Vector Graphics for Real-time 3D Rendering

by

Zheng Qin

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2009

© Zheng Qin 2009

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Algorithms are presented that enable the use of vector graphics representations of images in texture maps for 3D real time rendering. Vector graphics images are resolution independent and can be zoomed arbitrarily without losing detail or crispness. Many important types of images, including text and other symbolic information, are best represented in vector form. Vector graphics textures can also be used as transparency mattes to augment geometric detail in models via trim curves.

Spline curves are used to represent boundaries around regions in standard vector graphics representations, such as PDF and SVG. Antialiased rendering of such content can be obtained by thresholding implicit representations of these curves. The distance function is an especially useful implicit representation. Accurate distance function computations would also allow the implementation of special effects such as embossing. Unfortunately, computing the true distance to higher order spline curves is too expensive for real time rendering. Therefore, normally either the distance is approximated by normalizing some other implicit representation or the spline curves are approximated with simpler primitives.

In this thesis, three methods for rendering vector graphics textures in real time are introduced, based on various approximations of the distance computation.

The first and simplest approach to the distance computation approximates curves with line segments. Unfortunately, approximation with line segments gives only C^0 continuity. In order to improve smoothness, spline curves can also be approximated with circular arcs. This approximation has C^1 continuity and computing the distance to a circular arc is only slightly more expensive than computing the distance to a line segment. Finally an iterative algorithm is discussed that has good performance in practice and can compute the distance to any parametrically differentiable curve (including polynomial splines of any order) robustly. This algorithm is demonstrated in the context of a system capable of real-time rendering of SVG content in a texture map on a GPU.

Data structures and acceleration algorithms in the context of massively parallel GPU architectures are also discussed. These data structures and acceleration structures allow arbitrary vector content (with space-variant complexity, and overlapping regions) to be represented in a random-access texture.

Acknowledgements

I would like to thank my co-supervisors Michael McCool and Craig Kaplan for their great help through all these years. Michael McCool taught me that I could do things I didn't think I was capable of doing, while Craig Kaplan showed me that there is always one more way of looking at a problem.

I would like to thank my thesis readers Philip Beesley, Justin Wan, Steve Mann and Bruce Gooch for their constructive suggestions, and for Sanjeev Bedi for serving on my committee.

I would also like to thank IBM T.J. Watson Research for giving me the opportunity to work with them.

Finally, I need to give special thanks to my family, my parents and my friends who made it all possible.

Dedication

This thesis is dedicated to my dear daughter Tong Wu, who was my constant companion and my foundation—and for her patience and independence.

Contents

List of Tables **ix**

List of Figures **xi**

1 Introduction **1**

1.1 Texture Mapping and Antialiasing 2
1.2 Representations of Images 5
1.3 Vector Images in 3D Rendering 6
1.4 The GPU 8
1.5 Goals of the thesis 10
1.6 Definitions 11

2 Prior Work **13**

2.1 Methods Using Modified Interpolations 13
2.2 Methods For Antialiased Bi-level Images 16
2.3 Methods for General Scalable Vector Graphics 20

3 Computing Distance to Curves **23**

3.1 Finding Roots of Polynomials 24
3.2 General Approaches in Finding Roots 24
3.3 Finding Real Roots 26

4 Bilevel Images and Line Segment Features **28**

4.1 Introduction 28
4.2 Outline of Representation 29
4.3 Antialiasing 30
4.4 Features and Distance Functions 32

4.4.1 Line Segments 33
4.4.2 Corners 37
4.4.3 Quadratics 38
4.4.4 Performance 39
4.5 Voronoi Grid Accelerator 40
4.5.1 Voronoi Analysis 40
4.5.2 Grid Packing 41
4.5.3 Multiresolution 41
4.6 Glyph Sprite Mapping 43
4.7 Summary 45

5 Planar Regions and Arc Features 50

5.1 Introduction 50
5.2 Arc Splines 51
5.3 Feature Representations 52
5.4 Flattening of Images 55
5.5 Interval Voronoi Analysis 57
5.6 Results 60
5.7 Summary 62

6 Layered Regions and Arbitrary Parametric Features 65

6.1 Introduction 65
6.2 Algorithm 66
6.2.1 Preprocessing 67
6.2.2 Shader Computation 70
6.3 Extensions and Applications 77
6.4 Results 78
6.5 Summary 85

7 Parallel K-Nearest Neighbor Search and an Alternative Data Structure for Vector Texture Acceleration 87

7.1 A Data Structure for Vector Graphics Data 88
7.2 The Cell Architecture 88
7.3 The KNN Problem 90

7.3.1 Related Work 91
7.3.2 Parallel KNN Search 93
7.4 Results and Future Work 99

8 Conclusion and Future Work 101
8.1 Contributions 101
8.2 Limitations and Future Work 103
8.3 Long Term Impact 104

References 105

List of Tables

6.1 Screen shots of performance tests. 81
6.2 Screen shots of performance tests (continued). 82
6.3 Screen shots of performance tests (continued). 83
6.4 Performance results. 84
6.5 Performance comparison. 85

List of Figures

1.1 Image magnification artifacts 7
1.2 Antialiased sharp boundaries in my vector image rendering 7
1.3 Diagram of the NVIDIA G8800 GTS. 9
1.4 Rendering result with my vector texture system. 11

4.1 Isotropic filtering vs. anisotropic filtering. 32
4.2 Glyph A and its distance fields 34
4.3 Errors caused by vertex ambiguity 35
4.4 Glyph D and its distance fields 36
4.5 Outline examples of glyph A 39
4.6 Voronoi analysis of some TrueType glyphs. 40
4.7 Voronoi diagram packed in a texture 42
4.8 Some examples showing embossed glyphs 43
4.9 Sprite and font tables. 44
4.10 A map using a vector texture for labels. 45
4.11 Some example “documents” with multiple glyphs. 46
4.12 Some example “documents” with multiple glyphs. 47
4.13 Closeup example “documents” with multiple glyphs. 48

5.1 An arc feature and arc spline. 52
5.2 Representation of arcs with large heights. 53
5.3 Example of line segments approximated with arc splines. 55
5.4 Flattening paths into a planar arrangement. 56
5.5 Voronoi diagram undersampling error. 57
5.6 An arc subdivides space into four regions. 59
5.7 Using interval analysis to overcome undersampling error. 60

5.8 SVG examples with boundaries approximated by arc splines. 63
5.9 SVG examples with boundaries approximated by arc splines (continued). 64

6.1 All curves can be split into simple shapes. 67
6.2 Three textures are used to store the accelerator structure. 70
6.3 Inside or outside test errors at the corners. 71
6.4 Binary search to find the closest point. 73
6.5 The normal line only intersects a simple shape once. 73
6.6 Distance computation errors. 74
6.7 Antialiasing of embossing ridges. 76
6.8 The distances are modified to support borders(a) and strokes(b). 78
6.9 Scenes using vector textures. 79
6.10 Vector texture closeups. 80

7.1 The Cell Broadband Engine (Cell BE) architecture. 89
7.2 Hilbert curves of first three orders. 94
7.3 Packing Hilbert index and coordinates of 2D point. 95
7.4 Using Hilbert indices to group points and construct tree. 96

Chapter 1

Introduction

Texture mapping is a standard approach for increasing realism in computer graphics. In 3D real-time rendering, raster image representations have traditionally been used for representing sampled textures because of their ability to support random access and the historical limitations of hardware graphics accelerators. With the emergence of powerful new generations of programmable graphics processors (GPUs), a new possibility has arisen: the use of vector image representations directly in texture mapping during rendering. The resolution-independent nature of vector image representations provides an opportunity to overcome several problems intrinsic to raster image representations.

Vector graphics images are fundamentally more precise than raster images. They represent geometric information in an exact and manipulable form that is independent of its presentation on any particular display device. Conversion of vector graphics to raster form is necessary for display on raster devices, but in a vector graphics system this conversion is performed dynamically and deferred until the last possible moment before display. Particularly for symbolic information, vector graphics are a natural representation. The use of dynamic, on-demand rasterization enables the flexible display of symbolic information in much wider variety of contexts than would be possible with pre-rasterized (and therefore lossy) representations of the same content.

The aim of this thesis is to show that efficient, GPU accelerated, real-time implementation of texture mapping directly from vector graphics representations of images is feasible and desirable.

In this chapter, I first discuss the basic idea of texture mapping, the aliasing artifacts that arise in image sampling and in particular in raster image resampling, and the traditional approaches for eliminating these artifacts. Then I discuss image representations and motivate the use of vector representations for texture mapping, as opposed to the traditional raster representations. I also briefly review the architecture of GPUs and the constraints they place on efficient implementation of random-access vector textures.

1.1 Texture Mapping and Antialiasing

In texture mapping, images known as *texture maps* are pasted onto the surface of 3D objects to provide surface detail and increase the realism of 3D scenes [25, 21, 26, 27, 9]. Traditional texture mapping uses raster images to represent textures. Raster images are regular 2D arrays of color samples. Each such sample is usually called a *texel*. Although textures and raster images are often considered synonymous, this is not necessary. For example, there are many examples of images that are evaluated procedurally in shaders to provide spatially-varying detail on a surface. In this thesis when I refer to "texture" I will be referring to the more general concept of spatially-varying detail on a surface. This detail may be represented in a variety of ways, including but not limited to raster images.

Textures are parameterized in a continuous multidimensional space, usually two-dimensional. Coordinates in this space are called *texture coordinates*. In the description of the scene, texture coordinates are often associated with primitive vertices to specify where on an object each part of the texture map should be pasted. Later, when each primitive of the 3D scene is rendered on the screen, the graphics accelerator pipeline calculates the texture coordinates for each pixel via interpolation between the primitive's vertices. At the texture lookup stage, the graphics hardware retrieves the texture sample color from the texture image using these interpolated texture coordinates. Every screen pixel that should be covered by the texture mapped objects is mapped to some position within the texture map. Note that texture coordinates are conceptually continuous. The above assumes primitives are defined by interpolation between vertices. For other forms of primitives, such as algebraic primitives, a function likewise needs to be defined that associates a texture coordinate value with every point on its surface.

When raster images are used for texture maps, they need to be resampled during rendering. Ideal raster image resampling is only possible when the pixels on the screen have a one-to-one relationship with the texels (pixels of a raster image used as a texture). In other cases, texel values have to be interpolated and filtered for high-quality rendering.

Suppose an object to be rendered is a flat rectangle and the rectangle rendered on the screen has exactly the same number of pixels as texels in both horizontal and vertical directions. Only in this special case is the texture mapping ideal. But when the object is pulled away from the viewer and its size on the screen is smaller than the texture map, the area of one screen pixel, after mapping into texture coordinates, can cover multiple texels. This is called *image minification*. On the other hand, when the object is pulled closer and its size on the screen is larger than the texture map, multiple pixels can be mapped to a single texel. This is called *image magnification*.

Image minification and magnification can also be caused by the geometric shapes of the objects because not all objects are flat rectangles. Scene objects generally will have complicated geometric shapes, and the texture maps are warped when they are pasted onto these object surfaces. Even for a flat object, continuously variable magnification ratios can be caused by perspective. Image minification and magnification can therefore happen simultaneously on the same object and within the same texture.

Each screen pixel also covers a certain physical area on the screen. When the texture mapping coordinates are computed, the position of the center of the pixel is normally used. In the perfect case, each texel corresponds exactly to the center of each pixel, and the result will be ideal. However, this is often not the case. Even if the pixels and the texels have the same resolution, the centers of the pixels may fall between the texel samples. This poses a problem of which texel should be chosen to compute the rendered pixel color (and if multiple texels need to be combined, what rule should be used).

Image minification and magnification have similar problems. In image minification, one pixel's area corresponds to multiple texels, so we need to decide whether we should use the contribution from one texel or from multiple texels. If one texel is to be used, which one? If multiple texels are to be used, how should their values be combined? In contrast, for image magnification, multiple pixels map onto the same texel cell. We need to decide how to interpolate the color for each pixel so that the enlarged texture preserves as much detail as possible but without introducing artifacts. Computing the pixel color during magnification is called *reconstruction*, since a high resolution image is reconstructed from a smaller number of texels. During minification, each pixel color must be computed from a larger number of texels, which requires reducing the total amount of information. This process is called *filtering*. Often *filtering* is used for both processes, but for clarity I will distinguish them in this document.

Reconstruction is needed during magnification. The simplest reconstruction mode is nearest neighbor lookup, in which the texel closest to the texture coordinates of the mapped pixel center is chosen. This method gives reasonably good results when the pixel resolution and the texel resolution are similar.

Bilinear interpolation is a better reconstruction method that recovers a continuous signal from a sampled signal by bilinearly interpolating the four samples at the corners of a grid into which the texture coordinates of a given pixel falls.

Nearest neighbour lookup can also be used for filtering during minification. In this case, the nearest neighbor lookup rule choses only one texel color (the one closest to the pixel center projected back into texture space) and the other texels are discarded. Unfortunately, if this filtering approach is used artifacts such as Moiré patterns, jaggies, and crawling edges can occur.

Such artifacts are called *aliasing* because technically, they are caused by the frequencies of the pixel sampling grid "beating" against the texel raster grid, causing high frequencies in the texture to "alias" or "masquerade" as low-frequency artifacts. Due to minification, the pixel grid has a lower resolution than the texel grid. As a consequence, the pixel grid cannot represent all the information in the original raster image.

Techniques to remove these artifacts are called *antialiasing*. The basic idea of all antialiasing techniques is that the high frequency information that can result in aliasing needs to be removed before sampling.

Bilinear interpolation also can help reduce aliasing during minification since it averages four samples rather than just one. This averaging reduces the troublesome high frequencies that cause aliasing. However, a more accurate antialiasing approach is to

include the contributions of all texels that are covered by the footprint of a pixel, using a weighted average to combine their values. The weights can be chosen using signal processing theory to attenuate the frequencies that can cause aliasing, although in practice the weights are usually limited to radially monotonic positive functions to avoid ringing. Unfortunately, the size of a pixel footprint may vary and the projected shape may not be isotropic. Also, the footprint can be large and averaging all the texels in it can be expensive.

Various approaches have been developed for computing the texel contributions efficiently. The most common approach, which is also implemented in GPU hardware, is MIP-mapping [66]. MIP-mapping is a multiresolution approach that prefilters the image to a discrete set of resolutions and stores the results in an image pyramid, This pyramid contains images filtered with low pass filters with different cutoff frequencies. When the scene is rendered, depending on the scale of minification, a filtered image is chosen from the image pyramid, and a simple interpolation method, such as nearest neighbor search or bilinear interpolation, can then be used to retrieve the color. If the desired scale falls between the scales for two images, then two lookups can be performed and then interpolated. One problem with MIP-mapping is that only a single isotropic scale can be used, but the footprint can be scaled anisotropically, and the footprint may also not be axis aligned. This is currently resolved in hardware by taking multiple MIP-mapped samples and computing a weighted combination to approximate the desired filter kernel [37]. This technology is sufficiently advanced that raster texture antialiasing under minification can be considered a solved problem.

However, when the image is magnified, multiple pixels are mapped onto one texel and the resolution of the pixels is higher than that of the texels. In general, trying to recover high frequency information from sampled low frequency information accurately is impossible. One solution is to represent the images as compositions of primitives represented by mathematical equations. Primitives can include (but are not limited to) lines, curves, and filled regions bounded by lines and curves. This approach, used in vector graphics, is independent of any sampling grid and provides for unlimited magnification. However, such an approach has its own antialiasing challenges. Sharp edges contain infinitely high frequencies, and sampling them precisely at only pixel centers will result in aliasing. Instead, such sharp edges need to be filtered, even during magnification, by integrating over the footprint of the pixel back-projected into texture space.

For images whose boundaries are represented mathematically, both supersampling and analytic approaches can be used to compute these integrals. To use supersampling, the image of any resolution can be computed by first sampling it at a higher resolution, and then low-pass filtering it to the desired resolution.

Analytically, averaging over the footprint of a filter is equivalent to multiplying the Fourier transform of the image with the Fourier transform of the low-pass filter in the frequency domain, or equivalently convolving the low-pass filter with the image in the space domain. Supersampling is really just a discrete implementation of the convolution. An interesting observation, which I take advantage of later, is that the result of the convolution of a radial filter with a straight edge is just a function of the distance of the edge

from the center of the filter; in other words, straight edges in 2D images can be filtered by applying a 1D function to the distance from the edge [23].

In 3D rendering, because of different transformations and texture warping involved in rendering, the footprint of a pixel can be warped and the projection in the texture map of a circular pixel may not be a circle. It could be an ellipse or more complicated shape. Accurate antialiasing should take into account the shape and orientation of the footprint, and should be oriented along the major axis of the footprint. This kind of antialiasing is called *anisotropic antialising*.

1.2 Representations of Images

In computer graphics, there are two common ways to represent an image: as a raster image (bitmap) and as a vector image. A raster image uses an array of pixels to represent an image, while a vector image uses parameterized mathematical formulae to represent image components, including contours, shapes, colors, and shading. These representations are complementary to each other. Because of the different demands of applications, sometimes it is better to use raster images, and sometimes it is better to use vector images.

Raster images can support efficient random access, but have a resolution limitation. Display devices with larger resolutions require more pixels, which results in a larger storage size. If the raster image resolution is not increased to match the display resolution (at least), then artifacts will occur. Vector images, on the other hand, use a fixed amount of storage size regardless of the resolution at which they are displayed. Vector images can be sampled and displayed with high quality at any scale. However, the usual representation of vector images does not support random access.

Many applications of vector images can be found in 2D graphics, especially for the display of symbolic information. Vector graphics were used in some of the first computer displays, including that of the US SAGA air defense system, air traffic control systems, Ivan Sutherlands Sketchpad (one of the very first interactive graphics systems), Digitals GT40, and early game systems such as Vectrex, Asteroids, and Space Wars. Vector fonts have existed for decades, and resolution-independent vector graphics document, image, and web content representations like PDF and Flash are now popular.

Existing 2D software that renders vector images usually rasterizes images using a scan line method. In scan line methods, the pixel colors are processed sequentially along the lines to solve issues concerned with antialiased colors. Because of the sequential nature of this process, vector images have not been used in 3D real-time rendering, in which pixel colors are computed in parallel. However, vector image content is in more and more demand and it would often be useful to combine this content with 3D real-time rendering. I will show that this is feasible by exploiting the rapid improvement in GPU programmability and performance. Signs, text, and labels, if represented with vector images, can be scaled with high quality, can be made more legible, and the GPU memory for storing them in raster form can be saved. Cartoonish or symbol-oriented effects in games or movies can also be represented with vector images with the same

advantages. In my work, I have aimed at providing solutions for using vector images as a "native" content type for 3D real-time rendering.

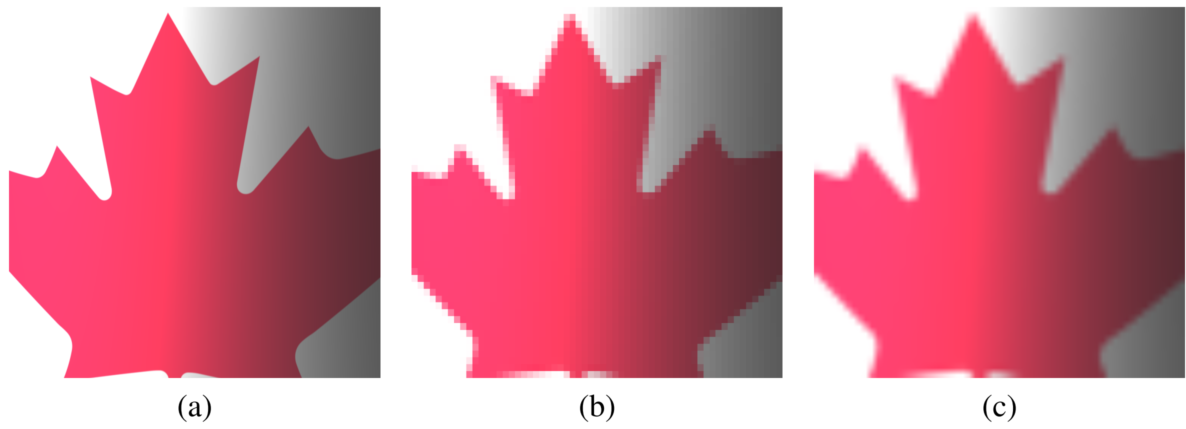
1.3 Vector Images in 3D Rendering

In 3D rendering, textures are used to increase realism. Traditional texture mapping uses raster images because of their efficient random access, especially in real-time rendering, which demands high performance. However, textures often contain sharp boundaries both in natural scenes and in artistic designs. Text, ornamentation, and other symbolic surface patterns are typical examples of sharp boundaries. In the real world, when we move close to or far away from such objects, the objects along with their decorations will smoothly become smaller or larger. Sharp boundaries remain sharp when the objects are close but should blur smoothly when they are far away.

Unfortunately, raster texture maps, the usual means of decorating surfaces in computer-generated imagery, do not have this property. As I mentioned before, when we zoom a raster texture mapped surface far away from us, a naive implementation will allow antialiasing artifacts such as Moiré and crawling edges to appear. These artifacts, caused by texture minification, can for the most part be solved by MIP-mapping. On the other hand, when we zoom a raster texture mapped surface close to us, the surface detail starts to appear blurred or pixelated as the texture is magnified. Such artifacts are demonstrated in Figure 1.1. This problem is caused by limited resolution in the underlying image representation, and I define it as the texture magnification problem.

A straightforward way to solve this problem is to increase the resolution of the raster image, then downsample using averaging before display. This approach is called *supersampling*. However, supersampling incurs a large storage and bandwidth penalty, and only defers the problem to a higher magnification. The solution I propose is to use vector images, whose boundaries are natively, explicitly and exactly described by mathematical equations. With vector images, we can preserve sharp boundaries at any magnification level with efficient use of storage. Because of the limitations of previous generations of GPUs and the complexity of vector image representations, vector graphics images have not been previously applied in 3D real-time rendering. However, this thesis targets and presents GPU-accelerated solutions for real-time rendering of resolution-independent texture content with high quality antialiasing. With my approaches, sharp boundaries can be well preserved as they are magnified. The result is shown in Figure 1.2.

The W3C standard for vector images is known as Scalable Vector Graphics (SVG). Existing software such as Adobe Illustrator and Inkscape that comply with this standard support editing and composition of SVG vector graphics images. My goal is to allow the use of 2D vector images as texture maps in real-time 3D rendering. Editing and composing 2D vector image content are not my main concerns. I assume that 2D vector image content will be created using these existing tools. I therefore choose to support the semantics of existing vector graphics formats so that artists can use existing software to create their work conveniently.



(a) (b) (c)

Figure 1.1: *Image magnification artifacts caused by the limited resolution of a raster image representation. (a) original image. (b) pixelated, when using nearest-neighbor interpolation. (c) blurred, when using linear interpolation.*

Figure 1.2: *Well preserved and properly antialiased sharp boundaries in my vector image rendering, rendered in real-time as a texture map using one of the approaches I will present.*

To use vector images in real-time rendering, a frame rate of at least 30 fps is desirable. A minimum of 15 fps is necessary for the illusion of motion and interactive manipulation. However, at this rate, the frames are still detectable. In addition, the technique must support random access to the texture content since the order in which textures are accessed during rendering is not known in advance. Hardware acceleration (in the form of Graphics Processing Units, or GPUs) are also widely used in real-time graphics and so the technique should be compatible with GPU acceleration and with other existing accelerated techniques for rendering 3D content. There has not been much work on

hardware-accelerated random-access vector graphics texturing to date due to the limitations on the programmability and the speed of GPUs. In general, any extra computation other than a single texture lookup to retrieve the color for each pixel will reduce rendering performance. However, recently GPUs have increased capabilities for programmability and computational performance, enabling new, more sophisticated approaches to representing and rendering image content. Also, as computational capability growth outstrips memory bandwidth growth, these more sophisticated approaches, with more efficient use of memory bandwidth, become much more practical and perhaps in the long run even superior to approaches based on simple table lookup into raster images.

Two basic issues have to be taken into account when I try to provide solutions for texture magnification in real-time rendering. First, efficiency has to be high enough so that the frame rate is adequate. Second, in addition to representing sharp boundaries, the sharp boundaries should be antialiased. The performance, programmability and memory bandwidth in the current generation of GPUs makes it possible for me to achieve these goals. However, I can also anticipate the coming generation of hardware, which will likely feature even greater computational and programmable capabilities, but likely with only modest improvements in memory bandwidth. In this future context my results are likely to be even more relevant.

1.4 The GPU

The Graphics Processing Unit (GPU) is an accelerator for 3D rendering. In general, the graphics pipeline consists of three parts: the application part, the geometry processing part, and the rasterization and per-pixel processing part. The application stage sets up the scene, the geometry stage carries out all the transformation, projection, and clipping work, and the rasterization stage rasterizes the image and computes the color for each pixel. In a complete graphics system, typically a general-purpose CPU carries out the application stage while a more specialized GPU executes the geometry and rasterization stages.

The first generation of graphics hardware had no programmability, and all graphics functions were hardwired into a fixed-function pipeline. Over the last ten years or so, however, programmability has been added to the geometry and rasterization stages supported by the GPU. In the geometry section, *vertex shaders* were added to allow the user to specify geometry transformations, and in the rasterization section, *fragment shaders* were added to provide customized surface shading.

At first, vertex shaders and fragment shaders were implemented in physically separate processing units. The shaders were different in programmable capabilities, texture lookup support, and functionality. Some features were supported in one shader type but not in the other. The vertex unit was specifically used for the shader computation for each vertex on the geometry mesh, and the fragment unit for the shader computation for each pixel on the screen. Memory for textures was also relatively small and control flow was poorly supported. Initially, shaders were written in assembly language because of

the lack of high-level parallel shader programming language. This was the state of the art when the work described here was begun.

However, each new generation of GPUs has gotten closer and closer in programmability to CPUs. There are now no more practical limitations on the number of instructions in a shader. Control flow is well supported although it still affects performance if the computations for a block of data are not consistent. Local addressable arrays are also now supported, allowing stacks and recursive algorithms. The most important improvement is that the vertex shader unit, fragment shader unit and other geometry processing units are now unified. The advantage of this is that the GPU can balance the workload of these stages in the graphics pipeline among a set of common physical processing elements. For example in the NVIDIA G8800 GTS (whose architectural diagram is shown in Figure 1.3), there are eight processing units. Each unit has two vector processors each of which is 8-way SIMD. The units take over different tasks according to the current workload. If the workload for the fragment shader is too heavy, more processing units will be allocated for the fragment shader, and the same for other scenarios. Also with the emergence of high-level parallel programming languages for GPUs, such as the RapidMind platform, GLSL and Cg, shaders can be written more efficiently and productively. In my work, I mainly use fragment shaders to allow the use of vector images as texture maps in 3D rendering. The algorithms I will present have also been designed to be efficient on GPU architectures, for example by avoiding complex pointer chasing (when possible) and avoiding control flow (when possible). The techniques presented here are not limited to GPUs. More generally, CPUs are evolving to include many of the features of GPUs, including wide SIMD instructions (SSE is 4-way, but AVX will be 8-way and the Intel Larrabee will be 16-way). The constraints upon algorithm design imposed by GPUs are not unique to GPUs, they are a consequence of the architectural strategies used by GPUs for high performance, strategies that are also rapidly being adopted by CPUs. Therefore I expect that the results in this thesis will be relevant to future high-performance CPUs as well as today's GPUs.

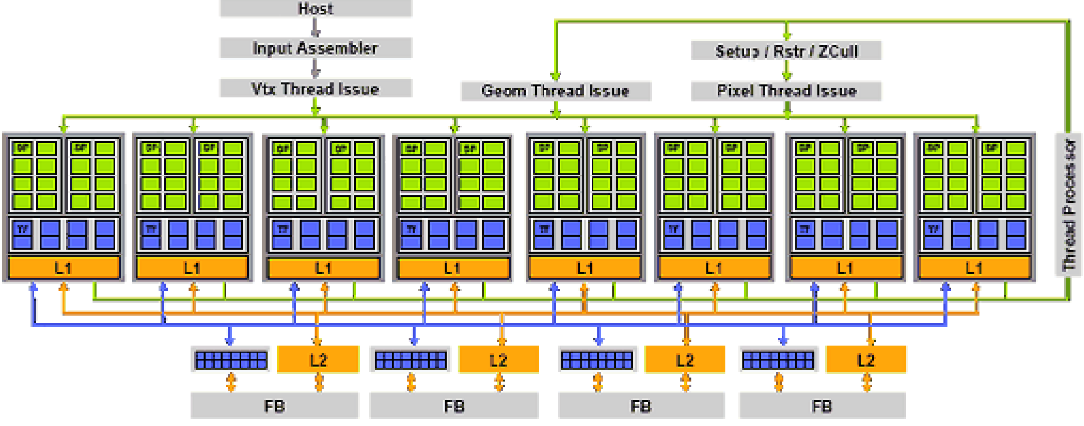


Figure 1.3: *Diagram of The NVIDIA G8800 GTS Graphics Processing Unit.*

1.5 Goals of the thesis

I set my goals as follows. I would like to develop representations of vector graphics images suitable for real-time rendering, such that the storage size is efficient compared to raster images. The natural properties of vector graphics should be supported, so my representation should be resolution-independent and allow infinite magnification. Since I want to support the use of existing tools and file formats, features in the original vector image should be rendered exactly as specified. Also, all image boundaries should be antialiased; this is an absolutely crucial property for image quality. Furthermore, my methods should be easily extended to render other special effects useful in a 3D context, such as embossing, halos, drop shadows, textured boundaries, and so on. Finally, the rendering speed needs to be real-time. Note that unlike some previous work, I target standard vector image content directly, rather than attempting to find an improved vector representation for images of natural scenes.

My main contributions are as follows. I have developed a system that supports the capabilities of the general SVG format and allows the use of this vector content as texture maps in 3D real-time rendering. My performance varies between about 30 fps to more than 100 fps depending on the complexity of the image. To achieve this, I have derived a simple and robust way to compute the distance to any curve. This can be used to efficiently compute high-quality antialiasing. The distance computation also allows special effects such as embossing and halo effects.

Vector textures have many applications in 3D rendering. They can be used not only as texture maps to provide high quality renderings, but also to simplify the underlying geometry of objects, while allowing the generation of complex visual effects. One of my final results is shown in Figure 1.4. Note, for example, that a matte (transparency) channel represented as a vector image can be used to represent complex geometric boundaries in a compact way.

The outline of this thesis is as follows: In Chapter 2, prior work on vector textures is introduced in chronological order. My published work is also referenced here in context. In Chapter 3, approaches to finding the closest distance to high order curves are introduced, along with the introduction to root finding methods for high order equations. In Chapter 4, a method that supports bi-level vector images in 3D rendering is introduced. In this method, boundaries are all approximated by line segments with C^0 continuity. In Chapter 5, the previous approach is extended to support a larger subset of the features of general SVG files, including overlapping regions of different colors and gradient fills. Here, boundaries are approximated with circular arcs with C^1 continuity. In Chapter 6, I will introduce more efficient data structures for storing vector images that can support rendering of all boundaries, including higher order splines, without approximation. In Chapter 7, a parallel K-Nearest Neighbor search algorithm on multi-core Cell BE processor is introduced. This problem is closely related to the distance computation used for antialiasing in my vector texture representation. This approach can be extended into an alternative way to store vector data that would allow dynamic updates, since the data structure itself can be built in a data-parallel fashion. In Chapter 8, I will give a summary of my research and discuss possibilities for future research.

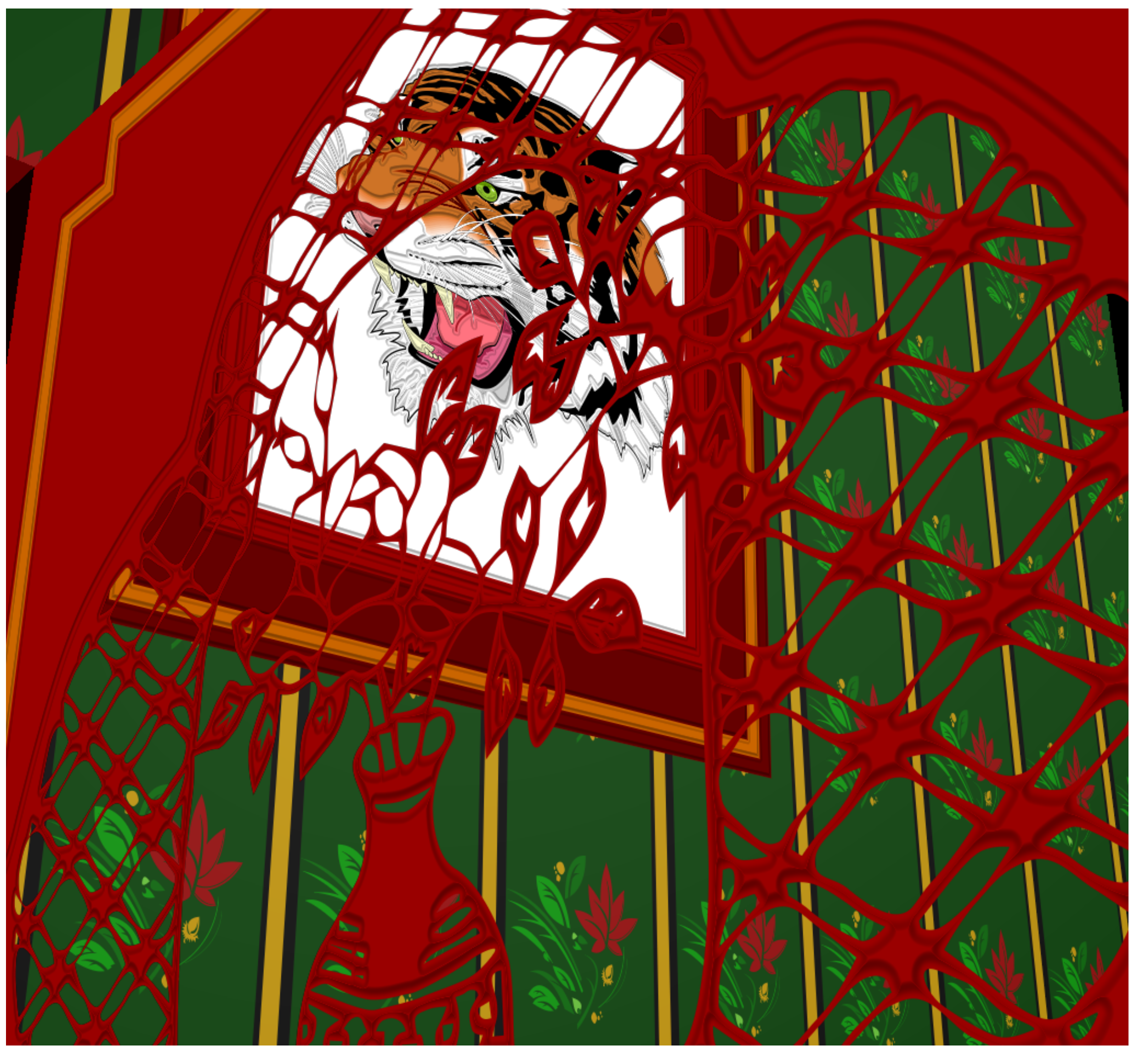


Figure 1.4: *Rendering result with my vector texture system.*

1.6 Definitions

In this thesis, the terms exact, accurate, interactive and real-time are used with the following specific meanings:

exact: Assuming infinite-precision arithmetic, the result can be computed exactly using a finite amount of computation. In practice, the same computation may be performed using finite precision arithmetic to generate a sufficiently accurate approximation.

accurate: Given a specific error tolerance (or "accuracy"), an approximate value can be computed that is within this tolerance of the true value. It should be possible to use a finite amount of computation to acheive any given finite tolerance, but the amount of computation required may be higher for more accurate solutions.

interactive: A rendering process is considered interactive if a consistent frame rate of at least 15 frames per second can be maintained, with a latency of no more than one frame. At this rate and latency, immediate feedback to user input and the illusion of motion can be supported.

real-time: in this thesis, I use this term to define rendering rate of at least 30 fps, which supports fluid motion. -¿ real-time: in this thesis, in contrast with the rest of computer science but consistent with its use in graphics, I use the term ”real-time” to describe a process with a rendering rate of at least 30 fps. At this rate, fluid motion can be supported.

Chapter 2

Prior Work

Research on GPU-accelerated vector-graphics approaches to image magnification started in 2003 with the emergence of programmable GPUs. As the capabilities of GPUs were improved gradually, different approaches ranging from simple extensions of raster image formats to more advanced and complete SVG-compatible representations have been developed. In the beginning, most approaches simply extended raster images to include limited boundary information. The traditional bilinear interpolation was modified such that the interpolation only used samples from within the same region, so sharp boundaries were preserved. Several different ways to determine the regions and extend the interpolation rules were developed. In these approaches, super-sampling was generally used for antialiasing, if it was considered at all. However, as GPUs became more powerful, more sophisticated methods were developed to support more general images, including representations capable of supporting nearly the full feature set of SVG file formats.

In this chapter, I give a comprehensive introduction to the approaches to image magnification published to date. The work will be introduced in chronological order. My own published work will be introduced in sequence with my survey of previous work so it can be seen in context. It also should be understood that this area has evolved alongside continually evolving hardware capabilities.

2.1 Methods Using Modified Interpolations

Early in 2003 to 2004, research on texture magnification focused on combining extra boundary information with the original raster samples, to extend raster image representations to have some of the properties of vector graphics images. In general, these methods superimpose a uniform grid over the raster image, then record the boundary configuration within each grid cell. With this approach, texture lookup is a generalization of bilinear interpolation and can achieve high rendering speed. The difference between these methods and traditional bilinear interpolation is that the modified interpolation rules do not cross the boundaries depicted in the grid cells and so can retain sharp boundaries when images are magnified. Note that although some of this research investigated the problem

of obtaining boundary information from the high-resolution raster images, I consider this problem to be out of the scope of this thesis. I simply assume boundary information is available.

Sen [53] uses two textures, a color bitmap texture and a "silhouette map" texture. The silhouette texture records the boundary information going through each cell. These two textures have about the same resolution and overlap each other. One of them is shifted by half of the grid cell size, so that the sample colors in the bitmap texture fall at the corners of the silhouette map cells. The boundaries in each cell separate the cell into different regions. The color interpolation for a point is defined only among the corner samples that belong to the same region.

Using Sen's approach, the computation of the color involves only bilinear interpolation, which is hardwired on GPUs. Although the modified interpolations are different from the standard bilinear interpolation, he developed a common equation that includes all scenarios supported in his approach. The computations in the fragment shader in the GPUs have no branches and only a few bilinear interpolations, and so this approach achieves high performance. At the time this approach was developed few GPUs supported true branching, so this approach was an absolute necessity. However, the approach is not general, as the number and configuration of boundaries in a cell is limited and higher order curves are not supported.

To reduce the memory required to store boundary information, Sen uses two bits to record the boundary configuration and a local coordinate system to store the boundary coordinates. Since only two bits are used to record the boundary configuration in each cell, this method only supports a limited cases of boundary configurations. In particular, there cannot be more than four boundaries in each cell, and only one boundary can pass through each cell edge. The resolution of the grid can always be chosen in an attempt to support the content, but one small detail in the image could force the uniform grid to be split until the boundary configuration is supported, which will increase texture size as a consequence. Still, this is not a general solution. Some configurations, such as the many lines meeting at the center of a pie chart, can never be represented accurately with this technique. More bits might be used to support more boundary configurations. However, a common formula that can handle these more complicated boundary configurations is hard to develop, may be too expensive to be practical, and might not exist at all. Control flow could be used to support additional cases, but this may cause lower performance. In short, to use this technique in practice, users have to accept certain limitations on its ability to represent content.

Since this method is simple to implement, Sen provided a GPU implementation. Antialiasing was not supported in this implementation although theoretically it could be added with some additional computation. It is also worth noting that Sen originally developed his approach for representing shadow maps. Due to the simplicity of the representation, it can be constructed on the GPU dynamically for this purpose.

The method developed by Tumblin and Choudhury [56] is similar to Sen's. It combines pixel sample values and boundary information into a single structure they call a "bixel". This method has more complicated interpolation rules to remove certain arti-

facts that appear between different interpolation patches. Any sudden changes of interpolation rules tends to cause false boundaries that do not exist in the original images, so the interpolation rules have to be set up to provide a smooth transition. However, to support the more complicated interpolation cases, they had to apply control flow to handle the various cases. The implementation was therefore more complex and the performance was slower than Sen's method. This approach was actually not implemented on GPUs, although this was primarily due to the lack of control flow in GPUs at the time. This approach, however, could be easily supported today.

Tumblin and Choudhury's work also uses a small number of bits to record the boundary information. The advantages and problems related to boundary configuration representations with a small number of bits exist in this method as well. To antialias images, they used super-sampling. As with Sen's approach, boundaries are also approximated with line segments.

Ramanarayanan et al. [47] presented a technique that can support higher order boundaries. Boundaries represented by cubic Bézier curves are retained and evaluated exactly. Curved boundaries are not approximated by line segments as in previous work. This method also is aimed at supporting any boundary configuration and the features of standard vector graphics file formats such as SVG. Boundaries intersecting each other within a cell split the cell into subregions. The interpolation happens in each subregion, and the interpolation samples are not only from the same texel, but also from reachable samples in nearby texels. Here more complicated data structures, such as kD-trees and feature lists, were used to store the boundary features.

Although Ramanarayanan et al.'s approach aimed at supporting all boundary configurations, loops within a texel were still not supported. Small loops will cause problems similar to the previous methods, since there may not be any color samples inside such loops, or enough to use for good interpolation. This approach also requires complex preprocessing. In particular, intersections between the boundaries represented by splines have to be computed, so the input vector graphics image can be flattened into a planar map of regions, and then these curves have to be segmented yet again into monotonic regions. During run time, the rendering system needs to find the subregion each point falls in and the reachable samples in the neighborhood. To compute this information, a general cubic root solver is needed, which is relatively expensive for real-time rendering. Due to this method's run-time complexity, it was not implemented on a GPU. The software implementation uses super-sampling for antialiasing. It should be noted that such an approach to antialiasing can be used with any method, but is not efficient.

Tarini and Cignoni developed the "pinchmap" representation [55], which targets fast rendering speed. Since changing the interpolation rules complicated the color computation, their method kept the standard bilinear interpolation but snapped the uniform samples to the vector content's boundaries. Any point that needs to be evaluated then falls into a region with four samples at its four corners, and interpolation can use existing GPU hardware. This method requires a few instructions and only one extra texture lookup for the snapping information. Antialiasing at one target scale can be achieved with little extra effort. Instead of snapping samples to the exact boundaries, they are snapped close to

the boundaries with small transition gaps. Points that fall into these gaps are interpolated by the colors on both sides. As a result, the color transitions are smooth. However, as the images are magnified, the transition zones become bigger because of the fixed gap in the texture maps. In the correct solution for antialiasing, the transition zone should be only one pixel wide on the screen, regardless of the texture transformation or projection. Therefore, this approach ultimately has the same problem as raster images, although it may allow a wider range of magnification factors in practice. Also, if the pinchmap is too far away, the transition zone may be too small, causing minification aliasing.

Although the rendering speed of Tarini and Cignoni's technique is good, it still places restrictions on the discontinuity configurations. For example, 2×2 grids cannot all have boundaries. When multiple boundaries cross each other, or they run too close to each other, some of the lines are lost, and the images appear blurred. As with most previous techniques, curves are represented by piecewise linear approximations.

One advantage of the above methods is that they can be used for natural images. Natural scenes are usually so complicated that they cannot be represented by regions enclosed by curves only, as in standard vector graphics images. Color samples are a better representation than mathematical equations for photographic content. However, all these methods represent only the low frequency and the high frequency information. The images are either over smoothed or very sharp. Information with in-between frequencies are lost. Therefore, the images tend to look cartoonish. A more robust, multiscale representation is needed. How to preserve the information of different scales in natural scene images, however, is not the topic of this thesis. Instead, I will focus exclusively on how to render vector graphics context as already represented in existing vector graphics image formats.

2.2 Methods For Antialiased Bi-level Images

Instead of embedding boundary information within raster images and trying to extend their semantics, we can accelerate a more limited vector graphics use case. One use case of particular interest is that of bi-level images, in which we want to precisely represent a boundary between a single foreground color and a single background color. The following methods were designed to render such bi-level images, in which color samples can be completely omitted. Bi-level images are simpler than general vector graphics images but still have many applications, such as typography, signs, and ornamental decorations composed of two colors. They can also be applied to the representation of alpha mattes (transparency masks) in the case that only foreground (fully opaque) and background (fully transparent) need to be distinguished. Such mattes are common, and appear in both compositing and as the result of trimming in computational solid geometry.

For bi-level images, the problem of color retrieval reduces to that of finding out what region (foreground or background) a query point falls in. All paths in bi-level images must be closed, so we have to determine if the point is inside any closed path around the foreground color. If it is not inside any such path, then it is the background color. Many

applications of bi-level image representations also make the simplifying assumption that the paths are simple and the foreground regions are non-overlapping.

Typography is a typical application of bi-level images. Each glyph in a typeface can be represented as a small number of closed paths(usually one or two) surrounding a foreground region. As with many applications of bi-level images, we need to antialias the result. One effective way to antialias a bi-level image is to compute the signed distance to the nearest path rather than a simple inside or outside value. Then a smooth step function can be used to antialias the shape by providing a transition between foreground and background. The width of the transition zone of this smooth step should be scaled so that it is roughly one pixel wide.

Distance fields can be computed exactly or can be approximated. Frisken et al. [18, 17, 43, 16] used adaptively sampled distance fields (an approximate representation, but with controllable error) to represent 2D fonts or 3D shapes. For each point on an adaptive grid, they computed the closest distance from this point to the curve/surface. A sign was also stored to indicate whether each point was inside or outside of a closed foreground path/shape. The boundary between foreground and background is then given by the locus of points with an interpolated distance of zero.

Other than sampling distances over a uniform space, adaptive approaches sample more densely in regions that have more details. The distances for points that fall between the samples are approximated by the linear interpolation of the samples. A hierarchical structure, such as an quadtree, is used to store the samples efficiently.

Adaptively sampled distances reflect the curves/shapes more accurately than non-adaptively sampled distances. With more samples around the small details, small high frequency features, like sharp corners, can be more accurately reconstructed. This improves upon a uniformly sampled distance field in both reconstruction quality and storage size. However, the more samples used in high frequency regions, the larger the storage will be. Also, once the high frequency information exceeds the resolution limit of the maximum sampling rate, the sharp features are still rounded off.

Ray et al. [48] used a cubic univariate Hermite function as the implicit function for each spline segment within a cell. This function takes four parameters: the entry and exit points of a curve over each cell, and the corresponding tangents. This function cannot represent all kinds of cubic splines. If a spline segment in a cell cannot be represented by the function, the grid will be refined until the spline segment can be represented. For thin features, sharp turns, and more complicated cases, they combined implicit functions in a CSG manner.

Using implicit functions only, Ray et al. obtained the same aliased results as in Sen's work [53]. To further improve the image quality, they used a region coverage ratio to support antialiasing. When an image is magnified, the footprint of a screen point on texture space is smaller than a texel. The ratio is defined as the area covered by one color over the whole projected area of a screen point. This ratio is approximated by the distance from the point to the edge over the average of the two partial derivatives of the projection function in texture space. This approach to antialiasing is cheaper than supersampling, since derivative instructions are provided by GPUs. They used a hierarchical

tree structure to reduce the memory cost for varying resolution images. This work was implemented on the GPU, and a font engine was provided.

The evaluation of the implicit functions and the region coverage ratio approximation are efficient methods, so the implementation of this algorithm achieved high rendering speed. Higher order representations for the splines preserve the smoothness of the spline curves. However, low degree implicit functions cannot represent parametric polynomial curves exactly, so parametric boundaries will have to be approximated.

Small boundaries that cannot be represented by implicit functions are recursively split, but this recursive split can result in a large storage cost. Also, using CSG composition at run time required control flow and many dependent texture reads, which was not efficient for real-time rendering.

Loviscach [34] suggested applying a filter function over MIP-map results. The parameters of the filter are set so that when the image is minified, the result is just the MIP-map result; when the image is magnified, the level 0 image is converted to a threshold image. There is a smooth transition from a greyscale image to a bi-level image, so the switching between level 1 and level 0 is hidden. Intuitively, this approach uses the partial derivatives of the projection function between screen space and texture space to estimate how far the points on the screen are from the boundaries. In case of minification, the relative MIP-map level is chosen and the greyscale result is returned. In case of magnification, the antialiased greyscale result from level 0 can be regarded as a distance approximation in texture space. It is then scaled by the partial derivatives to approximate the distance in screen space. This distance is later mapped through a smooth step function to produce antialiased sharp boundaries.

In this approach, texture minification and magnification are combined into one equation. It is fast because the equation is simple. The transition between minification and magnification is seamless. The problem with this method is that complicated boundaries are softened. Sharp corners are rounded, so this approach cannot represent vector graphics content with high fidelity.

Loop and Blinn [33] proved that any quadratic or cubic spline can be projected onto a canonical implicit spline. By assigning a fixed set of texture coordinates to the control points of an arbitrary spline, which side a point is on relative to the curve can be evaluated by substituting the point's texture coordinates into an implicit function. The sign of the evaluation can be used to find the query point's color.

For antialiasing, they used the approximate closest distance of a point from the curve. The approximate distance was computed using the ratio between the implicit function and the magnitude of its gradient. This approach is justified by approximating the implicit function with its first order Taylor expansion.

This is a fast rendering algorithm. However, it requires the vector images to be triangulated. Some images can result in complicated triangle meshes. The images cannot therefore be simply used as texture maps. To apply a vector image over an object, they need to intersect the triangle meshes of the image with that of the object, resulting in potentially a complex mesh with a poor distribution of triangle shapes and sizes. This is therefore not a practical method for texture mapping.

Inspired by the work by Frisken et al. [18, 17, 43, 16] of using distance fields for antialiasing, my methods [45, 44, 46] also used filtered distance for antialiasing because it is more efficient than super-sampling. However, my method differs from theirs in that an accurate closest distance, rather than an approximated distance, is computed for each query point. Therefore, in my approach high-frequency details, like sharp corners, are preserved regardless of the magnification. One of my goals was not only to provide antialiased boundaries, but also to provide mechanisms for special effects. With accurate distances, special effects, such as embossing, textured strokes, drop shadows, and halos, can be implemented easily because these effects are based on the distances from points to boundaries. In many schemes using approximate distances, in contrast, the error can be arbitrarily bad, especially far from the edge. Unlike Frisken et al.'s methods that store a sampled distance field, I also store the control points of the curves directly, which results in a more compact storage size.

My work has three stages. Each stage involves further development of methods to compute the closest distance to boundaries. I started with the simple case: images are bi-level images, and boundaries are approximated by line segments with C^0 continuity. In the second step, I targeted general vector images. To improve the smoothness of the approximation, higher order curves, such as quadratic and cubic splines, are approximated by circular arcs with C^1 (actually G^1) continuity. Finally, in my most recent work, I present a technique that supports distance to differentiable parametric curves (including polynomial curves of any order) without approximation. A brief introduction to each stage of my work will be given in this chapter, but one chapter will be devoted to each of these stages in the following part of the thesis.

In the first stage of my work [45] I target bi-level images and also approximate all curves by line segments. I use a uniform grid accelerator, and record for each cell only the boundary features relevant to the color computations of the points in the cell, i.e., those boundary features whose Voronoi regions overlap the cell. All boundaries that are certain distance away from the cell or blocked by other boundaries are considered not necessary in color computation. For each point in a cell, its closest distance to the boundary features is computed with the recorded features. The limited number of distance computations accelerates rendering. In my approach, all boundaries are oriented clockwise without loss of generality. A negative distance indicates the point is inside the path and therefore bears the foreground color, while a positive distance indicates the point is outside the path and bears the background color. Unlike previous work that had restricted the boundary configuration, my only constraint was that the number of boundary features in each cell can not exceed a certain number. For any given image this number can be increased as appropriate, although there is a trade-off with speed. In my representation sharp corners are well preserved with no geometric roundings. Antialiasing is achieved by passing the closest distance through a smooth step function, which is much more efficient compared to super-sampling. My work also shows anisotropic antialiasing can be efficiently implemented by using the Jacobian matrix which projects the distance gradient from texture map space to screen space. More details will be described in Chapter 4.

Green [20] also used an approximate closest distance for image magnification and

antialiasing of decals. First he generates a bi-level high-resolution raster image that contains the in/out state of the decal at every pixel. Then for a low-resolution distance representation texture, for each sample pixel, he computes the closest distance to a boundary pixel by comparing with the neighbours using a brute-force method. This texture is then interpolated using linear interpolation and used in alpha test for magnification. Distances are clamped to the interval [0,1]. Negative distances are mapped to the interval [0,0.5], and positive distances are mapped to the interval [0.5,1]. Points with distance 0.5 represent the points with the original distance zero, i.e., the boundaries. The threshold of alpha test is set to 0.5. With an alpha test threshold of 0.5, an approximation to the original vector image can be easily achieved.

Green's method is fast. For better quality, a simple pixel shader implementing alpha blending can be used for antialiasing. It can be used in low end GPUs, is practical, and has actually been used in Valve's Team Fortress game. This technique uses uniformly sampled grids, and is a special case of Frisken's distance field approach.

However, as with other simple methods, this method does not support complicated edge configurations. Corners are rounded, and only bi-level images can be supported. This technique cannot be extended to support general vector graphics, but because it is fast, it is still useful as a representation for iconic and textual information in a game environment.

2.3 Methods for General Scalable Vector Graphics

In the second stage of my work, I decide to extend my approach for bi-level images to support general vector graphics, in particular the features supported by SVG images [44]. By supporting the features of SVG images, I enable the use of standard vector graphics applications for creating and editing texture content. The most fundamental feature of a general SVG image is its support for multiple layered paths with different fills, which can include colors, gradients, and even raster images. In addition, paths can be outlined in different styles: path boundaries with borders, path with gradient fills, strokes with various widths, and so on.

In my work, to support multiple layers the images are flattened, so that there is only one layer that forms a partition of the plane. The idea is that after flattening, the image only contains closed paths that do not overlap. Each boundary is then shared between two paths. To compute the color for a point, I still compute the closest distance to the boundaries. Once I find the closest boundary and the two colors on each side of it, the color can be smoothly blended from the two colors. Although at some intersection points more than two paths may share the same corner, I find that just using the closest distance still gave good results.

In my previous work, in the examples of embossing, since I only support C^0 continuity some creases along the line segment junctions are obvious when the images are magnified. For better quality, I decide to use a higher order approximation in this stage of work, in which all boundary curves are approximated by circular arcs. Computing the

distance to a circular arc is almost as cheap as computing the distance to a line segment, and yet the continuity can be improved to C^1. More details about the second stage of my work is described in Chapter 5.

The texel program approach developed by Nehab and Hoppe [40] also supports general SVG images as texture maps in the context of real-time rendering. As with my work, they use a uniform grid accelerator, but since it is sparse they store it in a (perfect) hash table. For each cell, the boundary features that overlap with the cell are recorded in layer order as they are drawn, so that layer overlap can be resolved in the shader, avoiding the problems associated with flattening. This work was done in parallel with the final stage of my work and there is considerable overlap. However, they use an approximate approach to first approximate the cubic splines with quadratic splines and then use an approximate equation to compute the distances to quadratic splines. I develop an accurate approach to compute accurate distance to spline curves of any order within fixed error threshold.

In the pixel shader of Nehab and Hoppe's approach, to compute the color for a point, only the relevant boundaries are considered for each cell. The boundaries are processed layer by layer from back to front. In each layer, while they compute the closest distance to boundary, they also keep a count for the winding rule test. After a layer is processed, an inside or outside test computed with winding rule is used to evaluate the sign of the distance. This closest distance is then filtered for use in blending the current layer's color with the already computed color. In this work, all cubic splines are approximated by quadratic splines. Furthermore, a fast but approximate equation to compute the distance to the quadratic splines is developed. This approximated distance is good enough for antialiasing. However, away from the boundaries the distance accuracy drops dramatically. Thick borders were therefore converted to closed paths, and special effects like embossing cannot be supported. Another contribution of this work is a fast preprocessing algorithm and the use of a perfect hash to store the sparse accelerator structure.

In the last stage of my work [46] I develop a solution that overcomes the flattening problem, and can also render differentiable parametric curves of any kind precisely and efficiently. Similar to the approach used by Nehab and Hoppe [40], in each grid cell, relevant boundaries and corresponding path colors are listed in layer order. To compute the antialiased color, I still use the closest distance and the two colors on each side of it. As I go through all the layers, the closest distance and the colors are updated and eventually the distance to the closest visible boundary and the colors are obtained, which are then used to blend between and so antialias the boundary between two adjacent regions, or a stroke and region.

In a general SVG image, boundaries are normally described by a variety of parametric curves, including line segments, quadratic splines, cubic splines, and elliptical arcs. In all previous work, either the higher order boundaries are approximated by lower order primitives, which permits an easier distance computation, or the distance is approximated. No techniques so far have dealt with rendering cubic splines directly, even with approximated distances. Accurate distance to quadratic splines, cubic splines, or ellipses has also not been used. The main contribution of the final stage of my work is that I can

render any differentiable parametric curves, including in particular the important case of cubic splines, directly. With my approach, accurate distances can be computed for any curve as long as the point is on the convex side of the curve or on the concave side and within the radius of the curvature. The computation is easy, robust and practical for real-time rendering. More details are given in Chapter 6.

Chapter 3

Computing Distance to Curves

As mentioned before, a crucial part of my approach to rendering vector graphics requires the computation of the accurate distance to boundaries for antialiasing and special effects. In this chapter, I will review general approaches to finding the closest distance to curves for an arbitrary point, and explain the problems of using them in real-time rendering. In the following chapters, I will introduce my own methods for distance computation.

In SVG files, boundaries are composed of line segments, quadratic splines, cubic splines, and elliptical arcs. These are all parametric rather than implicit curves, and in some contexts the distance along the primitive is important (for example, for dashed curves). Distance computation to a line segment is easy and is used in the first stage of my work. Distance computation to higher order parametric curves is harder. Computing the distance to even a degree two parametric curve requires finding the roots of a cubic equation. Theoretically, such roots can be found analytically using a closed formula, but the solution is numerically unstable and involves many cases, and therefore is not suitable for GPU implementation. For a degree three curve, such as a cubic spline (which is the highest degree curve used in SVG files), the solution involves solving the roots of quintic equations. Finding these roots analytically is infeasible so a numerical approach must be taken.

Different approaches to finding the roots of polynomials are introduced here with their advantages and disadvantages. I survey a variety of algebraic approaches. However, in general the main disadvantages of direct algebraic approaches are computational expense and robustness. Instead of using any of these algebraic approaches, in Chapter 6 I develop an iterative algorithm that exploits the geometry of the nearest-distance problem. This iterative method finds the closest distance to a curve as well as the parameter along the curve at which that closest point lies (useful for texturing strokes). In Chapter 6, we develop an iterative numerical method to find the closest distance to a parametric curve, and the parameter along the curve at which that closest distance lies (useful for texturing curves). Compared with the classical (but more general) methods presented in this chapter, our approach is simple, robust and suitable for GPU implementation.

3.1 Finding Roots of Polynomials

I will begin with a general description of the problem and a mathematical formulation of its solution. Consider a parametric curve $\mathbf{P} : [0,1] \to \mathbb{R}^2$. Suppose we know the value $t^* \in [0,1]$ for which $\mathbf{P}(t^*)$ is closest to a given query point $\mathbf{Q}$. Then $\mathbf{P}(t^*)$ must satisfy the following equation (although it may not be the only point to do so):

$$(\mathbf{P}(t^*) - \mathbf{Q}) \cdot \mathbf{P}'(t^*) = 0 \tag{3.1}$$

This means the line between $\mathbf{P}(t^*)$ and $\mathbf{Q}$ is perpendicular to the tangent at $\mathbf{P}(t^*)$. The parametric value t^* is a root of this equation. Not all the solutions to this equation give minimum distances; some are maximum distances and some are not in the parametric segment of the curve we care about. However, if we can find the roots to this equation, we will be able to select the true minimum distance from among them. For quadratic splines, this equation is cubic, and for cubic splines, it is quintic. Finding roots for these equations however is not trivial. Analytic solutions for cubic roots are numerically unstable and involve a case analysis, which requires control flow. The formula has six cases, and two special cases to avoid division by zero. Many other techniques fail in special cases, and this is usually unacceptable in graphics. For example, the otherwise simple and elegant method given by Cardan [41] fails for the rare but possible case of three co-located real roots.

Since use of control flow on the GPU has a substantial negative influence on performance, I would like to avoid the use of case analysis. However, at the same time, quickly and robustly finding all roots in all conditions is essential.

Finding roots for quintic equations is even more complicated. The Abel-Ruffini theorem shows there is no general solutions for polynomial equations of degree five or higher in terms of only arithmetic operations and radicals [15]. Galois theory states what kind of quintic equations are solvable by radicals: a polynomial equation can be solved by radicals if and only if its Galois group is a solvable group. Even for the solvable quintic equations, the solutions generally are still fairly complex: a general quintic equation is converted to the Spring form, then we compute the roots of the Spring form, and finally compute roots to the original equation from the roots to the Spring form. However this solution can be so complicated that it might require hundreds of symbols, so even if the above equation could be converted to Spring form (which we did not attempt to prove) the result would still be impractical.

3.2 General Approaches in Finding Roots

In practice, people resort to numerical solutions. It is well known that Newton-Raphson and other similar iterative algorithms sometimes do not converge; we need a more robust solution. Although I only need real roots, I survey general approaches to finding roots, including complex roots.

Typically, the Jenkins-Traub [30] and the Laguerre [19, 64, 42] algorithms are used to find complex roots, because the Jenkins-Traub algorithm always converges, and Laguerre algorithm almost always converges. Although Laguerre is simpler, it cannot be chosen for my application, because even if the diverging cases are rare, any single case that fails can cause visually disturbing artifacts during rendering. In contrast, the Jenkins-Traub algorithm is too complicated to be used in real-time rendering. Also, both of these algorithms (and many other polynomial root-finding approaches) entail polynomial deflation. After one root is computed from the original polynomial, the polynomial is deflated to lower order, and the second root is computed. This procedure goes on until all the roots are found. Polynomial deflation is a another potential problem [8]. It can be numerically unstable if the roots already computed are not accurate enough, thus the roots from the lower order polynomials will be different from the roots in the original polynomial, especially if only single-precision arithmetic is used, as in GPUs. The deflation step is also complicated. Finally, to find each root, Jenkins-Traub uses a three-stage method, and each stage is an iterative procedure. The complexity of both these methods means that they are simply not suitable for real-time rendering.

Aberth's method [13, 1] is a relatively simple one. It approximates all the roots at the same time based on the fundamental theorem of algebra that each polynomial of degree n is identical to a product of n linear polynomials. In terms of complexity, in the case of a cubic spline, the quintic equation has five roots. For each root at each iteration, it evaluates the polynomial and its first order derivatives over the complex numbers. Aberth's method needs to compute all five roots simultaneously all the time. Although it is relatively simple, it still involves evaluation over complex numbers, which is complicated for GPUs. Durand-Kerner's method is a similar approach and has the similar problems.

Bairstow's method can find complex roots using only real arithmetic. The idea is that any polynomial can be factored into a quadratic polynomial and another lower order polynomial. If the roots for the quadratic polynomial are the real roots, the remainder of the original polynomial synthetically divided by the quadratic polynomial should be zero. This method then iteratively updates candidate roots for the quadratic polynomial until the remainder is zero. After two roots are found, the same method is applied to the quotient polynomial until the remainder polynomial becomes quadratic or linear. This method converges quadratically, except in some special cases.

Graeffee's method squares the polynomial roots repeatedly to separate the roots and then uses Vieta relations to approximate the roots. However, it does not work well under single precision floating point computation.

The splitting circle method [52] numerically solves the roots for complex polynomials. It uses the residue theorem to factor a polynomial to find the zeros within an arbitrary error threshold. It is a complicated method that involves finding a splitting circle that separates the zeros of the polynomial into zeros that are within the circle and those that are outside of the circle. It then computes the polynomial that vanishes at the zeros within the circle, and finds the other factor polynomial using polynomial division. The splitting circle is found by an iteration of Graeffe's method. This method is not only too complex for GPU implementation, but also will have similar numerical issues with

the single precision as the Graeffe's method.

Brent's method [7] is a root finding algorithm that combines the bisection method, the secant method and inverse quadratic interpolation. It is relatively complicated and involves many cases. It does not converge for all cases, but does apply to any kind of parametric curve.

To compute the closest distance using these methods and their variations, basically we need to find all the roots and choose the one with the smallest distance. So iterations need to be done for each root.

3.3 Finding Real Roots

Although the general approaches to finding roots are all complicated, fortunately only real roots are needed in my application. Therefore much simpler approaches can be applied.

Certain iterative methods such as Regula-Falsi always converge if we can find a monotonic, zero-crossing region of the curve and apply this method to such a region. Therefore, the problem of finding real roots reduces to the problem of segmenting the curve into zero-crossing monotonic regions. In case of quintic polynomial, one way to compute the monotonic regions is to compute the roots for its first order derivative equation. These roots include the maximum and minimum values of the polynomial. After sorting the roots, each region between two successive roots is monotonic. Within each region a stable method can be used to find the root robustly.

However, the first order derivative of a quintic is a quartic, and it is still hard to find the roots for a quartic. To find the quartic roots, we can nevertheless similarly find the roots of its derivative, which is a cubic. The same procedure can be applied recursively until the equation is a quadratic or linear where roots are easy to find analytically. However, I decided not to use this method in my application because it involves too many levels of iterative computations. With single precision floating point computation in GPUs, inaccurate roots at one level may cause wrong results at higher levels. It also requires constructing the quintic at run time, which requires the evaluation of both the cubic spline and its first order derivative.

Another approach takes advantage of the convex hull feature of Bézier curves. For a quintic equation, we can always transform it into a Bézier curve of degree five and represent it with six control points. Given such a curve in Bézier form, we need to test if the convex hull of its Bézier curve intersects with the line $y = 0$. If it does not, it means there is no real root at all. Otherwise, the Bézier curve is subdivided into two Bézier curves, and the intersection test is applied to each curve recursively. This step continues until the curve is smaller than a threshold. This approach is stable but only converges linearly.

A variation of this approach is to switch to a faster technique once we have proven the segment is monotonic and crosses zero. A monotonic curve can be determined by

its control points. If the control points are monotonic (in other words, if all differences between adjacent control points have the same sign), the curve is monotonic. Otherwise, it is not. Once a monotonic curve is obtained, a stable iterative method like Regula-Falsi can be used to find the roots.

However both of these methods require that we construct the quintic and transform it into Bézier form at run time. These computations are complicated for GPUs. The subdivision approach also needs a stack to store intermediate results and backtracking to return complete results. Although a stack and backtracking can be implemented in the latest generation of GPUs, such an approach would probably reduce the performance by an order of magnitude. They are, therefore, not efficient enough for real time rendering. Also, addressable arrays were not available in GPUs at the time I did my research.

As a comparison with other approaches, Wang et al [63] solved the same problem as mine in robotics. They need to find the closest distance to a spline curve from any point in real-time to control the behavior of a robot, with the assumption that the point is within the radius of the road curvature.

In their work, they test with both Newton and quadratic minimization methods, and develop a new method that combines the two. The experiments show that with quadratic minimization method alone, about 1/3 of the points converge within 8 iterations, but most of the points converge after hundreds of iterations, while a small number of points unfortunately do not converge at all. Using Newton's method only, nine tenth of the points converge quickly after three or four iterations, however one tenth of them need hundreds of iterations or diverge. In their new combined method, the quadratic minimization is used to refine the coarse initial guesses, and the Newton method is used to find the final accurate results. All the points find their closest points between 5-8 iterations. Unfortunately, in my application such erratic behaviour is simply unacceptable. I cannot trade faster convergence most of the time for stability.

In Chapter 6, I introduce a method based on geometric bisection that only converges linearly but which is simple and robust.

Chapter 4

Bilevel Images and Line Segment Features

In this chapter and the following two chapters, I am going to present my own work. This chapter is devoted to my work at the first stage of its development which aims at rendering bi-level vector images as texture maps, using only line segment features. As mentioned in previous chapters, although bi-level images are only a special simple case of general vector images, they have many applications in fonts, documents, signs, decals, and artistic designs. Research specifically for this case is therefore needed and directly useful. At this stage [45], I develop an approach to use bi-level vector images, fonts specifically, as texture maps in 3D real-time rendering, and provide a mechanism to render documents with dynamic update of textual information and the display of kerned text. I also develop my basic approach to efficient anisotropic antialiasing based on the projected gradient of the distance field.

4.1 Introduction

While I later will extend my approach to general vector graphics, in this chapter I have focused on one specific problem: the rendering of text. Rendering text is a classic problem in computer graphics, and one of the original motivations for antialiasing [10, 11, 65]. The standard approach to rendering text is to prerender antialiased glyphs into a table and then composite together images drawn from this table. However, this means that the glyphs are stored at a fixed (and usually low) resolution. Mapping them onto a 3D surface then requires resampling, which can degrade quality and introduce artifacts. Direct rendering of the glyphs on the GPU as I propose avoids these problems, since the antialiasing can be adapted procedurally to the local spatial distortion.

In this work I consider only the problem of rendering *display text*, not *body text*. For maximum readability, body text needs to be aligned with and fitted to the display grid, a process supported by hinting, or subtle variations in the shape of the characters. Hinting assumes a fixed-scale, orthographic projection. Hinting cannot be supported in a

texture map representation since the characters will be dynamically deformed by surface parameterization and perspective. For display text, the shape of the glyph is represented independently of the sampling grid.

Glyphs in scalable fonts are represented using geometric contours consisting of line segments, quadratic splines, and cubic splines. To render text, *many* of these glyphs have to be placed in an image with precise positioning relative to one another. The representations of the glyphs themselves can be precomputed, but it must be possible to quickly lay out and modify an arrangement of glyphs interactively. However, the problem of text rendering is simpler than that of general vector graphics: although glyphs must be instanced many times, glyphs in normal text rendering do not exceed a certain density in any one area and usually do not overlap. In the rare cases that glyphs do overlap, for instance for accents or strikethroughs, it is with a constant offset and we can generate new glyphs for these cases. Bounding box overlap must be supported for kerning, but this is only in the horizontal direction in normal text layout. I can handle more general cases such as support for glyph rotation or more complex overlap at some additional cost in memory and shader execution time.

This chapter makes the following contributions:

1. An efficient anisotropic antialiasing technique for texture-mapped vector graphics (Section 4.3);
2. A GPU-based representation of contours, including an inside or outside test, based on *exact* evaluation of distance-to-feature functions (Sections 4.2 and 4.4);
3. A packed grid accelerator structure based on Voronoi analysis supporting efficient constant-time evaluation of signed distance-to-contour functions (Section 4.5);
4. A sprite mapping technique that supports the dynamic composition of large numbers of glyphs in a single procedural texture map (Section 4.6); and
5. Techniques for supporting special effects such as outlining, miter rules, and embossing (Figures 4.5 and 4.8).

4.2 Outline of Representation

My goal is to design a semi-procedural representation of vector graphics images that could be used as if it were a random-access texture map in a shader. The representation must support arbitrary contours and efficient antialiasing. Also, as GPU fragment shaders do not support control flow without a loss of efficiency, and since such support was not yet universal at the time when this work was undertaken, I decided to avoid a dependency on control flow in my representation.

The representation used in this chapter consists both of data structures stored in texture maps and shader code to interpret that data. However, my system is implemented

using the object-oriented Sh GPU programming system [36, 35], which allows me to strongly encapsulate my representation and use it in any context that a raster texture could be used.

I begin with signed distance functions. Given a 2D shape, a signed distance function f measures the distance to the closest point on the shape's boundary contour. The distance function is negative inside the shape, positive outside, and zero on the contour. Thresholding these functions at zero reproduces the original contours, and it is easy to antialias edges by using a smooth transition function [23, 38], or to support special effects like outlining or embossing.

Signed distance functions can be precomputed and sampled. Interpolation of these samples can reproduce linear edges at any orientation, but interpolation tends to round off corner features. To address this problem, distance fields can be adaptively sampled [18]. This could be implemented on the GPU using multiresolution textures [32], but would require an extra level of indirection and, to support hardware bilinear interpolation, redundant storage. It also does not completely solve the problem: at *some* magnification, the corners will still be rounded.

Instead, I compute the distance function procedurally and accurately, using geometric feature information rather than interpolation of distance samples. I use a Voronoi diagram to build an accelerator, so at every point I only have to consider the minimal number of features required to evaluate the distance field. I overlay a Voronoi diagram with a grid, and in each grid cell I make a list of the contour features that contribute to the distance field in that cell. The resolution of the grid is adapted to the complexity of each glyph, so I can use less storage for simple glyphs and more for complex glyphs. My accelerator structure also takes advantage of spatial coherence to reduce redundancy by searching neighbouring texels for features. During preprocessing, an optimization process assigns features to texels in anticipation of this runtime local search.

The second major challenge with text is the necessary support for dynamic layout of large numbers of glyphs. First, the feature data for a set of desired glyphs are packed into a "font" texture. Then, at every texel of a "sprite" texture I store the offset, scale, and extent of a glyph that covers that texel's cell. During rendering, I access (using dependent texturing) a set of features from the glyph referenced from the current cell and from any neighbours whose glyphs might overlap the current cell. I then compute the overall minimum distance function to these features. This approach requires only one level of texture indirection, is spatially coherent, and takes a constant amount of time per rendered pixel.

4.3 Antialiasing

One of the major advantages of an implicit representation of contours is that it can be antialiased easily. For an infinite straight edge, it has been shown that using the signed distance function (normalized plane equation) as the implicit representation and using a

smooth transition function to convert the implicit form to intensity is equivalent to convolution of the original hard edge with a radial filter [23]. For corners, for a smooth thresholding to be equivalent to convolution, theoretically I have to do more work, using the distance to the two closest edges and a 2D lookup table. However, this is not necessary for good results [23]: in practice simpler rules with approximately the right behaviour suffice. For example, I can use the minimum distance as the implicit representation of a corner [57] (which is suggested by the usual implementation of CSG operations on implicit representations), or the thresholded edge functions can be combined with multiplication [38]. The important thing about antialiasing is to bound the bandwidth of a signal *before* sampling it, not to simulate convolution exactly. It is also not necessary to use the distance function, a fact that can be exploited to simplify the representation or to compute various special effects, such as mitering. Any implicit function with the same zero set can be used, although its gradient should be normalized to unity if we plan to control the width of the transition region [33].

The simplest smooth transition function is a clamped linear ramp, which in 1D would be equivalent to convolution by a box filter. A better choice is the smooth cubic curve given by the formula:

$$\text{smoothstep}(g) = \frac{1}{2} + g_c\left(\frac{3}{2} - 2g_c^2\right), \tag{4.1}$$

where g_c is g clamped to the interval $[-1,1]$. This is equivalent to a parabolic filter.

Let f be a signed distance function. Relative to the texture coordinates (u,v), the gradient $\nabla f(u,v)$ of the signed distance function has magnitude 1. It points towards the closest point on the contour on the inside (negative distances), and away from the closest point on the outside (positive distances). For implicit representations that are not true distance functions but whose gradient does not vanish, the gradient can be explicitly normalized to unit length.

The normalized texture-space gradient can be used to implement inexpensive anisotropic antialiasing. The texture-space gradient can be transformed into screen space coordinates (x,y) as follows:

$$\begin{aligned} \nabla f(x,y) &= \begin{bmatrix} \partial f/\partial x \\ \partial f/\partial y \end{bmatrix} && (4.2) \\ &= \begin{bmatrix} \partial u/\partial x & \partial v/\partial x \\ \partial u/\partial y & \partial v/\partial y \end{bmatrix} \begin{bmatrix} \partial f/\partial u \\ \partial f/\partial v \end{bmatrix} && (4.3) \\ &= \mathsf{J}_{x,y}(u,v)\nabla f(u,v). && (4.4) \end{aligned}$$

where J is the Jacobian of the (possibly procedural and usually nonlinear) transformation from texture space to screen space. GPUs provide the ability to approximate these derivatives using differencing in fragment shaders, although they could also be computed exactly using automatic differentiation.

Figure 4.1: *Isotropic filtering vs. anisotropic filtering.*

The gradient vector gives the direction of the maximum rate of change of an implicit function. I can therefore compute the width of the transition region based on the magnitude of ∇f in screen space, then reparameterize a smooth step function to get the desired final intensity value I,

$$s = \sqrt{(\partial f/\partial x)^2 + (\partial f/\partial y)^2}, \tag{4.5}$$
$$I = \text{smoothstep}(f/2sw), \tag{4.6}$$

where w is the desired width of the transition region in pixels.

By contrast, isotropic antialiasing, as is typically used in MIP-mapping [67], computes the scale factor using

$$m_x = \sqrt{(\partial u/\partial x)^2 + (\partial v/\partial x)^2}, \tag{4.7}$$
$$m_y = \sqrt{(\partial u/\partial y)^2 + (\partial v/\partial y)^2}, \tag{4.8}$$
$$s = \max(m_x, m_y). \tag{4.9}$$

The isotropic computation of s estimates the width of a worst case filter. Isotropic antialiasing is often too conservative and can cause severe blurring when the transformation is anisotropic, for instance under perspective foreshortening for oblique views near silhouette edges, or for highly nonlinear texture deformations (see Figure 4.1). Isotropic antialiasing can severely degrade the readability of text in particular. Fortunately, when the gradient of the distance function is available, anisotropic antialiasing is inexpensive, and produces a filter of constant screen-space width.

4.4 Features and Distance Functions

Each contour in a glyph is a piecewise path. The pieces of the path are line, quadratic curve, and cubic curve segments. In this chapter, I consider primarily line segments. Quadratic and cubic curves are adaptively approximated by line segments. I also present some preliminary results based on computing the distance to quadratics, but computation of distances to line segments is simpler and faster.

To construct a signed distance function for a contour, I break it down into geometric *features* in such a way that the distance function for the contour can be found as the distance to the closest feature. I consider two primary features: individual line segments, and "corners" consisting of two half-segments meeting at a vertex.

For any feature, I want to compute not only the distance, but the gradient of the distance function, a sign to be used in an inside or outside test, and optionally a "pseudodistance" to be used for mitering (see Figures 4.2 and 4.5). The pseudodistance is the distance to the closest point on the infinite line containing the closest segment; the true distance takes the segment endpoints into account.

The choice of features and distance function evaluation techniques for them is a separate decision from the rest of my system. However the distance field is generated, it can be used for antialiasing and glyph placement as described later.

Any 2D curve can be described parametrically as a function $\mathbf{P} : \mathbb{R} \mapsto \mathbb{R}^2$. Given a test point $\mathbf{Q}$, I can solve for the parameter t^* that minimizes $|\mathbf{Q} - \mathbf{P}(t^*)|$. For a curve *segment*, I have to consider the endpoints as well. Without loss of generality, let the endpoints of a curve segment be $\mathbf{P}_0 = \mathbf{P}(0)$ and $\mathbf{P}_1 = \mathbf{P}(1)$. If $t^* \in [0,1]$, then the closest distance is given by $|\mathbf{Q} - \mathbf{P}(t^*)|$, otherwise it is given by $\min(|\mathbf{Q} - \mathbf{P}_0|, |\mathbf{Q} - \mathbf{P}_1|)$.

For line segments, by clamping the value of t^* to $[0,1]$ I can reconstruct the true distance function and gradients, taking endpoints into account. If I do not clamp, I compute the pseudodistance. It is also convenient to compute only the squares of the distances and compare these, taking a single square root of the minimum distance to all features under consideration. Likewise I can defer certain operations, such as normalization of the gradient vector, until after I have found the closest of a set of features.

4.4.1 Line Segments

Figure 4.2 shows a glyph represented with line segments and the various fields associated with it.

It is relatively inexpensive to compute the distance to line segment features. However, I also have to compute the sign of the distance to the contour, which depends on whether my test point is on the inside or the outside of the contour. Unfortunately, certain difficulties arise at corners.

A naive approach to computing the sign is to determine the closest line segment, and then use the sign of the plane equation of that segment. However, this approach may fail at corners since the distance to a shared vertex will be the same in the region bounded by lines perpendicular to each segment, shown as c_1 to c_2 in Figure 4.3. This may result in the wrong sign being computed in the shaded regions shown in that figure.

One inexpensive (but inelegant) solution to this problem is to "shrink" the line segments by a small amount, putting small gaps in the contour. In Figure 4.3, the gap induces a separating plane at f that bisects the corner. Unfortunately, if the gap is too large, it can introduce artifacts in the rendering, and if it is too small, the angle of the separating

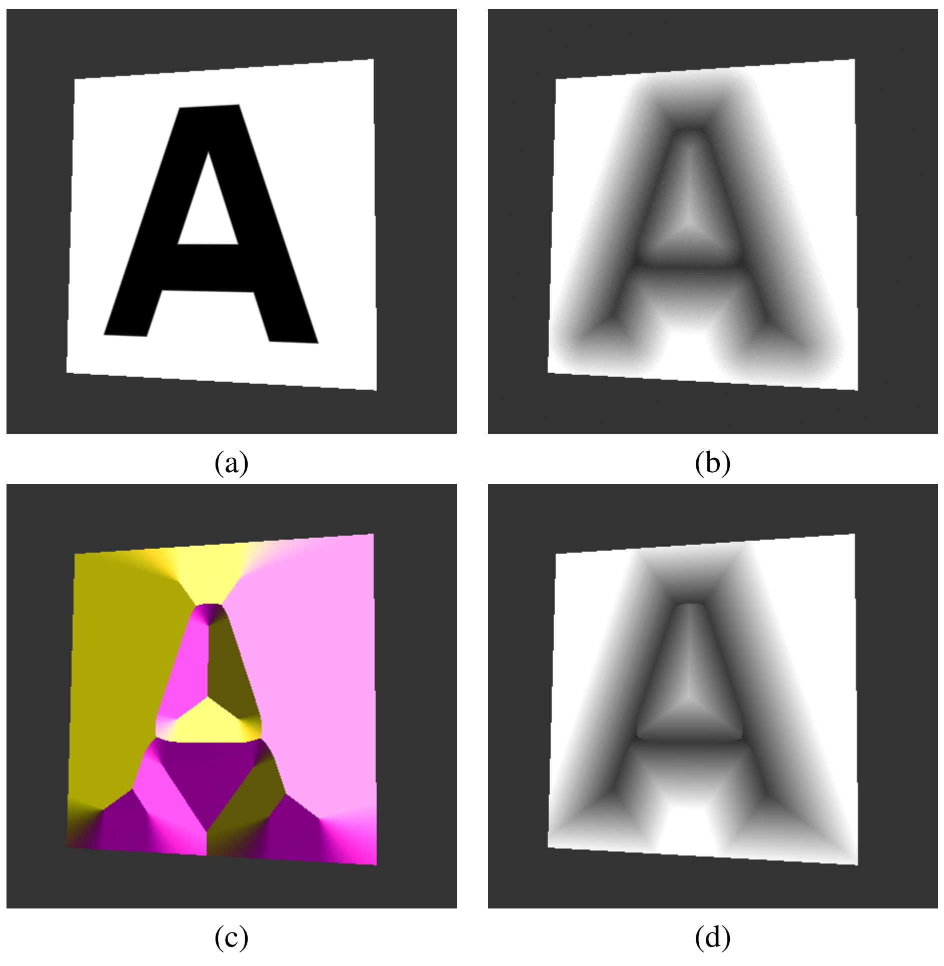


Figure 4.2: *(a) A glyph with 11 line segments, (b) its distance field, (c) the gradient of its distance field, and (d) its pseudodistance.*

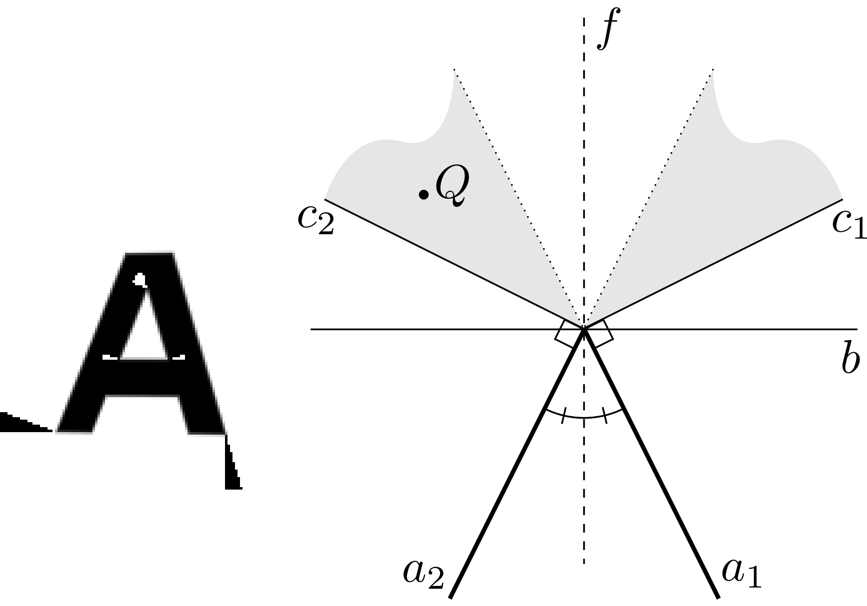


Figure 4.3: *Errors caused by vertex ambiguity. The inside or outside test for the points in the grey regions can be incorrect because the distances from each point to the endpoints of both lines are equal. However, only one of the two choices is correct.*

planes between the endpoints can be inaccurate (especially if the endpoints are stored at low precision). To get the correct bisection angle (important in mitering), the shrink distance has to be the same on both sides of a vertex. Care also has to be taken not to reverse or eliminate very short line segments, which often arise as the result of curve subdivision.

Another approach is to use the pseudodistance to break ties: above line b, the *maximum* absolute pseudodistance gives the *closest* line segment (consider that the pseudodistance is the distance to extensions of the segments a_1 and a_2). Below c_1 and c_2, the true distance gives the correct answer, and below b, the maximum absolute pseudodistance rule is wrong, so I need to *not* use it. To avoid pixel dropout, I need to switch between the two rules, where they are both correct. Using this disambiguating rule unfortunately results in a distance computation that requires about twice as many arithmetic operations as using shrunken line segments. A line segment is described by its endpoints $\mathbf{P}_0$ and $\mathbf{P}_1$. Parametrically, points along the line can be generated by linear interpolation between these points:

$$\mathbf{P}(t) = (1-t)\mathbf{P}_0 + t\mathbf{P}_1. \tag{4.10}$$

For points on the line segment, t must lie in the range $[0,1]$. Given a test point $\mathbf{Q} = (u,v)$, the parameter of the closest point on the (extrapolated) line can be found using the

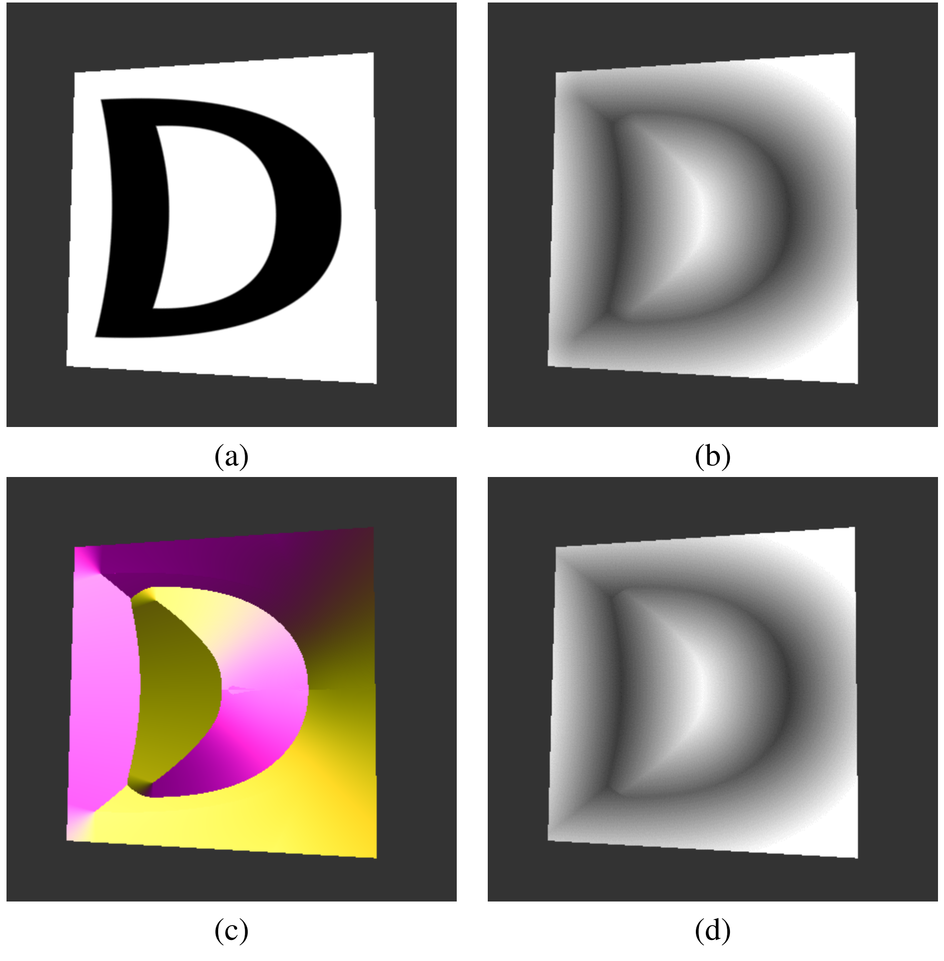


Figure 4.4: *(a) A glyph made with six quadratic segments, (b) its distance field, (c) the gradient of its distance field, and (d) its pseudodistance.*

following computations:

$$\vec{\mathbf{d}} = \mathbf{P}_1 - \mathbf{P}_0, \quad (4.11)$$
$$\vec{\mathbf{q}} = \mathbf{Q} - \mathbf{P}_0, \quad (4.12)$$
$$t^* = \vec{\mathbf{d}} \cdot \vec{\mathbf{q}} / \vec{\mathbf{d}} \cdot \vec{\mathbf{d}}. \quad (4.13)$$

Note that no square root is required, only a division. However, t^* may not lie in the interval $[0,1]$, so a clamped value should be generated: $t_c^* = \min(1, \max(0, t^*))$.

Then, the squares of the distance g and pseudodistance h can be calculated as

$$g^2 = (\mathbf{Q} - \mathbf{P}(t_c^*)) \cdot (\mathbf{Q} - \mathbf{P}(t_c^*)) \quad (4.14)$$
$$h^2 = (\mathbf{Q} - \mathbf{P}(t^*)) \cdot (\mathbf{Q} - \mathbf{P}(t^*)) \quad (4.15)$$

To compute the sign (that is, determine which side of the line the point $\mathbf{Q}$ is on), I can take the dot product of $\vec{\mathbf{q}}$ with the normal:

$$\vec{\mathbf{n}} = (d_y, -d_x), \quad (4.16)$$
$$s = \text{sign}(\vec{\mathbf{n}} \cdot \vec{\mathbf{q}}), \quad (4.17)$$

where

$$\text{sign}(a) = \begin{cases} -1 & : \quad a < 0 \\ 0 & : \quad a = 0 \\ 1 & : \quad a > 0 \end{cases} \quad (4.18)$$

Note that I do not have to normalize $\vec{\mathbf{n}}$ to unit length to get the correct sign. However, if I am willing to do so, an alternative way to compute the pseudodistance along with the sign is

$$\hat{\mathbf{n}} = \vec{\mathbf{n}} / |\vec{\mathbf{n}}|, \quad (4.19)$$
$$h = \vec{\mathbf{q}} \cdot \hat{\mathbf{n}}. \quad (4.20)$$

All the dot products in these computations are on two-tuples, but GPUs use four-tuple registers. However, I often compute distances to several line segments at once before comparing their magnitudes. A useful optimization in practice is to use "vertical" as well as "horizontal" SIMD computations. For instance, I can use four-tuple operations to compute two two-tuple operations in parallel, or four scalar operations to compute the distance to four line segments in parallel. Some GPUs can also co-issue two two-tuple instructions in a single cycle.

4.4.2 Corners

Corners are pairs of line segments meeting at a common point. A closed contour expressed as a sequence of N line segments can also be expressed as a sequence of N corners by dividing all line segments at their midpoints. Although it is slightly more

expensive to compute the distance to a corner, and corners take more parameters to represent, corners do not suffer from ambiguity about which one is closer, since they always meet with derivative continuity.

Corners are specified with three points: endpoints $\mathbf{P}_0$ and $\mathbf{P}_2$ and vertex $\mathbf{P}_1$. I compute the direction tangents, then use these to compute the normal of the bisecting line:

$$\vec{\mathbf{d}}_0 = \mathbf{P}_0 - \mathbf{P}_1, \quad (4.21)$$
$$\vec{\mathbf{n}}_0 = (d_{y,0}, -d_{x,0}), \quad (4.22)$$
$$\hat{\mathbf{n}}_0 = \vec{\mathbf{n}}_0 / |\vec{\mathbf{n}}_0|, \quad (4.23)$$
$$\vec{\mathbf{d}}_1 = \mathbf{P}_2 - \mathbf{P}_1, \quad (4.24)$$
$$\vec{\mathbf{n}}_1 = (d_{y,1}, -d_{x,1}), \quad (4.25)$$
$$\hat{\mathbf{n}}_1 = \vec{\mathbf{n}}_1 / |\vec{\mathbf{n}}_1|, \quad (4.26)$$
$$\vec{\mathbf{d}} = \hat{\mathbf{n}}_0 + \hat{\mathbf{n}}_1, \quad (4.27)$$
$$\vec{\mathbf{n}} = (d_y, -d_x). \quad (4.28)$$

I make both tangent vectors point away from the corner point. Averaging the perpendiculars of the direction tangents rather than the tangent vectors themselves avoids a degeneracy when the corner's vertices are colinear. Two normalizations are required to get the actual bisector. Once I have the normal of the bisector, though, it does not have to be normalized.

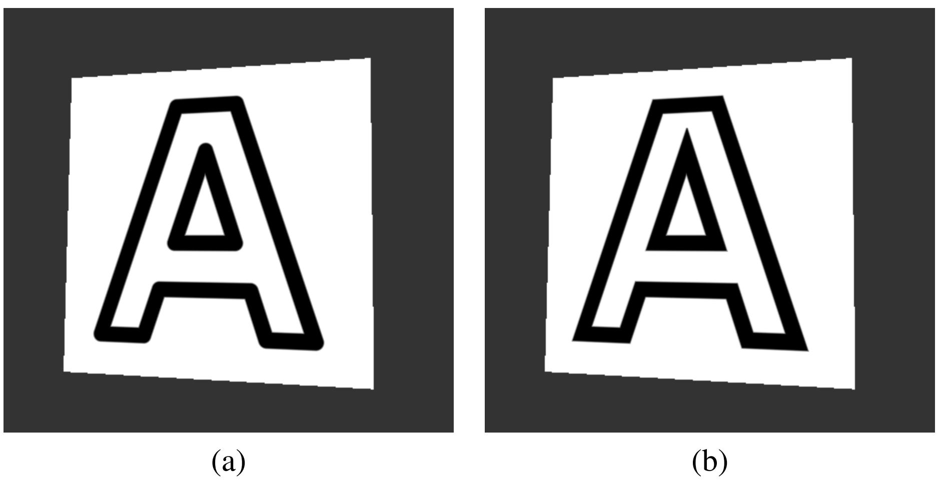
Then, I compute a vector from the center vertex of the corner to the test point $\mathbf{Q} = (u, v)$, and test this against the bisector normal:

$$\vec{\mathbf{q}} = \mathbf{Q} - \mathbf{P}_1, \quad (4.29)$$
$$s = \vec{\mathbf{q}} \cdot \vec{\mathbf{n}}. \quad (4.30)$$

If s is positive, then I compute the distance to the line segment given by $\mathbf{P}_0$ and $\mathbf{P}_1$, otherwise I compute the distance to the line segment given by $\mathbf{P}_1$ and $\mathbf{P}_2$, reusing the vector $\vec{\mathbf{q}}$. If $\vec{\mathbf{n}}$ is precomputed, the extra cost of computing the distance to a corner relative to a line segment is one dot product and a conditional assignment—plus the cost of storing the additional center point and the precomputed bisection normal. A corner feature therefore requires between 1.5 and 2 times as many stored values as a line segment feature. However, since there is no ambiguity problem, corners can potentially use lower precision for storage compared to shrunken line segments, which can result in an equivalent storage cost.

4.4.3 Quadratics

For higher quality, I can consider features based directly on polynomial curves. Glyphs in standard font formats use both quadratic and cubic segments, although quadratics are far more common. I only consider distances to quadratic segments at this stage of my research; a test glyph is shown in Figure 4.4. Unfortunately, solving for the nearest distance to a quadratic polynomial curve requires finding the roots of a cubic polynomial.



(a) (b)

Figure 4.5: *Using a smooth pulse rather than a smooth step gives antialiased outlines. (a) Using the true distance gives rounded outlines. The true distance takes the segment endpoints into account. (b) Using the pseudodistance gives sharp miters. The pseudodistance is the distance to the closest point on the infinite line containing the closest segment.*

I have found that it is most convenient to do this with an iterative root solver, since the analytic solution to a cubic equation requires cube roots and a case analysis (which I can avoid with suitable starting conditions on my own iterative solver). Quadratic splines, like corners, require a minimum of six coordinates to specify. It is also possible to speed up their computation by precomputation, but then twelve numbers are required for every quadratic, which requires three texture elements to store.

Quadratics suffer from the same problem as line segments: distances to endpoints can be ambiguous. As with line segments, either endpoint shrinking or midpoint subdivision and grouping into corners can be used to resolve this.

4.4.4 Performance

At the resolution of the results in this chapter, roughly 512×512, and on my test machine (Pentium 4 2.6GHz and an NVIDIA 7800GT GPU), a glyph with 11 line segments (the A glyph shown in the figures) runs at 150fps without explicit disambiguation (using shrunken line segments) and 80fps with pseudodistance disambiguation. Using 11 corners and precomputed separation planes, the same glyph runs at 60fps. Surprisingly, the 6 quadratic polynomial features used in the curved test glyph (the D glyph shown in the figure) runs at 70fps at the same resolution. In practice I find the quality of representations computed with linear features to be adequate. For the rest of the results in this chapter, I use only line segment features and the “shrinking” approach for disambiguation.

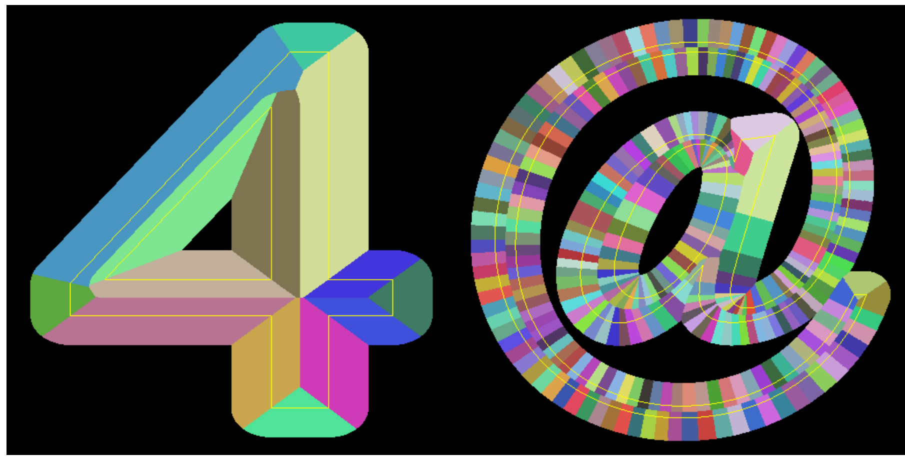


Figure 4.6: *Voronoi analysis of some TrueType glyphs. Each color represents one line segment.*

4.5 Voronoi Grid Accelerator

The test renderings and performance numbers given so far are generated using a brute-force approach, comparing the distances to all features in all contours of a glyph. This is not practical for real glyphs, and certainly not for a page of text. I therefore have to accelerate the computation by limiting the number of features considered at each texel.

4.5.1 Voronoi Analysis

To build an accelerated representation of the glyphs in a font, I read in the contour information using the FreeType library. I adaptively approximate quadratics and cubics by line segments, subdividing the splines recursively until a given error bound is met. I then compute the Voronoi diagram of each glyph using hardware acceleration [29]. The hardware acceleration approach to drawing a Voronoi diagram can also suffer from endpoint ambiguity, so I shrink endpoints to resolve it (this problem would not arise if corners were used as features). Some example analyses are shown in Figure 4.6. I compute the Voronoi diagram only within some finite distance d of the contour. I also stretch each character non-uniformly to fit a square with a minimum margin of d. The Voronoi cones I draw are non-uniformly scaled so the distance computation is still correct even though nonuniform scales are non-Euclidean: I get a scaled image of the Voronoi diagram of the unscaled contours. It would also be possible to use a brute-force shader computation to compute this diagram, to use a geometric (CPU only) Voronoi analysis, or to compute the diagram for corner or polynomial features.

4.5.2 Grid Packing

Once the Voronoi diagram is computed, it is overlaid with a regular grid. Within each cell of the grid, I make a list of all the features associated with regions in that cell. These are the only features that need be considered when computing the minimum distance to any point in this cell. Also, there are cells that are of distance greater than d from any contour. These cells are black in the diagram, but I perform an additional test on the CPU to determine if they are completely inside or completely outside of a region. In a `flag` texture, I store (in a suitably biased fashion) -1 for cells completely inside, 0 for cells on the boundary, and 1 for cells completely outside. Then, in a `feature` texture, I store the parameters of features. These can be clipped and quantized, since I do not care about features or parts of features more than distance d away. Note that only one feature can fit in each cell of the `feature` texture. However, adjacent cells often refer to the same features. In my shader, I will look at not only the features stored in the cell containing the test point, *but also at a number of neighbours* (four is reasonably fast, but nine can also be used). I use a simulated annealing process to assign features to texels of the `feature` texture so that all the required features for each cell will be accessed by the shader. Note that it is acceptable for a feature to be accessed and evaluated even if it is not necessary, a fact I exploit to avoid conditionals. The `flag` texture marks texels that are completely inside or outside, and more than distance d from the boundary. I multiply the value in `flag` by a large number and add it to the computed minimum distance. Thus, I can store "extra" features in these texels as well. The distance computation from these cells will be incorrect, but I am going to swap it with a value of the correct sign (and then clean up the distance field with a clamp), so it does not matter. This provides extra storage for features that will not fit near a contour. I also make sure that the border of the glyph is bounded by cells marked as being outside the contour. Figure 4.7 shows how the Voronoi diagram of a glyph is packed in a texture.

At a resolution of approximately 512×512, using four features per sample, a single magnified glyph renders at about 100fps on my test machine.

4.5.3 Multiresolution

It is possible that the features for complex glyphs cannot be "packed" at a given resolution. I therefore start at the lowest possible resolution and, if I cannot successfully pack the features at that resolution, increase the resolution of the accelerator in power-of-two steps until I find a packing. Due to subdivision, curved edges can result in a large number of features, whereas glyphs with straight edges can be stored at a relatively low resolution. If I want to use curved features, the process would be the same but the accelerator resolution required would be smaller, since there would be a smaller number of features required for similar quality.

Different glyphs may now require accelerator structures of different resolutions. I pack glyphs together into a quadtree, taking note of the scale factor and offset of each glyph. This latter information is not stored in a texture; rather, it is retained in a CPU

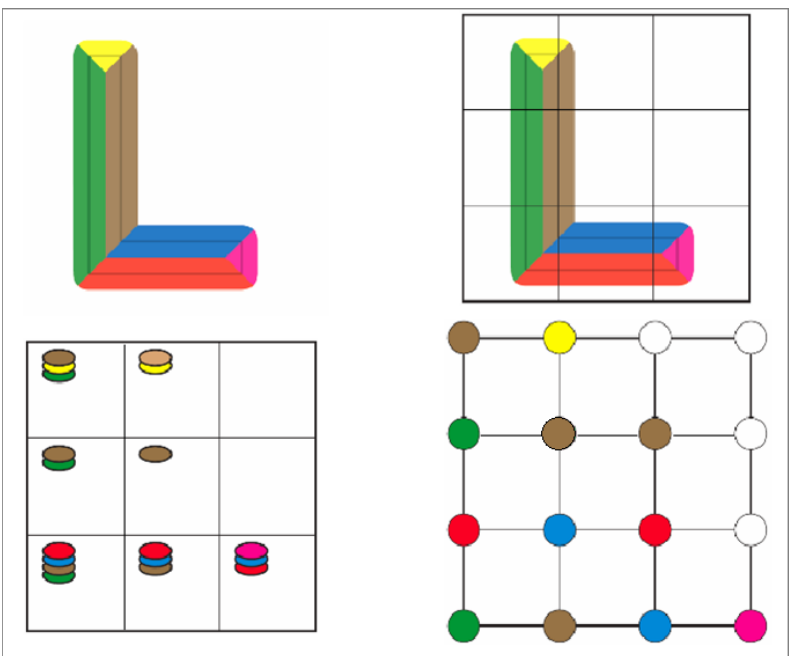


Figure 4.7: *Example showing the Voronoi diagram packed in a texture using simulated annealing. The Voronoi diagram of a glyph is first generated. Then a uniform grid is superimposed over the Voronoi diagram. Within each cell, the potential closest features to some point in the cell are listed. They are represented by different colors here. Simulated annealing put the closest features of each cell at its corners. Common features between cells can be shared.*

data structure and used later by the CPU when placing sprites, as discussed in the next section.

The analysis process takes about one second per glyph. This is not fast enough to be done in real time, but is fast enough that a font subset can be computed for individual textures during a preprocessed encoding of a given vector texture.

It is possible but rare that the analysis process will fail if the Voronoi diagram contains a vertex of high outdegree, where more regions are adjacent than the number of features accessed per pixel. In this case, I can

1. consider a larger number of features per pixel (globally);
2. accept an approximation of the distance field around that vertex, since they are usually far from the contour; or
3. on newer GPUs, use conditional execution to consider more features only around this point.

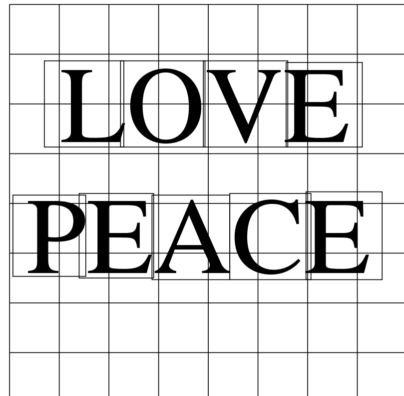
Use of conditional execution maintains high performance on average since extra computation will only be required in small regions.

Figure 4.8: *Some examples showing embossed glyphs. The gradient of the distance field is used to perturb the normal. In the top example, the inset image shows creases along line segment junctions when the glyph is magnified to a certain degree*

4.6 Glyph Sprite Mapping

The output of the previous step is a multiresolution font table, which is visualized on the right of Figure 4.9. Now I wish to instance multiple glyphs on a document. I use a sprite table for this (Figure 4.9, left). The size of a cell in a sprite table must be less than the minimum distance possible between glyphs. For text, this can be computed from the

advance distance for each glyph and the kerning information containined in the font file, as well as the desired minimum text size.



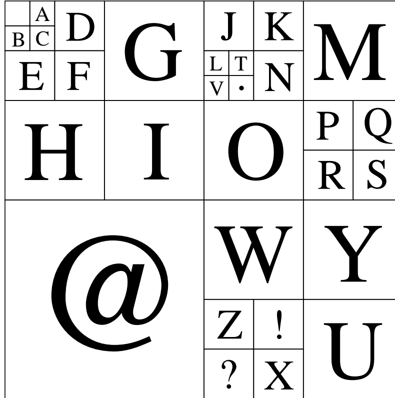


Figure 4.9: *Sprite and font tables.*

In each sprite cell, I store the 2D offset of a sprite that (partially) covers that cell, the 2D location of the origin of that sprite in the font table, the two factors by which I wish to scale the glyph in the vertical and horizontal directions (the ratio of these should be calculated to return a glyph to its original aspect ratio so that distance calculations are correct), and the total scale factor of the glyph relative to its absolute size in a canonical coordinate system (so I can correct the distance computations for each glyph and make them comparable). I must store this information in every cell covered by a sprite's bounding box, except for the cells on the right that are only partially covered. Two textures of four components each are required.

Now, to sample this "virtual" texture, I look up the sprite information in the cell containing the sample point *and the cell to the immediate left* of that cell. For each of pair of sprites, I access the appropriate set of features in the font table and compute the minimum distance to them, then compute the minimum distance of these two distances. By clamping the offsets in the font table to normalized coordinates for each cell, I can avoid picking up features from adjacent glyphs. The boundary of empty cells around each glyph in the font table gives us white space around each glyph. Sprite table cells that are not used must refer to the "blank" glyph in the font.

This process lets two sprites overlap horizontally. In my implementation, I consider four features per glyph, requiring the computation of eight point to feature distances. Example renderings are shown in Figure 4.10 to Figure 4.13; Performances of these scenes at a resolution of 512×512 are about 60fps on my test machine.

If I look at four sprites at once, one to the right, one below, and one to the right and below, and omit the top row of partially covered cells when placing sprites, I can also handle vertical overlap. However, this is not (usually) needed in normal text rendering, and doubles the cost. Likewise, I could add extra information to the sprites, such as rotations, to handle more general cases. The ability to rotate glyphs and permit four-way overlap might be useful in the rendering of map labels, for instance.

Figure 4.10: *A map using a vector texture for labels, with an enlargement of a distorted region to demonstrate anisotropic antialiasing.*

4.7 Summary

I have presented a technique for rendering outline font glyphs directly on the GPU as encapsulated procedural textures, and for placing multiple instances of these glyphs anywhere in a texture, subject to some constraints on overlap. My approach is based on *accurate* computation of distance fields and supports efficient anisotropic antialiasing. I have introduced a new representation for vector graphics in general, in which features of a vector graphics image are analysed using a Voronoi diagram and then packed into a grid under the assumption of lookup over a neighbourhood to avoid redundancy. My approach can be thought of as a generalization of texture sprites to implicit function generators.

For many purposes, font glyphs can be prerendered and placed in a texture map. Even then, the sprite approach may be useful for rendering pages of text using compositing rather than minimum distance computations. It is also possible to prerender sampled distance fields, which would still support efficient anisotropic antialiasing. However, the accurate procedural approach provides the assurance that no matter what 3D distortion or magnification is applied, the edges will always be crisp and the corners sharp. Since my representation gives the accurate distance field rather than an approximation, applications such as outlining and embossing, which rely on an accurate computation of the distance field far from the contour, are also possible.

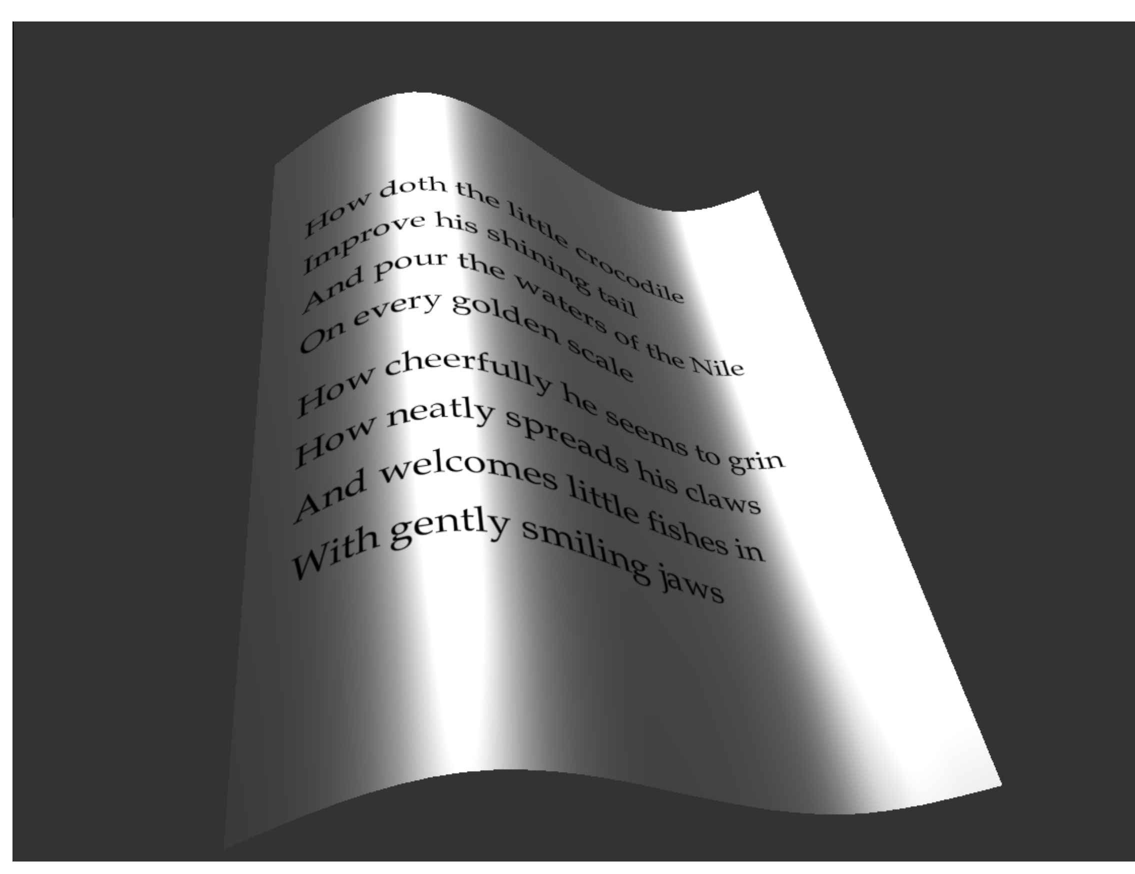


Figure 4.11: *Some example "documents" with multiple glyphs.*

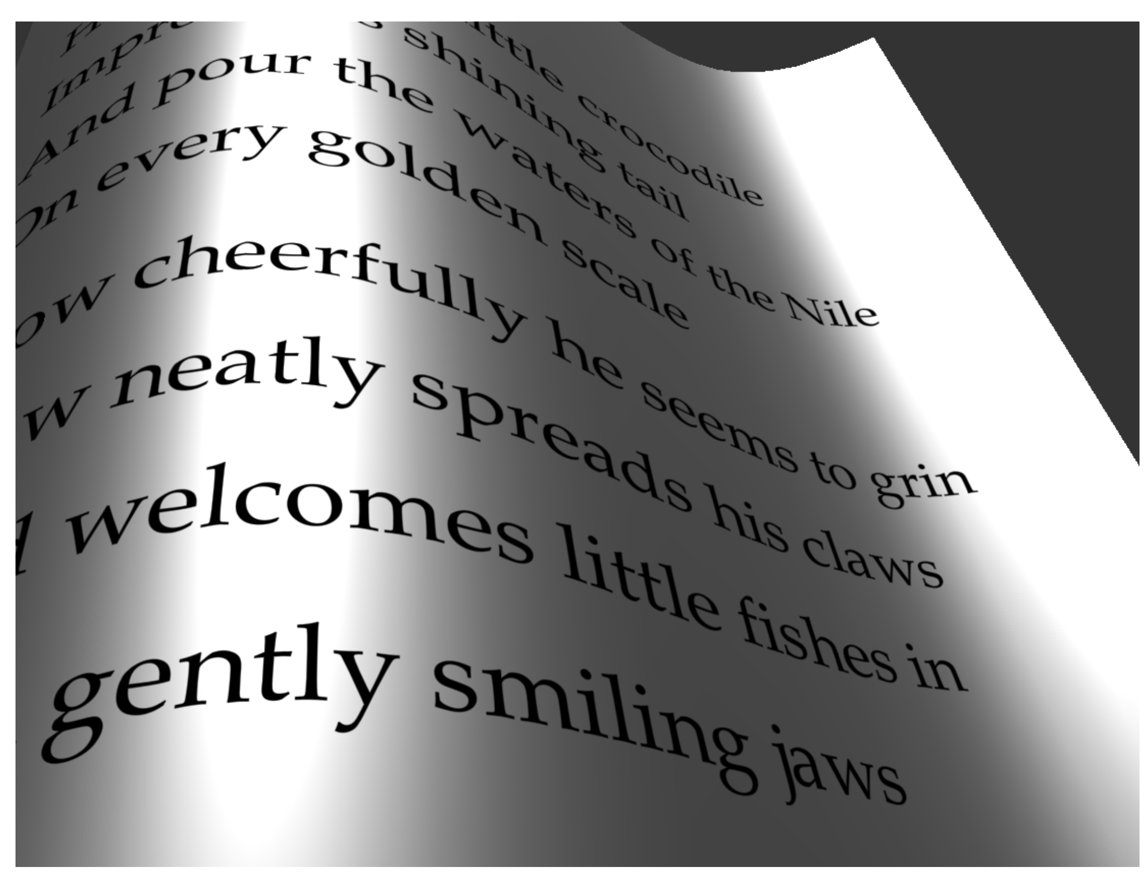


Figure 4.12: *Some example "documents" with multiple glyphs.*

Figure 4.13: *Closeup example "documents" with multiple glyphs.*

I have demonstrated that the storage and computational costs of vector textures are feasible. In fact, at high magnifications the bandwidth requirements of vector textures are lower than raster images of equivalent resolution would be, although of course the computational requirements are higher. However, since GPU computational performance is scaling much faster than bandwidth, ultimately image representations using more computation but less bandwidth will prove superior.

Chapter 5

Planar Regions and Arc Features

In the second stage of my work, I extend my previous research to support the features of general scalable vector graphics (SVG). SVG files support regions of different colors and borders, and because this file format is supported by a broad range of content creation tools, they have much broader application than bilevel vector images. Many artistic designs are not constrained to two colors and simple borders. SVG images are much more complicated. They can support many different kinds of paths, strokes, and also allow region overlap and occlusion. Although I still use the closest distance to the boundary for antialiasing, the color decision is more complicated. The rule for computing the color at a point needs to consider all the factors including gradient color, path color, stroke color, and occlusions. My new mechanism should support all these features, while still permitting fast random access.

5.1 Introduction

In my previous research, curves, including quadratic splines, cubic splines and ellipses, are all approximated by line segments. The advantage of line segment features is that they support an easy distance computation. The disadvantage of line segment approximations is that they only support C^0 continuity. If a static approximation is used, when the textures are magnified by a large scale, visible creases will be observed along the line segment junctions as in Figure 4.8, thus reducing the resolution-independent advantage of vector textures. The artifacts are more obvious in embossing examples, especially when a glossy reflectance model is used. Also, many line segments will be needed in general to approximate one spline curve, and this will increase storage space requirements.

I have two goals in this stage of work. First, I want to develop a mechanism that supports, if not all, then most of the features in SVG images, and can be extended ultimately to support all of them. My prototype supports multiple paths, solid fill colors, and gradient colors, which I feel is a sufficient foundation. For example, I do not support

raster fills, but by supporting gradients I demonstrate that I can generate the necessary coordinates for such an extension.

Second, I want to improve the smoothness of the approximation at least to C^1. Since distance computations to both quadratic and cubic splines are complicated (as discussed in Chapter 3), I decide to use circular arcs as an approximation primitive. Algorithms have already been developed to approximate arbitrary curves, including parametric polynomial splines, with arc splines (sequences of circular arcs), within a specified error tolerance [72, 71, 60, 69, 61, 39, 62]. Arc splines are used in many industries, such as computer aided geometric design, highway route design and architecture.

Choosing circular arcs was motivated by the fact [72, 71] that in arc splines the arc tangents on both sides of every junctions are the same, so this approximation gives C^1 (actually G^1, but it can be reparameterized to give C^1) continuity. Artifacts in embossing examples with line segment approximation are caused by the discrepancy in tangents along the junctions. Since C^1 continuity gives the same tangents at the junctions, the shading results will be smooth, at least for less glossy shading models. Another advantage of C^1 continuity of arc splines is that if I texture map the borders with dashed lines, the lengths of the dashes can be maintained consistently with the parameter. Also, computing the distance to a circular arc is nearly as inexpensive as computing the distance to a line segment, which will be shown later. The offset curves of arc splines are still arc splines. My representation of circular arcs can also represent line segments exactly as a special case. For a given error tolerance, fewer arc segments than line segments are typically required to approximate a general curve, which can reduce storage requirements. This approximation is not restricted to image magnification. It can be used in any situation where the distance to a curve is needed, for instance in the computation of offset curves in machining or force feedback.

In the rest of this chapter, circular arcs and arc splines are first introduced, then the mechanism for supporting multiple features of SVG files are discussed. As I will discuss at the end of this chapter, I do not find this mechanism an ideal one, and these problems motivate the next stage of my research.

5.2 Arc Splines

A curve represented with a sequence of arcs is called an *arc spline*. I use an approach by Yang [72, 71] to approximate a polynomial spline with an arc spline, This approach can approximate any parametric curve with arc splines. He starts by sampling the polynomial spline finely using an adaptive parameter step. Then he uses two arcs joined with tangent continuity (a *biarc*) to approximate each segment. Like a cubic spline segment, in a biarc the tangents of the endpoints can be chosen independently. If any segment cannot be approximated within a given error threshold, it is subdivided. Once the whole spline is approximated with biarcs, a merging step is applied to merge every three arcs into two arcs. The merging starts from the first three arcs; then each new pair of arcs in a biarc will be merged with the next arc. The merge step is executed recursively and stops when the approximation error is over the threshold, or when the whole spline is approximated.

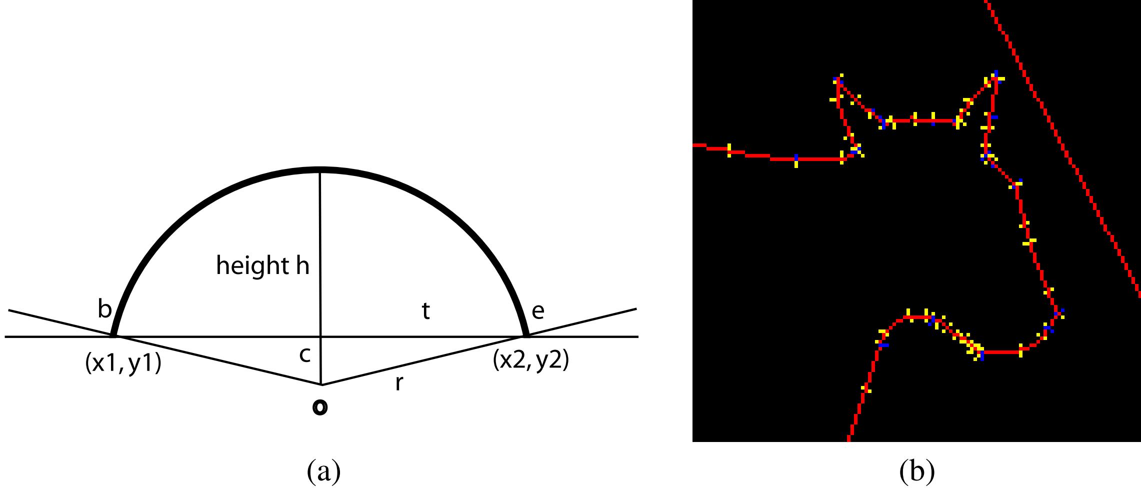


Figure 5.1: *(a) An arc feature is parameterized by its endpoints and the midpoint height relative to its chord. (b) The spline boundaries of a cow are approximated by circular arcs*

When an arc spline is used to approximate a segment of a polynomial spline curve, this algorithm guarantees that the tangents at the endpoints are the same as those in the original segment. The tangents at the joints will therefore be consistent so the entire resulting curve has C^1 continuity if the original curve was C^1 continuous. Elliptical arcs can also be approximated with arc splines similarly, although except for actual circular arcs, they cannot be represented exactly.

5.3 Feature Representations

In my representation, spline curves and elliptical arcs are approximated by circular arcs. Line segments are kept as line segments. Five parameters are required to represent circular arcs. The first four parameters are used to record the coordinates of the two endpoints: $\mathbf{b} = (x_1, y_1)$ and $\mathbf{e} = (x_2, y_2)$. By representing endpoints explicitly, my representation is robust under quantization of these endpoints, since endpoints of different arcs can be quantized in the same way. The fifth parameter is used to record h, the distance from the highest point on the arc to the chord formed by the two endpoints (I call this the h). The value of h can be positive, negative, or zero. When the concave side of the arc is inside, h is positive; Otherwise, it is negative. In the case of a line segment, h is zero. It is also possible for h to be larger than the radius of the arc. This case happens when the arc is bigger than half a circle (see Figure 5.2).

See Figure 5.1(a) for the representation. Figure 5.1(b) shows the results of a cow sign example, also shown in rendered form in Figure 5.8 and Figure 5.9.

Once I have these five parameters, I can compute the center point (origin) of the circle containing the arc, and use this to compute the closest distance from a query point to the

arc. Let c be the distance from the origin to the chord. The value of c can be negative when h is larger than the arc radius. Then by the Pythagorean Theorem and using the radius $r = |h| + c$ I have

$$\begin{aligned} (|h|+c)^2 &= h^2 + 2|h|c + c^2, \quad (5.1) \\ &= c^2 + t^2, \quad (5.2) \\ 2|h|c &= t^2 - h^2. \quad (5.3) \end{aligned}$$

where t is half the length of the chord, or half the distance between **b** and **e**. I can then solve for c, and use this value to find the origin point **o**:

$$\begin{aligned} c &= \frac{t^2 - h^2}{2|h|}, \quad (5.4) \\ \vec{\mathbf{v}} &= \mathbf{e} - \mathbf{b} \quad (5.5) \\ &= (x_v, y_v), \quad (5.6) \\ \vec{\mathbf{w}} &= (y_v, -x_v)/2t, \quad (5.7) \\ \mathbf{o} &= \mathbf{b} + \vec{\mathbf{v}}/2 + c\vec{\mathbf{w}}. \quad (5.8) \end{aligned}$$

Within the wedge formed by **ob** and **oe**, the distance to the arc from a query point **q** is then $\mathbf{dis_{in}} = \|\mathbf{q} - \mathbf{o}\| - r$ for an arc of radius r.

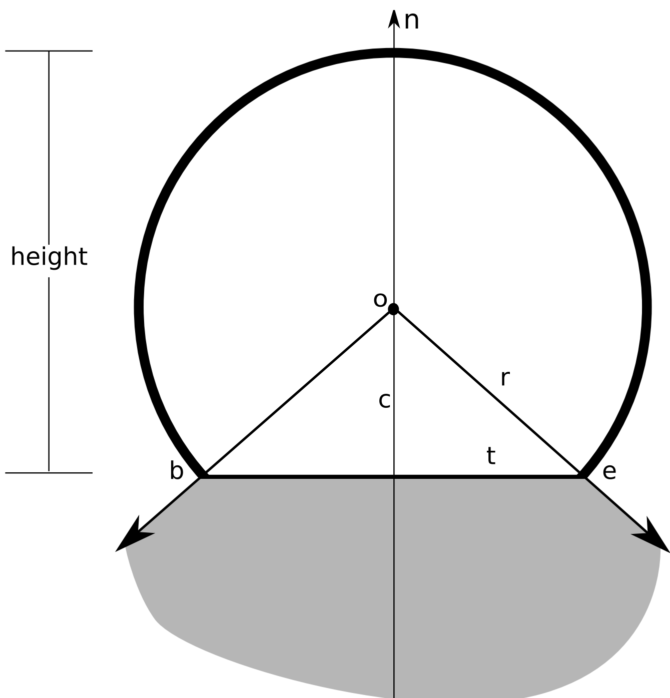


Figure 5.2: *Arcs with large heights can also be represented.*

Outside the wedge I have to use the distance to the endpoints of the arc. I can test if the query point is inside or outside of the wedge using plane equations generated by

the lines $(\mathbf{o},\mathbf{b})$ and $(\mathbf{o},\mathbf{e})$. For example in Figure 5.2, line $(\mathbf{o},\mathbf{n})$ bisects the arc and the shaded region. Suppose the closer endpoint is represented by **d**, and the distance to the closer endpoint by $\mathbf{dis_{out}}$. To find if the query point is outside of the wedge (falls in the shaded region), which endpoint is the closer one, and the closest distance, I use the following pseudo code. If a query is in the shaded area, I extend the tangent line from its closest endpoint, and determine which side the query falls in. If it is on the right side, the distance is negative; otherwise, it is positive.

```
// test if q is in shaded area
out = (right(q,on) && right(q,oe))
    || (left(q,on) && left(q,ob))
// test which endpoint is closer and compute the distance with sign
d = (right(q,on) && right(q,oe)) ? e : b,
dis_out = distance(q,d)
dis_out = dot(qd,od) > 0 ? -dis_out : dis_out
// merge the inside or outside cases
dis = out ? dis_out : dis_in.
```

One intuition about testing if a point falls in the shaded area in Figure 5.2 is to just test if the point is both on the left of **ob** and on the right of **oe**. This simple test however will not work for all cases. In particular, when the angle between **ob** and **oe** is smaller than 90 degree, the test fails.

With the intention of showing this, Figure 5.2 is drawn differently from Figure 5.1(a). In Figure 5.2, height **h** is larger than the radius **r**. The equations given previously for finding the circle origin were based on the case in Figure 5.1(a) when the angle between **ob** and **oe** is smaller than 90 degrees. Using this example I show that the equations are a general formula for all cases. In Figure 5.2, the value of **c** will be negative in equation 5.4, because the absolute value of **h** is larger than the absolute value of **t**. So the origin **o** is on the opposite side of the chord with normal $\vec{\mathbf{w}}$. In this case, the origin **o** is above chord **be** instead of being below it. However, using these formulas, the result is correct.

There are two ways to test if a point is inside or outside of a closed path. Since I am using boundary direction, the arc in Figure 5.1(a) and Figure 5.2 can be either a convex or a concave part of a boundary. To distinguish these two cases, the height h is set to positive if the arc is convex, and negative if concave. The computation of distance is still the same, since the absolute value of h is used. Its sign is only used to flip the sign of **dis**.

I could also improve performance at the expense of storage space by storing six parameters instead of five. Instead of storing height **h**, I could store origin coordinate **o**, thus saving the computation for the origin. Computing the distance to arcs will be as cheap as the distance to line segments. In my implementation, I use five parameters. Note that since five-tuples are not supported in hardware, the implementation uses a structure-of-arrays layout rather than an array-of-structures.

The parameter h can also be expressed relative to the length t so only the ratio h/t is stored, assuming t does not equal zero. This value can then be quantized over a finite

range. Quantization error may degrade exact C^1 continuity but the error can be made as small as desired.

Figure 5.3: *The lion with all the line segments approximated with arc spline computations.*

Line segments can also be represented by the same parameters. As before, the first four parameters record the coordinates of the two endpoints. By checking if the height h equals zero, I can tell if a feature is a line segment or not. If a feature is a line segment, I need to compute the distance to it using an alternative code path, since the above computation would result in a division by zero. Line segments could also be approximated by arcs with carefully chosen large radii, results are shown in Figure 5.3. Since the choice of radius is scale-dependent, I recommend that separate cases be used.

5.4 Flattening of Images

I choose to integrate the arc splines into my previous system. The original algorithm is designed for bilevel images with simple non-intersecting paths. In bilevel images, there are only two colors, a foreground color and a background color. The signed distance can be used to blend the two colors.

This system however cannot be applied to general SVG images directly. Vector graphics file formats include features to support multiple colors, linear and radial gradients, various kinds of strokes, compositing of multiple transparent layers, and even

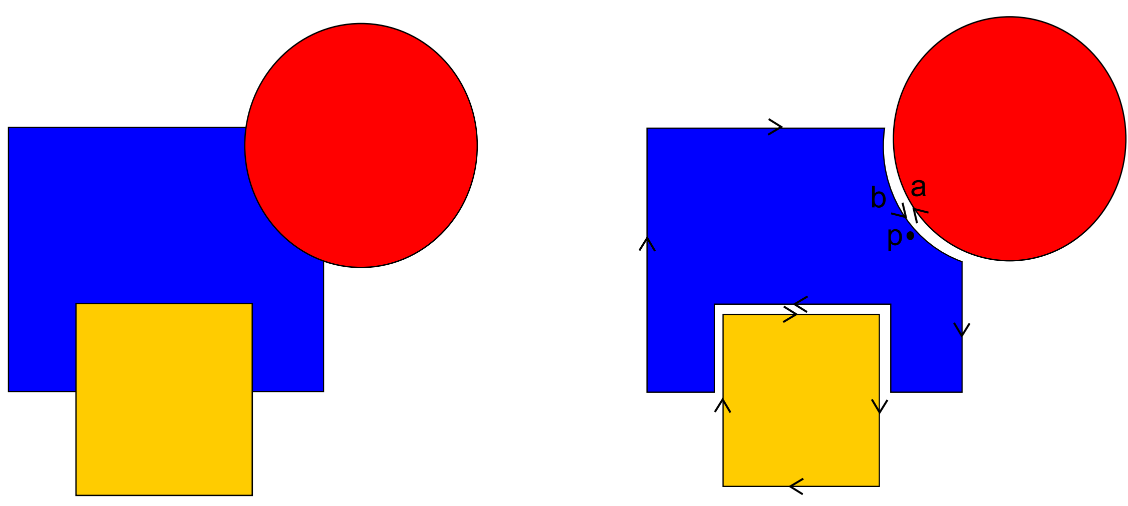


Figure 5.4: *Flattening paths into a planar arrangement.*

inclusion of raster images. A single path is also allowed to intersect itself multiple times, in which case the interior region for that path is determined by the winding rule (for example, the even/odd winding rule or the non-zero winding rule). In the general case, antialiasing along boundaries also involves more than just blending two colors. It could require the blending of multiple colored areas that overlap or touch each other. As an example, consider the center point of a pie chart with n colored regions. Also, using the closest signed distance for the inside or outside test does not work for self-intersecting paths.

Fortunately, most of the problems can be solved by flatterning the images. After flattening, the images have only one layer. Each region is a closed path, and there are no overlaps or occlusions. To flatten a vector image, it is necessary to compute the intersections of all boundaries, segment the existing paths, delete the parts of paths that are hidden, and join the surviving segments together so that every region is outlined by a single path. Ideally, I can find one color (or to be more general, region fill description) for each side of a boundary, and record them together with the single segment splitting the two regions. Since existing software, such as Adobe Illustrator and Inkscape, provide a flattening function that can be accessed by users, I decide to use the built-in tools for testing, even though they were not perfect. But in a "production" system, a more robust tool like that of Eigenwillig et al. [14] could be used.

A simple example in Figure 5.4 demonstrates how the intersection points and new contour paths are generated when a vector image is flattened. The boundary between two regions becomes two paths composed of the same coordinates along the shared part but oriented in opposite directions (if paths along the boundary of each region are traversed with a consistent orientation, in this case clockwise).

In my representation, after flattening the shared boundary is represented twice, once in each direction. Each segment stores not only its shape parameters, but also a pointer to a *region descriptor* for the region to its right. In computational geometry terms, I use

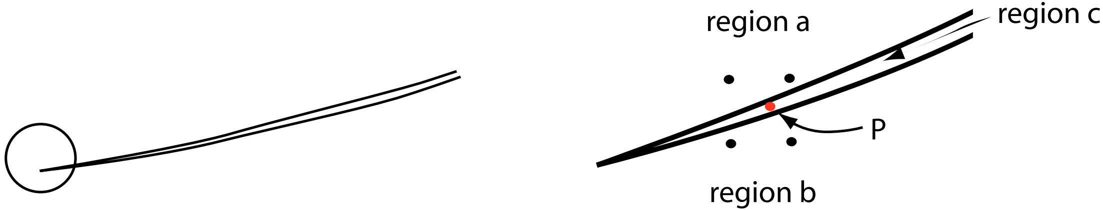


Figure 5.5: *Undersampling the Voronoi diagram can cause inside or outside classification errors.*

a half-edge structure to represent a planar arrangement. A region descriptor contains all the information needed to compute the color at any point in the interior of a region. In general, this could include different color types and even compositing of multiple raster images, but in my implementation I support only solid color and linear gradients. For antialiasing, instead of finding the closest feature, I look for the two closest features. The region descriptors associated with these two features can then be used to compute the color of the associated regions, and the signed distance functions can be used to compute appropriate blend weights to combine the two resulting colors. A feature might in general also contain a pointer to a stroke descriptor, and in this case a modified blend function can be used to outline the stroke.

This will not be completely accurate, since only two regions will be blended at once, and in some cases more than two regions may meet at a point. However, it is sufficiently accurate to give good image quality.

It would be more efficient to combine the two "half-features" on either side of a boundary into a single feature with two region descriptors, in other words, to use a full winged-edge data structure. I do not do this due to limitations in my flattening implementation, which did not itself generate this data, but this optimization has the potential to cut storage nearly in half and double performance.

5.5 Interval Voronoi Analysis

For this stage of my research I develop a new accelerator which, like my previous work, is also based on Voronoi analysis. First, I overlay a uniform grid onto the image. The grid splits the image into many small cells. Then I generate a Voronoi diagram for the image. In each small cell, I list all the features that are the closest feature to any points in this cell. To find the closest feature for a point, I only need to compare the distance with the features included in the cell containing this point.

I decide to compute the Voronoi diagram using GPU acceleration, but run into a number of issues due to sampling. In this section, I will present a new method for performing the required analysis robustly using interval analysis [54, 12].

My previous work uses a GPU-generated sampled Voronoi diagram to find the closest feature for each sampling point. Unfortunately, I find this method has some shortcomings when applied to general SVG images. Specifically, if I point sample the Voronoi diagram, errors can occur at sharp corners because of the limited sampling resolution as shown in Figure 5.5. In (b), the four black dots represent the Voronoi samples. The thin feature of region C will be omitted. For the points within this cell, the distance computation will not access the thin features as it should.

This problem does not appear in my previous work on vector glyphs using raster Voronoi accelerators because glyphs generally have a specific feature scale, and an appropriate sampling resolution for the Voronoi diagram could always be found. However, in general SVG images, this situation can occur quite often, since I often will convert narrow strokes to closely spaced parallel paths. One naïve way to try and solve this problem is to increase the sampling resolution until I can guarantee I will not miss the smallest region in the image. Unfortunately, since features in vector images can be arbitrarily small, in pathological cases I may end up using a very large number of sampling points.

Alternatively, I could compute the Voronoi diagram geometrically, resulting in a subdivision of the plane that can be intersected with the area of each cell. While theoretically possible, implementing this algorithm robustly is challenging, given that I am dealing with Voronoi regions generated by line segments and arcs, and not simple points, and boundaries between such Voronoi regions are in general parabolic, not linear. To my knowledge, although general theoretical algorithms are known, there is no off-the-shelf software that can deal with this more general situation geometrically in a fully robust fashion.

I therefore develop a new approach to this problem based on interval analysis. My method is robust and relatively easy to implement. It does not require geometric computation or construction of an explicit planar subdivision, but generates a conservative list of the features that are potentially nearest to some point in each cell. The basic idea is that I compute, over each cell $C_{i,j}$, the upper bound $U_{i,j,k}$ and lower bound $L_{i,j,k}$ on the distance to each feature F_k in the image:

$$\begin{aligned} L_{i,j,k} &= \min_{\mathbf{x}\in C_{i,j}} \min_{\mathbf{y}\in F_k} d(\mathbf{x},\mathbf{y}), \\ U_{i,j,k} &= \max_{\mathbf{x}\in C_{i,j}} \min_{\mathbf{y}\in F_k} d(\mathbf{x},\mathbf{y}) \end{aligned}$$

Note that the upper bound is still the maximum of the minimum distances from some point in the cell to the feature, and I am interested in unsigned distances here.

For lines, I compute the minimum distances from each corner of the cell to the line segment and from each edge of the cell to each endpoint of the line segment. If the line segment intersects the cell, the lower bound on the distance is zero, otherwise it is the minimum of all the other distances. The maximum distance is the maximum of the distances from the corner points of the cell to the line segment.

For arcs, the computation is more involved, since arcs do not divide space into strictly convex regions like line segments do. Consider Figure 5.6. Lines from the endpoints of

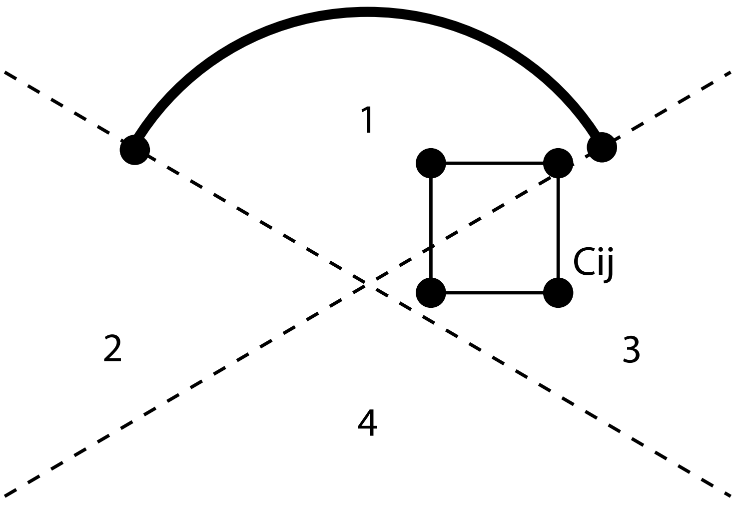


Figure 5.6: *An arc subdivides space into four regions, and a case analysis can be used to compute bounds on the distance to all points in a query cell.*

the arc to the origin of the circular arc divide space around the arc into four regions. The distance computation is different in each region. In Region 1, the distance can be computed as the distance to the center of the circle containing the arc, less the radius. An absolute value of this difference gives the distance to the arc from either above or below the arc. In Region 2 and 3, the distance to the arc is simply the distance to a single endpoint. In Region 4, the distance is the minimum of the distances to either endpoint. For each region, I can do an interval extension of these computations. Regions 2, 3, and 4 are straightforward, since they just involve distances to a point. Region 1 is more involved. I compute the distance from both the edge of the cell and the endpoints to the arc, and take the minimum of these; I also have to account for arc that enters the cell, when the minimum distance is zero. Of course a cell can straddle multiple regions. I process a cell edge by edge; each edge can be clipped into up to three subedges, for which the appropriate rule for the region it falls into is applied.

Once I have the upper and lower bounds for each cell and for each feature, I can then compute the minimum of the upper bounds in each cell M_{ij}:

$$M_{i,j} \quad = \quad \min_k U_{i,j,k}.$$

I can then make the observation that any feature whose lower bound is greater than $M_{i,j}$ will never be the closest feature to any point in cell $C_{i,j}$ and I can discard it from further consideration. I also note that for closed paths, I can generate a region for each feature based on its normals (by construction my paths will have continuous tangents) outside of which the feature cannot be the closest feature. With these two rules, I can generate for each cell a conservative list of features to consider for packing. I am still

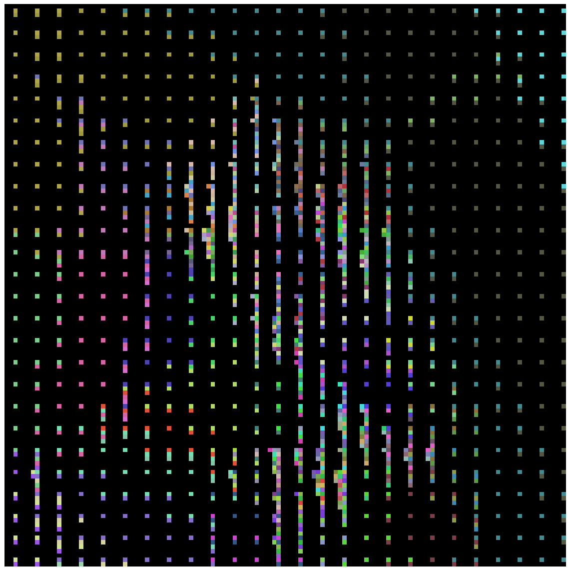


Figure 5.7: *Interval Voronoi analysis is used to generate lists of features to be considered from each cell in a grid. The density of the blocks in each cell represent the number of features in that cell. In this figure, I can see the computation complexity in each cell*

going to use the approach taken in the previous stage of my work: simulated annealing packing to eliminate redundant feature references and local search during rendering.

An example that visualizes the result of this analysis is shown in Figure 5.7, using the same cow sign example shown earlier, where the lists of features in each cell are represented by multiple colors in each of a grid of "superpixels". Note that in the boundaries between Voronoi regions, features from both adjoining Voronoi regions are reported.

My prototype use a brute force Voronoi analysis, simply analysing all features for every cell. However, it would be interesting and straightforward to use a top-down recursive subdivision approach to generate an adaptive resolution Voronoi analysis.

5.6 Results

Sample results are presented in Figure 5.8 and Figure 5.9. These samples are computed on an NVIDIA 7800GT with a 2.4GHz Pentium 4.

The circular arc approximation provides smoother results than line segments. I do not observe any obvious crease artifacts in my examples even when the images are greatly magnified. The preprocessing required for arc spline approximation is rapid, taking less than a second for the examples shown. These results show that the basic features of vector graphics images can be supported with extensions to previous work.

The rendering speed is not as good as I expected, on the order of 30fps to 60fps for these 512×512 examples. Although the distance-to-arc computation is fast, the uniform

grid accelerator structure affects the performance. In the uniform grid, I distribute features that influence a cell among the current cell grid and its neighbors. In general SVG files, often there are many small features in a local area. Either I have to increase the search area among the neighbors, or I have to refine the grid, and increase the texture size. In the first case, the performance is slowed down since more computations are involved for each query point. In the second case, sometimes the grid has to be split so many times to guarantee a certain amount of computation, that I might as well just use a raster image with high resolution. A better solution is to use control flow so that areas with fewer features can stop computation early. Control flow also allows us to allocate different storage for areas with different levels of details. Areas with fewer details use small storage. With the new generation of GPUs, it is possible to achieve this goal. Instead, I develop a completely new structure next chapter.

Another main reason for the relatively low performance is the way I find features using interval analysis. This is a conservative method and often many more features than necessary are included in a cell. This results not only in more computation, but also causes the texture size to increase. Simulated annealing often fails when the grid size is small because there are so many features that need to be stored for each cell. However, doing Voronoi analysis analytically and accurately for curves, even when limited to arcs and line segments, is a hard problem. To support the full generality of SVG files, including exact parametric cubic curves, I decide that I have to abandon the Voronoi diagram method and develop other approaches.

The third reason influencing the performance is that I represent each boundary twice. Therefore the complexity of the image is doubled at least. This is another shortcoming I overcome in my next work.

In my shader, I have two types of distance computation: distance to line segment and distance to circular arc. I can either use control flow to separate the two cases, or approximate line segments with arcs so that only one type of computation is used.

In my experiments, I tried both methods on some SVG files. These files are all composed of line segments and curves, which are therefore approximated by arcs. As expected, the control flow does adversely affect the performance. Using only one type of computation (so that line segments were a special case of arcs) the performance was increased by 4% to 22%.

However, in my representation a line segment has height 0. I cannot use the equations for the arc directly for line segments, because equation 5.4 will have a division by 0. To solve this problem, I set the height for line segment to a small number. It is equivalent to approximating the line segment using an arc with huge radius. In Figure 5.3, the ratio between the height and the length of the chord is set to 0.001. The approximation is good, and no bending in the line segments were observed. However, I still recommend using separate cases, because it is sometimes hard to choose an appropriate height value that is appropriate in all situations.

To test how much slower the arc spline computations are compared with line segments distance computation, I apply both the arc spline equations and the line segment equations to test cases with matching numbers of features. In my rendering system, the

speed of arc spline computation is $70\% - 78\%$ of the speed of the line segment computation. Remember here the arc is represented with five parameters, while line segments are four, so this ratio is in line with the additional bandwidth requirements. The distance computation involves finding the origin of the arc. If I use six parameters with the origin given, the distance computation will be even faster, but if the system is memory bound, the whole performance will be slower.

5.7 Summary

This system has much room for improvement. As mentioned before, the flattening doubles the number of boundaries. The interval Voronoi analysis is too conservative. The packing system is inefficient when the contents of the image are not uniformly distributed. These factors all affect the performance. While the system is real-time, the performance is not high enough to allow inclusion of this technique as part of a larger system.

I also encounte some practical problems in my implementation. The flattening functions in the existing software are unfortunately not accurate. After flattening, a boundary is represented twice, once in each path. Ideally, the coordinates for the two boundaries should be the same. However, the results returned are different because of numerical approximation in existing software. They could either overlap a little bit or leave a gap between the two regions. These small discrepancies cause artifacts in my examples which are not acceptable. My solution is to snap the coordinates that are close enough to a common coordinate. However, his strategy not only is inelegant, but also does not always work well, especially when the shared boundary is part of a longer boundary in one path, and the longer boundary is represented as a whole in that path, and not in the other. The snapping will generate gaps between the boundaries.

These problems inspired my the work in the next chapter, in which I take advantage of the features in the new 8800 generation of GPUs. Basically, control flow is used to provide more efficient distance computation and texture size. The flattening problem is avoided by using another strategy to solve the multiple layer occlusion problem.

Also, I decide to attempt to compute the exact distance to the feature curves in SVG files rather than trying to deal with approximations. Therefore, I abandon the arc spline approximation used in this chapter. However, approximating curves with circular arcs for fast distance computation is not only useful in vector graphics, but also benefits other applications, such as milling and machining, as they often have to solve the offset problem as in my application. Arc splines are interesting because their offsets are also arc splines.

Figure 5.8: *SVG examples with boundaries approximated by arc splines.*

Figure 5.9: *SVG examples with boundaries approximated by arc splines (continued).*

Chapter 6

Layered Regions and Arbitrary Parametric Features

As there are several weaknesses in my previous work, for the last stage of my research I develop a more efficient and general system exploring the power of the new generation of GPUs, as typified by the NVIDIA G80 architecture. To do this I revisit several design decisions that are influenced by the limitations of past GPUs but which are no longer as serious a concern.

6.1 Introduction

The G80 has much better support for control flow than previous GPUs. Thus control flow can be used to allow each cell to have a different number of features. Forcing the cells to have the same number of features not only increases the computation, but also increases the storage size. For cells that have fewer features, control flow ends the computation early, and only the minimum required storage space is needed to store the features. Although control flow affects performance in general, in real cases the performance improvement obtained by reducing computation and lowering texture lookup bandwidth is a win over the performance degradation caused by control flow.

Second, in the context of the newer GPUs and when using control flow the feature information in each cell can be packed together contiguously, with control flow used to traverse the section associated with each cell. This strategy greatly reduces the texture size.

Third, for multiple layer overlapping, I decide to avoid using flattening and Voronoi diagrams, but attempted to solve the occlusion problem with another approach. In this approach only the necessary features are included in the computation, and boundaries are not represented twice as in previous work. With this approach the original complexity is maintained, and the computation for each cell is reduced to the minimum.

Most importantly, in both stages of my previous work, the boundary curves are not rendered exactly. They are approximated by either line segments or circular arcs. These

approximations reduce the degree of smoothness in the curves. To this point there have been little research on rendering higher order curves, especially cubic splines, directly with exact distance computation. This is because the distance computations to these curves are complicated and the available numerical methods are not suitable for GPUs as is discussed in Chapter 3.

In this stage of the research I develop a simple, robust, and practical way to compute the accurate distance to any curved feature from a sample point within its radius of curvature. The biggest difference between my work and other research is that I always compute the accurate distances to the curves to support special effects. Other research uses line segments to approximate curves or uses quadratic splines to approximate cubic splines. For example the distance to quadratic curves are approximated as in Nehab and Hoppe's work [40] (while the quadratics are themselves used to approximate cubics).

Other researchers have made valuable contributions to the problem of fast rendering. However, they have achieved these results by using various approximations for both distance computations and boundary representations. Previous researchers have also imposed many restrictions or accpet certain limitations, such as rounded corners and poor antialiasing.

Given the previous work that involves many approximations and limitations, I decide to focus on rendering the vector graphics exactly as specified with no rounded corners, with higher order curves, especially cubic splines, rendered exactly as they are drawn, and with high quality antialiasing.

I also want special effects depending on accurate distances to be easily added, and to antialias these special effects properly as well. In particular, I develop a technique for special effects and second-order antialiasing of the ridge lines that can appear when using this special effect.

6.2 Algorithm

My approach has two parts: a preprocessed representation that embeds the vector image in a set of texture maps, and an algorithm implements in a shader for interpreting this information to compute the color of the vector image at any point. The representation must support fast access to the minimum amount of data at every point. The basis of the color computation is a fast distance computation.

To build my representation, a uniform grid is first superimposed on the vector image. Each cell is checked against the boundary features to find the subset of features that intersect it. All boundary features that intersect a cell are stored in a per-cell list. This list is then augmented by features within a certain distance of the cell to support antialiasing; the extension distance is bounded by using MIP-maps in the far field. The far field is defined as the point where the narrowest vector feature spans less than one pixel. At this point the image should be rasterized and filtered, so that undersampling artifacts can be avoided.

Color retrieval and boundary antialiasing for any point in a cell need only consider the cell's feature list. As mentioned previously, the distance to the closest boundary feature will be used for antialiasing, and the gradient of this function is used for anisotropic antialiasing.

I will talk about how to build the representation in Section 6.2.1 and then how to compute the distance and use it for antialiasing in Section 6.2.2.

6.2.1 Preprocessing

A typical SVG image is composed of closed paths or strokes. Paths can have borders of various widths and colors, and can be oriented in either a clockwise or counter-clockwise direction. The region enclosed by a path can be filled with a solid color or a gradient. Strokes may also have colors and widths. For now I will only deal with filled paths without border colors, but will discuss generalizations in section 6.3. I also assume the images only contain closed simple paths oriented in a clockwise fashion, and that these paths do not have any self-intersections. If a path intersects itself, it should be separated into multiple simple paths [14].

To build my representation, I first split curves into monotonic segments, which will simplify the distance computation later. I then index these feature segments with an accelerator structure that allows fast random access lookup of the features needed during rendering.

Splitting Curves

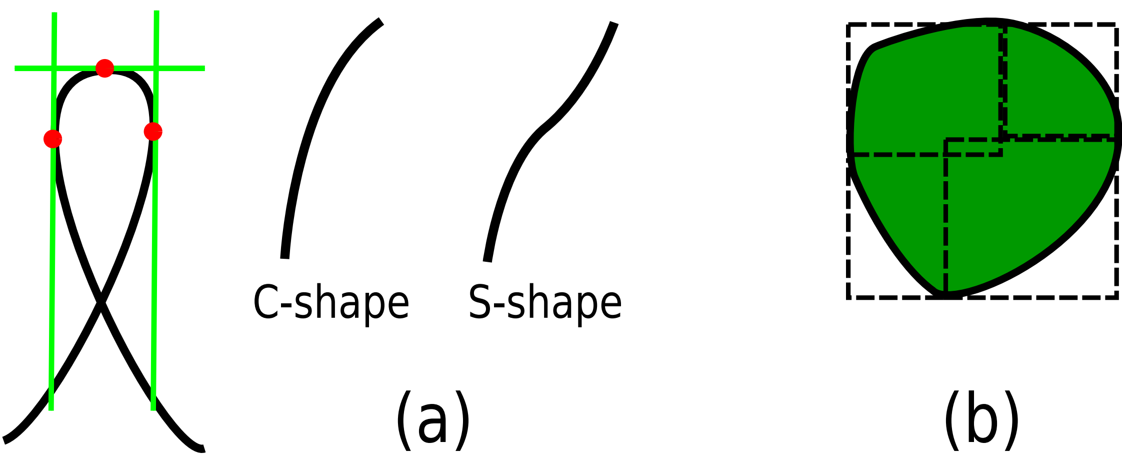


Figure 6.1: *(a) All curves are split into C and S shaped features at every point where the tangents are horizontal or vertical. (b) The bounding box of a path is the bounding box of its features.*

In order to simplify the distance computation, I split each curve at the points where its tangents are horizontal or vertical. For example, in Figure 6.1(a), the given spline is

split into four segments at the red dots. Let the parametric form of the curve be given by $P(u) = [x(u), y(u)]$. For cubic splines, the splitting points u^* can be found by solving for the roots of the quadratic parametric splines giving the components of the tangent $\vec{T}(u) = [x'(u), y'(u)]$ of $P(u)$:

$$x'(u^*) = 0 \qquad \text{or} \qquad y'(u^*) = 0.$$

There are at most four split points, as there are at most two solutions for each of the above equations, so curves are broken into at most five segments.

Each resulting segment has a simple shape. In fact, after splitting there are only two categories of shape, as shown in figure 6.1(a). I call these categories C-shapes and S-shapes. Note that over each segment, $x'(u)$ and $y'(u)$ are either completely positive or negative. I call such functions *monotonic*. The C-shape represents all curves with no inflection point. The S-shape represents all curves with one inflection point. Since the second derivative of a cubic is a linear function, there can be at most two such points: one for $x''(u*) = 0$ and one for $y''(u*) = 0$. I could further split the S-shape into C-shapes at these roots. However, since S-shape curves still work with my distance algorithm, I keep them as they are.

Finding the splitting points for (rotated) elliptical arcs and quadratics is similar. Line segments do not have to be split. After splitting, spline curves and elliptical arcs are still spline curves and elliptical arcs. The overall continuity and shape of the boundaries does not change.

Detection of Features Overlapping a Cell

In order to build my accelerator structure, a uniform grid of cells is superimposed on the vector image. For each cell I would like to find which paths completely contain the cell and which paths intersect it.

For every cell this process is done path by path in drawing order. Rendering order is important in vector images and as part of the preprocessing I also need to resolve visibility (and exploit it when possible to remove hidden paths).

To accelerate this procedure, the axis-aligned bounding box of each path is computed. If a cell does not overlap the bounding box of a path, the path cannot cover the cell or cross the cell boundary at all. Recall that all curves are split into monotonic segments. Because all my split features are monotonic, each one's bounding box can be constructed from its two end points. The bounding box for an entire path can be computed from the bounding boxes of its features, as shown in Figure 6.1(b).

If a path's bounding box overlaps a cell, all features of the path are checked against the cell. My monotonic features can be intersected with the four boundary lines of the cell using classical numerical methods. If a feature intersects any boundary or if it is completely contained by the cell, it is stored in the cell's list.

If none of the features in a path overlaps a cell, two situations are possible. The path can be outside the cell or it can completely contain it. In the latter case, all features that

are already listed for the cell are deleted, and the background color of the cell is set to the fill color of the current path. All previous features will be covered by the new path.

I use the winding rule to test if a cell is contained by a path [40]. Any point in the cell can be chosen for the test; I choose the lower left corner of the cell.

Cell Extension

When the path is checked against a cell, the cell extent needs to be expanded to tolerate some level of texture minification. The antialiasing filter will have a certain width in texture space, and I need to be able to compute distances to at least this width. The magnitude of the expansion depends on the level of image minification to be supported during vector rendering.

At higher minification levels, texture rendering should be implemented with a raster-based MIP-map. This raster image can be rendered (on the GPU) directly from the vector texture. Under minification MIP-mapping gives an accurate antialiased representation of the vector image, and transitions can be managed using the alpha channel of the raster texture. The finest resolution raster image should be stored with $\alpha = 0$, the other coarser levels should use $\alpha = 1$. Then trilinear interpolation of α will give the correct value for blending between raster and vector rendering. Vector rendering can be disabled using control flow in the shader when $\alpha = 1$. If the texture includes transparency, then the shader can compute the transition function explicitly using the Jacobian of the mapping from texture coordinate to screen space.

However, extending the cell by a fixed amount may not include all features needed to compute the inside or outside test correctly. The extension width should be chosen so that if a feature crossing the cell is included, all features that might be closer for any point in the cell should also be included.

Acceleration Structure

At this point, I have a list of relevant features and the background color for each cell. This information needs to be stored in an acceleration structure. I use three textures to store this information in an efficient way.

At the lowest level I list all features from all paths, including points, line segments, quadratic splines, cubic splines, ellipses, and so on in one texture. Each feature is stored only once. The features are grouped and sorted according to paths. Features belonging to the same path are put together with a header that stores per-path information, such as fill color, border color, border width, etc. This texture is shown in Figure 6.2(c).

Next, for each cell, the list of features that crosses this cell are sorted according to the paths they belong to. For example, suppose there are several features from two paths that overlap the cell. Some belong to the first path, and some belong to the second path. Then the features belonging to the first path are grouped together, so are the features belonging to the second path. Just before the list of features in each path, an extra storage slot is

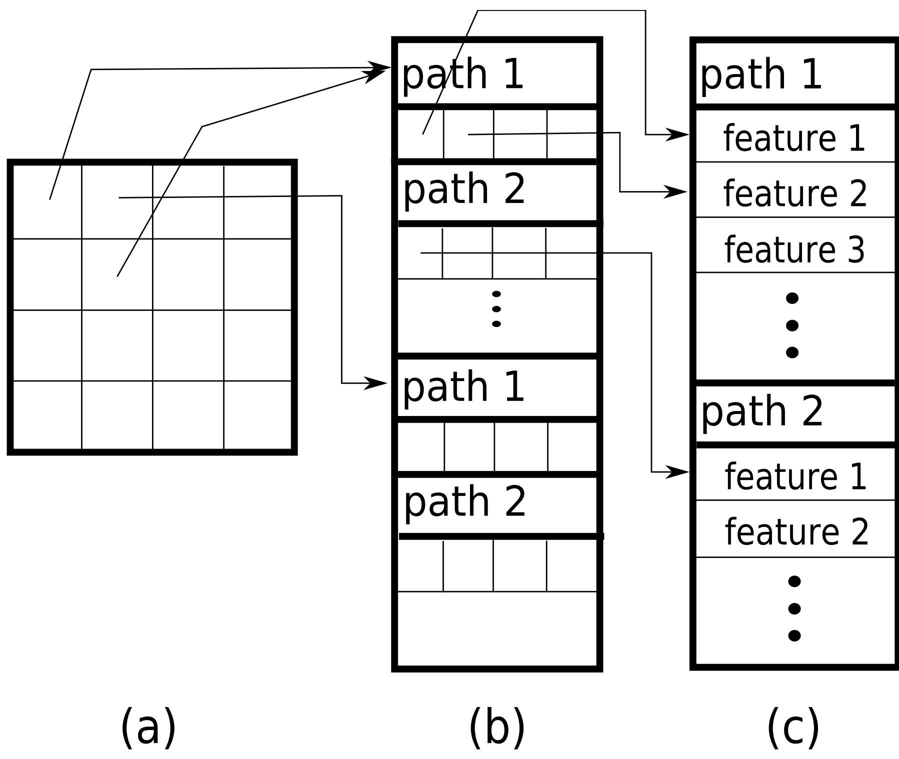


Figure 6.2: *Three textures are used to store the accelerator structure.*

used to record the number of features in the path. This texture stores pointers into the low-level texture that actually contains the features. Thus only one storage slot is needed for each feature. This stage is shown in Figure 6.2(b).

The final texture is a grid texture corresponding to the cells. For each cell it holds the start location of the list of features in the second texture. The corresponding list of features are exactly those that overlap the cell. It also stores the number of paths that go through this cell, and the background color for the cell. If two cells have identical feature lists, they share a single list in the list texture described above. Figure 6.2(a) gives an example, including two cells that share a feature list.

6.2.2 Shader Computation

Path Compositing

The order of the paths and feature lists in every cell needs to be consistent with the input order in the SVG file. I start with the background color and the list of paths and features for the cell. The signed distance to the closest feature in every path is computed in sequence. For every path, the sign is used to find if the test point is inside or outside that path. A point is inside a path if it is on the right side of the closest feature and in this case I use a negative distance. The closest distance to each new path may have the same or a different sign from the previous one, and may also be smaller or larger in magnitude.

I always keep track of the distance d with the smallest magnitude, a foreground color F, and background color B. These two colors are always the colors on each side of the closest boundary. Suppose D is the closest signed distance to the next path in the sequence and C is its fill color. I update d, F, and B as follows:

`bool` $q := D < 0 \quad \text{or} \quad |D| < |d|;$
`bool` $p := q \quad \text{and} \quad d < 0;$
$B := p \quad ? \quad F : B;$
$F := q \quad ? \quad C : F;$
$d := q \quad ? \quad D : d;$

At the end of the process, the distance value with the smallest magnitude and its sign are used to find the antialiasing blend weights using a smooth step, with the transition width found using the Jacobian of the texture coordinate transformation as in Chapter 4.

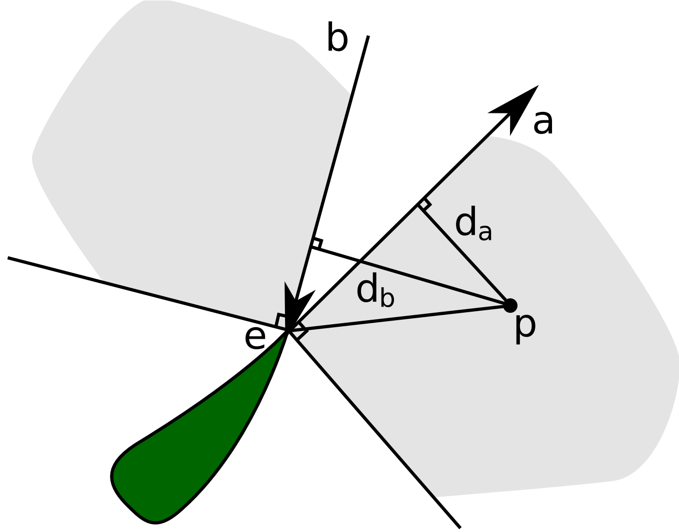


Figure 6.3: Inside or outside test errors at the corners.

There are different ways to blend colors along the boundaries. I only use two colors which is not accurate in some cases. There will be artifacts when one path overlaps another exactly. Suppose there is a third path under both of these paths, its color is the background color for the two paths. The correct blending along the boundaries should take place only between the color at the top path and the background color, since the path underneath the top one is completely occluded. According to my formula, however, the two colors come from these two overlapping paths. The occluded path is causing color bleeding. I choose this blending method anyway for better performance as this case happens rarely.

Another solution is to blend the color each time after a path is processed. Starting from the bottom layer, when the second layer is processed, their colors are blended along the boundary. This blended color can be regarded as the background color for further

layers. When the third layer is processed, its color is blended with the background color along the boundary. This process continues until all layers are processed. However, as in the case mentioned above, this approach also causes color bleeding from the occluded path. Also the color contribution of the first layer gets smaller and smaller as layers are processed.

An accurate antialiasing approach really should analyse the boundary configuration, and use percentage coverage for antialiasing. Winner et al. [68] computed accurate antialiasing by processing the layers from front to back. At each layer a mask is updated according to the boundary configuration between this layer and previous layers to reflect the percentage coverage of the path in each pixel. However, such analysis is too expensive for real-time rendering.

When a signed distance is used to detect if a point is inside or outside a path, an ambiguity is possible at sharp corners. This problem has been described in my first work in Chapter 4. to resolve this issue, I introduce a small gap in the contour. Here I use another solution. Before talking about the solution, I would like to briefly review the problem. In Figure 6.3, a closed path is represented by the green shape. Lines be and ea are tangent at the corner e. The directions of these lines are consistent with the direction of the path contour. For any point that falls inside the gray shaded area, its closest distance to the path boundary is the distance to the corner. In this figure, the closest distance of point p to the path is pe. As the two closest distances are the same, the simple minimum distance rule randomly choses either line be or line ea for the inside or outside test. The sign is computed by projecting pe onto a 90 degree rotation of the corresponding tangent. However, only tangent be gives the correct sign. To solve the ambiguity, I compute the distance from p to the extended tangent lines eb and ea and choose the feature that has the *larger* absolute value to these lines in case of a tie on the minimum distance.

Distance Computation

The core of my algorithm is the computation of the signed distance to a feature. The distance computation to a line segment is simple, and has been described in my first work in Chapter 4. However, the distance computation to a quadratic or cubic spline or an elliptical arc is not as straightforward, as shown in Chapter 3. We present an approach to distance computation based on binary search and can be applied to any kind of parametric curve. This approach is fast and simple, but only accurate when the query point is within the curve's radius of curvature. However, the correct sign is always achieved and the result is sufficiently accurate that images with high visual quality can be achieved. There are a few cases where the lack of accuracy can cause problems; after describing the algorithm I will discuss these in detail.

I first bound the parametric location of the closest point on the curve using an interval that contains the entire curve segment. At each iteration, the interval is split at its parametric midpoint. I then extend the normal at the splitting point into a splitting plane, as shown in Figure 6.4. The point is tested against this plane. The half curve that is on

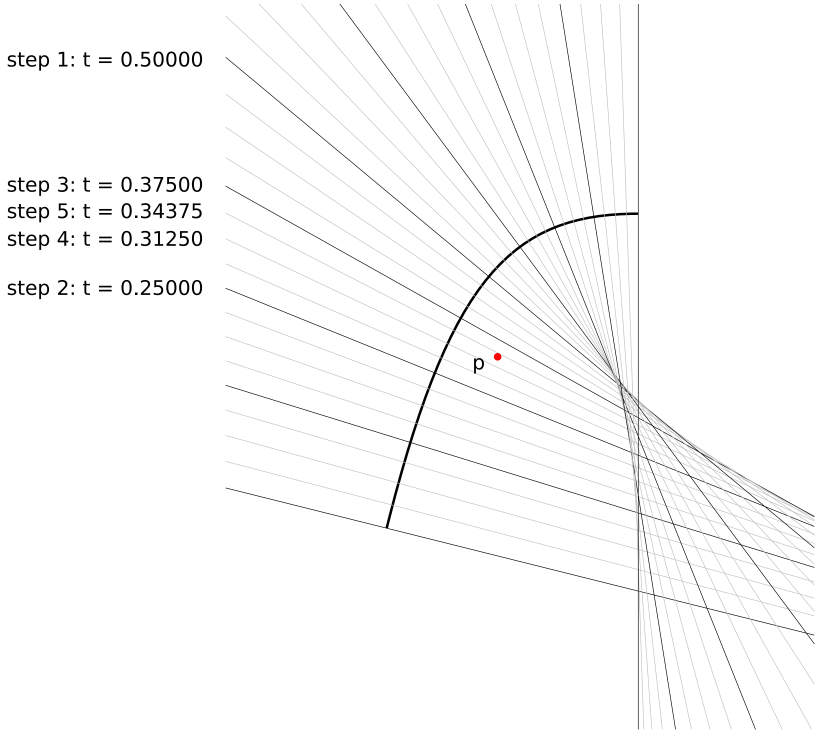


Figure 6.4: The closest point on a curve to an arbitrary point p can be found using a binary search based on normal line tests. In regions where normal lines cross, the binary search algorithm may fail, but normal lines only intersect each other beyond the smallest radius of curvature.

the same side of the plane as the point is chosen for the next iteration. Iterations can be repeated until a certain error tolerance is reached or for a fixed number of iterations.

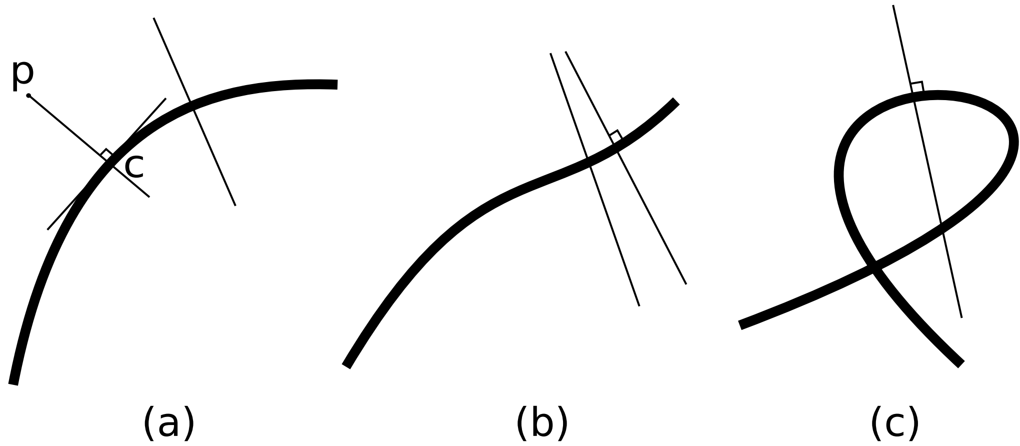
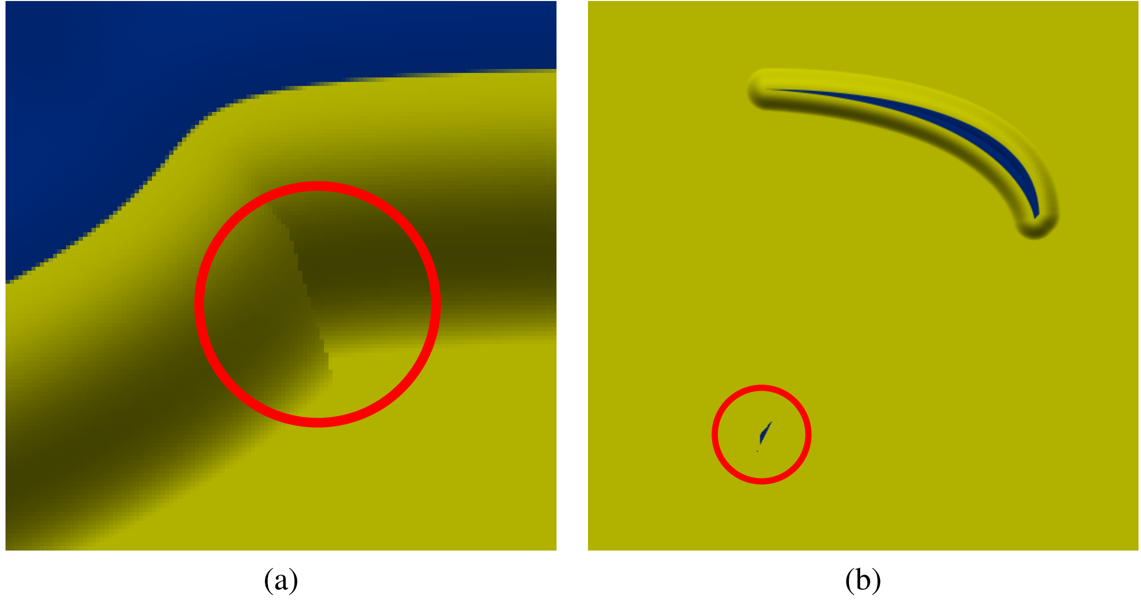


Figure 6.5: The normal lines for C-shaped and S-shaped curves do not intersect the curve itself again.

To understand the properties of this algorithm, first observe for C-shapes and S-shapes the normal lines always intersect the curve exactly once; see Figure 6.5(a),(b). Case (c) does not occur since I have split the curves. Second, observe that if I draw two normal lines at any two points on the curve, the normal lines will never intersect on the convex side of the curve. However, they always intersect on the concave side, but at a distance that is always greater than the smallest radius of curvature of the curve. *Therefore, for all points on the convex side and for points within the radius of curvature on the concave side, the normal lines impose a sequential ordering on the space around the curve.* My binary search locates the query point's position in this order. However, beyond the radius of curvature on the concave side the normal lines fold over and the ordering is lost, so the algorithm may fail. Failures return another point on the curve and a distance that may be larger than the minimum, but never smaller.

In all the examples used in this paper, despite the possibility for failure, no obvious visual errors are observed. This is because for antialiasing, the query points where the distance matters are close to the boundary features, well within the radius of curvature in all features in my sample images.



(a) (b)

Figure 6.6: (a) Distance computation errors occur for points whose distance to the curve is larger than the curvature radius. (b) In the worst case this can lead to errors in inside or outside classification.

In embossing, errors will only be visible if the embossing width is relatively large. To demonstrate the visual appearance of these errors, I increase the embossing width in an example until an error is visible. This is shown in Figure 6.6(a). Because some points do not compute the correct shortest distance, there are some discontinuities along the embossed edge.

In an implementation, the iteration can be binary, or I can evaluate four points (or more) at one time using the parallelism of GPUs to increase the performance.

I considered using other iterative methods to improve the convergence of the distance computation. The Newton-Raphson algorithm converges quadratically, but it is not *guaranteed* to converge, and can fail catastrophically. The Regula-Falsi or the Secant algorithm are guaranteed to converge given a monotonic region. However, it is hard to split general curves into monotonic regions for the distance function for all points in a cell. Before I start to explain the reason, I define a computation ”static” if it can be done in the preprocessing stage in CPU, and I define a computation ”dynamic”, if it can only be done in the shader in GPU at runtime.

For a given point, to compute its closest distance to a cubic spline, for example, I need to find the roots of a quintic equation. Suppose the cubic spline is fixed, but the points are different, the quintic equations for different points are different even though the cubic spline is the same one. If I use Regula-Falsi algorithm or Secant algorithm, I need to partition the quintic equations into monotonic regions. This partition step cannot be done in the preprocessing stage in CPU, because different points generate different quintic equations. In the preprocessing stage stage, I do not know how many points and which points are going to be rendered, because with different zoom scales, the point coordinates that are to be rendered will change. So I have to partition the quintic equation into monotonic regions in the shader in GPU at runtime when I compute the distance for a point. This is time consuming.

Note that any curve is guaranteed to be non-monotonic for some point. For a general non-linear curve I can always find two points on the curve with normal lines that intersect at some point. This intersection point has two local extremal distances, so at least for this intersection point, the distance function is not monotonic.

My solution can partition the spline curves into monotonic regions in the preprocessing stage in CPU. Once the monotonic region partition is done, it fits any points within the radius of curvature. So I do not need to do the partition in the shader of GPU at runtime, therefore it is fast.

My solution always returns some point with a correct distance sign, although the distance magnitude may be too large and may also return the wrong point on the curve as the “closest point”.

The sign is important in the inside or outside test. For points that are far away from the boundaries, the accuracy of the distances is not important because they are not involved in antialiasing or embossing. However, the distance sign is required in the inside or outside test. This issure also exists in other similar iterative algorithms such as the Bisection and Secant methods. Basically, these approaches would also have the same problems when the points are beyond the radius of curvature. Furthermore, the evaluation of binary search is simpler, since it only involves averaging and no division as in Regula-Falsi (or Newton-Raphson).

When the distance of a point to the curve is larger than the embossing width, it does not matter if the distance is correct or not, since it is not involved in antialiasing or embossing. However, the sign of its distance is always correct, so the inside or outside test, in general, should have no problem. However, some errors may happen at corners. In Figure 6.3, when a point p is in the gray shaded region it has equal distances to both

curve features that form the corner. Recall that I use the distance to the extended tangent line to break a tie. But if the distance to corner is wrong, and the two distances are not equal when they should be, the feature with the shorter distance will *always* be chosen. This feature could be the wrong feature, thus generating an inside-outside classification error.

Fortunately, this problem is easy to fix. Observe that the errors only happen for the points that are in the gray shaded areas in Figure 6.3. The real closest distance should be the distance to the end points. The distance to the two end points of any feature are therefore *always* computed, and the shortest of these two and the value computed by binary search is chosen as the final distance.

A problem can also occur if a shape is thin, *and* two boundary features are close, *and* the test point is far away relative to the curvature radius, *and* the query point is on the concave side of the curve. In this specific circumstance, the closest distances to these two features are not guaranteed to be accurate, and if the distance to the closer feature is computed incorrectly to be larger than the distance to the farther feature, the inside or outside test may be incorrect. This error can be avoided if the grid spacing is less than the minimum curvature radius. In my examples, I use a relatively large grid spacing, and yet do not have this error, so it may happen only rarely in real images. However, to demonstrate this issue I design the test case shown in Figure 6.6(b). A small classification error is visible in the lower left of this image; I have circled it.

Ridge Antialiasing

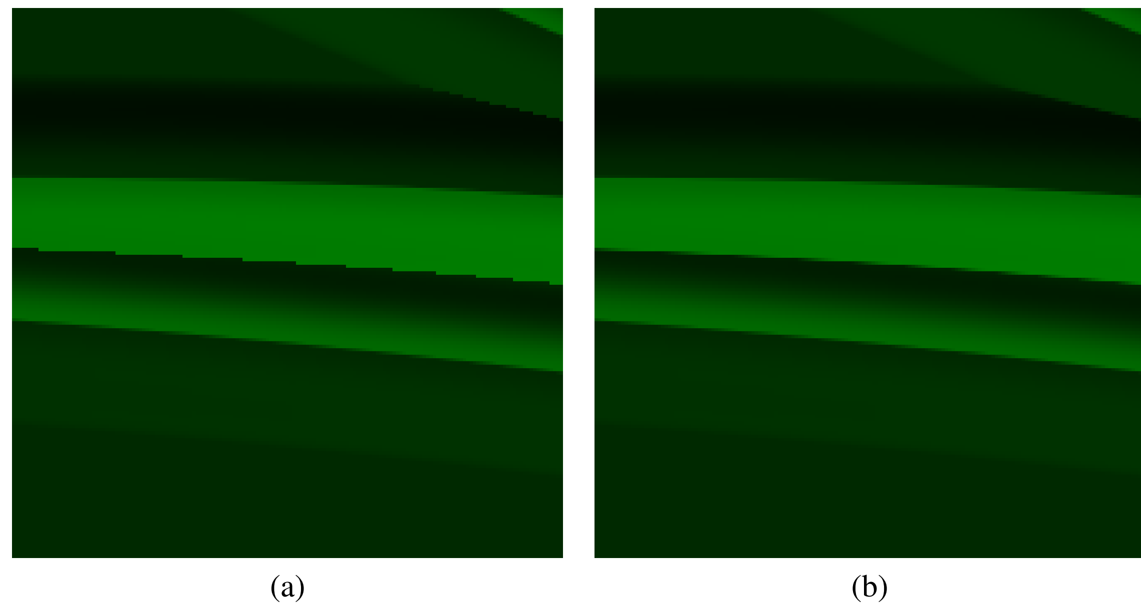


Figure 6.7: (a) Two close edges can cause aliasing along a ridge. (b) Computing the two closest features for each point, and blending the two normals will remove this alias.

If two features from the same embossed path are so close that the distance between them is smaller than the embossing width, the interaction of the two embossed regions will cause aliasing along a ridge as shown in Figure 6.7(a). This is because the closest distance to the two features have different directions, and therefore, different normals for shading. When they meet, the normals will switch direction suddenly, causing an aliasing artifact. To solve this problem, for each point, the *two* closest distances and their corresponding shading normals are computed. These two normals are then blended smoothly at the places where the two distances are nearly equal. The result is shown in Figure 6.7(b), where this form of magnification aliasing has been resolved. It is necessary to compare the similarity of the two normals. If the angle between the two normals is smaller than a threshold, the two distances may come from two segments that belong to the same original curve, but meet continuously. My monotonic splitting generates exactly this configuration frequently, but it can also occur naturally in the input data. In this case, there is no ridge in the original image, so I keep one distance and throw away the other. Embossing itself, like other forms of bump and normal mapping, can still suffer from highlight aliasing under minification. This is visible in some of my test images and needs to be addressed separately, by modifying the embossing depth and shading model with distance (for example, by clamping the Phong exponent in that lighting model when the screen-space derivative of the normal is large).

6.3 Extensions and Applications

Until now, I have discussed vector images containing only closed paths, solid fill colors, and with no borders. My algorithm can be easily extended to support other rendering styles. I describe a few of these extensions here.

To support closed paths with boundaries, extra space is needed to store boundary widths and colors in the texture. Then, for each point, after the closest distance d is obtained, it is shifted by half of the boundary width W towards the outside of the path, for instance, $d' = d - W/2$ as in Figure 6.8(a). After shifting, the fill color and the boundary color can be blended smoothly at $d' = -W$. The modified distance value and the blended color can then be used to update the closest feature and blending colors between paths.

To support strokes with specific stroke colors and widths, the closest distance also needs to be modified using $d' = (|d| < W/2)? - |d| : |d| - W/2$, as shown in Figure 6.8(b). This treats the strokes as closed paths, and the stroke color as the fill color. This modified distance and stroke color are used during layer compositing.

The modified distance is only used in color testing. For embossing, the embossing valley should be located where the original distance is zero.

I do not implement gradient fills or other non-solid fills in my test system, but they are a straightforward modification of the fill color computation. As before, extra space to record the gradient coefficients would be needed. Then instead of using a solid color directly for the fill, the color for any point inside a path would be computed from the gradient coefficients and used in place of the solid color for the region.

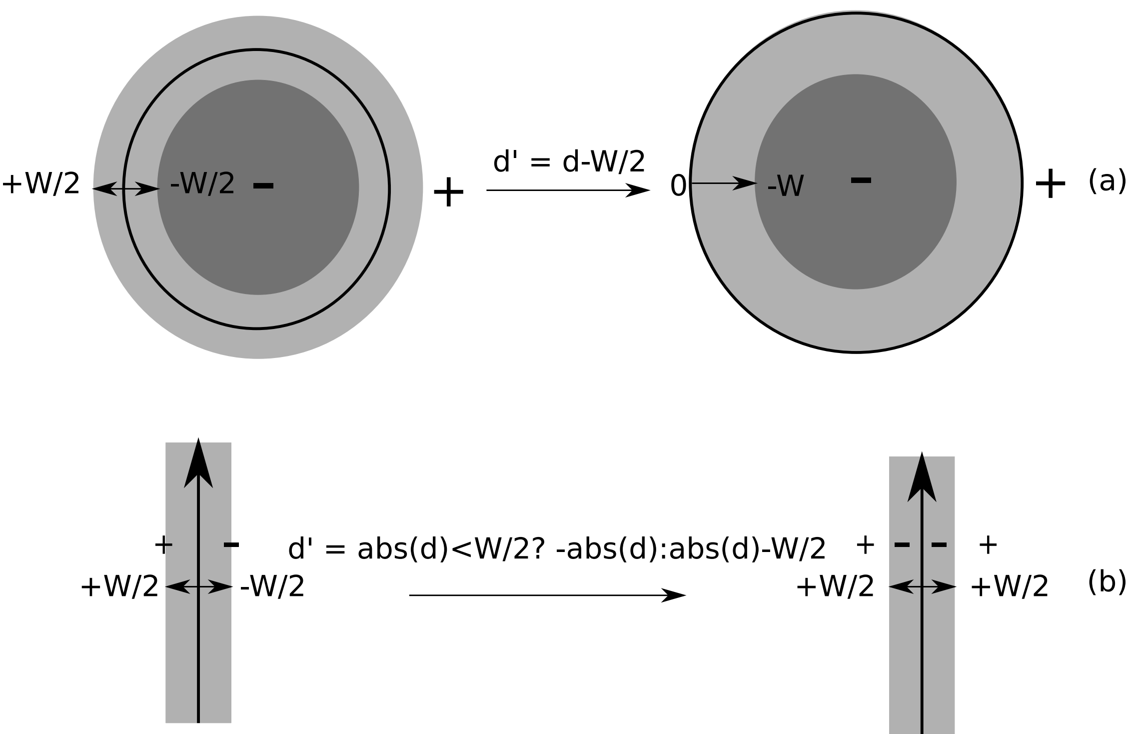


Figure 6.8: *The distances are modified to support borders(a) and strokes(b).*

Finally, some more interesting extensions are possible. The binary search computes not only the signed distance to each feature, but also the position *along* each feature that is closest to the query point. This defines a 2D coordinate system in the neighbourhood of the edge. The distance and the parametric value could be used together as texture coordinates to render textured strokes, where the texture could either be a raster texture or another vector texture. This could also be used to implement dashed and dotted lines procedurally. It would also be possible to permit the specification of programmable functions for the embossing profile, and use the distance for other purposes, such as drop shadows.

6.4 Results

Various examples rendered from standard SVG files are shown in Table 6.1, Table 6.2 and Table 6.3. Some of these examples, such as the tiger, also render thick border outlines using distance offsetting as in Section 6.3. The results of performance tests using an NVIDIA GeForce 8800 GTX are shown in Table 6.4 for a window size of 1024×1024 and the full-screen views shown in Table 6.1, Table 6.2 and Table 6.3. For each image, I test the performance with and without embossing, although anisotropic antialiasing is always enabled. I also test the performance using grid sizes of 64×64, 128×128, and

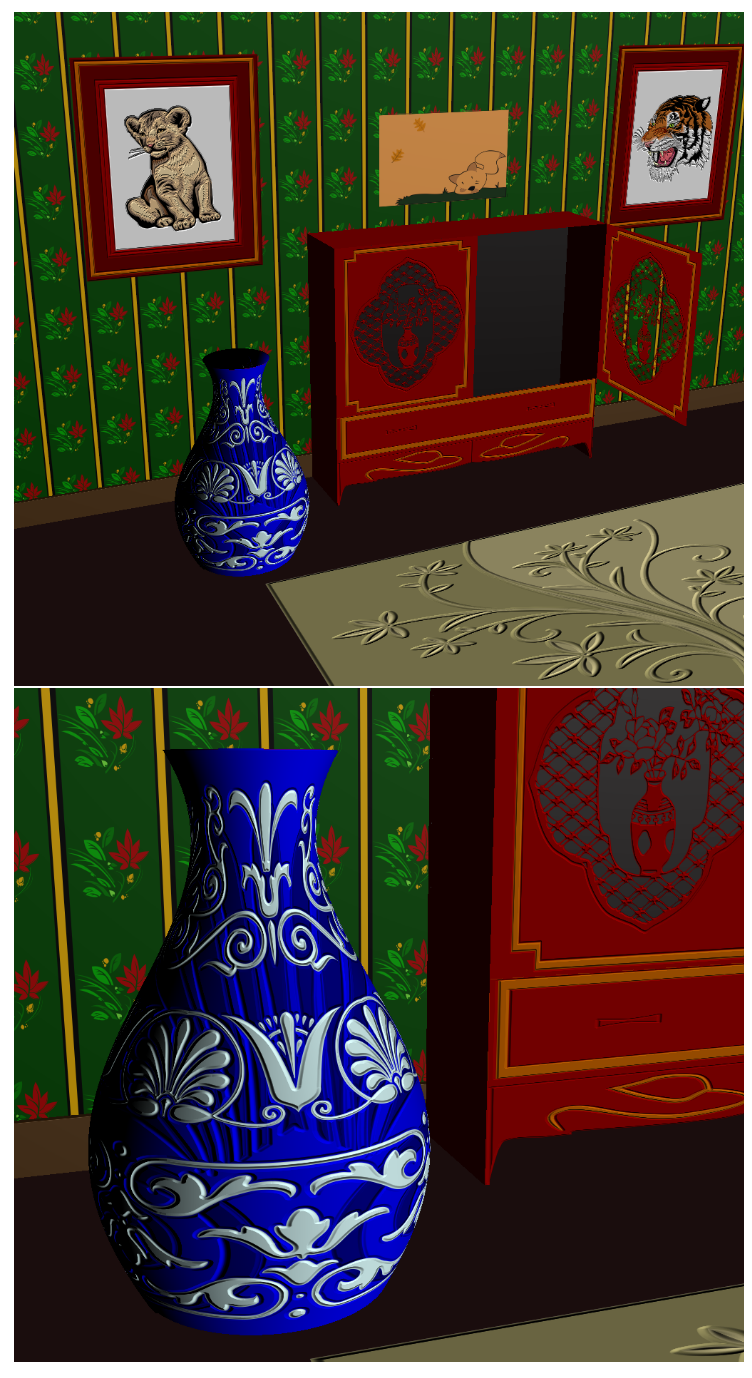


Figure 6.9: *Two scenes using several vector textures with some closeups showing embossing and transparency.*

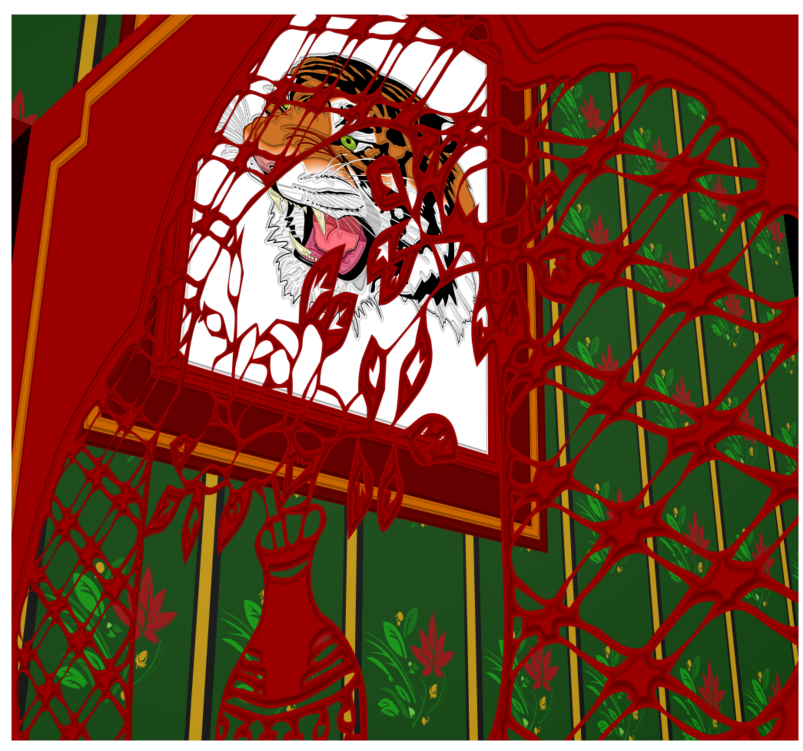
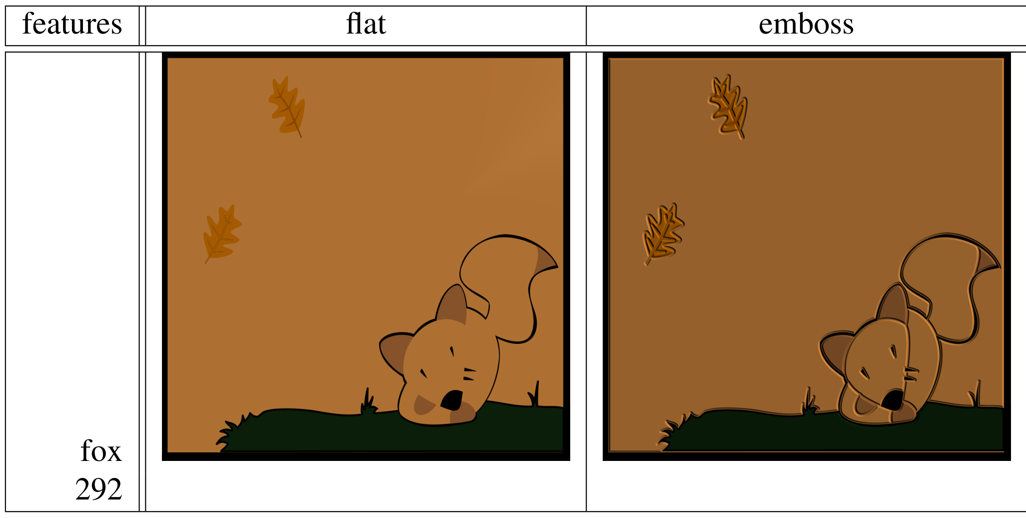


Figure 6.10: *Another scene using several vector textures with some closeups showing embossing and transparency.*

features	flat	emboss
vase 6904		
tiger 3847		
door 3147		

Table 6.1: *Screen shots of performance tests.*

Table 6.2: *Screen shots of performance tests (continued).*



features	flat	emboss
fox 292		

Table 6.3: *Screen shots of performance tests (continued).*

256×256. The storage used for each test and for every grid size is indicated in Table 6.4 as well.

My system is not highly optimized. Some more optimizations could have been done. A useful strategy to reduce bandwidth would be to quantize the boundary coordinates within cell local coordinate system, using 8 or 16 bits to represent them [40]. This can compress the storage and bandwidth to one quarter or one half. Potentially, it can increase both the performance and the storage size efficiency. This optimization is however not used in my implementation.

The lion test case has line segments only. To evaluate the performance of my cubic spline distance computation, I use distance computation to cubic spline with five iterations. The results are shown in Table 6.5. The speed is three times slower than using lines. In the tiger test case, which contains mostly cubic curves, I just apple distance computation to line segment upon the cubic splines. Although the result is incorrect, all I need is the speed when the distance computation to cubic spline is applied to these many features in my system. The speed is around three times faster. These test cases show that in general cubic spline distance computation is three times slower than line segment distance computation.

My distance computation not only iteratively computes the distance to the curve, but also computes the distances to both endpoints, and then chooses the minimum. This is required by the inside or outside test. Another way to test inside or outside is to use the winding rule. To test if a point is inside or outside a path, I can shoot a ray from the point in some direction (I always use the positive X direction), and see how many times it intersects the path boundaries. If the number of intersections is even, the test point is outside the path; otherwise it is inside the path. An implementation of the winding rule is used in Nehab and Hoppe's work [40].

I also do experiments on how the number of iterations in the binary search affects the speed. I use the tiger for a complicated example and the fox for a simple example.

	64×64			128×128			256×256		
	flat (fps)	emboss (fps)	storage (MB)	flat (fps)	emboss (fps)	storage (MB)	flat (fps)	emboss (fps)	storage (MB)
vase	18	17	0.50	24	22	0.75	29	27	1.50
tiger	20	18	0.31	29	26	0.52	37	33	1.31
door	27	25	0.27	39	35	0.45	47	42	0.97
rug	54	50	0.15	72	67	0.27	83	77	0.67
lion	83	67	0.18	114	91	0.34	127	103	0.85
wall	91	84	0.06	110	102	0.15	123	113	0.49
fox	101	95	0.05	130	128	0.15	149	149	0.50

Table 6.4: *Performance results. Window size is 1024 × 1024.*

For both images, I test with grids of 64×64, 128×128 and 256×256. Recall that the baseline test case uses five iterations; when I decrease the number of iterations to 0 (no search at all), I increase the rendering rate by 3 to 9 fps in all cases. Each additional iteration decreases frame rate by approximately 1 fps, (interestingly, this is independent of image complexity). Note that I render the same polygon repeatedly (with depth testing disabled) to reduce the overhead of shader loading. As for quality, using n iterations corresponds to approximating curves with 2^n line segments. In my test cases $n = 6$ resulted in renderings that are indistinguishable from any higher setting.

Also in the tiger test case, when all curve distance computations are replaced with line distance computations, when the grid size is 64×64, the speed is three times faster. However, with an increase of the grid size to 256×256, the speed increase dropped to 2.45. When the grid is coarse, the performance appears to be compute bound; however, when the grid is subdivided more finely, the performance starts to become memory bound. On the whole, I think the performance is still memory bound, because the shader contains two levels of dependent texture reads.

It is worthwhile comparing my results to Nehab and Hoppe's work [40]. They have demonstrated higher performance, although only with quadratic splines and without embossing. One significant difference is that their data structure uses differential encoding of the control points of paths, which results in lower bandwidth requirements. It would be useful to combine this approach to the storage of paths with my approach for computing distance, since I often appear to be memory bound.

The examples in Table 6.1, Table 6.2 and Table 6.3 are combined in the scenes in Figure 6.9 and Figure 6.10. The cabinet door is made transparent according to the opacity of the vector image. The closeup on the right in Figure 6.9 and Figure 6.10 is primarily composed of only three rectangular polygons. The foot of the cabinet is generated using transparency (the base geometry is a rectangle), and the edge is actually antialiased better than the edge of the standard geometry at the edge of the cabinet. Vector graphics offer not just unlimited resolution, but also the ability to generate complex antialiased geometry from simple 3D primitives.

	64×64		128×128		256×256	
	flat (fps)	emboss (fps)	flat (fps)	emboss (fps)	flat (fps)	emboss (fps)
lion with lines	83	67	114	91	127	103
lion with cubics	24	23	35	33	41	38

Table 6.5: *The performance difference between using distance to line segments and distance to cubics on the lion test case. The lion is modeled with line segments, so to test the latter these are replaced with equivalent degenerate cubics.*

In general, not all forms of aliasing can be addressed by my technique. In addition to aliasing at geometry and silhouette edges (see for example the edge of the vase), highlight aliasing is also present on embossed regions. If a vector region is too thin (for example, stripes on the wallpaper or rug) it can also alias. My antialiasing strategy works for color boundaries between two vector regions, but needs to be combined with other strategies to combat these other forms of aliasing.

For a window size of 1024×1024, a bitmap texture requires $1024 * 1024 * 4 = 4$MB. In the most complicated case I test, the storage is 1.5MB. In the simplest case, the storage is only 0.05MB, about 1.25% of the size of the raster image generated. Often a vector image can be stored more efficiently than a raster approximation.

6.5 Summary

In this chapter, I develop a robust system for rendering general SVG images. This system has overcome several weakness of my previous work.

I present a method to compute distances to arbitrary parametric curves, as long as their curvature is finite and the curve has a defined tangent at every point. I apply my approach to cubic parametric splines, but it could be applied to any differentiable parametric curve. This distance computation is simple and fast. It can be used to support vector texture mapping for 3D real-time rendering. The distance computation gives accurate distance for points that are close to the curve, so it can be used for special effects, such as embossing and stroke texturing.

Using vector textures and transparency, I can generate complicated geometry effects with antialiased edges. This can simplify the geometry of objects in a scene.

The distance computation used in this paper does not provide accurate distance for points that are far from boundary features. Errors may occur if accurate distances are needed for distant points. In the future, I would like to improve the accuracy of my distance computation in such cases.

If the winding rule is used for this test instead of the distance sign, there is no need to compute the distances to the endpoints, because all I need is the gradient. The gradient

always points toward the curve. If the x component of the gradient is positive, it means the curve is on the right of the test point; otherwise, it is on the left. It is worth mentioning that although the distance can be inaccurate and therefore the distance gradient also inaccurate, the sign of x component of the gradient is always correct. The way I find the closest point always finds a local extremum or one of the endpoints. If it is the extremum, the query is on the normal line at the extremum. The consistent shape of the curve ensures that the x value of the distance gradient is consistent.

A special case is possible: the horizontal component of the gradient x can be 0 when the closest point is one of the endpoints. In the general case, I can compute the y range between the endpoints of the curve. If the y value of the query point is within this range and the x component of the distance gradient is positive, I can determine that a ray shot to the right will intersect the curve. In case the x of the distance gradient is 0, I use the line segment that connects the two endpoints of the curve, and test if the ray intersects this line. These two different cases can be implemented by conditional assignment without branching. In all other cases, the ray does not intersect the curve.

Using the winding rule not only simplifies the distance computation, but also avoids the error shown on the right in Figure 6.6 which is caused by inaccurate distance computation. It also supports self-intersecting paths.

My system appears to be memory bound, so in future work I would like to develop more efficient data structures to reduce the memory bandwidth and texture lookup delay.

Chapter 7

Parallel K-Nearest Neighbor Search and an Alternative Data Structure for Vector Texture Acceleration

K-Nearest Neighbor (KNN) search is an important problem in computer graphics. It appears repeatedly in flock simulation, collision detection, robot planning, N-body simulation, surface reconstruction, remeshing, photon mapping, caustic reconstruction and other applications. In addition, the KNN search problem is closely related to the problem I have to solve in vector textures. In KNN, I am searching for the closest curves to a query point. Therefore, by studying the simpler problem of KNN I can hope to develop a better approach for accelerating the distance search in vector textures.

Usually, KNN is implemented using a spatial data structure based on a tree that is then recursively traversed using backtracking. This approach requires a stack and in the worst case the entire tree is searched. When I try to apply these algorithms to current many-core vector processors, such as the IBM multi-core Cell BE or GPUs, the conventional tree structures either cannot be applied or operate with degraded performance because these architectures have limited local memory, control flow is expensive, and some of them do not even support the addressable local memory needed for stacks. Also most tree data structures do not consider the impact of spatial coherence, which is substantial in many-core architectures. An alternative approach is also possible, the grid, but whatever approach is taken in the general KNN problem the potential for artitrarily variable density needs to be considered.

In this chapter, I explore R-tree on the Cell BE and GPU as a way to solve the KNN problem. The R-tree structure can be adapted to parallel computing for many-core vector processors and can also generate accurate KNN results. Most importantly, I show that this tree structue can be applied to acceleration of vector texture distance search and can reduce the required storage relative to other approaches.

7.1 A Data Structure for Vector Graphics Data

I use Hilbert R-tree to implement parallel K-Nearest Neighbor search on the IBM Cell Broadband Engine (Cell BE) processor. For the Cell architecture, the Hilvert R-tree is better than kD-trees, octrees, or adaptive octrees in that the Hilbert R-tree is self-balancing and also addresses the problem of spatial coherence. The balanced tree structure allows the workload not only to be evenly divided among multiple cores but also avoids the use of pointers. Avoiding pointers improves bandwidth utilization, reduces storage space, and improves spatial coherence.

The same data structure is later implemented on the NVIDIA GeForce, where I also use it for vector data storage. While this approach more precisely solves the nearest-neighbor problem with better theoretical asymptotic complexity than my previous approaches, it turns out this data structure is not suitable for real-time rendering because of many dependent texture lookups and tree backtracking. On current GPU architectures with their limited support for local memory the performance is slowed down by an order of magnitude. However, I still think it is worth mentioning as it results in more compact data storage, and it is useful in cases where storage size is more crucial than performance. Also, on future GPUs (or GPU programming interfaces) with better support for prefetching, local memory, and bulk memory transfers, this approach may perform better. Finally, if a fast sort is available it is possible to build the data structure on the GPU, which would allow for dynamic update, something I do not attempt in my previous work.

In the following sections, I will first review the Cell architecture, which is similar to GPUs in some ways but significantly different in others. Next, I will review previous work in this area. I will then describe my approach to solving the KNN problem, which is based on a balanced Hilbert R-tree.

7.2 The Cell Architecture

The Cell BE architecture is jointly developed by IBM, Sony, and Toshiba, starting in 2000. It is a heterogeneous multi-core architecture targeting a broad range of applications. Instead of putting multiple general-purpose CPUs together, this new architecture combines one IBM 64-bit Power Processor Element (PPE), and eight specialized high-performance vector cores, known as Synergistic Processor Elements (SPEs) to achieve high efficiency. With a clock speed of 3.2 GHz, the peak performance for single precision is approximately 204 GFlop/s. It should be noted that in the original version of the Cell BE the peak performance for double precision is only 21 GFlop/s. However, a new version of the Cell BE has been recently released that improves this to over 100 GFlop/s. In comparison, a single core from the 2.33 GHz quad-core Intel Xeon E5345 x86 has a theoretical peak of only 18.5 single-precision GFlop/s. Even two of these recent quad-core Intel processors have less performance than a single Cell BE processor. Figure 7.1 shows the layout of a Cell BE processor.

The PPE is the main control element for management tasks such as allocating threads,

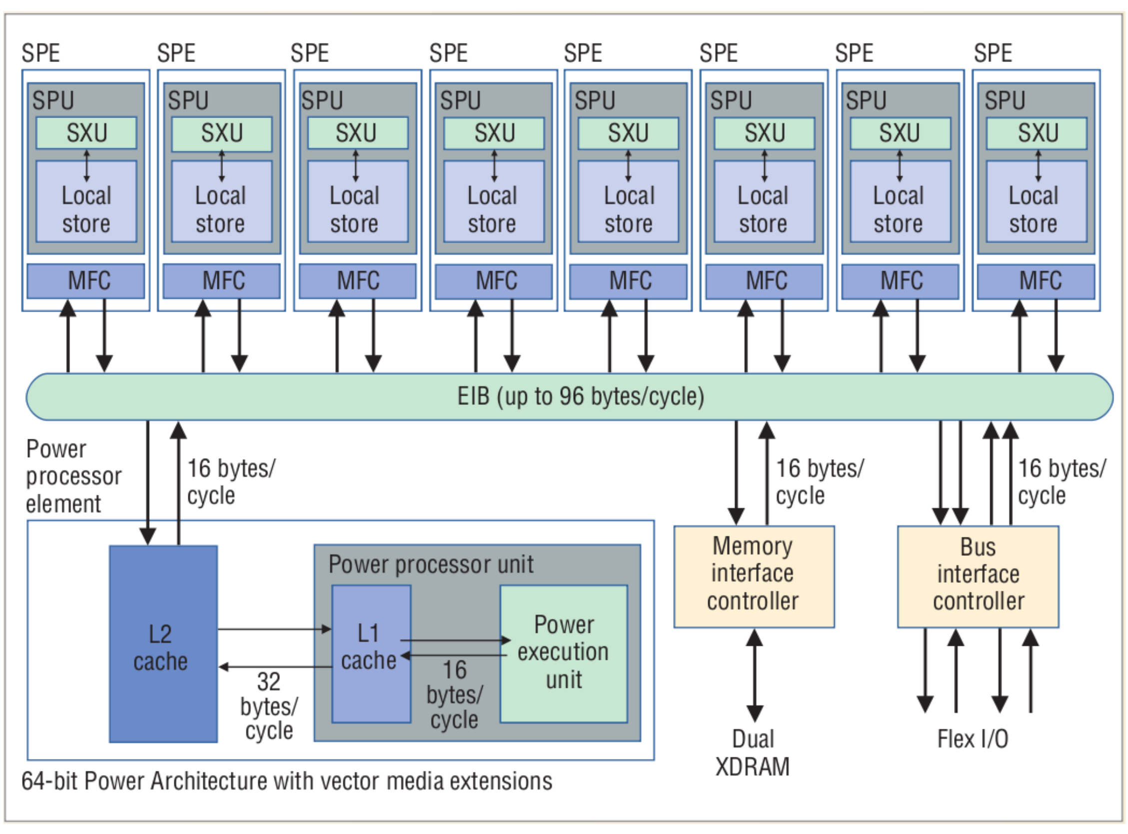


Figure 7.1: *The Cell Broadband Engine (Cell BE) architecture.*

assigning workloads among the SPEs, and monitoring and waiting for the computation to finish. Compared with the SPEs, the PPE has significantly lower performance, representing only 3% of the overall capability of the system. However, the PPE has a cache, unlike the SPEs, and also has the necessary memory management hardware to support virtual memory and protection. It is the only processor that runs the operating system; the SPEs are used to offload work in particular threads.

The SPEs are 4-way SIMD processors tuned for data-intensive computations such as multimedia, encryption, and scientific computation. The instructions and registers are 128 bits wide which can process one vectorized quadword at a time. The SPE branch architecture does not support dynamic branch prediction, but does support static branch prediction. The penalty for a mispredicted branch is high, more than 15 cycles. Each SPE has 256KB of local storage for both code and data, but has no cache.

The PPE and the SPEs can communicate through an on-chip high-bandwidth element interconnect bus (EIB) with a peak bandwidth of 204.8 GB/s. Data sharing between the SPE local memory and the PPE main memory is through memory flow controller (MFC). Each SPE can communicate with both the PPE and directly with other SPEs. The MFC can also transfer data between a local and an external memory using block transfers. The total external memory bandwidth is 25 GB/s. In the SPEs, data transfers and computation can be executed at the same time. Since cache is not supported, it is crucial to use the MFCs to schedule transfers of data to and from main memory in such a way as to hide the large memory latency, which is on the order of hundreds of cycles. In addition, large block transfers are significantly more efficient than small transfers.

7.3 The KNN Problem

Given a dataset consisting of a collection of N-dimensional geometry objects and some queries consisting of another set of such objects, the goal of KNN search is to find the K-nearest points for each query point. Depending on the applications, the dataset and the queries can be the same or different. For example, in information retrieval, we need to find the best matches but the queries are different from the database. In flock simulation, to avoid collisions among neighbors, the queries and the dataset are the same, although my approach could support the case where they are different. In my implementation, the queries and the dataset are the same. Also, I focus on the case where the query set is large so that sufficient parallelism can be derived by doing all searches in parallel.

In general, data in either the dataset or the query can represent a variety of different geometric shapes. Geometric primitives in spatial databases can consist of points, triangles, or curves. Also, abstract spatial data bases may store data with a large number of dimensions. For example, a computer vision database may store feature vectors for object recognition, and these feature vectors can be thought of as points in a high-dimensional space. In computer graphics, I mostly handle data with dimension two or three, while in other areas, the data can have higher dimensions. My algorithm is only implemented for dimension two and three because I mainly target graphics solutions. In

my implementation, I also simplify the data to points, although it can be extended to more complicated geometry shapes easily.

7.3.1 Related Work

Solving the KNN problem is a time consuming task, especially when both the dataset and the query set are large. Until now, applications in graphics involving KNN either have been rendered offline or have used approximate solutions. Approximate solutions work well in some applications that are not sensitive to the approximation. For example, in animal crowd simulation, human eyes are not sensitive to the number of the neighbors in terms of influence, and it is hard to predict how animals think, therefore, approximations suffice. However, in other applications, for example in photon mapping, approximate algorithms may cause artifacts. As a result, high quality photon mapping using more accurate KNN solutions are generally rendered offline. I would like to exploit the power of the Cell BE processor and GPU and develop solutions that achieve interactive speed, by which I mean greater than 15fps, for such graphics applications.

Typically the kD-tree [4, 5] is used in graphics to find exact solutions to the KNN problem. The kD-tree is a space partitioning data structure with N representing the dimensions of the space. The tree generation consists of partitioning the space with axis-aligned planes over one dimension at a time, and cycling through every axis repeatedly and recursively. At each level of tree, a split operation choses a pivot point and splits the points in a space into two sub-spaces along the designated dimension by comparing with the pivot coordinate. Variants of the algorithms store the objects being indexed in different ways. They can be stored either at all tree nodes, or only at the leaf nodes. The kD-tree has been used as an efficient data structure in a variety of applications such as range search, nearest neighbor search, ray tracing, and so on. When used in searching, the results can be either accurate or approximate depending on the termination rules.

The general kD-tree construction method does not guarantee that the tree is balanced. However, a balanced tree structure is useful for implementing KNN search on many-core processors and in particular on the Cell BE SPEs. An unbalanced tree requires extra data to record the indices of the children and parents for the tree nodes, and requires more control flow to handle the different cases possible and to terminate the search at different depths. As the Cell SPEs have limited local storage, the extra data may cause more data transfer between the local memory and the main memory. Furthermore, the SPEs are optimized for sequential computation, and control flow will reduce their performance. Finally, a balanced tree can be stored coherently in an array and this permits better use of the MFCs in the Cell processor.

To generate a balanced tree, the point coordinates can be sorted at each splitting and the median point chosen as the split point so that each subtree has an equal or nearly equal (within 1) number of points. Although the sorting is only within each subgroup, every point is involved in the sorting. Sorting can therefore be a bottleneck for the algorithm. Blelloch [6] has a parallel algorithm for KNN search using kD-tree. The kD-trees for

raytracing often do not use sorting like this since minimizing the surface area of the resulting kD-node is important for efficiency, and the median split does not give that.

The octree is another space-partitioning data structure appropriate for 3D space. At each node, the octree uniformly splits the space into eight sub-spaces simultaneously, using fixed split points in the middle of the cell. Its corresponding form in 2D is the quad-tree. While the octree generation is simple and fast, it does not guarantee a balanced tree in general if the points are not uniformly distributed. Therefore, the octree potentially has all the problems caused by an unbalanced tree.

It is also worth mentioning bounding volume hierarchies, which unlike kD-trees and octrees allow overlapping bounding regions around each subtree. The kD-tree has been extended to allow overlaps in a data structure called the interval kD-tree [58, 70] and the R-tree, which I will introduce shortly, also allows overlap. Bounding volume hierarchies are easier to be extended to geometric objects since with partitioning it may not be possible to find a plane that separates extended objects, which requires either inserting objects multiple times or splitting them. With bounding volume hierarchies such as the interval kD-tree and the R-tree this problem does not arise.

As mentioned above, some applications do not necessitate accurate KNN results, and since approximate algorithms are faster, they often adopt such algorithms. These applications include real-time simulations and renderings. In particular, a uniform grid is often used in practice [49, 50, 51, 22, 59]. However, the uniform grid is most useful when the maximum density is bounded, which is the case in some important applications such as flock simulation.

The uniform grid approach overlays a uniform grid on the space. In each grid cell, a list of the points that fall in this cell is created. During search, if we detect that some points in the grid cell need to be checked, all points in the grid cell are checked. When grids are used in the cell processor, the maximum number of points that can fall in one grid cell is not known, so a predefined maximum number is chosen manually. If the points exceed the maximum number, for example in flock simulation, we have to restrict the behavior of the flock so that they do not over crowd one cell. Special care should be paid to the size of the grid. If the grid is too coarse, there will be too many points in one cell, and the search will be reduced to brute-force search, especially when the points are non-uniform. If the grid size is too fine, and the maximum space is allocated for each cell to record its points, a large storage space is required. The size used to store the grid can be reduced by using a sparse data structure such as a hash table. However, this approach cancels out the spatial coherence advantages of using the grid.

An R-tree [24, 2, 3] organizes the data in a bounding box hierarchical tree. Each node is a bounding box enclosing all the subtrees. The number of children is between a minimum and a maximum. The actual data structure used to store the R-tree is often a B-tree, which supports variable-arity nodes but also supports coherent block transfers and automatic balancing. However, a B-tree can still waste space and bandwidth in the internal nodes of the tree. The principle of a good R-tree is that the bounding boxes are as small as possible, i.e., the data are grouped as tightly as possible. Space-filling curves are often used to convert the multi-dimensional data into one dimensional indices. The

Hilbert curve is one choice for ordering objects in a coherent fashion so that sorting can be used to group them [31]. This is based on the observation that if the Hilbert indices of the data are close, their spatial locations are also close. The opposite is not always true; however, it is true enough on average that this approach is useful.

7.3.2 Parallel KNN Search

For the Cell BE implementation of KNN, I choose the Hilbert R-tree structure because this approach can give me balanced trees even without a B-tree implementation. My algorithm has the following steps.

1. Convert the point coordinates to Hilbert indices.
2. Sort the data coordinates according to their Hilbert indices.
3. Construct a hierarchical R-tree by recursively splitting the sorted array at its midpoint.
4. Search for K-nearest neighbors.
5. When points move, dynamically rearrange point locations according to the influences of the neighbors.

My algorithm differs from a B-tree implementation of general R-trees in that there is no unused space for children. Every node has the maximum number of children. Therefore, not only is the tree well balanced, it also makes maximum use of bandwidth and storage space. It is not even necessary to store pointers in internal nodes. The data is simply placed in a sorted array or a sequence of sorted arrays.

In my implementation, the number of points in the data set I use for testing is $128 \times 128 = 16K$. The data are represented by points. I search for K=5 Nearest Neighbors for each and every point within the same dataset. I then use this information for a N-Body force field simulation where the locations of all the points are adjusted according to the influence from the five nearest neighbors. Potentially, every point may move to a new location. However, these updates are spatially coherent; in general, for every cycle of the simulation the points only moved a small distance.

As the Cell PPE is slower for heavy computation, but more flexible in control, I would like the workload to be distributed quickly to the SPEs, so the PPE only has to wait for the job to be done. As a general principle, all the SPEs should be utilized, and the workload should be equally split among the SPEs, so that none of them is overly loaded, and none of them sitting idle. In the implementation, the PPE allocates the workload, waits for the SPEs to finish, retrieves the results, and allocates the workload for the next step.

In my implementation, each step of the algorithm given above is done in parallel, with synchronization barriers between each step.

Step 1: Conversion of the coordinates

Hilbert curve is a space-filling curve that recursively traverses the whole of a bounded, square or cubical region. Any two successive points on the curve are close together physically, in the sense that there is a bound on the N-dimensional distance for a given distance along the Hilbert curve. The Hilbert curve itself is the limit of a recursive subdivision process. The first three steps of this process are shown in Figure 7.2. Note how the Hilbert curve gives a traversal through all the cells in a space so that I always step from neighbour to neighbour directly.

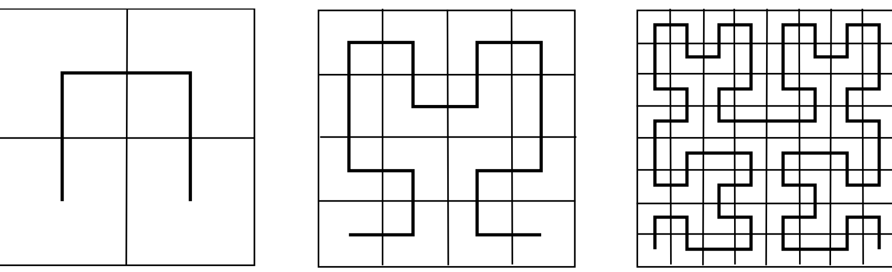


Figure 7.2: *Hilbert curves of first three orders.*

Rather than using recursive decomposition, I use an efficient bit-manipulation algorithm to convert the multidimensional input coordinates to the 1-dimensional Hilbert index [28], which is the distance along the curve. These conversion algorithms apply to positive integers only. Although the coordinates of the input points are represented using floating points and can be negative, I can always convert them into positive integers by finding the minimum and maximum of each coordinate and shifting, scaling and rounding up. Note that rounding up floating point numbers to integers snaps the floating coordinates to integer coordinates, and different coordinates may be snapped to the same integer coordinate. This is not a problem in my application, because if coordinates of points can be snapped together, it means they are physically close. Their Hilbert indices will be the same, and after sorting, these points are still together, although their internal sequence can be random.

Converting all the 2D or 3D coordinates in the dataset to Hilbert indices can be done in parallel. The workload is equally split among the SPEs. The conversion for each point takes the same amount of time, so theoretically all SPEs should finish at the same time.

Step 2: Sort the locations

The sorting is an implementation of a parallel bucket sort optimized for the Cell processor. The bucket sort is used to sort the Hilbert indices. However, sorting the Hilbert indices alone is not enough. To improve the coherence of search, I need to shuffle the original data according to the sorted indices. In the SPEs, operations process quadword (128 bit) vector data in parallel. In my implementation, the coordinates of the points have been converted to integers, the coordinates of a 2D point takes 2 words (64 bits), and a

3D point takes 3 words (96 bits). Their Hilbert indices are also 64-bit and 96-bit integers respectively. For a 2D point, I put its Hilbert index in the most important 64 bits and the coordinates in the least important 64 bits as shown in Figure 7.3. The sorting is based on the value of the 128-bit quadword data. This treats the 4 words as an unsigned 128-bit long integer. The sorting results of the quad-words should be the same as the sorting results of only Hilbert indices, because the the least important bits do not influence the order of the most important bits.

Hilbert index and integer coordinates are packed in 128 bits

Figure 7.3: *Hilbert index and coordinates of a 2D point are packed into one memory address.*

For 3D data, the Hilbert index of a point takes three words, i.e., 96 bits, so there is only one word left, which is not enough for the coordinates. One solution is to put a sequence number for the dataset at the least important word. When the indices are sorted, the sequence numbers are also sorted. Afterwards, I can use an extra step to shuffle the data based on the sequence numbers. Another solution is to allocate another quad-word for the coordinates for each index, and sort pairs of quadwords. While the index is shuffled in the sorting, the data "payload" can be shuffled as well.

Step 3: R-tree construction

A general R-tree allows its nodes to have multiple children as long as the number do not exceed a maximum. In my prototype, I use binary tree, although as I will discuss, since it is balanced I do not need pointers. Each node has exactly two children. A general R-tree also allows insertions and deletions. My implementation however does not leave space for insertions and does not support deletions. I apply this to a crowd simulation, so each point will change its location after finding its neighbors and computing their influence. If I delete and insert each point, it will involve pointer chasing, and the tree will be unbalanced soon even if I start with a balanced tree. Instead I re-sort the data set, and re-generate the R-tree every time after a change of locations. This way, I will have a complete balanced tree all the time, which is convenient for the KNN search later on. The benefits of a complete R-tree in searching will be described later. For the binary R-tree, if the data set is of size N, the binary tree takes $2N-1$ space. I could also use a quad-tree to save storage. I use an algorithm that sorts the data from scratch but do not take advantage of the fact that after perturbation the data is still "nearly" sorted. Future work could optimize the update by taking advantage of this.

The local store in one SPE is only 256K. In my implementation, I have $128 \times 128 =$ 16K 2D points. After sorting, the Hilbert indices are not useful anymore. All I need are the sorted coordinates, which will be used in constructing the R-tree. For storage

efficiency the coordinates for two 2D points can be put in one quad-word. After this compaction the 16K points take 128KB of storage.

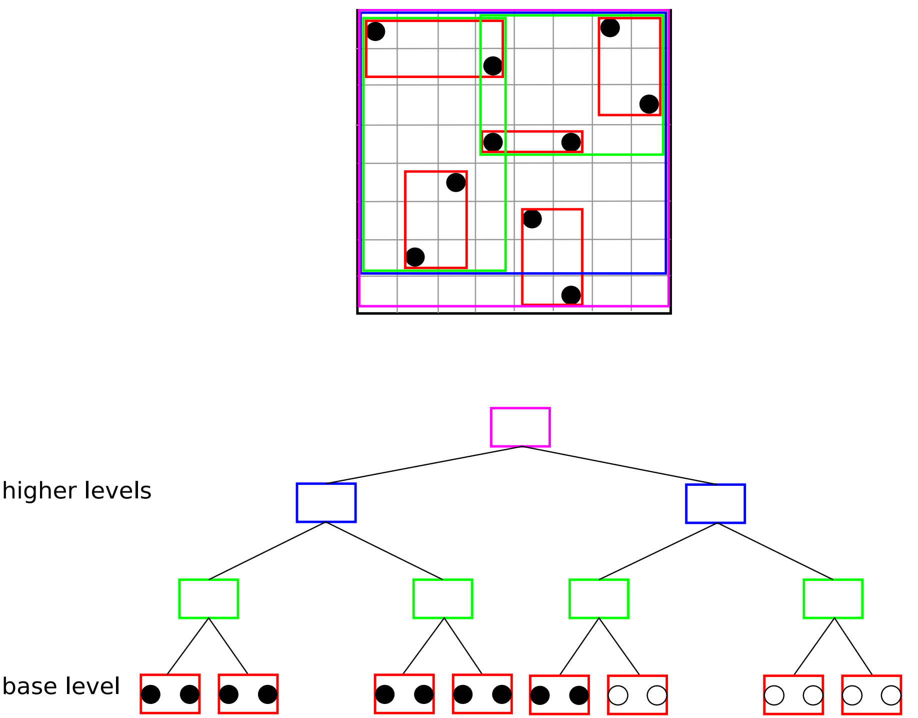


Figure 7.4: *Points are grouped according to their Hilbert indices, and an R-tree is constructed by computing the bounding box between every pair of adjacent nodes. If the number of nodes in the database is a power of two this results in a perfectly balanced tree. Otherwise, leaves can differ by at most one level in depth.*

The R-tree is constructed in a bottom-up way. The first stage in constructing the R-tree is to compute bounding boxes for the base level of the tree from the original coordinates. It computes the minimum and maximum of x and y coordinates from every pair of adjacent points. For 2D points, each bounding box takes a quad-word to store. So the base level of the tree also uses 128KB storage. The second stage constructs the higher levels of the tree from the base level. At each level, I compute the bounding box of the two adjacent nodes from the lower level until I have only one node left, and this node is set as the root. The bounding box computation is simple and only involves minimum and maximum operations. Figure 7.4 shows how points are grouped according to their Hilbert indices and how the R-tree is constructed. Different layers of bounding boxes are indicated with different colors. Here I assume the number of points is a power of 2, therefore the R-tree is completely balanced. If the number of points is not power of 2, I can always insert extra points that do not affect the search result to make it power of 2.

The higher levels of the tree altogether take (128KB-16B). For simplicity, I round the number up to 128KB. During the whole tree construction, I allocate 128KB for the data and leave 128KB for the codes. However, during search I only store the index (interior

nodes) in the local store, not the data for the leaves. I have to use the Memory Flow Controller (MFC) to fetch the leaves as needed.

If too much data is involved in the tree construction, they cannot fit into the local memory. What I could do to get around this is to split the original data equally for the SPEs. Each SPE constructs one subtree and returns the results to the main memory. For higher levels of the tree, the PPE combines all the subtrees. If the construction for higher levels of the tree can fit in one SPE, the PPE can give the job to one SPE to avoid latency caused by data transfer. If the data is still too big to fit in one SPE, the PPE could also distribute the workload to multiple SPEs again, using a multipass approach.

Step 4: KNN search

There are two strategies to implement the actual search. Suppose I have constructed the R-tree and the number of SPEs is N. The first strategy is to split the R-tree into N subtrees with equal size, so each SPE holds its own subtree. Given a query, every SPE searches for the K-nearest neighbors in its subtree and returns the K results. The PPE waits for all the SPEs to finish their work, collects the results, and sorts them to find the final K neighbors. This approach has some advantages. To find the K neighbors, each SPE only needs to load its subtree into its local storage. If the subtree is larger than the local storage, the subtree needs to be shuffled in part by part. However, if the subtree is small enough, the need to do this shuffling can be avoided.

I decide not to use this strategy because of the following consideration. For a given query, its K-nearest neighbors may not be uniformly distributed among the subtrees. Some SPEs may have more neighbors, and take longer time to search, while others may have fewer neighbors, and quit searching early. The PPE however has to wait for all the SPEs to return before it can proceed to sort the final results. If the SPEs that return early are asked to wait, it is a waste of their power. A more reasonable choice is to allocate new work to them, but the PPE has to hold the intermediate results in the main memory. The PPE has to keep track of the memory allocation for each query. This is a big workload for the PPE. One final problem with this approach is that partitioning the work is equivalent to a brute-force search over KN neighbours, so is not asymptotically efficient.

The alternative strategy is to separate the query data among the SPEs, so that each SPE has the same number of queries. Each SPE is responsible for finding the final KNN for a query. The workload for the PPE is therefore reduced to just assigning the query group and receiving the results. This strategy avoids the bottleneck of using the PPE in computation. However, it is obvious the search in each SPE has to be over the whole tree and could involve more data transfer to shuffle in the tree. Double buffering should be used to hide the latency, although this does cut the available storage in half. This is the approach I take in my implementation, although my dataset is small enough that I could avoid double-buffering.

Before searching, each SPE will load the R-tree into its local memory. If the tree is small enough to fit in the local memory, there will be no data transfer for the tree during the searching. Otherwise I load the upper levels of the tree into the SPEs. Because my

search is a top down search, every search goes through the top part of the index, but not necessarily through the lower subtrees. As a practical principle, for the 256KB local storage, normally 128KB is preserved for the code, and 128KB is used for the data. This allocation can be modified according to the data size and code complexity. As many levels of the tree as possible should be loaded into the local memory at the first time to make the following data transfers fewer and smaller. In my implementation, only higher levels of the R-tree are loaded into the local SPEs. The base level is not necessary, because I do not really need to check with the base level bounding box, I can just go to the original coordinates and compute the real distances directly. Since both the higher levels of the R-tree and the original coordinates of the points take 128KB, they cannot both be loaded into a SPE at once; otherwise, there will be no space for the code. What I do is to load the tree into the local space and only parts of the original coordinates into the SPEs. When I need to compute the distances to the points whose coordinates are not in the SPEs yet, I load them as needed.

In the SPEs, the tree is stored sequentially in an array. The root of the tree is in element 0, its two children are in elements 1 and 2, and other levels are stored similarly.

The search is a depth-first search. Before searching, I initialize the K-nearest distances. They could be set to infinity. However, since the data are sorted, I can compute the distances to its K neighbors in the array, and sort the distances in ascending order. This is a tighter estimate, which can exclude some subtrees, and increase the search performance.

The search starts from the root. For its two children, I compute the distances from the query to their bounding boxes. I compute the distance to the closest point on the bounding box. Since an accurate computation entails many cases and significant branching, a conservative method is used instead. First, compute the distance from the query to the center of the bounding box l, then compute the smallest circle of radius c that can enclose the bounding box, i.e., half of the length of the diagonal. The distance from the query to the box is estimated as $l-c$. This is a conservative estimate that is always smaller than the real distance. It guarantees that no candidates will be missed (in information retrieval terms, no false dismissals).

Suppose the distances are computed. If both the distances are larger than the Kth distance in the candidates, it means no points enclosed in the bounding boxes are closer than the candidates, and the search will terminate early. If one of the distances is closer, I only go down this subtree, and discard the other one. If both the distances are closer, I compare the two distances, and choose to go down the subtree with the smaller distance first, and push the other one into a stack. Once I go down one of the subtrees, I again check its two children, and repeat the same procedure as above. This process continues recursively until I reach the leaf nodes. Then the distances between the leaf nodes and the query are computed and compared. Remember the leaf nodes are the real coordinates of the points, so the distances are the exact distances between the query and its neighbors.

As the tree is completely balanced, given a node's location in an array, it is easy to compute the locations of its child nodes. I use three parameters in this computation, Given a node, suppose I know the tree level l it is in, its index number i in this level, and

the beginning location in the array of this tree level ls. The location of its children in the array is then computed as $ls + 2^l + i * 2$ and $ls + 2^l + i * 2 + 1$. In the initial case, I have the three parameters for the root, which are all 0. Similarly, I can compute the parent node location.

The next step is to handle the subtrees pushed into the stack. Every time a subtree is popped out, the distance to its bounding box is compared with the new K-nearest distance to exclude possible subtrees. If the distance is still smaller than the Kth candidate distance found so far, it is possible that a closer point might be found in the popped subtree, the search goes down into this subtree as described above. The procedure continues until the stack is empty.

It is possible that the tree is too large to be loaded into the local store at once. However, when I go down along a subtree, it is possible that the subtree data is still in the main memory, and needs to be loaded in with an MFC transaction.

Step 5: Rearrange point locations according to the influences of the neighbors

In the implementation, I use the 5 nearest neighbors and an update rule (whose exact form is unimportant to this discussion) to decide the new location for each point. This rule implements a physical "molecular" simulation involving attraction and repulsion between points. If two points are far away, they tend to attract each other; otherwise they repel each other. This simulation is not meant to be accurate or meaningful. It is just a way to show how crowd simulation could be implemented.

7.4 Results and Future Work

I test with 16K points, and look for 5 nearest neighbors. The query data are the same as the candidates. For this data set I achieve an interactive speed of 8fps with 16 SPEs.

The time measured does not include the time used to adjust the locations of the points. This time varies depending on different physical simulations. It does, however, include the time needed to rebuild the R-tree.

The R-tree is a good data structure for KNN search, especially for the Cell BE. It provides a completely balanced tree structure that avoids extra space for pointers, and also allows the workload to be allocated equally among SPEs. Extra space for pointers will take up the limited storage in SPE local memory, and fewer other data can be transferred to local memory. With a pointer-based structure more data tranfer between the local memory and main memory will occur, which inevitably will influence the performance. Also pointer chasing is not desirable in the SPEs, because SPEs are designed for SIMD computation and control flow affects performance.

The original version of this algorithm used the IBM SDK for the Cell BE. I also create a GPU implementation of this algorithm using RapidMind. The algorithm is extended into a general data structure supporting closest neighbor search, and it is applied

in my vector texture problem. In vector textures, the primitives that need to be stored include not only points, but also line segments, splines, and elliptical arcs. Similar to the approach with points, I compute the bounding box of these features and constructed a binary R-tree on top of them. Then I search for the closest boundary feature for a query point. The searching is similar but with small changes. Instead of computing the distance between points, I compute the distance between the query point and the primitives, which can be any one of the types mentioned above. Other than these, the computations such as the distance from the query point to the bounding boxes and the choice of which subtree to descend are all the same. Here I only look for the closest neighbor instead of K neighbors.

One thing worth mentioning is that when the algorithm was implemented, GPUs did not support random local memory, so I could not implement a real stack. However, because the tree is balanced, and I can compute the locations of a node's parent and children easily, I am able to simulate a stack with simple arithmetic operations and an unsigned integer (using left and right shifts to implement a stack with a small number of bits). Although my simulation only works for binary trees, it is possible to extend it to support quad-trees.

I regard the balanced R-tree as a suitable structure for similar searching algorithms in graphics. In terms of storage size, it takes double the space of the original data. In the examples used in the third stage of my research, the most complex example had 6904 features after splitting all boundary features into monotonic regions. Suppose all these are cubic splines (which is not true in the input data, but I covert them for the simplicity of computation) and each spline has four control points, all represented by floating point numbers. The four control points for a cubic spline take 32 bytes. Altogether, these features plus an R-tree index would take about $6904 \times 2 \times 32 = 0.44$MB. This number is smaller than storage size of $0.5MB$ that results in my previous results using a grid with grid size 64×64. In other cases, however, not all features will be cubic splines, and line segments take half of the storage at the leaves but still take the same amount of space to index with the R-tree. However, in theory the R-tree will scale better and be more robust for non-uniformly distributed data than the grid.

In terms of implementation, I have shown that this algorithm can be implemented even if the GPUs do not support random memory access, so it can be used in older generation of GPUs.

However, because of the complex tree searching and backtracking, when I use this approach in vector textures, the absolute performance is an order of magnitude slower than a grid-based approach. In applications whose performance is crucial, the grid structure currently outperforms the R-tree. However, in applications whose memory storage is crucial, the R-tree is likely to be better than the grid structure. Also, as GPUs evolve and better support local memories which can be used as stacks, better implementations of the R-tree approach will become possible.

In the future, I would like to explore more applications of the R-tree in graphics and other research areas.

Chapter 8

Conclusion and Future Work

The research described in thesis is a relatively new area in computer graphics. This research could not have been done earlier because of the limited programmability and performance of GPUs. With the improvement of new generations of GPUs in recent years, I am able to provide higher quality images for 3D real-time rendering. Resolution-independent vector textures are useful in many contexts since they provide both compact storage size and high quality.

Vector graphics is playing a more and more important role in computer graphics. Traditional industries such as movies, games, and architectures require higher quality rendering results. New potential applications are also appearing in internet content. So far, internet content formats such as Flash only support 2D vector images, but it is likely that they will support 3D vector textures soon. Also, graphical user interfaces are evolving to use both 3D effects and vector graphics content.

I explore several different ways to support vector textures in 3D real-time rendering. My approaches considered both bilevel images and general vector images as described in the SVG format. My research goals include compact data structures for vector images, high quality images with anisotropic antialiasing, high performance suitable for real-time rendering, and efficient distance computations for special effects.

8.1 Contributions

The contributions in this thesis include:

1. To support high performance random access to vector textures, different methods are developed to reduce the global computation problem to a local computation problem. This results in a limited number of features being used to compute the color for each query point.

2. To store the vector image information efficiently, both grids and R-tree data structures are explored. These data structures provide both compact storage and fast feature retrieval, although at present grids are superior in terms of performance.

3. My simulated annealing packing scheme for grids provides a single sequence of instructions for the computations of all query points. This feature specifically fits in well with the SIMD nature of graphics acceleration architectures.

4. My anisotropic and isotropic antialiasing method for rendering high quality images is fast and easy to implement compared with other antialiasing approaches. Anisotropic antialiasing is especially important for the legibility of text.

5. I use both line segments and arc splines to approximate high order curves. I show the cost of a distance computation to an arc is comparable to the cost of computing the distance to a line segment. However, arc splines approximations give better continuity and potentially require less storage.

6. I develop an easy, fast and robust way to compute the distance to any parametric curve, for query points within the radius of curvature. Using this approach, I am able to render cubic splines and elliptical arcs with any required degree of accuracy without increasing storage space.

7. I am able to render distance-based special effects. It would be easy to extend my results to other distance-based effects such as variable-width strokes, halos, and drop shadows. My system is the first that is able to render higher order curves exactly along with such special effects.

8. I provide a solution for second order antialiasing along embossing ridges by developing a rule for merging the two closest distances to curves along the ridges.

9. Using vector textures, it is possible for a user to generate visual effects that imply complicated geometry with simple underlying geometry.

10. My system can also potentially be used to texture strokes *along* the boundaries, because I not only return the closest distance to the curves, but also return the parameter along the curves. In particular, a line segment for arc spline approximation with C^1 continuity can be used to texture dashed lines along the borders with uniform dash lengths, since it is easy to reparameterize these representations by arc length.

11. Using my representations of vector textures, sharp corners are retained at any scale. They do not suffer the softening and rounding which are usually caused by approximated or interpolated sampled distances.

12. My prototype systems support all the main features of the SVG file format. I support mutiple layers, different types of boundary borders, and different types of fill colors. I feel it is important that a vector texture system support the full feature set of standard file formats so that standard vector content and existing content creation tools can be used.

13. I show that multiple layers can be rendered in different ways. Multiple layers can be rendered either by flattening the images or by compositing the images layer by layer at run time.

14. My approaches achieved real-time rendering performance, showing that vector textures can be used in practical systems.

8.2 Limitations and Future Work

My prototype systems still have many limitations that will need to be improved upon in future work:

1. Because of the long latency of dependent texture lookup in the GPUs, the performance of my latest work is memory bound. Alternative acceleration strategies should be developed that allow for fewer levels of dependent texture lookup to support higher performance. Note that this will also be a performance limit in future many-core CPUs.

2. The distance computation for parametric curves is only accurate when the points are on the convex side of the curve or on the concave side but within the radius of the curvature of the curve. Although I have shown through many empirical tests that this is good enough for antialiasing and the special effects mentioned in my work, applications that need accurate distance computation for points that are far away need a more robust distance computation.

3. My systems do not support paths with self-intersection. To support self-intersection, the winding rules should be used in the point location test rather than the signed distance. This also resolves several ambiguity problems at corners and may reduce the overall complexity of the system and improve performance.

4. I do not currently support real-time modification of vector textures. Supporting this, as well as animation, would be another interesting direction for future work. As online 3D virtual environments are getting more and more popular, it is desirable for users to design and decorate their own 3D online space. It is possible that eventually we will be able to draw artistic designs directly on surfaces in 3D scenes in real-time, although currently this can be a challenge even for raster textures.

5. My color blending only uses the closest boundary. This approach is not accurate for pixels that contain multiple boundaries, although the approximation is good enough to render high quality images. A more accurate method could be developed that could handle more complex situations around corners and thin features.

6. My work should also be extended to support periodic textures. It would be straightforward to transform the texture and make it repeat seamlessly along the boundaries.

8.3 Long Term Impact

My approaches are developed specifically for graphics accelerator GPUs. Within the constraints of current GPU architectures, many choices are made that would not be necessary if those limitations did not exist. For example, if GPUs supported control flow as well as CPUs, the distance computations for different points would not need to be the same; if GPUs supported random memory access as well as CPUs, multiple levels of dependent texture lookups would not be a problem either. However, many of the limitations of GPUs are there to improve performance. If GPUs are as general as CPUs, they might be as slow as them, as well. For example, GPUs are massively parallel, and as CPUs increase the number of cores they use (since they are no longer increasing in performance by scaling clock rate), they may have to shift to a similar programming model. In fact, CPUs are even increasing their vector widths. In 2010, Intel will introduce AVX to mainstream processors, which uses a vector width of 8 rather than the width of 4 used by SSE currently. This will also drive the performance characteristics of CPUs towards those of GPUs.

In addition, many of my research results are not limited to GPU-like architectures only. They are useful algorithms for other architectures and other applications. Reducing the number of computations for each pixel and efficient storage size as done in my approaches will always provide a benefit no matter what kind of architecture is used. I use a grid structure to restrict the computation for each pixel to a small subset of all the boundary features. I use Voronoi diagram analysis to accurately determine the boundary features that contribute to the color computation for each pixel. My methods of computing distances to boundaries and rendering antialiased boundaries either directly or approximately are all fast. These research results and some others are not restricted to GPU architectures only. They are general results useful for all kinds of computer architectures.

The data structures I use can also be applied in other applications in graphics such as ray tracing, collision detection, and so on. These applications are solving similar problems except the features are in 3D. The data information can be stored in grids or R-trees for fast searching of the nearest feature. Displacement mapping using local surface ray-tracing of features extruded from vector textures, for example, would be interesting.

The approximation of higher order curves with arc splines and the iterative computation of the distance to parametric curves can also be applied in other contexts, such as CAD, architecture, milling, and robot path planning.

Although my algorithms are specifically developed for 3D real-time rendering, they can definitely be applied in 2D and offline applications either directly or with modifications. The high-performance nature of my algorithms is an advantage even in applications that do not require real-time performance. They can be easily extended to support more advanced rendering algorithms.

In summary, I have developed a completely new way to represent texture images for real-time 3D rendering that provides major benefits in image quality. I feel that these new capabilities will be especially important in evolving "symbolic" uses of 3D

graphics in non-photorealisitic rendering, visualization, and graphical user interfaces. Vector graphics images represent such symbolic information directly and exactly. This direct representation not only allows a more accurate rendering of this information, it also preserves the information so that it can be manipulated as needed to convey the desired information in different contexts. As a result, information represented in vector images can be more fluidly transformed into different context and presentation modes. The symbolic and continuous nature of vector images are a natural complement to the sampled and digital nature of raster images. The computational power of GPUs now makes it possible to enjoy the benefits of vector images in interactive contexts.

References

[1] O. Aberth. Iteration methods for finding all zeros of a polynomial simutaneously. *Math. Comp.*, 27(122):339–344, 1973.

[2] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. In *Proc. 2004 ACM SIGMOD International Conference on Management of data*, pages 347–358, 2004.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proc. of ACM SIGMOD International Conference on Management of data*, pages 322–331, 1990.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. In *Commun. ACM*, pages 509–517, 1975.

[5] J. L. Bentley. K-d trees for semidynamic point sets. In *SCG '90: Proc. 6th Annual Symposium on Computational Geometry*, pages 187–197, 1990.

[6] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT press, 1990. ISBN 0-262-02313-X.

[7] R. P. Brent. *Algorithms for Minimization without Derivatives, Chapter 4*. Prentice-Hall, Englewood Cliffs, 1973. ISBN 0-13-022335-2.

[8] C. G. Broyden and J.A.Ford. A new method of polynomial deflation. *IMA Journal of Applied Mathematics*, pages 16: 271–281, 1975.

[9] Edwin Catmull. *A subdivision algorithm for computer display and curved surfaces*. PhD thesis, University of Utah, 1974.

[10] Franklin C. Crow. The Aliasing Problem in Computer-Generated Shaded Images. *Communications of the ACM*, 20(11):799–805, 1977.

[11] Franklin C. Crow. The Use of Grayscale for Improved Raster Display of Vectors and Characters. In *SIGGRAPH*, pages 1–5, 1978.

[12] Tom Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. In *Proc. SIGGRAPH*, pages 131–138, July 1992.

[13] L. W. Ehrlich. A modified newton method for polynomials. *Comm. ACM*, 10(2):107–108, 1967.

[14] A. Eigenwillig, L. Kettner, E. Schomer, and N. Wolpert. Exact, efficient and complete arrangement computation for cubic curves. *Computational Geometry 35*, pages 36–73, 2006.

[15] John B. Fraleigh. *A First Course in Abstract Algebra*. Addison-Wesley, fifth edition, 1994.

[16] Sarah Frisken and Ronald Perry. Method for generating an adaptively sampled distance field of an object with specialized cells. *U.S. Patent Application 20040189642*, 2004.

[17] Sarah Frisken, Ronald Perry, and Thouis Jones. Detail-directed hierarchical distance fields. *U.S. Patent 6396492*, July 12 2005.

[18] 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 *SIGGRAPH*, pages 249–254, July 2000.

[19] S. Goedecker. Remark on algorithms to find roots of polynomials. *SIAM J. Sci. Comput.*, 15(5):1059–1063, 1994.

[20] Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 courses, SECTION: Course 28: Advanced real-time rendering in 3D graphics and games*, pages 9–18, 2007.

[21] Ned Greene and Paul S. Heckbert. Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics and Applications*, 6(6):21–27, 1986.

[22] Christiaan P. Gribble, Thiago Ize, Andrew Kensler, Ingo Wald, and Steven G. Parker. A coherent grid traversal approach to visualize particle-based simulation data. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):758–768, July 2007.

[23] Satish Gupta and Robert F. Sproull. Filtering edges for gray-scale displays. In *SIGGRAPH*, pages 1–5, 1981.

[24] Antonin Guttman. R-tree: A dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[25] Paul Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, U.C. Berkeley, CS Division, June 1989.

[26] Paul S. Heckbert. Texture mapping polygons in perspective. *Technical Memo No. 13, NYIT Computer Graphics Lab*, 1983.

[27] Paul S. Heckbert. Filtering by repeated integration. *Computer Graphics (SIGGRAPH'86 Proceedings)*, 20(4):317–321, August 1986.

[28] Jr. Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2002. ISBN 978-0-201-91465-8.

[29] Kenneth E. Hoff III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast Computation of Generalized Voronoi Diagrams using Graphics Hardware. In *SIGGRAPH*, pages 277–286, 1999.

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

[31] Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proc. of VLDB Conference*, pages 500–509, Santiago, Chile, 1994.

[32] M. Kraus and T. Ertl. Adaptive Texture Maps. In *Proc. Graphics Hardware*, pages 7–15, 2002.

[33] Charles Loop and Jim Blinn. Resolution independent curve rendering using programmable graphics hardware. In *Proceedings of ACM SIGGRAPH 2005*, pages 1000–1010, 2005.

[34] J. Loviscach. Efficient magnification of bi-level textures. In *ACM SIGGRAPH Sketches*, 2005.

[35] Michael McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh*. AK Peters, 2004.

[36] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader Algebra. In *ACM Trans. on Graphics (Proc. SIGGRAPH)*, volume 23, pages 787–795, August 2004.

[37] Joel McCormack, Ronald Perry, Keith Farkas, and Norman P. Jouppi. Feline: Fast elliptical lines for anisotropic texture mapping. In *Proceedings of SIGGRAPH'99*, pages 243–250, 1999.

[38] Robert McNamara, Joel McCormack, and Norman P. Jouppi. Prefiltered antialiased lines using half-plane distance functions. In *SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 77–85, 2000.

[39] D. S. Meek and D. J. Walton. Approximating smooth planar curves by arc splines. *Journal of Computational and Applied Mathematics 59(1995)*, pages 221–231, 1995.

[40] Diego Nehab and Hugues Hoppe. Texel programs for random-access antialiased vector graphics. Technical report, Microsoft Research Technical Report MSR-TR-2007-95, July, 2007.

[41] R.W.D. Nickalls. A new approach to solving of the cubic: Cardan's solution revealed. In *The Mathematical Gazette*, pages 77:354–359, 1993.

[42] V. Y. Pan. Solving a polynomial equation: Some history and recent progress. *SIAM Rev.*, 39(2):187–220, 1997.

[43] Ronald Perry and Sarah Frisken. Method and apparatus for rendering cell-based distance fields using texture mapping. *U.S. Patent 6,917,369*, July 12 2005.

[44] Z. Qin, C. S. Kaplan, and M. D. McCool. Circular arcs as primitives for vector textures. Technical Report cs-2007-41, School of Computer Science, University of Waterloo, 2007.

[45] Z. Qin, M. D. McCool, and C. S. Kaplan. Real-time texture-mapped vector glyphs. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 125–132. ACM SIGGRAPH, March 2006.

[46] Z. Qin, M. D. McCool, and C. S. Kaplan. Precise vector textures for real-time 3d rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 199–206. ACM SIGGRAPH, February 2008.

[47] G. Ramanarayanan, Kavita Bala, and Bruce Walter. Feature-based textures. In *Eurographics Symposium on Rendering*, 2004.

[48] Nicolas Ray, Thibaut Neiger, Xavier Cavin, and Bruno Levy. Vector texture maps on the GPU. Technical Report Technical Report ALICE-TR-05-003, 2005.

[49] C. W. Reynolds. Flocks, herbs, and schools: A distributed behavioral model. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 25–34, July 1987.

[50] C. W. Reynolds. Interaction with groups of autonomous characters. In *Proceedings of Game Developers Conference*, pages 449–460, San Francisco, 2000.

[51] C. W. Reynolds. Big fast crowds on PS3. In *Proceedings of Sandbox (an ACM Video Games Symposium)*, Boston Massachusetts, 2006.

[52] Arnold Schönhage. The fundamental theorem of algebra in terms of computational complexity. *Preliminary Report, Math. Inst. Univ. Tübingen*, 1982.

[53] Pradeep Sen. Silhouette maps for improved texture magnification. In *Proc. Graphics Hardware*, pages 65–74,147, 2004.

[54] John M. Snyder. Interval analysis for computer graphics. *Computer Graphics (Proc. SIGGRAPH)*, 26(2):121–130, July 1992.

[55] M. Tarini and P. Cignoni. Pinchmaps: Textures with customizable discountnuities. In *Eurographics*, pages 557–568, 2005.

[56] Jack Tumblin and Prasun Choudhury. Bixels: Picture samples with sharp embedded boundaries. In *Eurographics Symposium on Rendering*, 2004.

[57] Kenneth Turkowski. Anti-aliasing through the use of coordinate transformations. *ACM Trans. on Graphics*, 1(3):215–234, July 1982.

[58] Ingo Wald, Soloman Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics*, 26(1):485 – 493, Jan 2007.

[59] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, pages 485–493, 2006.

[60] J. Wallner. Generalized multiresolution analysis for arc splines. *Mathematical Methods for Curves and Surfaces II*, pages 537–544, 1998. ISBN 0-8265-1315-8.

[61] D. J. Walton and D. S. Meek. Approximation of quadratic bézier curves by arc splines. *Journal of Computational and Applied Mathematics 54(1994)*, pages 107–120, 1994.

[62] D. J. Walton and D. S. Meek. Approximation of a planar cubic bézier spiral by circular arcs. *Journal of Computational and Applied Mathematics 75(1996)*, pages 47–56, 1996.

[63] Hongling Wang, Joseph Kearney, and Kendall Atkinson. Robust and efficient computation of the closest point on a spline curve. In *Curve and Surface Design: Saint-Malo*, pages 397–405, 2002.

[64] R. M. Wankere. Iterative methods for roots of polynomials. Master's thesis, University of Oxford, 2001.

[65] John E. Warnock. The Display of Characters Using Gray Level Sample Arrays. *Computer Graphics (Proceedings of SIGGRAPH 80)*, pages 302–307, 1980.

[66] Lance Williams. Pyramidal parametrics. *Proceedings of SIGGRAPH'83, Computer Graphics*, 17(3):1–11, 1983.

[67] Lance Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, pages 1–11, July 1983.

[68] S. Winner, M. Kelley, B. Pease, B. Rivard, and A. Yen. Hardware accelerated rendering of antialiasing using a modified a-buffer algorithm. In *ACM SIGGRAPH*, pages 307–316, 1997.

[69] Christian Wortmann. Approximation of spatial polygonal curves by G^1 arc splines. Technical report, Universität Dortmund, FB Informatik LS VII, D-44221 Dortmund, Germany, June 2006.

[70] Carsten WŁchter and Alexander Keller. *Instant Ray Tracing: The Bounding Interval Hierarchy*. 2006.

[71] X. Yang. Efficient circular arc interpolation based on active tolerance control. *Computer Aided Design*, 23:1037–1046, 2002.

[72] Xunnian Yang. Approximating NURBS curves by arc splines. In *Geometric Modeling and Processing 2000, Theory and Applications.*, pages 175–183, Hong Kong, China, 2000. ISBN 0-7695-0562-7.