Vectorization of Raster Images Using B-Spline Surfaces

Curtis A. Armstrong
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Computer Sciences Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/763

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
VECTORIZATION OF RASTER IMAGES USING B-SPLINE SURFACES

by

Curtis A. Armstrong

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

July 2006
This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date

Thomas W. Sederberg, Chair

Date

William A. Barrett

Date

Eric K. Ringger
As chair of the candidate's graduate committee, I have read the thesis of Curtis A. Armstrong in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Thomas W. Sederberg
Chair, Graduate Committee

Accepted for the Department

Parris K. Egbert
Graduate Coordinator

Accepted for the College

Thomas W. Sederberg
Associate Dean, College of Physical and Mathematical Sciences
ABSTRACT

VECTORIZATION OF RASTER IMAGES USING B-SPLINE SURFACES

Curtis A. Armstrong

Department of Computer Science

Master of Science

A system has been developed for converting raster images into vector images. Raster images are made of pixels, while vector images are made of smoother shapes. The image is first segmented, and the segments are layered. The boundary of each segment is approximated with a periodic B-Spline curve. This curve is then used to create a B-Spline surface to approximate the interior of the segment. The algorithm fits each B-Spline to the colors of the image using least-squares approximation. The color and shape of each B-Spline surface are extrapolated into regions behind other segments. The result is a vector image made of layered B-Spline surfaces.
Table of Contents

1 Introduction .............................. 1
  1.1 Overview ................................... 4
  1.2 Related Works ................................ 11

2 Segmentation .............................. 19
  2.1 On the Nature of Segments ............... 20
  2.2 Segment Boundaries .......................... 24

3 Layering .................................... 29
  3.1 Layering Using Convexity ..................... 31
  3.2 Layering Using Angles ......................... 34

4 Boundary Approximation .................... 37
  4.1 Approximating Visible Sections .............. 39
  4.2 Approximating Occluded Sections .............. 49
  4.3 Putting the Boundary Together ............... 59

5 Geometry Approximation .................... 61
  5.1 Finding the Corners ......................... 62
  5.2 Initializing the Geometry ..................... 70
  5.3 Adjusting the Geometry ....................... 83
  5.4 Mesh Untangling .............................. 84

6 Color Approximation ....................... 95
  6.1 Choosing Samples ......................... 96
  6.2 Approximating the Color ..................... 99
  6.3 Refining the Color Approximation ............. 102
  6.4 Extrapolating Colors ....................... 104
# Table of Contents

## 7 Conclusion

- 7.1 Results ............................................................................. 109
- 7.2 Future Work ..................................................................... 127

## A B-Splines ............................................................................. 131

## B Least-Squares Approximation ............................................. 145
## List of Figures

1.1 A vector image and a raster image ........................................ 2
1.2 An image for vectorization .................................................. 6
1.3 Image segmentation ......................................................... 6
1.4 Segment layering ............................................................ 7
1.5 Boundary approximation ................................................... 7
1.6 Geometry approximation .................................................... 8
1.7 Color approximation ........................................................ 9
1.8 Vectorization output ......................................................... 9

2.1 Overlapping segments ....................................................... 21
2.2 Segments may not surround other segments ............................. 23
2.3 Segments, segment boundaries, and segment borders ................. 25
2.4 Clockwise segment boundaries ............................................ 26

3.1 Determining the convexity of a border ................................ 33
3.2 Using angles to determine the layers .................................... 34

4.1 Visible and occluded sections of a boundary ............................ 38
4.2 Measuring the distance from a boundary to its approximation .... 41
4.3 Refining the boundary approximation .................................... 45
4.4 Constraints on occluded boundary approximations .................. 50
4.5 A problematic boundary approximation ................................. 52

5.1 Computing the orientation of the second derivative .................. 67
5.2 Choosing the corners ....................................................... 70
5.3 A boundary with corners identified ..................................... 73
5.4 Splitting a periodic B-Spline curve into four pieces .................. 74
5.5 Control grid for surface $\mathbf{S}_0$ ......................................... 77
5.6 Control grid for surface $\mathbf{S}_1$ ........................................... 78
5.7 Surface $\mathbf{S}_0$ after inserting knots to match surface $\mathbf{S}_1$ ....... 79
5.8 Surface $\mathbf{S}_1$ after inserting knots to match surface $\mathbf{S}_0$ ....... 80
A.3 A degree three B-Spline curve ........................................... 134
A.4 A bicubic B-Spline surface ................................................. 140
B.1 Approximating a set of points ............................................. 146
Chapter 1

Introduction

Most computer images are represented in “raster” format. Raster images are common because virtually all scanners and digital cameras produce raster images, and most monitors and printers are designed to display raster images. Raster images are made of small dots called pixels. Each pixel represents the color of the image at a particular point. Usually, the pixels are arranged in a regular rectangular grid.

Although raster images are easy to produce and display, they are not always the best way to represent an image. One disadvantage of raster images is that the quality degrades when performing certain operations on the image. For example, zooming in on a raster image makes the image appear blocky (as illustrated by Figure 1.1). It is possible to reduce this effect using techniques such as bilinear interpolation, but such techniques often blur the image as well.
CHAPTER 1. INTRODUCTION

Another disadvantage of raster images is the difficulty for users to edit images. In order for an image to have good quality, it must be made of thousands or even millions of pixels. It is simply not practical for a user to edit each pixel individually to produce the desired result. Image editing tools are designed to allow the user to edit many pixels at once, but generally speaking, it is difficult to produce powerful, high-level raster-based editing tools that are easy to use. Simply providing ways for the user to intelligently and easily select which pixels to use in a specific operation is a non-trivial task.

Raster images also take a significant amount of storage space. Considering the
image may contain millions of pixels, and each pixel is usually represented using up to 64 bits, it takes a significant amount of space to represent an image. There are a number of ways to compress an image based on certain assumptions about the variation from pixel to pixel or by degrading the quality in a way that is difficult for humans to notice. It is difficult to develop a compression algorithm that produces a high quality image and a high compression ratio.

An alternate image format is “vector” format. While raster images are made of pixels, vector images are made of smoother shapes, such as lines, polygons, and curves. Raster images are a discrete representation of a function, but vector images define a continuous function.

Because a vector image is made of shapes, it can be moved, rotated, or scaled without any ill effect by simply moving, rotating, or scaling the shapes. Figure 1.1 shows the difference between scaling an image in both raster and vector format.

Vector images are usually easier to edit than raster images since the user is working on a more logical level (shapes instead of pixels). They are frequently composed of a manageable number of shapes as compared to the millions of pixels in a raster image. Those shapes can be moved, rotated, or otherwise altered with relative ease. Vector images often occupy less storage space, and the space needed depends on the image complexity rather than the image size and resolution.

Because of the advantages of vector images, it is useful to be able to convert a
raster image into a vector image. The process of converting a raster image into a vector image is called vectorization.

Although converting a vector image into a raster image is usually straightforward, it is not as easy to convert a raster image into a vector image. One reason for the difficulty is that there may be many different vector images corresponding to the same raster image. For example, the sailboat in Figure 1.1 could be represented using a separate shape for each of the sails, masts, ropes, and hull. Alternately, it could be represented using a single, complex shape for the entire ship. Either representation could produce the exact same raster image.

We have developed a vectorization process that converts raster images into vector images. The process involves approximating the image with a set of B-Spline surfaces. B-Spline surfaces are commonly used in computer graphics because they can represent a variety of shapes in a manner that is easy for the user to edit. Appendix A provides an introduction to B-Splines for those who are unfamiliar with them.

1.1 Overview

This section provides an overview of the vectorization algorithm. The details of each step are explained in later chapters.

The input image may have been acquired from a number of sources, such as a digital camera or a scanner. Our vectorization system is designed to work on
photographs or similar images rather than line drawings or scanned documents. The image is vectorized by first dividing it up into segments and then fitting a B-Spline surface to each segment. The resulting vector image is made of multiple B-Spline surfaces that define the color as well as the boundary of each object.

There are five steps in the vectorization system:

1. Segmentation (Chapter 2),

2. Layering (Chapter 3),

3. Boundary approximation (Chapter 4),

4. Geometry approximation (Chapter 5), and

5. Color approximation (Chapter 6)

We start with an image such as the one in Figure 1.2. This image was originally a vector image, so it should be easy to vectorize. In the first step, the image is segmented, which means it is divided into different regions to be approximated separately. Figure 1.3 shows a possible segmentation of Figure 1.2.

After segmentation, the segments are layered to determine which segments are in front of others, as illustrated by Figure 1.4.

Step 3 approximates the boundary of each segment with a B-Spline curve. Figure 1.5 shows the approximation of the boundary of one of the segments from Figure 1.3.
Figure 1.2: An image for vectorization.

Figure 1.3: A possible segmentation of the image in Figure 1.2. A thin blue line outlines each segment.
Figure 1.4: The segments in Figure 1.3 have been placed in layers. Dark colors indicate background layers while lighter colors indicate foreground layers.

Figure 1.5: The boundary approximation of one of the segments in Figure 1.3.
CHAPTER 1. INTRODUCTION

Figure 1.6: The geometry approximation of the segment in Figure 1.5.

The B-Spline curve that forms the boundary approximation is then converted into a B-Spline surface in step 4, “geometry approximation.” This step involves determining the location of each of the control points for the B-Spline surface. Figure 1.6 shows such a B-Spline surface and its control grid.

Once we have a B-Spline surface for the segment, we assign colors to each of the control points, resulting in a colored B-Spline surface that approximates the segment, as illustrated by Figure 1.7. The final approximation of Figure 1.2 is shown in Figure 1.8.

This research is primarily focused on the last three steps, which perform the actual vectorization. The segmentation and layering steps are discussed since their
1.1. OVERVIEW

Figure 1.7: This shows the result of approximating the color of a segment.

Figure 1.8: The final vector image.
output is needed for later processing, but they are not the primary focus of this research. Both can be omitted, but including them yields better results.

Normally, these steps are executed in this order, but there are a few situations when it may be best to repeat a step after performing later steps. For example, the color approximation sometimes changes the knot vector for a B-Spline. Since the knot vector of a B-Spline affects its shape, it may be beneficial to repeat the boundary and geometry approximation using the new knot vector. It is also possible to skip some of the steps in the vectorization process. An interesting example is the situation where the image is not segmented, but rather approximated as a single segment.

It would be best to have a vectorization system that would take any image as input and automatically convert it to a vector image without any user interaction. Unfortunately, that is a very difficult problem. It is our goal to make the system reasonably easy to use, while requiring some user interaction during the vectorization process. Although minimal user interaction is usually desired, involving the user in the vectorization process can be an advantage since it gives the user greater control over the outcome. The user presumably will want to edit the image after it has been vectorized, so the parts of vectorization that allow user interaction permit the user to adjust the outcome according to personal style. This interaction is retroactive when possible: the algorithm does what it can automatically, then allows the user to adjust the result before moving on to the next step.
Each step of vectorization requires various parameters from the user. The chosen parameters can affect the outcome considerably. Most steps have default parameters which work well on many images, reducing the need for user interaction. If the default parameters produce unsatisfactory results, the user can adjust the parameters and try again. It is also usually possible to manually adjust the result when minor changes are desired.

A few steps may have multiple approaches (segmentation is one example). If there are several alternate solutions to the problem, the user may choose which method to use. The default approach should solve the problem without further input from the user. Again, the user has the ability to go back and try a different approach if the default approach does not work well.

In designing this vectorization system, a major goal is to produce a result that not only looks good, but also has as few control points as possible. A vector image with few control points is easier to edit and uses less memory than more complicated images.

1.2 Related Works

The first vectorization systems were designed to convert black-and-white images consisting primarily of thin lines into vector images in which the basic shapes are line segments. This type of image, commonly called a “line drawing,” is frequently used
for figures in engineering, science, mathematics, or other fields. Line drawings may contain curves in addition to lines.

Examples of such vectorization systems include early research by Freeman [6] and some later work by Chiang [3], Dori and Liu [5], and Jennings and Parker [11]. Song et al. describe two different line drawing vectorization schemes in 2000 and 2002 [21, 22]. These systems primarily focus on vectorizing engineering drawings. Boatto et al. created a system for vectorizing maps that makes use of line drawing vectorization schemes [1]. Tanaka, Kamimura, and Shashkov present another map vectorization scheme that focuses mainly on the line drawing portions of the map [23].

The process of converting a line drawing from raster to vector form is called vectorization because the line segments produced by early systems are represented as vectors relative to the endpoint of the previous line. Since that time, the terms vectorization and vector image have become more general to include other types of images.

This type of vectorization consists of fitting a line to groups of neighboring black pixels that are in a straight line. Because of the discrete nature of the raster image, the pixels form a straight line only in very unusual circumstances. Normally, they are simply *almost* in a straight line, and some method must approximate the line from the pixel locations. Additionally, it can be difficult to determine whether the
1.2. RELATED WORKS

variations from one pixel to the next are because of the discrete nature of the line, or because the pixels belong to different lines.

It is possible to represent a curve as a series of short, connected line segments, but a smoother result is achieved by including curves in the vectorization. While curves are more difficult to vectorize than lines, the approach is still the same: identify pixels that belong together, and fit a curve to them. Enforcing continuity between curves is also challenging. It can be difficult to determine the degree of continuity at a transition, and it can be difficult to pick curves that are continuous to the desired degree and approximate the pixels accurately.

The type of curve used varies from one vectorization system to the next; many systems use circular arcs, conic sections, Bézier curves, or B-Splines to represent curves. Hori and Tanigawa [10], and Rööśli and Monagan [19] use straight line segments and circular arcs. Nagasamy and Lagrana approximate curves using conic sections [16]. Chang and Yan use Bézier curves to approximate the drawing [2].

There are several factors to consider when choosing the representation of curves in the vector image. In order to allow the greatest flexibility, the curves should be able to represent many kinds of objects. For this reason, circular arcs are not a good choice.

One should also choose curves that are easy to edit. Most shapes are defined using a set of control points. The control points should at least lie near the curve, and it is
better for the control points to lie on the curve. Moving the control points changes the shape, and the resulting change should be predictable. Generally, a curve is easier to edit if it has only a few control points, so a vector image should be defined with as few control points as possible.

B-Spline curves and surfaces have many good properties for the basic shape in a vector image. For those unfamiliar with B-Splines, there is a more detailed description in Appendix A. B-Splines are made of a set of control points that lie near the curve but generally do not lie on the curve. B-Splines are easy to edit because moving the control points changes the curve predictably, and moving a particular control point only changes the portion of the curve near that point. The mathematics of B-Splines allow various degrees of continuity to produce smooth shapes or sharp corners.

It is important to consider the continuity of the curves because continuity affects how smooth the image appears. A good vectorization algorithm enforces the continuity constraints specified by the user so that control points can be moved without violating those constraints. The simplest form of continuity is called $G^0$ or $C^0$ continuity, which means the curves touch at the endpoints. Other kinds of continuity are divided into two groups: geometric continuity, which is represented with the letter G, and parametric continuity, which is represented with the letter C.

Geometric continuity comes in various degrees. The first is $G^0$ continuity, as already mentioned. The next degree is $G^1$ continuity, which means the curves touch at
the endpoints and the curves approach that point from exactly the opposite direction. This makes a smooth transition from one curve to the next. The next degree is $G^2$ continuity, which includes both $G^0$ and $G^1$ continuity, and additionally requires the curves to have the same curvature at the point of intersection, and that they curve in the same direction. This causes an even smoother transition from one curve to the next.

Parametric continuity only applies to curves that can be represented using parametric equations. In general $C^n$ curvature means that the $n^{th}$ derivative of the parametric equations is the same for both curves at the point where they meet. All smaller derivatives are also the same since $C^n$ continuity includes $C^{n-1}$ continuity. Usually, $C^n$ continuity implies $G^n$ continuity, except in degenerate situations.

If we introduce color (or gray scale) into a line image, we can still use the basic line image vectorization described above with minor modifications. First, the background color must be chosen. For a line drawing, this is normally the most common color in the image. The lines (or curves) can be identified as groups of pixels that are a different color than the background. This may be difficult if the image is noisy (there are small random variations in color). Once the pixels are identified, they are approximated with a line or curve, and a color is chosen to match the color of the pixels.

Black and white pictures often contain large regions that are all black, violating
the definition of a line drawing. To vectorize such images, a region-based vector-
ization system is used. Such a system is similar to line-drawing vectorization, but
approximates region boundaries with lines and curves.

Kasturi et al. describe a system for vectorizing line drawings that also allows the
drawings to contain solid binary regions, whose outlines are approximated, so this
vectorization scheme fits into both the region and line vectorization categories [13].
Likewise, Nagasamy and Lagrana vectorize engineering drawings that may contain
large solid objects by approximating the boundaries of the solid objects [16]. Jimenez
and Navalon use a region-based vectorization scheme, but the result is a set of polygon
boundaries, not smoother curves [12]. Mikheev, Vincent, and Faber propose another
system for vectorization based on region boundaries, but it is only designed to ap-
proximate polygons, not smooth curves [14]. Yang, Lu, and Lee use Bézier curves to
approximate the outlines of Chinese calligraphy characters [25].

A region-based vectorization system can also be extended to include color by
requiring all pixels within a region to be the same color. First, the image is divided
into sets of consecutive pixels that have the same color. Next, the boundary of each of
those regions is approximated with appropriate curves. Finally, the shape is assigned
a color based on the color of the pixels.

Although filling each shape with a single color is common, real objects are seldom
filled with a single color. For example, a picture taken with a digital camera usually
contains objects with varying color, so it is better to vectorize those objects using shapes that vary in color. It is harder to determine which regions belong together when they are not uniform in color. The gradient color of each region must be approximated instead of simply assigning a single color to the entire region.

Ge describes a system to approximate images using a set of triangles [8]. Although the intended use of this triangulation system is image morphing, it could easily be seen as a region-based vectorization system. Its primary advantage over other vectorization systems is that it approximates gradient colors. However, it still has many shortcomings when used for vectorization. One is that both the region boundaries and the color are approximated using linear shapes, so the result is by no means smooth. This also means that in order to approximate curves and regions that are not linearly shaded, the algorithm produces an unnecessarily large number of control points (triangle vertices), making the image difficult to edit. The result is also not well suited for user editing because it fails to segment the image into objects, nor does it layer those objects. If the user wishes to move an object, it is necessary to group all the triangles that belong to the object and manually set the layer.

Another vectorization algorithm is being developed by Price [18]. It approximates regions with Bézier patches and then refines the approximation by subdividing the patches. This is repeated until the approximation is sufficient within a user-defined threshold.
There are currently many commercial vectorization products. For example, the Adobe Streamline program converts drawings into a vector format based on Bézier curves. In general, these commercial products are geared toward vectorizing line drawings, regions of similar color, or text. Their documentation usually suggests they not be used on actual photographs because of the difficulty in vectorizing photographs.
Chapter 2

Segmentation

Segmentation is the process of dividing an object into various regions called segments. Each segment usually corresponds to a single object in the picture. Each segment should consist of a contiguous set of pixels whose colors vary smoothly. A single object that contains several vastly different parts may vectorize better if divided into separate segments. The purpose of this work is not to discuss how to segment an image; the vectorization system could use any segmentation algorithm that produces acceptable results. Our definition of acceptable is discussed in Section 2.1.

The literature presents many viable segmentation methods. We have implemented the “intelligent scissors” algorithm [15], which produces reasonable results. Intelligent scissors is an edge-based segmentation method, meaning it segments the image
by finding strong borders between segments. An alternative is region-based segmentation, which identifies homogeneous regions.

Intelligent scissors requires some guidance from the user. The user specifies a few points near the boundary of a segment, but not necessarily on the boundary. The algorithm computes the least-cost path between these points using a variation of Dijkstra’s algorithm. The “cost” is defined so that it is lower near strong edges; the Sobel edge filter and Laplacian zero crossings work well as cost factors.

The lowest-cost path between the starting location and any other pixel in the image can be computed in a reasonable amount of time, and then as the user moves the mouse, the computer can retrieve the path to that point very quickly and display it. In this way, the user can interactively preview where the path would be without having to commit to any particular location. This gives the user ultimate control over the location of the path.

2.1 On the Nature of Segments

There are a few constraints on segmentation. First, each pixel in the image must be in exactly one segment. There are two parts to this constraint: each pixel must be in at most one segment, and each pixel must be in at least one segment.

The constraint that each pixel must be in at most one segment guarantees that the segments do not overlap and provides a mechanism for looking up the segment
2.1. ON THE NATURE OF SEGMENTS

Given the pixel. (Section 4.2 describes a way that the approximations of segments can overlap, but that is a different issue.) Figure 2.1 shows an example of an image with overlapping segments and a correction to the problem.

It is reasonable to develop a system in which the overlap constraint is violated, especially if the resulting image can have transparency, such as a color space that includes an alpha channel. For example, if one object is partially transparent, it is possible to see through to the object(s) behind it. In this situation, the segments overlap, and some pixels are located in both segments. If this happens, it is important to compute the “original” colors for the foreground and background pixels, and to determine the alpha values for the foreground pixels.
Each pixel is required to be in at least one segment to prevent holes in the resulting vector image. The only reason one may want pixels that are not in any segment is if those pixels are to be part of the background, which is often not vectorized. Chapter 3 describes a different way to treat the background separate from the foreground. That mechanism simply requires separate segments for the background.

Another constraint is that all pixels in a segment must be contiguous, preventing a single segment from being separated into multiple disjoint pieces. It is obvious that an object may actually appear disjoint in an image since part of that object could be covered by another object. This is a source of ambiguity in any image because two segments that appear similar may not really be the same object at all. Human vision uses higher levels of reasoning to guess whether the regions are the same object, but this still does not completely solve the ambiguity. For simplicity, we assume that a segment is not allowed to be disjoint, but the concepts of approximating occluded boundaries (Section 4.2) and extrapolating colors outside the segment boundary (Section 6.4) could be extended to work for disjoint segments as well.

Another problem that arises from occlusion is if an object is located in front of another object in a way that the background object completely surrounds the foreground object in the image. Picture a hot air balloon against a solid blue sky. An example is Figure 2.2 with Region 1 in the foreground, and Region 2 in the background. Once again, it would be best to consider the occluded object as a single
2.1. ON THE NATURE OF SEGMENTS

Figure 2.2: No segment should completely surround another segment, as in the image on the left. The outside segment must be divided in two to overcome this problem.

A more serious problem arises if the segment is allowed to have a hole in it, so that objects behind the segment are visible. For example, suppose Region 1 in Figure 2.2 is the background and Region 2 is the foreground. If the segment has a single hole, this causes a problem for the vectorization because B-Splines do not in general have holes in them. There is a way to construct a B-Spline surface with exactly one hole by making it a periodic B-Spline, but that causes other complications in the geometry.
approximation (Chapter 5) because the knot vectors of the inner boundary and the outer boundary must be matched in a way that prevents the spline from twisting.

A periodic B-Spline is only allowed to have a single hole. Segments with more than one hole become even more complicated to approximate. The multiple hole problem could be solved with trimming curves around each hole. It could also be solved by constraining one part of the inner edge of a periodic B-Spline to be continuous with the opposite part of the edge. We have not yet researched algorithms to approximate segments with multiple holes.

2.2 Segment Boundaries

An important attribute of each segment is its boundary. The boundary surrounds the segment and divides the region inside the segment from the region outside the segment. Because segments consist of a set of pixels, segment boundaries are made of pixel edges, the short lines between neighboring pixels. The meaning of these terms is emphasized in Figure 2.3.

It is sufficient to represent a segment using its boundary, since a simple procedure can determine the pixels inside a boundary. It is also sufficient to represent a segment boundary using the set of pixels in the segment, since it is fairly easy to trace around a segment to compute its boundary. In practice, both representations are used since the boundary representation is useful for boundary approximation (Chapter 4), and
the interior representation is useful for color approximation (Chapter 6).

The boundary of a segment consists of one continuous loop of pixel edges. The boundary of a segment with holes consists of more than one loop. It is helpful to have all boundaries traced in the same direction, so in this paper, all boundaries are traced in a clockwise direction, as in Figure 2.4. However, the portion of a boundary that defines a hole is counter-clockwise. This way, if you imagine yourself walking along the boundary of a segment, the “inside” of the segment is always to your right, while the “outside” stays on your left.

The term “boundary” can be confused with the term “border.” A border is a portion of the boundary that divides the segment from one of its neighboring segments.
Figure 2.4: The boundary of a segment is traced clockwise (green), except around a hole in the segment (blue). The two segments along a border trace the border in opposite directions (red), since it is traced clockwise relative to a different segment.
Figure 2.3 clarifies the definition of these two terms. By this definition, the combination of all the borders around a segment is its boundary. Notice that the boundary of the neighboring segment is traced in the opposite direction along a border, as shown in Figure 2.4. If two segments border in two distinct locations, it is considered two separate borders.
Chapter 3

Layering

Once an image has been segmented, layering orders the segments from back to front. Layering is optional; it is possible to vectorize an image without layering it. However, vectorizing a layered image usually produces better results. There are a few reasons for this. First, the segments can be vectorized independently without requiring that the boundary of one B-Spline correspond exactly with the boundary of another. Second, allowing the resulting B-Splines to overlap can lead to fewer control points when approximating occluded boundaries. Not using layering would require that both segments along a border approximate that portion of their boundaries with equal detail, while having one segment in front of the other allows the occluded boundary approximation to be simpler, since it is behind the other segment. Section 4.2 gives more detail about how to approximate these occluded boundary sections.
The layering of an image is defined using a relation among the segments of an image. Each segment can be compared to any other segment to determine if it is “in front of,” “behind,” or “in the same layer” as the other segment. For simplicity, we do not allow cycles in these relations: if segment A is in front of segment B, and segment B is in front of segment C, segment C cannot be in front of segment A. Such cycles are possible in real images, as illustrated by Figure 7.8 in the Results section, but permitting cyclical layering would make boundary approximation more difficult.

This definition of layering requires that every segment must be layered relative to every other segment in the image. It may seem that the relative layer is only important for segments that are neighbors, but it is necessary to define the layering relative to all other segments because the approximation of a segment may well extend beyond the space occupied by its immediate neighbors. Allowing the approximations to overlap is, after all, one of the purposes of using layers.

From this definition, it is not necessary that each segment be in a separate layer than all other segments in the image. Segments that are in the same layer are allowed, but it is often desired that neighboring segments be in different layers. If neighboring segments are in the same layer, there are usually holes in the resulting vectorization.

The easiest way to achieve layering is to assign each segment an integer that defines its layer. Smaller numbers are assigned to background layers, and larger numbers are assigned to foreground layers. This way, the relative layering of two segments is
determined by comparing these numbers; a segment is in front of another segment if the number of its layer is greater. The background layer is normally zero. It is not vectorized unless the user chooses to vectorize it.

One way to layer an image would be to have the user meticulously assign a layer to each segment individually. Although easy to program, this is more difficult for the user, especially if an image consists of a large number of segments. Clearly, a better method would produce good layers automatically (or nearly automatically) and then allow the user to correct anything if necessary. Unfortunately, it is not easy to produce such an automatic layering algorithm. As demonstrated in the conclusion, Section 7.1, our solution to automatic layering does not work well. Implementing good automatic layering has not been a significant priority in our research from the start, so we have not spent as much time trying to overcome the automatic layering problem as we have spent on other issues. Future work will need to research automatic layering more completely.

### 3.1 Layering Using Convexity

We have explored a few approaches to automatic layering. The first is based on convexity. If we assume that most objects in the world are convex, it is possible to compute the layering based on convexity. Although this may seem unreasonable, it is also true that concave segments must have at least some convex portions along
their boundaries, so assuming that most sections of the boundary are convex is more reasonable. In practice, however, the common occurrence of concave boundaries is one reason the automatic layering algorithm we implemented frequently fails.

To determine layering using convexity, the border between two patches is determined to be convex or concave, and then the segment that has the convex border is layered in front of the segment with the concave border. In practice, it is difficult to determine if a border is convex or concave since the border is made of pixel edges. Segment borders tend to zig-zag along pixel edges, creating many concave intersections even if the overall trend is convex. Additionally, it is unclear how to layer two segments if the border between them is S-shaped, which happens occasionally because not all objects are convex.

One solution to this problem is to draw a line from one end of the border to the other, and count the number of pixel edges that are on either side of the line, as illustrated in Figure 3.1. The “concave” side of the line is the side that has more pixel edges. Edges that are nearly on the line are ignored so that the convexity measurement is not biased by the discrete nature of pixel edges that are nearly on the line.

Determining the layer based on convexity presents some problems, particularly since many objects are concave. A single concave object can ruin the entire layering algorithm if encountered too soon.
Figure 3.1: The convexity of a border is determined by the number of pixel edges on either side of a line. In this example, the upper-left segment is concave along the border because more pixel edges are on that side of the line. The lower-right segment is convex along the same border. The pixel edges in the shaded area are ignored.
3.2 Layering Using Angles

An alternate layering mechanism looks at the points where three segments meet, rather than the convexity of the border. Assuming objects have smooth borders, they meet in T-shaped intersections. The segment along the flat side of the T is in front of the other two. There is no way to determine a relation between the other two based only on this information.
Again, because of the discrete nature of the boundaries, it is difficult to define a measure that can determine which of the three segments is in front. One solution is to look at the relative angles, as illustrated in Figure 3.2. The angles are measured by counting a small number of edges away from the intersection in each of the three directions. A line is drawn from the intersection point to each of those three points as a way to determine the general direction that border approaches the intersection. The angles between these three lines are computed, and the largest angle is taken to correspond to the segment that is in front. This also provides a way to measure how certain that assumption is: if the angle is close to 120 degrees, it is not very likely that we can use this junction to determine the layers. On the other hand, if the angle is 180 degrees or greater, it is quite likely that the segment we determined to be in front really is.

Another solution looks at the continuity of the three borders at that point. The borders that belong to the segment in front should be (nearly) $C^1$ continuous, while the borders that belong to the other segments should be only $C^0$ continuous. Again, we must use a discrete approximation to determine continuity.

All these methods occasionally produce incorrect layering, often causing cycles. For example, there may have been a concave border on a foreground object, or an object may have a sharp corner near a T-junction, causing the T to appear the wrong way. We must have a way to eliminate these cycles.
A naïve approach would be to simply remember past decisions, and any new decision is rejected if it would cause a cycle. This is unfortunate if the bad decision is made early on, since it could ruin many later decisions. Future research will need to find a better solution.
Chapter 4

Boundary Approximation

The next step in the vectorization process is to approximate each segment boundary with a B-Spline curve. The boundary approximation has two parts. The first part pertains to approximating the sections of the boundary that border segments behind a given segment. We call these the “visible sections” of the boundary because they are not covered by the neighboring segments. The second part of boundary approximation pertains to the sections of the boundary that border segments in front of it. These sections are called “occluded sections.”

The boundary is first divided into visible sections and occluded sections, as illustrated in Figure 4.1. These sections alternate around the boundary: first a visible section, then an occluded section, then a visible section, and so on until the last occluded section. If two consecutive borders are both visible, they are part of the same
Figure 4.1: The sections of the boundary highlighted in green are visible, while the purple sections are occluded. This boundary has two visible sections and two occluded sections. The numbers indicate the layer of each segment.
4.1. APPROXIMATING VISIBLE SECTIONS

visible section. There are two cases that must be handled specially: boundaries that are entirely visible, and boundaries that are entirely occluded.

4.1 Approximating Visible Sections

Approximating the visible sections of the boundary consists of choosing a B-Spline curve that passes near the center of each pixel edge on the boundary. There are many algorithms in the literature that describe ways to fit a curve to a set of data points [9, 17, 20]. Our algorithm uses least squares approximation to fit a curve to the boundary.

Each visible section of the boundary is approximated with a single non-periodic B-Spline curve, unless the boundary is entirely visible. That special case uses a periodic B-Spline curve, as described at the end of this section. The non-periodic B-Splines that approximate the visible sections are later combined with the occluded sections to form a single periodic B-Spline, as described in Section 4.3.

We start the approximation by choosing an initial knot vector. It is best to start with a simple knot vector such as

\[ [0, 0, 0, 0, n - 1, n - 1, n - 1, n - 1] \] (4.1)

where \( n \) is the number of pixel edges in the boundary section. It is also necessary to pick a parameter value \( t_i \) to associate with each pixel edge along the boundary. The
The easiest solution is to let the parameter value be the index of the edge.

\[ t_i = i \]  

(4.2)

The index of the first edge is zero, and the last edge is \( n - 1 \), which is why the knot vector ends at \( n - 1 \) instead of \( n \).

Our goal is to minimize the distance between the curve and the boundary. It is too difficult to minimize the distance to the nearest point on the curve, so instead we minimize the distance to the curve evaluated at \( t_i \). Recall that a B-Spline curve is defined using the parametric equations

\[
x(t) = \sum_{j=0}^{m-1} p_{xj} B_j^k(t) \]

(4.3)

\[
y(t) = \sum_{j=0}^{m-1} p_{yj} B_j^k(t) \]

(4.4)

where the control points are \((p_{xj}, p_{yj})\) and the B-Spline basis functions are \( B_j^k(t) \).

This means we can compute the distance from the curve to edge \( i \) using the formula

\[
d_i = \sqrt{ \left[ x_i - \sum_{j=0}^{m-1} p_{xj} B_j^k(t_i) \right]^2 + \left[ y_i - \sum_{j=0}^{m-1} p_{yj} B_j^k(t_i) \right]^2 } \]

(4.5)

The distance \( d_i \) depends only on the position of the control points, since \( x_i, y_i \), and \( t_i \) are all known. In practice, we never deal directly with \( d_i \) but rather the square distance \( d_i^2 \); the least-squares approximation minimizes the sum of the squares of the distances from each edge to the curve. The coordinates of the center of each pixel edge along the boundary section are used for \((x_i, y_i)\), as shown on the right in Figure 4.2.
4.1. APPROXIMATING VISIBLE SECTIONS

Figure 4.2: The boundary approximation tries to minimize the sum of the squares of the distances \( d_i \) from the center \((x_i, y_i)\) of each pixel edge to the corresponding points \( t_i \) on the approximation curve. In practice, the resulting approximation would likely be much closer to the boundary (unless it is only a piece of a much larger boundary), but this image shows it to the side to illustrate how the distances are measured.
The center of a pixel edge always has one integer coordinate and one coordinate halfway between integers (for example, (1.5, 2) or (183, 237.5)). Figure 4.2 illustrates how to measure the distances from the boundary to its approximation curve.

We can now use least-squares approximation to pick the control points \((p_{xj}, p_{yj})\) that produce a curve close to the boundary. The least-squares approximation is computed by solving the matrix equation

\[
A^TAX = A^TV \tag{4.6}
\]

The matrix \(A\) has one row for each pixel edge, and each column contains one of the basis functions, evaluated at the \(t_i\) value for that pixel edge:

\[
A = \begin{bmatrix}
B_0^k(t_0) & B_1^k(t_0) & \cdots & B_{m-1}^k(t_0) \\
B_0^k(t_1) & B_1^k(t_1) & \cdots & B_{m-1}^k(t_1) \\
\vdots & \vdots & \ddots & \vdots \\
B_0^k(t_{n-1}) & B_1^k(t_{n-1}) & \cdots & B_{m-1}^k(t_{n-1})
\end{bmatrix} \tag{4.7}
\]

The matrix \(V\) also contains one row per pixel edge. The columns of \(V\) are the coordinates of the middle of the pixel edge:

\[
V = \begin{bmatrix}
x_0 & y_0 \\
x_1 & y_1 \\
\vdots & \vdots \\
x_{n-1} & y_{n-1}
\end{bmatrix} \tag{4.8}
\]
4.1. APPROXIMATING VISIBLE SECTIONS

Solving Equation 4.6 for $X$ yields the control points that produce the best-fitting curve:

$$X = \begin{bmatrix}
  p_{x0} & p_{y0} \\
  p_{x1} & p_{y1} \\
  \vdots & \vdots \\
  p_{x(m-1)} & p_{y(m-1)}
\end{bmatrix}$$  \hspace{1cm} (4.9)

Since $n$ is usually much larger than $m$, it is best to save memory by computing $A^T A$ and $A^T V$ iteratively rather than allocating space for the matrices $A$ and $V$. For each edge, compute the vector $a_i$, which consists of the basis functions evaluated at $t_i$, and the vector $v_i$, which is the coordinates of the center of the edge. Now, compute the matrices $a_i a_i^T$ and $a_i v_i^T$. It is easy to determine that

$$A^T A = \sum_{i=0}^{n-1} a_i a_i^T$$  \hspace{1cm} (4.10)$$

$$A^T V = \sum_{i=0}^{n-1} a_i v_i^T$$  \hspace{1cm} (4.11)

We have computed $A^T A$ and $A^T V$ without ever computing $A$ or $V$.

Once a best-fit curve is found for the initial knot vector, the error is computed. We investigated three methods to determine the error. One method is the sum of the squares of the distances from the pixel edges to the curve ($d_i$ in Equation 4.5).

$$\epsilon_{\text{sum}} = \sum_{i=0}^{n-1} d_i^2$$  \hspace{1cm} (4.12)
This error is called the total square error. An alternate way to compute the error is to average the squares of the distances.

\[ \epsilon_{\text{mean}} = \frac{\sum_{i=0}^{n-1} d_i^2}{n} \]  

(4.13)

This is called the mean square error. Finally, the error could also be computed by finding the point on the boundary that is farthest away from the curve.

\[ \epsilon_{\text{max}} = \max_{i=0}^{n-1} d_i^2 \]  

(4.14)

This is called the maximum error. Any of these three methods may be used to compute the error, but using the maximum error usually leads to better results.

Once the error is computed, it is compared to an error threshold that can be adjusted by the user, depending on how accurate an approximation is desired. A good threshold for the maximum error is in the range 0.5 to 1 pixel, which corresponds to \( d_i^2 \) values of 0.25 to 1. If the error is less than or equal to the error threshold, we have computed a good approximation to the boundary.

If the error is greater than the threshold, an iterative method is used to refine the curve in an attempt to improve the approximation. Knots are inserted into the knot vector so the curve can more closely approximate the boundary, as illustrated in Figure 4.3. At each iteration, we find the edge that is farthest from the curve, as determined by \( d_i \) in Equation 4.5. That point would benefit the most from a knot. The new knot is the \( t_i \) value for that edge.
4.1. APPROXIMATING VISIBLE SECTIONS

Figure 4.3: The boundary approximation is refined by (a) identifying the point where the error is greatest, (b) inserting a knot at that point, and (c) repeating the least-squares approximation. This repeats until the error is less than a certain threshold.
Unfortunately, it is not always possible to insert a knot at that point. Specifically, if there are already one or more knots at that point, inserting another knot decreases the continuity of the curve at that point. Decreasing the continuity may lead to a better approximation, but the continuity cannot be less than the minimum specified by the user. The default minimum continuity is $C^2$, so unless the user chooses a lower continuity any duplicate knots are not allowed.

In addition to the continuity constraint, it is best to require a minimum knot interval. The knot interval is the difference between adjacent knots. This is necessary because small knot intervals can cause the curve to change directions very quickly, giving the appearance of a discontinuity. A good value for the minimum knot interval is 1, but the user is allowed to change this amount.

If inserting a knot at $t_i$ would cause a violation of the minimum continuity or minimum knot interval, a knot is inserted near $t_i$ according to the following algorithm. The knot immediately before $t_i$ is called $t_b$ and the knot immediately after $t_i$ is called $t_a$.

1. Determine the greater of $t_i - t_b$ and $t_a - t_i$.

2. Divide that range in half by computing $(t_i + t_b) / 2$ or $(t_i + t_a) / 2$.

3. Insert a knot at that point, unless it would also cause a violation of the minimum knot interval.
4.1. APPROXIMATING VISIBLE SECTIONS

4. If it is not possible to insert a knot there, move to the knot intervals before \( t_b \) and after \( t_a \), identify which of those two is greater, and try splitting it in half.

5. Continue this process by moving progressively farther from \( t_i \) until a suitable knot interval to split in half is found.

Once a new knot has been inserted, the best-fit curve is found using least-squares just as before. This process continues until the error decreases below the threshold.

Inserting a knot never causes the total square error to increase because it is possible to represent the curve that produced the previous error using the new knot vector. Because least-squares approximation finds the minimum of this error, the error after each iteration is at most the same as the previous error. This means that if the error threshold is set using the total error, the algorithm is guaranteed to terminate (unless the minimum knot interval is so high that no intervals can be cut in half). Unfortunately, no similar analysis can be made for the maximum error, but in general, inserting a knot decreases the maximum error as well.

Another clue that the algorithm is guaranteed to terminate is that it is possible to determine a curve that exactly passes through the center of each pixel edge, if there is a knot corresponding to each of the \( t_i \) values. Using more knots than that causes problems. If there are more knots than pixel edges (in other words, \( m > n \)), the least-squares approximation fails because the linear system described in Equation 4.6
is unsolvable, since the determinant of $A^T A$ is zero. This can also happen if $m \leq n$ for certain knot vectors.

Imposing a good minimum knot interval helps keep the least-squares problem solvable. Another solution is to increase $n$ by using more samples along the boundary. For example, use the pixel corners in addition to the centers of the pixel edges. However, it is important to remember that increasing the number of samples usually increases the error, especially if pixel corners are used since the corners zig-zag much more than the centers of the pixel edges.

In addition to inserting knots, it is also possible to get a better fit by adjusting the values of $t_i$ for each point along the curve. In approximation theory, a popular way to adjust $t_i$ uses Newton’s root-finding algorithm:

$$
t_i \leftarrow t_i - \frac{f(t_i)}{f'(t_i)}
$$

(4.15)

where $f(t_i)$ is based on the distance from a point to the curve, $d_i$ in Equation 4.5 [17, 20]. We have not explored the effectiveness of including this in our implementation.

Boundary approximation appears to work well without adjusting the $t_i$ values.

If the boundary is entirely visible (because the segment is in front of all its neighbors), there must be some adjustments to the boundary approximation algorithm described above. The major change is using a periodic B-Spline curve for the approximation in order to produce a loop that approximates all edges in the boundary. The least-squares approximation described above still works as long as the basis functions
4.2. APPROXIMATING OCCLUDED SECTIONS

in $A$ reflect the change to a periodic B-Spline curve. The iterative refinement of an entirely visible boundary inserts knots near the point of maximum error, just as approximating the visible sections of a partially occluded boundary.

However, the initial knot vector of an entirely visible boundary is determined in a much different method. The knot vector of a cubic periodic B-Spline must have at least five knots (in other words, at least four knot intervals). The value of these knots is not as obvious as before, since the boundary represented as a loop has no clear start or end. As it turns out, the best values for the initial knot vector for such a boundary follows the same principles as finding the corners for the geometry approximation. For a description of the corner-finding mechanism, see Section 5.1.

4.2 Approximating Occluded Sections

Approximating the occluded sections of the boundary is considerably more difficult than approximating the visible sections because we really do not know where the hidden sections of the boundary are. With layering, an occluded section of the boundary indicates that the “real” boundary of the object is not known because another object is covering it. What we see as the segment boundary is really the boundary of the occluding segment.

Although it would be possible to approximate occluded sections the same way as the visible sections of the boundary, that leads to bad results. One reason is that
Figure 4.4: An occluded boundary approximation must stay within areas that are layered in front of the segment being approximated. In this figure, those regions are blue. We are approximating the yellow segment. The green parts of the approximation are exposed and the purple parts are hidden.

the resulting approximation would have many control points, probably more than necessary if the goal is to find an approximation that merely stays somewhere behind the occluding segment. Remember we have a goal of minimizing the number of control points. Another problem is that the least-squares approximation used before simply finds a curve that is close to the boundary without guaranteeing that the approximating curve stay on one side of the boundary. Because of this, approximating an occluded boundary using least-squares frequently causes gaps in the resulting vector image, revealing portions of the background layers that should remain hidden. Clearly, a better solution must be found.
4.2. APPROXIMATING OCCLUDED SECTIONS

The approximation of an occluded section of the boundary can be seen as a constraint problem. One constraint is that the curve must not pass within the segment we are approximating; it must stay outside the boundary of that segment. Figure 4.4 shows a segment boundary and its approximation. The approximating curve must not pass within the yellow region since that uncovers part of the background.

A possible constraint on the other side could be that the curve must not pass beyond the boundary of the neighboring segment. We have greater freedom if we require the curve to stay within any region where the layer is in front of the segment we are approximating. To see why this is helpful, picture a situation where the immediate neighbor of the segment is rather thin. The curve could easily pass beyond this segment to the next segment over. If that next segment is in front of the segment we are approximating, this is not a problem. However, if it is behind the segment we are approximating (or in the same layer), it exposes the curve that should be hidden, as seen in Figure 4.4.

Hence, we establish the constraint that the curve must stay within segments that are in front of the segment we are approximating, such as the blue region in Figure 4.4. Notice that this constraint encompasses our first constraint that the curve not pass within the segment we are approximating, since the layer of that segment is not strictly in front of the layer we are approximating. To avoid confusion with the terms “occluded” and “visible,” a curve that stays within this constraint is called “hidden,”
Figure 4.5: Even though the boundary approximation is entirely hidden, it still causes problems because the resulting B-spline covers up the one segment that is behind the segment we are approximating.

while one which violates this constraint is “exposed.”

Unfortunately, this constraint doesn’t quite handle all layering problems. Consider a situation in which the occluding region surrounds a small region of a lower layer (Figure 4.5). If the curve passes around the far side of the hole, the curve itself stays hidden, but the resulting surface is exposed through the hole. This is a very difficult situation to deal with, and possibly the only solution is to trace through the region within the curve pixel by pixel and make sure those pixels belong to segments of a higher layer.

The occluded curve is also constrained to the minimum continuity specified by the user. This continuity constraint not only applies to the occluded curve itself, but
4.2. APPROXIMATING OCCLUDED SECTIONS

it must also be continuous with the neighboring visible curves.

Because the visible curves may not quite reach the boundary of the occluding segment, it is necessary to allow a small portion of the occluded curve to be exposed near the junction with the visible section. Typically, that exposed portion is less than a few pixels long.

To keep the occluded boundary approximation within these constraints, we use a trial-and-error approach to approximating occluded boundaries. The algorithm works by repeatedly adjusting the curve in an attempt to get it within the constraint region, and inserting knots to get more freedom in the curve’s location.

First, an initial position for the curve is chosen subject to the continuity constraints. Making some minor changes to the visible boundary approximation method yields a good initial approximation of the occluded sections. Recall that the visible boundary approximation starts with the knot vector

\[ [0, 0, 0, 0, n - 1, n - 1, n - 1, n - 1] \] (4.16)

If we choose a different knot vector for the visible section, it gives us an initial approximation of the occluded portion of the boundary as well. Consider two B-Spline curves with knot vectors

\[ [\ldots, t_0, t_1, t_2, t_3, t_4, t_5, t_6] \] (4.17)

\[ [t_3, t_4, t_5, t_6, t_7, t_8, t_9, \ldots] \] (4.18)
We can easily construct a B-Spline curve that connects these two curves using the knot vector

\[ [t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9] \] (4.19)

The control points of the connecting curve are simply the last three control points of the first curve and the first three control points of the second curve. The points of intersection are \( C^2 \) continuous if the \( t_i \) values are all unique. Notice that in order for this to work, the end conditions of the two curves must match. The simple Bézier end condition of Equation 4.16 would not suffice.

We compensate by choosing a different initial knot vector for visible sections. Consider a section of the boundary that is visible from edge \( i_1 \) to edge \( i_2 \) (inclusive), then occluded from edge \( i_2 + 1 \) to edge \( i_3 - 1 \), and then it is visible again until edge \( i_4 \). If we choose initial knot vectors of the form

\[ \ldots, i_1, i_2, (1 - \phi)i_2 + \phi i_3, \phi i_2 + (1 - \phi)i_3, i_3 \] (4.20)

\[ [i_2, (1 - \phi)i_2 + \phi i_3, \phi i_2 + (1 - \phi)i_3, i_3, i_4, \ldots] \] (4.21)

the end conditions are preserved in the final approximation, as long as we only insert knots between \( i_1 \) and \( i_2 \) or between \( i_3 \) and \( i_4 \). The resulting effect is that the shape of the boundary is extrapolated into the area where it is not visible.

The variable \( \phi \) affects the shape of the occluded curve. It can take on any value between 0 and 0.5, since other values violate the requirement that knot vectors must
be non-decreasing. If $\phi$ is 0, the occluded curve is $C^1$ with the visible curves. If $\phi$ is 0.5, the occluded curve has a point of $C^1$ continuity at its midpoint. All values of $\phi > 0$ cause the occluded curve to have $C^2$ continuity with the visible curves. Beyond these observations, it is actually unclear exactly how $\phi$ affects the shape of the curve. An example of the effect of different values of $\phi$ can be found in the Results Section, Figure 7.13. The user is allowed to adjust the value of $\phi$, and the default is 0.1.

It is also possible to use a less symmetric knot vector by simply choosing any value between $i_2$ and $i_3$ for the two knots that depend on $\phi$. A symmetric approach that depends on a single user input $\phi$ is simpler and usually provides enough flexibility that the user can easily adjust it if the default value does not produce an acceptable result.

The approximating curve of an occluded boundary is initialized to a knot vector as described above, and it uses the last three control points of the curve before it and the first three control points of the curve after it. We must leave these six control points in place in order to maintain the continuity. Inserting knots may move the locations of the three control points on either end, but otherwise those control points are not allowed to move. It is easier to stitch the B-Splines together into a single curve if you also insert the equivalent knots into the neighboring visible curves.

Once the initial curve is computed, some method must be used to identify any exposed portions. Finding exposed portions exactly is a difficult problem, but it is
usually sufficient to evaluate the curve at fixed intervals and determine whether each sample is exposed or not. The interval should be as small as computationally feasible since an exposed portion smaller than that interval might not be discovered.

The largest exposed portion is identified, and a knot is inserted at its midpoint. Once again, care must be taken to ensure that inserting a knot does not violate the user’s continuity constraint or the minimum knot interval constraint. We use a similar method as before to adjust the knots if they violate these constraints (see Section 4.1).

Inserting a knot always gives us one control point we can move to any location without adversely affecting other portions of the curve. Usually, it is the middle of the three control points produced by the knot insertion. However, if the knot was inserted near either end of the curve, moving that control point would sometimes ruin the continuity with the neighboring curves. In this situation, use the control point that is farther away from the end of the curve.

Ideally, we want to move the control point to a position that would hide the entire curve, if possible. However, it is difficult to determine exactly which positions hide the entire curve. As an alternate approach, we only move the control point so one particular point on the curve is hidden. The chosen point is at the middle of the exposed portion.

Suppose we want the curve to pass through point \((x_i, y_i)\) at \(t = t_i\). We can guarantee that the curve passes through the point using the B-Spline parametric
4.2. APPROXIMATING OCCLUDED SECTIONS

equations

\[
x_i = \sum_{j=0}^{m-1} p_{xj} B_j^k (t_i)
\]

\[
y_i = \sum_{j=0}^{m-1} p_{yj} B_j^k (t_i)
\]

Solving these equations for a specific control point \((p_{xi}, p_{yi})\) yields

\[
p_{xi} = \frac{x_i - \sum_{j=0}^{i-1} p_{xj} B_j^k (t_i) - \sum_{j=i+1}^{m-1} p_{xj} B_j^k (t_i)}{B_i^k (t_i)}
\]

\[
p_{yi} = \frac{y_i - \sum_{j=0}^{i-1} p_{yj} B_j^k (t_i) - \sum_{j=i+1}^{m-1} p_{yj} B_j^k (t_i)}{B_i^k (t_i)}
\]

These equations give a formula for where to move the control point so that the curve passes through a given point \((x_i, y_i)\). If the basis function of that control point is zero \((B_i^k (t_i) = 0)\), these equations would have a division by zero. In this situation, moving that particular control point would not affect the position of the curve at \(t = t_i\), so we must choose a different point.

The last important question to ask is where we should move that point on the curve. In theory, any location on the image would suffice as long as it is hidden, but truthfully, some locations are better than others. For example, an ideal location not only causes that point on the curve to be hidden, but also many other exposed points. Additionally, the location should not cause the curve to wiggle too much or extend too far away from the segment we are approximating.

Picking a location that meets these requirements is very difficult. We have implemented a system that focuses on the third property described above: the curve
should not extend too far away from the segment we are approximating. In order to achieve this goal, we locate the point on the segment boundary that corresponds to the point we are moving. We then pick a point that is a fixed distance away from that location, measured perpendicularly to the general orientation of the boundary at that point. The general orientation of the boundary is determined by tracing along the boundary a few pixel edges in either direction and drawing a line between those two points. If this location is not hidden, it usually indicates that the occluding segment is very thin. If this happens, a point closer to the boundary is chosen, and the process repeats until we reach a point that is hidden.

Inserting one knot and moving a single point on the curve usually does well at reducing the amount of the curve that is exposed, but it frequently falls short of hiding the entire curve. In order to achieve the best possible result, the process is repeated many times. For each iteration, a portion of the curve is identified that needs to be hidden. Then, a knot is inserted at the middle of that portion of the curve. Finally, one control point is moved so as to hide that point on the curve. This basic process is repeated until the entire curve is hidden.

Unfortunately, this system sometimes repeats indefinitely since there is no guarantee that the amount of exposed curve is reduced. It is best to impose a limit on the number of iterations that will be executed to prevent an infinite loop. If that number is exceeded, the program simply reports to the user that it has failed to hide
4.3 Putting the Boundary Together

After all the visible and occluded sections of the boundary have been approximated, the curves are then pieced together into a single periodic B-Spline curve. As long as
the algorithm has been careful to keep matching knot vectors for consecutive curves, this is not difficult; simply combine the knot vectors and control points from each of the curves.

Once the boundary has been approximated, it is a good idea to show the result to the user and let the user manually adjust the control points before proceeding with the vectorization. This is necessary because the boundary approximation sometimes produces undesirable results, especially when approximating the occluded portions.
Chapter 5

Geometry Approximation

The next step in the vectorization process is “geometry approximation.” In this step, the B-Spline curve that approximates the boundary is converted into a B-Spline surface that approximates the shape of the segment. Another way to think of geometry approximation is the process of choosing positions for the control points that lead to a good approximation of the segment.

The end goal of geometry approximation is to produce a function of the form

$$ (x, y) = f(s, t) $$

that maps the parametric coordinates of the B-Spline surface, \((s, t)\), to the geometric coordinates of the image, \((x, y)\). From the equation of the B-Spline surface,

$$ (x, y) = \left( \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} p_{xij} B_i^k(s) B_j^k(t), \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} p_{yij} B_i^k(s) B_j^k(t) \right) $$

(5.2)
The geometry approximation determines the locations of the control points \((p_{xij}, p_{yij})\) and the two knot vectors.

The geometry approximation is primarily based on the boundary approximation since the boundary approximation already has a good approximation to the shape of the segment. The boundary approximation does not, however, specify anything about the nature of the interior of the segment, so the geometry approximation must produce that on its own.

There are a few steps to producing the boundary approximation. First, the algorithm must pick four locations on the boundary approximation to serve as corners for the B-Spline surface. Then, an initial geometry approximation is determined based on the shape of the boundary. After that, the approximation is adjusted to better fit lines and areas of detail in the interior of the segment. Finally, the B-Spline surface is untangled so that it does not overlap itself in any location. The following sections discuss these steps in detail.

## 5.1 Finding the Corners

In order to create a B-Spline surface that approximates the geometry of a segment, we first decide where to locate the four corners of the B-Spline. This not only affects the shape of the B-Spline, but also its orientation. The corners must be four distinct points on the boundary approximation curve. Equivalently, the corners could be four
5.1. FINDING THE CORNERS

points on the segment boundary itself, and the $t$ value associated with that point defines the true location of the corner. In fact, we are less concerned with the actual physical location of the corners than their $t$ values.

Although some shapes, such as rectangles, have very obvious corners, most real-world shapes do not. In order to accommodate non-rectangular objects, we define a set of qualities that ideal corners possess. The algorithm searches for four points on the boundary that best meet those ideals. We describe these qualities and how to measure them before presenting the specific algorithm used in our implementation.

The first quality of an ideal corner is obvious: corners are sharp. Recall that a B-Spline curve that is $C^0$ continuous causes a sharp bend in the curve. These points correspond to triple knots in the B-Spline’s knot vector. Such points are easily chosen as corners. In general, it is a good idea to pick corners where there are one or two knots even if there are no triple knots.

If the boundary approximation does not have exactly four triple knots (as is seldom the case), we can also find sharp corners by finding locations on the curve with high curvature. Curvature is related to the second derivative of a curve, and it is high at locations where the curve is changing directions rapidly. A corner algorithm could search for points where the curvature is high.

Instead of computing points of high curvature on the boundary approximation, we can approximate points of high curvature using the segment boundary itself. Because
of the discrete nature of the boundary, we must use a discrete approximation to the true curvature. There are many ways to approximate the curvature of a discrete line.

A second property of ideal corners is that they are usually located at the object’s extremities. They are often located at points of local maxima away from the center of the object.

The corners of a segment should be fairly evenly spaced around the boundary. It is not a good idea to have all four corners near each other, and there should never be two corners at the same point.

A related property is that the corners obey a certain amount of symmetry. While most objects are not perfectly symmetric, many objects are somewhat symmetric, and the corners should reflect that symmetry. One way symmetry is reflected in the corners is that corners are often grouped in sets of two; each corner is close to one of the other corners but far from the other two. This property best applies to long skinny objects.

All the qualities described so far pertain only to the shape of the boundary. The nature of colors within the segment can also be used to choose the corners. This is done because ideal corners affect not only the geometry approximation, but also the color approximation. In particular, good corners are located such that the orientation of the resulting B-Spline surface follows the orientation of lines in the image.

Our corner-picking algorithm mainly focuses on this last quality of good corners,
5.1. FINDING THE CORNERS

and mostly ignores some of the other qualities. It is important to note that this is not the only way to compute the corners; it is simply one of many possible ways.

First, the algorithm computes an axis that defines the “color orientation” of the segment. This axis represents a general overall orientation of any lines that may be visible in the segment. The second derivative is used because lines generally create high second derivatives.

To find the color orientation, the algorithm first computes an approximation to the second derivative at each pixel on the segment. This approximation is based on the gray levels of the pixel and its eight nearest neighbors. If the image is a color image, the colors are converted to their equivalent gray values. There are three second derivatives of this function that are computed as follows:

\[
\begin{align*}
  f_{xx} &= \frac{d^2 f}{dx^2} = f_{21} - 2f_{11} + f_{01} \\
  f_{yy} &= \frac{d^2 f}{dy^2} = f_{12} - 2f_{11} + f_{10} \\
  f_{xy} &= \frac{d^2 f}{dx dy} = \frac{(f_{22} - f_{02}) - (f_{20} - f_{00})}{4}
\end{align*}
\]

The \( f_{ij} \) values in these formulas represent the gray levels of the nine pixels used to derive the derivative. They are arranged in the following order, where \( f_{11} \) is the pixel
at the point where we wish to compute the second derivative:

\[
\begin{array}{ccc}
  f_{02} & f_{12} & f_{22} \\
  f_{01} & f_{11} & f_{21} \\
  f_{00} & f_{10} & f_{20}
\end{array}
\]

The formulas for the derivative with respect to \(x\) and the derivative with respect to \(y\) (represented by \(f_{xx}\) and \(f_{yy}\), respectively) come from fitting a parabola to the three points and determining the second derivative of the parabola. The formula for \(f_{xy}\) is more complicated, but essentially it comes from a bilinear interpolation of the four diagonal neighbors. Since the four points are spaced two pixels apart, we must divide by four.

The values of \(f_{xx}\) and \(f_{yy}\) approximate the second derivative in the \(x\) direction and the \(y\) direction, respectively. Alternately, we can measure what the second derivative with respect to \(x\) would be if we rotated the image by an angle \(\phi\) (Figure 5.1). The first derivative after rotating by \(\phi\) is the derivative with respect to a vector \(\vec{v} = (\cos \phi, \sin \phi)\) and is computed using the equation

\[
\frac{df}{d\vec{v}} = \left( \frac{df}{dx}, \frac{df}{dy} \right) \cdot \vec{v}
\]

or

\[
\frac{df}{d\vec{v}} = \frac{df}{dx} \cos \phi + \frac{df}{dy} \sin \phi
\]

The second derivative with respect to \(\vec{v}\) is computed as

\[
\frac{d^2f}{d\vec{v}^2} = \frac{d}{d\vec{v}} \left( \frac{df}{d\vec{v}} \right)
\]
5.1. FINDING THE CORNERS

Figure 5.1: The second derivative at any pixel on the image is approximated using the gray value of that pixel and its eight neighbors. After computing the derivative with respect to the $x$ axis and to the $y$ axis, the derivative may be computed with respect to a vector $\vec{v}$ by effectively rotating the image by an angle $\phi$. The angle $\phi$ that maximizes the absolute value of the second derivative is used as the general orientation of the second derivative at that point.

\[
\begin{align*}
\frac{d^2 f}{d\vec{v}^2} &= \left[ \frac{d}{dx} \left( \frac{df}{d\vec{v}} \right), \frac{d}{dy} \left( \frac{df}{d\vec{v}} \right) \right] \cdot \vec{v} \\
&= \left[ \frac{d^2 f}{dx^2} \cos^2 \phi + \frac{d^2 f}{dy^2} \sin^2 \phi + 2 \frac{d^2 f}{dydx} \cos \phi \sin \phi \middle/ \right. \\
&= f_{xx} \cos^2 \phi + 2 f_{xy} \cos \phi \sin \phi + f_{yy} \sin^2 \phi
\end{align*}
\]
We want to find the direction $\phi$ that maximizes the second derivative, since that likely identifies the orientation of any lines that travel through that point. The second derivative reaches a maximum (or a minimum) at all angles $\phi$ such that

$$\frac{d}{d\phi} \left( \frac{d^2 f}{d\vec{v}^2} \right) = 0 \quad (5.14)$$

$$-2f_{xx} \cos \phi \sin \phi - 2f_{xy} \sin^2 \phi +$$

$$+ 2f_{xy} \cos^2 \phi + 2f_{yy} \sin \phi \cos \phi = 0 \quad (5.15)$$

$$f_{xx} \tan \phi + f_{xy} \tan^2 \phi - f_{xy} - f_{yy} \tan \phi = 0 \quad (5.16)$$

$$f_{xy} \tan^2 \phi + (f_{xx} - f_{yy}) \tan \phi - f_{xy} = 0 \quad (5.17)$$

There are two maxima and two minima, and they are located at the four angles $\phi$ such that

$$\tan \phi = \frac{-(f_{xx} - f_{yy}) \pm \sqrt{(f_{xx} - f_{yy})^2 + 4f_{xy}^2}}{2f_{xy}} \quad (5.18)$$

It is often better to use the vector $\vec{v}$ instead:

$$\vec{v} = \left( \frac{a}{c}, \frac{b}{c} \right) \quad (5.19)$$

with

$$a = 2f_{xy} \quad (5.20)$$

$$b = (f_{yy} - f_{xx}) \pm \sqrt{(f_{yy} - f_{xx})^2 + 4f_{xy}^2} \quad (5.21)$$

$$c = \sqrt{a^2 + b^2} \quad (5.22)$$
5.1. FINDING THE CORNERS

The solutions to these equations are located at right angles to each other. The solutions are substituted into Equation 5.13 to determine which produces the largest absolute value. We call this direction the “second derivative axis” at that point. We only need to compare two of them since the vectors in opposite directions produce the same second derivative.

This formula gives us a general idea of the orientation of the second derivative function at any point in the segment. The best way to determine the general orientation of the entire segment is to average the second derivatives \((f_{xx}, f_{xy}, \text{and} f_{yy})\) of each pixel in the segment. Then, the axis for the entire segment is computed by substituting the average second derivatives into Equation 5.18. Equivalently, the sums of the second derivatives may be used to avoid the need for division.

We use the second derivative axis to determine a location for the four corners (see Figure 5.2). The segment is rotated so that the second derivative axis makes a 45-degree angle with the \(x\) axis. Then, the maximum and minimum points on the boundary in the \(x\) and \(y\) directions are identified. If the object is a rectangle, this process identifies the four corners. Otherwise, it at least identifies some extremities that may be good corners.

Once those points are chosen, if any two of them are the same, or any two of them are within a certain small distance from each other, they are adjusted slightly to prevent singularities that occur if the corners are too close to each other.
The corners have now been identified. The described method for choosing the corners is based primarily on finding the general orientation of the colors in the image. There are, as noted earlier, many other possibilities we have not yet explored.

5.2 Initializing the Geometry

The initial approximation of the geometry is based entirely on the boundary curve and the corners that have been chosen. In this phase of vectorization, the positions of the boundary control points must be used to determine suitable locations for the interior control points. Once those control points are placed, they define a B-Spline
surface that can map the parametric coordinates \((s, t)\) to geometric coordinates \((x, y)\).

The process for initializing the geometry approximation is based on the concept of Coons surfaces \([4]\). Coons surfaces are a method for interpolating four boundary curves to form a surface between them. Usually, Bézier curves are used for the boundary curves, but Coons surfaces can be bounded by any parametrically-defined curve.

A Coons surface starts with four curves \(C_0(t)\), \(C_1(t)\), \(C_2(t)\), and \(C_3(t)\). The curves must share end points, in other words,

\[
\begin{align*}
C_0(1) &= C_1(0) & (5.23) \\
C_1(1) &= C_2(1) & (5.24) \\
C_2(0) &= C_3(1) & (5.25) \\
C_3(0) &= C_0(0) & (5.26)
\end{align*}
\]

Now, define surface \(S_0\) that is a linear interpolation of \(C_0\) and \(C_2\):

\[
S_0(s, t) = (1 - t) C_0(s) + (t) C_2(s) \tag{5.27}
\]

A second surface \(S_1\) is the linear interpolation of \(C_1\) and \(C_3\):

\[
S_1(s, t) = (s) C_1(t) + (1 - s) C_3(t) \tag{5.28}
\]

A third surface \(S_3\) is the bilinear interpolation of the four corners:

\[
S_2(s, t) = (1 - s)(1 - t) C_0(0) + (s)(1 - t) C_1(0) +
\]
Finally, the Coons surface $S$ is defined by

$$S(s, t) = S_0(s, t) + S_1(s, t) - S_2(s, t)$$

(5.30)

This surface is bounded by the four curves since

$$S(s, 0) = C_0(s)$$

(5.31)

$$S(1, t) = C_1(t)$$

(5.32)

$$S(s, 1) = C_2(s)$$

(5.33)

$$S(0, t) = C_3(t)$$

(5.34)

and the surface interpolates the space between them. These properties of Coons surfaces make them ideal for initializing the geometry approximation.

This process can be directly applied to the case where the boundary curves come from a single, periodic B-Spline curve. Fortunately, a Coons patch with B-Spline curves on the boundary can be expressed as a B-Spline surface. The process starts with a periodic B-Spline curve $P(t)$ of degree $n$, as in Figure 5.3. The knot vector of this example is

$$[0, 5, 9, 17, 17, 17, 24, 29, 36, 42, 44, 52]$$

The corners are to be located at $t_0, t_1, t_2,$ and $t_3$. These corners must be in order with $t_0 < t_1 < t_2 < t_3$. The example for Figure 5.3 uses corners at $t_0 = 0, t_1 = 9,$
5.2. INITIALIZING THE GEOMETRY

Figure 5.3: A boundary with corners identified

$t_2 = 24$, and $t_3 = 36$. In this example, the corners are located on knots; that is not required, but it makes the example simpler. There are several steps to initialize a B-Spline surface from this B-Spline curve.

First, insert knots in the boundary curve at the four corners. Knots are inserted until the multiplicity at those points is the same as the degree of the curve; for a degree three curve, knots are inserted at $t_0$, $t_1$, $t_2$, and $t_3$ until each corner is a triple knot. This produces a control polygon as in Figure 5.4.

Now, extract the four curves from $P$ by copying the knots and the control points, as illustrated in Figure 5.4. The first $n + 1$ knots in the knot vector for curve $C_0$ are
Figure 5.4: The periodic B-Spline curve in Figure 5.3 has been split into four B-Spline curves by inserting knots at the four corners. The control polygon for each of the resulting curves has been drawn a different color.
5.2. INITIALIZING THE GEOMETRY

t_0. The next knot for C_0 is the first knot after t_0 in the knot vector for P, then the knot after that, and so forth until the n knots t_1 are copied. The last knot is also t_1. Similarly, the first control point for C_0 is the control point that corresponds to t_0 on the curve, and the control points are copied exactly as they are up to and including the control point that corresponds to t_1. A similar process is used to extract the other three boundary curves. Care must be taken when passing the end of P’s knot vector. After that knot, return to the beginning of the knot vector and increment each knot by the period of the curve. This is because P is a periodic B-Spline, but the C_i curves are not. In the example, the knot vectors for C_0, C_1, C_2, and C_3 are

\[[0, 0, 0, 0, 5, 9, 9, 9, 9] \]
\[[9, 9, 9, 9, 17, 17, 17, 24, 24, 24] \]
\[[24, 24, 24, 24, 29, 36, 36, 36, 36] \]
\[[36, 36, 36, 36, 42, 44, 52, 52, 52, 52] \]

respectively. This example is easy because 0 is one of the knots (52 refers to the same knot); if that had not been the case, we would have to increment some of the knots by 52 before placing them in the knot vector for C_3.

At this point, these curves are parameterized exactly as they are in the original curve. They must be reparameterized so they fit the Coons surface parameterization from 0 to 1. This reparameterization affects only the knot vectors; the control points
stay the same. For \( C_0 \), each knot \( t \) is replaced by \( (t - t_0) / (t_1 - t_0) \). Similarly, the knots for \( C_1 \) are replaced by \( (t - t_1) / (t_2 - t_1) \). It is important to note that the other two curves are backwards: they must be reparameterized so that the start of the curve is the end of the curve. This is done by replacing each knot \( t \) with \( (t_3 - t) / (t_3 - t_2) \) for \( C_2 \), and \( (t_0 + P - t) / (t_0 + P - t_3) \) for \( C_3 \). (\( P \) is the period of the original curve.) For these two curves, both the knot vector and the control points must be reversed. After reparameterizing, the knot vectors in our example are

\[
[0, 0, 0, 0, 5/9, 1, 1, 1, 1]
\]
\[
[0, 0, 0, 0, 8/15, 8/15, 8/15, 1, 1, 1]
\]
\[
[0, 0, 0, 0, 7/12, 1, 1, 1, 1]
\]
\[
[0, 0, 0, 0, 8/16, 10/16, 1, 1, 1, 1]
\]

In order to represent the final result as a B-Spline surface, the knot vectors of each set of opposing boundary curves must match. To achieve this, the knot vector of \( C_0 \) is compared with the knot vector of \( C_2 \). Any knots that are in \( C_0 \) but not in \( C_2 \) are inserted in \( C_2 \). Similarly, any knots in \( C_2 \) that are not already in \( C_0 \) are inserted in \( C_0 \). The same process is used to make \( C_1 \) have the same knot vector as \( C_3 \). For the example,

\[
[0, 0, 0, 0, 5/9, 7/12, 1, 1, 1, 1]
\]
5.2. INITIALIZING THE GEOMETRY

Figure 5.5: Control grid for surface $S_0$.

is the knot vector for $C_0$ and $C_2$, and

$$[0, 0, 0, 0, 1/2, 8/15, 8/15, 8/15, 5/8, 1, 1, 1, 1]$$

is the knot vector for $C_1$ and $C_3$.

Now, we are ready to produce the three intermediate surfaces. Surface $S_0$ is a degree $n \times 1$ B-Spline surface. The knot vector in the $s$ direction is the same as the knot vector for $C_0$ (which is the same as the knot vector for $C_2$). The knot vector in the $t$ direction is $[0, 0, 1, 1]$. There are two rows of control points. The first row contains the control points from $C_0$. The second row contains the control points from $C_2$. The control grid for this B-Spline surface is illustrated in Figure 5.5.
Similarly, surface $S_1$ is a degree $1 \times n$ B-Spline surface. The knot vector in the $s$ direction is $[0, 0, 1, 1]$, and the knot vector in the $t$ direction is the same as the knot vector for $C_1$ and $C_3$. There are two columns of control points. The first column contains the control points from $C_3$ and the second column contains the control points from $C_1$. The control grid for this B-Spline surface is illustrated in Figure 5.6.

Surface $S_2$ is a degree $1 \times 1$ B-Spline surface with both knot vectors $[0, 0, 1, 1]$. The four control points are the control points from the boundary curves at each of the four corners.

For the next step, each of the B-Splines must be degree elevated to a degree $n \times n$
5.2. INITIALIZING THE GEOMETRY

Figure 5.7: Surface $S_0$ after inserting knots to match surface $S_1$.

B-Spline. Elevating the degree of these three surfaces is really easy. For degree three B-Splines, two control points are inserted in the middle of each row of control points for surface $S_0$, located $1/3$ and $2/3$ of the way between the other two points. Similarly, each column of control points in $S_1$ has two control points inserted $1/3$ and $2/3$ of the way between the other two. For $S_2$, two control points are inserted on each row, and then two points are inserted on each of the four resulting columns, for a total of sixteen control points.

Next, the knot vectors of all three surfaces must be matched. This is exactly like matching the knot vectors of the curves. All the $s$ knots in $S_0$ that are not in the $s$
Figure 5.8: Surface $S_1$ after inserting knots to match surface $S_0$.

knot vector for $S_1$ are inserted in $S_1$. All the $s$ knots in $S_1$’s knot vector are already in $S_0$’s $s$ knot vector. This produces the surface in Figure 5.7. Similarly, all the $t$ knots in $S_1$ that are not in the $t$ knot vector for $S_0$ are inserted in $S_0$, producing the surface in Figure 5.8. All these knots are also inserted in $S_2$’s knot vectors, creating the surface in Figure 5.9.

The final step is to combine all these surfaces into one. Fortunately, because of the nature of B-Splines, adding the equations of two B-Splines that have the same knot vector is the same as adding the control points. So, each control point $P_i$ in $S$
Figure 5.9: Surface $S_2$ after inserting knots to match surfaces $S_0$ and $S_1$. 
Figure 5.10: This figure shows the geometry approximation as initialized from the boundary. The control grid for the B-Spline is also shown.

is

$$P_i = P_{0i} + P_{1i} - P_{2i}$$  \hspace{1cm} (5.35)

where $P_{0i}$, $P_{1i}$, and $P_{2i}$ are the control points from $S_0$, $S_1$, and $S_2$, respectively. The resulting surface is shown in Figure 5.10.

This B-Spline surface $S$ is the initial geometry approximation for the segment. It is based entirely on the boundary approximation, so it normally extends exactly to the boundary approximation. However, in some situations a concave boundary may cause the resulting surface to extend outside the boundary, then fold over on
5.3 Adjusting the Geometry

Because the initial geometry approximation described in Section 5.2 is based entirely on the boundary and the corners, it is often ill-suited for producing an ideal color approximation. For this reason, once the initial approximation has been made, the control points may be adjusted based on the colors of the image. In order to preserve the shape of the boundary, the control points around the edge of the surface must not move, but all other points are candidates for adjustment based on color. Geometry adjustment based on color is a topic of future research. We have not yet implemented it, but we describe some of the concepts that might be used to implement this kind of adjustment.

There are a few ideal situations which should be achieved by geometry adjustments. One is that the isoparameter lines of the B-Spline surface should follow lines in the image. It is good if the isoparameter lines corresponding to knots are along sharp lines in the image, and even better if they are double or triple knots. Another ideal is that there should be more control points in places with high amounts of detail or high-frequency texture. That leads to a better approximation of those areas.

A possible solution uses the second derivatives computed in Section 5.1. As stated
before, the direction of the maximum second derivative is frequently perpendicular to any lines. Because of that, the second derivative could be used to adjust the control points.

It is also true that the magnitude of the second derivative is higher in regions that have high-frequency textures or fine detail. The geometry adjustment could use the magnitude of the second derivative as an attractive force for nearby control points, causing them to move towards regions with higher second derivatives and away from regions with lower second derivatives.

Finally, it is very important to let the user move the control points during this step in the approximation. Placing the control points is a difficult problem, so it is always best to give the user the final say in where the control points should be located.

5.4 Mesh Untangling

The B-Spline surfaces developed using the Coons-patch algorithm described in Section 5.2 sometimes have locations where the surface folds over on itself. In certain cases, these wrinkles may even cause the surface to extend beyond its boundary. Frequently, they make a visible line on the resulting vector image where there should be no line. These folds are usually associated with concave portions of the segment boundary, although not all concave boundaries cause geometry folds. Folds may result
from interpolating boundary wiggles into the segment’s interior.

Frequently, you can “iron out” these wrinkles by moving the control points. The control points of a B-Spline surface form a grid, and the nature of the grid affects the underlying surface. Studying the shape of the quadrilaterals defined by four adjacent nodes on this grid can help eliminate folds in the surface. In particular, control points that form a concave quadrilateral (like quadrilateral ABCD in Figure 5.11) sometimes (but not always) cause surface folds. Similarly, quadrilaterals that are twisted in a bow tie shape (like quadrilateral EFGH in Figure 5.11) can cause wrinkles.
CHAPTER 5. GEOMETRY APPROXIMATION

Figure 5.12: The feasible set is built from half spaces formed by the surrounding mesh edges. This diagram shows one of the half spaces for the feasible set of this node.

Identifying and fixing those quadrilaterals usually gets rid of the folds.

Unfortunately, it is not always easy to fix erroneous quadrilaterals. Moving one control point to correct one quadrilateral often causes a different quadrilateral to become invalid. Grid theory literature describes many methods for untangling grids. We discuss two methods that have proven to be quite successful.

The first method, described in [24], is called the “feasible set method.” It involves iterating across each of the nodes in the grid and identifying which of them violate our constraints. When a node is found to violate the constraint, the positions of the neighboring nodes are used to determine the set of all positions for that node that do not cause one of the four neighboring quadrilaterals to be invalid. The region that
5.4. MESH UNTANGLING

Figure 5.13: The half space in Figure 5.12 has been intersected with another half space.

satisfies this constraint is called the node’s feasible set.

The feasible set is the intersection of the half spaces bounded by each of the eight edges around that point; one such half space is shown in Figure 5.12. Computing the feasible set involves starting with a polygonal representation of the entire plane and iteratively “trimming off” each of the half spaces, as shown in Figure 5.13. The resulting feasible set is shown in Figure 5.14.

The feasible set is always the region inside a convex polygon, though it may be an unbounded polygon (one that continues to infinity on one side), a degenerate polygon (such as a line segment), or even the empty set. Unbounded feasible sets only occur when the node in question lies on the edge of the grid, and those nodes should not
be moved anyway since moving them would ruin the boundary approximation.

If the feasible set is empty, there is no way to untangle the grid by only moving that one point. If this happens, it is often possible to correct the situation by moving one of the neighboring nodes into its feasible set. That does not handle all situations though, so this method is not guaranteed to untangle all possible grids although it succeeds in most situations.

Once the feasible set is computed, the node is moved to a point that is in the feasible set. Since the feasible set is a convex polygon, an easy way to find a point in its interior is to compute the average of the polygon’s vertices. The node in Figure 5.14 has been moved to the center of its feasible set.
Another way to untangle grids involves solving a maximization problem. We decide on a particular measurement of grid quality, and try to maximize that measurement by moving nodes around. Maximizing the heuristic across the whole grid should untangle the grid, unless it is impossible to untangle the grid. It would be ideal if we could find a heuristic that considers the grid as a whole. That way, the solution does not depend on the order the control points were visited. Unfortunately, it is nearly impossible to solve that kind of problem because it involves so many free variables. Because of this fact, the heuristic usually only depends on the position of a single node, and untangling the mesh involves iterating through the control points and maximizing the heuristic for each of them.

We use a system called the “maximum minimum area method,” based on an untangling algorithm by Freitag and Plassman [7]. The heuristic is computed by dividing each quadrilateral along its diagonals. The area of the four resulting triangles is measured: two of the triangles are formed by the sides of the quadrilateral and one of its diagonals, and the other two triangles are formed by the other diagonal, as illustrated by Figure 5.15. Notice that these triangles overlap. If a triangle is inverted, like triangle GPA in Figure 5.15, it is counted as having negative area. Inverted triangles are determined by the order of their vertices: counterclockwise order is positive, and clockwise order is negative. Moving a control point affects the relative areas of the triangles surrounding it, although the total area of the triangles
Figure 5.15: The four interior triangles highlighted on the left and the eight triangles shown on the right are used to straighten a problem node in the control grid. The four exterior triangles are not important because the location of $P$ does not affect the areas of those triangles. Triangle $GPA$ (highlighted with a yellow checker pattern) and triangle $APB$ are inverted, so their areas are counted as negative.

stays the same. The negative-area triangles are an indication that the grid is invalid.

We can avoid invalid grids by maximizing the area of the smallest triangle.

Consider node $P$ in Figure 5.15, located at position $(x, y)$. The neighboring nodes $A$ through $H$ are located at $(x_i, y_i)$ for $0 \leq i < 8$, numbered clockwise starting at the node immediately to the left (node $A$). There are sixteen triangles in the vicinity, but only twelve of them are affected by $P$; triangles $ABC$, $CDE$, $EFG$, and $GHA$ do not depend on the location of $P$. The area of the four interior triangles is

\[
2 \cdot \text{Area (APC)} = (x_2 - x, y_2 - y) \times (x_0 - x, y_0 - y) \tag{5.36}
\]
\[
= (y_2 - y_0)x + (x_0 - x_2)y + (x_2y_0 - x_0y_2) \tag{5.37}
\]
5.4. MESH UNTANGLING

\[ 2 \cdot \text{Area (CPE)} = (y_4 - y_2)x + (x_2 - x_4)y + (x_4y_2 - x_2y_4) \] (5.38)

\[ 2 \cdot \text{Area (EPG)} = (y_6 - y_4)x + (x_4 - x_6)y + (x_6y_4 - x_4y_6) \] (5.39)

\[ 2 \cdot \text{Area (GPA)} = (y_0 - y_6)x + (x_6 - x_0)y + (x_0y_6 - x_6y_0) \] (5.40)

The area of the eight other triangles is

\[ 2 \cdot \text{Area (APB)} = (y_1 - y_0)x + (x_0 - x_1)y + (x_1y_0 - x_0y_1) \] (5.41)

\[ 2 \cdot \text{Area (BPC)} = (y_2 - y_1)x + (x_1 - x_2)y + (x_2y_1 - x_1y_2) \] (5.42)

\[ 2 \cdot \text{Area (CPD)} = (y_3 - y_2)x + (x_2 - x_3)y + (x_3y_2 - x_2y_3) \] (5.43)

\[ 2 \cdot \text{Area (DPE)} = (y_4 - y_3)x + (x_3 - x_4)y + (x_4y_3 - x_3y_4) \] (5.44)

\[ 2 \cdot \text{Area (EPF)} = (y_5 - y_4)x + (x_4 - x_5)y + (x_5y_4 - x_4y_5) \] (5.45)

\[ 2 \cdot \text{Area (FPG)} = (y_6 - y_5)x + (x_5 - x_6)y + (x_6y_5 - x_5y_6) \] (5.46)

\[ 2 \cdot \text{Area (GPH)} = (y_7 - y_6)x + (x_6 - x_7)y + (x_7y_6 - x_6y_7) \] (5.47)

\[ 2 \cdot \text{Area (HPA)} = (y_0 - y_7)x + (x_7 - x_0)y + (x_0y_7 - x_7y_0) \] (5.48)

Although it is possible to consider all twelve of these triangles, that makes the maximization problem more difficult to solve. On the other hand, only considering the area of the interior triangles usually leads to an adequate solution. Unfortunately, unless all twelve triangles are considered, there are certain tangles which this method cannot fix.

If only the four triangles are used, the function we are maximizing is composed of four planes. Clearly, the maximum must be one of the four points of intersection
Figure 5.16: The grid has been fixed using the maximum minimum area method. In addition to the selected point (Q), the other four candidate points are marked.

of the various planes, so the simplest way to find the maximum is to compute the four points of intersection. These four points represent points where three of the four triangles have the same area. From those four points, eliminate the ones that do not qualify as the minimum area (because the area of the other triangle is smaller than the three that have the same area). The largest of the remaining points is the point with the maximum minimum area.

For example, if the node in Figure 5.16 is positioned at P, the area of three triangles is 4540.3, and the area of the other triangle is 4212.1. Similarly, at point Q, the area of three triangles is 4328.2, and the other triangle is 4848.4. At point R, the area of three triangles is 4628.1, and the other triangle is 3948.6. At point S, the
area of three triangles is 4361.9, and the other triangle is 4747.2. Points \( P \) and \( R \) are eliminated because the three matching triangles have greater area than the other triangle. Of the two remaining points, \( Q \) has the larger area for the three matching triangles, so the node is positioned there.

If all twelve triangles are considered, it is better to compute the maximum by dynamic programming. We have not implemented the maximum minimum area method with all twelve triangles.

The maximum minimum area method usually does not untangle the grid on the first try. Rather, with each iteration it makes the grid a little better.

Both of these methods work well at untangling certain grids, but neither can untangle all situations. It is best to use them together. First execute several iterations of the maximum minimum area method, followed by a single iteration of the feasible set method. This process is repeated until the grid is untangled.
Chapter 6

Color Approximation

The next step in the vectorization process is “color approximation.” This step assigns colors to the control points so that the B-Spline defines the color as well as the geometry. The B-Spline equation will then define five variables:

$$(x, y, r, g, b) = f(s, t) \quad (6.1)$$

The method for choosing the correct color for each control point is similar to boundary approximation. It is also done using least-squares approximation with an iterative refinement of the approximation.
CHAPTER 6. COLOR APPROXIMATION

6.1 Choosing Samples

In order to form a good approximation of a segment, one must choose the sample points to which the data is fit. For each sample point, we not only need to know its geometric coordinates \((x_i, y_i)\), but also the equivalent parametric coordinates \((s_i, t_i)\) in the geometry approximation. The geometric coordinates are needed to determine the color of that sample point, and the parametric coordinates are needed to compute the basis functions.

There are two options for choosing the sample points. One possibility is to have one sample at the middle of each pixel in the segment, as illustrated by Figure 6.1. Using this method, the geometric coordinates of the sample point are easily known:
6.1. CHOOSING SAMPLES

Figure 6.2: Samples placed by evaluating the parameter space of the B-Spline surface at regular intervals.

they are simply the \((x, y)\) coordinates of the pixel. However, it is not easy to compute the parametric coordinates given the geometric coordinates.

One way to compute the parametric coordinates of each pixel is to draw the B-Spline surface using any standard B-Spline drawing routine (one such routine is described in Appendix A), but instead of drawing the color at each pixel, record the \((s, t)\) coordinates in a buffer. After the spline has been drawn, the buffer contains the parametric coordinates associated with each pixel in the segment. One of the reasons it is important that the B-Spline surface has no wrinkles is to guarantee there is a one-to-one correspondence between the parametric coordinates and geometric coordinates. Otherwise, the results are unpredictable.
The other option for choosing the sample points is to choose the parametric coordinates first, then evaluate the B-Spline equations to determine the corresponding geometric coordinates, as illustrated by Figure 6.2. For example, the samples can be chosen at regular intervals in parameter space: \((0, 0)\), then \((\Delta s, 0)\), then \((2\Delta s, 0)\), until the first row is complete. Then the second row is \((0, \Delta t)\), followed by \((\Delta s, \Delta t)\), \((2\Delta s, \Delta t)\), and so forth.

Invariably, with this method, some of the sample points are located outside the segment. Usually, this is because that region of the segment is behind another segment. It is also possible that the geometry approximation extends slightly beyond a visible segment boundary, but samples that close to the edge of the segment are less significant. In order to get the best possible color approximation, samples outside the segment boundary should be removed from the set of samples, so that only colors from inside the segment are approximated.

For the final color approximation, the values for \(\Delta s\) and \(\Delta t\) must be small enough that there is at least one sample in each pixel. Otherwise, the color of some pixels is not approximated. However, it is possible to sample at a larger interval for intermediate approximations and then sample at a small interval for the final color approximation. This is an easy way to speed up the color approximation.

Sampling at fixed intervals in parameter space has a strong advantage because the B-Spline basis functions can be computed very quickly using forward differencing.
However, it usually leads to more than one sample per pixel, which can slow the computation down considerably.

6.2 Approximating the Color

Approximating the color of the B-Spline surface is very similar to approximating the boundary (Section 4.1). It also uses least-squares approximation, but using B-Spline surfaces instead of curves. The least-squares approximation is most successful if the color space of the image is linear. A color space is linear if the intensity of light is directly proportional to the number representing that color. Non-linear color spaces, such as indexed color, HSV, CMYK, and non-linear RGB should first be converted to a linear color space. The result may then be converted back to the original color space, but the transformation may be imprecise.

For the sake of simplicity, this document deals primarily with RGB images, but the principles described for three color components can be extended to more (such as an alpha channel) or fewer (such as gray scale). If the image has an alpha channel it is simply treated as a separate color component.

Once we have the coordinates of each sample, we determine the color of that point by looking it up in the image. If the sample point does not lie on a pixel center, simply choose the color of the nearest pixel. The basis functions at each sample point are also stored, since they are needed for least-squares approximation.
For each sample, we compute $x_i$ and $y_i$, the geometric coordinates; $s_i$ and $t_i$, the parametric coordinates; $r_i$, $g_i$, and $b_i$, the color; and $B_0(s_i,t_i)$, $B_1(s_i,t_i)$, $B_2(s_i,t_i)$, $\ldots$, $B_{n-1}(s_i,t_i)$, the basis functions. Unless we change the geometry by moving a control point, most of these values stay the same; only the basis functions near an inserted knot change. In other words, if we have enough memory to spare, we can store these values in a table between iterations, only updating the values that change. Unfortunately, the table can be very large if there are many samples (on the order of a few Gigabytes). In order to conserve memory, we could process this table in parts, computing only the pieces of the table we need and discarding the information when it is no longer needed.

Our goal is to solve the least squares problem defined by the matrix equation

$$A^TAX = A^TV$$

(6.2)

In this case, the matrix $A$ contains the basis functions for each sample:

$$A = \begin{bmatrix} B_0(s_0,t_0) & B_1(s_0,t_0) & \cdots & B_{n-1}(s_0,t_0) \\ B_0(s_1,t_1) & B_1(s_1,t_1) & \cdots & B_{n-1}(s_1,t_1) \\ B_0(s_2,t_2) & B_1(s_2,t_2) & \cdots & B_{n-1}(s_2,t_2) \\ \vdots & \vdots & \ddots & \vdots \\ B_0(s_{m-1},t_{m-1}) & B_1(s_{m-1},t_{m-1}) & \cdots & B_{n-1}(s_{m-1},t_{m-1}) \end{bmatrix}$$

(6.3)
The matrix $V$ contains the color at each of the samples:

$$V = \begin{bmatrix}
r_0 & g_0 & b_0 \\
r_1 & g_1 & b_1 \\
r_2 & g_2 & b_2 \\
\vdots & \vdots & \vdots \\
r_{m-1} & g_{m-1} & b_{m-1}
\end{bmatrix} \quad (6.4)$$

Often, $m$ is very large, so these two matrices can be difficult to store in memory. The matrix $X$ contains the solution to the least-squares approximation: a color for each of the control points.

Evaluating $A^TA$ results in a matrix with the following value in row $i$ and column $j$:

$$\sum_{k=0}^{m-1} B_i(s_k, t_k) B_j(s_k, t_k) \quad (6.5)$$

If we define a vector $a_k$ containing the basis functions evaluated at $(s_k, t_k)$

$$a_k = \begin{bmatrix}
B_0(s_k, t_k) \\
B_1(s_k, t_k) \\
B_2(s_k, t_k) \\
\vdots \\
B_{n_1}(s_k, t_k)
\end{bmatrix} \quad (6.6)$$

we can compute $A^T A$ without ever storing $A$ in its entirety.

$$A^T A = \sum_{k=0}^{m-1} a_k a_k^T \quad (6.7)$$
Similarly, we can compute $A^T V$ using the equation

$$A^T A = \sum_{k=0}^{m-1} a_k v_k^T$$

(6.8)

where the vector $v_k$ contains the colors of the sample.

Another clever trick involves noticing that only 16 values in $a_k$ are not zero, so it is easy to eliminate the multiplications and additions that involve zero. This means we have 304 multiplications and additions per sample ($304 = 16 \cdot 16 + 16 \cdot 3$). We also observe that one of the matrices is symmetric, so it can be computed in 136 multiplications and additions per sample instead of 256.

Computing these matrix multiplications is the most time-consuming task of the entire vectorization process. The speedups described above allow it to execute in linear time, since there are exactly 184 multiplications (and the same number of additions) for each sample. However, the number of samples is often large enough that this step may take several seconds to compute a single approximation.

### 6.3 Refining the Color Approximation

Like boundary approximation, the color approximation can be refined iteratively. Each step involves computing the least-squares approximation described above, then determining if the error exceeds a user-defined threshold. If it does, a knot is inserted.
and the process repeats until the error is small enough. The new knot provides more freedom so the approximation can be more accurate, reducing the error.

Determining where to insert a knot is not as easy as with boundary approximation since the B-Spline surface has two different knot vectors. One solution is to identify the point of maximum error and insert a knot in both the $s$ and the $t$ knot vectors at that point (or, in alternating iterations insert it in $s$ or in $t$). Another solution is to sum up the error of an entire column along an isoparameter line, and identify the column that has a maximum total error. With the latter method, the totals should be normalized by dividing by the length of the column. Then, the maximum in $s$ and the maximum in $t$ can be compared to determine which knot vector needs to be refined. (Neglecting to normalize usually means that most knots are inserted along the short axis of the object.) Usually, this second method for determining where to insert the knot is better.

Once a knot has been inserted, it is sometimes beneficial to approximate the segment boundary again so the new boundary B-Spline is closer to the true segment boundary. It is also possible to return to the geometry approximation step and adjust the new control points to better line up with lines in the segment. However, each of these steps must be taken with care as they often have adverse results. Our implementation does not try to re-approximate the boundary nor the geometry unless the user explicitly chooses to.
6.4 Extrapolating Colors

Solving Equation 6.2 often leads to a situation where $A^T A$ is not invertible. This happens if one row of $A^T A$ can be expressed as a weighted sum of the other rows, which usually causes library routines that solve matrix equations to give up without producing an acceptable answer. It is important to consider what causes this problem.

Remember that we ignore any sample that lies outside the segment. Where one segment overlaps another, there are large regions of ignored samples. If these regions are far enough outside the segment, we can assign any color to the control points in those regions without affecting the overall approximation. Because of this, the matrix equations are “unsolvable.” However, if we implement our own matrix-solving routine instead of using a library routine, we can assign a default value for those points whose color do not affect the overall appearance of the visible portions of the segment.

Our matrix-solving routine (see Appendix B) involves subtracting a multiple of one row of $A^T A$ from another row, and performing the same operation on $A^T V$. Each time this is done, one entry of the matrix can be forced to zero. The matrix-solving routine first eliminates all the nonzero values below the diagonal, which can be done even if $A^T A$ is not invertible. The second step tries to eliminate the nonzero values above the diagonal, but if an entry on the diagonal is zero, that can’t be done because the matrix is not invertible. The solution is to identify the rows that have a zero on the diagonal (they correspond to the control points that can be assigned
any color), and replace them with a row that contains one on the diagonal (and zero elsewhere). The equivalent row in $\mathbf{A}^T \mathbf{V}$ is replaced with a “default” color for that control point.

For the first iteration, the default color is the average color for that segment. For subsequent iterations, the default is to leave the color of the control point the same as the result of the previous iteration since a color that worked well in a previous iteration (before the singularity arose) is usually closer to other colors that may be near that region.

The overall effect of this is that the colors are extrapolated into regions that are behind other segments. The color approximation guesses what the colors might be where the segment is not visible. It is true that those colors do not affect the resulting vector image because of the layering (we could easily have used black for all of the defaults). However, if the user chooses to edit the image after it has been vectorized, the hidden portions of the segment can be uncovered, revealing a pretty good interpretation of what the color might have been in those regions. It is important to note, however, that this is still an extrapolation, and extrapolations are often very wrong. In fact, the color extrapolations produced by this modified least-squares approximation often produce colors with gray levels below zero or above white, like the unsightly light and dark areas in Figure 7.6.
Chapter 7

Conclusion

This vectorization method works well on many images, producing a reasonably similar vector image to many raster images. However, some images do not vectorize easily. The vectorization does not work well on images that have fine details or high-frequency textures since the color approximation of such images requires too many control points. Having too many control points makes the image more difficult for the user to edit. For other images, though, it produces a result with a manageable number of control points.

A chief advantage of this vectorization system over many others is in editability. It uses B-Spline surfaces, which are easy to edit and familiar to many users. The system tries to minimize the number of control points in the B-Splines to increase editability, and for images that are sufficiently simple, the number of control points
produced is small. The image has been segmented and layered, which also helps the user manage the shapes in the vector image.

It usually takes several minutes to a few hours to vectorize an image because of the user interaction required to vectorize the image successfully. The slowest part of the user interaction is generally segmentation, since the process is not completely automated. The slowest part performed by the computer is the color approximation. This may take several seconds per iteration for some segments, and producing the initial color approximation of the whole image may take several minutes.

Figure 7.1: This is a computer-generated image that should be easy to vectorize.
7.1 Results

The first example is a computer-generated image of a flower, shown in Figure 7.1. Since this image was originally a vector image, it should be easy to vectorize. The image dimensions are 508 × 575 pixels, and it uses an 8-bit indexed color space that has been converted to RGB.

Figure 7.2 shows a possible segmentation of the sample image. This image has been segmented using the Intelligent Scissors algorithm. There are 35 segments.

The segments were then layered manually, producing the layers shown in Figure 7.3. The automatic layering algorithm does not produce an acceptable result for
Figure 7.3: The segments in Figure 7.2 have been layered manually. Dark colors indicate background layers (lower numbers), and light colors indicate foreground layers (higher numbers).

this image, so we chose to layer the image manually. In this picture, the dark segments are background and the light segments are foreground. The background layer consists essentially of all the segments that were completely white in the original, and they are shown completely black in the layer image. This background has not been approximated in any future steps. There are a total of fourteen layers, since many segments are in the same layers as other segments (but not in the same layer as neighboring segments).

Once the segments were layered, the boundary of each segment was approximated, and the exposed portions of the boundary approximations were hidden. The resulting boundary approximation of one segment is shown in Figure 7.4. The purple curve is
the visible portion of the boundary approximation, and the red curve is the hidden portion. The control points are shown as yellow spots.

The boundary approximation of each segment was then used to create an initial approximation of the geometry. Some of the resulting B-Spline surfaces were untangled using a combination of the feasible set method and the minimum maximum area method. Figure 7.5 shows the geometry approximation of the segment in Figure 7.4. The purple points are control points, and the yellow lines are the control grid. The green curves show isoparameter lines corresponding to each of the knots, and provide a way to visualize the surface itself. Barely visible are four bright green spots representing the chosen corners for the geometry.
Figure 7.5: The boundary approximation from Figure 7.4 was used to initialize the geometry of this B-Spline surface.

Figure 7.6: The color of this segment has been approximated, producing the B-Spline surface shown.
Finally, the color of each segment was approximated, as shown in Figure 7.6. The dark and light regions around the fringe are examples of the results of color extrapolation into the hidden portions of the image. Those regions would have been hidden behind other segments. Those other segments were omitted in this image except for their boundaries and the original image in the background. The dark spot extends well below zero. Artifacts such as this come from extending the general color trend of the image near the boundary. If the image colors are getting darker as they approach the edge of the segment, those colors will continue to get darker beyond the edge of the image, even if that is not the “correct” color outside the segment. This is a problem with extrapolation in any context, and it is worse at higher degrees. This vectorization system uses a higher-degree extrapolation technique, extending the curvature in addition to the linear trend, which can cause even less “correct” extrapolations in some contexts, although it is more accurate in others. Future research may include improving this extrapolation technique. However, it is important to note that the “correct” color is a matter of personal preference, so it is likely that the user may need to adjust the color even if the algorithm were to choose a more visually pleasing result.

The final vectorized image is shown in Figure 7.7. The result is close to the original, but has lost some detail, particularly visible in the center of the flower and the loss of the veins of the leaves.
Figure 7.7: This shows the final vectorization of the image from Figure 7.1.

Figure 7.8: This is another raster image for vectorization. Unlike the previous example, it is a photograph of a real flower.
7.1. RESULTS

Figure 7.9: This shows a possible segmentation of Figure 7.8.

If we can vectorize computer-generated pictures of flowers, we should be able to vectorize pictures of real flowers as well. Figure 7.8 shows such a flower. This is a 363 × 477 image. The image is first divided into nine segments, as shown in Figure 7.9.

The next step is layering the image. First, we attempt to layer the segments automatically using concavity. The result is shown in Figure 7.10. For the most part, this layering is fairly consistent with the visual appearance of the image, however, this image poses a significant problem for layering. The original layering of the petals of this flower is cyclical, with each petal behind the petal immediately clockwise of it. That layering is possible in three dimensions because petals presumably slope towards the background. However, it violates our constraint that the layering must
be acyclic so we can assign layers using integers.

The cyclical layering of this image is avoided by layering it in a manner that is not consistent with its visual appearance; one petal is allowed to be in front of both its neighbors and one is allowed to be behind both its neighbors, as shown in Figure 7.11. Although most segments have been adjusted a little, the primary difference between the manual layering and the automatic layering is two petals have been moved back and three petals have been moved forward so there is only one violation of the original three dimensional layering. There are two background segments, which are not approximated. This background includes the stem and leaves as well as the green background of the original image. The other seven segments were each placed in a
7.1. RESULTS

Figure 7.11: The layers in Figure 7.10 have been adjusted manually.

The next step is to approximate the boundaries of the segments. First, the visible sections of each boundary are approximated, as illustrated in Figure 7.12. This is done by performing a least squares approximation, and then iteratively refining this approximation until it meets the user’s requirement (in this case, the user requires a maximum error of one pixel).

In approximating the boundary, the user may adjust the value φ used to initialize the occluded boundary approximations, as explained in Section 4.2. Figure 7.13 illustrates how this parameter affects the occluded boundary approximation. Once the boundaries are approximated, any exposed sections must be hidden. Figure 7.14
Figure 7.12: This series of images shows the process of approximating the visible sections of the boundary and refining it to get a closer approximation. The final approximation is shown in the lower right.
7.1. RESULTS

Figure 7.13: These images display the variation produced by different values for the hidden boundary parameter \( \phi \).

Figure 7.14: The boundary approximation on the left contains an exposed section. It has been hidden in the approximation on the right.
Figure 7.15: The short blue lines indicate the axis of maximum second derivative for each segment.

illustrates this process.

The next step is to choose the corners using the axis of maximum second derivative. Figure 7.15 shows this axis for each segment in the image. In theory, the axis should be approximately perpendicular to the major lines in the segment. Usually, this is true, but two of the axes in this image are not very perpendicular to the lines. Figure 7.16 shows the corners of one segment.

The corners and the boundary approximation are used to initialize the geometry approximation, as illustrated in Figure 7.17. The geometry of this segment has a $15 \times 17$ control grid. Figure 7.18 shows the effect of smoothing the control grid.

The next step is to approximate the colors of the image using least squares, which
7.1. RESULTS

Figure 7.16: The corners of this segment were chosen by rotating the segment 45-degrees from its second derivative axis and taking the maximum and minimum in \(x\) and \(y\). The corners are the green spots.

Figure 7.17: This figure shows the initial geometry approximation of the segment in Figure 7.16.
Figure 7.18: The control grid of the geometry approximation in Figure 7.17 has been smoothed.

Figure 7.19: This shows the color approximation of the image in Figure 7.8.
produces the image in Figure 7.19. This initial approximation has 1966 control points. The initial color approximation is then refined until it meets the user’s error tolerance. Each refinement produces a slightly better approximation. Figure 7.20 illustrates the result of refining to 3650 control points and then to 6429 control points. Although the refinement produces a better approximation to the original image, that improvement comes at a cost. First, it takes a long time to refine the approximation to great levels of detail. Second, the higher number of control points reduces the editability of the resulting vector image.

Figure 7.21 contains another image that vectorizes well. Unlike the previous ex-
Figure 7.21: An example image of a basket of fruit from a painting by Henri Fantin-Latour.

Figure 7.22: The automatic layering algorithm works fairly well on this image. Only the concave basket is layering incorrectly.
7.1. RESULTS

Figure 7.23: The image from Figure 7.21 has been vectorized. This vector image has 5276 control points, and it took about two hours to produce it.

amples, the automatic layering algorithm performs nearly flawlessly on this image, producing the layers in Figure 7.22. The only error in layering is the basket itself, which is layered behind the fruit because of its concave boundary. It does not take much effort to correct such a minor problem by hand. The result of vectorizing this image is shown in Figure 7.23. Overall, it took about two hours to vectorize this image. About half an hour was spent segmenting the image, and most of the rest of the time was spent in color approximation. The segmentation involves intense user interaction, while the color approximation is mostly done automatically, so the user can do other things while waiting for the heavy computation involved in refining this approximation. The user performing this vectorization opted not to refine the basket
nor the table to high levels of detail in the interest of time.

Vectorizing an image of a human face, such as the one pictured in Figure 7.24 poses a significant challenge to image vectorization. Part of the problem stems from the issue of segmentation. The parts of a face, such as the chin and cheek, are frequently not divided by a distinct line but rather a smooth change. Segmenting the image into such pieces produces a result like Figure 7.25, in which the lines from the segmentation are visible. An alternate approach would be to approximate the entire face as a single segment. This presents a different problem: the geometry should be aligned with the lines in the image. For example, one of the horizontal isoparameter lines should follow the eyebrows (assuming the subject has eyebrows), and two of the
vertical lines should follow the sides of the nose. This second approach to vectorizing a face is probably better, but would require developing an algorithm that can follow the lines better. Other than the segmentation issue and geometry alignment, the face of da Vinci’s *Mona Lisa* vectorizes well, since it contains little fine detail. Even her eyes are much more smooth than the detailed eyes of a photograph.

7.2 Future Work

While the vectorization process is fully functional as described herein, many things could be improved. We have discussed these at appropriate places throughout this text, but we summarize them here.
The process would greatly benefit from more automation. Future research should investigate ways to decrease the amount of user interaction needed. In particular, a more automatic segmentation process would be a great improvement. Also, the current automatic layering algorithm does not work well, so investigations should include ways to improve it.

Future research may also look into ways to allow segments with holes in them, as well as disjoint segments. As described earlier, this is not too difficult if the segment has holes or is disjoint because it is partially covered by another segment. It is more difficult to vectorize segments with true holes allowing segments behind to be visible.

The approximation of occluded boundaries could use improvement, since there are some situations where it does not produce ideal results. The corner-choosing algorithm could be improved to consider the shape of the boundary more, and not just the colors on the interior of the segment.

We discussed the possibility of automatically adjusting the geometry approximation to better align the control points with visible lines and regions of detail.

The major concern in color approximation is the speed; future research may include ways to decrease computation time. Additionally, future research may include ways to produce vector images with fewer control points overall.

While making these improvements could greatly improve the functionality of the
7.2. FUTURE WORK

process, it is recognized that many of them are difficult research projects. The algo-

rithm described in this paper effectively converts raster images into vector images by

dividing the image into segments and approximating each segment with a B-Spline

surface.
Appendix A

B-Splines

A B-Spline curve is a kind of curve that is commonly used in computer-aided design. B-Spline curves are parametric curves, which means that each point on the curve corresponds to a numeric parameter value. The curve can be expressed as a function of the parameter value, so that evaluating that function at any given parameter value yields a point on the curve. As the parameter values increase, the curve is traced out. B-Spline curves can always be decomposed into a set of Bézier curves, another popular parametric curve used in computer-aided design.

A B-Spline curve is defined using a set of control points, $P_0, P_1, \ldots, P_{n-1}$. The curve passes near the control points, though it usually does not pass through them. There can be any number of control points, but the number of control points must be at least one greater than the degree. The polyline formed by connecting each pair
Figure A.1: A degree one B-Spline curve is simply a polyline.

of consecutive control points is called the control polygon.

The degree of the B-Spline is a positive integer. It determines the continuity of the curve. The simplest B-Splines are degree one, such as the one shown in Figure A.1. A degree one B-Spline curve is simply the same as connecting the control points with line segments to form a polyline. Between each line segment (at each control point), there is a point of $C^0$ continuity. These line segments can be expressed as degree one Bézier curves.

Figure A.2 shows a degree two B-Spline curve. Degree two B-Spline curves are made of conic sections (usually parabolas, unless the curve is rational). Each parabola is connected to its neighbors by a point of $C^1$ continuity. Although the parabolas do not (in general) pass through the control points, they are tangent to the lines
Figure A.2: A degree two B-Spline curve, shown with its control polygon. The knot vector for this curve is $[-7, -3, 1, 1.5, 4, 7, 8, 9, 9]$. Connecting the control points, and those points of tangency are the points where the curve is $C^1$ continuous. Each parabola can be expressed as a degree two Bézier curve.

The most common degree for B-Spline curves is three, like the one shown in Figure A.3. A degree three B-Spline curve can be decomposed into Bézier curves of degree three. The curves are connected by points of $C^2$ continuity. Degree three B-Spline curves are usually called “cubic” B-Spline curves. Higher degrees than three are also possible, but usually $C^2$ continuity is sufficient, so higher degree curves are not common. In this paper, we generally work with cubic B-Splines, though the concepts can be applied to arbitrary degree.

Each B-Spline curve is defined using a set of numbers called knots. Together, they are known as the knot vector. The knot vector determines how close to each
control point the curve passes, the precise shape of the curve, and the relation between parameter values and points on the curve. The number of knots is always $k + 1$ greater than the number of control points, where $k$ is the degree of the curve.$^1$ The knots are always expressed in non-decreasing order; for the knot vector

$$[t_0, t_1, t_2, t_3, \ldots, t_{n+k-1}, t_{n+k}]$$

the order must satisfy

$$t_0 \leq t_1 \leq t_2 \leq t_3 \leq \cdots \leq t_{n+k-1} \leq t_{n+k}$$

With the exception of the first $k$ and last $k$ knots, each pair of consecutive knots

---

$^1$ Since the invention of B-Splines, it was determined that the first and last knot do not have any effect on the shape of the curve, so they are completely unnecessary. For this reason, some people omit the first and last knots, so that the number of knots is $k - 1$ greater than the number of control points. In this paper, we adhere to the convention of including those two useless knots.
defines the parameter range for one of the Bézier curves that compose the B-Spline. The first \( k \) and last \( k \) knots are known as the end condition knots. They define the shape of the curve near each end. The curve is defined from \( t = t_k \) to \( t = t_n \), where \( k \) is the degree and \( n \) is the number of control points.

The knot vector may contain duplicate knots. If a knot is repeated, rather than inserting a new Bézier curve at that point, the continuity at that point is decreased. In general, a knot that is repeated \( n \) times corresponds to a point on the curve with \( C^{k-n} \) continuity. For a cubic B-Spline, a double knot creates a point of \( C^1 \) continuity and a triple knot creates a point of \( C^0 \) continuity. In general, a knot is not allowed to repeat more than the degree of the curve, except at the beginning or end of the knot vector, where it may repeat \( k + 1 \) times.

Duplicate knots can also cause the curve to touch the control polygon. If a knot is repeated \( k - 1 \) times, the curve is tangent to the control polygon at that point (and it is a point of \( C^1 \) continuity). If a knot is repeated \( k \) times, the curve passes through one of the control points (and it is \( C^0 \) continuous at that point). It is also tangent to the control polygon at that point. Additionally, if two consecutive knots are repeated \( k \) times, then the control points near there are the same as the control points for the Bézier curve for that part of the curve. If the knot vector starts or ends with a knot that is repeated \( k + 1 \) times, it is called a Bézier end condition, since the curve behaves similar to a Bézier curve at that end.
If the difference between each two consecutive knots is the same, the B-Spline is called a “uniform” B-Spline. For example, the knot vector

\[ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] \]

causes a uniform B-Spline. Other B-Splines are called “non-uniform” B-Splines.

Moving a control point only changes part of the curve, and the portion that is changed can be determined by the knot vector. Control point \( P_i \) influences the curve in the range \( t_i < t < t_{i+k+1} \).

It is possible to insert a knot in the knot vector without changing the shape of the curve. This creates a new control point, and moves other control points near the point of insertion. Consider a portion of a B-Spline curve with the knot vector

\[ \ldots, t_{i-2}, t_{i-1}, t_i, t_{i+1}, \ldots \] \hspace{1cm} (A.1)

and the control points

\[ \ldots, P_i, P_{i+1}, P_{i+2}, \ldots \] \hspace{1cm} (A.2)

If we insert a knot \( \tilde{t}_i \), such that \( t_{i-1} \leq \tilde{t}_i < t_i \) the control points from \( P_i \) to \( P_{i+k-2} \) are replaced by control points \( \tilde{P}_i \) to \( \tilde{P}_{i+k-1} \), where

\[ \tilde{P}_j = \frac{t_{j+k} - \tilde{t}_i}{t_{j+k} - t_j} P_j + \frac{\tilde{t}_i - t_j}{t_{j+k} - t_j} P_{j+1} \] \hspace{1cm} (A.3)

Since repeating a knot \( k \) times causes the curve to pass through a control point, inserting a knot \( k \) times can be used to evaluate the curve at a particular point.
It is also possible to consider knot insertion geometrically instead of algebraically. First, we introduce the idea of knot intervals. Instead of using a knot vector, it is possible to express the knots in terms of the intervals between them. Each of these knot intervals is associated with one edge of the control polygon, as long as \( k \) is odd. Knot insertion now becomes interval splitting, since one of the knot intervals is divided in two.

To split the knot interval, first divide the corresponding edge of the control polygon into \( k \) pieces. The length of each piece is proportional to each of the \( k \) nearest knot intervals—\((k - 1)/2\) before it and \((k - 1)/2\) after it. Similarly, the \( k \) nearest edges are also divided into \( k \) pieces corresponding to the \( k \) intervals nearest to them. In each case, identify which segment corresponds to the knot interval being split, and divide it into two pieces, with the length of each piece proportional to the split interval. The point where this division is made is the location of the control point.

B-Spline curves may be expressed algebraically using the parametric equation:

\[
P(t) = \sum_{i=0}^{n-1} P_i B_i^k(t)  \tag{A.4}
\]

The functions \( B_i^k(t) \) are known as basis functions or blending functions. They form a weighted sum of the control points, where the weight depends on the parameter \( t \). The precise value of the basis functions depends on the knot vector. The basis functions are defined such that each basis function is never less than 0 nor greater than 1, and the sum of the blending functions is 1 for any value of \( t \). The recursive
definition for the B-Spline blending functions is

\[ B_i^0(t) = \begin{cases} 
0 & \text{if } t_i \leq t < t_{i+1} \\
1 & \text{otherwise} 
\end{cases} \]  
(A.5)

\[ B_i^k(t) = \frac{t_{i+k} - t}{t_{i+k} - t_i} B_{i}^{k-1}(t) + \frac{t - t_{i+1}}{t_{i+k+1} - t_{i+1}} B_{i+1}^{k-1}(t) \]  
(A.6)

Although it is common to work with B-Spline curves in two dimensions, it is possible to express the B-Spline curve equation as a parametric equation in any dimension by changing the number of dimensions for each of control point. A two dimensional curve is represented by

\[(x, y) = P(t)\]  
(A.7)

and the control points are

\[P_i = (p_{xi}, p_{yi})\]  
(A.8)

while a three dimensional curve is represented by

\[(x, y, z) = P(t)\]  
(A.9)

and the control points are

\[P_i = (p_{xi}, p_{yi}, p_{zi})\]  
(A.10)

A variation on B-Spline curves is to associate each control point with a weight. Larger weights pull the curve closer to the point, and smaller weights push the curve farther away. This kind of B-Spline is called a “rational” B-Spline. If the weights are
all one, this degenerates to a non-rational B-Spline. In order for this to work, the B-Spline equation (Equation A.4) must be modified to become

\[ \mathbf{P}(t) = \frac{\sum_{i=0}^{n-1} w_i \mathbf{P}_i \mathbf{B}_i^k(t)}{\sum_{i=0}^{n-1} w_i \mathbf{B}_i^k(t)} \]  

(A.11)

where \( w_i \) is the weight of control point \( \mathbf{P}_i \). A rational B-Spline is often called a NURBS, which is an acronym for Non-Uniform Rational B-Spline. The “non-uniform” refers to the fact that the intervals between knots are not always the same.

Sometimes it is desirable to put the two ends of the B-Spline together to make a loop. While this is possible using a standard B-Spline, it is usually difficult to get the ends to meet together with a high degree of continuity. In order to easily make a looped B-Spline, it is easier to use a variation on the B-Spline called a “periodic” B-Spline. A periodic B-Spline has no definite start or finish but instead forms a closed loop.

A periodic B-Spline can be thought of as a B-Spline in which the control points are repeated over and over. After the last control point, return to the first and so forth. In order to accommodate this, the intervals in the knot vector repeat over and over as well. The knot after \( t_n \) is the same as \( t_1 + t_n - t_0 \). The next knot is \( t_2 + t_n - t_0 \) and so forth. It is often easier to consider the knot vector in terms of knot intervals instead of knot values. The knot intervals simply repeat over and over.

The principle of B-Splines can be extended to surfaces as well as curves. A B-Spline surface is also defined with a parametric equation, only there are now two
Figure A.4: This figure illustrates the concept of a B-Spline surface in two dimensions. Each black curve in the figure is an isoparameter line corresponding to one of the knots. The control grid is also shown, with the control points labeled. The $s$ knot vector (horizontal) is $[-7, -3, 1, 1.5, 4, 7, 8, 9, 9, 9, 9, 9]$ and the $t$ knot vector (vertical) is $[0, 0, 0, 0, 1, 3, 3, 3, 3]$.

parameters: $s$ and $t$. Changing one parameter traces out a line on the surface in one direction, while changing the other parameter traces out a line in a different direction. These lines are called “isoparameter lines,” and it is interesting to note that the isoparameter lines on a B-Spline surface are in fact B-Spline curves.

Like B-Spline curves, B-Spline surfaces are expressed using a set of control points. Instead of a single series of control points, though, the control points of a B-Spline surface are arranged in a rectangular grid pattern, as in Figure A.4. Just as with B-Spline curves, moving the control points of a B-Spline surface causes the surface
near that control point to move.

B-Spline surfaces are expressed using two knot vectors instead of one. One knot vector defines the knots in the $s$ direction and the other defines the knots in the $t$ direction. The number of knots in each knot vector depends on the number of control points. If there are $n$ columns and $m$ rows in the control grid, the number of knots in the $s$ knot vector is $n + k_s + 1$ and the number of knots in the $t$ knot vector is $m + k_t + 1$; $k_s$ and $k_t$ refer to the degree in $s$ and $t$, respectively.

There is no requirement that the degree of the B-Spline surface is the same in both directions, so the degree is often expressed as two values like so: degree $k_s \times k_t$. B-Spline. The most common type is $3 \times 3$, known as a “bicubic” B-Spline surface.

Inserting a knot in the $s$ knot vector adds a whole new column of control points, and moves the control points in $k_s - 1$ other columns. Similarly, inserting a knot in the $t$ knot vector adds a new row and moves $k_t - 1$ rows. Inserting the same knot $k_s$ or $k_t$ times can be used to compute the control points of an isoparameter line.

The equation for a B-Spline surface is:

$$P(s, t) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} P_{ij} B_{k_s}^i (s) B_{k_t}^j (t)$$ (A.12)

The basis functions $B_{k_s}^i (s)$ and $B_{k_t}^j (t)$ are the same as the basis functions for B-Spline curves, with the $B_{k_s}^i (s)$ functions defined relative to the $s$ knot vector and the $B_{k_t}^j (t)$ defined relative to the $t$ knot vector. For simplicity, the basis functions can be
combined into one by substituting

\[ B_{ij}(s, t) = B_{ik}^i(s) B_{kj}^j(t) \]  \hspace{1cm} (A.13)

It is often easiest to think of B-Spline surfaces in three dimensions, but they can be expressed in any number of dimensions. This paper frequently refers to two dimensional and five dimensional B-Spline surfaces. It is difficult to see the point in defining a two dimensional B-Spline surface: how can there be a “surface” on a plane? It is easier to think of a two dimensional B-Spline as a map from \( s \) and \( t \) coordinates into \( x \) and \( y \) coordinates:

\[ P(s, t) = (x, y) \]  \hspace{1cm} (A.14)

This is achieved by making each control point two dimensional:

\[ P_{ij} = (p_{xij}, p_{yij}) \]  \hspace{1cm} (A.15)

The five dimensional B-Spline surface is an extension of this idea that also defines the color at that location:

\[ P(s, t) = (x, y, r, g, b) \]  \hspace{1cm} (A.16)

The values of \( r \), \( g \), and \( b \) specify the amount of red, green, and blue, respectively. This is achieved by assigning a color to each control point:

\[ P_{ij} = (p_{xij}, p_{yij}, p_{rij}, p_{gij}, p_{bij}) \]  \hspace{1cm} (A.17)
As with B-Spline curves, B-Spline surfaces can be rational by assigning a weight to each control point. The term “NURBS” refers to either a rational B-Spline surface or a rational B-Spline curve, but most people use it to refer to a surface.

Similarly, B-Spline surfaces can be periodic, although they are usually only periodic in $s$ or in $t$ but not both, making them tube-shaped. The ends of the tube can be closed by requiring that the first and last row (or column) of control points be the same point and giving the knot vector a Bézier end condition. A surface that is periodic in both directions only makes sense for surfaces that are topologically equivalent to a torus.
Appendix B

Least-Squares Approximation

Suppose that we have a set of points \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\) on the \(xy\) plane, like the ones shown in Figure B.1. We wish to find a line that approximates them. The points might be data points of the results of an experiment, and we wish to find a line that expresses the general trend of the data. In other words, we wish to find a linear function that expresses \(y\) in terms of \(x\). To simplify things, we require that the line pass through the origin, so it can be expressed as \(y = ax\).

To solve this problem, we need a way to determine how correct the solution is, that is, we need an error function. The error function is smaller when the line is closer to the data points and greater when the line is farther away. A suitable error function uses the vertical distance from each data point to the line, \(y_i - ax_i\). We compute the
total error as the sum of the squares of these distances:

\[ \epsilon(a) = \sum_{i=1}^{n} (y_i - ax_i)^2 \]  

(B.1)

Now, we just need to find the value for \( a \) that minimizes this error. The minimum occurs where the first derivative with respect to \( a \) is zero:

\[ \frac{d\epsilon}{da} = \sum_{i=1}^{n} 2 (y_i - ax_i) (-x_i) = 0 \]  

(B.2)

Solving for \( a \) yields

\[ a = \frac{\sum_{i=1}^{n} x_i y_i}{\sum_{i=1}^{n} x_i^2} \]  

(B.3)

which is where the error is at a minimum. Notice that if all the data points are
located at \( x_i = 0 \), the numerator and denominator are both be zero. This means any value for \( a \) would cause the same error.

We can extend this idea into more than one dimension. Consider some \( m + 1 \) dimensional data points for which we wish to express one variable in terms of the other \( m \) variables. The data points are of the form

\[
(a_{11}, a_{21}, a_{31}, \ldots, a_{m1}, v_1)
\]
\[
(a_{12}, a_{22}, a_{32}, \ldots, a_{m2}, v_2)
\]
\[
\vdots
\]
\[
(a_{1n}, a_{2n}, a_{3n}, \ldots, a_{mn}, v_n)
\]

We wish to find a linear function of the form

\[
v = x_1 a_1 + x_2 a_2 + \cdots + x_m a_m
\]

that closely approximates the data points. The error function is

\[
\epsilon = \sum_{i=1}^{n} \left( v_i - \sum_{j=1}^{m} x_j a_{ij} \right)^2
\]

In order to find the minimum of this equation, we find the point where the derivative with respect to each of the \( x_j \) is zero.

\[
\sum_{i=1}^{n} 2 \left( v_i - \sum_{j=1}^{m} x_j a_{ij} \right) (-a_{i1}) = 0 \quad \text{(B.7)}
\]
\[
\sum_{i=1}^{n} 2 \left( v_i - \sum_{j=1}^{m} x_j a_{ij} \right) (-a_{i2}) = 0 \quad \text{(B.8)}
\]
\[
\vdots
\]

(B.9)
\[
\sum_{i=1}^{n} 2 \left( v_i - \sum_{j=1}^{m} x_j a_{ij} \right) (-a_{im}) = 0 \quad (B.10)
\]

This is a set of equations that may be expressed in linear algebra form:

\[
\begin{bmatrix}
\sum_{i=1}^{n} a_{i1} a_{i1} & \sum_{i=1}^{n} a_{i2} a_{i1} & \cdots & \sum_{i=1}^{n} a_{im} a_{i1} \\
\sum_{i=1}^{n} a_{i1} a_{i2} & \sum_{i=1}^{n} a_{i2} a_{i2} & \cdots & \sum_{i=1}^{n} a_{im} a_{i2} \\
\vdots & \vdots & \ddots & \vdots \\
\sum_{i=1}^{n} a_{i1} a_{im} & \sum_{i=1}^{n} a_{i2} a_{im} & \cdots & \sum_{i=1}^{n} a_{im} a_{im}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_m
\end{bmatrix}
=
\begin{bmatrix}
\sum_{i=1}^{n} v_i a_{i1} \\
\sum_{i=1}^{n} v_i a_{i2} \\
\vdots \\
\sum_{i=1}^{n} v_i a_{im}
\end{bmatrix} \quad (B.11)
\]

Notice that the matrix on the left is the same as \( A^T A \), where

\[
A = \begin{bmatrix}
a_{11} & a_{12} & \cdots & a_{1m} \\
a_{21} & a_{22} & \cdots & a_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nm}
\end{bmatrix} \quad (B.12)
\]

and the vector on the right is the same as \( A^T v \), where

\[
v = \begin{bmatrix}
v_1 \\
v_2 \\
\vdots \\
v_n
\end{bmatrix} \quad (B.13)
\]
and \( \mathbf{A} \) is defined as before. If we also define the vector \( \mathbf{x} \) to be

\[
\mathbf{x} = \begin{bmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_m
\end{bmatrix}
\]  
(B.14)

we can express the entire equation as

\[
\mathbf{A}^T \mathbf{A} \mathbf{x} = \mathbf{A}^T \mathbf{v}
\]  
(B.15)

The solution to that equation is the hyperplane that most closely approximates the data points.

There are many algorithms for solving matrix equations like this one. Given a matrix equation

\[
\mathbf{B} \mathbf{x} = \mathbf{y}
\]  
(B.16)

we wish to solve for \( \mathbf{x} \), where \( \mathbf{B} = \mathbf{A}^T \mathbf{A} \) and \( \mathbf{y} = \mathbf{A}^T \mathbf{v} \). An easy solution is Gaussian reduction, which performs a series of operations to both sides of the equation that make the matrix simpler but do not change the solution. For example, you could add a multiple of any row of \( \mathbf{B} \) to another row of \( \mathbf{B} \) without changing the solution, as long as you perform the same operation using the same two rows of \( \mathbf{y} \).

It is possible to perform this same operation over and over until \( \mathbf{B} \) becomes an
upper triangular matrix. For example, if

$$B = \begin{bmatrix}
    b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\
    b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \\
    b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\
    b_{41} & b_{42} & b_{43} & b_{44} & b_{45} \\
    b_{51} & b_{52} & b_{53} & b_{54} & b_{55}
\end{bmatrix} \quad (B.17)$$

you would first subtract $b_{21}/b_{11}$ times the first row from the second row, leaving the second row with a zero in the first column (and do the same to y). Then, subtract $b_{31}/b_{11}$ times the first row from the third row, leaving the third row with a zero in the first column. This process continues until the first column is all zero. Then, you use the second row to eliminate all the entries below it in the second column. This process continues until each column is entirely zero below the diagonal

$$B' = \begin{bmatrix}
    b_{11}' & b_{12}' & b_{13}' & b_{14}' & b_{15}' \\
    0 & b_{22}' & b_{23}' & b_{24}' & b_{25}' \\
    0 & 0 & b_{33}' & b_{34}' & b_{35}' \\
    0 & 0 & 0 & b_{44}' & b_{45}' \\
    0 & 0 & 0 & 0 & b_{55}'
\end{bmatrix} \quad (B.18)$$

You can then use a similar process to eliminate the entries above the diagonal. First, subtract $b_{45}'/b_{55}'$ times the fifth row from the fourth row. Then, subtract $b_{35}'/b_{55}'$ times the fifth row from the third row. This continues until all the fifth column is
zero except the fifth row. Next, subtract \( \frac{b_{34}}{b_{44}} \) times the fourth row from the third row, and so forth until all the entries are zero except the diagonal.

Finally, divide each row by the entry on the diagonal so that each row has a one on the diagonal (remember to divide the vector \( y \) by the same value). Now, the matrix is the identity matrix, so the modified \( y \) contains the solution.

Unfortunately, sometimes an entry on the diagonal is zero, which prevents using that entry to eliminate other entries. If this happens when you are trying to produce the upper-triangular matrix, you can sometimes swap it with one of the rows below it that does not have a zero in that column. But, if all the rows below it also have a zero in that column, there is no way to eliminate the zero in that column in the solution. This presents a problem when trying to produce the diagonal matrix.

Let’s consider what the zero on the diagonal means. It is similar to the problem that occurs when all the data points were zero in the 2D example. It means that we can set that value of \( x_i \) to any value without changing the total error. We solve the problem by replacing that row with a row containing a one on the diagonal, and \( y_i \) may be any value we want. Then, we can continue solving the matrix. This modified form of least-squares allows us to use a default value if we cannot find an exact solution.
Bibliography


