apparently lie completely inside the shape and neighboring display pixels who's texture coordinate lies completely outside of the shape. This can be improved by computing the in-out test for multiple samples inside the same display pixel fragment, and then averaging over all these values. This is known as Supersampling, and gives a much smoother and representative transition between the inside and outside of a shape, especially on curved edges. This is very similar to Full-Scene Anti-Aliasing (FSAA), where the entire scene would be rendered in a resolution that was twice or quadruple the size of the original display and then down-sampled to get the final result. However, this is done for the entire texture and it might not be necessary for every display pixel. It is more efficient to only apply supersampling on a per-fragment basis in the fragment shader, only for the fragments that require this extra sampling. This work, for example, deals with solid color shapes and would only need supersampling in those fragments that lie on the edge of the shape. Figure 2.5 shows an example of supersampling for an example display pixel on the boundary of a shape. The sample method is a 4 by 4 uniform grid, but many other sample patterns are available, like the *Poisson disc* pattern and the *High-resolution antialiasing (HRAA)*, or *Quincunx* pattern [20]. On the left, the sample locations are shown as small circles, with their sampled values in the middle grid. White and blue values mean outside and inside the shape respectively. The square on the right shows the averaged value for this display pixel.



(a) Sample locations (b) Sampled valued (c) Averaged value

Figure 2.5: 4x4 supersampling result for a boundary pixel

Figure 2.6 shows the supersampling result for a curved vector shape on an 8x8 display. The transition at the boundary of the shape is much improved after supersampling each fragment, removing the jaggedness from the pixelated edge.

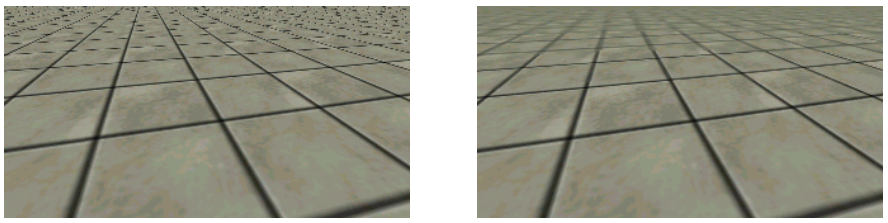


(a) No anti-aliasing (b) Supersampling

Figure 2.6: Effects of 4x4 supersampling on shape boundary

2.3.2. Minification

Minification occurs when a texture is shrunk to fit in a smaller number of display pixels than it's own texel resolution. Effectively, this means that in these cases there are more texels behind every display pixel. But because all of these sub-texels are represented by the one texel coordinate that happens to be in the center of their display pixel, spatial aliasing can occur. The most commonly known form of this is known as *Moiré*-patterns, illustrated in the top of Figure 2.7a. These patterns are the result of undersampling of a texture. There is more information than the number of samples. A prominent method to combat this is known as the pre-processing step named MIP-maps.

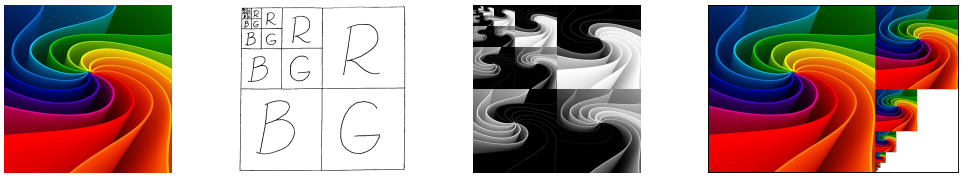


(a) Moiré patterns in the distance (b) Moiré solved with MIP-maps

Figure 2.7: Floor pattern with and without Moiré-patterns [29]

MIP-maps

MIP-mapping was originally introduced by Williams, L. in 1983 [31], with a (currently less popular) variant introduced by [4]. Williams describes a technique to store multiple resolutions of color images (having the three Red, Green, Blue channels) into a single grayscale texture. Each sub-resolution is a power of two smaller than the resolution above it. This is illustrated in Figure 2.8b and 2.8c. They depict the placement of lower resolution images in the final texture. This a very efficient method of storing multi-resolution information that can be easily accessed by a fragment shader, dependent on the amount that a texture has been shrunk, with a simple change in texture coordinates. This approach is limited by 3 channel graphics though. To enable MIP-mapping for 4 channel graphics, with their added Alpha channel, another structure is used. As illustrated in Figure 2.8d, all color- and alpha data is located in their respective channel, with smaller resolution variants positioned alongside the original resolution. Again, each resolution is a factor of two smaller than the one above it.



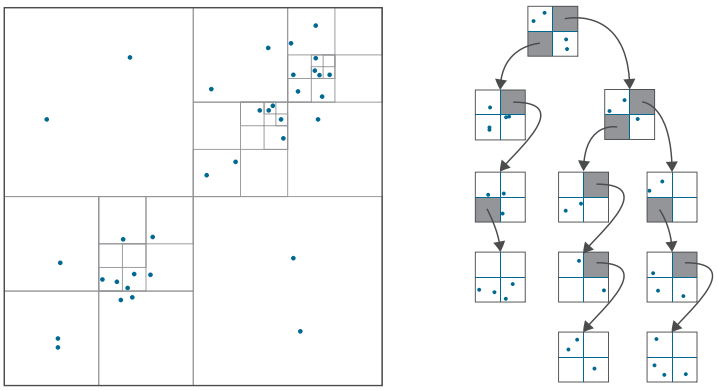
(a) Example Texture (b) 3-channel Structure (c) 3-channel MIP-map (d) 4-channel MIP-map

Figure 2.8: Illustrations for a 3-channel and 4-channel color MIP-map

Quad-trees

MIP-map structures have the property that the entire texture gets included on every stored MIP-resolution. This is fine for raster-based graphics, where details are spread out evenly over the entire texture as texels. Vector-based graphics, on the other hand, can have a lot of detail in some areas, and little to no detail in others. To reduce the amount of redundant information in such a texture, we can take advantage of a tree-like structure that was proposed by [15] for two and three dimensions. Depending on the dimensionality, these are respectively known as Quad-trees and Oct-trees. A specific tree-structure for textures was

proposed by [6] under the name Zipmaps. This technique was applied by [11] for interactive 3D visualizations for the CPU and later by [16] and [13] for the GPU. The method itself describes the iterative process of subdividing the texture- or geometry-space into equally sized subspaces until a certain limit is reached. This results in a tree-like structure, with shallow branches for areas with less detail and deeper branches for areas with more detail. Figure 2.9 shows an example of data points that are subdivided by a quad-tree. This can also be seen as vertices in a 3D scene. The rule for this specific quad-tree was to allow a maximum of 2 data points per cell. Beginning at the highest hierarchical quad, which contains the entire area, each cell counts a number of data points that it contains. If this value exceeds 2, the cell gets subdivided and the process is repeated. This provides a structure in which large areas that can be represented in a simple way, in this case by their two data points, are picked out quickly, and become available at a higher (and therefore easier to reach) branch in the tree.



(a) Subdivision of space

(b) Quad-tree structure for subdivision

Figure 2.9: An example of a the subdivision of a space by a quad-tree, with 2 points per subspace.

2.4. Vector based rendering techniques

Rendering vector-based graphics differs fundamentally from rendering raster-based graphics, it is not defined as the easy-to-interpolate look-up grid that defines a raster-based graphic. Since only the boundary of a shape is known, it is a non-trivial task to do the in-out test to determine if a given texture coordinate lies either inside or outside the vector shape. This is especially important when the GPU is involved since the fragment shader relies mostly on a specific texture coordinate to calculate the color of a display pixel.

In-out test variants

Two popular variants of this test are the *Even-odd rule* and the *Non-zero winding rule* [7]. Both of these rules can give a different result for vector shapes with intersecting segments, and it is therefore often stated at the start of the shape description which rule should be used in the in-out test.

The Even-odd rule for a specific texture coordinate can be specified as moving along a line, in any direction, from that texture coordinate to infinity, while switching its state between inside to outside at every path segment that it crosses. An even amount of crosses results in the coordinate being outside the shape and inside for an off amount of path segment crosses.

The Non-zero winding rule is similar to the even-odd rule but relies on the direction of every path segment that it crosses. When the line crosses a segment with a clockwise direction, the total score for that coordinate gets one point added to its *winding score*. One point is subtracted for each counter-clockwise segment. A total score of zero means that the coordinate lies outside the shape and a non-zero score results in the coordinate being marked as inside the shape.

Using these rules, the entire shape can be rasterized by selecting a regular grid of coordinates, depending on the required display resolution, and performing the selected in-out test for each of those coordinates. Because this is still a brute force approach, other attempts have been made to improve this process, including the approach of this work. The following approaches relate to this work in the way that we utilize render speedup that the parallel nature of dedicated graphics hardware offers. We do this in a texture based approach that supports curved shape boundaries, up to cubic Bézier curves. The proposed texture format gives a fragment shader all the required shape information that is necessary to compute a simple in-out test for a specific texture coordinate. The texture format also uses an efficient distribution of detail, where the amount of stored data does not just naively scale with the surface that the vector shape covers, but with the amount of detail that it has. The format is also extended with extra information that allows the fragment shader to handle minification artifacts, all in a single render pass. Lastly, this work discusses a method of converting existing vector shaped to this new texture format. This approach was inspired by combining some techniques that have been proposed before. These consist of geometry- and texture-based approaches.

2.4.1. Geometry approaches

One of the more popular geometry-based approaches was proposed by Loop & Blinn [17] and describe an elegant method using Delaunay Triangulations [2] to convert a curved shape into triangle geometry. The coordinate system inside the triangles on the edge of the shape is chosen in such a way that the final in-out test comes down to the reduced curve shape $f(u,v) = u^2 - v$, where u and v are the texture coordinates. All that then remains, to validate if the texture coordinate is underneath that curve, is to check if $f(u,v) < 0$. This method was later improved by Kokojima et al. [14], by changing the triangulation method to *Line-edged triangle fans* and utilizing stencil-buffer and multi-sampling techniques. Similar to our approach, these techniques cut up the initial vector shape into individual triangles that provide enough information to the fragment shader. The fragment shader then only needs to do a fast in-out check to compute the 'insideness' of a texture coordinate.

Although these techniques are promising, they do not provide sufficient support for minification and the artifacts that go along with it. In the case where there is a lot of geometry under a specific display pixel, under-sampling will occur, resulting in undesired Moiré patterns. This work does provide support for these minification scenario's but through a texture-based approach.

2.4.2. Texture approaches

Besides using geometry in the graphics pipeline, there is a wide variety of approaches that utilize textures to store vector information.

Distance field

To provide shape information for individual texels, Frisken et al. proposed a technique [8] that uses a distance map to deliver this information. This approach was later implemented and tailor fitted for textures by Green [9]. This method uses an additional texture to store a grayscale distance map, that can be bi-linearly interpolated to any desired resolution, to aid in creating sharp edges for the shapes that were represented in the color texture. This approach is very promising, also because it is compatible with graphics hardware. Though, because it is based on a discrete sampling of the distances, the estimation becomes more and rougher when too much magnification is applied.

Shadow maps

One area where the combination of vector and texture data is very relevant is in shadow maps. Shadow maps are generally raster-based and used to represent the projected outline of a 3D model as a shadow onto a textured surface. The triangles of 3D models are vector based, which are then rasterized into a shadow map, using ray-tracing from the perspective of the light source. The resolution of these shadow textures has a huge effect on the precision and quality of the resulting shadow. Especially when a light source if very far away from the surface where the shadow is cast on.

Sen et al. [28] proposed a method to encode boundary corner information in individual texels. This boundary information is then interpolated in a later stadium, at the point when the shadow is actually rendered on a surface. A year later, Sen, P. proposed an extension of this technique [27] to be used on actual geometry textures, allowing for improved texture magnification. Still, both techniques only allow for straight vector lines but were good examples in the sense that they stored information in textures that later could be interpreted in the rendering process to allow for improved texture magnification.

Sharp boundaries

This idea was also picked up the same year by Tumblin et al. [10] and Ramanarayanan et al. [23], where Tumblin et al. introduced *Bixels* to embed sharp, straight boundaries in pixel data to improve an otherwise blurry and imprecise magnification result. This technique combines the texture's color data with straight vector boundaries in a single texture, allowing for great magnification. This technique was unfortunately limited to only straight lines and the resolution of the vector definition had to be the same size as the original texture resolution.

Ramanarayanan et al. suggested a technique that also stores features in textures, which did include curved vector lines. Where [10] used two straight lines per texel to represent vector boundaries, Ramanarayanan et al. used two vector curves to represent sharp boundaries in textures. This increased the applicability of this technique immensely over earlier approaches, but it was not designed to work with dedicated graphics hardware.

The following year, Tarini et al. proposed another technique named *Pinchmaps* [30], suggesting the use of an extra texture to store the pinching information alongside the original color texture. This pinching information is mainly located around the pixels that are on the edge of a shape, describing how the color on both sides of the edge should be pinched together to create a smooth boundary. Because the vector information would be distributed evenly over the texture, similar to color information in raster graphics, a lot of memory space is wasted on areas that do not have edges in them.

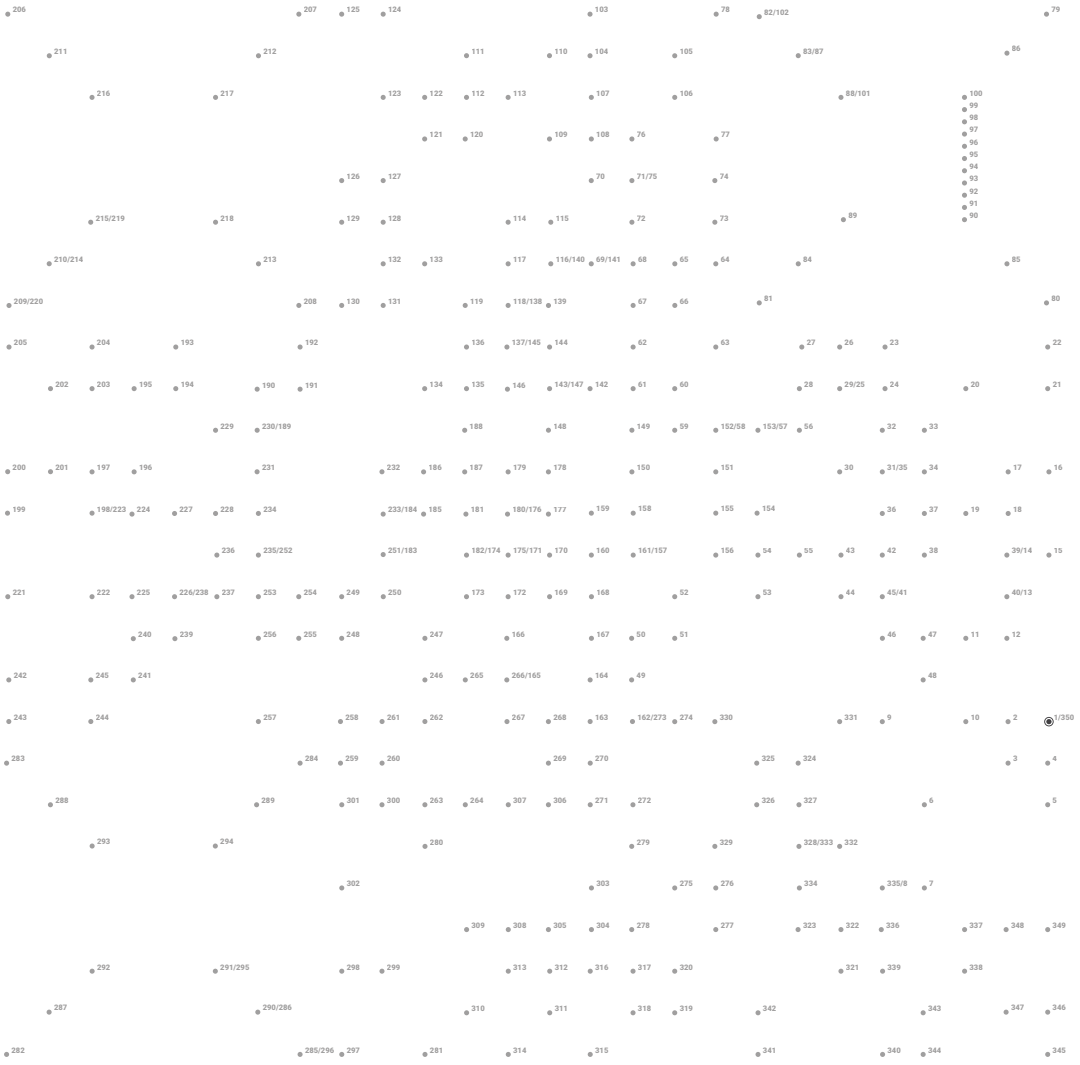
GPU

In order for vector information to be easily obtainable inside the graphics pipeline, it has to be compatible with random-access. This means that a fragment shader should be able to take any random coordinate inside the texture and compute the in-out test without requiring the entire shape. Ray et al. [24] propose a method takes this into account, by storing two curves inside each texel. This then represents a piece-wise representation of the original vector shape. This segmentation is combined with a 16-tree structure to deal with the uneven distribution of detail in vector graphics, very similar to the approach in this work. Unfortunately, they provide no automatic conversion or fitting from any raster- or vector-representation to their specific texture format.

Such a pre-process is described by the work of Parilov et al. [21], where a shape's outer edge is described as piece-wise quadratic Bézier curves. The texture is split up into a regular grid of texels, with each texel containing up to two quadratic curves. This approach does not take the distribution of detail over the texture into account, which allows it to use an interesting method of calculating the distance of a coordinate to the nearest curve segment. This distance calculation was also thought of to reduce the aliasing effects for this work but did not work with in combination with the implemented quad-tree structure. This structure meant that every cell can have an arbitrary size, where the cells in a regular grid all have the same size.

Random-access rendering of vector graphics was also the main topic in an approach by Nehab et al. [18]. They extended previous work by allowing multiple vector shapes to layer on top of each other, by allowing a variable amount of data and shape information for each of the distinct cells. This is combined with a clipping scheme that they claim is faster than hierarchical clipping schemes. This does have the downside that the entire shape is rasterized into fixed-size cells, resulting in a higher memory strain for shapes with unevenly distributed detail. Similar to this work, anti-aliasing is achieved in a single rendering pass. A similar technique was proposed by Qin et al. [22], with the addition that cubic Bézier curves were used to better mimic the input SVG files of an arbitrary complexity.

The next chapter will discuss the actual method that was used to answer this work's questions in detail.



3

Methods

This chapter solves the question that was presented in the introduction, *"How can we store vector shape descriptions in such a way, that dedicated graphics hardware can efficiently render them in a single, parallel pass at any given pixel resolution?"*. This is done by solving a number of smaller sub-problems. The following sections address these sub-problems in order. Section 3.1 discusses how the boundary of vector shapes are split up into piece-wise curves to fit the structure of the graphics pipeline. This is followed by Section 3.2, that discusses how texture coordinates for each of the grid cells are converted to local coordinates and then to a normalized coordinate system. Given this normalized coordinate system, Section 3.3 explains in detail how to end up with a simple height comparison of the normalized texture coordinate and the height of the curve at that same x-value. This then determines the in-out test for that specific texture coordinate. It is arbitrarily decided that, when following a curve from P_0 to P_3, the inside of the shape always lies on the right side. The next section, Section 3.4, expands on the Section 3.1, by taking advantage of both the quad-tree and MIP-map structures, to store the curve information in a more resolution-independent structure. This is then combined with MIP-values to provide a less spatially redundant structure in Section 3.5. Lastly, Section 3.6 summarizes all the mentioned techniques and illustrates the effect that they have on the final render when combined into a single method.

3.1. Curves in the graphics pipeline

The graphics pipeline and its programmable shaders require a certain data structure and programming style where data gets interpolated for specific texture coordinates on a texture. Vector shapes are normally not limited to a certain rasterized resolution, so where it is easy for a raster-based shape to check if a specific fragment shader should return either an inside- or outside-value for a shape, for vector shapes this is less trivial.

The solution here is to convert the vector shape into raster-based chunks that provide all information at a texture-coordinate level for every running fragment shader. This allows the global vector edge data to be available in those local cells since every cell is then represented by a single cubic Bézier curve. This is even the case for cells that completely fall inside or outside the shape. The curves in these cells can represent both full and empty cells. An example is shown in Figure 3.1, where a vector shape is subdivided into smaller rectangles in such a resolution that the vector shape in every cell can be represented by a cubic Bézier curve. These raster cells each represent a single cubic Bézier curve and can be seen as pixels in a texture (texels). Each of these texels then stores all the required information to describe the curve that fills that texel. It should be noted that this will be improved with the addition of a quad-tree in Section 3.4. This allows for large empty and full areas to be represented by a single empty or full cell, thereby focusing mostly on the shape's boundary pixels.

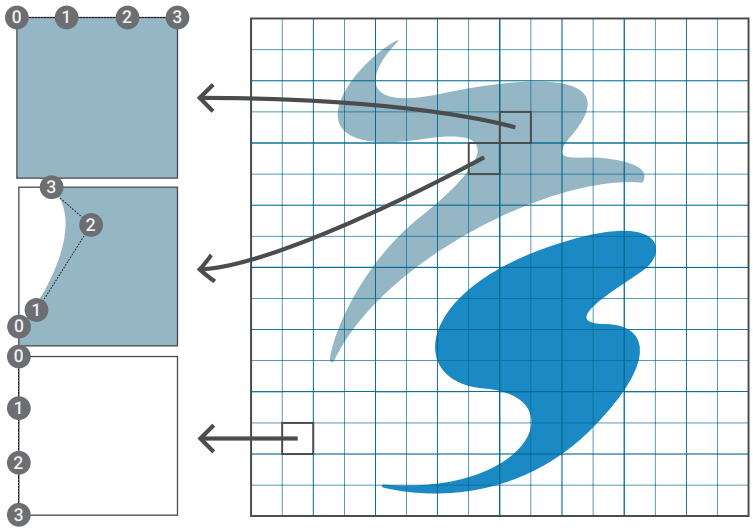


Figure 3.1: Square-wise subdivision of the vector space with example curve sections for full, partial and empty curves.

3.2. Normalized coordinate system

Given a rasterization like the one that is shown in Figure 3.1, every cell contains a single cubic Bézier curve, defined by its four control points P_0 to P_3. Because P_0 and P_3 are not bound to the same axis, or even the same cell edge, a change of basis is required to convert the coordinate inside the cell to a normalized basis where P_0 and P_3 have the simple XY-coordinates $(0, 0)$ and $(1, 0)$, respectively. This makes it easier to check if a normalized texture coordinate is above or below the curve, by only having to compare a single height value. Changing the basis is done by a simple matrix multiplication with the change-of-basis matrix, as defined by the position of P_0 and P_3 on the edge of the texel. This is illustrated in Figure 3.2, where the coordinate system of a curve with its four control points is changed from the original cell's texture coordinate system to the normalized XY-coordinate system. It also shows the conversion of an example fragment coordinate C.

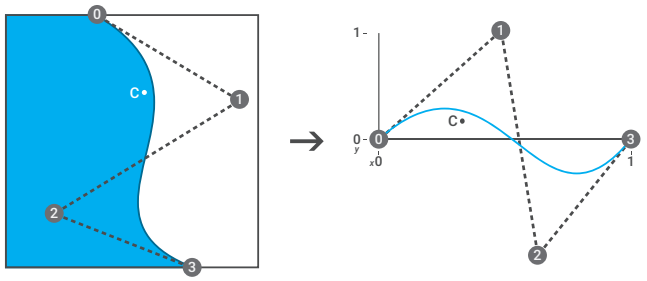
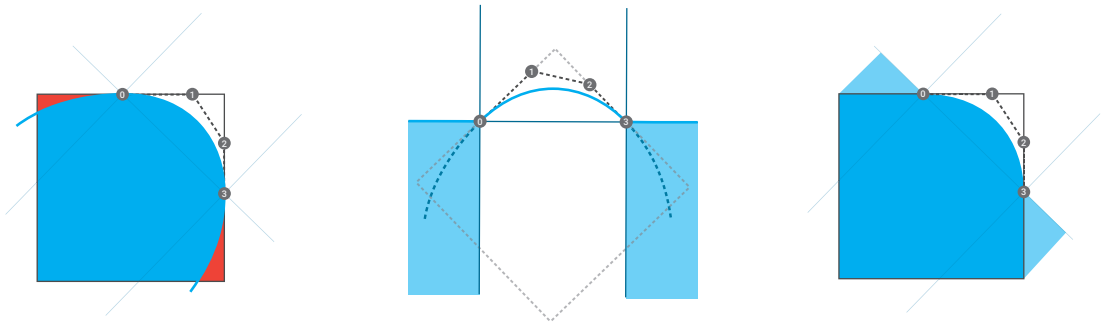


Figure 3.2: Coordinate normalization of an example texel

There are special precautions that have to be taken since the defined curve is parametric and does exist outside the zero-to-one range of the normalized coordinate system. This can lead to unwanted artifacts, as shown in Figure 3.3. This is caused by the parameterized nature of the curve. It is very well defined if t remains between zero and one, but because the curve is based on a continuous function, the curve also exists outside that range. This can lead to a specific scenario shown in the first figure. Since the behavior outside the zero-to-one range is not relevant, the solution is to ignore the curve's values outside this range and replace it with a single check. As illustrated, the light shaded areas are defined to always be inside, depending on the sign of the rebased y-value. The coordinate is inside the shape if this value is below zero, otherwise, it is outside the shape.



(a) Issue: The read area is drawn as 'above' the curve

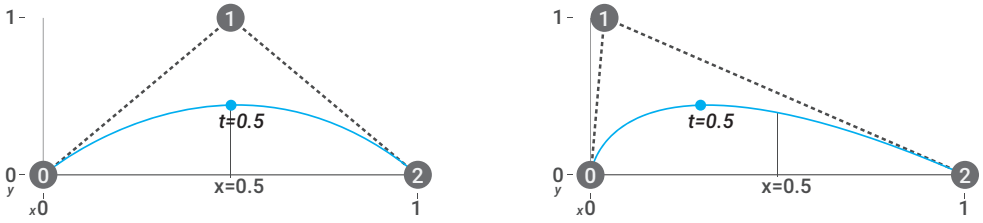
(b) In the normalized coordinate system, force the coordinates in the shaded area to always be 'under' the curve

(c) Issue resolved by this check

Figure 3.3: Artifacts from parameterized Bézier curve

3.3. Height comparison

After the fragment's texture coordinate and the curve's control points are normalized to their new XY-coordinates, they are be used to calculate the height of the curve and check if that is above or below the fragment's XY-coordinate. Bézier curves are parametric, this means that the XY-coordinates along the curve are calculated as a result of a single input value t. This value t has a range between 0 and 1, where a value of 0 results in the same coordinate as the start point P_0 and a value of 1 results in the same coordinate as the endpoint P_3. Because the fragment shader initializes from a coordinate C, we need to find the value t that represents the point along the curve that has the same x-value as C. This point along the curve is then directly below or above fragment coordinate C. With this t-value found, we calculate the height of the curve and then compare this height with C to check if the fragment coordinate lies below or above the curve.



(a) Symmetric

(b) Asymetric

Figure 3.4: The difference between symmetric and asymmetric Bézier curves

To recap section 2.2.1, the equation that describes a cubic Bézier curve, with control points P_0, P_1, P_2 and P_3, is as follows:

$$B(t) = P_0(1-t)^3 + 3P_1(1-t)^2t + 3P_2(1-t)t^2 + P_3t^3, \text{with } 0 \leq t \leq 1$$

We split this cubic Bézier equation for the x- and y-coordinate, while also substituting coordinates for P_0 and P_3 from the normalized coordinate system, gives the following (again, $0 \leq t \leq 1$ still applies):

$$P_0 = (0,0)$$
$$P_3 = (1,0)$$

$$\begin{aligned} B(t)_x &= (1-t)^3 P_{0,x} + 3(1-t)^2 t P_{1,x} + 3(1-t)t^2 P_{2,x} + t^3 P_{3,x} \\ &= 3(1-t)^2 t P_{1,x} + 3(1-t)t^2 P_{2,x} + t^3 P_{3,x} \end{aligned} \tag{3.1}$$

$$\begin{aligned} B(t)_y &= (1-t)^3 P_{0,y} + 3(1-t)^2 t P_{1,y} + 3(1-t)t^2 P_{2,y} + t^3 P_{3,y} \\ &= 3(1-t)^2 t P_{1,y} + 3(1-t)t^2 P_{2,y} \end{aligned} \tag{3.2}$$

Now we use the equation for $B(t)_x$ (3.1) to find t, since $B(t)_x = C_x$, and then fill in this t in the equation for $B(t)_y$ (3.2) to find the height of the Bézier curve at x-coordinate C_x. Solving $B(t)_x$ for t requires that it is first converted to a standard from with the following substitutions:

$$a = 3P_{1,x} - 3P_{2,x} + 1$$
$$b = 3P_{2,x} - 6P_{1,x}$$
$$c = 3P_{1,x}$$
$$d = -C_x$$

Resulting in the following simpler equation:

$$0 = at^3 + bt^2 + ct + d \tag{3.3}$$

A method for solving this equation was found and published by François Viète (1540–1603) and later extended by René Descartes (1596–1650) [19]. There are 9 scenarios for different relevant inputs, where for each one the t-value has to be calculated with another equation. This is partly the case, because of the trigonometric nature of the solutions and the limited input that is allowed for some of those functions. An overview of all these scenario's and when to use each one of them is presented in Figure 3.5. This figure also references to Equations 3.4 through 3.8 and Equation 3.10 through 3.12. These will be discussed in the upcoming sections, along with what needs to be done with some of the t's that are calculated. This is because some scenarios have multiple t's for a single fragment coordinate, so the correct one needs to be selected. As Figure 3.5 shows, this is only necessary for four of the nine scenarios.

3.3.1. The straight case ($a = 0$ and $b = 0$)

It can happen that both a an b are zero. For cubic Bézier curves is this the case when $P_{1,x} = \frac{1}{3}$ and $P_{2,x} = \frac{2}{3}$. This is the specific symmetric case that was explained in Section 3.3, where $P_{1,x}$ and $P_{2,x}$ together mimic a virtual control point P that describes the cubic curve as a quadratic one. If this is the case, the solution for t becomes trivial:

$$\begin{aligned} t &= -\frac{d}{c} = -\frac{-C_x}{3P_{1,x}} = \frac{C_x}{3P_{1,x}} \\ &= C_x, \text{ because } P_{1,x} = \frac{1}{3} \end{aligned} \tag{3.4}$$

Using this value t, we calculate the height of the Bézier curve with Equation 3.2 and compare it with the fragment coordinate's y-value C_y. If $C_y < B(t)_y$, this fragment lies under the curve, otherwise it lies above the Bézier curve.

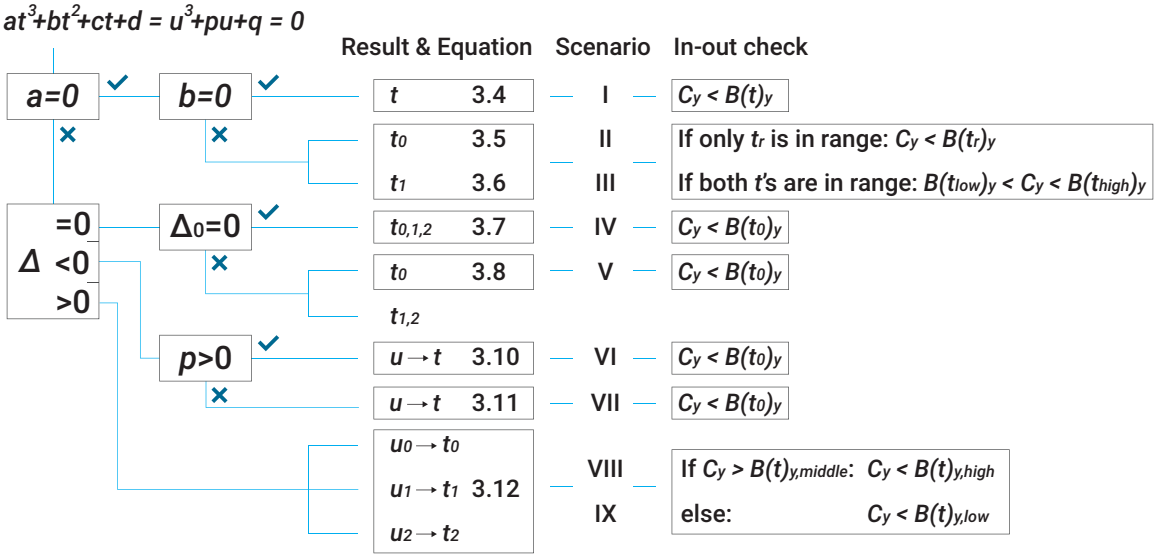


Figure 3.5: All solution paths summarized

3.3.2. The quadratic case ($a = 0$)

In the case where only a is zero, the curve can still be described as a quadratic curve, meaning that the difference between $P_{1,x}$ and $P_{2,x}$ is $\frac{1}{3}$, but it is not symmetric. In this case, the (now 2-nd power) equation can be solved with the ABC-formula:

$$t = \frac{c \pm \sqrt{c^2 - 4bd}}{-2b} = \frac{3P_{1,x} \pm \sqrt{\left(3P_{1,x}\right)^2 - \left(-4(3P_{2,x} - 6P_{1,x})C_x\right)}}{-2\left(3P_{2,x} - 6P_{1,x}\right)}$$

$$= \frac{3P_{1,x} \pm \sqrt{3^2\left(P_{1,x}^2 - C_x\left(2\frac{2}{3}P_{1,x} - 1\frac{1}{3}P_{2,x}\right)\right)}}{3\left(4P_{1,x} - 2P_{2,x}\right)}$$

$$= \frac{3P_{1,x} \pm 3\sqrt{P_{1,x}^2 - \frac{2}{3}C_x\left(4P_{1,x} - 2P_{2,x}\right)}}{3\left(4P_{1,x} - 2P_{2,x}\right)}$$

$$= \frac{P_{1,x} \pm \sqrt{P_{1,x}^2 - \frac{2}{3}C_x\left(4P_{1,x} - 2P_{2,x}\right)}}{4P_{1,x} - 2P_{2,x}}$$

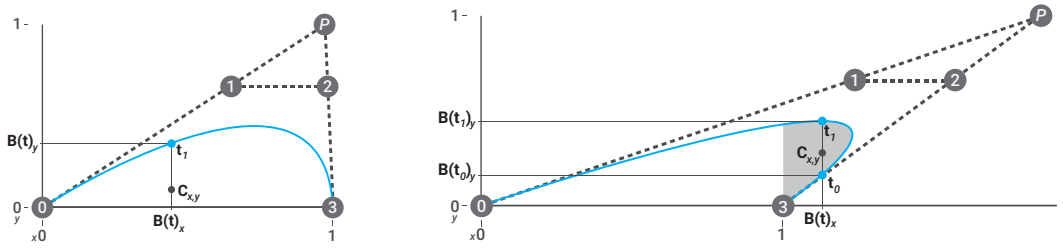
Resulting in the following two solutions:

$$t_0 = \frac{P_{1,x} + \sqrt{P_{1,x}^2 - \frac{2}{3}C_x\left(4P_{1,x} - 2P_{2,x}\right)}}{4P_{1,x} - 2P_{2,x}} \tag{3.5}$$

$$t_1 = \frac{P_{1,x} - \sqrt{P_{1,x}^2 - \frac{2}{3}C_x\left(4P_{1,x} - 2P_{2,x}\right)}}{4P_{1,x} - 2P_{2,x}} \tag{3.6}$$

Both solutions can be valid, in the case that is illustrated in Figure 3.6b. The curve here can still be described as quadratic, but the virtual control point P has an x-value that does not lie between P_0 and P_3. This means that for certain fragment coordinates (outside of the 0-1 range) there are two valid solutions for t. This is checked by looking at t_0 and t_1 separately. If both their values are between 0 and 1, they are both valid. Otherwise, just the value of t that is between 0 and 1 is the valid solution.

If there is only one valid solution for t, like in Figure 3.6a, we calculate the height value with Equation 3.2 and check if C_y lies above or below that value. However, if there are two solutions for t, we need to change the height comparison slightly. We then calculate the Bézier height for both t's and check if C_y lies between those two heights. If this is the case, the fragment lies inside the shape. Otherwise, it lies outside.



(a) Quadratic with single solution

(b) Quadratic with two solutions

Figure 3.6: Quadratic curves with single and multiple solutions for t

3.3.3. The cubic case ($a \neq 0$)

When a is not equal to 0, we have an actual cubic Bézier case, with possible inflection points, cusps, and loops. These scenarios mean that there are two and even three valid t-values for a given fragment coordinate C. In order to find out which scenario the current curve is in, we follow the method proposed by Viète and Descartes. First, we check which scenario we are in by calculating the curve's discriminant Δ:

$$\Delta_0 = b^2 - 3ac$$
$$\Delta_1 = 2b^3 - 9abc + 27a^2d$$
$$\Delta_2 = \Delta_1^2 - 4\Delta_0^3$$
$$\Delta = \frac{\Delta_2}{-27a^2}$$

We distinguish three different scenarios, depending on the sign of the discriminant:

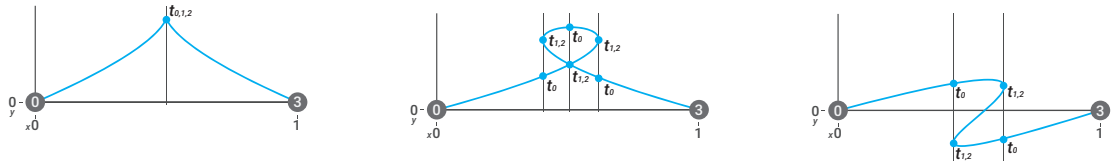
1. $\Delta = 0$ means that two or three solutions are identical.
2. $\Delta < 0$ means that there is only a single unique solution.
3. $\Delta > 0$ means that there are three unique solutions.

The following three subsections will explain what to do in each of these three scenarios.

Scenario 1 ($\Delta = 0$): Two or three identical solutions

There is a scenario in which the control points P_1 and P_2 are positioned in such a way that for specific fragment coordinates two of the three, or all three, solutions for t become identical. These cases are illustrated in Figure 3.7. The first figure shows a cusp, in which all three solutions are identical to each other. The second and third figure show a loop and a swirl, respectively, with vertical lines at the fragment coordinates where two of the three solutions become identical. It shows that this is the case where the curve changes horizontal direction, or where two lines of the curve intersect each other.
These two cases are separated by checking if $\Delta_0 = 0$. If this is the case, then we have a cusp and all three solutions $t_{0,1,2}$ are identical. these three solutions are then calculated as follows:

$$t_{0,1,2} = -\frac{b}{3a} = \frac{6P_{1,x} - 3P_{2.x}}{3P_{1,x} - 3P_{2,x} + 1} = \frac{2P_{1,x} - P_{2.x}}{P_{1,x} - P_{2,x} + \frac{1}{3}} \tag{3.7}$$



(a) Solutions $t_{0,1,2}$ in a cusp (b) Solutions $t_{1,2}$ in a loop (c) Solution $t_{1,2}$ in a swirl

Figure 3.7: Identical solutions in a cusp, loop and swirl

In the other case, where $\Delta_0 \neq 0$, we have the unique solution t_0 and the two identical solutions $t_{1,2}$. These are calculated as follows:

$$t_0 = \frac{4abc - 9a^2d - b^3}{a\Delta_0} \qquad (3.8)$$
$$t_{1,2} = \frac{9ad - bc}{2\Delta_0}$$

Looking at the previous figures, we see that at the coordinates where there are two valid solutions for t, we still do a simple height check with just the unique solution t_0. Because the identical solutions $t_{1,2}$ is the point where the curve changed direction, the shape does not continue any further from that point in the horizontal direction and should thus be omitted in favor of t_0. This t_0 is then used to compute the curve height, which is compared with C_y, like before.

The following two scenarios, one and three unique solutions, require the original equation (3.3) to be converted to its *depressed* form, where the quadratic term has coefficient 0. This is done by diving it by a and using *Viéte's substitution* $t = u - \frac{b}{3a}$, resulting in the following equation:

$$u^3 + pu + q = 0 \qquad (3.9)$$

With:

$$p = \frac{-\Delta_0}{3a^2} \text{ and } q = \frac{\Delta_1}{27a^3}$$

Scenario 2 ($\Delta < 0$): One unique solutions

Given the depressed form of the original equation (3.9) and the fact that there is one unique solution in this scenario, there are two cases to discuss. Both cases are computed with hyperbolic functions that have a limited input range and require a check in advance to determine which of the two has to be used. The used hyperbolic functions sinh, $\sinh^{-1}$, cosh and $\cosh^{-1}$ are defined as:

$$\sinh(x) = \frac{e^x - e^{-x}}{2} \qquad \sinh^{-1}(x) = \ln\left(x + \sqrt{x^2 + 1}\right)$$
$$\cosh(x) = \frac{e^x + e^{-x}}{2} \qquad \cosh^{-1}(x) = \ln\left(x + \sqrt{x^2 - 1}\right)$$

For example, the $\cosh^{-1}$ function needs its input to be between -1 and 1. The check that separates both cases is written as $p > 0$ or $4p^3 + 27q^2 > 0$, where the latter implies $p < 0$ and is necessary to ensure that x in $\cosh^{-1}(x)$ is between -1 and 1.

In case $p > 0$:

$$u = -2\sqrt{\frac{p}{3}}\sinh\left(\frac{1}{3}\sinh^{-1}\left(\frac{3q}{2p}\sqrt{-\frac{3}{p}}\right)\right) \qquad (3.10)$$

In case $p < 0$ and $4p^3 + 27q^2 > 0$:

$$u = -2\frac{|q|}{q}\sqrt{-\frac{p}{3}}\cosh\left(\frac{1}{3}\cosh^{-1}\left(\frac{-3|q|}{2p}\sqrt{-\frac{3}{p}}\right)\right) \tag{3.11}$$

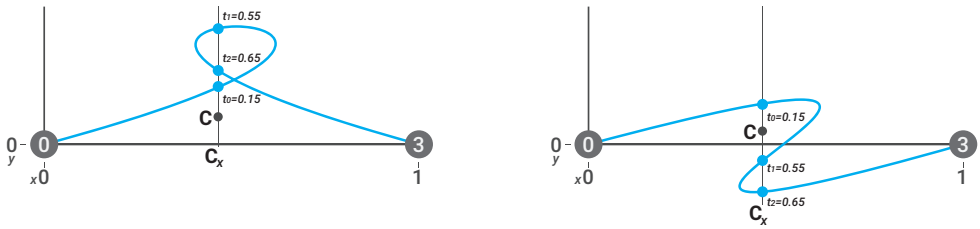
Equations 3.10 and 3.11 are used to find u. This is then used to calculate t, by using Viéte's substitution $\left(t = u - \frac{b}{3a}\right)$ again. Once t is obtained, it is used to calculate the curve height like before, which is then compared with C_y to see if the fragment coordinate is above or below the curve height.

Scenario 3 ($\Delta > 0$): Three unique solutions

Lastly, since loops and swirls can occur, there are fragment coordinates that have three unique solutions for t. Two examples of these are shown in Figure 3.8. One of a shader coordinate that has three unique solutions for t because it falls inside a loop, and another similar one, but then for a swirl. All three t values are given for both examples.

Similar to the previous scenario, this scenario also uses the depressed equation to calculate the three solutions $u_{0,1,2}$ with the following equation:

$$u_k = 2\sqrt{-\frac{p}{3}}\cos\left(\frac{1}{3}\left(\cos^{-1}\left(\frac{3q}{2p}\sqrt{-\frac{3}{p}}\right) - 2\pi k\right)\right), \text{ for } k = 0, 1, 2 \tag{3.12}$$



(a) Solutions in a loop

(b) Solutions in a swirl

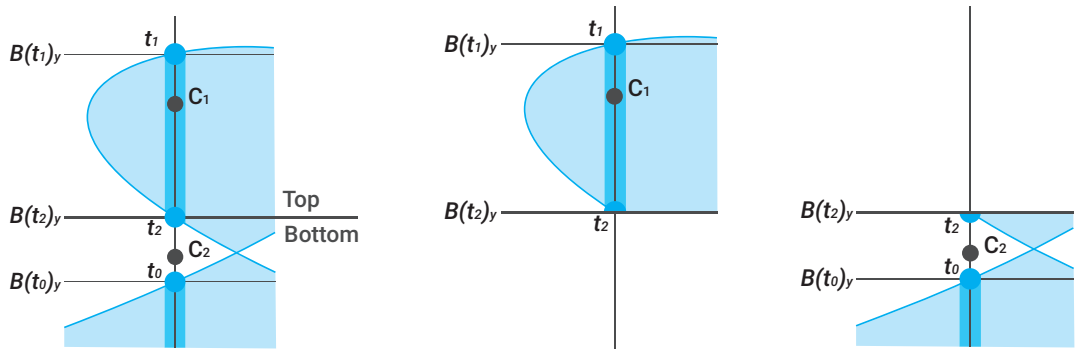
Figure 3.8: Three unique solutions $t_{0,1,2}$ in a loop and swirl

The u-values that are calculated with Equation 3.12 are used in Viete's substitution to calculate the corresponding t-values, which in turn are used to calculate the heights $B(t_k)_y$ of each of the three points. However, since there are now three height values to compare with C_y, we have to find which of those we need to use to get the correct result out of the comparison. Figure 3.9 serves as an illustration of this. It shows example fragment coordinates C_1 and C_2 in a loop, with the three heights $B(t_0)_y$, $B(t_1)_y$ and $B(t_2)_y$ for t_0, t_1 and t_2, respectively.

- $B(t_2)_y$ is the middle value, so this is first compared with $C_{1,y}$ and $C_{2,y}$, resulting in $C_{1,y} > B(t_2)_y$ and $C_{2,y} < B(t_2)_y$.
- This means that for final height comparison $C_{1,y}$ will be compared with $B(t_1)_y$, since $B(t_1)_y > B(t_2)_y$.
- Similarly, $C_{2,y}$ will be compared with $B(t_0)_y$, since $B(t_0)_y < B(t_2)_y$.

This is shown in Figure 3.9b and 3.9c, respectively. In both cases we know the value of C_y, and can therefor still use the single value comparison, after checking if it is higher or lower than the middle $B(t)_y$ value.

Summarizing, we are able to calculate the correct t value for all inputs of P_1, P_2, given a fragment coordinate C. This t is then used to calculate the relevant height of the Bézier curve at the point that is precisely above or below this coordinate C. With this, a simple check verifies if C is located inside or outside the vector shape, which is piece-wise rasterized into separate cubic Bézier curves, to make sure that only the relevant piece of the shape is available for every fragment coordinate. The next section talks about a smarter distribution of these curve pieces, depending on the distribution of detail along the shape's edge.



(a) Overview with fragment coordinates C_1 and C_2

(b) Top area, containing C_1

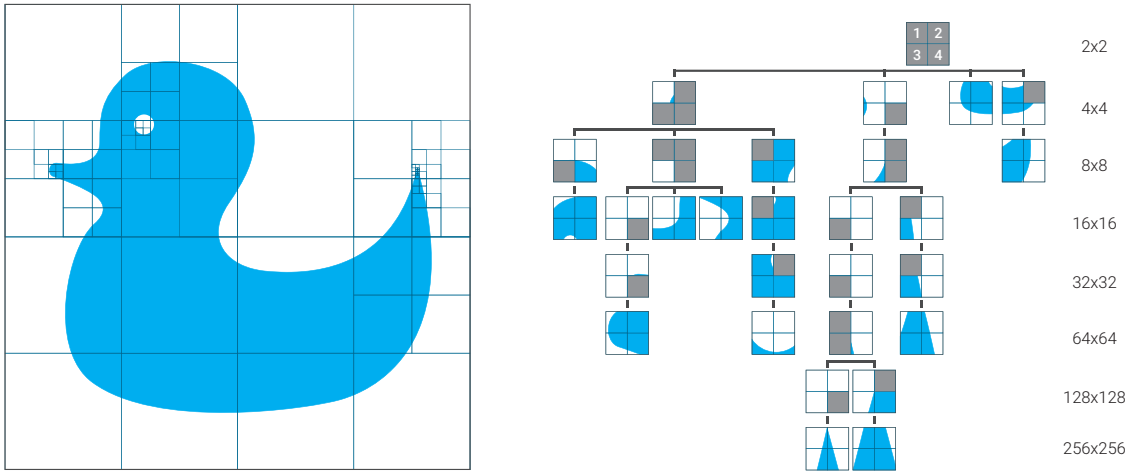
(c) Bottom area, containing C_2

Figure 3.9: Height comparison in a cubic loop, for three unique solutions and two example coordinates

3.4. Hierarchy

As announced in the beginning of this chapter, the original rasterization of a vector shape can be improved by not just using the smallest resolution that allows for the encoding of the smallest detail in the vector shape. This could potentially waste a lot of memory, as illustrated in Figure 3.10a. If the entire shape would have been subdivided in the resolution of the smallest detail of the shape (near the beak and tail of the duck), then all the big cells would have been subdivided as well, resulting in a regular grid of 256 by 256 vector texels. It is clear that large chunks of the shape can be represented by a larger curve, so we use a quad-tree structure to subdivide the shape into smaller cells when they cannot be represented by a single cubic Bézier curve. This structure is organized in a hierarchy that is also useful to store MIP-data and each depth layer in the hierarchy is illustrated in Figure 3.10b.

- Full gray cells refer to one of the underlying sub-quads, which are laid out in the order that is shown in the quad at the top of the structure.
- The other cells in the quad structure represent piece-wise curve definitions that can be traced back to the original shape. The different scales of these curves in their cells are irrelevant since the shape is defined in a resolution-independent vector format.



(a) Quad-tree subdivision of the vector shape

(b) Memory structure at a cell level

Figure 3.10: Quad-tree subdivided structure

The structure in Figure 3.10b is separated into two lists, one for all the quads, and one for all the curve definitions. This separation is illustrated in Figure 3.11, where their vertical position is still linked to the way they were represented in Figure 3.10b for clarity. In reality, both the quads and curve lists are simply flattened into their own 1-dimensional list, with their vertical position being their index on the list. Each quad contains a boolean value and an index value for each of the four cells that it contains. The boolean value determines if the index should be interpreted as a quad-index or as a curve-index. Each of the curves in the curve-list has values that represent its control points P_0 to P_3.

This also comes with a significant decrease in storage size, when comparing the original 256-by-256-rasterization of 65.536 curve-cells with the total sum of all the cells in these two quad-tree lists. This example uses 29 quads, each describing 4 cells, so 116 quad-cells in total. These quads refer to a list of 56 curve-cells, bringing the total cell count up to 172. This is only 0.26% of the original cell count, clearly showing the advantage that this approach of structuring the data brings.

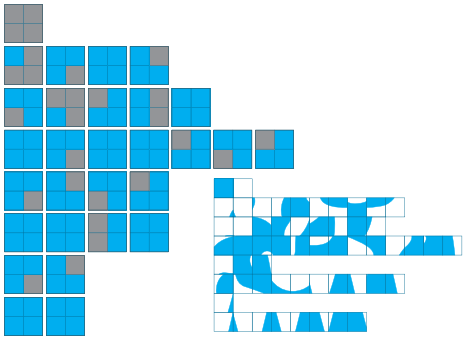


Figure 3.11: Quad structure per depth level

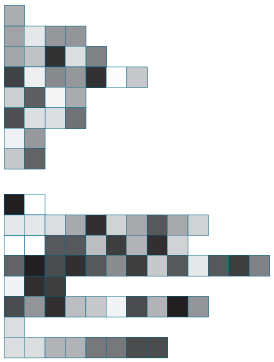


Figure 3.12: Minification texels for the quads and curves

3.5. MIP-mapping

Because this quad-structure contains hierarchy information, it is very suitable to also store MIP-values in the same way. We combine the concept of MIP values and the hierarchical structure by storing MIP summary values alongside the quads in the tree structure in order to prevent aliasing effects. When a quad is smaller than a display pixel, the fragment shader takes it MIP value to represent the entire quad, instead of going deeper into the hierarchy.

The acronym MIP stands for *Multum In Parvo*, meaning “much in little”. As described in Section 2.3.2, aliasing effects can be prevented by using a single value to summarize over all the detail in a specific texture area. Regular MIP-maps use this by storing a series of lower resolution versions of the original texture, each for half the size of the previous resolution. This means that a texel in a lower resolution version represents the average of the four texels from the higher resolution that it represents. This type of data averaging is also used in our method. Looking back at Figure 3.10b, both quads at the bottom contain four curves to average over, where the highest quad contains all the curve information that is structured in the tree.

Take the worst case example, where the shape is scaled down so much that the entire shape has to be represented by one single display pixel. In this case, you don’t want to go through the entire quad-tree to end up somewhere in the bottom, where a MIP-value that represents the highest quad, stored at the same level as that quad, would be enough to represent all the curves and detail that is underneath that top-level quad. Therefore, we store for each of the quads a single value that represents a summary of all the sub-quads and curves that are underneath them. This prevents the shader, when reading the texture and going through the quad-tree, from going deeper in the tree than is required, and use the ‘summary’ MIP-value for this display pixel’s quad instead. The MIP-values for the example shape of Figure 3.10 can be seen in Figure 3.12, where they are also still structured to visualize their depth and origin, like the quads and curves in Figure 3.11.

The quad's MIP-values are calculated bottom-up, starting with the quads that only contain curve cells. Initially, each curve-cell calculates the amount of area that is covered by the curve it contains. Then, for each of the quads, the average of its cells is taken and set as its MIP-value. This is a normalized value between 0 and 1. When a quad contains cells that are sub-quads, it can take the MIP-values of these sub-quads, since these should represent all the quads and curves that are inside this sub-quad.

3.6. In summary

Combining all the previously mentioned techniques leads to our approach, involving the rendering of piece-wise curve sections with dedicated graphics hardware and shaders, storing this in an efficient quad-tree and finally using Supersampling and MIP-values to get rid of aliasing effects like Moiré patterns. This is summarized in Figure 3.13, with each column showing the results for a different resolution of display pixels that the same shape is rendered in. The first column shows the shape, as it would be rendered in a single display pixel, up to the fifth column, where it shows results from the shape being rendered at a resolution of 16 by 16 display pixels.

The rows show the differences with and without some of the previously mentioned techniques, though all of the rows take advantage of the quad-tree structure. The first row shows the result of just naively traversing down the tree, and only using the value of the coordinate in the center of each display pixel as the fragment coordinate. Since only the center value is used, this is really hit-or-miss, as can be seen in the top left image. Because the center of the display pixel falls just outside the shape, the entire display pixels stays empty, as if there was nothing there. Also, the outer edge of the rendered shape does not appear smooth at all.

The second and third rows show the effect of using the MIP-values. The second row shows the actual rendered result, with the third row showing a reference, per pixel, of the origin of the rendered pixels in the second row. Grey cells note that the rendered value is actually a MIP-value that was stored for those specific cells, where blue cells identify cells that use the curve data at the end of the tree branches. Note that the MIP-values add some smoothing in the final render, but only to the cells that used them. The other rendered pixels still lack this smoothness, just like in the first row.

The last row shows the result when both MIP-values and Supersampling are used. Rendered pixels use the same MIP-values as the renders in the second row, but the other cells can now take advantage of this supersampling, where instead of just the center of each display pixel, an average is taken over 16 evenly spread out sub-coordinates, giving the entire shape a smooth edge. In these cases, the cells are already rendered over multiple pixels, showing that, even when this gets up-scaled to over 256 by 256 pixels (the original rasterized resolution where the smallest curve would have its own display pixel), this method will render smooth edged vector shapes.

The next chapter describes the steps that were taken to implement the concepts from this chapter.

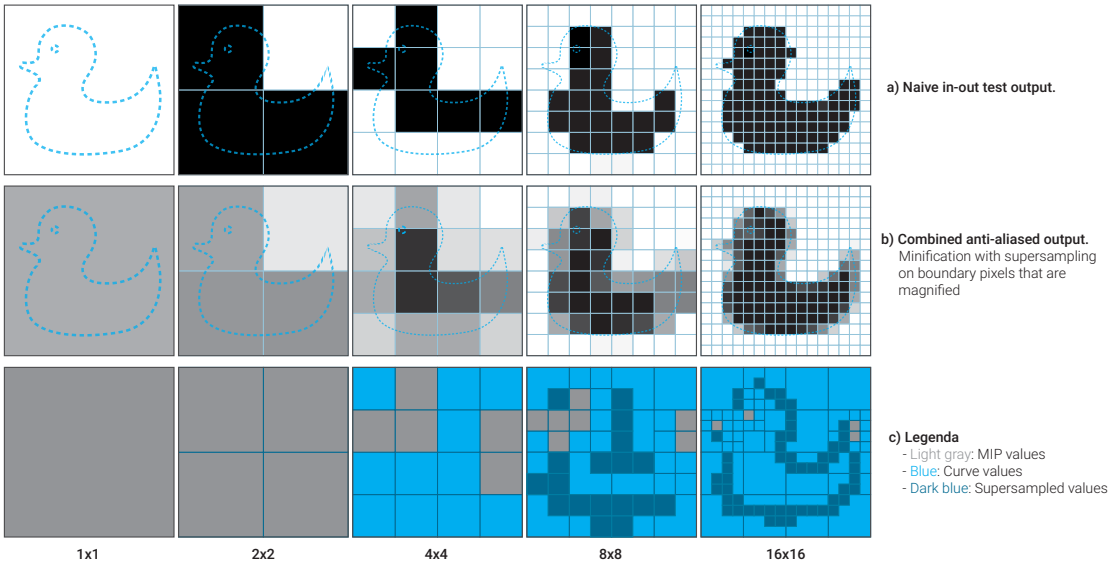


Figure 3.13: Multi-resolution comparison, with and without the addition of the MIP-values and Supersampling

4

Implementation

The implementation of the theoretical solution, as described in the previous chapter, is spread out over multiple stages, languages, and frameworks. In the first stage, the environments are introduced in which the implementation has taken place, as well as off-the-shelf code that was used. The second and third stage discusses the serialization of the vector data and writing this data to texture files, respectively. Next, in the fourth stage, the procedure of reading these texture files in a custom fragment shader is addressed. This includes deserializing the hierarchical data structure and using the added MIP-data to prevent aliasing effects. Lastly, some Deltares specific implementations are mentioned. The implemented source code can be found under DOI: 10.5281/zenodo.1168950

4.1. Use-case and setup

Because of the involvement of both Deltares and the USGS (United States Geological Survey) with this project, some additional requirements were added to the implementation. Firstly, it was requested that the implementation would be Web-based, meaning in a modern web browser. This was because of their wish to have the use-case's visualization be easily sharable. The use-case involves visualizing a flood model's shoreline from both high in the sky, as well as from someone standing on the ground. The scope of the project focused mainly on square graphics so that the solution would be easier included in any of the existing *Chart Tile Services*. These services are based on large 2D graphics, that will generally be only partly rendered to a display. The 2D graphics are split up into square tiles and only loaded when they are required, for example in the event of a user swiping to a new area in Google Maps™. Only the relevant tiles are loaded for the user, saving significantly on bandwidth and processing power.

Using dedicated graphics hardware in the web-browsers requires compatibilities with an immense amount of different hardware configurations. This is already mostly solved by using a JavaScript framework called *three.js*. This popular framework handles the rendering and manipulations of all objects and textures in a 3D scene. It can be easily set up to load any input texture files and *GLSL* (OpenGL Shading Language) scripts that will then be used in its WebGL graphics pipeline to run for all pixel fragments in the web page's display space.

All code was run and tested on a machine with an Intel Core™ i7-3630QM processor, running at 2.40 GHz, 8 GB of RAM and an NVIDIA GeForce GT 650M dedicated graphics card, which has 2048 MB of internal RAM.

4.2. Serializing vector data

Because of the complex representation of the data in the vector textures, it would require a separate UI to create new shapes and store these in the files. To save time, it was decided to instead convert already existing vector graphics to our vector texture format. This code was written in Matlab, because is one of the main programs that is used to manipulate coastal

data by both Deltares and the USGS, so it increases their ease of adaptation if this solution smoothly connects to their existing workflow. Secondly, some Matlab code was available online that would allow for out-of-the-box fitting of points to the parameters of a cubic Bézier curve. This reduced the amount of work significantly. Alongside this, some Matlab code was developed to verify the results of the graphics pipeline. This was necessary because of the parallel nature of the graphics pipeline that made it difficult to quickly debug shader code.

SVG files were selected as input, because of their wide availability and wide-spread compatibility with existing graphics systems. These vector shapes could also be provided in a raster format like PNG, but would then need an additional pass in a program like *Potrace* [26] to obtain the vector representation. The code is structured in a number of separate functions that are defined by their own stage in the process.

4.2.1. Shape boundary sampling

The first function takes samples along the shape's boundary path and stores these in a list of 2D points, based on a variable density. Initially, this density is based on the dimensions of the SVG's viewbox. This viewbox is the square coordinate system in which each of the path's segments is defined. This is just used as the initial density but will be increased if required. All the points stay in the order they were sampled in and each point gets a reference to the index of the segment it was sampled from. The latter means that when samples of a higher density are required for a certain path segment, these can be sampled and then added into the list, replacing the path's points that were sampled with the previous density.

4.2.2. Fit curves to sampled points

Taking this list of sampled points, we start an iterative process that takes a square space, find all the samples points that it contains and checks if these sampled points can be represented by a single cubic Bézier curve. If this is the case, this cell can be represented by this curve, and it gets stored in the list of curves. Otherwise, this cell gets subdivided into four equal-sized sub-cells, turning the cell into a quad of cells. This quad gets stored in the list of quads, with indexes pointing to each of its four sub-cells. These sub-cells get added to the queue of cells that need to be processed in this iterative process. The initial cell that gets processed if the shape's viewbox, containing all the sampled points.

In order to preserve continuity between neighboring cells where the shape passes through, it is important that the cubic Bézier curves inside these cells meet on the exact same location. Because the sampling of points does not take the location of potential cell boundaries into account, extra points need to be interpolated to achieve this. An example of this is shown in Figure 4.1. It visualizes the vertical boundary between two cells where the path segment passes through. When processing the left cell, the first point inside the right cell gets used to interpolate the point between them, at the exact location where the path segment crosses the boundary. This interpolated point gets added to the list of sampled points for this cell. A similar thing happens when processing the right cell, but here the interpolated point is the first point that gets added to the list.

It can also happen that the initial sampling was not detailed enough and there are no points available for a specific cell, even though the path segment crosses this cell. In this case, there will still be two boundary-points that were interpolated and we know the segments that these points originally came from. New points are sampled for this path segments with three times the original density. These points replace the original, smaller list of points that were calculated with the previous sample density. The value of three is used to make sure there are enough points between the two boundary points for a curve fitting. This concept is shown in Figure 4.2.

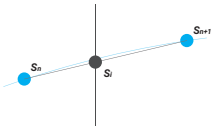


Figure 4.1: Positions of interpolated sample S_i on the edge between samples S_n and S_{n+1}

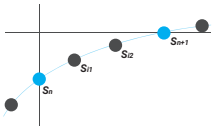


Figure 4.2: Positions of more densely interpolated samples S_{i1} and S_{i2} between the original samples S_n and S_{n+1}

After making sure there are enough sample points per cell, the cubic Bézier fit is performed. The code that was used for this was provided by Dr. M. Khan [12] on the MathWorks File exchange in 2009 and updated as recently as January 2016. This code requires a list of sample points and a least-squared error parameter and returns a number of cubic Bézier path segments with their control points P_0, P_1, P_2 and P_3. Because it was made sure in advance that there would be enough sample points for at least a single curve segment, the only thing that needs to be checked is if there is more than one path segment for the points in this cell. This is the check that determines if the cell can be described by a single curve or needs to become a quad with four sub-cells.

Since it is very common that a cell is completely empty or full, these special cases are added to the curve list at the start. When a cell is determined to be completely empty or full, there is no new curve added to the curve list, but instead, the cell in the quad will refer to the first or second curve-list index, depending if the cell is full or empty, respectively.

4.2.3. Generate MIP-values

Once the entire shape is fitted inside the quad-structure, we calculate the MIP-values for every quad, so they can be packed in the final texture files. To start, we first determine a surface value for each one of the fitted curves. For the sake of simplicity, P_1 and P_2 are ignored and the surface is calculated as if there was a straight line between P_0 and P_3. This is a valid approach because, in the bigger scheme, all these values will be averaged over, and the deeper a curve is inside the quad-tree, the less it contributes to the final average of a quad that lies higher in the tree.

Since P_0 and P_3 can be on any of the four edges of the cell they are in, they are first made rotation-invariant. This is done by rotating both points 90 degrees until P_1 is located on the (arbitrarily chosen) top edge. The equation to calculate the area A under the curve now only depends on the edge that P_2 is on. All six of these scenario's and their equations to calculate the curve's area A are detailed in Figure 4.3.

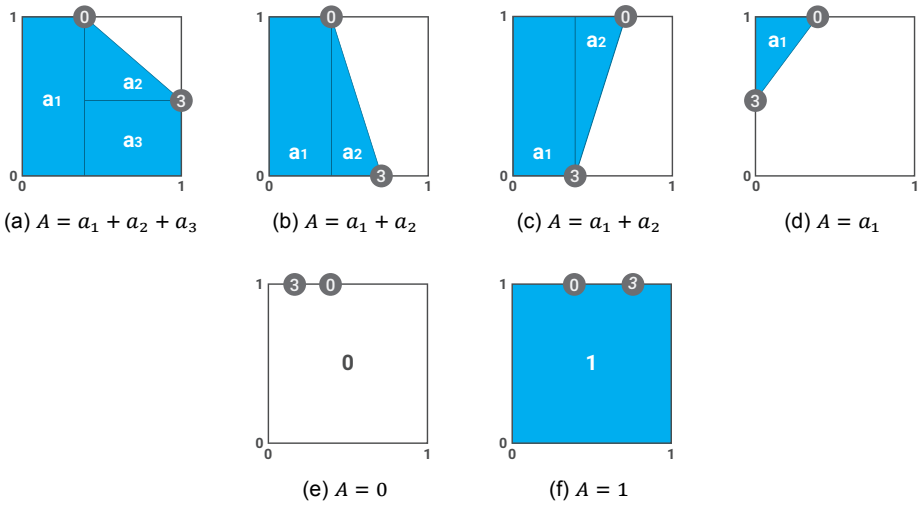


Figure 4.3: All scenarios for P_0 and P_3 and equations for their MIP-value A

Now that all curves have a corresponding value that represents the average surface of their cells, we turn to the quads. These quads were also stored in a list, with each of their four cells having a reference to the index of either another quad or one of the stored curves. The quads are handled from the bottom up to make sure that every quad can average over four already calculated MIP-values of its cells. The most bottom quad contains four curves, and can therefore simply average their surfaces. Higher up on the list, the quads' cells also point to other quads, which might not have their MIP value calculated yet.

4.2.4. Data model normalization and improvements

Before the data is written to the final texture files, it needs to be normalized. All values are stored in the separate color channels of the texture image files and will be read in the fragment shader as a value between 0 and 1. To ensure that the encoding and decoding scales, independent on the number of bits that the texture file has for a single color channel, normalization of the values in both the quad- and curve lists is required. When a list contains less than 256 curves or quads, this fits well in an 8-bit image file, where each color channel has 8 bits of space to define a number. When the list size increases however, in order to be able to identify unique indexes, a 16-bit image needs to be used to store these indexes. When these color values are read in the fragment shader, this gets returned as a value between 0 and 1, regardless of the number of bits per color channel there were in the actual texture file. This means that when all values are normalized, the actual data files can increase their bit depth per color channel when required, without needing any change in fragment shader code. Small shapes with less than 256 quads or curves can be defined in an 8-bit image file, where more complex shapes can use up to 65.536 unique indexes that 16-but image files provide. Alongside this, in another effort to save memory in storage and bandwidth, some optimization is performed in the way the control points P_0 and P_3 are normalized.

Edge point normalization

All four control points of a curve are defined by an x- and y-coordinate, based on the normalized coordinate system inside the curve's cell. Though, we know that P_0 and P_3 by definition always lay on the edge of its cell. This means that we can represent this 2-dimensional value with a single value that indicates how far that point is from a defined starting point, moving along the edge. The starting point and direction are arbitrarily chosen to be the top left of the cell and clockwise, respectively. This results in a single value between 0 and 1 that is easily be converted back to an x,y-coordinate. This conversion is illustrated in Figure 4.4 and Table 4.1.

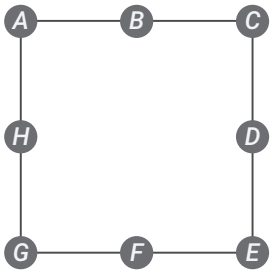


Figure 4.4: Eight example points on a cell's edge

Point	x	y	index
A	0	1	0.0
B	0.5	1	0.125
C	1	1	0.25
D	1	0.5	0.375
E	1	0	0.5
F	0.5	0	0.625
G	0	0	0.75
H	0	0.5	0.875

Table 4.1: Edge index values for points in figure 4.4

Control point normalization

The other half of each curve's control points, P_1 and P_2, cannot be reduced to a single value, but both their x- and y-values have to be normalized to a value between 0 and 1. This requires a minimal and maximal value for both the x- and y-axis to scale the control point's x- and y-value on. To obtain these maximum values, we look at the most extreme condition in which the cell's curve fills as much of the cell as possible. This is the case when both P_0 and P_3 are very close together and the other to control points P_1 and P_2 are positioned in such a way that the curve almost touches the two opposite edges of the cell. This is illustrated in Figure 4.5. This same boundary is reached in the case that is illustrated in Figure 4.6. Here, both control points P_1 and P_2 are at the maximum value, with the curve almost touching three of the cell's edges.

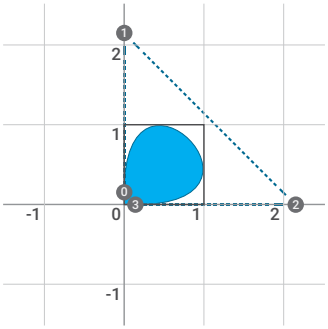


Figure 4.5: Example of extreme case with P_0 and P_3 in a corner

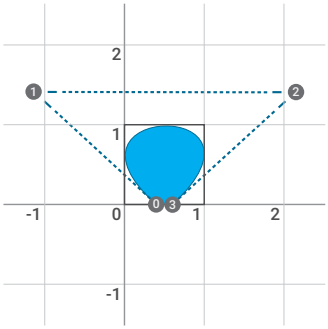


Figure 4.6: Example of extreme case with P_0 and P_3 on the edge

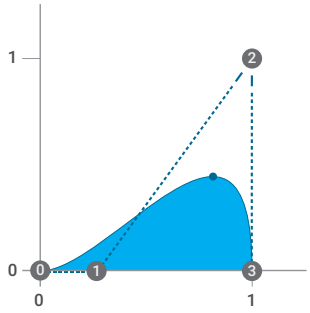


Figure 4.7: Situation with only P_2 contributing to $B(t)_y$

To compute the value of this boundary we use the scenario as illustrated in Figure 4.7. We set the y-value of all control points at 0, except $P_{2,y}$, which get set to 1. Next, we take Equation 3.2 for $B(t)_y$ and fill in these y-values, simplifying it to the following equation:

$$B(t)_y = 3(1-t)t^2 \tag{4.1}$$

When we find the maximum value of $B(t)_y$, the highest point of the curve in Figure 4.7, we find the following:

$$\max[B(t)_y | 0 \leq t \leq 1] = \frac{4}{9} \text{ at } t = \frac{2}{3} \tag{4.2}$$

Therefore, the point where this curve would almost touch the top of the cell is found by dividing $P_{2,y}$ by $\frac{4}{9}$, resulting in $\frac{9}{4} = 2.25$. This is the maximum value that a control point can move from an axis until the curve touches the cell's edge. Given this value, the range that produces valid curves for both the x- and y-values of P_1 and P_2 is [-1.25, 2.25]. This corresponds exactly with the results in Figure 4.5 and 4.6.

Quad index normalization

As mentioned before, because all values need to be normalized before they are stored in a texture file, this also applies to the indexes for each of the cells in the quads list. Before this step, the values are integers that refer to a specific index in the list of curves or quads. To normalize this, each cell value gets scaled with the size of one of the two lists, depending on if refers to a curve in the curve list, or to another quad in the quad list.

4.3. Writing serialized vector data to texture files

In order to store the optimized and normalized quad- and vector-data, we write these to image files. This data consists of two lists, one containing definitions of all quads and the other for the definitions of all curves. Each quad is defined by 9 variables. A boolean and index value for each of the four cells and a MIP-value that acts as a summary of the surface of the shape inside this quad. Each curve is defined by 6 variables. A normalized edge location for control points P_0 and P_3 and an x- and y-value for the other two control points P_1 and P_2. This section discusses the file format that was chosen as a container for these values, and how they are organized in the different color channels that are available within these files.

4.3.1. File format

Common file formats that are used to store raster-based data are the BMP, JPEG, PNG and TIFF format. JPEG is a lossy format, optimized for low file sizes with little loss of overall quality. This does mean that individual pixels in that image will be adjusted significantly, with no method of recovering that 'lost' data. The other three file formats do support lossless storage of pixel data and would, therefore, be valid containers.

TIFF would be a nice format to use since it is very commonplace in the GIS community. Unfortunately, there is little to no support for this format in the three.js framework and it is therefore discarded.

BMP would be most accurate since values are stored per-pixel. This does mean an increased file size, something that the other formats try to solve by using different encodings. It is just as supported as the PNG format, but is limited by 8-bit color channels, where the PNG format supports 16-bit color channels, allowing for lists sizes up to 65.536 curves or quads, instead of the maximum of 256 quads or curves in 8-bit channels. Because the quad and curve data is normalized, both of these bit-depths are interchangeable, as long as the number of quads or curves does not exceed 256.

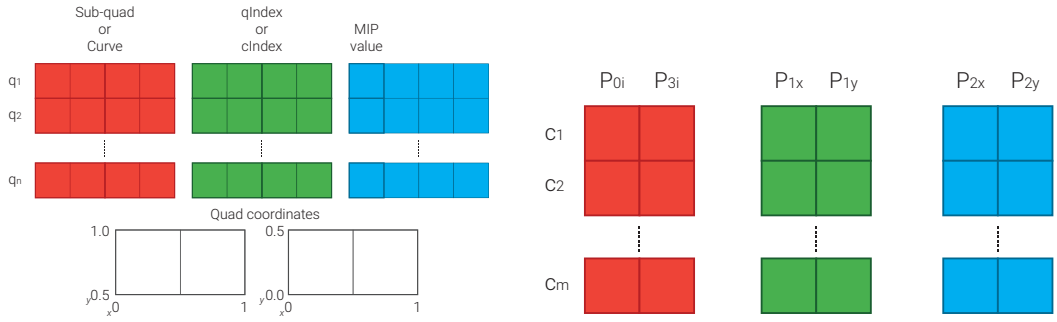
Since PNG offers a slightly smaller file size than BMP and the compatibility with larger curve lists that come with more complex vector shapes, this is the chosen file format for this work. Since most example shapes in this work stay under 256 quads and curves, we use 8-bit PNG files to store the quad and curve data.

4.3.2. Placement of variables

When placing variables inside a texture file, one thing we take into account is the ease of deserialization, meaning the ease in which shaders can read out the values from the files. Since it is very unlikely that the list of curves has the same length of as the list of quads, each list is stored in a separate file. This section details how the different values for each quad and curves are stored in the different color channels of both image files.

List of quads

The variables for each quad in the list are placed in such a way that it both resembles the list-structure, as well as provide ease of coordinate conversion from the original fragment coordinate to the final texture coordinate in the correct quad from the implicit tree structure. This is done by splitting up the square area into a top and a bottom part and positioning these next to each other. This is illustrated in Figure 4.8a. Each row represents a single quad, with the variables spread out over the different color channels. The Red channel stores if each of the four cells in the quad refer to another quad or to a curve value. The Green channel stores the respective index to this quad or curve. Because each pixel row represents a different quad, there is no wasted space here. Finally, the Blue channel is used to store a MIP-value, representing all the data that is inside this quad.



(a) Quad texture (b) Curve texture

Figure 4.8: Placement of quad- and curve values in the Red, Green and Blue channels of their texture pixels

List of curves

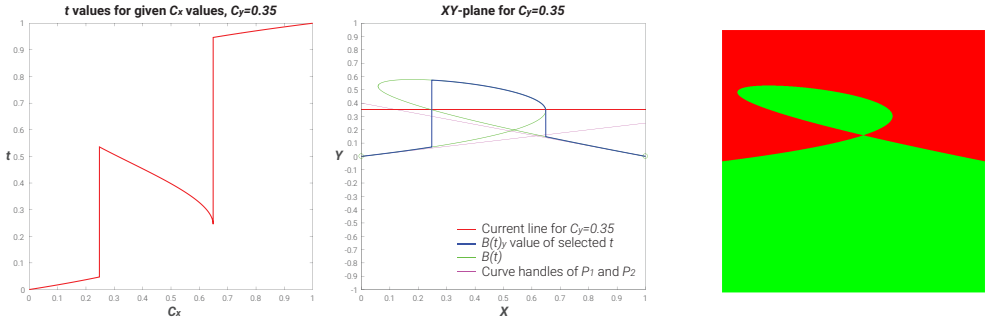
The curve texture follows the same concept, one row per curve. All additional information per curve is stored in the columns. The different color channels are used to keep related information close together. The Red channel holds the edge index values of P_0 and P_3 and the Green and Blue channels hold the normalized x- and y-coordinates of P_1 and P_2, respectively. This is illustrated in Figure 4.8b.

4.4. Reading and rendering vector data

Values that are read from each of the texture channels are also returned by the GLSL fragment shader as normalized values, between 0 and 1. This makes it irrelevant if these originally came from an 8- or 16-bit texture file. After converting these values to their required ranges, the curves can be rendered to the display. The variety of possible curves is shown first, after which some points are made on efficient parallel programming.

4.4.1. Curve rendering

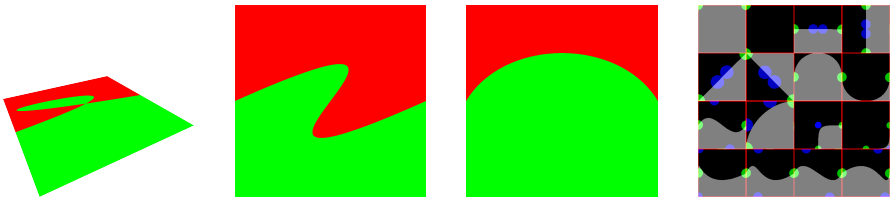
To verify that the correct curves are rendered to the display, equivalent code was run in Matlab. This allows for more control over the debug process. Specific values for P_0, P_1, P_2 and P_3 were selected, rendered in a Matlab graph and compared with the WebGL version that had the same hard-coded points. These WebGL points were also animated with sinusoidal variations depending on a time variable, while monitoring the output for unexpected behavior, to verify that no edge cases were left unattended. Figure 4.9 shows the comparison between the results as calculated in Matlab and through WebGL. Figure 4.9a represents the relevant t-values and corresponding height for $C_y = 0.35$. Figure 4.9b shows the result of the actual fragment shader. Figure 4.10 shows the variety of cubic Bézier curves that can be defined and rendered.



(a) Matlab results for $C_y = 0.35$

(b) WebGL result for entire texture

Figure 4.9: Matlab results that verify the WebGL results



(a) Rotated WebGL result

(b) Different P_1 and P_2

(c) Different P_1 and P_2

(d) Different P_0 and P_3

Figure 4.10: More WebGL curves, showing the input flexibility

4.4.2. Loops in parallel programming

Because of the parallel nature of programming in GLSL, there are specific things that need to be taken into account, that work differently when compared with conventional serial programming. These concepts mainly concern working with loops.

To read the exported texture files in the custom written fragment shader, we simply keep going deeper in the quad hierarchy until a cell returns an index for the curves texture. This kind of structure poses a problem for the specific environment that was chosen for this implementation. The used three.js version uses WebGL to render content to a webpage. This version of WebGL does not allow *while*-loops and recursion in its programming language

GLSL. This can be replaced by a *for*-loop that loops maximally once for each quad (this is the worst case), but *for*-loops in our version of GLSL have the limitation that the stop condition (the number of quads) cannot be a variable. This has to be a value that is static for the shader, regardless of input.

This problem is solved by manually generating the shader code after the input textures are read and their size is determined. The number of quads and curves is now injected into the shader code as constants before it is passed to the graphics card. This allows the *for*-loop to run with the maximal length of the worst case. When a curve is found, however, the loop pauses and does not continue for the number of quads. When all threads that run this block of code have found their curves these values are returned without completing the full loop.

4.4.3. Anti-Aliasing

Normal MIP-mapping takes advantage of a set of shader functions called *dFdx* and *dFdy* to calculate the difference in a variable between two neighboring pixels in both the x- and y-direction, respectively. This variable is often the current texel's coordinate, to figure out how many texels are squeezed into a single display pixel. The change that is calculated is then used to calculate a value called the Level-of-Detail, and directly relates to the resolution in the available MIP-map where the color information needs to be obtained. Since our vector representation does not have a discrete texel resolution and different parts of the texture having different depths, this needs to be adapted a little to find the correct depth in which to obtain the MIP-value, when necessary. The depth distribution of an example is visualized in Figure 4.13, where a brighter red means a deeper section of the quad-tree.

Figure 4.11: Satellite image for the example shape

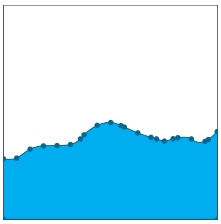
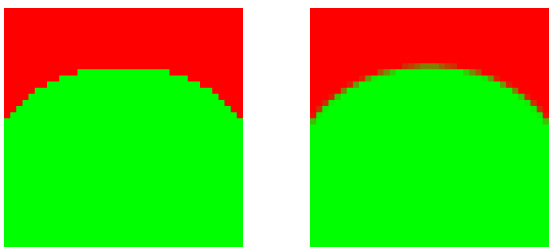


Figure 4.12: Vector shape for the example

Figure 4.13: Example shape tree depth distribution

Supersampling

Regular 4 by 4 supersampling has been implemented in the middle of the shader code. Since the quad-tree traversal is the same for all 16 locations, it would be a waste of resources to do this 16 times, with the same outcome each time. The sample coordinates are taken inside the cell, just before they are all normalized to the axis that is defined by the curve's control points P_0 and P_3. This results in 16 coordinates of which the t-values are calculated, which are then converted to the height of the Bézier curve and compared with the normalized height of each of the corresponding sub-sample coordinates. This results in 16 binary in-out values that are then averaged to produce a smoothly rendered curve, as shown in Figure 4.14.



(a) Without anti-aliasing

(b) With 4x4 supersampling

Figure 4.14: Difference in edge smoothness with and without supersampling

MIP-maps

Before the quad-tree-depth loop is started, we calculate the difference in texture coordinates between neighboring display pixels using *dFdx* and *dFdy*. The coordinate is not multiplied by the texture resolution, something that is normally done in normal Level-of-Detail calculations but remains in its normalized state. This coordinate difference is then used, while traversing every depth of the quad-tree, to calculate if the current quad completely fits inside the current fragment's display pixel. This is done by calculating how many times the distance between the neighboring fragment coordinates fits in the size of the quad. The implementation is as follows:

$$\text{timesBigger} = \text{fragmentUVDistance} \times 2^{\text{currentDepth}}$$

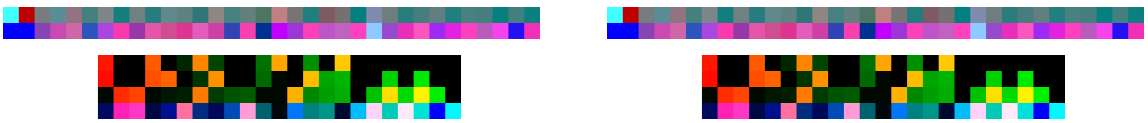
If *timesBigger* is greater than one, the loop gets broken, and the MIP-value of the current quad gets returned. Otherwise, the loop continues going deeper into the tree until it either returns a deeper MIP-value or the curve information at the last branch of the tree. To visualize this for the example, we 30x exaggerated the point where MIP-values would be used instead of the actual curve values. This means that when a quad completely fits inside 30 by 30-pixel cell, the tree-loop ends prematurely, returning the MIP-value for that cell. Figure 4.15 shows this result for five different display resolutions, where the green and red rendered areas are fragments that are calculated from the actual curve information, and the gray squares are the MIP-values that were returned from the loop. Lighter MIP-values mean a higher percentage of that cell is filled with the shape.

(a) 480x480 pixels (b) 240x240 pixels (c) 120x120 pixels (d) 60x60 pixels (e) 30x30 pixels

Figure 4.15: 30x Exaggerated MIP-value results for multiple display resolutions. Gray squares are MIP-values, returned when a quad was smaller than 30 by 30 pixels. Red and green are direct in-out values, dependent on the available curve data on that texture coordinate.

4.5. Final texture files

The discussed implementation converts a shape's vector description to a list of quad and a list of curves. These are then stored in individual texture files. For the last flood-model example in the last chapter, having 23 quads and 34 curves, this resulted in the two PNG texture files that are shown in Figure 4.16 for both the 8-bit and 16-bit version. Visually these look the same, because of the normalization of the values, but the 16-bit version is effectively less dense. The current use-case can easily use the 8-bit version since a number of curves or quads does not exceed 256. If this would be the case, the 16-bit version would need to be used. This switch does not require any change in shader implementation. Both of these are rotated 90 degrees counter-clockwise to better fit this page. The rendered comparison of this vector shape is shown at the end of the next section.



(a) 8-bit texture files (602 B) (b) 16-bit texture files (933 B)

Figure 4.16: Example PNG texture files with their size for both the 8- and 16-bit version (Rotated 90 degrees counter-clockwise

4.6. Specific flood-model visualization implementations

To use a real-world example where this technique can be used, we take a specific square area in Santa Cruz, CA, represented by two input files. The first is a satellite image of this area in PNG format and the second is a vector representation of the body of water that can be seen on this satellite, encoded in an SVG file. Both these files are shown in Figure 4.11 and 4.12.

The use-case, as proposed by Deltares, is described as a “lifelike” ocean, overlaying a satellite image, with a sharp boundary between land and see, regardless of the zoom-level. This required the addition of an ocean shader and an animation technique called *Texture Folding*.

4.6.1. Ocean shader

Instead of using a solid color to represent the ocean surface, we extend an existing ocean shader that is included in the three.js framework. We inject custom shader code that runs after the shader surface has been calculated and determines a transparency value, depending on the vector shape that is determined by the vector textures. This concept is illustrated in Figure 4.17.

4.6.2. Texture folding

Another effect that was added to improve the ocean surface’s “realism” at a more zoomed in perspective, is known as *texture folding*. This technique was initially proposed by [25] and uses a similar method of storing animation data in textures, so they can be used in the graphics pipeline. This animation data is then used in a layer between the initial texture coordinates for each fragment and an adjusted texture coordinate that takes into account the movement information that is stored in these folding textures.

It consists of a single texture, using the texel color channels to store end-positions, interpolation types and timing information for each of the pixels in the texture raster. The original texture coordinate for a fragment gets then modified, depending on the folding interpolation that is set for that coordinate. This then gives the illusion of the ocean’s shoreline sliding over the satellite image, as if waves roll in and out of the beach.

The original technique allows for multiple interpolation variants, including periodicity and smooth starts and finishes, but for the current use-case only a rough, periodic tidal movement is used.

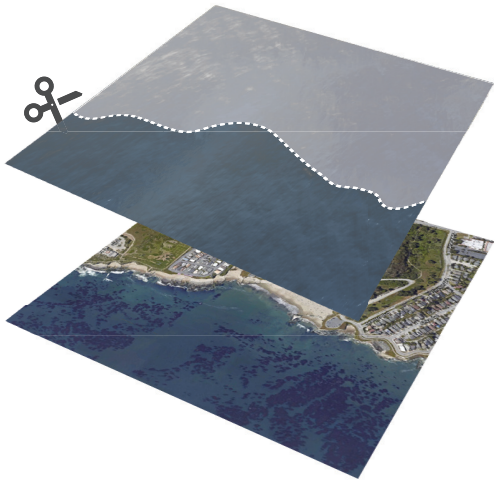


Figure 4.17: Using the vector shape as a mask to combine both the ocean shader and the satellite image

5

Results and Examples

This chapter presents the results that were obtained while solving the question posed in Chapter 1 with the method that was detailed in Chapter 3 and implemented as described in Chapter 4. First, we show metrics from different vector shapes, and their converted results. This is then followed by a visual side by side comparison of vector shapes, showing a default rendering, the render result of our method, the quad structure that our method generates and the actual texture files.

In regular vector shapes, the position of the vector splines does not change anything in the data structure. This is not the case for the proposed vector textures, because of the exponential subdivision of the shape's viewbox. This means that a small change in a shape's position or scale can have a large effect on the converted data structure. To demonstrate the that our proposed technique can handle these situations, more complex examples are shown as well. All examples were rendered on a MSI GE63VR 7RE-Raider laptop, with an FPS between 111 and 115, very close to the laptop's maximum refresh rate of 120 FPS.

5.1. Vector shape conversion

To compare the difference between generated vector textures, we created a small variety of the same shape and altered its scale, position and the amount of these shapes. As a baseline, we take the shape that is shown in Figure 5.1. This shape has both smooth and sharp corners, and is not too big, consisting of only 5 vector segments. Table 5.1 shows a number of segments, the conversion time and some details of the resulting vector texture. Because neither the number of curves nor the number of quads exceeds 65.536 (2^{16}), we only need to compare the 8-Bit file size of the resulting textures with the original vector file size to see that there is no significant memory overhead after converting the chosen examples. The examples are ordered based on the number of segments that described the original vector shape. This gives a rough indication of their complexity.

	Original shape		Conversion	Converted files		Converted file size		
	#Segments	File size	Time (ms)	#Curves	#Quads	8-Bit	16-Bit	FPS
Simple small	5	699 B	53.1	18	13	414 B	601 B	113
Simple big	5	696 B	70.7	21	14	445 B	639 B	115
Simple offset	5	691 B	98.0	19	12	416 B	625 B	115
Simple duo	10	883 B	187.5	36	23	617 B	964 B	114
Simple quartet	20	1269 B	382.9	71	47	1006 B	1727 B	113
Duck	21	1277 B	300.0	61	34	866 B	1462 B	113
Thumb	22	908 B	236.8	45	26	703 B	1142 B	114
Flame	23	1418 B	523.1	109	56	1306 B	2334 B	112
Cake	31	1283 B	536.1	115	70	1422 B	2496 B	111
Font	42	1426 B	645.4	121	64	1447 B	2495 B	113
Ocean	31	1817 B	3632.2	90	59	1205 B	2115 B	114

Table 5.1: Comparison of multiple shape conversions

5.1.1. Simple examples

We start by presenting five variations on a simple shape, saving the more complex shapes in the next section. This simple shape has been chosen because it is one of the most simple shapes with both smooth and straight corners. Starting from Figure 5.1, the succeeding figures show what happens to the subdivided data structure, and the final texture size, when the original shape: changes in size (5.2), changes in position (5.3) and shares the space with multiple other shapes (5.4 and 5.5).

Each example shows a rendering of the original vector shape in the top left, with the final rendered result in the bottom left. The middle shows the subdivision of space, with the added background colors of each cell that represent its MIP-value. Lastly, the right side shows both texture files that are fed to the fragment shader which renders them as the bottom left result.

Sharp edges

Something that is immediately clear from the subdivisions is that when a shape contains sharp edges, these are approximated with a deeper subdivision of space than necessary. Figure 5.1 and 5.2 show vertically symmetrical shapes, but this is not reflected in the computed subdivision. More on this in Chapter 6.

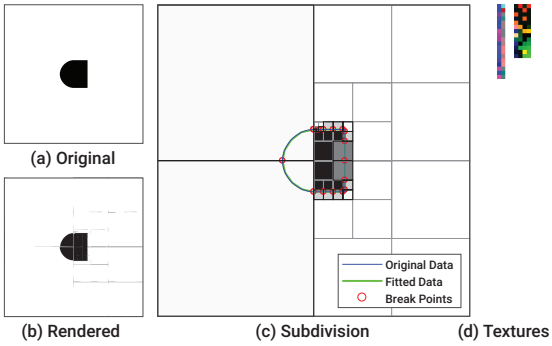


Figure 5.1: Simple small

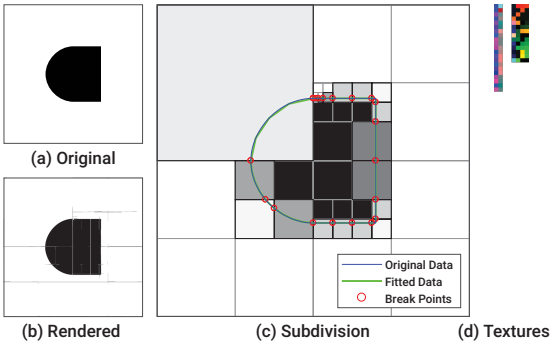


Figure 5.2: Simple big

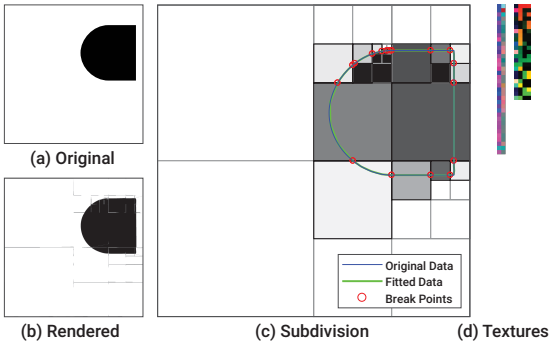


Figure 5.3: Simple offset

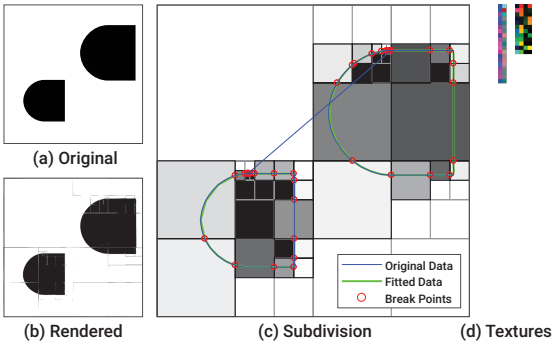


Figure 5.4: Simple duo

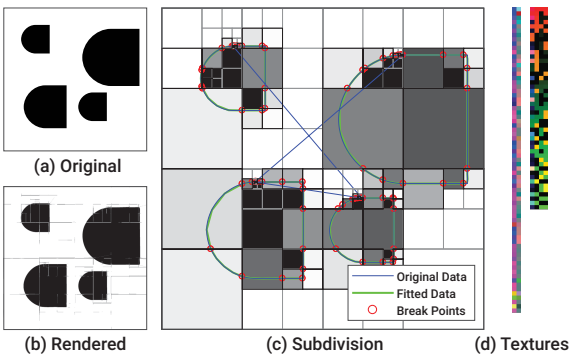


Figure 5.5: Simple quartet

5.1.2. Complex examples

The proposed technique can, besides full, single-edged shapes, render more complex shapes containing multiple compound shapes, that will act as holes in the parent shape, as well as varying degrees of detail over the entire shape. This is illustrated with five examples, as shown in Figure 5.6 through 5.11. Figure 5.6 shows the conversion results of a manually created vector shape. The second three figures, 5.7, 5.8 and 5.9 were obtained from the Internet. Figure 5.10 was created from the Helvetica OpenType font and lastly, Figure 5.11 shows a representation of part of the coastal ocean a little southwest of Santa Cruz, CA.

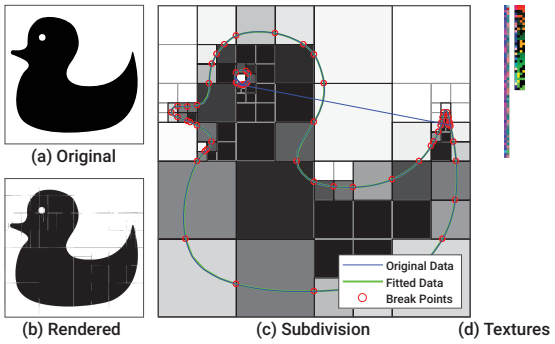


Figure 5.6: Duck

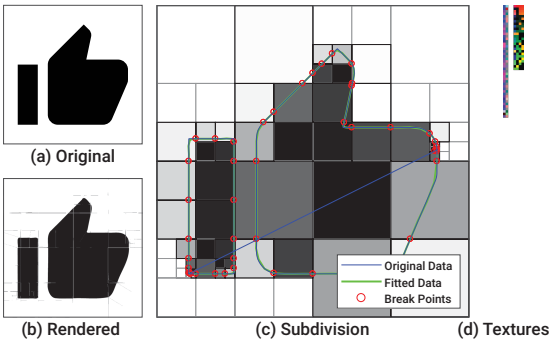


Figure 5.7: Thumb

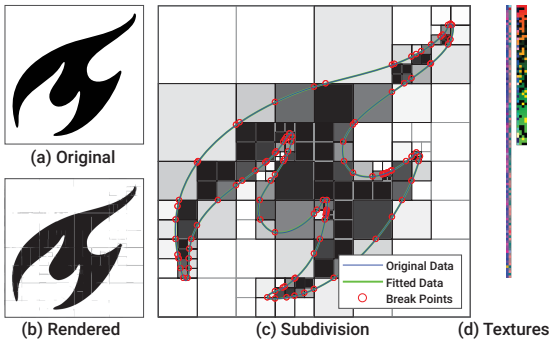


Figure 5.8: Flame

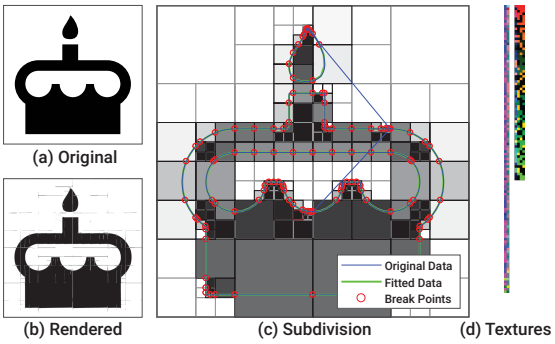


Figure 5.9: Cake

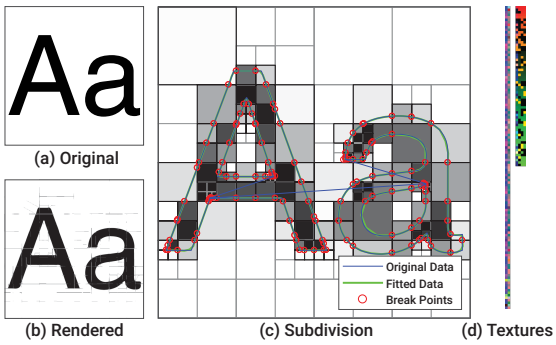


Figure 5.10: Font

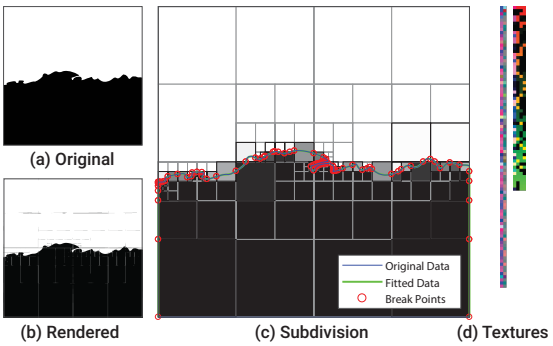


Figure 5.11: Ocean shape

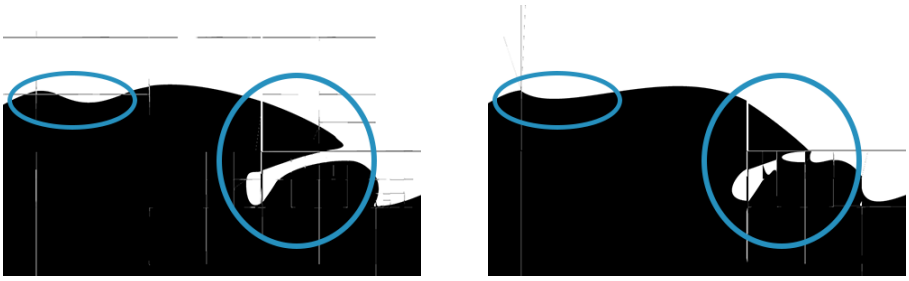
5.2. Vector texture rendering

As shown in the figures before, the fragment shader is capable of rendering a big variety of different vector shapes. Though, there are some visual artifacts that are visible in some rendered results that should be addressed. Next, we show the results of the Deltares-specific implementation, that uses the shape of Figure 5.11 and adds more data to end up with a “lifelike” and interactive visualization.

5.2.1. Visual artifacts

Choosing an allowed error for fitting curves to the originally sampled data might result in incorrect results, as shown in Figure 5.12. It shows both the rendered results of an allowed fitting error that produced a correct conversion and a more loose conversion where the allowed error was 10 times as high. In order to preserve the entire smallest defined detail, the allowed error should be based upon the scale of that smallest detail.

A significant artifact that is always visible in the current implementation are the horizontal and vertical lines near the cell edges. These are not the result of the proposed method, but a side-effect of the render precision that three.js can provide while traversing down the quad-tree.

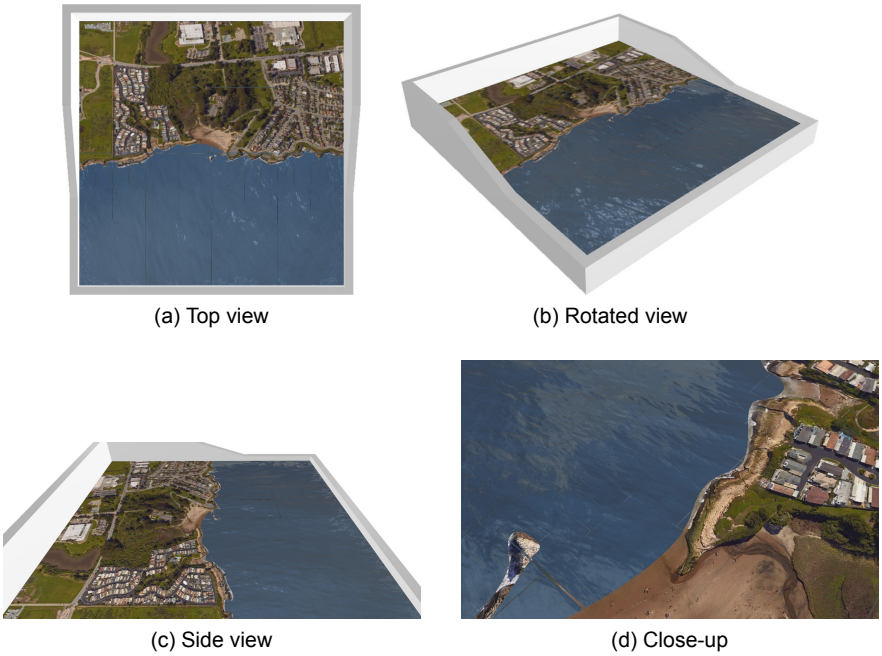


(a) Default fitting error (b) 10x allowed fitting error

Figure 5.12: Close-up rendered results where the allowed fitting errors become large

5.2.2. Flood-model visualization example

The specific flood-model use-case involved rendering a “lifelike” ocean with a sharp edge on top of a satellite image. This surface is then put inside a 3D-modeled box, to give similarity to their *Sandbox* project. The following rendered visualizations use the shape that was introduced in Figure 5.11, with the default allowed error, resulting in the graphics of Figure 5.13.



(a) Top view (b) Rotated view

(c) Side view (d) Close-up

Figure 5.13: Results for default allowed error

Given the time and implementation limitations, there is a lot that can improve the current implementation. This is all discussed in Chapter 6.

6

Discussion

The results show that resolution independent vector shapes, defined inside a quad-structured hierarchy and stored in rasterized texture files, can be rendered with the graphics pipeline using only a single pass. This also included vector shapes that have holes inside of them. Given the presented results, it is clear that there are still some aspects that this method can improve upon. This includes the method of representing the vector shape, the current conversion from vector shapes to their texture representation and the rendering of these textures to the display.

6.1. Method

The results show that the method is not perfect in representing all possible vector shapes as vector textures. Currently, there are artificial limitations set in the definition that could be loosened, in order to get a more accurate representation of any original vector shape with less data.

6.1.1. Control normalization

Currently, the control points P_1 and P_2 are clamped between an x-value of 0 and 1. This means that loops can never be defined since this requires that P_1 and P_2 move outside the 0-to-1 range for their x-value. This is illustrated in Figure 2.3. Loosening up this range from -1-to-2 would allow for a wider range of curve definitions. There would be no need to change anything in the shader since it is already capable of rendering looping curves.

6.1.2. Sharp corners

Something that becomes clear when looking at the subdivided results, is that in the case of sharp corners, the cubic Bézier fitter attempts to approximate this with a curved corner, and will keep subdividing the area of this corner until it reaches the maximally allowed fitting error. This is a waste of definition space and conversion time, compared to the concise shape definition of this part in the original vector shape. This can be seen in most figures in Chapter 5.

This could be solved by extending the definition of the quad texture. Currently, this contains a single boolean to describe if a quad-cell refers to another quad, or to a curve. If this field would allow three values, a sub-quad, a curve or a straight corner, it would prevent the conversion algorithm to keep unnecessarily sub-dividing the quad.

The straight corner itself can be defined by 4 values, as well. Two edge-indices and an x- and y-value for the coordinate of the corner inside the cell. This would fit inside the curve-texture, and could just be read out differently by a fragment shader, depending on the definition in the quad-texture.

Rendering a straight corner in the fragment shader also doesn't pose a problem, since it only has two cases, depending on the y-value of the corner coordinate, when its basis is changed to a coordinate system that is built upon the two edge coordinates. The in-out test

then boils down to checking if the fragment coordinate, which fewer to the new coordinate system, falls inside one of the four quadrants, divided by the system's axis.

6.2. conversion

The current conversion was done in Matlab with example specific settings and some off-the-shelf code to fit cubic Bézier curves to a set of points. Both of these can be improved.

6.2.1. Interpolation

Right now the examples are first interpolated based on the resolution of their defined viewbox. This could be improved by either finding the smallest curve segment in the original shape definition and basing the interpolation density on that, or by not interpolating the curve segments, but instead cutting them up into smaller segments on the points where they cross the edge of a subdivided quad's cell.

The subdivision algorithm would then first check if there is only a single path segment in a cell and if so, use that to define the curve or straight corner in the cell. Otherwise, if all path segments are connected to each other, interpolate them with a specific density depending on the resolution of the cell and check if all these path segments can be represented as a single curve or straight corner, with a given error. This error might also depend on the size of the cell. If all these cases fail, subdivide the cell further and try again for each of the sub-cells.

This way, more of the actual shape information would translate directly into the vector texture, and it would become less of an approximation.

6.2.2. Unnecessary subdivisions

The current off-the-shelf fitting code that was used does not produce consistent results. Looking at the difference between Figure 5.1c and 5.2c, the lower left quadrant is described as a single curve in the first example, but has been subdivided multiple times in the latter example. The original vector shape is defined by a single curve in that quadrant, so there is no reason why this couldn't extend to its vector texture representation. Especially since it does allow the top left quadrant of the latter example to be represented as a single curve.

6.2.3. MIP-values

The generation of the MIP-values is currently done quite coarsely, by ignoring the control points P_1 and P_2. This could be improved by using the concept of MultiSample Anti-Aliasing and taking the average of 16 samples of each specific curve. This would give a more accurate MIP-value of a curve.

6.3. Render artifacts

Even though the rendered results have a very close resemblance to the original vector shape, the current three.js implementation does show some visible and unwanted artifacts. These mainly consist of the horizontal and vertical lines that can be seen in every rendered result, but most clearly in Figure 5.12. They represent gaps in precision when traversing down the quad-tree when looking for the curve information at the end of a tree branch.

6.4. Working examples

One of the examples that fails to render and convert is the case where path segments intersect. The behavior in these cases is not standardized, but a common solution is the even-odd rule, where areas are inside or outside the shape depending on the even or odd amount of borders that need to be crossed to reach them fro the outside of the shape. The current representation should not have any issues with this since global shape information is stored at local coordinates. The only area's that would be badly defined by the current method are near the actual intersection points. Looking at the subdivision pattern with sharp corners, these points can be made infinitely small, depending on how small the error should be.

7

Conclusions

To conclude this work, we answer the question that was posed in the introduction:

"How can we store vector shape descriptions in such a way, that dedicated graphics hardware can efficiently render them in a single, parallel pass at any given pixel resolution?".

The proposed solution answers this question by enabling random-access rendering for vector shapes, without having to take the entire shape into account. Random-access means that any display pixel can be rendered in parallel, without the prerequisite that other display pixels are rendered in advance.

We do this efficiently by providing the fragment shader with only the relevant data for a specific display pixel. This is achieved by re-distributing and storing the relevant vector data. The vector shape is split up into a quad-tree of piece-wise sections of the shape's boundary, taking into account the amount of vector detail that each boundary section has. This data is then represented as a list of quads and a list of curves, that are each stored in their own texture file, the required data container of fragment shaders. Some compression is applied to these texture files, in an effort to save network- and shader-bandwith. Finally, a custom fragment shader then reads these textures and renders the complete vector shape to a display, using random-access.

The results show that the proposed technique can render a wide range of vector shapes, without the need of pre-rasterizing the vector shape for each rendered frame. These rendered shapes are visually equivalent to renders where the shape is computed with a normal winding-rule test, while also being slightly smaller in file size.

In the current state, both the conversion and shader implementation are not good enough to be used in a production environment. The conversion implementation has not been optimized to convert all possible SVG-defined vector shapes to their texture representation and the shader implementation suffers from visual artifacts that can be attributed to the used framework and should not be seen as a limitation of the proposed technique.

A wide adoptation would require easy creation and manipulation possibilities of vector graphics in this format. The format's current form is not directly compatible with the wide range of customer solutions for creating and editing vector graphics, like Adobe Illustrator, Inkscape or Sketch. This would need to be addressed.

However, the technique does prove that random-access rendering of vector shapes by converting vector shape descriptions and storing this in texture files can be done efficiently and by adding the improvements that are listed in the discussion chapter, this technique could could grow into a versatile base for future vector rendering techniques.

Bibliography

[1] S.N. Bernshtein. Collected works, 1952.

[2] L. Paul Chew. Constrained delaunay triangulations. *Algorithmica*, 4:97–108, 1989.

[3] P. Cochrane. Virtual mathematics. *The Mathematical Gazette*, 80(488):267—278, 1996.

[4] Franklin C. Crow. Summed-area tables for texture mapping. *SIGGRAPH Comput. Graph.*, 18(3):207–212, January 1984. ISSN 0097-8930. doi: 10.1145/964965.808600. URL http://doi.acm.org/10.1145/964965.808600.

[5] Jason Davies. Animated bézier curves. https://www.jasondavies.com/animated-bezier/, 2010. Online; accessed 28-July-2017.

[6] Martin Eisemann and Marcus A Magnor. Zipmaps: Zoom-into-parts texture maps. In *VMV*, pages 291–297, 2010.

[7] Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. *Scalable vector graphics (SVG) 1.0 specification*. iuniverse, 2000.

[8] Sarah F Frisken, Ronald N Perry, Alyn P Rockwood, and Thouis R Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 249–254. ACM Press/Addison-Wesley Publishing Co., 2000.

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

[10] H. W. Jensen, A. Keller (editors, Jack Tumblin, and Prasun Choudhury. Bixels: Picture samples with sharp embedded boundaries.

[11] Oliver Kersting and Jürgen Döllner. Interactive 3d visualization of vector data in gis. In *In ACM international symposium on advances in geographic information systems (GIS) (2002*, pages 107–112.

[12] Dr. M. Khan. cubic bezier least square fitting. https://nl.mathworks.com/matlabcentral/fileexchange/15542-cubic-bezier-least-square-fitting. Online; accessed 26-june-2017.

[13] Joe Kniss, Aaron Lefohn, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. Octree textures on graphics hardware. In *In ACM SIGGRAPH 2005 Conference Abstracts and Applications*, page 3, 2005.

[14] Yoshiyuki Kokojima, Kaoru Sugita, Takahiro Saito, and Takashi Takemoto. Resolution independent rendering of deformable vector objects using graphics hardware. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, New York, NY, USA, 2006. ACM. ISBN 1-59593-364-6. doi: 10.1145/1179849.1179997. URL http://doi.acm.org/10.1145/1179849.1179997.

[15] Martin Kraus and Thomas Ertl. Adaptive Texture Maps. In Thomas Ertl, Wolfgang Heidrich, and Michael Doggett, editors, *SIGGRAPH/Eurographics Workshop on Graphics Hardware*. The Eurographics Association, 2002. ISBN 1-58113-580-1. doi: 10.2312/EGGH/EGGH02/007-015.

[16] Sylvain Lefebvre, Samuel Hornus, Fabrice Neyret, et al. Octree textures on the gpu. *GPU gems*, 2:595–613, 2005.

[17] Charles Loop and Jim Blinn. Resolution independent curve rendering using programmable graphics hardware. Association for Computing Machinery, Inc., January 2005. URL https://www.microsoft.com/en-us/research/publication/resolution-independent-curve-rendering-using-programmable-graphics-hardware/.

[18] Diego Nehab and Hugues Hoppe. Random-access rendering of general vector graphics. In *ACM Transactions on Graphics (TOG)*, volume 27, page 135. ACM, 2008.

[19] R.W.D. Nickalls. Viète, descartes and the cubic equation. *The Mathematical Gazette*, 90(518):203—208, 2006. URL http://www.nickalls.org/dick/papers/maths/descartes2006.pdf.

[20] *HRAA: High-Resolution Antialaising through Multisampling*. Nvidia, 2001.

[21] Evgueni Parilov and Denis Zorin. Real-time rendering of textures with feature curves. *ACM Transactions on Graphics (TOG)*, 27(1):3, 2008.

[22] Zheng Qin, Michael D McCool, and Craig Kaplan. Precise vector textures for real-time 3d rendering. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 199–206. ACM, 2008.

[23] G. Ramanarayanan, K. Bala, and B. J. Walter. Feature-based textures. In *In Eurographics Symposium on Rendering*, 2004.

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

[25] Thorben Schulde. Folding textures. Master's thesis, Institut für Computergraphik, Technische Universität Braunschweig, 2012.

[26] Peter Selinger. Potrace: a polygon-based tracing algorithm. *Potrace (online), http://potrace. sourceforge. net/potrace. pdf (2009-07-01)*, 2003.

[27] Pradeep Sen. Silhouette maps for improved texture magnification. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 65–73. ACM, 2004.

[28] Pradeep Sen, Mike Cammarano, and Pat Hanrahan. Shadow silhouette maps. *ACM SIGGRAPH*, 22:521–526, 2003.

[29] Johnathan Skinner. Mip-mapping in direct3d. https://www.gamedev.net/articles/programming/graphics/mip-mapping-in-direct3d-r1233, 2000. Online; accessed 5-October-2017.

[30] Marco Tarini and Paolo Cignoni. Pinchmaps: Textures with customizable discontinuities. In *Computer Graphics Forum*, volume 24, pages 557–568. Wiley Online Library, 2005.

[31] Lance Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings*, 1983.