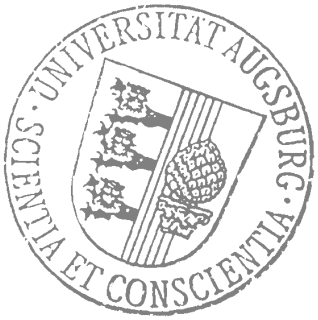
Vectorization of Pixel Art

Diploma Thesis

Christian Loos



January 11, 2012

Supervisor: Prof. Dr. Rainer Lienhart

Universität Augsburg
Fakultät für Angewandte Informatik
Institut für Informatik
Lehrstuhl für Multimedia Computing

Abstract

This thesis presents an approach to convert low resolution pixel art images into a resolution independent vector graphic representation, that can be scaled to arbitrary sizes without suffering from staircasing artifacts. To achieve this, we need to preserve very small features of the original image and resolve the problems that arise from the fact that diagonal neighbors in pixel art images share only a common vertex, which can lead to ambiguities in the connectedness of diagonal neighbors. We show a method that detects regions of similar color, solves ambiguous connections and reshapes pixel cells so that diagonally connected cells share an edge.

Afterwards smooth B-spline curves are fitted to the outlines of these regions. We then further optimize the spline curves to increase the smoothness further and to reduce staircasing artifacts that result from the low resolution of the input images. Then we present a method for rendering these curves that uses the depth buffer of the GPU to determine the color of each point in the image. Subsequently we diffuse the color to create soft transitions between different colors inside the same image region. In the end we qualitatively compare our output to the one achieved by other tools for vectorization.

Contents

1 Introduction

Recent years have seen computer and TV screens becoming bigger, and their prices dropping. With larger screen sizes and the introduction of high-definition television, screen resolutions have increased significantly. On the other hand games companies like Microsoft and Sony offer old games from the early days of computing at a cheap price. These works from the 80s and early 90s are introduced to a new generation. Their graphics were represented in pixel art, low resolution images where each pixel was often arranged by hand, instead of simply downscaling from high resolution images. Because of the low resolution, only few pixels were available to represent a feature, like the eye of a character in an image, making every single pixel important. These pixel art images are now displayed on the big modern screens, which makes them appear blocky.

General upscaling techniques that make no assumptions about the data in the image, like Nearest-Neighbor or Bicubic, either result in blocky images or blur sharp edges. Other approaches like the one from Fattal [11] or Glasner et al. [12] assume the data to have properties that are usually found in real-world pictures. These properties are usually not present in pixel art.

Then there exist specialized algorithms for upscaling of pixel art. Most of these algorithms originated in the emulator community and were not published in scientific papers. However, for many of them open source implementations exist. Generally these methods work on the pixel level of the image and analyze the local neighborhood of a pixel. Some produce their results by comparing the pixel configuration to a large number of cases [29]. Usually the specialized algorithms produce good results but all of them are limited to a fixed upscaling factor of either 2x, 3x, or 4x. Probably the earliest method was EPX [16]. It produces a magnification of factor 2 and interpolates between the colors in the 4-connected neighborhood of a pixel. Other algorithms based on the same principle are Scale2x [22] and 2xSaI [9]. Another specialized upscaling variant are the algorithms of the hqx family [29]. They regard 3 x 3 pixel blocks and compare the color of the center pixel to those in the surrounding pixels using a lookup table. This results in 256

different possible configurations, each giving a different output for the 3 x 3 block. There exist variants for upscaling up to factor 4.

A third possibility is converting raster images to vector graphics, thus achieving a resolution independent representation of an image. One such algorithm is called Potrace [27] but it works only on black and white images. Regarding every color channel separately would lead to different shapes in different channels.
There are a couple of algorithms that rely on segmentation algorithms to divide the image in different regions. In the work of Lai et al. [19] and Sun et al. [30] raster images are converted to gradient meshes. Lecot and Lévy [20] introduce a system called ARDECO to extract vector primitives and first- and second-order gradients from raster images. MacDonnald and Lang [21] follow the approach of a hierarchical segmentation, creating multiple layers containing different levels of detail of an image. The process they describe however is aimed at vectorizing natural images.
Other vectorization algorithms use edge detection to extract a vector graphic representation. Xia et al. [32] use a method decomposing the input image into non-overlapping triangles. Orazan et al. [25] partition the image by diffusion curves and then diffuse the color on both sides of the curve. They also describe a method to extract this representation from an image. However, both of these algorithms rely on Canny edge detection, which uses a Gauss filter. This filtering process looses the small features we find in pixel art and can therefor not be used for our purposes.
There also exist different tools that offer automatic vectorization of raster graphics such as Adobe Live Trace [2] or KVEC [17]. The exact nature of the used algorithms is not publicly available, but as we will see in comparisons performed later, these tools do not perform well on pixel art.

The approach we describe in this thesis is basically an implementation of the work of Kopf and Lischinski [18], whose method was specifically created for the use on pixel art. They reshape pixel cells based on similar colors found in the surroundings of a pixel and then extract quadratic B-spline curves from these cells to create smooth edges in the vector graphic image. Afterwards we will use a rendering method proposed by Jeschke et al. [15] that uses diffusion curves to compute the color of each pixel in an output image.

This thesis is organized as follows. Chapter 2 gives a brief overview of the differences between raster graphic and vector graphics representations. In Chapter 3 we introduce B-spline curves which will serve as the main primitive for our vector image. We then start with our actual approach to vectorization with solving

connectivity issues and reshaping pixel cells in Chapter 4. Afterwards, Chapter 5 extracts the B-spline curves from the reshaped pixels cells, before refining the position of their control points in Chapter 6. Then we present a method to render the vector image on a screen in Chapter 7. We conclude this thesis by comparing our results to those achieved by other vectorization algorithms and giving an overview of the runtime of our method in Chapter 8.

2 Vector Graphics

In this chapter we learn what vector graphics are, what they are used for, and how they differ from raster graphics. Therefor we first give a brief introduction to raster graphics in which our pixel art is stored. Then vector graphics will be presented, giving an impression of the different elements that can be used in vector graphics. This chapter concludes with a comparison of the two concepts of storing image data.

2.1 Raster Graphics

Raster graphics use a rectangular array of elements called pixels to represent an image. Each pixel is usually defined by its position in the array and either red, green and blue color values or a grayscale value. Thus many small points with a specific color form the whole image, often called a bitmap. This way of storing image data can be compared to the way the human eye works. Most modern display devices like television- and computer-screens work with a fixed raster of image cells. Raster graphics are therefor a natural way of storing image data intended to be displayed on such devices. It is also the representation of choice for image data representing photographs.

2.2 Vector Graphics

In [23] vector graphics are described in comparison to raster graphics as follows:

> "Vector [graphics] files contain, instead, mathematical descriptions of one or more image elements, which are used by the rendering application to construct a final image. Vector files are thus said to be made up of descriptions of image elements or *objects*, rather than pixel values."

In other words, vector graphics are a set of geometrical shapes with specific coordinates. One can think of vector graphics as the instructions to draw something.

Those instructions can be given in a number of different ways, often depending on the purpose the graphics were intended for. Fields of use include Computer-aided design (CAD), designing graphics for high resolution printers, printing and imaging languages like Adobe PostScript or Macromedia Flash animations [7]. This results in many different file formats for vector graphics. Some examples are the Computer Graphics Metafile (CGM) [28] an open international standard first defined in 1986, the XML-based Scalable Vector Graphics (SVG) format [31] which was defined by the World Wide Web Consortium (W3C) or Adobe PostScript [14] which is also used in PDF or Adobe Illustrator (AI) files. Given that vector graphics are basically instructions for a drawing it should not surprise us to see vector graphics stored in text form by languages like XML or PostScript. This also means we only need a text editor to create or modify vector graphics.

Now we have a look at the primitive objects that most vector graphics formats support. These are:

- Line
- Circle
- Ellipse
- Polyline
- Polygon
- Bézier curve
- Color gradient
- Text

Some formats support more complex objects like splines.

As we can see, text is directly supported as an object. Fonts store the outlines of their characters as a series of primitive vector data elements, which define how each character looks [23]. But the important part is that text in vector graphics knows it is text and thus is searchable, no matter how it is rotated or transformed in the image [31].

Primitive objects can be combined into more complex one and stored as templates for later use. These templates can then be copied to the desired location in

the image. Furthermore every object and template can be scaled, moved, rotated or modified by other mathematical operations.

2.3 Comparison of Raster and Vector Graphics

In this section we want to compare the two methods to store image data (i.e. vector and raster graphics) and highlight the strengths and weaknesses of each approach. We will use a simple example to illustrate some of the differences. The image we use is a simplified drawing of the sun, like a child would paint it on paper. The raster image is 150x150 pixels in size, the vector image used the same canvas size as a basis to define its objects. Both can be seen in Figure 2.1. Both images are almost identical. Due to the higher resolution of the display (or print) of this document the vector image is a little smoother then the raster one. In case they were really displayed at the same resolution they would be identical.

Internally both representations differ greatly. In the bitmap the color of each of the 150x150 pixels is defined. The vector graphics file on the other hand contains the instructions to draw a filled yellow circle with center $(75, 75)$ and radius 45, a red polyline with specific coordinates forms the mouth, two more red circles for the eyes and 8 lines for the rays.

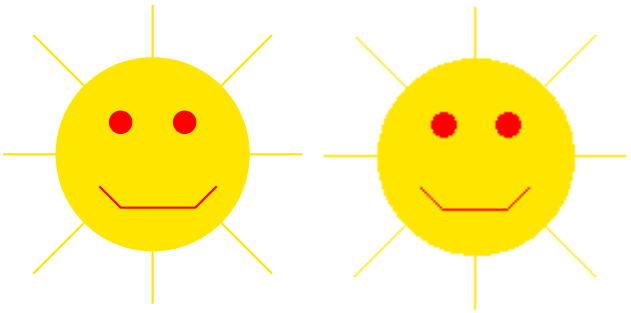


Figure 2.1: Vector (left) and raster (right) image of a simplified sun

This brings us to an important difference, the scalability of both representations. If we zoom in or otherwise enlarge our picture, the raster image has to create new pixels. A common method to do that is to increase the size of each pixel, that means if we wanted to enlarge a picture by a factor of 4, each pixel of our original image would become 4 pixels high and 4 wide. On the other hand, the vector image just multiplies all coordinates by 4 and draws the shapes at the full resolution the display or printer is capable of. Figure 2.2 shows the result of

zooming in on a part of the image by factor 4. We can see that the raster image to the right has quite blocky edges while the vector image is still smooth. There are other possibilities in raster graphics which produce smoother edges when enlarging images, but those make the result look a little blurred as can be seen in Figure 2.3.

Figure 2.2: By factor 4 enlarged segment of the previous sun image. The vector image is on the left, the raster image on the right

Figure 2.3: With method bicubic enlarged segment of the simplified sun.

We have seen that scalability is a great advantage for vector graphics. It is clear that vector graphics is the representation of choice when drawing lots of geometrical shapes that can be easily created with vector primitives. Images that contain much text should also use vector graphics, since text becomes searchable and depending on the font might even look smoother than in raster graphics.

The strong points of raster images are in displaying natural images and photos, since there are not many geometrical shapes in nature. They are well suited for output on common displays, whereas vector images have first to be translated in a suitable dot-based form.

3 B-Splines

B-spline curves and surfaces are of great interest in the field of computer aided design, where they have become a standard for curve and surface description. As such, there exist a number of books containing chapters about B-splines [4][26][10]. In this chapter we present basis splines or B-splines as a representation for piecewise polynomial functions.
Fist we introduce B-spline basis functions and some of their properties. The following section uses these basis functions to generate curves in a two dimensional space. Those B-spline curves are the foundation of the final vector graphics image we want to create. All edges of our final image will be represented by B-spline curves fitted to sequences of points we will extract from the raster graphic. Chapter 5 shows how to convert the edges we find in our input image into B-splines.

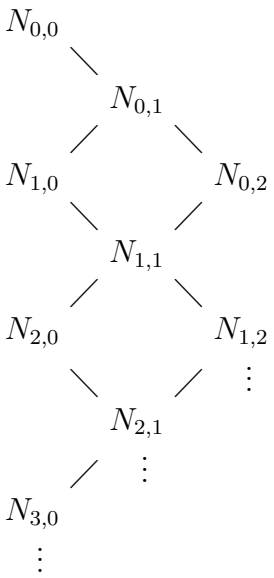
3.1 B-Spline Basis Functions

As all B-spline curves are constructed from several basis functions we will have a look at those first. This section gives a definition of B-spline basis functions and derives many of their properties. For this purpose it is common to use the **Cox-de Boor recursion formula** [4][26] given in equation (3.1).

Definition 3.1.1. *Let $U = u_0, \ldots, u_m$ be a sequence of non-decreasing real numbers $u_0 \leq \ldots \leq u_m$. The u_i are called* knots*, the set U is the* knot vector*, and the half-open interval $[u_i, u_{i+1})$ the* i-th knot span*. The i-th B-spline basis function of degree p, denoted $N_{i,p}(u)$, is recursively defined as:*

$$
\begin{aligned}
N_{i,0}(u) &= \begin{cases} 1 & \text{if } u_i \leq u < u_{i+1} \\ 0 & \text{otherwise} \end{cases} \\
N_{i,p}(u) &= \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u)
\end{aligned}
\tag{3.1}
$$

By this definition a knot span can have length zero, if $u_i = u_{i+1}$. In this case the coefficients of the second part of the equation can become the quotient $\frac{0}{0}$. This is then defined to be 0. Looking at the equation above we notice, that all basis functions of degree $p = 0$ are step functions with a value of 1 if u is inside the i-th knot span $[u_i, u_{i+1})$. For all $p > 0$ the $N_{i,p}$ are calculated as a linear combination of basis functions of degree $(p - 1)$. This results in the following triangular computation scheme.



Since we will use only quadratic B-spline curves (i.e. a curve of degree 2) in our approach to vectorization, we truncate basis functions of degree $p > 2$ from the scheme.

Next, we look at a small example calculation.Only a knot vector is needed to compute a basis function. We choose the knot vector $U = \{0, 1, 2, 3\}$. Since all knots have the same distance from each other, the knot sequence is said to be uniform. As we can easily see from Equation (3.1), $N_{0,0}(u) = 1$ on $[0, 1)$ and 0 elsewhere, $N_{1,0}(u) = 1$ on $[1, 2)$ and 0 elsewhere, and $N_{2,0}(u) = 1$ on $[2, 3)$ and 0 everywhere else. This means our basis functions of degree 0 look as shown in Figure 3.1.

Now, we can put them in the Cox-de Boor recursion formula to compute the

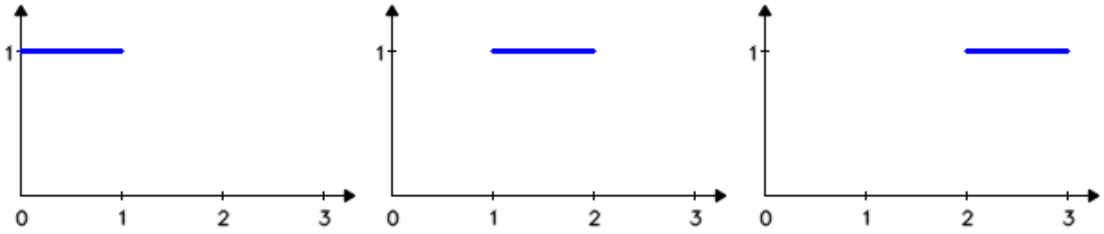


Figure 3.1: All zeroth degree basis functions, $U = \{0, 1, 2, 3\}$

first degree basis funtions, which are depicted in 3.2.

$$N_{0,1}(u) = \frac{u - u_0}{u_1 - u_0} N_{0,0}(u) + \frac{u_2 - u}{u_2 - u_1} N_{1,0}(u)$$

$$= \frac{u - 0}{1 - 0} N_{0,0}(u) + \frac{2 - u}{2 - 1} N_{1,0}(u) = \begin{cases} u & \text{if } 0 \leq u < 1 \\ 2 - u & \text{if } 1 \leq u < 2 \\ 0 & \text{otherwise} \end{cases}$$

$$N_{1,1}(u) = \frac{u - 1}{2 - 1} N_{1,0}(u) + \frac{3 - u}{3 - 2} N_{2,0}(u) = \begin{cases} u - 1 & \text{if } 1 \leq u < 2 \\ 3 - u & \text{if } 2 \leq u < 3 \\ 0 & \text{otherwise} \end{cases}$$

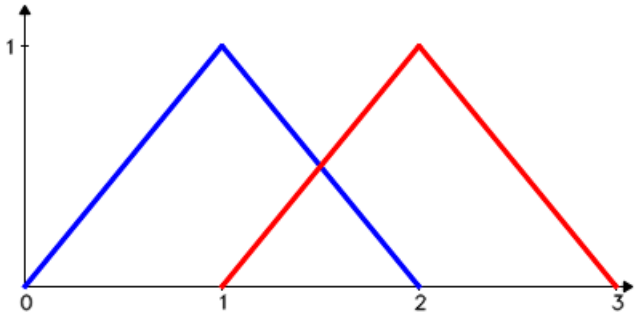


Figure 3.2: The two first degree basis functions $N_{0,1}$ in blue and $N_{1,1}$ in red , $U = \{0, 1, 2, 3\}$

As we can see the result of $N_{0,1}(u)$ depends on the knot span in which we evaluate it. In the zeroth knot span $N_{1,0}(u)$ is 0 and thus only the first term of the equation remains. $N_{0,0}(u)$ is 1 in the relevant span, so the result becomes u. The evaluation in other knot spans is done in the same fashion. With $N_{0,1}(u)$

and $N_{1,1}(u)$ available we can now calculate the second degree basis $N_{0,2}(u)$.

$$N_{0,2}(u) = \frac{u-0}{2-0}N_{0,1}(u) + \frac{3-u}{3-1}N_{1,1}(u) = \begin{cases} \frac{1}{2}u^2 & \text{if } 0 \leq u < 1 \\ \frac{1}{2}6u - 2u^2 - 3 & \text{if } 1 \leq u < 2 \\ \frac{1}{2}(3-u)^2 & \text{if } 2 \leq u < 3 \\ 0 & \text{otherwise} \end{cases}$$

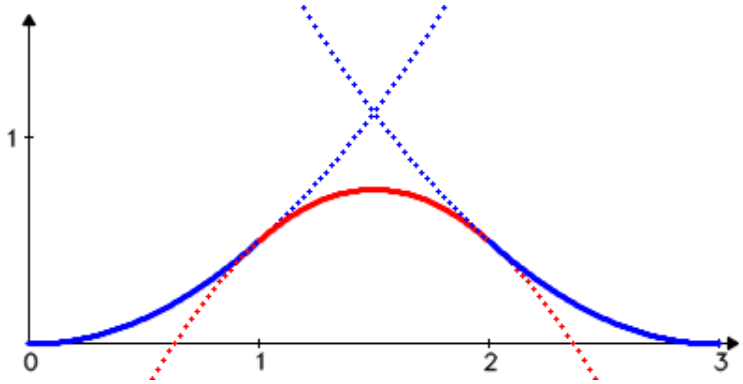


Figure 3.3: The three polynomial pieces compositing the second degree basis function, $U = \{0, 1, 2, 3\}$

Our example showed us, all basis functions $N_{i,p}(u)$ are piecewise polynomial functions with the knots as break points. In Figure 3.3 we can see the 3 parabolas that build $N_{0,2}(u)$. Clearly the first piece is joined to the second at $u_1 = 1$, and the second to the third at $u_2 = 2$.

The derivative of a B-spline basis function can be calculated as follows:

$$N'_{i,p}(u) = \frac{p}{u_{i+p} - u_i}N_{i,p-1}(u) - \frac{p}{u_{i+p+1} - u_{i+1}}N_{i+1,p-1}(u) \tag{3.2}$$

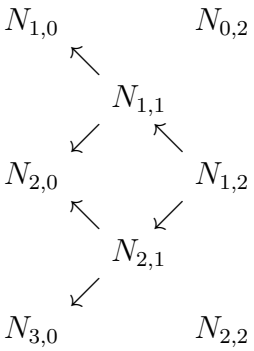
The proof for this equation is rather long and is therefor omitted. It can be found in [26].

3.1.1 Properties of B-spline basis functions

Next we present some properties of B-spline basis functions. The ones listed here and some more can be found in [26]. Where proofs of these properties are given they also closely follow those given by Piegl and Tiller in the same book.

Property 3.1.1. - *Local support*
$N_{i,p}(u) = 0$ *if* $u \notin [u_i, u_{i+p+1})$

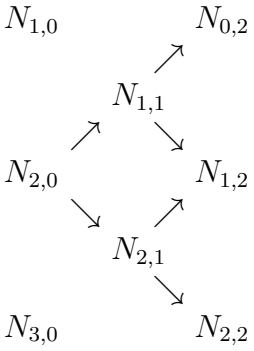
This property can be shown with the triangular scheme we used for calculating basis functions of higher degree.



Only the basis functions that are reached by an arrow contribute to $N_{1,2}(u)$. If u is not in one of the associated knot spans, all those basis functions will be zero, and consequently $N_{1,2}(u) = 0$ for all $u \notin [u_i, u_{i+p+1})$.

Property 3.1.2. *For any given knot span* $[u_j, u_{j+1})$ *at most* $p+1$ *of the* $N_{i,p}$ *are not zero. This are the functions* $N_{j-p,p} \dots N_{j,p}$.

The following diagram illustrates which functions are non-zero on the interval $[u_2, u_3)$. We can see that of the functions with degree 0 only $N_{2,0}$ is greater than zero. All functions that use $N_{2,0}$ in their calculation are reached by an arrow in the diagram. All other are zero in the relevant knot span. This makes $N_{0,2} \dots N_{2,2}$ the only second degree bases not zero on $[u_2, u_3)$.



Property 3.1.3. - *Non-negativity*
$N_{i,p}(u) \geq 0$ *for all* $i, p,$ *and* u.

Proof. We prove this by induction. It is obvious that for $p = 0$, $N_{i,p} \geq 0$ for all i and u.
Assumption: our claim holds for $p-1, p \geq 0$ and arbitrary i, u.
Then Equation (3.1) gives us:

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u)$$

$N_{i,p-1}(u)$ and $N_{i+1,p-1}(u)$ are non-negative by our assumption. Furthermore we know from Property 3.1.1 that $N_{i,p-1}(u) = 0$ when $u \notin [u_i, u_{i+p})$. For $u \in [u_i, u_{i+p})$ the numerator of the first coefficient is non-negative since $u \geq u_i$, and since our knot vector is a non-decreasing sequence $u_{i+p} - u_i$ can not be negative. The same can be shown for the second term and thus all $N_{i,p}(u)$ are non-negative. □

Property 3.1.4. - *Partition of Unity*
For an arbitrary knot span, $[u_i, u_{i+1}), \sum_{j=i-p}^{i} N_{j,p}(u) = 1$ *for all* $u \in [u_i, u_{i+1})$.

Proof. First we apply the Cox-de Boor recursion formula to $\sum_{j=i-p}^{i} N_{j,p}(u)$. This gives us:

$$\sum_{j=i-p}^{i} N_{j,p}(u) = \sum_{j=i-p}^{i} \frac{u - u_j}{u_{j+p} - u_j} N_{j,p-1}(u) + \sum_{j=i-p}^{i} \frac{u_{j+p+1} - u}{u_{j+p+1} - u_{j+1}} N_{j+1,p-1}(u)$$

Then we change the summation variable of the second sum from $i-p$ to $i-p+1$. Now, from Property 3.1.2 we know $N_{i-p,p-1}(u) = 0$ and $N_{i+1,p-1}(u) = 0$ on the selected knot span. This means the first term of the first sum, and the last term of the second sum become zero, which leaves us with:

$$\begin{aligned} \sum_{j=i-p}^{i} N_{j,p}(u) &= \sum_{j=i-p+1}^{i} \left[\frac{u - u_j}{u_{j+p} - u_j} + \frac{u_{j+p} - u}{u_{j+p} - u_j} \right] N_{j,p-1}(u) \\ &= \sum_{j=i-p+1}^{i} N_{j,p-1}(u) \end{aligned}$$

Applying the same principle recursively yields:

$$\sum_{j=i-p}^{i} N_{j,p}(u) = \sum_{j=i-p+1}^{i} N_{j,p-1}(u) = \sum_{j=i-p+2}^{i} N_{j,p-2}(u)$$
$$= \ldots = \sum_{j=i}^{i} N_{j,0}(u) = 1$$

□

3.2 B-Spline Curves

After our previous discussion of B-spline basis functions we can now use them to create B-spline curves. In our approach we want to fit these curves to a sequence of connected edges detected by our algorithm. The vertices connecting these edges will be called control points, as they will define the shape of the resulting curve. This section will give a definition of B-spline curves, then present different types of curves and finally derive different properties that arise from those of the B-spline basis functions.

Definition 3.2.1. *Given $n+1$ control points $\mathbf{P_0}, \ldots, \mathbf{P_n}$ and a knot vector $U = \{u_0, \ldots, u_m\}$ the p-th degree B-spline curve is defined as:*

$$\mathbf{C}(u) = \sum_{i=0}^{n} N_{i,p}(u)\mathbf{P_i} \tag{3.3}$$

where $N_{i,p}(u)$ are the B-spline basis functions from Equation (3.1).

In general we choose $u_0 = 0$ and $u_m = 1$, so that $0 \leq u \leq 1$. The polygon formed by the $\mathbf{P_i}$ is called the control polygon, and as we will see later, determines the general shape of a B-spline curve. A point $\mathbf{C}(u_i)$ on the curve which corresponds to the knot u_i is a *knot point*. These knot points divide the curve into curve segments, each defined on a knot span. The different figures used in this section depict knot points as yellow dots on the spline curve. Note that the equation $m = n + p + 1$ has to hold, which means that our knot vector has to contain $n + p + 2$ knots. In other words, the number of knots is determined by the number of control points and the degree of the curve. The reason for this can easily be seen in the triangular computation scheme from the previous section.

Now that we know how many knots are needed for a given number of control points to calculate the B-spline curve, we present 2 different choices for the knot vector and the influence it has on the final curve. First we differentiate between open and clamped curves. Both are depicted in Figure 3.4.

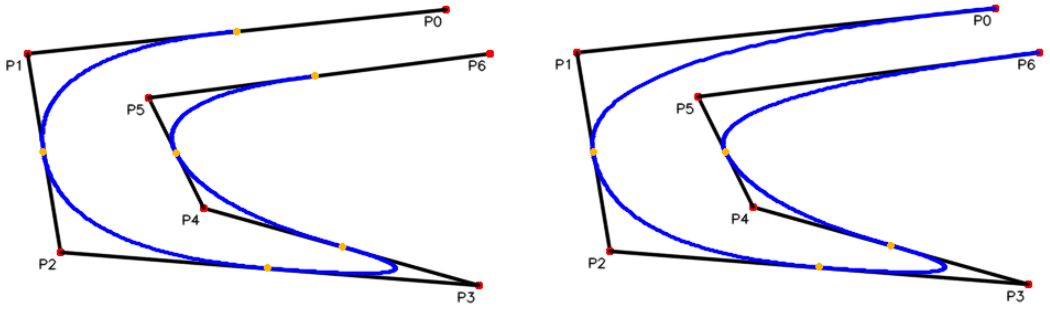


Figure 3.4: Open (left) and clamped (right) quadratic B-spline curve created from the same control polygon.

For our example we used a control polygon with $n + 1 = 7$ vertices in Figure 3.4. Since our curve is of degree $p = 2$ we get a knot vector of size $m = n+p+2 = 10$. The open curve has the knots spaced uniformly, giving us a knot vector $U = \{0, 0.1, \ldots, 1\}$. As we can see, the open curve does not reach the endpoints P_0 and P_6 of the control polygon. Recall from property 3.1.2 that each knot span has at most $p + 1$ non-zero basis functions. On open curves of degree 2 the first and second, and the last and next to last knot spans do not have the full number of 3 non-zero basis functions and are ignored for the calculation of open spline curves. This explains why the open curves do not reach the first and last control points.

In our application the control polygons are sequences of edges from a raster image, representing for example the leg of a character. As with any character the leg should be connected to torso and thus the spline representing the leg has to be connected to the one forming the torso. Our control polygons extracted from the input will be connected at a single control point where leg and torso meet. With an open knot vector the spline curves do not reach this control point and our two splines could not be connected. Thus we can not use open B-spline curves. Fortunately we can modify our knot vector to get a curve that starts at the first control point and ends at the last one. This type of knot vector and the resulting curves are called clamped. To get a clamped knot vector we have to repeat the first and last control points p times. Those knots are then said to have multiplicity $p+1$. That means $u_0 = \ldots = u_p = 0$ and $u_{m-p} = \ldots = u_m = 1$.

The other knots can be placed arbitrarily, but we again choose uniform spacing for the interior knots. Since the total number of knots we can use remains the same as in the open knot vector, this means in our example we now have 3 knots at 0 and 3 knots at 1, leaving only 4 knots for the range in between. Those are $u_3 = 0.2, u_4 = 0.4, u_5 = 0.6, u_6 = 0.8$. We now want to confirm that our curve starts at the first control point, so we calculate $\mathbf{C}(0)$ as follows:

$$\mathbf{C}(0) = \sum_{i=0}^{n} N_{i,2}(0)\mathbf{P_i}$$

Since $u_0 = u_1 = u_2$ causes the first and second knot span to have length zero, $u = 0$ lies in the third knot span. This in turn means, only basis functions that contain $N_{2,0}$ are non-zero. According to Property 3.1.2 this leaves us with $N_{0,2}, N_{1,2}, N_{2,2}$ as second degree basis functions, which reduces the equation above to:

$$\mathbf{C}(0) = N_{0,2}(0)\mathbf{P_0} + N_{1,2}(0)\mathbf{P_1} + N_{2,2}(0)\mathbf{P_2}$$

Applying the Cox-de Boor recursion formula brings us to:

$$\begin{aligned}
\mathbf{C}(0) = &\left[\frac{0}{0}N_{0,1}(0) + \frac{u_3}{u_3}N_{1,1}(0)\right]\mathbf{P_0} \\
&+ \left[\frac{0}{u_3}N_{1,1}(0) + \frac{u_4}{u_4}N_{2,1}(0)\right]\mathbf{P_1} \\
&+ \left[\frac{0}{u_4}N_{2,1}(0) + \frac{u_5}{u_5 - u_3}N_{3,1}(0)\right]\mathbf{P_2} \\
&= N_{1,1}(0)\mathbf{P_0} + N_{2,1}(0)\mathbf{P_1} + \frac{u_5}{u_5 - u_3}N_{3,1}(0)\mathbf{P_2}
\end{aligned}$$

Property 3.1.2 shows us, $N_{3,1}$ is also zero on $[u_2, u_3)$, which removes the last term from the equation. We apply the same recursion formula once more:

$$\begin{aligned}\mathbf{C}(0) &= \left[\frac{0}{u_3}N_{1,0}(0) + \frac{u_4}{u_4}N_{2,0}(0)\right]\mathbf{P_0} \\ &+ \left[\frac{0}{u_4}N_{2,0}(0) + \frac{u_5}{u_5 - u_3}N_{3,0}(0)\right]\mathbf{P_1} \\ &= N_{2,0}(0)\mathbf{P_0} \\ &= \mathbf{P_0}\end{aligned}$$

Similarly, can be shown that $\mathbf{C}(1) = P_n$. thus proving that indeed the first point of a clamped B-spline curve is P_0 and the last is P_n.

Additionally, we will need another type of B-spline curve in our approach to vectorization, the closed curve. A closed curve, as the name indicates, is a curve that returns to its beginning. This can be achieved by replicating the first p control points at the end of the control polygon, i.e. $P_0 = P_{n-p+1}, \ldots, P_{p-1} = P_n$. Since we do not want the closed curve to begin or end at a control point, we have to use a non-clamped knot vector. The result can be seen in Figure 3.5.

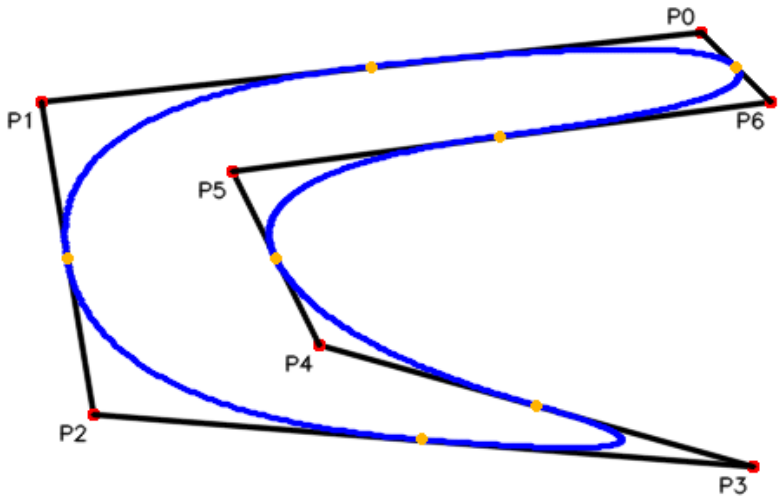


Figure 3.5: Closed quadratic B-spline curve.

Calculating the derivatives of a B-spline curve can be done by simply calculating the derivatives of its basis functions [26].

$$C'(u) = \sum_{i=0}^{n} N'_{i,p}(u)\mathbf{P_i} \tag{3.4}$$

3.2.1 Properties of B-spline curves

Now that we know how to construct the B-spline curves we need, we want to look at their properties. Piegl and Tiller list many properties in [26]. The ones presented here, though in some cases a little modified, have been taken from their book and proofs follow along the lines of those given there.

Property 3.2.1. $\mathbf{C}(u)$ *is a piecewise polynomial curve, since the* $N_{i,p}(u)$ *are piecewise polynomials.*

Property 3.2.2. - *Endpoint interpolation*
If the knot vector U *is clamped, then* $\mathbf{C}(0) = \mathbf{P_0}$ *and* $\mathbf{C}(1) = \mathbf{P_n}$

Property 3.2.3. - *Affine invariance*
Affine transformations like translations, rotations, scalings and shears can be applied to the curve by applying them to the control points.

Proof. Let $\mathbf{r}$ be a point in ε^2 (the two-dimensional Euclidean space). An affine transformation, denoted by Φ, maps ε^2 into ε^2 and has the form

$$\Phi(\mathbf{r}) = \mathbf{A}\mathbf{r} + \mathbf{v}$$

where $\mathbf{A}$ is a 2x2 matrix and $\mathbf{v}$ is a vector. The affine invariance property for B-spline curves follows from Property 3.1.4, the partition of unity of the $N_{i,p}(u)$. Thus, let $\mathbf{r} = \sum \alpha_i \mathbf{p_i}$, where $\mathbf{p_i} \in \varepsilon^2$ and $\sum \alpha_i = 1$. Then

$$\begin{aligned}
\Phi(\mathbf{r}) &= \Phi\left(\sum \alpha_i \mathbf{p_i}\right) \\
&= \mathbf{A}\left(\sum \alpha_i \mathbf{p_i}\right) + \mathbf{v} \\
&= \sum \alpha_i \mathbf{A}\mathbf{p_i} + \sum \alpha_i \mathbf{v} \\
&= \sum \alpha_i \left(\mathbf{A}\mathbf{p_i} + \mathbf{v}\right) \\
&= \sum \alpha_i \Phi(\mathbf{p_i})
\end{aligned}$$

□

Property 3.2.4. - *Strong convex hull*
A B-spline curve is contained in the convex hull of its control polygon. More specifically, if $u \in [u_i, u_{i+1})$ *then* $\mathbf{C}(u)$ *is in the convex hull of control points* $P_{i-p}, \ldots, P_i$.

Proof. The first part of this property follows directly from Property 3.1.3 and Property 3.1.4, the non-negativity and partition of unity of the $N_{i,p}(u)$. The second part is due to the fact that $N_{j,p}(u) = 0$ for $j < i - p$ and $j > i$ when $u \in [u_i, u_{i+1})$ (Property 3.1.2).

□

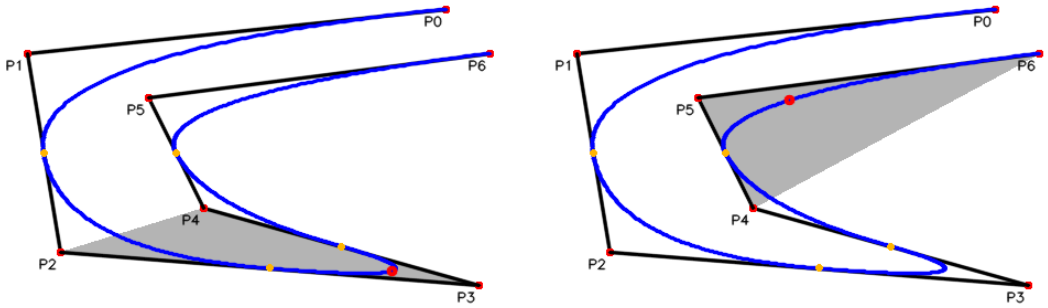


Figure 3.6: Showing the strong convex hull property. The left image shows the convex hull for the third curve segment, the right one the same for segment 5.

Figure 3.6 illustrates the strong convex hull property with two examples from our already familiar control polygon. The convex hull of the appropriate control points is colored in gray. The yellow circles mark the knot points. Crossing a knot point means passing from one knot span to the next. According to the convex hull property each curve segment lies in a different convex hull. This property also means that when we approximate the edges of a region in an image by B-spline curves, our curve will never be outside of the convex hull of the original region.

Property 3.2.5. - *Local modification scheme*
Moving a control point $\mathbf{P_i}$ *changes* $\mathbf{C}(u)$ *only in the interval* $[u_i, u_{i+p+1})$. *(Illustrated in Figure 3.7)*

Proof. This is a direct result from the fact that $N_{i,p} = 0$ if $u \notin [u_i, u_{i+p+1})$, stated in Property 3.1.1.

□

As a result of the local modification scheme, we can later smooth our curve locally, without affecting the general shape of the spline.

This concludes the chapter about B-splines. We have seen that B-spline basis functions are a representation for piecewise polynomial functions, how they are

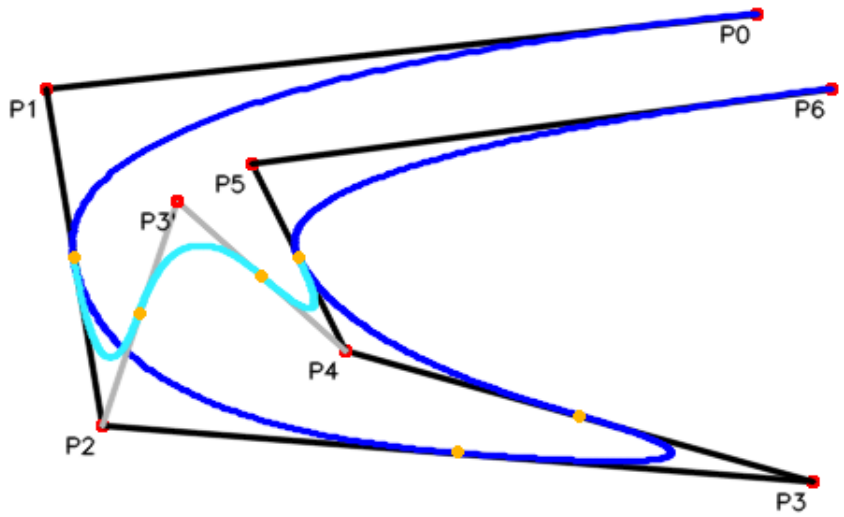


Figure 3.7: Moving $\mathbf{P_3}$ to $\mathbf{P'_3}$ changes the quadratic curve in the interval $[u_3, u_6)$ from the blue one to the turquoise one.

calculated from a knot vector, and what their properties are. Then we used those basis functions to construct piecewise polynomial curves that lie inside the convex hull of a control polygon. Finally we noticed, that changing a single point of the control polygon changes the B-spline curve only in a locally restricted part.

4 Reshaping pixel cells

The approach described in this chapter has been proposed by Kopf and Lischinski in [18]. We take a raster graphic input image and prepare it, so that B-spline curves discussed in a previous chapter can be extracted from it. Therefor, we have to detect edges separating pixels of different color and figure out how these edges are connected. Our first step is then to determine regions of connected pixels in the image, which boils down to comparing the color of one pixel to all of adjacent ones. When looking at all eight possibly connected neighbors, due to the quadratic nature of pixels, there arise some problems with diagonal connections in which pixels only share a common vertex. Sometimes two diagonal connections intersect, and it is clear that not both of them can be connected at the same time. We will use three different heuristics to determine which of the intersecting connections will be kept in the final image, and which must be severed. Our result is a similarity graph storing for each pixel which of its eight neighbors is connected to it and which is not. Finally we use the similarity graph to reshape the pixels cells. Diagonally connected pixels which share only a common cornor in the input image, will have a common edge afterwards. This serves the purpose to make the connections better visible, and simultaneously results in a smoother appearance of the image. At the end of this chapter each pixel will be represented by its color and the coordinates of its vertices which, in fact, turns our pixels into filled polygons.

This chapter is structured as follows. Section 4.1 describes the construction of a similarity graph by examining all 8 neighbors of each pixel. Then in Section 4.2 we use different heuristics to resolve ambiguities created by intersecting diagonal connections, before we finally reshape the pixels into polygons in Section 4.3, which enables us to proceed with the extraction of B-spline curves in Chapter 5.

4.1 Creating a similarity graph

Our ultimate goal in this chapter is to be able to extract B-spline curves to have a vector representation of our image. As we know from Chapter 3, spline curves approximate a sequence of control points. We therefor need a coordinate system to define the position of our control points, and we need to determine which points should form a spline curve.

Definition 4.1.1. *Let the input image be of width w and height h. Define a square lattice graph of $(w+1)\cdot(h+1)$ nodes. Then each pixel of the input image is represented by a cell in this graph of width one and height one. We will call this a pixel cell. Each edge in this cell graph separates two pixel cells and initially each vertex lies at the point where four pixel cells meet. The graph has an underlying coordinate system with the origin being at the top-left, and the y-coordinate increasing downwards.*

As we have seen previously, B-spline curves can be used to fit a curve to a sequence of points, which means we first need to find these sequences. There are two possibilities that can be used as a such a sequence. We could use a set of pixels, or rather their position in the image as control points. This might work if our image had only regions of width one. For larger filled regions there exists a better choice than to fit a spline curve through every pixel of the region. Instead of using the pixel positions, we use the endpoints of edges separating two pixels as control points, which means we need to perform some kind of edge detection.

Many edge detection algorithm like the one Canny developed in [5] use a Gaussian filter to reduce noise. Due to the tiny features in pixel art, sometimes consisting of only a single pixel (e.g. the eye of a character), such methods are not suitable for us. The small features would be lost in the filter. Fortunately, we can use a simple method to detect edges in our image. Basically, any two adjacent pixels are separated by an edge if they lie on the same vertical or horizontal line, or by a vertex if they lie on a diagonal line. Of course, these edges are only visible if neighboring pixels are of a different color. Since we only want visible edges, our next step is to determine for each pixel in the input image which of its eight neighbors are connected and which are not. Therefor we construct a similarity graph. Every pixel is a node in the graph, and two nodes are connected if they are neighbors in the image, and of a color that is similar enough. The criteria used for the similarity is taken from the hqx algorithm [29]. We convert the color to the YUV color space and compare the three channels separately. We denote

the channels of the currently examined pixel with an index C and those of the neighbor we compare it to, with an index N. Then two colors are deemed similar if the following three conditions hold:

$$|Y_C - Y_N| \leq 48$$
$$|U_C - U_N| \leq 7$$
$$|V_C - V_N| \leq 6$$

That means, two colors can vary a lot more in the brightness, represented by the Y channel, than in the color components and still count as similar to each other.

To construct the similarity graph we move in scanline order over the input image and compare each pixel to its neighbors. When checking the neighborhood of a pixel P for similar colors, we number the pixels as depicted in Figure 4.1. While processing P it suffices to compare P to neighbors 4 through 7, assuming we start at the top left corner of the image. P will be compared to pixels 1 through 3 while they are processed. Thus, we avoid checking the same pixels multiple times for similar color. Also, we store for each node in the graph the number of similar neighbors it has, which we call its valence. The valence of nodes will be needed in two of the three heuristics in Section 4.2.

0	1	2
3	P	4
5	6	7

Figure 4.1: Numbering of all 8 neighboring pixels to a pixel P

Special care has to be taken when dealing with diagonally connected pixels, since we will have to reshape every cell that has a diagonal connection in the similarity graph later. Also we may find two crossing diagonals in our graph, which means we do not know which pixels should be connected, yet. There are two cases of crossing diagonal connections in our graph. The first being a fully connected 2 x 2 block, i.e. a region of pixels with similar color. Since there will be no visible edges inside such a region, those diagonal connections can be safely

removed without changing the final output.
The second case is a 2 x 2 block which has only diagonal connections and no vertical or horizontal connections. This means we have two crossing lines, essentially an ambiguous state, since we do not know yet which line should be connected in the final result, if one at all. Clear is, at most one diagonal can be kept and be part of a continuous line in the final image, the other has to be severed. We will determine which connection to keep in Section 4.2 and afterwards we use the completed similarity graph to reshape the pixel cells and to get all visible edges in the image.

Figure 4.2 shows an initial similarity graph on top of our input image. The image has been upscaled for better visibility. Similar pixels are connected by blue lines, while two crossing diagonals, that still have to be addressed, are drawn in red.

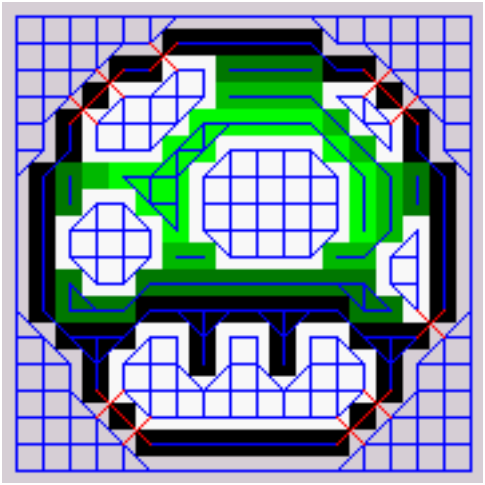


Figure 4.2: Initial similarity graph. Pixels of similar color are connected in blue. Red edges depict diagonals where removing one or the other changes the final output. (Input image ©Nintendo Co., Ltd.)

4.2 Solving ambiguities

The decision which of two crossing diagonal connections to keep requires some thought, since it can not be made locally. Looking at such an ambiguous 2 x 2 block, there are no clues to help in our decision, in the block itself. Because of that, we have to regard a region around each of those connections in question. When considering which diagonal to keep, the criteria we use are mostly derived from the way human perception works.

In our approach, we use three heuristics proposed by Kopf and Lischinski in [18] to rank both intersecting connections. These are the Curves , the Sparse pixels and the Islands heuristics. Each of these criteria will be applied to both diagonals and assign a weight to the connection. The weights each heuristic determines are absolute values and will not be scaled in any way. All three weights are then added up, and the diagonal connection with the higher accumulated weight is kept. Should both aggregated weights be the same, none of the connections will be kept. The following subsections give the details of each of the three heuristics.

4.2.1 The Curves heuristic

The first criteria to be checked is, if two possibly connected pixels lie on a curve, i.e. a long feature that has a width of only one pixel. Is the diagonal part of such a curve we want to keep it. Longer curves have priority over shorter ones. Formally, a curve is a sequence of connected edges in the similarity graph, where each node has a valence of two. That means, there can be no junctions on a curve, and each curve starts and ends either at a node of valence one, or of valence greater than two. The only exception is a closed curve that returns to its beginning.

For the Curves heuristic we calculate the length of the curve each diagonal connection is a part of. The length is the number of edges it is made of. Obviously, the minimum length of a curve in this heuristic is one, since the feature contains at least the edge connecting the two pixels of the diagonal. The weight determined by this criteria is the difference in length of both curves and counts towards keeping the connection lying on the longer curve.

Figure 4.3 as an example shows a segment of a similarity graph, where two blocks of size 2 x 2, marked in red, are still unresolved. In both cases the diagonals from the top-left to bottom-right pixel lie on the curve formed by the white pixels. This curve is of length 8. The intersecting connections have both a node of valence one at the lower-left pixel and a node of valence three at the top-right, which means they are not part of a curve and the length assigned to them is 1. Accordingly, the Curves heuristic assigns weight 7 to both diagonal connections lying on the white curve, while the other connections gain no weight from this criteria.

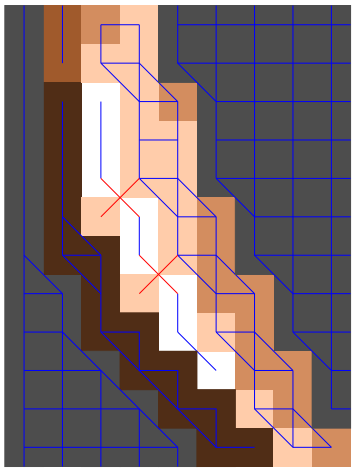


Figure 4.3: Part of a similarity graph with unresolved diagonal connections.The Curve heuristic votes in both cases for keeping the white pixels connected.

4.2.2 The Sparse pixels heuristic

The next heuristic we apply is called Sparse pixels. In human vision when we look at a drawing containing only two colors, we perceive the sparser color as being in the foreground while the other colors builds the background. An example of this would be a dotted line on a white board, where the human brain interprets the sequence of dots to belong together and form a line. Therefor, we want the sparser color to be connected.

While calculating the weight for the Sparse pixels heuristic, we regard only a region of 8 x 8 pixels centered around the diagonal connection in question. This size is big enough to differentiate between foreground and background color, and to contribute a significant impact on the overall weight of a connection. Since it is not possible to determine the quantity of correct decisions made, the actual size of the region used had to be determined experimentally by judging the overall output result. For each of the diagonals we count the number of nodes in the relevant 8 x 8 block of the similarity graph connected to the respective diagonal. That means, we determine the size of the two components containing the diagonals. The resulting weight returned by the Sparse pixels heuristic is then the difference of the component sizes and counts in favor of the connection lying on the smaller component.

Figure 4.4 illustrated the Sparse pixels heuristic. It depicts the 8 x 8 region centered around two intersecting connections. We can see the two compo-

nents, green belonging to the top-left to bottom-right diagonal, and red to the other one. The red component has 13 nodes, while the green region has size 47. Consequently, the weight resulting from this heuristic is 34 in favor of the red connection.

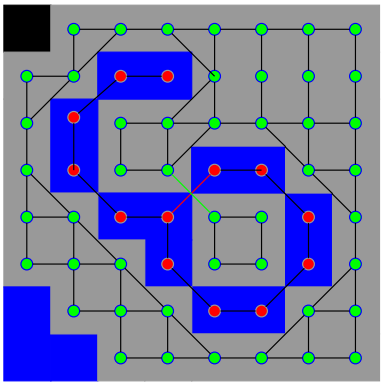


Figure 4.4: Part of a similarity graph with an unresolved diagonal connection.The Sparse pixels heuristic determines the size of the red and the green component and returns the difference of both, in favor of the smaller, in this case the red one.

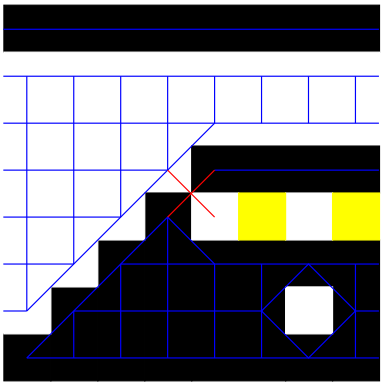


Figure 4.5: Part of a similarity graph with an unresolved diagonal connection. Severing the top-left to bottom-right diagonal would leave a single disconnected pixel. The Island heuristic tries to avoid that.

4.2.3 The Islands heuristic

The Island heuristic is the last to check and it is the simplest of the three. We try to avoid having many small disconnected islands in our image. Thus, we try to keep connections that attach a single pixels to another component of similar color. Therefor, this heuristic checks for a connection in question if one of its

endpoints has valence one. In that case, severing the connection would create an island of a single disconnected pixel. To avoid that a weight of 5 is assigned to such a connection. This weight has been empirically determined in [18] and seems to generate good results.

In Figure 4.5 we can see the Islands heuristic illustrated for an unresolved connection. The white pixel at the bottom right of the ambiguous 2 x 2 block would become disconnected if the top-left to bottom-right of the red diagonals is removed. To avoid this, the Island heuristic assigns weight 5 to this connection.

4.3 Creating Polygons from Pixels

After resolving the problem of intersecting connections in the previous section, our similarity graph is now planar. Since we want to extract B-spline curves from our image, we still need sequences of nodes (and their coordinates) connected by visible edges. As we have discussed before, we want the edges to lie between the pixels and the nodes to be the points at the corners of our pixels. But that is not the only goal here. We also want to reshape the form of the pixels so that a connection between their nodes in the similarity graph means their cells share an edge in the reshaped image. Thus it will be clearly visible if two pixel cells are connected or not. Additionally, by reshaping the cells corresponding to a pixel we reduce staircasing artifacts and consequently get a smoother image.

One possibility to achieve this goal is creating a Voronoi diagram [3], which is a division of a plane into regions determined by the distance to certain objects. But we do not need the accuracy of such a diagram and can instead use a faster method to modify our pixel cells. We identify the shape of each pixel cell in the image by looking at its local neighborhood in the similarity graph, because the eight pixels surrounding a center pixel are the only thing affecting the outline of the new cell.

Since the eight neighbors are all that is needed to determine the new shape of a pixel cell, we can once again move in scanline order through the pixels in the input image, or rather their corresponding nodes in the similarity graph, and reshape each cell sequentially. As before, we do not regard the neighbors with smaller row number than the current pixel, while creating the new shape. This means the upper part of a reshaped cell has already been created while processing the neighbor above and we only take neighbors 3 through 7 (depicted in Figure

4.1) into account for the lower part of the shape. Basically, a cell is only reshaped if a diagonal connection exists in its neighborhood in the similarity graph. If there are no diagonals, a pixel cell already shares a common edge with all neighbors of similar color, and we do not need to change the cell. In all other cases, edges and nodes may need to be inserted into the graph from Definition 4.1.1 and other nodes need to be moved. Figure 4.6 shows 18 cases and their resulting shapes.

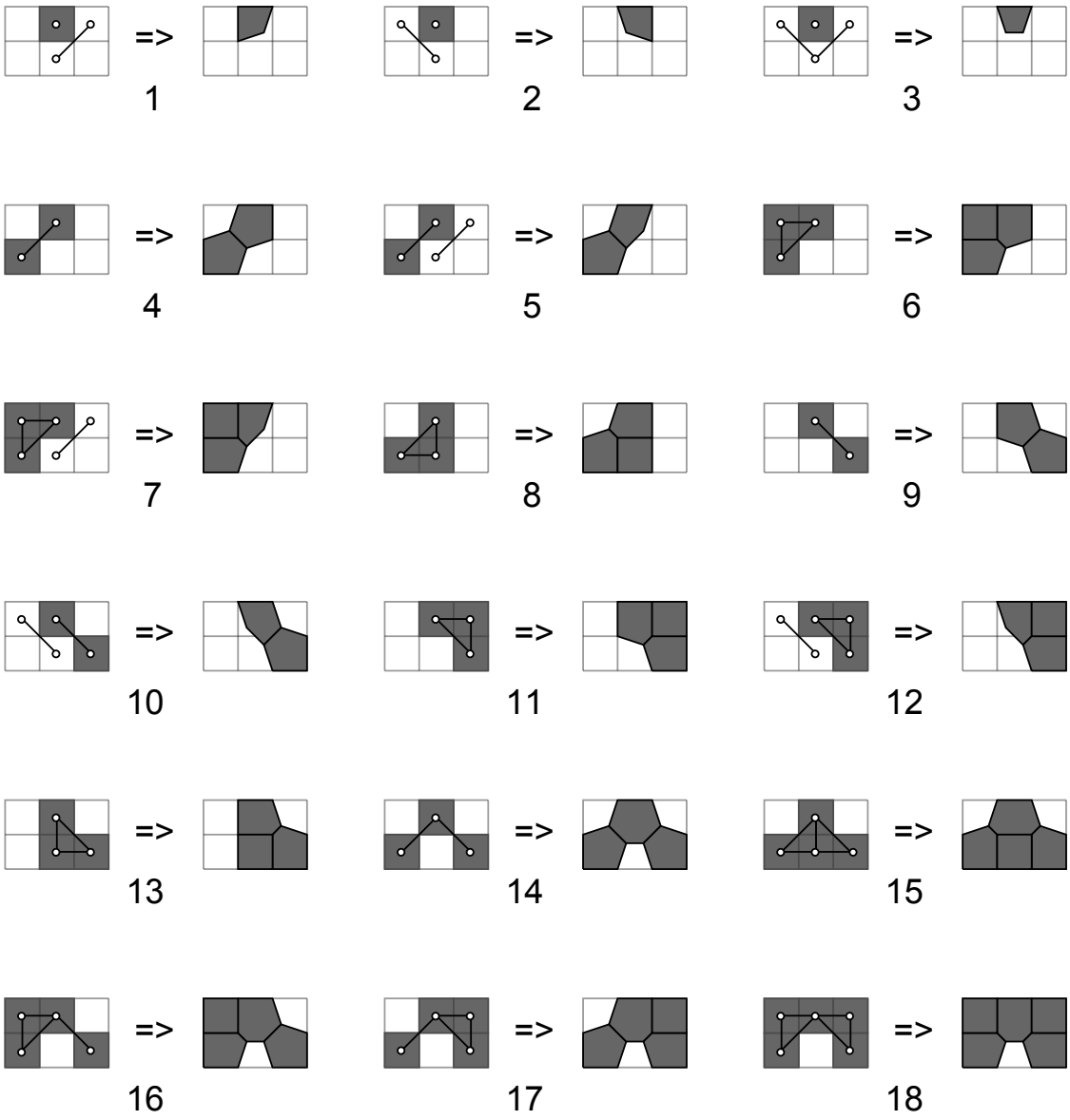


Figure 4.6: 18 cases for creating a polygon from a node in the similarity graph and the distinct cell shapes that result from them. Each case shows a part of the similarity graph over the original pixel cells to the left, and the resulting polygon configuration to the right. The cell to be transformed is always the one in the middle of the top row. The left image of each case shows edges in the similarity graph as black lines.

We will regard a pixel cell P. In each of the depicted cases, P is the cell in the middle of the top row. Its node in the similarity graph and all other relevant nodes are shown as white circles, and connections between them are black lines. P is drawn in gray and all cells of similar color are in the same gray. White areas in the example are of a different color than P, but not all of the white cells are necessarily of the same color. If white cells are of the same color they are connected by an edge in the similarity graph.

Cases 1 through 3 show all cases where P is not connected to another cell, but diagonal connections in its neighborhood require P to be reshaped. Depending on the case, the lower-left or lower-right nodes of P have to be moved. At this stage, a precision of a quarter pixel is enough. Assuming P is the top left pixel of the input image, and we found case 1, the new position of the lower-right node would then be $(0.75, 0.75)$. Case 4 shows the first example of an inserted node and edge. We want P and its lower-left neighbor to share an edge in our reshaped cell graph. To achieve that, we move the lower-left node of P with previous position (x, y) to $(x - 0.25, y - 0.25)$, and a quarter pixel to the top. Afterwards a new node is inserted at coordinates $(x + 0.25, y + 0.25)$ and an edge is placed between those two nodes. Generally, nodes that lie not directly on the underlying pixel grid (drawn in light gray) are only moved by a quarter pixel in both directions. Note that while determining the shape of a cell, we should mark the edges that are visible, i.e. separating the cells fo different color, as we will need that information when extracting the B-spline curves.

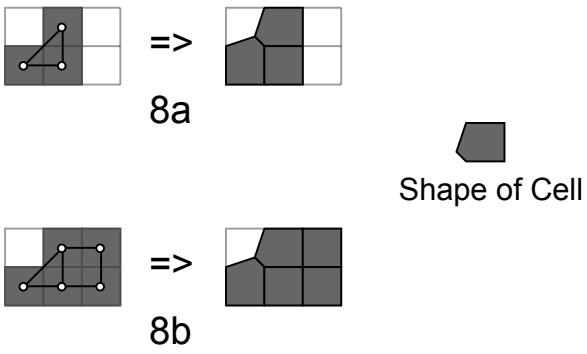


Figure 4.7: Two sub-cases resulting in the same final shape. A side containing no diagonal connections makes no difference to the overall shape.

Figure 4.6 does not show all possible cases. There are some sub-cases resulting in the same shape of the pixel cell and thus we omitted the sub-cases and show only a representative. For example, Case 8 could be divided into cases 8a and 8b. This is shown in Figure 4.7. Note, that the resulting shape stays the same even

though in Case 8b there are two more cells of the same color to the right of P. With these two extra cells, there is a 2 x 2 block with pixels of the same color. Recall from the previous section, that such a block does not contain diagonal connections and since there are no new diagonals, the shape of P will not be affected. The result is, that it makes no difference to the shape of P, if these two cells are of similar color as P or completely different. The only thing that matters for this case is that neighbor cells 4 and 7 are of the same type, i.e. either both similar to P or both dissimilar.

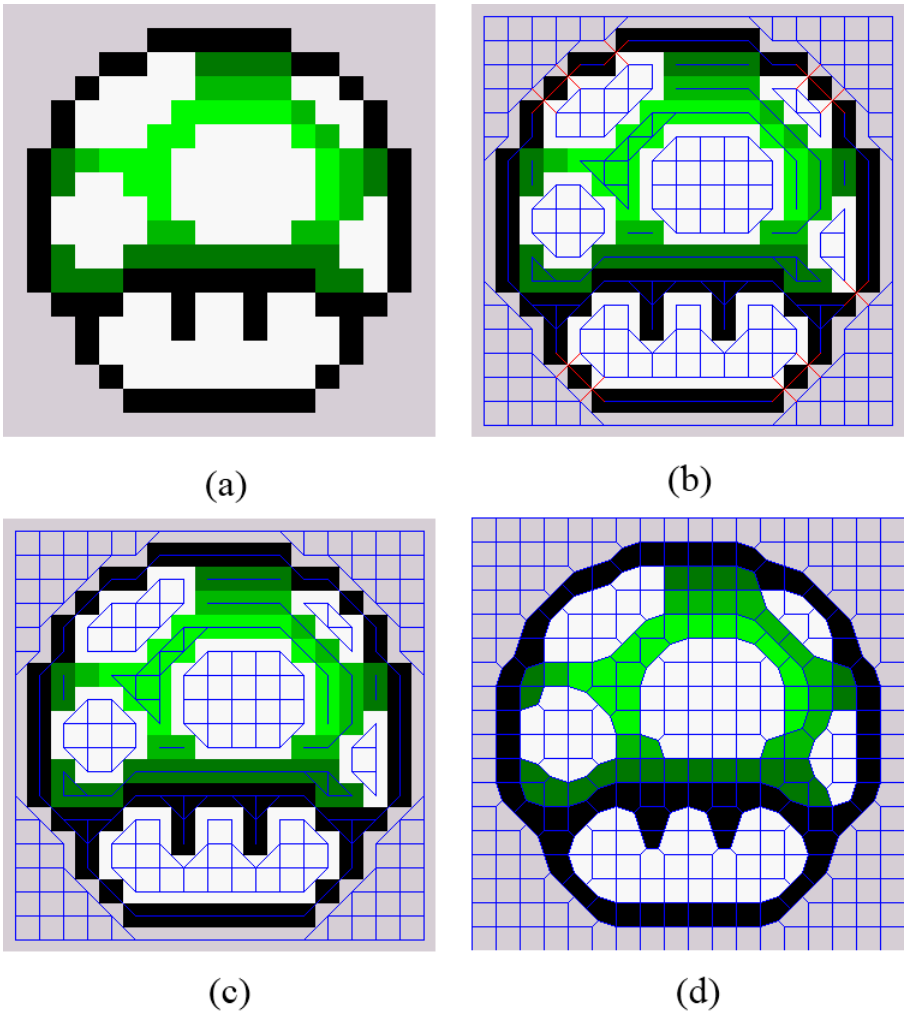


Figure 4.8: Overview of the approach described in this chapter: (a) magnified input image. (b) the similarity graph with still unresolved intersecting diagonal connections in red. (c) similarity graph with ambiguities resolved. (d) reshaped pixel cells according to the similarity graph from (c)

After using Figure 4.6 to determine the shape of each cell, we have a reshaped cell graph containing nodes for the vertices of each cell, edges between them, and

faces with the color of each cell. Figure 4.8(d) shows the result of reshaping the pixels cells on the previous mushroom example. As we can see, the process performed in this chapter already improves the visual quality of the upscaled image quite a bit. The reshaped cells give the overall image a smoother appearance. Technically, this graph representation is already a vector representation, since the cells are polygons and all vertices are at precise coordinates. But our goal is to get an even better appearance by replacing the polygons with B-spline curves. This next step will be described in the following chapter.

5 Extracting Spline Curves

This chapter deals with the extraction of quadratic B-spline curves. The graph created at the end of the previous chapter distinguishes between visible and invisible edges, allowing us to pick the right edges for the extraction. From these visible edges we extract the nodes at each endpoint as the control points of our spline curves. But we still need to decide which sequence of edges a specific spline is to follow. With the control point sequences and the degree of the curve, we finally have the edges of our image represented as B-splines. Afterwards we can still improve on our B-splines by merging some of them. Therefor, we define a criteria that tells us which splines to merge. This leads to an even smoother image which is illustrated in Figure 5.1.

The structure of this chapter is as follows: Section 5.1 details how to extract the control points from the reshaped cell graph and more importantly, which nodes should form a spline curve. Then, in Section 5.2 a criteria is developed that will tell us which splines should be merged and afterwards we act on this criteria.

5.1 Building Control Point Sequences

In Chapter 4 we created a graph of reshaped pixel cells, that now allows us to extract everything we need to construct B-spline curves. Recall from the definition of B-spline curves that all we need to fully define such a curve is a degree p and a sequence of control points. First we have to decide which degree our spline curves should have. The lowest degree yielding any results is of course degree 1, which means the "curve" is a sequence of straight lines connecting the control points. In fact, that is what we already have in our reshaped cell graph without even using spline curves. So it is clear we have to use at least degree 2. Figure 5.2 shows a part of an image using B-spline curves of different degree. We can see that using curves of degree 2 already results in a rather smooth appearance. While a higher

degree still improves on this result a little, the difference is not that significant but needs more time to compute, since the basis functions of higher degree have to be computed. Since we will further smooth our spline curves in Chapter 6, we decide using degree 2 is good enough for our purpose.

Figure 5.1: Overview of the approach described in this chapter. (a) Part of an image after the cell graph has been created. (b) Image after splines have been extracted. (c) Output after merging splines at junctions.

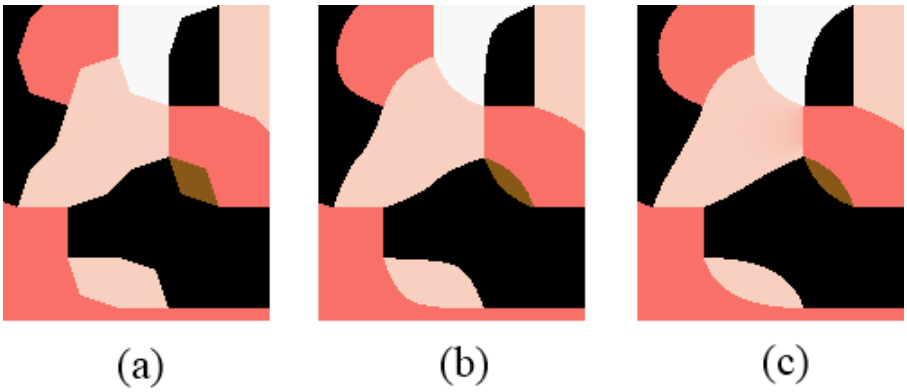


Figure 5.2: Difference between degree of B-spline curves. (a) Degree 1. (b) Degree 2. (c) Degree 3.

Next we have to create sequences of control points that are supposed to form a spline curve together. Our approach here is quite simple. We pick a visible edge from the reshaped cell graph and take the coordinates of the two nodes at the ends of this edge, as the positions of our control points. Then we check the valence of these nodes. Note that right now we are only interested in visible edges, so only those edges count for the valence of a node. If the valence of a node is 2 it means that just one other visible edge is connected to it. Since it does not make much sense to keep these edges in a separate spline, we add the

coordinates of the additional node and add it to our control points. We continue to do so, until we reach a node that has a valence not equal to 2 or we reach one of the nodes of our starting edge. Obviously, if a node has only one visible edge, we can not connect any more edges. If the valence is 3 or higher we do not know, yet, with which edge to continue. Consequently, we start a new spline at that node. We proceed in this way until every visible edge, or rather the coordinates of both nodes it connects, have been added to a spline. So to sum up, we convert sequences of edges containing only nodes of valence 2 into sequences of control points. Each such sequence starts and end with a node with valence not equal to 2, unless we found a sequence of edges returning to its starting point.

With those sequences of control points we just determined, we now have our quadratic B-spline curves fully defined. Since our splines are defined in a range of $[0, 1]$, we can now traverse our splines from beginning to end by iterating from 0 to 1 and calculating the resulting point on the curve with Equation 3.3. While, we now have converted all visible edges of our input image into spline curves, right now we only have the shape of the splines without color information. Since our curves separate two differently colored parts of the image, it makes sense to store for each spline segment (from one control point to the next) the color at both sides of the curve. We can easily do this while constructing the sequence of control points. As we will see in Chapter 7 the B-spline curves and color information at both sides will be everything we need to draw our finished vector graphic image. This means with our control point sequences and their respective color information we have completely described all edges in a mathematical representation. Since Elder et al. [8] have shown that it is possible to reconstruct an image from an edge representation, our spline curves are all we need to represent the image. If we want to store the result as a vector graphic representation, this is all we need to save to disc. But we are not quite finished, yet. We still want to visually improve the result. The first step to do that is merging splines, which is described in the next section.

5.2 Merging Splines

After we created the sequences of control points and thus finally arrived at a vector representation of the input image, we want to improve the output. The first step in that direction is merging splines by simply appending one sequence of control points to another. For this to give good results we need to establish

a criteria that tells us which splines to merge. The only locations where we can merge splines is at nodes of valence greater than 2. Therefor we regard nodes of valence 3 and have look at the meeting edges.

Kopf and Lischinski [18] suggest to check if one of the edges is a shading edge. This is a edge that separates regions of only slightly different color, that was nonetheless different enough to be still classified as visible. To check if an edge is a shading edge, we once again compare the YUV channels of the regions to both its sides. If the following condition is met, the edge is called a shading edge, otherwise it is a contour edge.

$$|Y_{region1} - Y_{region2}| + |U_{region1} - U_{region2}| + |V_{region1} - V_{region2}| \leq 100$$

Since contour edges are what defines a region, while shading edges only divide areas of slightly different color, we want to connect contour edges. The first criteria for merging splines is then: If there are one shading and two contour edges present at a junction, merge the contour edges. Should this criteria not decide which splines to merge, we determine the angle between the three edges. Those two edges that form the angle closest to 180 degree will be connected. Figure 5.3 illustrates these two conditions. The images show the result after merging two splines. As we can see, three spline curves meet at the orange control points. We then determine which of the these splines should be merged using the method described above. Then the merging process is very simple. Since in our implementation we store a sequence of control points for each spline, alls we do to merge two splines is add one sequence of points to the other. The resulting combined sequence then represents the new merged spline curve.

One thing has to be taken into consideration when merging splines. We know that splines only approximate their control points, which means they do not generally pass through them, unless it is the first or last control point of the spline. Thus, after merging two splines the curve will no longer go through the node of valence 3. Because of that it is necessary to move the control point of the spline that has not been merged, so that is lies directly on the path of the other curve.

How the process of merging splines at junctions improves the smoothness of an image is illustrated in Figure 5.1. The first image shows our input at the beginning of the chapter, i.e. the reshaped cell graph. Next we see the image after the edges have been converted to quadratic B-spline curves. The final image depicts the effect of merging splines at junctions. Following the splines between

Figure 5.3: Merging two splines at a junction. (a) The edge between the green regions is a shading edge, therefor the other splines are connected. (b) No shading edge is present. The two edges along the red region meet at the angle closest to 180 degrees and their splines are merged.

the white and green regions illustrates the change best. Merging clearly reduces sharp bends along this curve.

6 Smoothing the Curves

After extracting spline curves in the previous chapter, we arrived at an image that is already a lot smoother than the result from our reshaped cell graph. However, there are still staircasing artifacts present, resulting from the fact that a pixel is a relatively large unit in the low resolution of the input image. This can be seen in Figure 6.1(b) on the outline of the ghost, which clearly exhibits many inflection points. In this chapter we smooth our spline curves by manipulating the position of their control points in order to reduce such behavior.

If we want to smooth B-spline curves we first need to determine a way to measure smoothness. Therefor we use energy functions in this chapter that have been proposed in [18]. The first defines the smoothness as follows:

$$E_s^{(i)} = \int_{s \in r(i)} |\kappa(s)| ds$$

where $E_s^{(i)}$ is the smoothness energy value for a control point p_i, $r(i)$ is the region of the curve that is affected by p_i according to Property 3.2.5 and $\kappa(s)$ is the curvature at point s. Generally the curvature of a curve is a measure of its deviation from a straight line. If a curve is given by parametric equations as $x = x(t)$ and $y = y(t)$ then the curvature can be calculated as follows [6]:

$$\kappa = \frac{x'y'' - y'x''}{(x'^2 + y'^2)^{3/2}}$$

We compute the integral numerically by sampling the curve at fixed intervals and using the extended 2-point Newton-Cotes-Formula [1] given as:

$$\int_{x_1}^{x_n} f(x)dx = h(\frac{1}{2}f_1 + f_2 + f_3 + \ldots + f_{n-2} + f_{n-1} + \frac{1}{2}f_n)$$

where $f_i = f(x_i)$ and h is the distance between the sampling points.

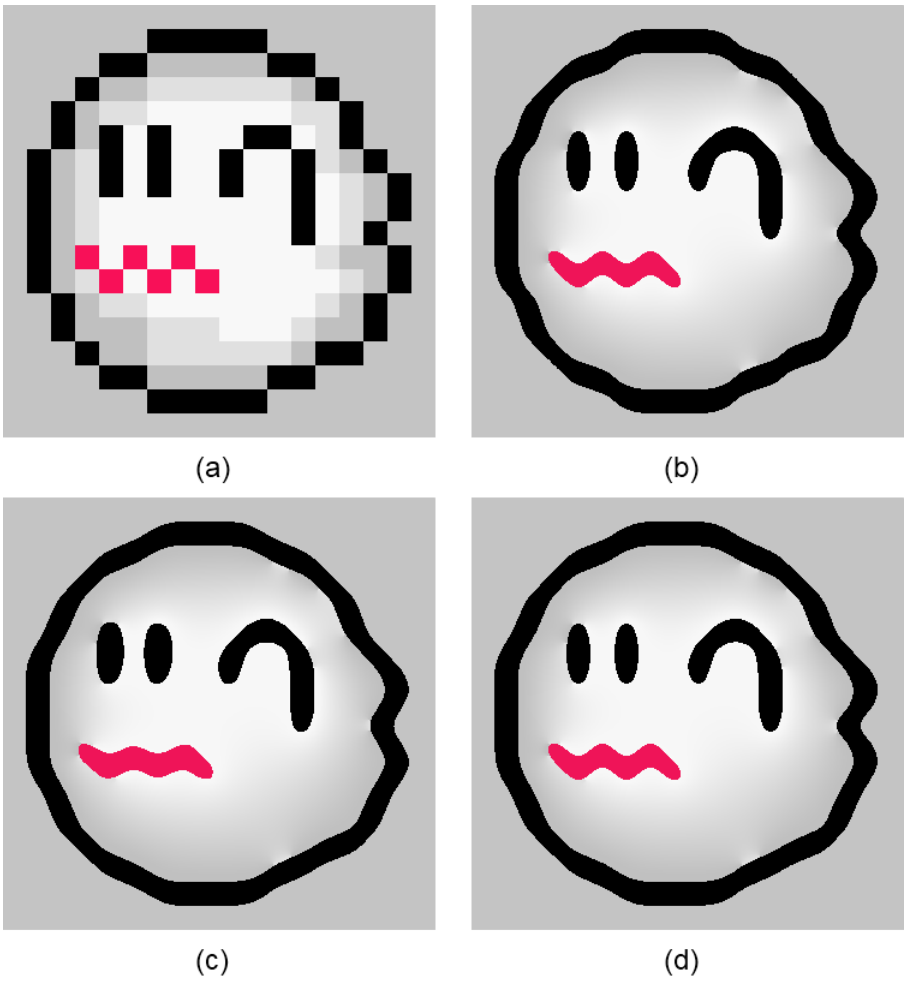


Figure 6.1: Smoothing the B-spline curves. (a)Input image (magnified). (b)Image after fitting spline curves. (c) Result after smoothing the splines without pattern detection. (d) Result after smoothing the spline curves with pattern detection.(Original image ©Nintendo Co., Ltd.)

With this we could now optimize our control point locations to minimize E_s and thus the curvature, which in turn means making our spline curves "straighter". We want to do this only to some extent, while mostly keeping to the initial shape of the spline. Because of that we use a positional energy term E_p in addition to E_s, defined as follows:

$$E_p^{(i)} = ||p_i - \hat{p}_i||^4$$

where p_i is the location of the i-th control point and $\hat{p}_i$ is its initial location. With this term we penalize larger deviations from the initial position of the control point. We then combine these energy terms to calculate the total energy

of a node as

$$E^{(i)} = E_p^{(i)} + E_s^{(i)}.$$

To smooth the B-spline curves we attempt to minimize this energy term locally for each control point. We do that by iterating over all control points and trying several new locations for each node before moving to the next. The new locations for each node are generated by randomly offsetting the current node location in both directions. Should the new position result in a smaller energy than the previous one, we keep that location. In our implementation we used a sixteenth of a pixel as the maximum offset, try 10 random positions per iteration and perform 15 iterations over every control point. Tests have shown that node locations hardly change afterwards. The result of this smoothing process can be seen in Figure 6.1(c). While the outline of the figure has been smoothed as intended, the eyes and mouth are now distorted making them look worse than before. Clearly there are configurations of control points we want to keep as they were before. Kopf and Lischinski [18] suggest 5 such patterns that should remain unchanged. They are shown in Figure 6.2(a). Three of these five patterns can be found in our image, as depicted in Figure 6.2(b).

Of course we also need to detect rotations and reflections of these patterns. The fact that all pixel positions prior to smoothing are quantized to multiples of a quarter pixel makes recognition a lot easier. In our implementation we define each pattern by the angles formed between its lines. These angles stay the same for their rotations and reflections, except for the reflection of the second pattern along a vertical line. We introduce this reflection as a separate sample. To check if a pattern is present on a spline we take the first four points, calculate the angles they form between their lines and compare them to those of our patterns. Then we swap the first of our four points with the next one on the spline and repeat the process. Is a pattern found, we exclude the control points forming that configuration from the smoothing process. The fifth pattern is a special case, since it is formed by five points. When checking for this pattern we compare to its first two angles and only if those match, take the next angle into account. After this detection process has been performed we apply our smoothing process only on those control points not contained in one of those patterns. The result of this can be seen in Figure 6.1(d).

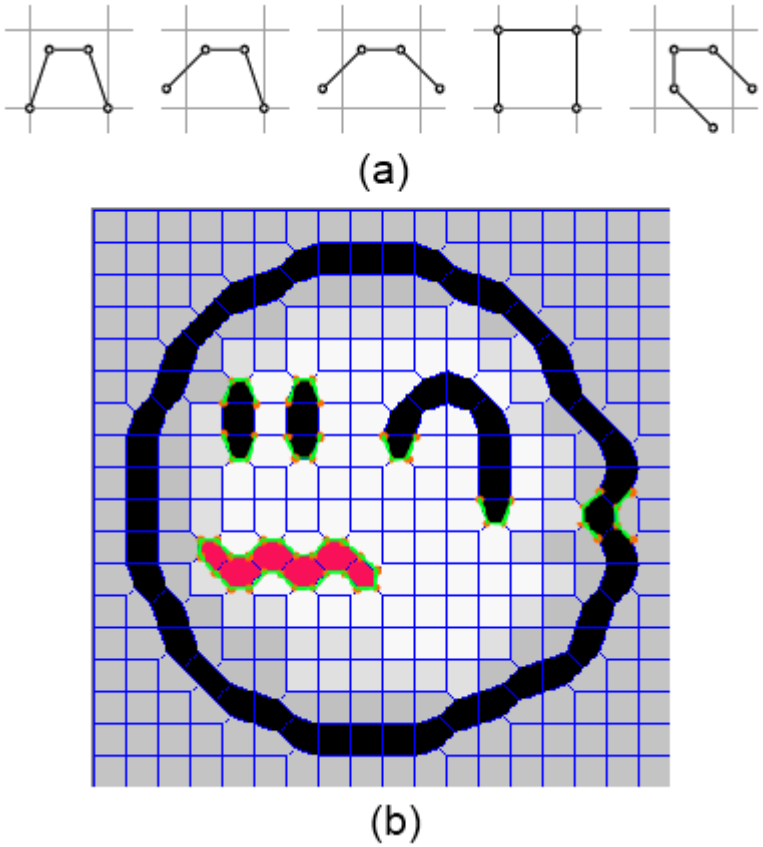


Figure 6.2: (a) Patterns that should remain unchanged by smoothing. The original pixel grid is shown in gray. (Taken from [18]) (b)Detected patterns marked in green.

7 Rendering

In this chapter we render the finished image on our screen. Unfortunately we can not simply use a common vector graphics format, since the open standard do not support spline curves. Thus, we need to find other means to render the image. Kopf and Lischinski [18] used a method in their approach which they describe as "slow but simple". In it they use truncated Gaussian influence functions at the center of each cell to compute the color of each pixel of the image. Nehab and Hoppe [24] use a different method that could be used to render our image, but to use it we first would have to figure out which splines form closed curves together. Instead we follow a different approach proposed by Jeschke et al. [15]. It is well suited for the vector graphics representation we already have and gives an initial guess of the color of each pixel by determining for each point in the image which curve is the closest. Afterwards the color of this guess will be diffused to create a smoothly shaded region.

In Section 7.1 we see how to recreate a full image from just our B-spline curves, which involves determining the color in between the curves. We then use color diffusion on this image to produce the finished output. Section 7.2 gives a description of this process.

7.1 Curve Rasterization

At the end of Chapter 5 we arrived at a representation of our image elements that is similar to the diffusion curves described by Orzan et al. [25], with which it is possible to fully construct our output image. As we know the usual display devices like computer screens are all raster devices which makes it necessary to transform a vector graphic image back to a raster representation in order for it to be displayed.

The approach we are going to use has been developed by Jeschke et al. [15]. It consists of two parts, the rasterization and the diffusion. The rasterization process builds on the work of Hoff et al. [13]. Recall that at this moment we still have only our quadratic B-spline curves and color values to each side of the curves, meaning we have a representation similar to the one depicted in Figure 7.1. We do not really know yet how to color the areas between spline curves. The basic idea is that every pixel of our result should be in the color of the appropriate side of the spline curve that is closest to the pixel. To do that we usually would have to determine the distance to all curves for every pixel in the image. In this approach however, we make use of a feature of the GPU called the depth buffer. When rendering a 3 dimensional scene the depth buffer determines for all pixels of the image the object that is closest to the position of the viewer and displays the pixel in the color of that object. To make use of this we have to create objects in the 3 dimensional space that represent the distance to a spline curve.

Figure 7.1: Diffusion curve representation of an image. Color in the gray area is not yet known and still has to be determined.(Image taken from [15])

We illustrate the concept of this approach by a simple example. Assume we want to color the area between a red and a blue line so that each pixel between the lines has the same color as the nearest line. We then create a rectangular polygon, called a quadrilateral or short quad, for each line and place it as follows. We start the rectangle directly at our line in the image plane. Then the side of this rectangle that is parallel to the line has to be moved further away from the image plane, since we want greater distance from our line to be translated into greater distance from the image plane. We define the image plane to be at depth 0, while the rectangle ends at depth 1. Then we do the same for the other line in our image. Figure 7.2 illustrates this process. Now we have represented each line by a polygon in the 3 dimensional space. The depth buffer now determines for

every pixel in the image which object (in our case quad) has the least depth. The dashed black line marks the position where the polygons intersect, which means starting form the red line in the image plane, the red polygon has the least depth until we reach the dotted line, then the blue quad has less depth. Accordingly, the area to the left of the dotted line will be colored red and the area to the right will be colored in blue.

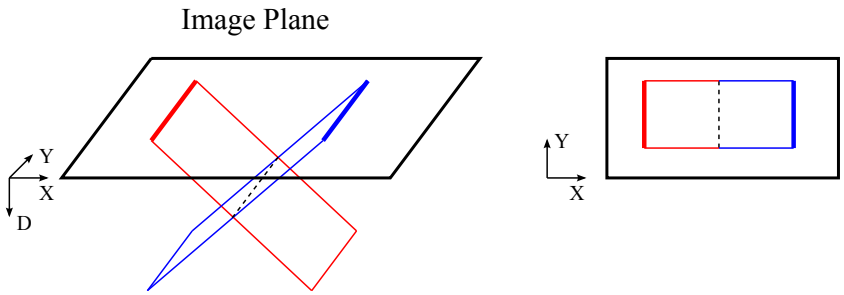


Figure 7.2: (Left) Distance from two lines represented by two rectangular polygons. The further away from a line the ploygon gets the greater its depth in the image. (Right) Top-view of the left image.

Now we want to use the same principle for our spline curves. Therefor we first divide our curves into a number of linear segments. Since we want to determine the color on both sides of the curve, we have to create two quads, each in the same color as the appropriate side of the spline. We already know that each quad starts at a line segment with depth 0 and ends at depth 1. In the projection on the image plane (i.e. the top-view of the image), the length of the polygon is s, which we choose to be the same as the length of the diagonal of our input image. Like that we can ensure that each polygon is capable of covering the whole image. Since our linear segments approximate a curve, there is an angle between connected lines that is not yet covered by the quads we constructed. To fill these gaps we construct a fan of triangles at the outer side of the curve. The triangle starts at depth 0 at the point where two segments meet and reaches depth 1 in a distance of s on the image plane. We construct each triangle so that the maximum angle at the junction point is 45 degrees. If the gap is larger than that we combine more triangles to cover the gap. Last we have to cover the endpoints of the curves. Here we use triangle fans with an angle of 10 degrees to form an end cap covering the whole 180 degrees. These triangle fans at the end of a curve do not need to be constructed if the endpoint is at a t-junction between three spline curves. Figure 7.3 shows the result of this process performed on a curve, though the distance s has been reduced in the image for better visibility.

After repeating the process, described above, for every spline curve we now

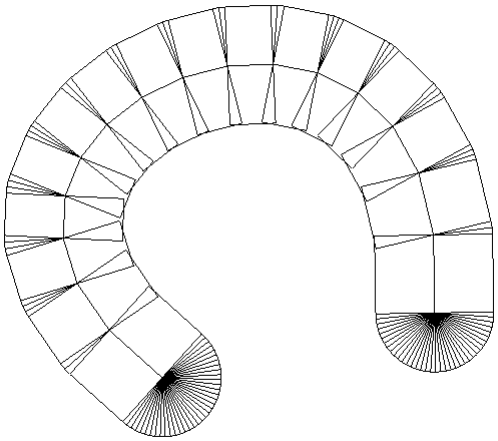


Figure 7.3: Result of constructing quads and triangles along a curve that has been divided into 16 segments. (Image taken from [13])

render the image using an orthogonal projection, looking along the depth axis. The color of each pixel in the image is now calculated automatically in the depth buffer of the GPU. In the case where there is only a single color between two spline curves, we now have our final output. Figure 7.4 shows an image that still needs diffusion. Points that are equidistant to two splines are clearly visible as sharp edges inside the ghost.

7.2 Color Diffusion

The second part of our rendering method is diffusing the colors in the image to smoothly blend the colors inside regions. Jeschke et al. [15] suggest a "Jacobi solver that iterates towards the solution by setting each pixel to the average of its direct neighbors":

$$G(x,y) = \frac{1}{4}\sum_{i=1}^{4} G(n_i) \tag{7.1}$$

where (x, y) is the position of a pixel and n_i are the pixels in the 4-connected neighborhood of (x, y).

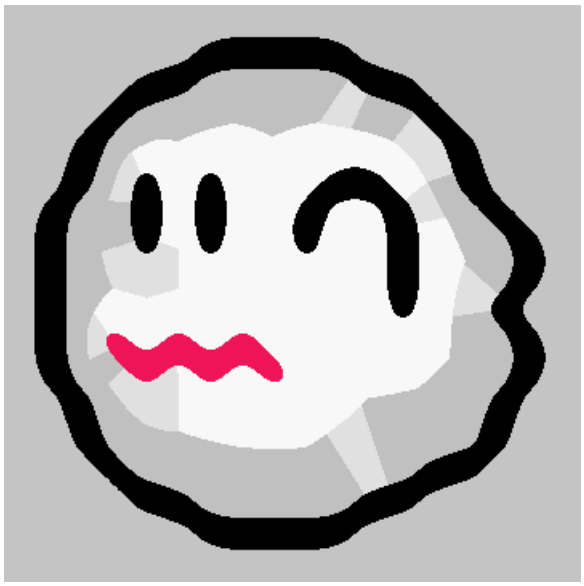


Figure 7.4: Output after rasterization process. (Original image ©Nintendo Co., Ltd.)

Since we want our B-spline curves to clearly separate differently colored regions from each other, we have to prevent diffusion across spline curves. We achieve this by fixing the color value near the splines (i.e. the color of pixels directly adjacent to a curve stays constant). When using this method the color is diffused quite slowly. Figure 7.5(a) and (b) show the image after 8 and 50 iterations. The edges inside the gray region of the ghost are still clearly visible. The main reason is that it takes color many iterations to travel from one part of the image to another. However Jeschke et al. improve further on their approach by increasing the distance of the neighboring pixels used to calculate the new color value. We set the new diffusion radius of our solver to the largest value that still keeps us from crossing any spline curves, which is of course exactly the distance of a pixel to the closest curve. This bigger radius lets color travel much quicker to other parts of an image, thus reducing the number of iterations. However, this introduces a new problem. Banding artifacts can occur because the color of a pixel has net been properly diffused in its local neighborhood. These banding artifacts show as straight lines through an otherwise smooth color gradient. An example of this can be seen in Figure 7.5(c) to the left of the eyes. The effect in this image is quite minor, so we might need to zoom into the image in the PDF file to see it clearly. A solution to that problem is shrinking the radius with each iteration, so that in the first step we have the radius equal to the distance to the nearest curve, while using a more local neighborhood in later iterations. We shrink the radius r before each iteration i as follows:

$$r = r * \left(1 - \frac{i}{n}\right)$$

where n is the total number of iterations performed. Note that we start with i at 0 and iterate until $i = n - 1$, otherwise our radius would be zero in the last iteration. Using this method of shrinking produces good results with as little as 8 iterations. An example is shown Figure 7.5(d).

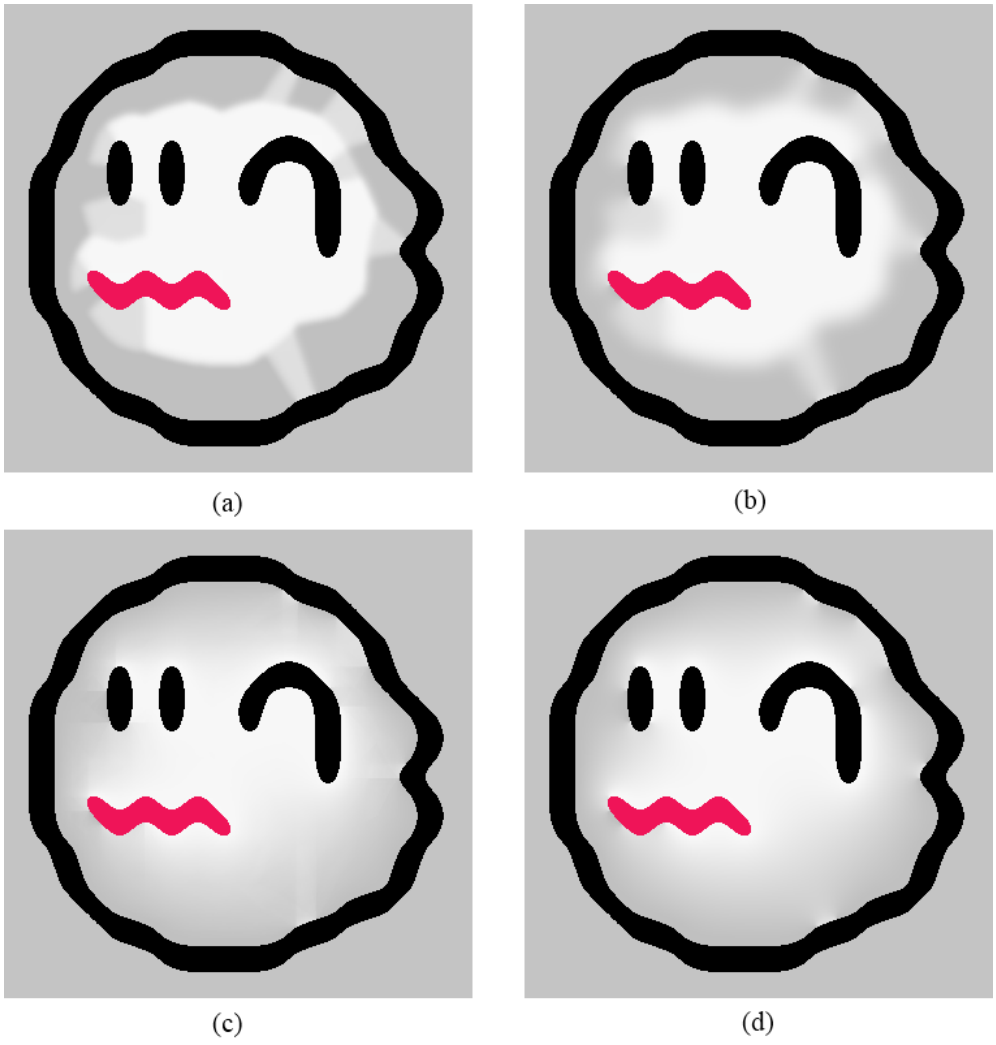


Figure 7.5: Output after diffusion. (a) Standard Jacobi solver after 8 iterations. (b) Standard Jacobi solver after 50 iterations. (c) Solver with radius equal to distance nearest curve after 8 iterations. (d) Same as (c) but with radius reduced in each iteration to reduce banding artifacts.

8 Evaluation

In this chapter we have a look at the results of our approach. We evaluate the runtime of different sections of the algorithm, and compare the output images to the results of other upscaling and vectorization methods. Generally our approach produces images superior to those of other vectorization tools. When compared to other upscaling methods, the specialized algorithms for pixel art perform well on magnifications up to factor 4 but fall behind when increasing the upscaling factor.

Since our main goal was to get a resolution independent representation of a pixel art input image, we first present one result at different magnifications. Our implementation, which is included with this thesis, offers the ability to directly set the upscaling factor of the output. The result will be displayed in an OpenGL window, that can be resized by the user at any time. The resized result will be displayed immediately. Figure 8.1 illustrates the upscaling capabilities of our algorithm. We can see that upscaling works well, which was expected since displaying in a higher resolution simply means multiplying the coordinates of the B-spline curves.

Now, that we confirmed that our approach is working as intended, we want to know how long it takes to convert a raster image into a vector image. To determine the runtime we had our algorithm vectorize a variety of 55 images, ranging from game characters to old windows icons and determined minimum, maximum, average and median values for the conversion. The algorithm was executed on a single core of a 3.2 Ghz CPU. The runtime of the algorithm depends most of all on the number of splines and the number of control points that are extracted from an image, and thus on the size of the input, since larger images usually result in more control points curves. The exact numbers are listed in Table 8.1. Note that the longest time is spent on optimizing the control point positions. The reason for that are the many different points that are tried at random. Each new point requires a new calculation of the spline and its derivatives, which in turn will be evaluated at multiple positions. Since we do that for every spline, this amounts

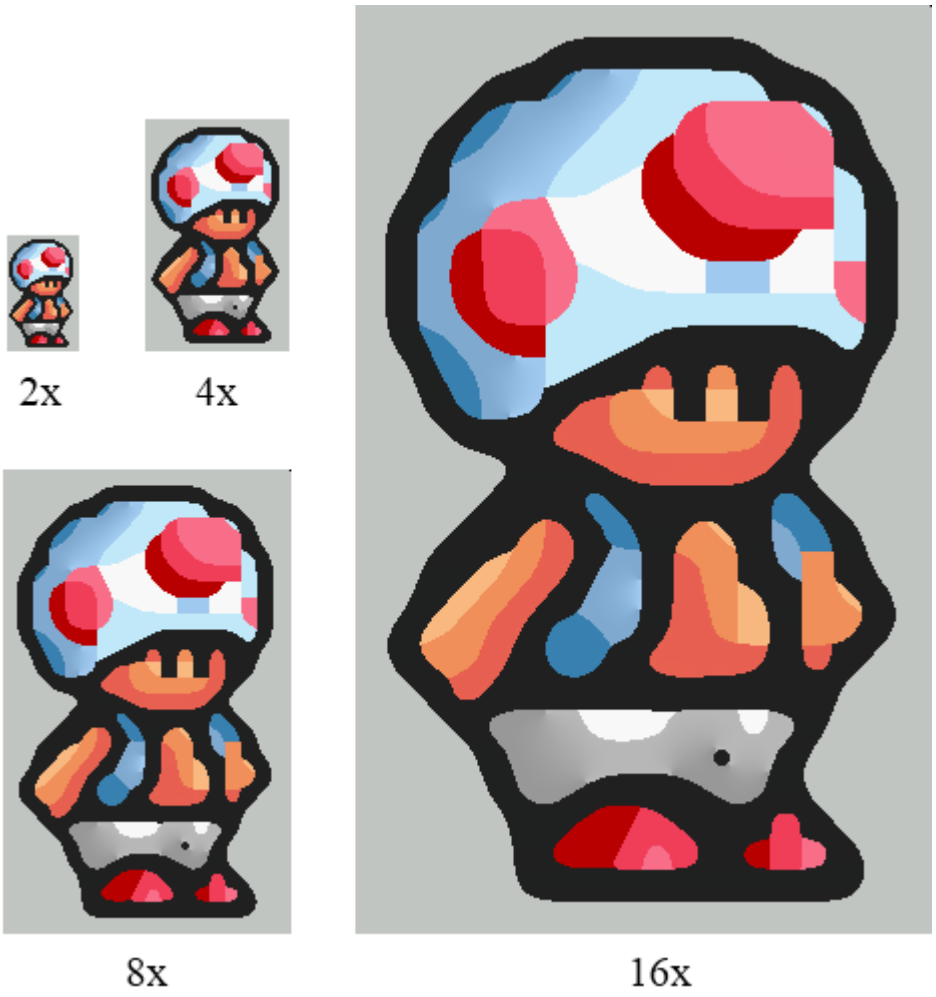


Figure 8.1: Result of an input image at different upscaling factors.(Original image ©Nintendo Co., Ltd.)

to several hundred thousand recalculations of spline curves. Considering that without this step our algorithm is near the performance for real time conversion, we could argue that the smoothing process takes too much time for too little improvement. Thus, our implementation has the option to skip the optimization of the control points. The runtime of the other components of our approach is almost negligible, the exception being the diffusion process while rendering. This step is also the only one where the runtime increases with the size of the output image, since diffusion is performed on every pixel of the result. While determining the runtime of the algorithm we used an upscaling factor of 10.

Since the smoothing process is the one taking the most time we analyze its behavior for different images. We note for each image the number of splines and the number of control points and compare the time it takes the smoothing

	Median	Average	Minimum	Maximum
Similarity Graph Construction	0.00s	0.00s	0.00s	0.02s
Reshaping of pixel cells	0.00s	0.00s	0.00s	0.02s
Spline Extraction	0.00s	0.01s	0.00s	0.03s
Spline Smoothing	0.42s	0.45s	0.02s	1.19s
Color Diffusion	0.03s	0.03s	0.02s	0.09s
Total	0.45s	0.49s	0.03s	1.34s

Table 8.1: Runtime of our approach, split into different parts of the algorithm

process to perform different number of iterations. The results can be seen in Table 8.2. One thing we notice from the results is that the smoothing time increases linearly with the number of iterations. An interesting fact is that smoothing time decreases with the number of splines in the image, as long as the number of control points stays constant. This means smoothing is performed a lot faster on many small splines than on few large splines. This can be seen when comparing the data for "Bomber" and "Dolphin". Both have about the same number of control points, but "Bomber" has about 4 times as many splines. The result is that the smoothing time for "Bomber" is about half the time taken for "Dolphin". A part of the faster runtime can be explained by the fact that we hold the endpoints of a spline in place, which reduces the effective number of control points that have to be smoothed. One other thing we take from this table is that smoothing in its current form is certainly not feasible in an emulator environment. Both "Chrono" and "Zelda" are whole frames taken from an emulator and the smoothing time is several seconds for each, which is much too long for the use in an emulator.

			Smoothing Time		
	#Splines	#C-Points	1 iteration	5 iterations	15 iterations
Invader	4	136	0.01s	0.01s	0.03s
Disk	10	581	0.03s	0.11s	0.35s
Bomber	71	934	0.02s	0.09s	0.26s
Dolphin	17	972	0.04s	0.17s	0.51s
Grandpa	196	2093	0.04s	0.18s	0.53s
Skeleton	152	2658	0.07s	0.33s	0.98s
Chrono	8042	125435	2.28s	11.04s	32.77s
Zelda	3546	57467	1.19s	5.79s	17.17s

Table 8.2: Comparison of the number of splines and control points of different images and the time the spline smoothing process took for different number of iterations.

Next we compare our results with those achieved by other approaches. Figure

8.2 shows a comparison with other upscaling methods. The upscale factors in these images vary between 4 and 8 and are thus at or near the value other algorithms like hq4x were intended for. Nonetheless, we achieve results that are at least on par with those of the specialized pixel art upscaling algorithms. Looking at the results of the "386" image, we notice that Scale2x has problems solving the ambiguities in the figure eight, while our heuristic does quite well. On the "Boo" input image we gain an advantage over the hq4x result because of our approaches ability to smoothly blend the colors inside the ghost.

In Figure 8.3 we see our results next to those of other vectorization tools and approaches. While the specialized pixel art upscaling algorithms perform generally well, most vector graphics tools have severe problems when dealing with the small features and low resolutions of pixel art. Most results tend to be too round, as can be seen with the "Sword" and "Keyboard" input, or they do not resolve ambiguities correctly, which can be seen at the cord of the "Keyboard" result. Our approach does the cord really well, but it also suffers a little from the rounding problem. The keys of the keyboard should clearly be rectangular, but become ovoids in our algorithm. But this is a minor problem. Generally our results look a lot better than those of other vectorization tools.

Finally we tested our algorithm on a larger example than a single game character sprite. Therefor we took a whole frame from an emulator and used our approach to vectorize this frame and increase its size by a factor of 4. We compare our result to that of the hq4x algorithm and a simple nearest neighbor upscaling in Figure 8.4. In this comparison the hq4x result is probably the best, mainly because our algorithm looses small details in the wood of the table and the stakes. This is due to the similarity in the brown pixels which causes their color to be diffused and not separated, resulting in a blurred appearance of the wood. Otherwise our algorithm's result looks quite pleasing.

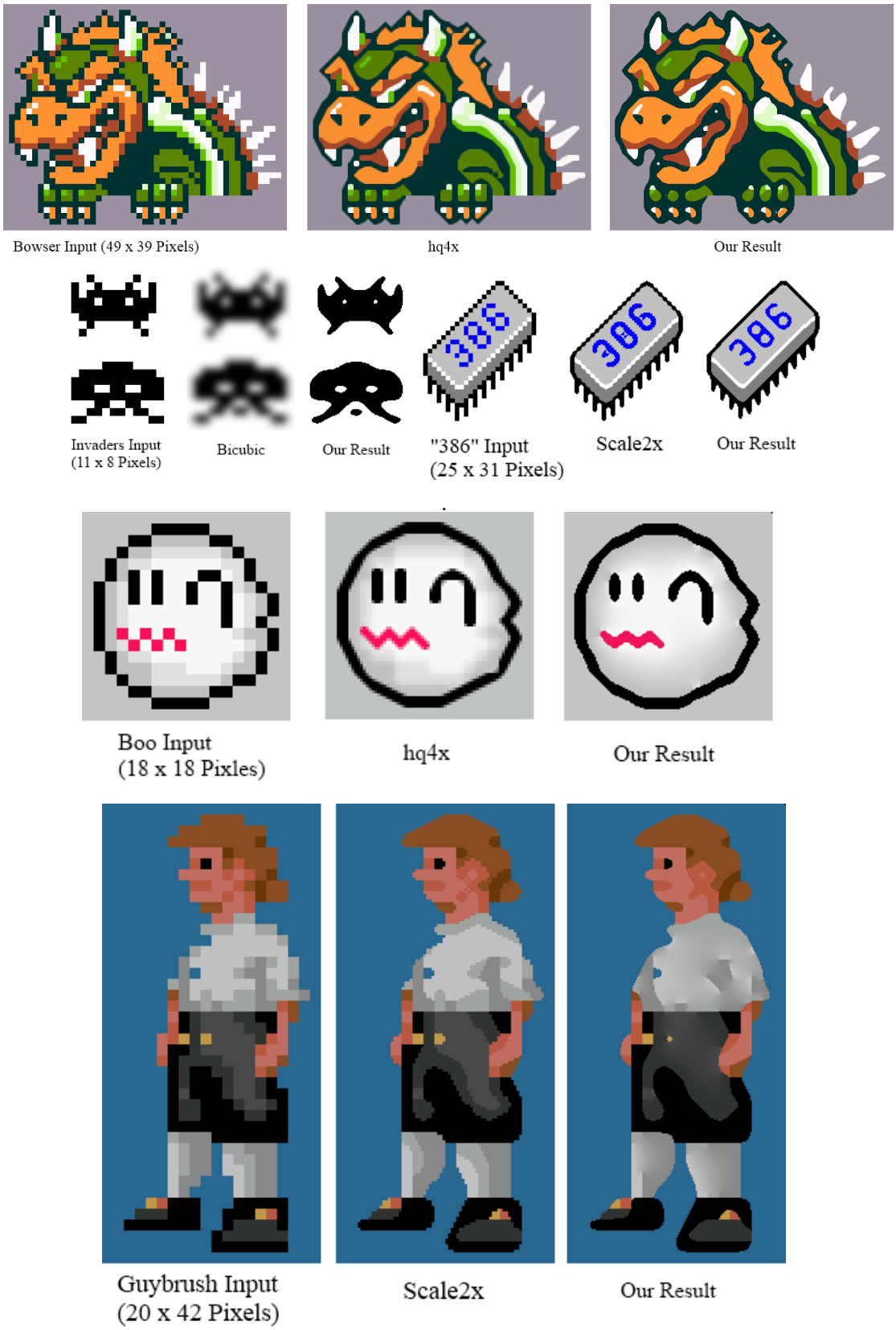


Figure 8.2: Comparison of some of our results with those of other upscaling methods. (Images: Bowser, Boo ©Nintendo Co., Ltd.; 386 ©Microsoft Corp.; Invaders ©Taito Corp.; Guybrush ©LucasArts, LLC)

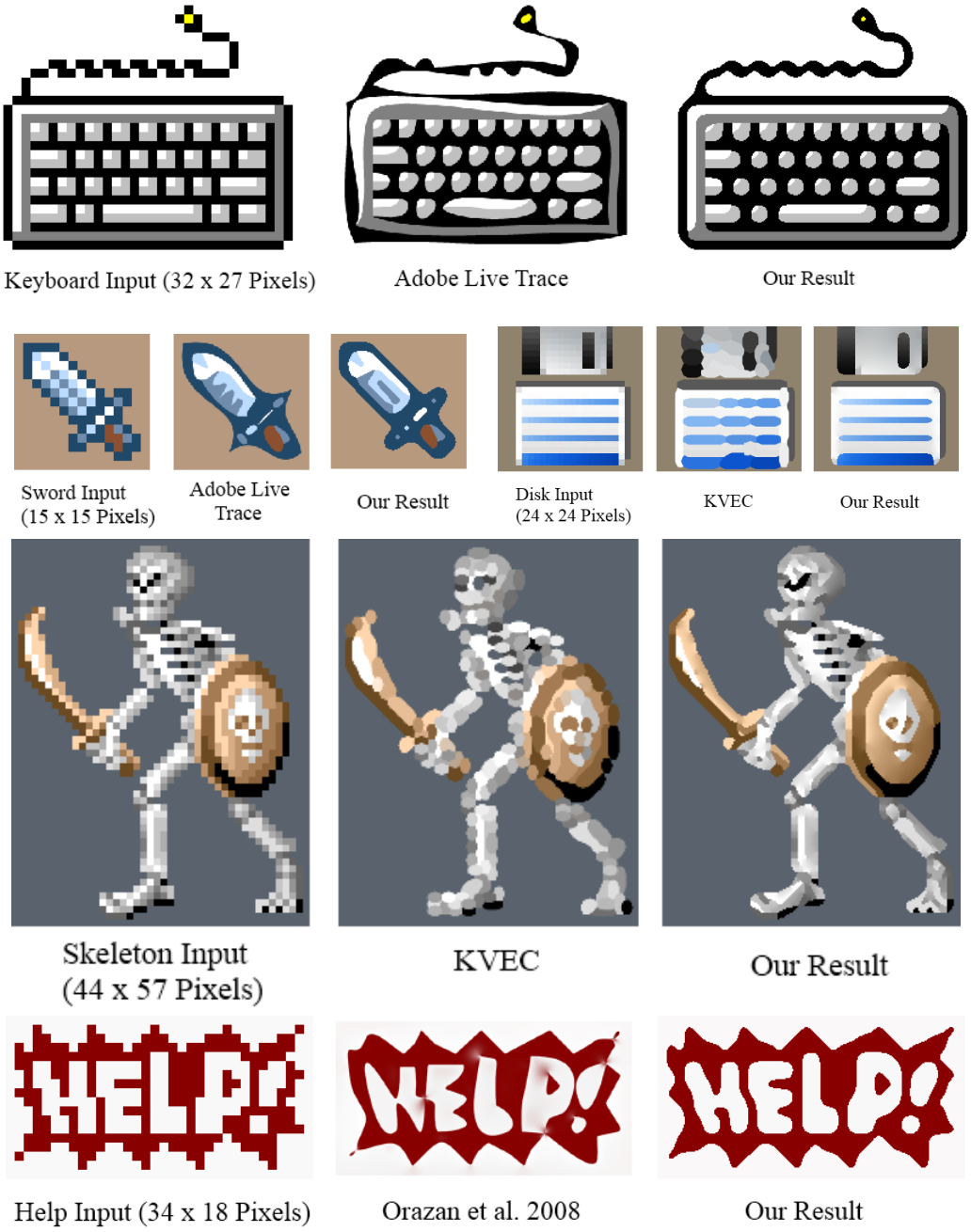


Figure 8.3: Comparison of some of our results with those of other vectorization methods. (Images: Help ©Nintendo Co., Ltd.; Keyboard ©Microsoft Corp.; Skeleton ©Sega Corp.; Sword ©Square Co., Ltd.; Orazan et al. example taken from [18])

Nearest Neighbor Result

hq4x Result

Our Result

Figure 8.4: Pixel art upscaling on a game environment. The input image is generated by an emulator and the upscaling factor is 4. Only a part of the image is shown here to increase visible details. For more detail zoom into the PDF file. (Image ©Square Co., Ltd.)

9 Conclusion

In this thesis we presented a method to convert low resolution raster graphics into a arbitrarily scalable vector representation. We determined regions of similar color in the input image and resolved ambiguous configurations on diagonally connected pixels. Our algorithms then reshapes the pixel cells so that neighboring pixels of similar color always share an edge, no matter if they are connected vertically, horizontally or diagonally. We extract edges separating differently colored regions and represent them by smooth B-spline curves. Additionally we showed that other vectorization methods often produce poor results on pixel art, while our algorithms handles a variety of inputs very well. Only specialized pixel art upscaling techniques produce comparable output but are restricted to fixed upscaling factors.

There are some possibilities for future work. For example we could analyze the occasional wrong decision made by our heuristics in Chapter 4 and try to scale the weights of these heuristics to improve the results. Obviously, it would be great to improve the performance of this approach so that it might be applied in real time in an emulator or similar piece of software. The first point to address for that has to be the optimization of curve points, as that is the slowest part of our algorithm. Furthermore, we have seen that our algorithm can not handle small details presented as little variations in color. With our rendering method these details get blurred into a single color. We could try to distinguish between details in the image and an intentional color gradient.

Bibliography

[1] M. Abramowitz and I.A. Stegun. *Handbook of mathematical functions with formulas, graphs, and mathematical tables*. Applied mathematics series v. 55, no. 1972. U.S. Govt. Print. Off., 1964. ISBN: 9780486612720.

[2] Inc. Adobe. *Adobe Illustrator CS5*. 2011. URL: `http://www.adobe.com/products/illustrator.html`.

[3] Franz Aurenhammer. "Voronoi diagrams - a survey of a fundamental geometric data structure". In: *ACM Comput. Surv.* 23 (3 1991), pp. 345–405. ISSN: 0360-0300.

[4] C.D. Boor. *A practical guide to splines*. Applied mathematical sciences. Springer, 2001. ISBN: 9780387953663.

[5] J. Canny. "A computational approach to edge detection." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8.6 (1986), pp. 679–698.

[6] J. Casey. *Exploring curvature*. Vieweg mathematics. Vieweg, 1996. ISBN: 9783528064754.

[7] E. Eisenberg and J. Eisenberg. *SVG Essentials*. O'Reilly Media, 2011. ISBN: 9781449313210.

[8] James H. Elder and Richard M. Goldberg. "Image Editing in the Contour Domain". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 23 (3 2001), pp. 291–296. ISSN: 0162-8828.

[9] Derek Liauw Kie Fa. *2xSaI*. 2001. URL: `http://vdnoort.home.xs4all.nl/emulation/2xsai/`.

[10] G.E. Farin. *Curves and surfaces for CAGD: a practical guide*. The Morgan Kaufmann series in computer graphics and geometric modeling. Morgan Kaufmann, 2002. ISBN: 9781558607378.

[11] Raanan Fattal. "Image upsampling via imposed edge statistics". In: *ACM SIGGRAPH 2007 papers*. SIGGRAPH '07. San Diego, California: ACM, 2007.

[12] Daniel Glasner, Shai Bagon, and Michal Irani. "Super-resolution from a single image". In: *2009 IEEE 12th International Conference on Computer Vision* Iccv (2009), pp. 349–356.

[13] Kenneth E. Hoff III et al. "Fast computation of generalized Voronoi diagrams using graphics hardware". In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 277–286. ISBN: 0-201-48560-5.

[14] Adobe Inc. *PostScript Language Reference, third edition*. 1999. URL: `http://partners.adobe.com/public/developer/en/ps/PLRM.pdf`.

[15] Stefan Jeschke, David Cline, and Peter Wonka. "A GPU Laplacian solver for diffusion curves and Poisson image editing". In: *ACM SIGGRAPH Asia 2009 papers*. SIGGRAPH Asia '09. Yokohama, Japan: ACM, 2009, 116:1–116:8. ISBN: 978-1-60558-858-2.

[16] Eric Johnston. *Eric's pixel expansion*. 1992. URL: `http://http://en.wikipedia.org/wiki/Pixel_art_scaling_algorithms`.

[17] KK-Software. *KVEC*. 2010. URL: `http://www.kvec.de/english/index.html`.

[18] Johannes Kopf and Dani Lischinski. "Depixelizing pixel art". In: *ACM SIGGRAPH 2011 papers*. SIGGRAPH '11. Vancouver, British Columbia, Canada: ACM, 2011, 99:1–99:8. ISBN: 978-1-4503-0943-1.

[19] Yu-Kun Lai, Shi-Min Hu, and Ralph R. Martin. "Automatic and topology-preserving gradient mesh generation for image vectorization". In: *ACM Trans. Graph.* 28 (3 July 2009), 85:1–85:8. ISSN: 0730-0301.

[20] Gregory Lecot and Bruno Lévy. "ARDECO: Automatic Region DEtection and Conversion". English. In: *17th Eurographics Symposium on Rendering - EGSR'06*. Nicosia/Cyprus, June 2006, pp. 349–360.

[21] Darren MacDonald and Jochen Lang. *Bitmap to Vector Conversion for Multi-level Analysis and Visualization*. 2008. URL: `http://www.svgopen.org/2008/papers/42-Bitmap_to_Vector_Conversion_for_Multilevel_Analysis_and_Visualization/`.

[22] Andrea Mazzoleni. *Scale2x*. 2001. URL: `http://scale2x.sourceforge.net/`.

[23] J.D. Murray and W. VanRyper. *Encyclopedia of graphics file formats*. O'Reilly Series. O'Reilly & Associates, 1996. ISBN: 9781565921610.

[24] Diego Nehab and Hugues Hoppe. "Random-access rendering of general vector graphics". In: *ACM Trans. Graph.* 27 (5 2008), 135:1–135:10. ISSN: 0730-0301.

[25] Alexandrina Orzan et al. "Diffusion curves: a vector representation for smooth-shaded images". In: *ACM Trans. Graph.* 27 (3 2008), 92:1–92:8. ISSN: 0730-0301.

[26] L.A. Piegl and W. Tiller. *The NURBS book*. Monographs in visual communication. Springer, 1997. ISBN: 9783540615453.

[27] Peter Selinger. *Potrace: a polygon-based tracing algorithm*. 2003. URL: `http://potrace.sourceforge.net/`.

[28] International Organization for Standardization/International Electrotechnical Commission. *ISO/IEC 8632*. 1999.

[29] Maxim Stepin. *hqx Magnification Filter*. URL: `http://web.archive.org/web/20070703061942/http://www.hiend3d.com/hq3x.html`.

[30] Jian Sun et al. "Image vectorization using optimized gradient meshes". In: *ACM SIGGRAPH 2007 papers*. SIGGRAPH '07. San Diego, California: ACM, 2007.

[31] W3C. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. Aug. 2011. URL: `http://www.w3.org/TR/SVG11/`.

[32] Tian Xia, Binbin Liao, and Yizhou Yu. "Patch-based image vectorization with automatic curvilinear feature alignment". In: *ACM Trans. Graph.* 28 (5 2009), 115:1–115:10. ISSN: 0730-0301.

Danksagung

Ich möchte mich herzlich bei Herrn Prof. Dr. Rainer Lienhart für die Vergabe des Themas und die Betreuung der Arbeit bedanken. Mein besonderer Dank gilt meinem Betreuer Fabian Richter, der stets für mich erreichbar war und mich mit konstruktiver Kritik in der Anfertigung dieser Arbeit unterstützte.
Weiterhin danke ich meinen Freunden die mir in Gesprächen die nötige moralische Unterstützung gaben und mir auch sonst mit Rat zur Seite standen. Besonders möchte ich hier Thomas Rothörl und To Quyen Hoang hervorheben.
Großer Dank gebührt auch meinen Eltern die mich stets unterstützt haben, und die mir dieses Studium erst ermöglichten.

Appendix A

Test Program

The approach described in this thesis has been implemented and included on a CD. The evaluation in Chapter 8 has been performed with this implementation. Converting an image from raster graphic to vector graphic can be done by simply opening a console window and running:

Vectorization.exe *input-file*

If the input file is an image file it will be converted into a vector representation and displayed on the screen. Furthermore it is possible to store the converted image on the hard drive in a custom format with the extension *.vec*. When using a previously stores vector image as the input file, this image will simply be displayed again. There exist a number of different settings for this implementation that can be changed by modifying the file *settings.cfg* located in the same directory as the program. The option that can be set in this file are the following:

- **intermediateResults** - determines if the similarity graph and the reshaped pixel image should be displayed.
- **smooth** - enables/disables smoothing of the spline curves.
- **merge** - enables/disables merging of spline curves as described in Section 5.2.
- **diffuse** - enables/disables color diffusion on image during rendering.
- **showCells** - when intermediateResults is enabled this will frame each pixel cell in the reshaped cell graph to with a blue line.

- **scale** - sets the upscaling factor of our approach.
- **diffuseIterations** - sets the number of iterations that are used when diffusing the color of the image.
- **smoothIterations** - sets the number of iterations that are used when smoothing the spline curves.
- **saveToHD** - when enabled the program stores the result of the conversion on the hard drive.

Appendix B

DVD Content

The attached DVD contains the following:

- The Visual Studio 2008 project containing the implementation of our approach
- 58 input files used to test the implementation
- The Wild Magic Geometric Tool 5.7 library
- The freeglut library
- The boost 1.48.0 serialization library
- The OpenCV 2.2 library
- Related work papers as PDF files where available.
- The LaTeX files, PDF output file and image files used in figures for this thesis