Depixelizing Pixel Art – Current Techniques and Improvements

Judith Jeszenkovits

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2016

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 27, 2016

Judith Jeszenkovits

Contents

Abstract

Image scaling is one of the basic topics in the field of computer graphics and nearly everyone has already used some kind of upsampling algorithm, for example by enlarging a photo with Photoshop or Paint. Upscaling an image can be difficult, especially if the resulting image should not only be of a higher resolution but also of higher visual quality. A good result is most difficult to achieve if the input image is a very small pixel image which is defined by only a few pixels and colors, like a pixel art image. This is because there is just a very small space for all the images' information to be placed. Several approaches have been implemented to extract all this information and generate a new image of larger size, while keeping or even improve the visual outcome.

The main part of this master thesis is to reimplement one of these algorithms, to be specific, the *Depixelizing Pixel Art* algorithm by Johannes Kopf and Dani Lischinski, which was published as scientific paper only and not as Open Source implementation code. The main focus of this thesis is on understanding how this process works and which steps are necessary for the best possible outcome. The algorithm is implemented as ImageJ Plugin and produces a vector based representation of the pixel art image as final output. Along with this basic implementation some additional improvements are introduced. Included in this extensions are an export of the output image as PDF file and the introduction of a color gradient wherever possible. The goal of these last steps is to make the image look even smoother. The algorithm is tested with several test images and the results are shown and discussed at the very end of this thesis.

Kurzfassung

Eine Änderung der Größe und der Auflösung eines Bildes, auch Skalierung genannt, ist eine der Grundoperationen im Bereich Computer Grafik. Nahezu jeder hat diese schon einmal beim Vergrößern eines Bildes in Photoshop oder Paint verwendet. Beim Vergrößern der Bilder sind einige Aspekte zu beachten, besonders wenn das Bild nicht nur größere Dimensionen, sondern auch eine bessere visuelle Qualität haben soll. Das ist besonders schwierig, wenn das zu vergrößernde Bild sehr klein ist und nur aus wenigen Pixeln und Farben besteht. Der Grund dafür ist, dass sehr viel Information in sehr kleinen Dimensionen gespeichert werden muss. Es wurden bereits einige Ansätze implementiert, die sowohl diese extrahierten Informationen verarbeiten, als auch die visuelle Repräsentation des vergrößerten Bildes verbessern.

Der Hauptteil dieser Masterarbeit beschäftigt sich mit der Reimplementierung eines Ansatzes von Johannes Kopf und Dani Lischsinki namens *Depixelizing Pixel Art*, der nur in Form eines wissenschaftlichen Papers, nicht aber als Open Source Code veröffentlicht wurde. Zu verstehen, wie dieser Ansatz funktioniert und welche Schritte für das bestmögliche Ergebnis nötig sind, steht dabei im Mittelpunkt dieser Arbeit. Der Algorithmus wird als ImageJ Plugin implementiert und generiert ein, auf Vektoren basiertes, Endergebnis. Zusätzlich zu dieser Basis-Implementierung werden auch einige Erweiterungen vorgestellt und implementiert, die das Endresultat als PDF-Datei exportieren. In weiterer Folge wird ein Farbverlauf an Stellen hinzufügt, an denen es sinnvoll erscheint, um die Qualität des vergrößerten Bildes noch weiter zu verbessern. Der gesamte Algorithmus wird mit diversen Bildern getestet und die Endresultate, sowie eventuelle Schwierigkeiten werden am Ende der Arbeit behandelt.

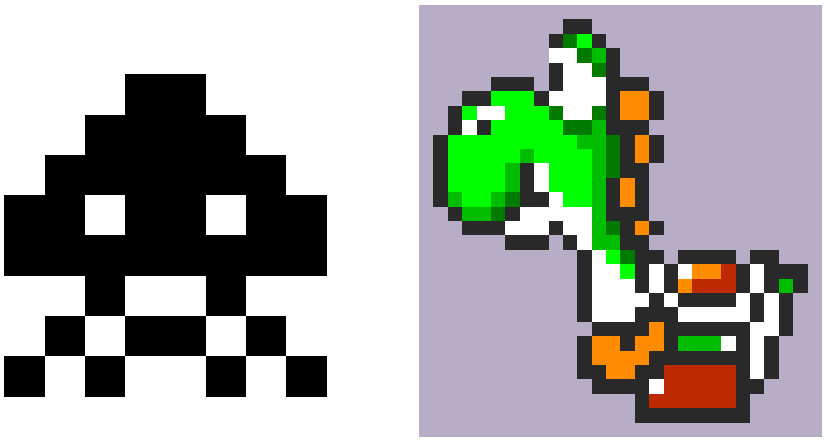
Chapter 1

Introduction

1.1 Motivation

The motivation for choosing this topic for the master thesis was to fully understand and being able to describe and reimplement the process of image upsampling which Kopf and Lischinski introduced in 2011 [6] for the first time. The *Depixelizing Pixel Art* technique includes multiple steps of which each single step will be described in every detail to completely understand the algorithm. In addition to the basic implementation per the documentation, some extensions were implemented afterwards to improve the outcome even more.

The thesis starts off with this introduction chapter that also explains some basic terms like “image editing”, “image processing”, “raster graphics” and “vector images”, as well as delivering a definition of pixel art images. Afterwards, Chap. 2 describes some already implemented algorithms for image upsampling, where pixel art images are taken into account in particular. Chapter 3 comprises the documentation of the first part of the implemented algorithm where every single step from generating a similarity graph to drawing a resulting image with ImageJ overlays is described. Chapter 4 is about the extensions that were implemented to make the image look smoother. First the result is exported as PDF file and afterwards a color gradient is introduced to the resulting image wherever it is feasible. Afterwards, the different implementation states are displayed in Chap. 5 using several test images that show eventual difficulties during the implementation process. An evaluation of each of these steps is done and the results are discussed. Finally, a conclusion (see Chap. 6) ends the thesis with some final thoughts and some ideas for future work.



(a) (b)

Figure 1.1: Visual representation of two test images. In (a) there is the well known "Space Invader", which is the smallest test image with only 10×10 pixels and two colors. Figure (b) pictures "Yoshi", which is known from the Super Mario games. This image is the largest test image with 28×30 pixels and a color-depth of 8.

1.2 Pixel Art

Pixel art is a term that describes digital art at a pixel level. But other than photos or other digital paintings nowadays, which also consist of pixels, real pixel art images are created since the mid-1990s and were used in the most early computer games. At this time developers were forced to create images using limited graphics and computing resources. Because of these limitations the images were as small as possible and also contained only as many different colors as really needed. In this type of image every pixel was set and colored by hand. Because of this fact every pixel in the image matters and deleting a pixel or offsetting just a few pixels could have a significant effect on the image.

The pixel art images used for developing and testing the implemented algorithm are well known from old computer games. The smallest one of these test images had a size of 10×10 pixels and contained only two different colors, whereas the largest one had a size of 28×30 pixels and contained eight different colors. These two images are shown in Fig. 1.1 where they were scaled to the same size for a better visual representation. For more information about pixel art or how to create it see [18, 24, 11].

1.3 Image Processing vs. Image Editing

To get started with the whole upsampling topic one has to understand some basic techniques and keywords. Two of these keywords are *image editing* and *image processing*. The severe difference between these two expressions, as well as their meaning, are described in this section.

1.3.1 Image Processing

Image processing in most cases refers to digital image processing. But it is also a denotation for optical and analog image processing, irrespective of not being used commonly.

Digital image processing, which can be translated into German as "Bildverarbeitung", is the use of computer algorithms to perform image processing on digital images. Basically it is the computational transformation of an image signal. According to this, image processing is any form of signal processing for which the input is an image, such as a photograph, illustration or video frame. The phrase *image processing* is mainly used with a technical background whereas *image editing* is commonly used in an artistic way. Image processing includes the processes of registering, identifying, analyzing, editing, saving and displaying images, photos, illustrations, single video frames or other digital graphical documents.

Tom Fletcher defined image processing as "the study of any algorithm that takes an image as input and returns an image as output" [4], which simplifies the previous definitions.

Image processing is often used in the fields of computer graphic and computer vision. Computer graphic mainly uses images, which are made from physical models manually, instead of being generated from imaging devices such as a camera. More information about image processing can be found on [13, 14, 27, 32].

1.3.2 Image Editing

Image editing is a specific part of image processing, which is, compared to image processing, considered a creative, artistic act.

Image editing [15, 28, 32] (translated into German as "Bildbearbeitung") describes the processes of altering images, including digital photographs, traditional photographs or illustrations. Traditional analog image editing is known as photo retouching, using tools such as an airbrush to modify photographs, or editing illustrations with any traditional art medium. With the use of graphic software programs, such as Adobe Illustrator[1], Adobe

[1] http://www.adobe.com/at/products/illustrator.html

Photoshop[2] or Paint[3], the user can manipulate, enhance and transform images with, for example, removing unwanted elements like dust specks and scratches, adjusting the geometry of the image like rotating and cropping, sharpening or softening the image, making color changes or adding special effects to the image.

The main goal of every image editing process is to improve the quality of the input image. Some main procedures, which nearly every image editing tool supports, are contrast adjustment, noise removal, change of image orientation and image size, cropping, color and contrast change, gamma correction and the use of the image histogram.

Most image editing programs can handle vector graphics as well as raster graphics. The difference of these two types and their advantages and disadvantages compared to each other, can sometimes be crucial in the editing process. Because of this they are described in more detail in the following section.

1.4 Pixel Images vs. Vector Graphics

There are several sources, online as well as in a printed form [2, 30, 33, 35], that define the difference between pixel graphics and vector images.

1.4.1 Pixel Images

Pixel images are also known as bitmap or raster graphics. Images of this type are made up of the smallest possible image parts, called pixels. These pixels are arranged as regular image matrix with discrete coordinates, each one containing a pixel value (see Fig. 1.2(b)). Furthermore, the more pixels an image contains the more possibilities of different pixel values are given. According to this the color-depth of an image is defined by the amount of colors represented in the graphic.

Pixel graphics are mostly used in photography, as this type of graphic is suitable especially for the representation of photos and complex color gradients. There are also some very convenient image editing programs, for example Adobe Photoshop, Adobe Elements[4], Adobe Lightroom[5] or Gimp[6]. Pixel graphics can be identified by the file extensions JPG/JPEG[7], PNG (Portable Network Graphics), PSD (Photoshop Document), TIF (Tagged Image File Format), BMP (Bitmap) and GIF (Graphics Interchange Format).

[2] http://www.adobe.com/at/products/photoshop.html
[3] http://www.getpaint.net/index.html
[4] http://www.adobe.com/at/products/photoshop-elements.html
[5] http://www.adobe.com/at/products/photoshop-lightroom.html
[6] https://www.gimp.org/
[7] The file extension is named after the Joint Photographic Experts Group that developed the JPEG norm.

One advantage of pixel images is that the resolution (dpi) can be changed as desired. *Dpi* is short for "dots per inch" and therefore a measuring unit. It describes how many single pixels are placed on a one inch line. An image with a dpi of 300 has 300 pixels placed on such a small line. These pixels describe colors, shadings and details of the graphic. The more dpi an image has the more details can be recognized and more difficult and smoother color gradients are possible. Another advantage is the fact that every single pixel of such an image could be edited separately if needed. The possibilities of changing the image at such a deep level are nearly endless.

Of course there are disadvantages as well with this type of image. The biggest one is probably that, when scaling a pixel image, the appearance becomes granular. This is why image upscaling with raster images in general does not work quite well. Another disadvantage is the file size of a raster image. A photo that was edited with Adobe Photoshop probably has several layers and the size can increase up to over 100 MB very easily. This happens because each pixel information has to be stored in the image file, which is very storage intense. To counteract this storage problem different ways of compression have been introduced in the past. One of them is, for example, the LZW compression[8], which is used in the TIFF format. This compression stores several neighbored pixels with the same color as one color value. This saves time and reduces the needed storage capacity.

1.4.2 Vector Graphics

Vector graphics on the other hand do not consist of pixels but of basic geometrical elements such as lines, curves, polygons or circles, which are defined as mathematical functions. These elements do not need many parameters and are easy to create (see Fig. 1.2(c)). A circle for example is just defined by its radius and center. In addition, one can include several properties like line width, contour and fill color or even fill patterns and color gradients. Changes like this are executed on mathematically defined regions or on elements itself, in contrast to pixel graphics where every single pixel has to be changed.

Vector graphics are often used in the printing industry to create geometric designs, logos, icons, info graphics and fonts. The favored program to work with using vector graphics or vector designs is Adobe Illustrator [19]. Vector images in general are saved as AI (Adobe Illustrator), EPS (Encapsulates Post Script), SVG (Scalable Vector Graphic) or PDF (Portable Document File) files. EPS, AI and PDF are more often used in prepress procedures whereas SVG is used for the web.

One advantage of vector images over pixel images is the fact that they need less storage capacity. This is the result of geometrical forms and vectors

[8]The name is compound of the programmers' last names A. Lempel, J. Ziv, T. Welch.

Figure 1.2: The Difference of a pixel and a vector image. The original image (a) can either be provided as pixel image (b) or as vector image (c). In (b) the image part looks granular and one can see the underlying pixel structure. In (c) on the other hand, the outer contour is defined by a curve, which is displayed as a clear line with no pixel artifacts. The outer contour is defined by interpolation points through which the line is passing. Two control points for every interpolation point define the entrance angle as well as the exit angle of the line.

being defined by only a few points or vectors. Small forms and fonts can therefore be described with just a few mathematical formulas. As a result, these image files have a smaller size than pixel images and need less execution time for being generated. Another advantage is that these kind of images can be scaled (up or down) easily without getting blurry or granular and without any information loss. This is because the mathematically defined forms are newly calculated every time the image is resized so that a new form is generated in the desired size. Compared to an upscaled pixel image, an upscaled vector image looks smoother and does not suffer from any stair-casing effects. Moreover vector graphics can be edited more easily than pixel graphics, as they are not reduced to one layer in an editing program. One can edit the color and the line thickness for each element individually without any problems and save the image several times in different sizes for different situations. For example, a logo can be used in a smaller size on a website and the same logo can be used in a much bigger size to be printed on a banner.

One disadvantage that comes with the usage of vector images is that such images can never show color gradients or details as natural as pixel images can. With a photo much more detail, shading and depth information can be provided. To get a similar result with a vector image a large amount of elements has to be generated. The elements are getting very small whereby the file size gets bigger and the complexity of such a detailed illustration is very high. Furthermore, it is not possible to edit every single point in the image but only a path. If a path segment is deleted another element has to fill up or has to be put over the missing spot, besides one wants to define a transparent region in the image.

Chapter 2

State of the Art

The algorithm that was implemented as master project belonging to this written master thesis (see Chap. 3 for the whole implementation process) is basically a vectorization program, which takes a raster image as input and delivers a vector image as output file. The advantage of a vector based image as described in Sec. 1.4 is that it can easily be scaled with no stair-casing artifacts or information loss. To get a better understanding why this fact is so important for the implementation, the next section explains the most important differences between some state of the art scaling algorithms based on pixel graphics as well as some well known vectorization algorithms.

2.1 Image Upsampling

There are several upsampling techniques for pixel images. The conventional way is to apply a linear filter to the image, which is generated by using either analytical interpolation or signal processing theory. The most used techniques are Nearest-Neighbor, bilinear, Bicubic and Lanczos interpolation. In general, these techniques work well for larger images but, as an example in Fig. 2.3 shows, the results for pixel art images suffer from blurring or stair-case artifacts in some cases.

2.1.1 Nearest-Neighbor Upsampling

The Nearest-Neighbor upsampling [2, 16, 12] is the most basic of all these algorithms and it requires the least processing time because it only considers one pixel of the original image. The algorithm has the effect of simply making each pixel larger by replacing every pixel with a number of pixels of the same color. The size of the interpolation kernel can be chosen individually. There is an example of a 2D Nearest-Neighbor interpolation kernel in Fig. 2.1. The resulting image is larger than the original and preserves all the original's details but may show some stair-casing effect, especially where

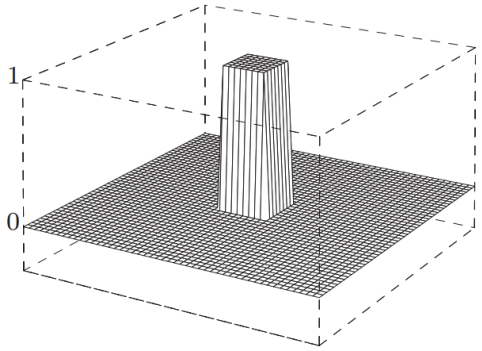


Figure 2.1: This image shows an interpolation kernel of the Nearest-Neighbor interpolation $k_{nn}(x, y)$ in 2D with $-3 \leq x, y \leq 3$. The graphic is taken from [2].

diagonal lines are enlarged in the original image. This stair-casing artifact is one negative side effect of the Nearest-Neighbor interpolation. One example image is shown in Fig. 2.3(a) with the original image looking just the same but being 16 times smaller.

2.1.2 Bilinear Upsampling

In mathematics, bilinear interpolation [2, 16, 12] is an extension of linear interpolation for interpolating functions of two variables (e.g. x and y) on a rectilinear 2D grid.

The key idea is to perform linear interpolation first in one and then in the other direction. Although each step is linear in the sampled values and in the position, the interpolation as a whole is not linear but rather quadratic in the sample location. The two dimensional kernel k_{bilin} can be described as the product of the two belonging one dimensional kernels $k_{\text{lin}}(x)$ and $k_{\text{lin}}(y)$ for the x and the y dimension respectively. These two kernels can generally be described as

$$k_{\text{lin}}(a) = \begin{cases} 1 - a & \text{for } |a| < 1, \\ 0 & \text{for } |a| \geq 1. \end{cases} \tag{2.1}$$

The two dimensional kernel can then be described as

$$\begin{aligned} k_{\text{bilin}}(x, y) &= k_{\text{lin}}(x) \cdot k_{\text{lin}}(y) \\ &= \begin{cases} 1 - x - y + xy & \text{for } 0 \leq |x|, |y| < 1, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{2.2}$$

The one- and two-dimensional interpolation kernels of the bilinear interpolation are displayed in Fig. 2.2.

When using bilinear interpolation for upsampling images also some undesirable blurring of details can occur. Nevertheless in computer vision and

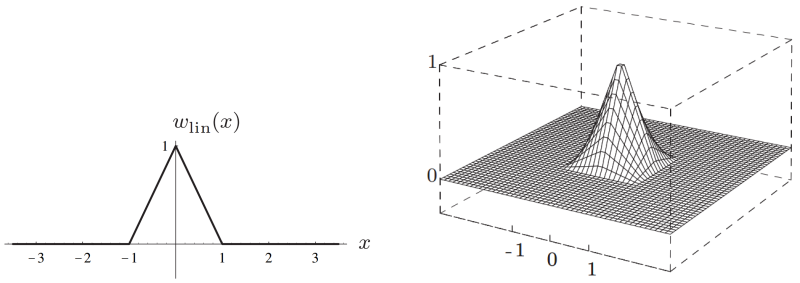


Figure 2.2: Example images of the bilinear interpolation according to [2]. The one-dimensional bilinear interpolation kernel (a) and the interpolation kernel of the bilinear interpolation $k_{\text{bilin}}(x, y)$ in 2D (b) with $-3 \leq x, y \leq 3$.

image processing, bilinear interpolation is one of the most used resampling techniques.

Unlike other interpolation techniques such as Nearest-Neighbor and bicubic interpolation, this interpolation technique considers the values of the four neighbors closest to the known pixel coordinate (x_0, y_0), which are defined as

$$\begin{aligned} A &= I(u_0, v_0), & B &= I(u_0 + 1, v_0), \\ C &= I(u_0, v_0 + 1), & D &= I(u_0 + 1, v_0 + 1), \end{aligned} \tag{2.3}$$

with $u_0 = \lfloor x_0 \rfloor$ and $v_0 = \lfloor y_0 \rfloor$. After a linear interpolation in both the horizontal and the vertical direction, two new values E, F are calculated as

$$\begin{aligned} E &= A + (x_0 - u_0) \cdot (B - A) = A + a \cdot (B - A), \\ F &= C + (x_0 - u_0) \cdot (D - C) = C + a \cdot (D - C) \end{aligned} \tag{2.4}$$

as the distance a is defined as $x_0 - u_0$. The final interpolation value G can then be calculated as

$$\begin{aligned} \hat{I}(x_0, y_0) = G &= E + (y_0 - v_0) \cdot (F - E) \\ &= E + b \cdot (F - E) \\ &= (a - 1)(b - 1)A + a(1 - b)B + (1 - a)bC + abD. \end{aligned} \tag{2.5}$$

The result of the bilinear interpolation looks much smoother and more natural than the one with the Nearest-Neighbor.[1] However, stair-casing can happen with this approach as well. The algorithm is also quite fast due to its simplicity.

Despite the bilinear approach and the Nearest-Neighbor interpolation, which are the interpolation methods used most of the time, there are the

[1] Compare Fig. 2.3(a) and Fig. 2.3(b) for better understanding.

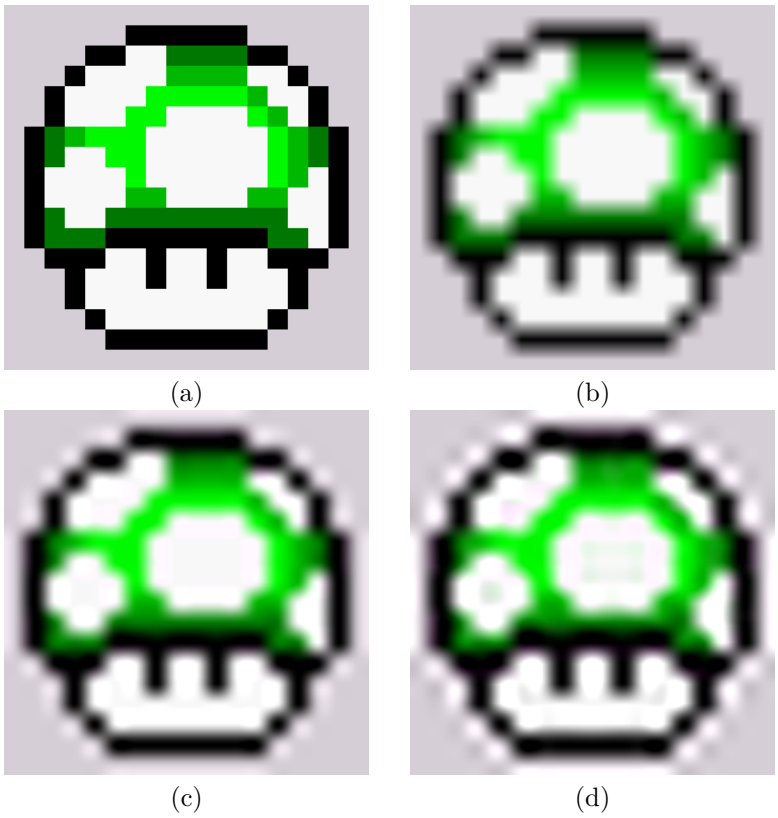


Figure 2.3: Comparison of different upsampling techniques. The original Pixel Art image is used as input. This image is upsampled in different ways, the first being Nearest-Neighbor interpolation figured in (a). The second image (b) is the result of the input image being interpolated with the bilinear interpolation, in (c) the result of the bicubic interpolation is captured and (d) shows the resulting image with the Lanczos interpolation.

Bicubic and the Lanczos interpolation. These techniques are not very different in the end results, which can also be seen in Figs. 2.3(c) and 2.3(d), compared to the bilinear interpolation. The Bicubic interpolation consideres more neighbored pixels as the bilinear approach, which causes some kind of ringing effect in the example picture. The Lanczos interpolation delivers an even worse ringing end result. This is because there are not that many different colored neighbor pixels of a specific pixel which causes this error. A more detailed description of how these two interpolation methods work can be found on [2, 16, 12].

2.2 Pixel Art Upsampling Techniques

Because of the fact that the images in the very first computer games were quite small, upsampling techniques that use filters like the Nearest-Neighbor or the bilinear approach described in Sec. 2.1.2 do not work well for such small pixel art images. This is why some different algorithms were developed over time. All of these new algorithms, which were especially implemented for pixel art images, were used in the emulation area. Nearly none of these techniques was introduced at a conference or a public venue. However, most of them are Open Source and free to use. Because of this, only two different techniques, namely the EPX/Scale2X algorithm and the hqx algorithm are described in the next section.

2.2.1 EPX and Scale2x Algorithm

The EPX and the Scale2x interpolation technique were developed at different times but both end up producing the exact same output. These two algorithms are described in this section.

EPX Algorithm

The first algorithm to mention is the EPX algorithm [5]. Developed somewhere around the early 1990's by Eric Johnston – EPX is short written for Eric's Pixel Expansion – it was primarily used to convert computer games in such a way that they can be played not only on the computer, which they were initially made for, but on other devices as well.

The EPX technique expands every pixel of the original image I into four new pixels (see Fig. 2.4(a)) by considering the color P of the original pixel and four more colors of the neighboured pixels, as can be seen in Fig. 2.4(b). The colors of the initial pixels are defined as

$$\begin{aligned} P &= I(u,v), & T &= I(u,v-1), \\ R &= I(u+1,v), & B &= I(u,v+1), \\ L &= I(u-1,v). \end{aligned} \tag{2.6}$$

The resulting image has a width and a height, which are twice as high as the dimensions of the original input image. To determine the color of the new pixels a four-connection-neighborhood is assumed with the original pixel.

The color P_1 of the newly generated upper left pixel is defined by considering the color of the neighbored pixels on the top (T) and the left side (L) of the original pixel respectively. If these two neighbors have the same color, also P_1 is assigned the same color. If they do not share the same color the original color P is assigned to P_1. This procedure is done for the three

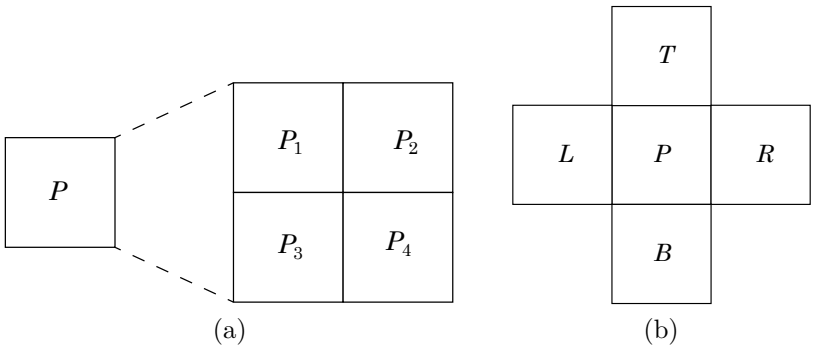


Figure 2.4: Pixel extension and neighborhood situation used by the EPX algorithm. Picture (a) shows how an initial pixel with color $P = I(u, v)$ is converted into four new pixels with the colors P_1, P_2, P_3, P_4 and (b) shows all neighbors of the initial pixel in the original image, which have the colors T, L, B, R assigned, that are considered during the coloring process.

other new pixels as well, considering the colors of the top and right neighbors (T and R) for assigning a color to P_2, the colors of the bottom and left neighbors (B and L) for coloring the new pixel on the bottom left in the color P_3 and the bottom and right neighbors' colors (B and R) for assigning a color to P_4. After the four new parts of the image are colored, one more examination has to be done. If three or more out of the four colors T, L, R and B are the same all four new pixels are painted in the color of the original pixel, which is P, because otherwise there might be a loss of color information in the end result.

For a better understanding some pseudocode is provided in Prog. 2.1. There are also some examples of different coloring situations in Fig. 2.5. The image shows the colors of the neighboring situation (P, T, L, B and R) and the four resulting colors P_1, P_2, P_3 and P_4 for the resulting pixels.

Scale2x Algorithm

The Scale2x algorithm was developed by Andrea Mazzoleni [25, 26] to improve the quality of old games with a low video resolution. At this time Mazzoleni did not know about the EPX implementation, but delevoped an algorithm that came out with the exact same result as the EPX technique. The only difference is that the definition of the Scale2x algorithm (which can be seen as pseudocode in Prog. 2.2) is slightly different, wherefore this approach is slightly faster. The four new pixels are initialized with the color of the original pixel $P = I(u, v)$ in the first step, whereupon it is checked whether tree or more of the four initial neighboring pixels have the same color. In this case, the new pixels are colored in the same way as in the EPX algorithm, otherwise the next pixel is processed.

Algorithm 2.1: The EPX algorithm

function EPXUPSAMPLING(P, T, L, B, R)
  $(P_1, P_2, P_3, P_4) \leftarrow ((0,0,0),(0,0,0),(0,0,0),(0,0,0))$
  if $T = L$ **then**
    $P_1 \leftarrow T$
  end if
  if $T = R$ **then**
    $P_2 \leftarrow T$
  end if
  if $B = L$ **then**
    $P_3 \leftarrow B$
  end if
  if $B = R$ **then**
    $P_4 \leftarrow B$
  end if
  if 3 colors of $\{T, L, R, B\}$ are the same **then**
    $(P_1, P_2, P_3, P_4) \leftarrow (P, P, P, P)$
  end if
return (P_1, P_2, P_3, P_4)
end function

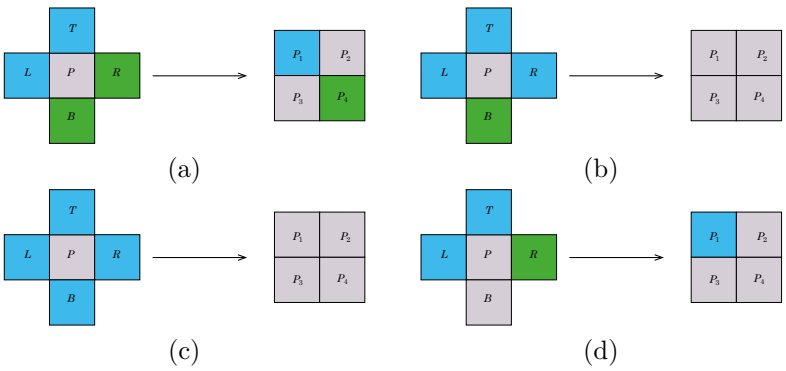
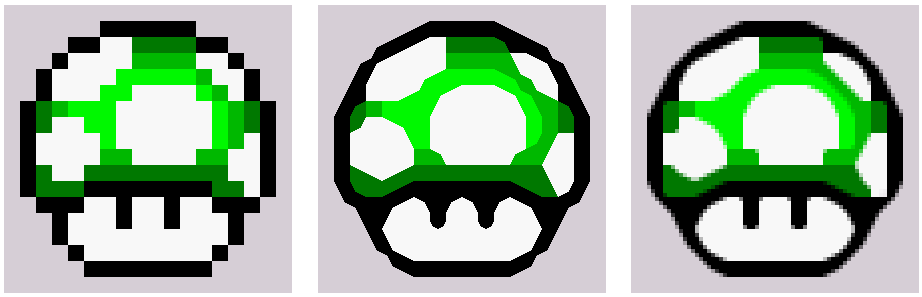


Figure 2.5: Some initial neighborhood situations and the solution for the new pixels' colors. In (a) the four neighbors are of only two different colors, which means that for every pixel the color is defined by the adjacent neighbor pixels. The upper left pixel is colored blue because of the colors T and L being blue. The new bottom right pixel is painted, because of R and B being green, in a green color. The top right and bottom left pixels are colored in the original pixels' color gray, because the respective neighbors differ in color. In (b) and (c) more than three neighbored pixels have the same initial color, whereas all new pixels are painted gray. For (d) only the upper left pixel is painted blue because of its neighbors being the same color.

Algorithm 2.2: The Scale2x algorithm

```
1:  function SCALE2XUPSAMPLING(P, T, L, B, R)
2:      (P_1, P_2, P_3, P_4) ← (P, P, P, P)
3:      if T ≠ B ∧ L ≠ R then
4:          if L = T then
5:              P_1 ← T
6:          end if
7:          if R = T then
8:              P_2 ← T
9:          end if
10:         if L = B then
11:             P_3 ← B
12:         end if
13:         if R = B then
14:             P_4 ← B
15:         end if
16:     end if
17: return (P_1, P_2, P_3, P_4)
18: end function
```



(a) (b) (c)

Figure 2.6: Comparison of the EXP/Scale2x algorithm and the hq4x procedure. In (a) the original image is to see, which is upscaled with the Nearest-Neighbor algorithm. Figure (b) shows the result of the EPX/Scale2x procedure and (c) is the end result for the the hq4x algorithm.

The border situation is solved the same way in both algorithms. If the initial pixel is located at the image border the value of the neighbored pixels, that are not included in the image any more, is assumed to be the same as the value of the border pixels. Figure 2.6 shows example images with the first one being the original input image upscaled with the nearest neighbor approach (pictured in Fig. 2.6(a)). The second one (see Fig. 2.6(b)) shows the result of the EPX/Scale2x upsamling.

2.2.2 The hqx Upsampling Algorithms

The problem with the EPX and the Scale2x algorithm described before is that these algorithms in their original form can upsample an image only by a factor of 2. The hqx family however is a set of algorithms, namely the hq2x, hq3x and hq4x algorithm, which can upsample an image by a factor of 2, 3 or 4 respectively. These techniques were developed by Maxim Stepin [34] especially for small pixel art images. The algorithms are quite fast and produce high-quality result images.[2] They are used in a number of emulation systems until now. For this there are also a number of implementations in different languages, like the Java implementation by Edu Garcia [17] or the C# implementation by Tamme Schichler [29], which are both Open Source. The procedure of all of these implementations, however, is the same.

Starting with a specific pixel P determine a 8-neighborhood-connection. The first step is to analize the 3×3 area around the source pixel. For each of the eight neighbors the color difference between the neighbor and the central pixel P is calculated in the YUV color space. If the colors differ more than 48, 7 or 6 units in the Y, U or V channel respectively the neighbor pixel is considered *distant* to the center pixel. If the channels do not differ that much the pixel is considered *close* to P. By comparing all eight neighbors there are $2^8 = 256$ different possibilities of distant/close combinations.

For every single one of these distant/close combinations one color entry in a lookup-table is generated considering a predefined neighborhood (which gets greater with the algorithms degree) around the source pixel. In this way repetitive color combinations can be found more easily and edges, lines and color patterns are prevented by interpolating the pixels of the original image.

It is just a logical consequence that the larger the input image is, the longer the algorithm takes to produce the lookup-table and the resulting output image. Because of this, it is possible but not reasonable to take a photograph as input image. Instead the pixel art images and images up to a resolution of 256×256 can be interpolated in real time. In Fig. 2.6(c) there is an example of an image being upscaled with the hq4x algorithm.

2.3 Image Vectorization Techniques

As mentioned in Sec. 1.4 there are severe differences between pixel images and vector graphics. So besides the upsampling techniques described in Secs. 2.1 and 2.2, which produce only pixel graphics as end results, there are several techniques that produce a scale invariant vector graphic as end result. This section gives a more detailed overview about several vectorization techniques, namely the Potrace, the Diffusion Curves and the Adobe Live Trace algorithm.

[2] Hqx stands for high quality (hq) magnification (x).

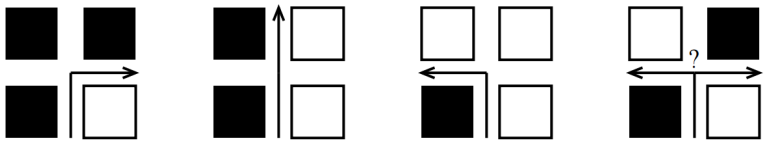


Figure 2.7: Showing the way of how a path is generated with the *Potrace* algorithm. Starting at a specific vertex the path moves forward always keeping the white pixels on the right and the black pixels on the left side. In the case of the very last image there are two possible ways for the path to continue. This situation is solved with the turn policy of the *Potrace* algorithm. This image is taken from [31].

2.3.1 Potrace

Peter Selinger introduced an algorithm called *Potrace*[3] [31]. The algorithm is fast, simple and very efficient. Furthermore "it tends to produce excellent results". However, this algorithm is only suitable for binary images and therefore not usable with the most pixel art images or color images.

As many other algorithms, the *Potrace* performs several steps for the best result possible. This steps are

- generating several paths from the input image using a turn policy,
- generating polygons from these paths,
- applying vertex adjustments and smoothing and
- doing curve optimization to finalize the output.

Starting with the input image (see Fig. 2.9(a) for example) it is assumed that the background color is the white color (continuing over the borders of the image) and the foreground is made up of all pixels with a black color.

The Potrace algorithm starts at a pair of adjacent pixel being of different color. It then travels along the edges keeping the black pixel on the left and the white pixel on the right hand side. Whenever hitting a corner the algorithm decides either to go straight ahead, turn left or to turn right depending on the colors of the surrounding pixels (as can be seen in Fig. 2.7). In the case of a checkerboard pattern, the algorithm has to decide whether to go left or right. To decide which way to go, there are several turn policies implemented. In terms of the *left-* and the *right-policy* the path is always extended to the left or the right side. The *black-* and the *white-policy* prefer to connect black or white components respectively. The *majority-* and the *minority-policy* prefer to connect the color that occurs most/least frequently in a predefined neighborhood. Last but not least there is the *random-policy* that makes a random choice. This is repeated as long as the path returns

[3]Potrace is a combination of the words *polygon* and *tracer*, which describes the algorithm in just one word. See http://potrace.sourceforge.net/.

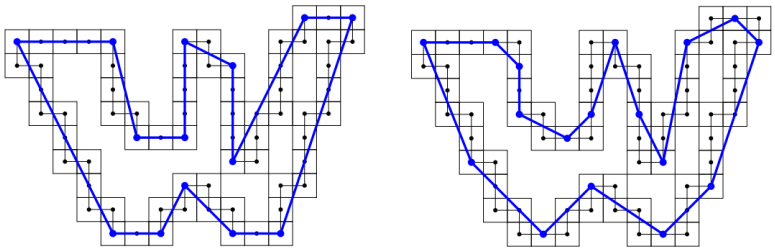


Figure 2.8: Two possible results for a polygon of a specific closed path with (a) being the optimal polygon and (b) being a non-optimal polygon, depending on the number of line segments. The images is taken from [31].

to its starting point and becomes a closed path. After finding such a closed path, it is removed from the image by inverting all pixel values inside of the path. The result is a new image, to which the algorithm is applied recursively until no more black pixels are left. The end result of this step is a set of multiple closed paths that describe the outer contour of each region in the original image.

The next step is to generate an optimal polygon, approximating a specific closed path for all of the closed paths being found in the earlier step. Such a polygon consists of several straight lines approximating the shape of the closed path. Note that there can be more than one possibility of a polygon for a certain sequence of vertices (see Fig. 2.8(a) and Fig. 2.8(b)). To decide which polygon is the more optimal one, the number of segments is counted and the one with the lower number of segments in the polygon is assumed to be the better one. The result of this step can be seen in Fig. 2.9(b) for a specific example.

The next step is to adjust the vertices of the polygon and to smooth everything. For each set of three consecutive vertices a_i, a_{i+1} and a_{i-1} of the polygon, two line segments (a_i, a_{i+1}) and (a_i, a_{i-1}) are created and the midpoints of these lines are defined as b_1 and b_2. Considering the angle between the three points b_1, a_i and b_2 and depending on the result of a corner detection method it is decided whether to connect the two midpoints via a quadratic Bézier curve, with the original vertex a_i being the control point, or via two straight line segments (b_1, a_i) and (a_i, b_2).

The next step is additional, as it does not change the visible output so much that the normal user would see any difference. However, the optimal last step is to further optimize the curve consisting out of Bézier curves and straight line elements. For this, adjacent Bézier curve segments are joint together wherever this is possible (see the result of this step in Fig. 2.9(c)).

To finish the approach the curves are filled with the belonging color repectively to the original input image. Afterwards the resulting image can

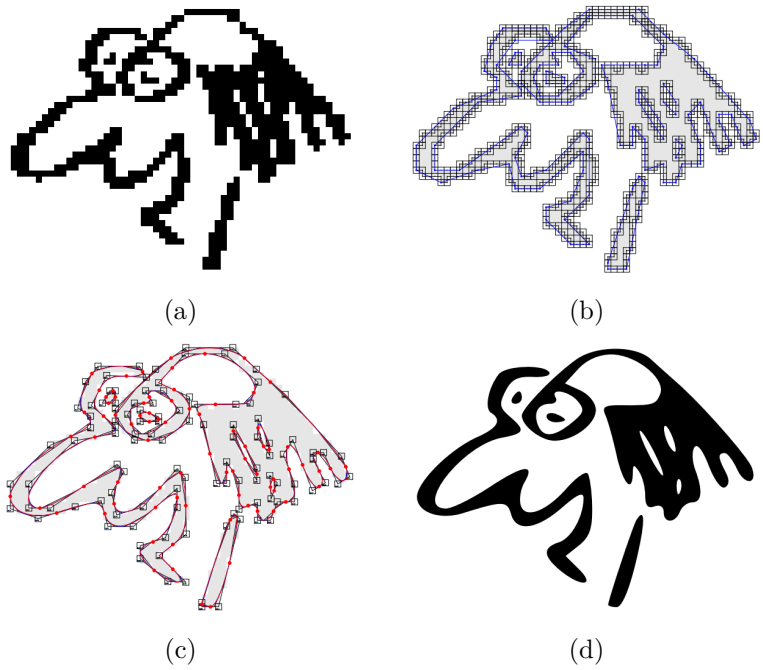


Figure 2.9: All steps of the Potrace algorithm using a test image, which is taken from the original paper [31]. The example image (a) is the original image. This image is devided into multiple closed subpaths, of which an optimal polygon is created (b). Furthermore corners are detected, Bézier curves and straight line segments are generated and the resulting contour is optimized by conjoining adjacent Bézier curves wherever possible (c). The final output image is shown in (d).

be rendered. The output image for the example is shown in Fig. 2.9(d) with all Bézier curves being connected and the regions being colored.

2.3.2 Diffusion Curves

Another vectorization algorithm was introduced by Alexandrina Orzan et al. [9]. It converts an image into several vector-based primitives to make it look more smooth-shaded. This primitives are called *diffusion curves*.

The general idea of a diffusion curve is to define different colors on either side of the curve, which smoothly change along the curve as well. An image is defined by a number of these diffusion curves whereas the color and sharpness of the curve, which is also interpreted as contour, can be defined individually. These curves also support some geometrical editing or keyframe animation. Another advantage of this curves is that they deliver a resolution independent representation of the resulting image, which makes it easily scalable without losing image information as discussed in Sec. 1.4.

The curves can either be drawn by hand or an already existing image can be traced to automatically fit the curves. What makes the diffusion curves more practical than, for example, the gradient meshes from Adobe[4] is that they offer the same complexity with additional advantages: The diffusion curves are sparse and meaningful features are corresponded by them. Furthermore, they are easy to create, manipulate and animate. If there is no need for manual editing, there still is the option of a fully automatic conversion of a bitmap image into a diffusion curve representation.

The disadvantage of this algorithm is the fact that it uses the Canny edge detection [3], which is not suitable for such small pictures as pixel art images.

The diffusion curve itself is a geometric curve defined as cubic Bézier spline. This spline is formed by a set of control points P, which can be seen in Fig. 2.10(a). Furthermore, there are two sets of color control points C_l and C_r for the color control points on the left and the right hand side of the curve (see Fig. 2.10(b)). Note that the array for the control points on one side can be bigger as the one for the other side if there are more color transitions on one of the two sides. Additionally, there is a set of blur control points Σ (see Fig. 2.10(c)) to define the smoothness of the color transition done between the two sets of color control points. Black hereby indicates a sharp transition whereas white indicates a smooth transition. The final output for the example image is shown in Fig. 2.10(d).

As mentioned before, there are different ways to create an image using diffusion curves. One way is to manually create an image by first sketching the lines where the diffusion curves should be located and then filling in the color manually on all the diffusion curves. An example of this approach is pictured in Fig. 2.11.

Another way is the fully automated approach where the algorithm converts an image into a diffusion curve representation by using the Canny detector [3] to trace the visible edges of the original input image. Afterwards each pixel-chain is approximated with a diffusion curve. In Fig. 2.12 the automatic transformation of an example image is shown.

A third approach is a mix of both approaches mentioned before. It is called “assisted” method. The user can trace parts of the original image himself and the algorithm recovers the underlying color of the original image. To provide some help for the artist there is a tool called “Active Contours”, which attracts the active contour to the highest gradient values in the input image. The curve automatically snaps to the nearest edge.

[4]See [22] for further information about the gradient mesh.

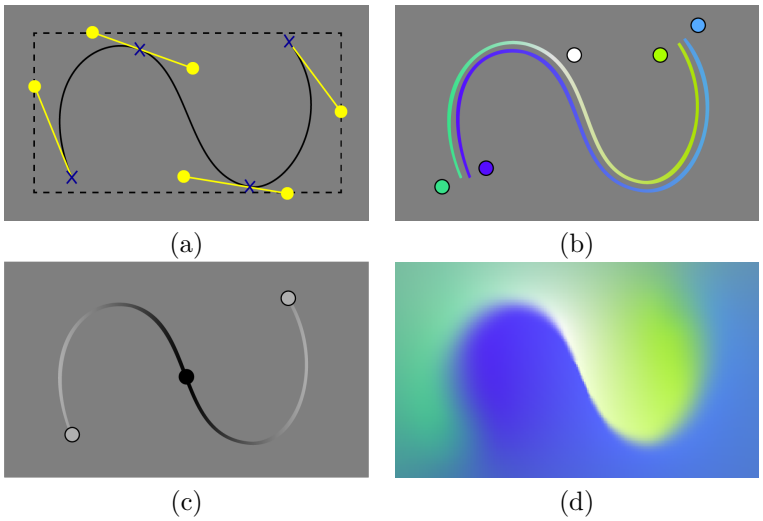


Figure 2.10: The general functionality of the diffusion curves as described in [9]. A diffusion curve is defined as a basic Bézier curve with an array of control points P as can be seen in (a). Additionally there are two arrays C_l, C_r for the color control points of the left and the right side of the curve (see (b)). Furthermore, an array of blur control points Σ is given in (c), which indicates how the control points of either side are interpolated. Black hereby indicates a sharp, white a smooth transition. The final output is shown in (d).

2.3.3 Adobe Live Trace

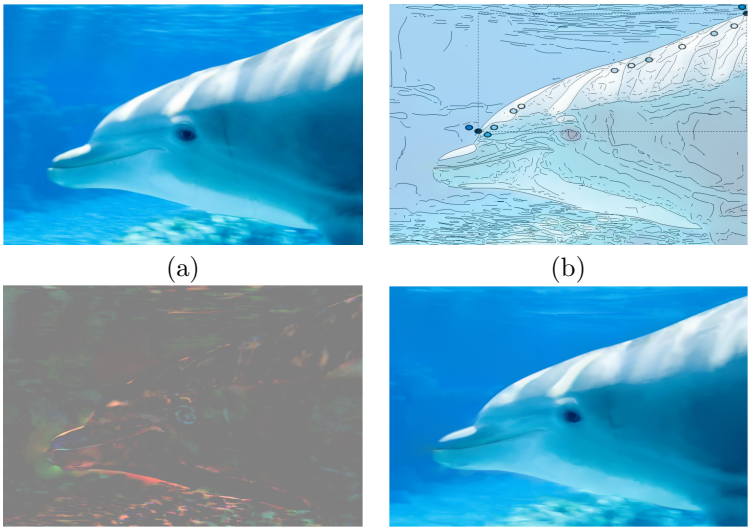
Adobe developed another tracing algorithm called *Live Trace* [19, 21, 23]. The underlying algorithm is neither Open Source nor published in any way. But similar to the other vectorization approaches it transforms a pixel graphic into a vector image, tracing the visible edges of an image. The artist or user can also do some pre-adjustments, for example choose a specific color, and one still has full control to finesse the tracing with an Image Trace panel that presents all options in one place.

The process is very easy for the user, as only one button needs to be clicked. This approach is particularly helpful when converting a sketch on paper into a vector image that can then be further modified. For this the original image is scanned or painted with a computer program like Photoshop that handles pixel images. After opening and selecting the scan or image with Illustrator there is a button, which says “Image Trace” in English or “Interaktiv Nachzeichnen” in German. By clicking this button the image is traced automatically with the Live Tracing algorithm of the program.

One can either use a predefined setting or generate a new setting where different parameters can be defined individually. If the underlying image, for example, is a gray scale image a specific threshold value can be chosen to generate a binary image. Every pixel lighter than this threshold is assumed

(a) (b) (c)

Figure 2.11: An example of manually created diffusion curves. The manual creation of an image using diffusion curves starts the same as the creation of any other image: with some sketching (a). After adjusting the sketch and positioning the curves the color is added and the plain diffusion curves look like the ones in (b). The generated output image is pictured in (c). These images are taken from the original paper [9].



(a) (b) (c) (d)

Figure 2.12: Example images of automatically created diffusion curves. The input image (a) is automatically converted into a finished diffusion curve representation (d). For a better imagination one plain diffusion curve with the color control points is pictured in (b) along with the color differences of the original image (c). The images are taken from [9].

to be white, all darker ones are assumed to be black. With a specific palette of colors one can define a number of colors that should be used to trace the underlying image. For every pixel in the original image the color of the palette with the smallest color distance is chosen. According to this

parameter also a maximum number of colors for the tracing can be set. There are some more parameters to chose like a maximal/minimal contour length or line width.

After either generating a new tracing setting or choosing a standard approach there is furthermore the possibility to convert the result into a vector representation that can be changed by the user. The individual curves with the control points are enabled and one can adjust the form of the curves as well as for example the fill color or delete/add any control points to adjust the shape even more.

What is also really important to know is that the resolution of the image has a deep impact on how the result turns out. A higher resolution will make the vector graphic look more detailed, whereas a lower resolution smooths the image out by not caring too much about smaller details. An example is shown in Fig. 2.13. The tracing of an image with a higher resolution takes longer than the tracing of an image with lower resolution but the result of the image with the higher resolution in general looks more pleasing. The execption to this general rule is an artist who wants to achieve a more plain or cartoony look. For this a lower resolution might be preferable.

2.4 Algorithm by Kopf and Lischinski

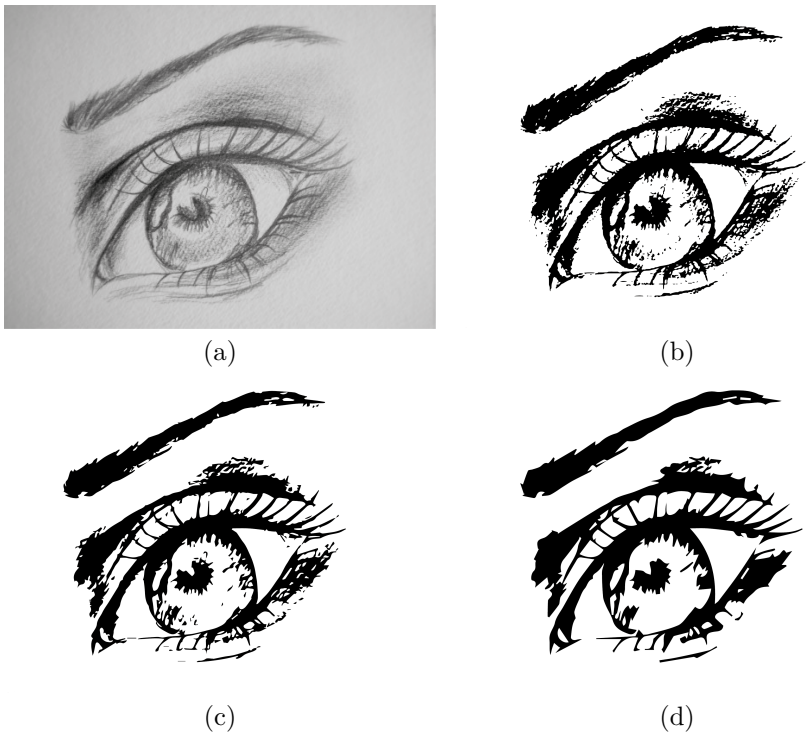
The precedent sections all explained some algorithms for image upsampling, but all of them had some issues to deal with when it comes to pixel art images. Johannes Kopf and Dani Lischinski proposed an approach [6] where they introduced a new kind of upsampling technique for the specific kind of pixel art images. In this section a short overview about this approach is given as it is explained in more detail in Chap. 3 where the reimplementation of the algorithm with Java/ImageJ is described.

The challenges Kopf and Lischinski figured out while working with such small images are not that much in number, but essential in their effect if they are not considered. All issues, that came up, can be summarized into four basic challenges:

First, every pixel matters. In such small images every pixel has its right to exist. Despite the fact that in the beginning every pixel of an image was placed by hand, there is no room for any additional pixels. So every pixel carries as much information (position and color for example) as possible.

Second, it has to be assumed that every pixel is connected to eight neighbored pixels at most. This is to make sure that even pixels that may seem to be visually disconnected under magnification are still connected in the original image.

Third, the difficulty about an eventual checkerboard pattern has to be considered. It has to be discovered which of the two colors meeting in the checkerboard pattern is assumed to be the foreground and which is assumed



(a) (b)

(c) (d)

Figure 2.13: The Adobe *Live Trace* algorithm is another way to convert a pixel image into a vector image. However, it has to be kept in mind that the resolution of the input image is important for the outcome. Here an original image with a resolution of 300 dpi (a) is traced. The resulting image can be seen in (b). The details are clear to see as there are a lot of small holes, for example, in the region of the eyebrow or the shadows around the eye. The same image, but with a smaller resolution of 150 dpi, looks like (c) when traced. The small holes close up more and more and the image does not look like a sketch any more. The result with a resolution of 72 dpi is shown in (d). Here the details are reduced to a minimum and the image looks very cartoony and plain compared to the image shown in (b).

to be the background color. According to the result the pixels in the foreground color are connected.

Last but not least, it is still very hard to say if there are jaggies in the image or if the assumed jaggies are a feature of the small image. Diagonally connected pixels, for example, can in some cases be wiggly in the resulting image, in other cases however they should be smoothed out for a better result.

To solve this difficulties, Kopf and Lischinski developed a completely new approach for vectorizing very small images. The basic steps of the algorithm are pictured in Fig. 2.14 for a better understanding. Kopf and Lischinski start of with generating a similarity graph (like the one in Fig. 2.14(b))

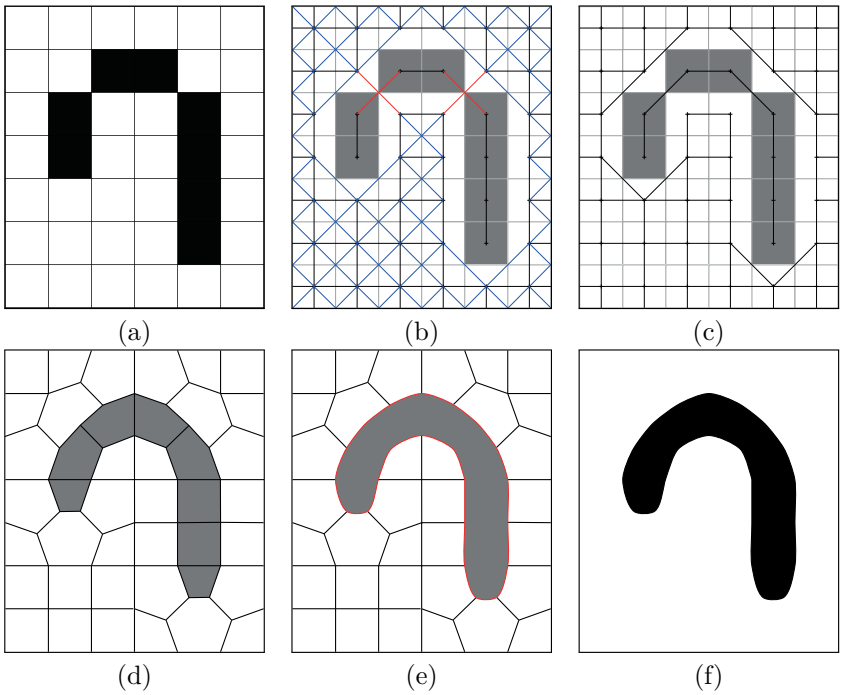


Figure 2.14: The algorithm of Kopf and Lischinski described with images for the basic steps. In (a) the input pixel art image is shown. A similarity graph is generated containing all possible connections of every pixel in (b). This similarity graph is furthermore flattened in (c) and used to generate a Voronoi diagram shown in (d). At the visible edges of this Voronoi diagram the quadratic B-Splines are fitted (see (e)) for producing the final output image, which is shown in (f).

depending on the original input image pictured in Fig. 2.14(a). For this all pixels are connected with their eight neighbors but only if their color is the same. Pixels with a different color are never connected. In case of a checkerboard pattern in a 2×2 pixel region there are two diagonal connections for both colors as can be seen in Fig. 2.14(b), indicated by the red diagonal lines. Only one of these two connections can stay and the other one has to be deleted. The connection that stays should always be the edge, which connects the foreground pixels. To find out which edge is connecting the foreground pixels, an own heuristic is implemented. These heuristics determine whether the edge is connecting an island pixel to a larger part (island-heuristic), if the connection is part of a longer curve connection (curve-heuristic), or if the edge connects two pixels of a sparse region (sparse-heuristic).[5] After executing the heuristic and identifying the foreground connection, the edge connecting the background pixels can be deleted. Also both diagonal

[5]These heuristics are explained in detail in Sec. 3.2.2.

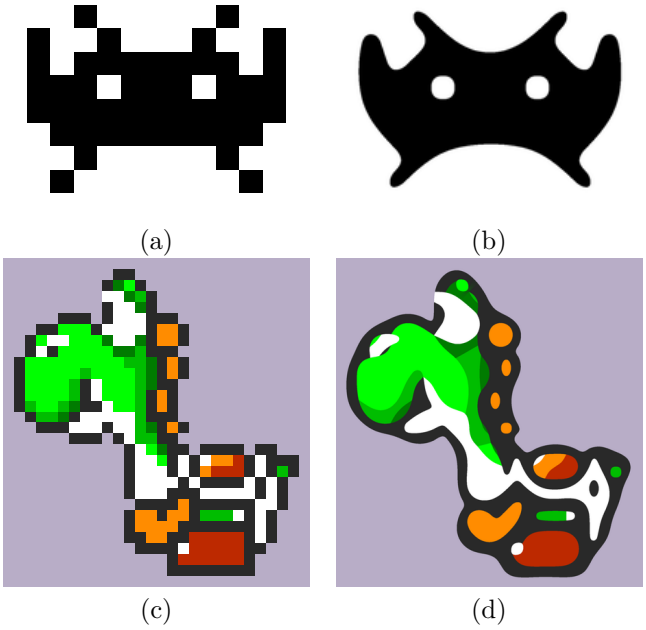


Figure 2.15: Example images of the *Depixelizing Pixel Art* algorithm. After tracing the example input image in (a) the algorithm leads to the resulting image shown in (b). The same is done with the slightly larger example image "Yoshi" (c) again. The result, which is pictured in (d), looks very clean and smooth compared to (c).

connections in a 2×2 pixel region, where all pixels have the same color, are deleted. The result after this step is a planar similarity graph like the one in Fig. 2.14(c). Using this planar similarity graph a Voronoi diagram is generated by splitting all remaining diagonal connections in the middle, generating new intersection points and reconnecting all seeds to the nearest neighbors, as shown in Fig. 2.14(d). The final step is then to fit quadratic B-Splines to the visible edges of Fig. 2.14(e), which leads to the final output shown in Fig. 2.14(f).

This algorithm delivers really good results especially for pixel art images. While it works for virtually all kinds of images, the computation time for regular images with higher resolution is too long. In Fig. 2.15 there are two more examples of the "Depixelizing Pixel Art" algorithm.

This approach was used as starting point for the master project, which was a reimplementation of the algorithm using Java/ImageJ, including some adjustments on the algorithm and exporting the image as vector representation. All the steps, changes and difficulties are described in Chaps. 3 and 4.

Chapter 3

Java/ImageJ Implementation

In this section the algorithm which was implemented as master project, using Java/ImageJ [36], is described and explained in detail. All steps of the approach are also done by Kopf and Lischinski, however, some adjustments had to be done in Java to accomplish the same result.

3.1 Generating the Reference Image

The original input image, which is a pixel image, is opened with ImageJ and displayed as a two-dimensional image matrix $I(u, v)$ as shown in Fig. 3.1. The width of the image is referred to as M and the height of the image is referred to as N with $I : M \times N \mapsto \mathbb{R}$.

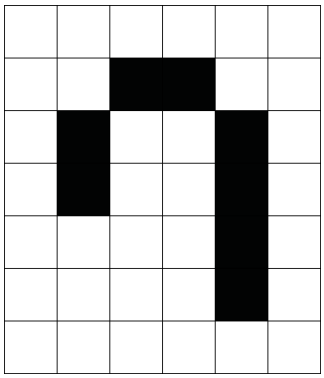
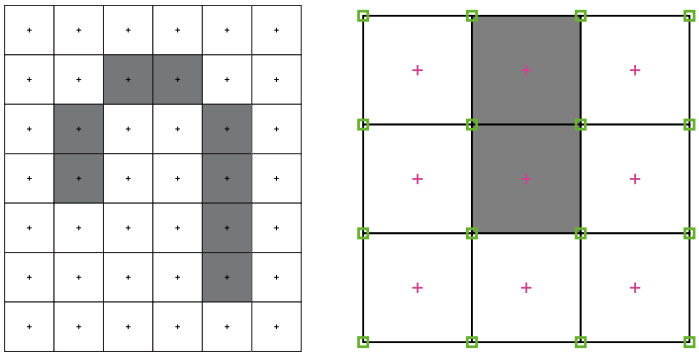


Figure 3.1: The input image is represented as two dimensional image matrix.



(a) (b)

Figure 3.2: The pixel centers and pixel corners are marked for each pixel in the image. In (a) all pixel centers of the image are marked. A detailed version of a pixel set is displayed in (b). The center points of the pixels are shown as a magenta colored crosses and the corner points are surrounded by green colored squares.

3.2 Generating the Similarity Graph

After opening the image, the next step is to generate a similarity graph G_S. A graph $G = \langle S, E \rangle$ consists of a set of seeds S and a set of edges E that connect the seeds. A similarity graph G_S graphically shows the connections between each pixel s_0 and its multiple neighbor seeds (for example s_1), whereby only pixels with the same color or with a very similar one (depending on a predefined threshold) are connected via an edge e_{s_0,s_1} in the pixel center (see Figure 3.2). The pixel centers of the original image representation $I(u, v)$ are described as

$$\boldsymbol{u} = (u + 0.5, v + 0.5), \tag{3.1}$$

with $u \in [0, M-1]$ and $v \in [0, N-1]$ because it is worked with overlays in ImageJ and these ones have the origin placed in the upper left corner, which is also a pixel corner. The center points are marked in Fig. 3.2 (b) as magenta colored crosses and are referred to as seeds (S). All seeds that are considered neighbors to one specific seed v are described as $\mathcal{N}(v)$ and can include eight neighbors at maximum with a *8-neighborhood-connection*. The corner points of each pixel are described as

$$x_k(u, v) := \begin{cases} \binom{u}{v} & \text{for } k = 0, \\ \binom{u+1}{v} & \text{for } k = 1, \\ \binom{u+1}{v+1} & \text{for } k = 2, \\ \binom{u}{v+1} & \text{for } k = 3 \end{cases} \tag{3.2}$$

and are marked in Fig. 3.2 (b) as green colored squares.

Neighbored pixels can share up to two pixel corners if they have a straight connection and one at maximum if they are diagonally connected. Depending on this, one pixel corner can be a pixel corner of three other pixels as a maximum, which can be described as

$$\begin{aligned}
x_0(u,v) &= x_1(u-1,v) &&= x_2(u-1,v-1) &&= x_3(u,v-1),\\
x_1(u,v) &= x_0(u+1,v-1) &&= x_2(u,v-1) &&= x_3(u+1,v-1),\\
x_2(u,v) &= x_0(u+1,v+1) &&= x_1(u,v+1) &&= x_3(u+1,v),\\
x_3(u,v) &= x_0(u,v+1) &&= x_1(u-1,v+1) &&= x_2(u-1,v).
\end{aligned} \tag{3.3}$$

There are basically two steps included for generating a similarity graph $G_S = \langle S_S, E_S \rangle$, which is initialized with an Array or a List of seeds as S_S and an empty List of edges E_S in the beginning. In this case the seeds to work with are all the pixel centers $\boldsymbol{u}$ of the inpu image I.

3.2.1 Generating All Possible Connections

To connect one specific seed with all eight neighbors the first thought is to generate eight edges e from a specific seed $\boldsymbol{s} = (u, v)$ to all the neighbored seeds $\boldsymbol{s}_j = (u_j, v_j)$. But as this similarity graph is a non-directed, non-weighted graph representation, it does not matter if the connection is generated as e_{s,s_j} or $e_{s_j,s}$, because this is basically the same edge just created in two different directions. What actually matters is if the edge already exists. To ensure that no edge is generated twice, each pixel only generates four edges at most in predefined directions with the purpose to connect a specific seed $\boldsymbol{s}$ with just four of its connected neighbor seeds, which are $\boldsymbol{s}_0 = (u+1, v-1)$, $\boldsymbol{s}_1 = (u+1, v)$, $\boldsymbol{s}_2 = (u+1, v+1)$ and $\boldsymbol{s}_3 = (u, v+1)$ in specific, if the pixel is a non-border pixel. If the seed is located at the border each of these edges is generated only if the second seed, which should be connected to the seed $\boldsymbol{s}$, exists. In general all edges of a specific seed $\boldsymbol{s}$ can be defined as

$$E(S) = \{e_{s,j} \mid j \in [0,3] \wedge s_j \in S\}. \tag{3.4}$$

If the color of the two pixels which should be combined is the same, the edge e_{s,s_j} between the seeds s and s_j is added into the edge set E_S of the similarity graph. The general procedure is described as pseudo code in Alg. 3.1. If all possible edges (without considering the fact of only connecting pixels with the same color) would be connected in the similarity graph the maximal amount of edges can be described as

$$A_E = (5 + (N-2) \cdot 4) \cdot (M-1) + N - 1. \tag{3.5}$$

Algorithm 3.1: The general procedure of generating the similarity graph with using the dimensions of width (M) and height (N) of the input image.

```
1: function GenerateSimGraph()
2:     S_S ← {}, E_S ← {}, G_S = ⟨S_S, E_S⟩
3:     for (u, v), with u ∈ M, v ∈ N do
4:         S_S ← u
5:     end for
6:     for all s in S_S do
7:         (s_0, s_1, s_2, s_3) = getNeighborPixels(s)
8:         if s_0 ∈ S_S then
9:             if s_0.color is similar to s.color then
10:                E_S ∪ e_{s,s_0}
11:            end if
12:        end if
13:        if s_1 ∈ S_S then
14:            if s_1.color is similar to s.color then
15:                E_S ∪ e_{s,s_1}
16:            end if
17:        end if
18:        if s_2 ∈ S_S then
19:            if s_2.color is similar to s.color then
20:                E_S ∪ e_{s,s_2}
21:            end if
22:        end if
23:        if s_3 ∈ S_S then
24:            if s_3.color is similar to s.color then
25:                E_S ∪ e_{s,s_3}
26:            end if
27:        end if
28:    end for
29: return G_S
30: end function
```

3.2.2 Resolving and Deleting the Diagonal Connections

After this step all crossing diagonal edges between four pixels have to be resolved, which is called simplifying the graph. Therefore a pixel set of 2×2 pixels is extracted from the original image representation and the diagonal pixels are compared. As all crossing blue lines are connecting 4 pixels with the same color, both of the diagonal connections can be removed from the edge set E_S. If there is only one diagonal edge in a 2×2 pixel set, this edge is kept, as can be seen in Fig. 3.4(b). However, if the extracted 2×2 section includes two diagonal connections that represent two pixel sets of

different colors, both diagonals were generated but have to be reworked to delete one edge. This is shown in Fig. 3.4(c) with the red crossing lines. Only one of the diagonals must be kept, whereas the other one has to be deleted from the edge set because there can only be one diagonal edge in a 2×2 pixel set at most. To determine which connection has to be removed an own heuristic has been implemented, calculating the sum of three different weights for each diagonal and keeping the one with the larger result. The three heuristic parts are the following:

Curve

If an edge is part of a curve, connecting several seeds in a line, the edge should be kept. A curve is a sequence of edges in the graph which only connects valence-2 seeds. Valence-2 seeds are seeds that have two connected neighbors at max. The curve heuristic calculates the length of both curves, of which the two diagonal connections are part of, and votes for keeping the diagonal which is part of the longer curve. An example is shown in Fig. 3.4(c), where the two red lines connecting the black pixels are both part of the same curve of length 7. The red diagonals connecting the white pixels only have a length of 1 as they have no valence-2 neighbours. Therefore the curve heuristic suggests to keep the connection between the black pixels.

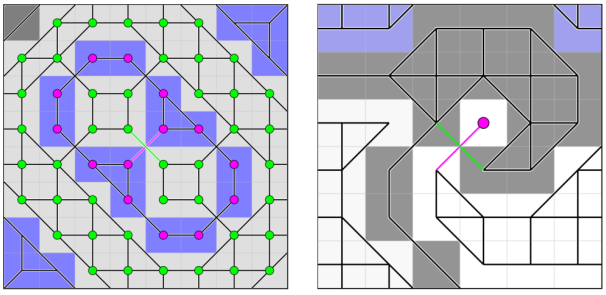
Sparse Pixel

The sparse pixel heuristic measures the size of the component connected to the diagonal. As can be seen in Fig. 3.3(a) the magenta colored component has a smaller size (includes less seeds) than the green component. As a result the green colored component is supposed to be part of the background. The green edge is deleted, the magenta edge is kept. The heuristic implements exactly the procedure that a human eye automatically performs, as humans tend to recognize the sparser color as foreground color, whereas the more capacious region is imagined as background.

Island

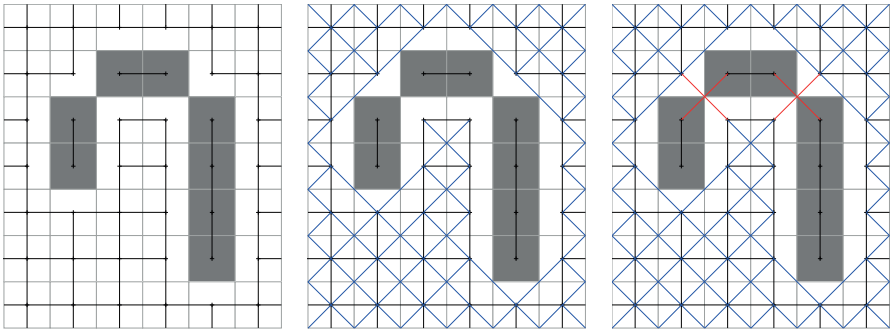
Figure 3.3(b) shows the situation of a valence-1 node (a node with only one neighbored connection) being connected to only one diagonal neighbor. This diagonal connection is part of an edge pair that has to be resolved. In case of deleting the magenta edge, a single island pixel would be generated. As the image should not be cut into too many small regions, the heuristic votes for deleting the green diagonal and keeping the magenta one, connecting the single island pixel to a greater number of all connected seeds.

Summing up the results of all three parts of the heuristic the diagonal edge with the smaller result is deleted. After all the diagonal edge pairs are resolved, the resulting planar similarity graph looks like Fig. 3.5(a).



(a) (b)

Figure 3.3: Two of the heuristics explained by using example images. The sparse pixel heuristic in (a) shows that the magenta colored component is sparser than the green colored component and therefore votes for keeping the magenta connection. The island heuristic shown in (b) detects that a single island pixel is created if the magenta connection would be deleted. The island heuristic suggests to keep the magenta connection.

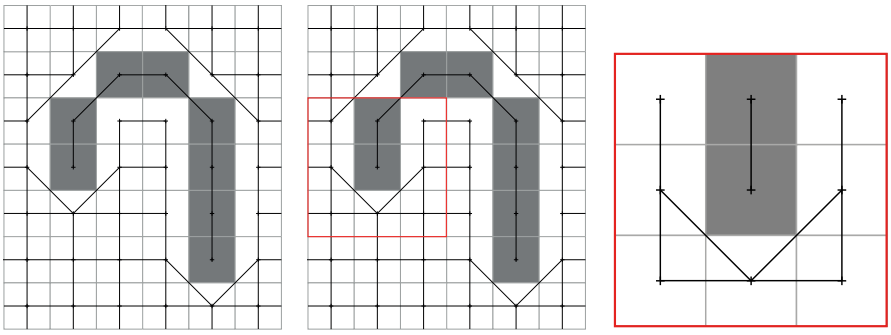


(a) (b) (c)

Figure 3.4: The detailed steps of generating the similarity graph. The edges for the straight neighbors are drawn in a black color in (a), the diagonal neighbors that do not have to be reworked are shown in a blue color (see (b)) and the diagonal neighbors that have to be reworked are indicated by a red color in (c).

3.3 Generating the Voronoi Diagram

To be able to generate a Voronoi diagram it is important to know what a Voronoi diagram is in the first place and what it is used for. The definition of a Voronoi diagram and the concrete implementation description are part of this section.



(a) (b) (c)

Figure 3.5: The planar similarity graph with the crossing edges being resolved. The final similarity graph is shown in (a). A smaller section of this graph which is marked in (b) is taken for further explanations. The example image for further explanation is shown in more detail in (c).

3.3.1 General Information

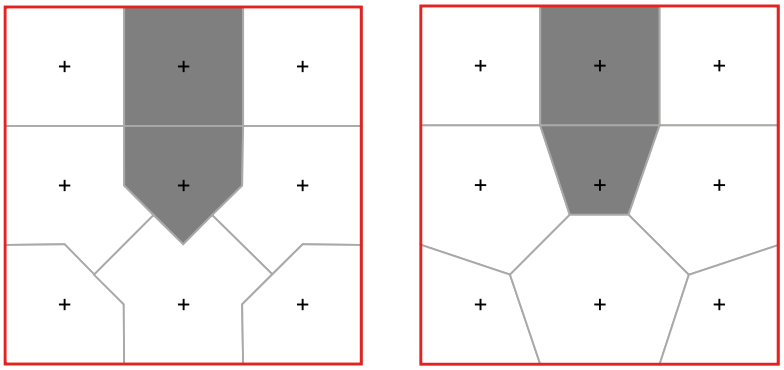
A Voronoi diagram in general is a partitioning of a plane into regions based on distance to points in a specific subset of the plane. That set of points, which are also called *seeds*, is specified beforehand. For each seed there is a corresponding region consisting of all points closer to that seed than to any other. These regions are called Voronoi cells. The cells are defined as

$$R_n = \{n \in N \mid d(n,k) \leq d(n,j)\}, \qquad \text{for } k,j \in N, j \neq k. \tag{3.6}$$

So the region R_n is defined as every seed n of all seeds in the graph N of which the distance to a specific seed k is smaller than to any other seed j in the graph, with k must not be equal to j. In Fig. 3.6(a) such a Voronoi diagram is displayed. The black crosses are associated with the seeds in this case and the gray lines indicate the borders of the Voronoi cells. It has to be mentioned that an accurate Voronoi diagram consists of cells which can be of either a convex or a concave form.

3.3.2 Creating the Voronoi Diagram from the Similarity Graph

For this approach the version of a simplified Voronoi diagram is used, to not get any concave Voronoi cells. To explain how the Voronoi diagram is generated from the similarity graph in the next step of the algorithm, a smaller section of the original similarity graph is used (see Figs. 3.5(a) and 3.5(b)).



(a) (b)

Figure 3.6: Showing the difference of an accurate and a simplified Voronoi diagram. In (a) an accurate Voronoi diagram is shown, consisting of both, convex and concave elements. In (b) a simplified version, which is only consisting of convex elements, is printed.

Initializing the Seeds

The Voronoi diagram is also represented as a graph $G_V = \langle S_V, E_V \rangle$ with $E_V \leftarrow \{\}$ in the beginning and S_V is initialized with the corner points of the original input image. So for every pixel in $I(u, v)$ the upper left corner $x_0(u, v)$ is inserted in the seed set S_V of the Voronoi diagram. As at this point not all seeds are included into S_V some special cases have to be considered. If $u = M - 1$ the upper right corner $x_1(u, v)$ is inserted in S_V and used as seed as well and if $v = N - 1$ the corner on the bottom left $x_3(u, v)$ is handled the same way. As a last seed the bottom right corner $x_2(u, v)$ is included into S_V in the case of $u = M - 1$ and $v = N - 1$. This includes all corner points at the right/bottom border of the image as well.

Generating the Edges

As a next step, all diagonal edges of the similarity graph are cut in half (see Fig. 3.7(a)). As the position of the cut is always a corner position, exactly four pixels meet at this position and all four of their pixel centers $\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3, \boldsymbol{u}_4$ need to be known for further steps. As two of them are the ones that are connected via the diagonal edge, the pixel centers of these are already known as $\boldsymbol{u}_1 = (x_1, y_1)$ and $\boldsymbol{u}_2 = (x_2, y_2)$. According to this, the two remaining pixel centers can be specified as

$$\boldsymbol{u}_3 = (x_1, y_2), \qquad \boldsymbol{u}_4 = (x_2, y_1). \tag{3.7}$$

As the position of the cut mentioned in the beginning is a corner point, this position can be found by identifying the common corner of all four

pixels. This common corner $\boldsymbol{x}_k(u,v)$ with $k \in [0,3]$ is also a seed in the newly generated seed set of the Voronoi diagram S_V, which is used for further steps.

As the diagonal edge of the similarity graph connects $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$, a new line is generated from $\boldsymbol{u}_3$ to $\boldsymbol{u}_4$. This is also displayed in Fig. 3.7(b). Afterwards the seed $\boldsymbol{x}_k(u,v)$ is deleted from S_V and two new seeds $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ are generated at the same position where $\boldsymbol{x}_k(u,v)$ was located initially. These two seeds are moved on the newly generated line exactly to the middle position between the original position and the two neighbored pixel centers, which can be seen in Fig. 3.7(c). In the specific case of this algorithm the new positions can be simplified as

$$\boldsymbol{s}_1 = (u + 0.25, v + 0.25), \qquad \boldsymbol{s}_2 = (u - 0.25, v - 0.25) \tag{3.8}$$

if the newly generated diagonal line segment (u_3, u_4) is connecting the bottom left and the top right pixel of the 2×2 pixel block. Otherwise, if the diagonal connects the bottom right pixel an the top left pixel of the 2×2 block the new positions of $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ are initialized as

$$\boldsymbol{s}_1 = (u - 0.25, v + 0.25), \qquad \boldsymbol{s}_2 = (u + 0.25, v - 0.25). \tag{3.9}$$

After this step is done, the two new seeds are connected via an edge $e = e_{s_1,s_2}$, which is included in the edge set E_V (see Fig. 3.7(d)).

The last steps to finish the Voronoi diagram are to connect the newly generated seeds $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ to two other seeds in the graph, to be fully connected. For this the distances of the new seeds and all other seeds in S_V are calculated and the two seeds with the smallest distance are chosen to generate a new edge. Each seed $\boldsymbol{s}$ in S_V, which is generated because of a diagonal edge being cut, has exactly three neighbors connected via an edge. The seeds that are located at a corner position of the original image have four connections. In Fig. 3.7(e) all seeds of S_V are shown and in Fig. 3.7(f) all connections have been initialized correctly. The resulting Voronoi diagram can be seen in Fig. 3.6(b).

3.4 Generating the Small Shapes from the Voronoi Diagram

To be able to detect even larger shapes in the image first of all, each Voronoi cell has to be described as a shape itself. Such a shape consists of an outer contour made up of a list of adjacent seeds and the related connecting edges. As in the resulting Voronoi diagram representation G_V there is no information about the individual cells itself, a minimalistic Dijkstra algorithm is used to generate all small shapes of the Voronoi diagram.

The Dijkstra algorithm is an approach to find the shortest path between an initial node and an end node in a graphical representation. As with the

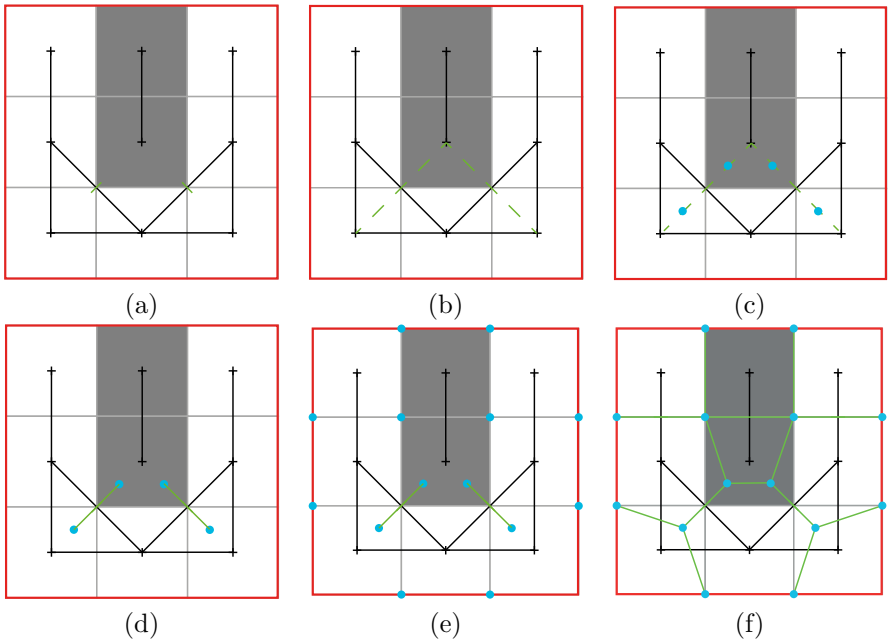


Figure 3.7: This images show how the Voronoi diagram is generated from the given similarity graph. In (a) the diagonal connections of the similarity graph are cut in half and at the position of the cut the seed s is deleted from S_V. After this a line, which is perpendicular to the diagonal, is generated in (b) and newly generated seeds are placed on this line, located on either side of the diagonal (c). These new seeds are connected by an edge (d). Seeds that were generated because of an edge being cut are furthermore connected to two more seeds of S. For this, the two minimal distances between the new seed and any seed in S are calculated and the new seed is connected to these two seeds. Any new seed is therefore connected to three other seeds at most. Seeds that lie on a corner position of the original image can be connected to four other seeds at most. In (e) all seeds that have to be connected are displayed as blue dots and in (f) the finished Voronoi diagram is visible.

Voronoi diagram a non-weighted, non-directed graph is used, only the step count from the initial node to the end node is considered.

To generate a small shape from the Voronoi diagram, first a copy of the graph needs to be created. Next, two seeds which are connected via an edge are used to start with. It does not matter if these two seeds are located at the border of the graph or anywhere in the middle. In Fig. 3.8(a) the Voronoi diagram is shown with two seeds marked. The green seed S is the source seed and the magenta seed is the target seed. The positions of both of these seeds are known.

The next step of the approach is to delete the direct connection, the edge between the two seeds, from the graphical representation. The reason

for this is the following: If this connection stays in the graph, the shortest path between the source and the target seed would be this direct connection and a shape would never be found.

In the next step, all neighbors of the current seed S are searched for and it is checked if one of the neighbor seed is the actual target seed. If this is not the case the seed with the smallest distance to S is saved in a Hashmap. As all three neighbors have a distance of 1 in the first step, all three detected neighbor seeds A, B and C are saved to search from their position for the target seed. As an additional important information, the current seed S is saved as previous seed for A, B and C. This approach is shown in Fig. 3.8(c). The light blue color of a seed indicates that this seed is in the list of seeds of which the neighbors have to be detected and tested for being the target seed.

In the next iteration the same procedure happens with the current seed being moved from S to A. So all neighbor seeds of A are determined, whereat the connection from the new current seed A to its previous seed S is not stored. After that it is checked, whether the target seed is one of them or not and if not, all neighbors are saved in the list of seeds to be searched forward. The current node again is stored as previous seed for the new found seeds D and E. This second iteration can be seen in Fig. 3.8(d). Two more iterations where done in exactly the same way with using B and C as current seeds.

The current seed in the fifth iteration is now D. As one of the neighbors of D is the target seed, the search can be stopped and the shortest path has been found as shown in Fig. 3.8(e). At this point all seeds of the shape are known, as the previous seed for every current seed was stored. So to generate the connecting edges, which make up the outer contour of the new shape, basically the way has to be gone in the opposite direction by starting at the end point and connecting this seed via an edge to its previous seed. The previous seed afterwards becomes the current seed and is connected to its previous seed again via an edge. This procedure is repeated until the last edge from the current seed to the initial seed is generated.

For the last step, the edge from the initial seed to the target seed is included into the edges of the shapes outer contour to make up a closed contour. The final shape is then fully specified by its linked seeds S, A, D and I and all the connecting outer contour edges which can be seen in Fig. 3.8(f).

To make sure that all shapes of the Voronoi diagram are found but none of them exists twice, a special procedure is executed after every shape generation. Edges of the shape that connect border seeds in the Voronoi diagram are deleted right after the shape generation, because these edges cannot be part of a second shape. Edges of the newly generated shape which already exist in another shape are deleted as well, because of the fact that no edge can be used in more than two different shapes. As edges are deleted after every new shape generation at some point seeds with no connections

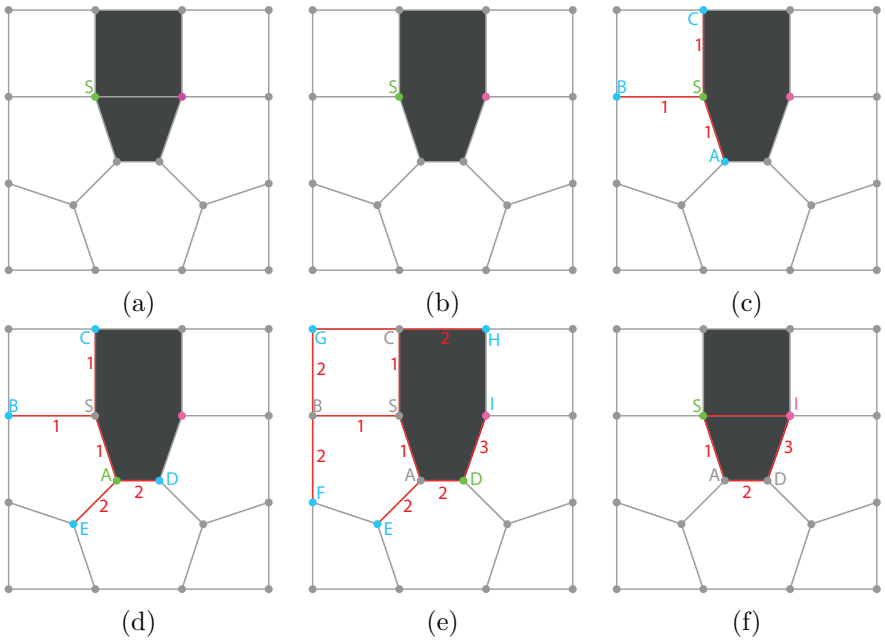


Figure 3.8: The generation of a shape using the Dijkstra algorithm. The first step is to determine the initial seed S and the target seed, which is shown in (a), the direct connection is deleted in (b) and the neighbors of the current node S are found. The current node is then moved to the neighbor with the smallest distance. As all three of the found neighbors have the same distance, all three are saved as further current nodes which is indicated with the light blue color of the seeds in (c). After the second iteration the current node is A and the new found seeds D and F are also stored as further current seeds (see (d)). After three more iterations one neighbor of the current node is the actual target node and as this happens the searching can be stopped immediately in (e). The final shape is then defined by the seeds that were stored as previous seeds of the current seed, as well as the target and the initial seed. The edges are generated by going back to the initial seed step by step as can be seen in (f).

to neighbored seeds exist in the Voronoi diagram. These seeds are deleted as well. The reason behind this is to not generate a shape twice and to not disturb the generation of the shapes with the remaining seeds and edges in the graph. The procedure of generating shapes is repeated until no edges and seeds are left in the initial Voronoi diagram.

The outcome of this step looks exactly like the resulting image of the Voronoi diagram but with the big difference that now the image is not represented as a graph anymore, but as a number of shapes. So what happened is that every single Voronoi cell has transformed into a real shape being described by an outer contour. This outer contour is a list of linked seeds as well as a list of consecutive edges at the same time.

3.5 Generating the Big Shapes from the Small Shapes

The generation of the small shapes has the advantage that every shape is initialized with an associated color. This color is determined by calculating the average x and y position $\overline{x}, \overline{y}$ of all seeds s, which are m in number, in the shape's outer contour as

$$\overline{x} = \frac{1}{m} \cdot \sum_{i=0}^{m-1} s_{i,x}, \qquad \overline{y} \ = \frac{1}{m} \cdot \sum_{i=0}^{m-1} s_{i,y} \tag{3.10}$$

and getting the color of the original image exactly at this average position. This color information is relevant for the next part of the algorithm.

All small shapes that are connected via a common edge and have the same color are combined to one larger shape. The practice to do so is described here and shown in Fig. 3.9.

In the beginning, one small shape out of all the shapes that were generated in the previous step is taken as initial shape. Afterwards all neighbored shapes, which have the same color and share an edge with the initial shape, are temporarily stored in a list as shown in Fig. 3.9(a) with the top shape being assumed the initial shape.

The last step of the unification is to generate a new shape from all edges of the initial and the neighbored shapes (see Fig. 3.9(b)). This is done for one neighbor at a time, repeating as many times as necessary to combine all neighbored shapes into one resulting big shape. To unify two shapes, all edges, except the common edge of both shapes, are included into the new shapes outer contour. As this edge list is not sorted, one initial edge is taken out of the pool of edges. It does not matter which one is taken as the contour is always a closed one. The chosen edge consists of two seeds. One seed is assumed to be the final seed and the second seed is taken as the current seed. This initial edge is then deleted from the unsorted and added to a sorted contour list. Afterwards the list of unsorted edges is searched for the second edge including the current seed. There always have to be two edges sharing one seed and these edges are adjacent. After finding this specific edge, which includes the current seed as well, this edge is also deleted from the unsorted and added into the sorted contour list. The second seed, which is not the shared one, is finally taken as new current seed. This procedure is repeated as many times as necessary to completely empty the unsorted edge list. After combining two shapes to one new shape the initial small shapes are deleted from the shape representation working with. Duplicated representations of the graph should be used for such calculations to not lose any important information of the former calculations.

The same example as before is shown in Fig. 3.9(c) after the first iteration. The new initial shape (in this case the largest one) should be further

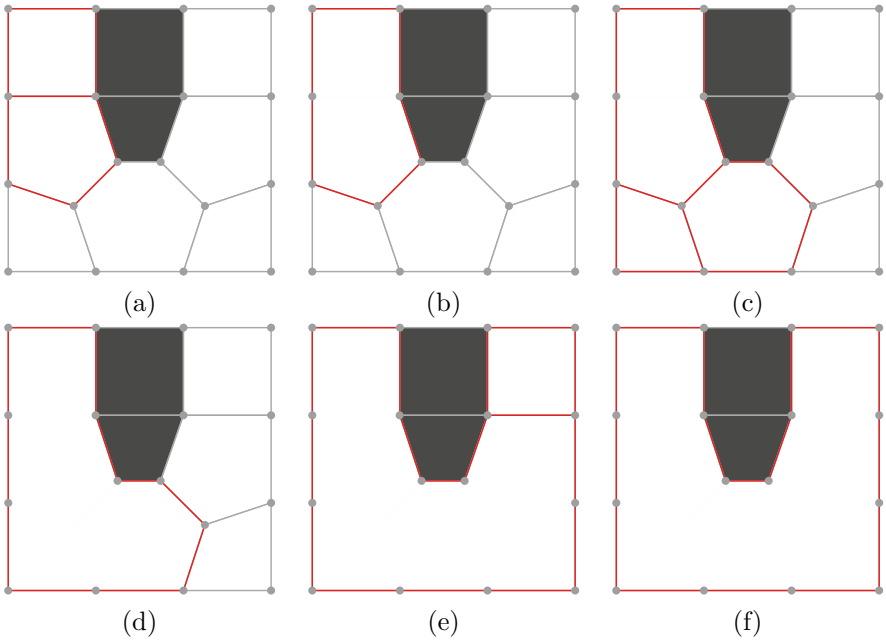
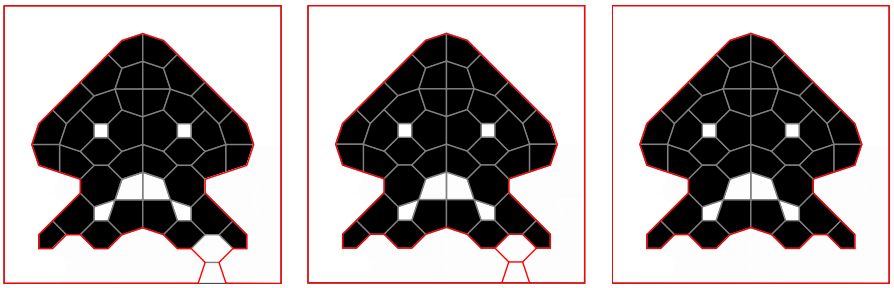


Figure 3.9: The basic approach to combine two shapes. An initial shape (in this case the top one) is determined in (a) and all neighbored shapes (that share a common edge) are found in (a) as well. A new shape is created out of all edges of both shapes, except the common edge. This edge list is sorted for a better comparison and to order the seeds. The final shape is shown in (b). Further examples are shown in (c) and (e) where shapes that should be combined are shown. In (d) and (f) the resulting single shape is printed. After the combination of the resulting image in (f) no more combinations of the white shape with any neighbors can be done.

connected to two other neighbored shapes on the bottom. For this case, also the common edges are deleted and the remaining edges are being sorted to make calculations and comparison easier (see Fig. 3.9(d)). The final step for this specific example image is shown in Fig. 3.9(e), where the big shape is combined with a smaller one to generate a big shape that includes all white Voronoi cells (see Fig. 3.9(f)). If no more neighbors with the same color are found the big shape is completed and the next initial shape is taken into account. The Dijkstra is running recursively as long as no single small shape is left in the shape representation and all big shapes have been generated successfully.

With this approach there is a small difficulty to overcome. To explain this difficulty, a more complicated example is shown in Fig. 3.10. As can be seen in Fig. 3.10(a) the current shape, which is worked with, is the big shape surrounded by the red line. The red line indicates the outer contour. Figure 3.10(b) shows, that there are two more shapes that can be combined



(a) (b) (c)

Figure 3.10: With this image the difficulty of combining two or more shapes and not only generating an outer, but also an inner contour is shown. In (a) the initial shape is marked via the red outer contour. In (b) two more shapes are shown, which can be combined with the initial shape. In (c) it is clearly visible that after the combination of the three shapes an outer contour as well as an inner contour has been created.

with the big shape. With this example as the combination is completed the shape is not only defined by an outer, but also by an inner contour. While sorting all edges in the unsorted edge list the seed, which was determined as initial seed, is found after some time again with the closing edge of the outer contour. But the unsorted edge list is not empty because some edges are not part of the outer contour any more, but they are forming an inner contour. The resulting perception of this fact is now, as the first contour is closed while sorting the unsorted edge list but there are still edges in this list, an inner contour has to be defined. There can be multiple inner contours of one shape. These are all saved in a separate list.

3.6 Fitting the Splines

With the big shapes all being created from the smaller shapes, the spline generation can be initialized. In general, this is not complicated, because all shapes include a linked list which includes all seeds of the outer contour. The spline fitting is done to make the resulting image look smoother and not as edgy as the Voronoi diagram does. In this chapter only the procedure of calculating the different spline interpolations is described. The resulting images and eventual problems are described in Sec. 3.7.

3.6.1 Catmull-Rom Splines

The Catmull-Rom splines [1] are a family of local interpolation splines. This type of spline is defined by several *control points* $\boldsymbol{p}_i = (x_i, y_i)$ through which the line passes during the interpolation process. These splines are generated by a piecewise cubic interpolation of four control points (see Fig. 3.11 for a

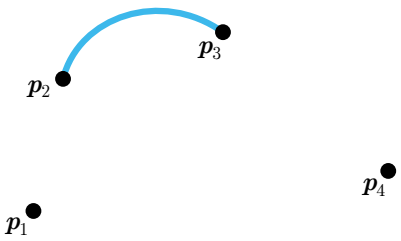


Figure 3.11: Spline interpolation with Catmull-Rom Splines.

better understanding). The line only passes through the two control points in the middle. Given four control points $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, $\boldsymbol{p}_3$ and $\boldsymbol{p}_4$, which are all seeds of the shape's outer contour, the connected line is drawn between $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$. To calculate the interpolation points between these two points, the function $C(t)$ is used, which produces x- and y-coordinates located between $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$. The variable t is a steadily increasing number between 0 and 1. This means if $t = 0$ the x- and y-coordinates calculated are exactly these of $\boldsymbol{p}_2$ and if $t = 1$ the resulting coordinates are the ones of $\boldsymbol{p}_3$. The smaller the increasing steps of t are, the more interpolation points are calculated. The smoothness of the outer contours' appearance is directly dependent on the number of interpolation points. This means that, the more interpolation points there are, the smoother the outcome will look.

The function $C(t)$ with $t \in [0, 1]$ can be written as either

$$C(t) = 0.5 \cdot \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \boldsymbol{p}_3 \\ \boldsymbol{p}_4 \end{bmatrix} \tag{3.11}$$

or as an equation in scalar form

$$\begin{aligned} C(t) = 0.5 \cdot (t^3 \cdot (-\boldsymbol{p}_1 + 3\boldsymbol{p}_2 - 3\boldsymbol{p}_3 + \boldsymbol{p}_4) + \\ t^2 \cdot (2\boldsymbol{p}_1 - 5\boldsymbol{p}_2 + 4\boldsymbol{p}_3 - \boldsymbol{p}_4) + \\ t \cdot (-\boldsymbol{p}_1 + \boldsymbol{p}_3) + 2\boldsymbol{p}_2), \end{aligned} \tag{3.12}$$

which furthermore can be divided into the x- and y-function

$$\begin{aligned} x(t) = 0.5 \cdot (t^3 \cdot (-x_1 + 3x_2 - 3x_3 + x_4) + \\ t^2 \cdot (2x_1 - 5x_2 + 4x_3 - x_4) + \\ t \cdot (-x_1 + x_3) + 2x_2), \end{aligned} \tag{3.13}$$

$$\begin{aligned} y(t) = 0.5 \cdot (t^3 \cdot (-y_1 + 3y_2 - 3y_3 + y_4) + \\ t^2 \cdot (2y_1 - 5y_2 + 4y_3 - y_4) + \\ t \cdot (-y_1 + y_3) + 2y_2). \end{aligned} \tag{3.14}$$

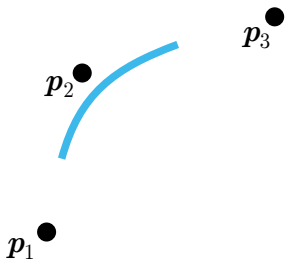


Figure 3.12: Spline interpolation with the quadratic B-Spline.

3.6.2 Quadratic B-Splines

B-Spline is short written for Basis-Spline. B-Splines [10] have certain advantages over Bézier curves as their degree is not dependent on the number of control points and the individual segments of a B-Spline are easy to connect. As the B-Spline is an approximating curve it has to be defined by *control points*. The control points in this case are all seeds of a shape's outer contour. In addition to the control points one has to define so called *knots*. The knots are real numbers that offer additional control over the curve. There are several types of B-Splines but the two most important ones are the *uniform* and the *nonuniform* B-Splines.

Using the uniform B-Spline the knots are spaced equally. In the nonuniform B-Spline the knots can be specified by the user and do not have to be equally spaced.

In this case only the uniform B-Spline is used with a degree of 2. It is called the *quadratic uniform* B-Spline. Assuming three seeds, which are the control points $\boldsymbol{p}_1$, $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$, to work with. These three seeds are consecutive seeds of the shape's outer contour. The generated curve segment will not touch any of these points as it is a B-Spline approximation (see Fig. 3.12). The start point $\boldsymbol{K}_1$ and end point $\boldsymbol{K}_2$ of the curve can be calculated as

$$\boldsymbol{K}_1 = \frac{1}{2} \cdot (\boldsymbol{p}_1 + \boldsymbol{p}_2), \qquad \boldsymbol{K}_2 = \frac{1}{2} \cdot (\boldsymbol{p}_2 + \boldsymbol{p}_3). \tag{3.15}$$

The quadratic B-Spline segment can be calculated as

$$\boldsymbol{P}_1(t) = 0.5 \cdot \begin{bmatrix} t^2 & t & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & -2 & 1 \\ -2 & 2 & 0 \\ 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \boldsymbol{p}_3 \end{bmatrix} \tag{3.16}$$

with $t \in [0, 1]$, which can again be written as the scalar equation

$$\begin{aligned} \boldsymbol{P}_1(t) = {} & 0.5 \cdot (t^2 - 2t + 1) \cdot \boldsymbol{p}_1 + \\ & 0.5 \cdot (-2t^2 + 2t + 1) \cdot \boldsymbol{p}_2 + \\ & 0.5 \cdot t^2 \cdot \boldsymbol{p}_3. \end{aligned} \tag{3.17}$$

3.6.3 Cubic Bézier Curves

The Bézier curve [8, Chap. 1], [10] in general is a parametric curve $\boldsymbol{P}(t)$ which is a polynomial function of the parameter t. Other than with the B-Spline the degree of the Bézier curve always depends on the number of control points. A quadratic Bézier curve therefore is controlled by three points, namely a starting point, a control point and an end point. Interpolation with this method would not be that different from the B-Spline interpolation approach and the curve could not be controlled as easily as it should. To achieve better control of the curve the cubic Bézier curve was implemented in the algorithm. The cubic Bézier curve is controlled by four points, whereat it is running through the most outer ones and attracted by the interior control points. This allows much more control over the curve, which is the reason why this type of curve is used for PostScript [20], as well as for a number of draw applications like Adobe Illustrator [19].

To achieve a similar result as in the previous approaches it is necessary to generate new interpolation as well as control points for the curve. The next paragraph explains how these points are calculated. To get a better understanding of the process see Fig. 3.13.

Assume a big shape made up of an outer contour $C = (\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_{n-1})$ of length n. In the final image, all in all, n single Bézier curves are generated which should smoothly connect to a spline. The goal is not to interpolate but approximate these n initial points. To do so, new interpolation points have to be declared as first and last point of each Bézier curve. These points $\boldsymbol{M}_i$ lie exactly in the middle of the path between a specific point $\boldsymbol{p}_i$ and its neighbored point $\boldsymbol{p}_{i-1}$ with $i \in [0, n-1]$. The belonging equation would look like

$$\boldsymbol{M}_i = |(\boldsymbol{p}_i - \boldsymbol{p}_{i-1}) \cdot 0.5 - \boldsymbol{p}_i|. \tag{3.18}$$

For the specific case of the first point $\boldsymbol{p}_0$ with $i = 0$ the neighbor point is not $\boldsymbol{p}_{i-1}$, as this point does not exist, but $\boldsymbol{p}_{n-1}$ as it is always a closed shape.

In the example shown in Fig. 3.13 there are four initial points $\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2$ and $\boldsymbol{p}_3$. The new interpolation points for this shape are calculated according to Eq. 3.18 as the points exactly in the middle between $\boldsymbol{p}_3$ and $\boldsymbol{p}_0$, $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$, $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ and $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$ and are called $\boldsymbol{M}_0, \boldsymbol{M}_1, \boldsymbol{M}_2$ and $\boldsymbol{M}_3$ respectively (see Fig. 3.13(b)).

The next step is to generate the two control points for each cubic Bézier curve. Therefore each initial point $\boldsymbol{p}_i$ and the two neighbored interpolation points $\boldsymbol{M}_i$ and $\boldsymbol{M}_{i+1}$ are necessary. Again for the specific case of $i = n-1$ the both neighbors of $\boldsymbol{p}_i$ are $\boldsymbol{M}_i$ and $\boldsymbol{M}_0$ due to the closed shape. The both control points of this set of three points are located at three quarters of the way between the interpolation points and the initial point and are referred

to as $\boldsymbol{c}_{i\cdot 2}$ and $\boldsymbol{c}_{i\cdot 2+1}$. The equations for this control points can be written as

$$\begin{aligned} \boldsymbol{c}_{i\cdot 2} &= |(\boldsymbol{M}_i - \boldsymbol{p}_i) \cdot 0.75 - \boldsymbol{M}_i|, \\ \boldsymbol{c}_{i\cdot 2+1} &= |(\boldsymbol{M}_{i+1} - \boldsymbol{p}_i) \cdot 0.75 - \boldsymbol{M}_{i+1}|, \end{aligned} \tag{3.19}$$

with $i \in [0, n-1]$. For the specific set in Fig. 3.13(b) of $\boldsymbol{M}_0$ and $\boldsymbol{M}_1$ as interpolation points and $\boldsymbol{p}_0$ as the belonging initial point the first control point generated is located at three quarters of the way between $\boldsymbol{M}_0$ and $\boldsymbol{p}_0$, whereas the second control point is located at three quarters of the way between $\boldsymbol{M}_1$ and $\boldsymbol{p}_0$. These control points are called $\boldsymbol{c}_0$ and $\boldsymbol{c}_1$ and the equations for calculating them are

$$\begin{aligned} \boldsymbol{c}_0 &= |(\boldsymbol{M}_0 - \boldsymbol{p}_0) \cdot 0.75 - \boldsymbol{M}_0|, \\ \boldsymbol{c}_1 &= |(\boldsymbol{M}_1 - \boldsymbol{p}_0) \cdot 0.75 - \boldsymbol{M}_1|. \end{aligned} \tag{3.20}$$

After all control points are generated (which can be seen in Fig. 3.13(c)) the actual interpolation takes place. Each Bézier curve is generated with two interpolation points $\boldsymbol{M}_i$ and $\boldsymbol{M}_{i+1}$ and the belonging control points $\boldsymbol{c}_{i\cdot 2}$ and $\boldsymbol{c}_{i\cdot 2+1}$ with $i \in [0, n-1]$. Again the last segment is a special case with the interpolation points of $\boldsymbol{M}_{n-1}$ and $\boldsymbol{M}_0$ and the control points of $\boldsymbol{c}_{(n-1)\cdot 2}$ and $\boldsymbol{c}_{(n-1)\cdot 2+1}$. The interpolation points for the Bézier curve in general can be calculated with the formula

$$\boldsymbol{P}_i(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{M}_i \\ \boldsymbol{c}_{i\cdot 2} \\ \boldsymbol{c}_{i\cdot 2+1} \\ \boldsymbol{M}_{i+1} \end{bmatrix}, \tag{3.21}$$

which would be

$$\boldsymbol{P}_0(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{M}_0 \\ \boldsymbol{c}_0 \\ \boldsymbol{c}_1 \\ \boldsymbol{M}_1 \end{bmatrix} \tag{3.22}$$

for the first Bézier curve interpolation. This formula can again be written as the scalar equation

$$\begin{aligned} \boldsymbol{P}_0(t) = &(1-t)^3 \cdot \boldsymbol{M}_0 + \\ &3t(1-t)^2 \cdot \boldsymbol{c}_0 + \\ &3t^2(1-t) \cdot \boldsymbol{c}_1 + \\ &t^3 \cdot \boldsymbol{M}_1, \end{aligned} \tag{3.23}$$

for which the assumption holds that the smaller the increasing steps of t are, the smoother the shape's contour in the resulting image will be. The final output of the interpolation for the chosen example shape is shown in Fig. 3.13(d).

3.6.4 Optimizing the Cubic Bézier Curve

In Sec. 3.7 the issue of holes occurring whenever more than two shapes share a seed (see Fig. 3.16(b) for a better understanding) is described. To eliminate this problem the approach of the cubic Bézier curve is optimized with a small change.

The initial situation with a shape of n initial points stays the same (as can be seen in Fig. 3.14(a)). The optimization happens in the second step of the approach when each point is examined whether being used by more than two shapes or not. If the seed is used by only two shapes nothing changes in the procedure. If the point is used by at least three shapes the initial point is interpreted as interpolation point as well. So such an initial point $\boldsymbol{p}_i$ is used to create an interpolation point using Eq. 3.18 and after that the point is included into the list of interpolation points as well. All interpolation points for the specific example of Fig. 3.14 are shown as blue circles in Fig. 3.14(b) assuming that the initial points $\boldsymbol{p}_0$ and $\boldsymbol{p}_2$ are used in multiple shapes.

The generation of the control points is the next step. Basically, this is not different to the approach without the optimization. Each initial point and the belonging neighbored interpolation points are used to calculate the control points' position with Eq. 3.19. If this specific initial point is an interpolation point as well there are two more control points generated at one third of the length between those points as well. These two points control the point that is an initial point, as well as an interpolation point. The result for the calculated control points can be seen in Fig. 3.14(c). The last step is again the same as with the non-optimized cubic Bézier curve as an interpolation between two interpolation points and the two interior control points is done according to Eq. 3.21.

3.7 Showing the Output

As in this implementation of the algorithm ImageJ is used, an overlay is created to display the resulting image in a first step. Because ImageJ itself can not render a filled spline, the generated splines have to be interpreted as polygons. For this approach a certain amount of the interpolation points are calculated and stored in a list as described in Eq. 3.21 for the cubic Bézier curve or in Eq. 3.11 for the Catmull-Rom interpolation. Afterwards a polygon is created in ImageJ via drawing small lines from one stored interpolation point to the next. The polygon then can be filled with a specific color and printed on the overlay. Each spline type interpolates a little bit different because of which all results look similar but not the same.

In Figure 3.15(a), for example, the resulting image with the Catmull-Rom interpolation can be seen. As this method passes through all the control points there are no holes visible in this image but also due to this fact the polygons, that are generated to be displayed by ImageJ, are overlapping.

This is clearly visible in Fig. 3.15(b) with only the outline of the polygons being visible. Furthermore, this means that at some points the depth information (of which shape is lying over another) can be deluded by the render order of the single polygons. The earlier a polygon is rendered, the more it lies on the bottom of the image and the other way round. The advantage is that the roundness and smoothness of the image looks fine, however not perfect.

Figure 3.16, amongst others, shows the interpolation results of the Quadratic B-Splines. In Fig. 3.16(a) it is clearly visible which disadvantage the usage of pure B-Splines has on the output image if three or more splines contain the same seed. With the seed not being passed through, but only approximated there are holes visible in the image wherever a seed is used by three or more shapes. In Fig. 3.16(b) the resulting image for the cubic Bézier curves is shown. This method produces holes at the same position of the multiple used seeds but compared to the method shown in the first image (Fig. 3.16(a)) these holes are much smaller because the two control points used make it possible to approximate even more than the one control point used with the quadratic B-Spline. To fully eliminate the spacing the optimized cubic Bézier spline, as described in Sec. 3.6.4, is used to include seeds, which are used multiple times into the list of interpolation points. The interpolation afterwards is exactly the same procedure as described in Eq. 3.21. In this way all gaps are closed and no more holes remain in the resulting image (see Fig. 3.16(c)). One minor disadvantage of this approach is that the smoothness suffers a little bit at exactly these multiple used interpolation points and the image does not appear as “round” as the resulting image of the Catmull-Rom interpolation.

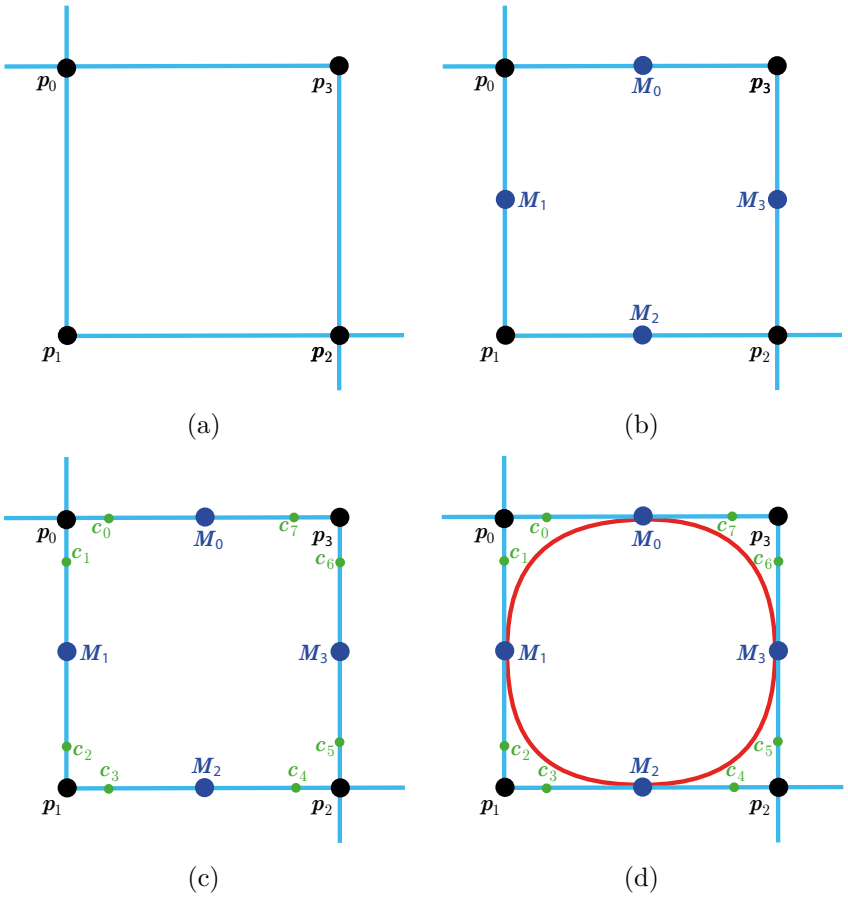


Figure 3.13: These images show the whole process of generating a Bézier curve according to a given shape made up of the four initial points $\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2$ and $\boldsymbol{p}_3$. In the first step all interpolation points for the shape pictured in (a) are calculated with Eq. 3.18 for all initial points of the shape. The resulting midpoints are shown as blue circles in (b). For the second step the control points for a set of one initial point and two belonging interpolation points are generated with Eq. 3.19 for each initial point. These control points are displayed as smaller green circles in (c). For the last step two interpolation points and the interior two control points are used to interpolate the cubic Bézier curve according to Eq. 3.21. All in all there are as many Bézier curves generated as interpolation points exist. All these Bézier curves are smoothly connecting because of the interpolation points' location. The resulting curve can be seen in (d) as red curve.

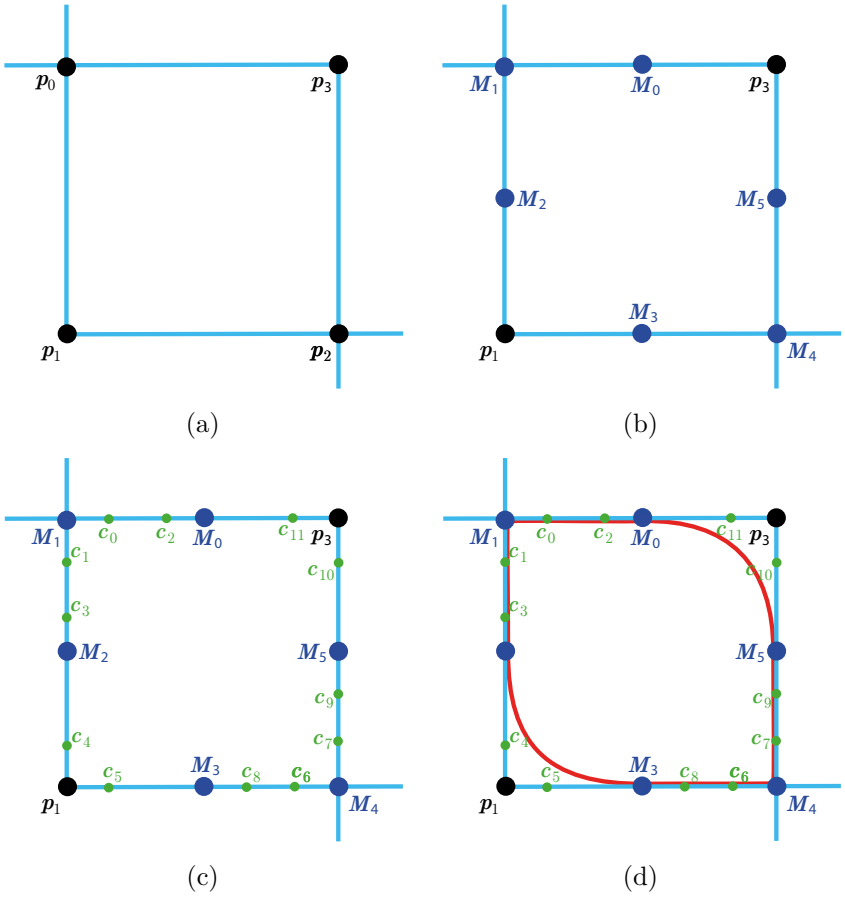
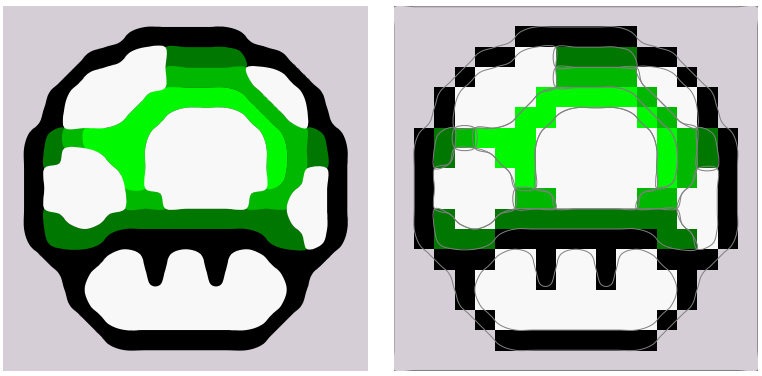
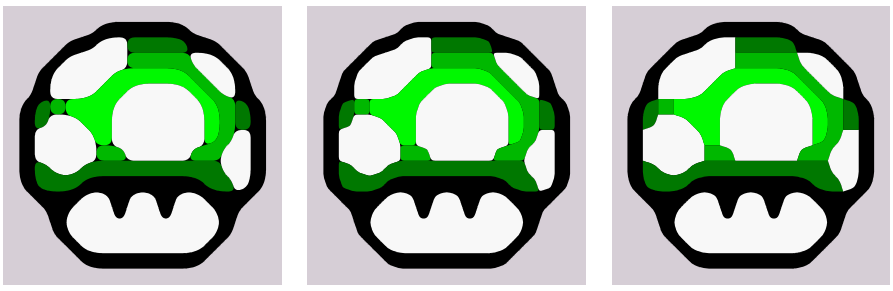


Figure 3.14: These images show the whole process of generating a Bézier curve with a given shape made up of the four initial points $\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2$ and $\boldsymbol{p}_3$ pictured in (a). In the first step all interpolation points for the shape are calculated with Eq. 3.18 for all initial points of the shape. Included in this step now is an examination of every initial point. If an initial point is part of more than two shapes this point is interpreted as interpolation point as well. The resulting interpolation points are shown as blue circles in (b). For the second step the control points for a set of one initial point and two belonging interpolation points are generated with Eq. 3.19 for each initial point. If the initial point is also interpreted as interpolation point this process is repeated with also generating two control points at one quarter of the way. The resulting control points are displayed as smaller green circles in (c). For the last step two interpolation points and the interior two control points are used to interpolate the cubic Bézier curve according to Eq. 3.21. All in all there are as many Bézier curves generated as interpolation points exist. All these Bézier curves are smoothly connecting because of the interpolation points' location. The resulting curve can be seen in (d) as red curve.



(a) (b)

Figure 3.15: The result of the Catmull-Rom interpolation. The final output with the Catmull-Rom spline interpolation is visible in (a). The image looks nicely round and smooth. In (b) only the outlines of different polygons rendered with ImageJ are shown. The disadvantage of these polygons overlapping is clearly visible.



(a) (b) (c)

Figure 3.16: Resulting images for different interpolation techniques. Picture (a) shows the disadvantage of B-Splines generating holes where more than two splines meet. Image (b) pictures the same points as in (a) being interpolated with a cubic Bézier curve. Also in this result holes appear, wherever three or more shapes share one seed, even if these gaps are smaller than the ones produced by the B-Spline approach. To overcome this problem a seed that is used by more than two shapes is handled as interpolation point in (c), whereby all splines are forced to touch the seeds position. According to this the holes are closed but the smoothness of the resulting image suffers a little bit.

Chapter 4

Extensions

Chapter 3 ended with a proper, however not optimal, solution for the rendering. This section describes some extensions that were implemented additionally to the basic approach of Kopf and Lischinski. Because there is no information about the exact rendering process used by Kopf and Lischinski given in the original paper, another way of extracting a PDF file is used in this implementation. Among this, also a method for introducing a color gradient to the resulting image is explained.

4.1 Generating a PDF File Using Bézier Curves

As a matter of fact, the method described before uses a polygon approximation of the different spline types, which is not an optimal solution because no real Bézier splines are generated but only approximated by interpolation. Therefore another way of extracting these splines has to be found. The problem is solved with the use of *iText*.[1] It is used in particular whenever content is generated dynamically including personalized or customized content.

When using iText [7] for generating a PDF-file there are basically five steps to follow, which are

- generating a document with dimensions as big as the final output,
- generating a `PdfWriter` instance,
- opening the document,
- adding content to it, and finally
- closing the document.

For creating, a new document is instantiated as big in size as the resulting image should be. To do so, a rectangle with a width and a height according to the resulting dimensions is created via using the parameters like this:

```
Document document = new Document(new Rectangle(width, height));
```

[1] https://sourceforge.net/projects/itext/

The width and the height used are instantiated as the width and height of the original input image multiplied with a magnification factor that can be determined by the user. Next a `PdfWriter` instance has to be created to determine the PDF format of the document. The `PfdWriter` is implemented as a specific implementation of the abstract `DocWriter` class, which can also be implemented as `HtmlWriter` or `RtfWriter`. Creating a `PdfWriter` like

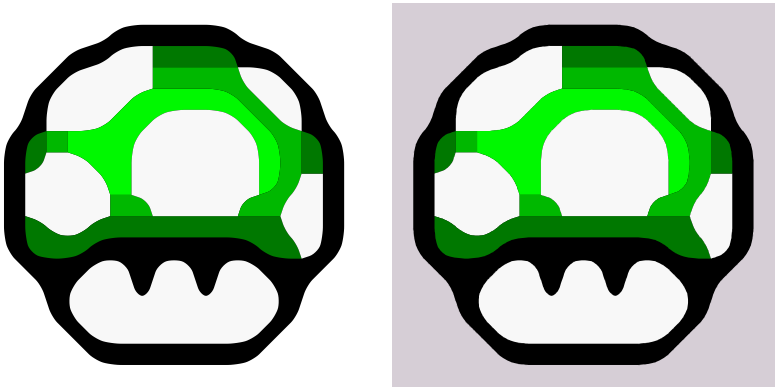
```
PdfWriter writer =
PdfWriter.getInstance(document, new FileOutputStream(path));
```

creates a `PdfWriter` instance with the `document` object as parameter as well as a `FileOutputStream` as parameter, which specifies an absolute path as location where the document is saved after being generated. After creating the writer, the documents is opened via the method `document.open()` whereupon some background initializations take place. After opening the document, one is able to fill content into it by placing, for example, quadratic or cubic Bézier curves. This is done by using the `writer.getdirectContent()` method, which receives the so called `PdfContentByte`. Furthermore content can be applied to this `PdfContentByte` object, which is further referred to as "cb". To generate a Bézier curve in the content of the PDF the `curveTo()` method is used. This method can be used with either four or six parameters. With four parameters used the curve is described as quadratic Bézier curve using the current x-, y-coordinates as starting point, the first and second parameters as x-, y-coordinates of the single control point and the third and fourth parameters as the x-, y-coordinates of the endpoint. The same procedure is done with six parameters with the only difference of the curve being described as a cubic Bézier curve, now with the x-, y-coordinates of the control points being defined by the first and second and the third and fourth parameter respectively. The last two parameters are the coordinates of the end point. A cubic Bézier is therefore generated as

```
cb.curveTo(p1.getX(),p1.getY(),p2.getX(),p2.getY(),p3.getX(),p3.getY());
```

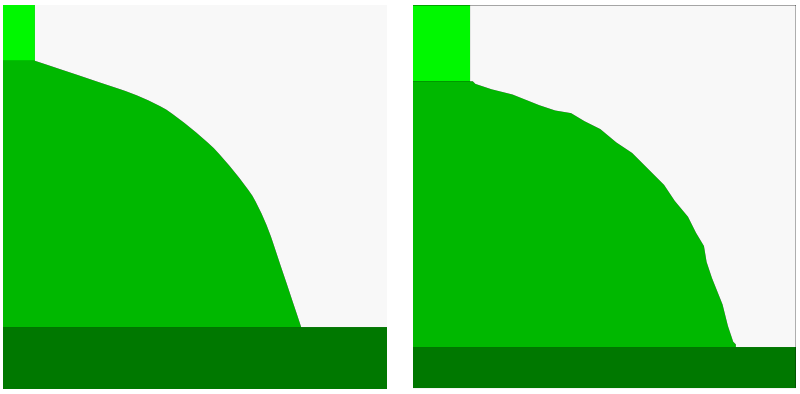
with `p1` being the first and `p2` being the second control point, `p3` is defined as the end point of the Bézier curve.

In the implementation only cubic Bézier curves are generated because these came out the best. The coordinates for the control and intersection points have already been calculated for the previous step (as described in Secs. 3.6.3 and 3.6.4) so the only difficulty was to include the magnification scale. This was done by multiplying the original coordinates by the magnification factor chosen by the user in the beginning. The end result of the curves being saved as PDF-file can be seen in Fig. 4.1(a) compared to the one being done with the polygon approach shown in Fig. 4.1(b). By looking at a specific part of the image with a clear curve visible it is to see that the polygon approach produces tiny straight lines as outer contour (see Fig. 4.2(b), which is not visible when using larger images. With the iText approach however, clear Bézier curves with no straight sections in between



(a) (b)

Figure 4.1: Example images of the polygon approximation methode and the export with iText. In (a) the resulting image with the PDF export using the iText approach is visible whereas in (b) the old version with the polygon approximation using plain ImageJ overlays can be seen.

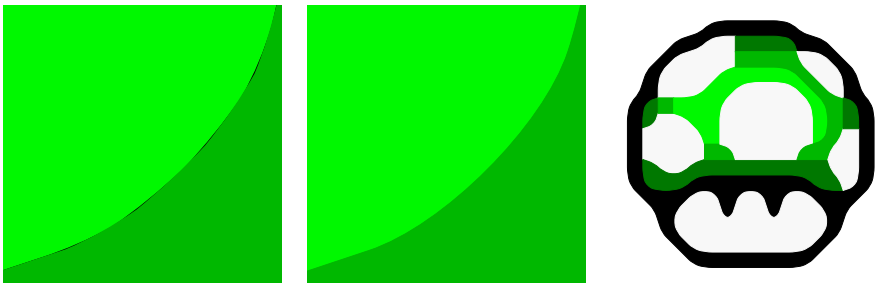


(a) (b)

Figure 4.2: The differences of the exported Bézier curves and the polygon approximation in detail. In (a) the contour between the shapes is a smooth line caused by the PDF export of the Bézier curves, in (b) on the other hand there are some unwanted artifacts and the lines do not look that smooth because of the piecewise line approximation with the polygon approach.

are generated, which leads to a cleaner and smoother output image shown in Fig. 4.2(a).

One minor issue that came up when comparing the resulting images was that in some cases the exported PDF images were presented with little holes between neighbored shapes as can be seen in Fig. 4.3(a). This happened because in some cases the interpolation points of the two curves in this section did not consist of exactly the same coordinates. To eliminate these



(a) (b) (c)

Figure 4.3: Closeup of a contour between two shapes showing small gaps. In (a) one can see an enlarged extraction of the exported image 4.1(a) before the stroke was added. The holes at the contour where the two shapes meet are clearly to see in this magnified image. In (b) the same extraction is shown with the stroke added and the coordinates being resolved. All the gaps are gone and the contour is smooth and not perceptibly larger. The figure in (c) shows the complete image after resolving the gaps. Compared to the image in Fig. 4.1(a) no difference can be seen in this size but with a higher resolution the differences could be seen as pictured in (a) and (b).

small gaps two different approaches were implemented. On the one hand the interpolation points were checked a second time to be exactly the same coordinates when sharing the same position. On the other hand a small stroke was drawn at the outer contour of the shape being as small as possible so that the gaps were closed but the outcome was not manipulated too much. This stroke has a width w depending on the magnification scale M to be sure that this approach works in all different sizes. It was defined as $w = 0.01 \cdot M/2$ due to some testing and finding this as best result. The resulting figure is shown in Fig. 4.3(c) after resolving the gaps. Compared to the image in Fig. 4.1(a) no difference can be seen at this size but with a higher resolution the differences could be seen in exactly the same way as shown in Figs. 4.3(a) and 4.3(b).

4.2 Introducing the Color Gradient

As the image is now defined by clear lines and constructed by using cubic Bézier curves the improvement compared to the original pixel art image from the beginning can not be overlooked. To take it another step further in parts of generating a smooth output image, the idea was to introduce a color gradient wherever neighbored colors in the image are similar enough to blend. How this was implemented is described in this section.

4.2.1 Color Gradients with iText

For the approach of the color gradients iText provides a method by using a so called *shading pattern* as fill or stroke color of a shape. Such a pattern can be initialized in two ways, either as radial gradient or as axial gradient. The radial gradient is defined by two circles, between which the gradient varies from a start color to an end color. These circles are defined by a center point as well as a radius. To initialize such a radial gradient one has to use the static method `PdfShading.simpleRadial` like

```
1 PdfShading radial = PdfShading.simpleRadial(writer, x0, y0, r0, x1, y1,
      r1, colorStart, colorEnd, true, true);
2 PdfShadingPattern radialGradient = new PdfShadingPattern(radial);
3 cb.setShadingFill(radialGradient);
```

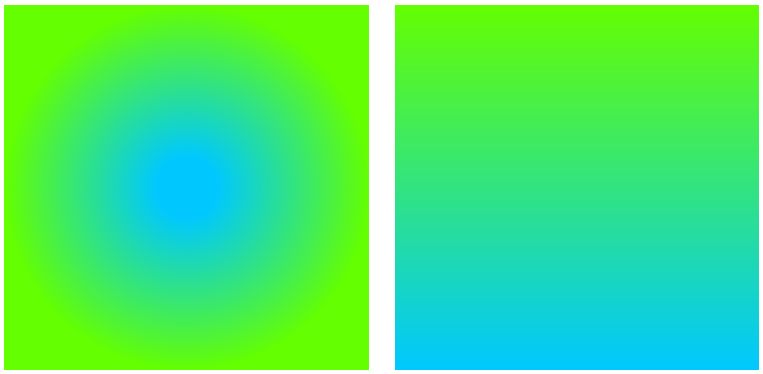
which defines the first circle of the gradient with a center point $\boldsymbol{p}_0 = (x_0, y_0)$ and a radius r_0 and the second circle with a center point $\boldsymbol{p}_1 = (x_1, y_1)$ and a radius r_1. The parameters `colorStart`, `colorEnd` can be defined as RGB colors and the two boolean values in the very end define if the starting/ending colors are extended in excess of the start and/or the end. To use this pattern in place of the fill color for example, the `PdfShading` object is used to create a `PdfShadingPattern` object as can be seen in line two of the code snippet. Finally this `radialGradient` of type `PdfShadingPattern` is set as fill pattern in the third line.

The axial gradient works very similar. All steps are the same as with the radial gradient, except the initialization of the `PdfShading` object. The axial gradient is not defined as two circles but as two points, a start point and an end point, between which the gradient happens. In case of this type of gradient the `PdfShading` object is defined as

```
PdfShading axial = PdfShading.simpleAxial(writer, x0, y0, x1, y1,
    colorStart, colorEnd, true, true);
```

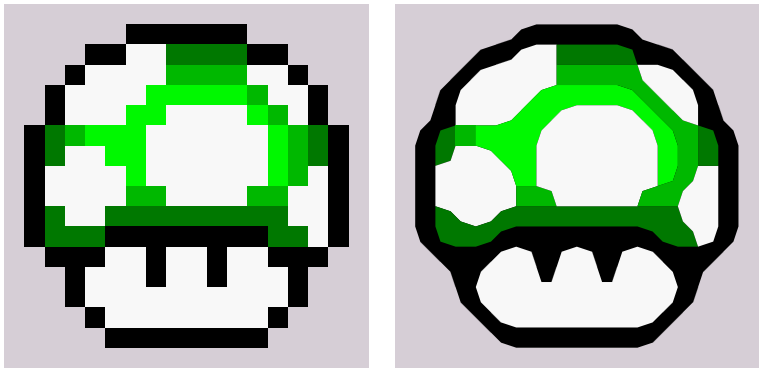
where $\boldsymbol{p}_0 = (x_0, y_0)$ is the starting point and $\boldsymbol{p}_1 = (x_1, y_1)$ is the end point of the gradient. The parameters `colorStart`, `colorEnd` again define the two colors at the beginning and the end of the gradient and the two boolean variables in the end of the code snippet define whether the colors are extended in excess of the start and/or the end positions. The `PdfShading` object can be set as `PdfShadingPattern` as before with the radial gradient.

Figure 4.4 shows two example images, one picturing a radial gradient (see Fig. 4.4(a)) and the other showing an axial gradient (see Fig. 4.4(b)), which were implemented with this basic methods of ImageJ. Parts of the plugin are incorporated in the appendix (see Sec. A.2). The whole plugin is included into the final implementation of the master's project.



(a) (b)

Figure 4.4: The resulting images of a radial gradient (a) and an axial gradient (b) generated with iText. The area is a 18×18 canvas and the coordinates for the center points/radii of the two circles for the radial gradient as well as the two points for the axial gradient are adjusted to this.



(a) (b)

Figure 4.5: The original input image and the result after generating the big shapes. Image (a) shows the original input image that was used to create the image in (b), which shows the subdivision of the input image in larger shapes after combining same colored neighbor shapes of the Voronoi diagram.

4.2.2 Gradients in the Image

To get the gradients included in the output image of the project each shape and its neighbors have to be looked at individually. To begin this process all different colors of the image have to be considered. So for a better understanding take the mushroom in Fig. 4.5(a) as input image again. After the first steps described in Chap. 3 the result looks like Fig. 4.5(b) and consists of a series of big shapes. These shapes are defined by an outer contour of consecutive seeds and a color, basically.

The first step for the gradients is to extract all different colors of the image and save all of them in a list. To get an even better result, the background shape was ignored for this and the background color was considered non-existent if it would only appear in one shape of the image.

If the image contains only four or less colors there is no gradient done at all because the image contains not enough color information and the information loss of the image using a gradient in such a situation would be too big. Also, the color distances are to big in this situations in most cases.

After inserting all different colors in a list this list is sorted from brightest to darkest color. To do so, the color difference c_{diff} from all pairs of colors in the list is calculated and sorted in ascending order. The distance calculation is done according to the equation

$$c_{\text{diff}} = \frac{1}{3} \cdot ((C_{0,R} - C_{1,R}) + (C_{0,G} - C_{1,G}) + (C_{0,B} - C_{1,B})), \tag{4.1}$$

where two colors $\boldsymbol{C}_0 = (C_{0,R}, C_{0,G}, C_{0,B})$ and $\boldsymbol{C}_1 = (C_{1,R}, C_{1,G}, C_{1,B})$ are compared according to the medium differences in the R, G, and B value. If c_{diff} is a negative number $\boldsymbol{C}_1$ is consider brighter than $\boldsymbol{C}_0$ but if c_{diff} is a positive number it is considered the other way round. If the distance is exactly zero the colors are considered the same brightness and listed one after another.

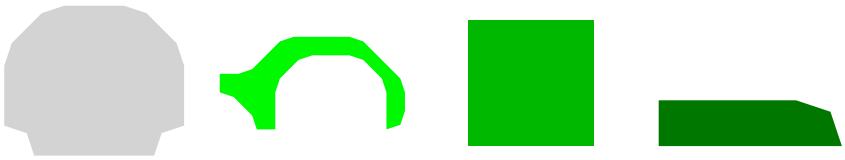
After sorting the list of colors the first two colors in the list are taken and all shapes in the image with the lighter color are identified and stored in a list $L = (l_0, l_1, l_2, \ldots, l_{n-1}, l_n)$. Iterating over all of these elements in the list L for each Shape $l_x, x \in [0, n]$ all neighbored shapes[2] with the second color, the darker one, are identified and as well stored in a list $D = (d_0, d_1, d_2, \ldots, d_{n-1}, d_n)$.

The situation to work with in one iteration is now the following: There is a single shape of the lighter color l_x and a list of neighbored shapes of the darker color D. To make sure that the right gradient is used for smoothing this neighbored shapes (this could either be a radial or an axial gradient) first of all the circularity of the lighter shape l_x has to be calculated as

$$\text{Circularity}(S) = 4 \cdot \pi \cdot \frac{A(S)}{P(S)^2}, \tag{4.2}$$

with $\text{Circularity}(S) \in [0, 1]$ in general, where S is any arbitrary shape that is defined by a closed outer contour consisting of consecutive seeds. The nominator $A(S)$ is defined as the area and the denominator $P(S)$ is defined as the circumference of the shape. The outer contour of S is a list (of length m) of several seeds $(\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_{m-1})$ where each seed $\boldsymbol{p}_n, n \in [0, m-1]$ is defined by a x- and a y-value like $\boldsymbol{p}_n = (x_n, y_n)$ and $\boldsymbol{p}_0$ and $\boldsymbol{p}_n$ are

[2] Again neighbored shapes are shapes that share at least one edge.



(a) (b) (c) (d)

Figure 4.6: Axial and radial shapes according to their circularity. The figure in (a) is of a pretty round shape as one can see. The circularity of this shape is 0.8906, for what this shape is considered radial. The circularity of figure(b) is 0.2159, which actually indicates a non round shape. But the smaller the circularity gets, the more a shape is forming a roundish shape with a hole in it. Also the shape in (b) looks a bit like half a donut shape and is assumed to be round because of this. The shapes in (c) and (d) have a circularity of 0.7853 and 0.5133 and are therefore considered axial.

consecutive seeds. Because of this the circumference $P(S)$ of a shape S can be calculated as

$$P(S) = \sum_{i=0}^{n} \sqrt{(\boldsymbol{p}_{i+1,x} - \boldsymbol{p}_{i,x})^2 + (\boldsymbol{p}_{i+1,y} - \boldsymbol{p}_{i,y})^2}, \tag{4.3}$$

which is the sum of the length of all pairs of consecutive seeds in the outer contour. Furthermore, the area $A(S)$ is defined as

$$A(S) = \frac{1}{2} \cdot \sum_{i=0}^{n} (\boldsymbol{p}_{i,x} - \boldsymbol{p}_{i+1,x}) \cdot (\boldsymbol{p}_{i,y} + \boldsymbol{p}_{i+1,y}). \tag{4.4}$$

The results for the calculation of the circularity of the shape l_x is a number in range [0, 1]. To be able to continue with the gradients in an appropriate way a shape S is either considered *axial* if $0.245 < \text{Circularity}(S) < 0.800$ or *radial* if $0.245 \geq \text{Circularity}(S) \geq 0.800$. These threshold values were found to work best after testing. In Fig. 4.6 examples of radial and axial shapes are pictured along with their circularity.

Axial Shapes

So if a shape is considered *axial* according to its circularity the situation to work with is the following: There is one axial shape of the lighter color l_x and a list of darker neighbored shapes D. There are some different situations that have to be considered when introducing an axial gradient as well. First of all, it has to be checked if the shape of the lighter color already has a gradient introduced, which can happen in later iterations.

The initial shape l_x holds no gradient: If there is no gradient in l_x yet the amount of neighbored shapes has to be figured out. If D only contains one shape the axial gradient is introduced using Eqs. 4.7–4.9. If there is more than one neighbor one has to go further.

It has to be checked if the lighter shape is *surrounded* by the neighbor shapes. This is done by first saving all outer contour nodes of the darker shapes in a list $C_d = (\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_{m-1})$ with a length of m and then finding both, the smallest and the greatest x-value as well as the smallest and the greatest y-value in this list according to

$$\begin{aligned} x_{\min} &= \min \boldsymbol{p}_{i,x}, \\ x_{\max} &= \max \boldsymbol{p}_{i,x}, \\ y_{\min} &= \min \boldsymbol{p}_{i,y}, \\ y_{\max} &= \max \boldsymbol{p}_{i,y} \end{aligned} \tag{4.5}$$

where $i \in [0, m-1]$. After this the minimum/maximum x-values and the minimum/maximum y-values of the outer contour nodes of l_x, witch are named $x_{sMin}, x_{sMax}, y_{sMin}, y_{sMax}$ respectively, are saved as well using the same method as for the darker nodes. Now if the following condition holds the lighter colored shape is assumed to be surrounded by the darker neighbored shapes:

$$(x_{min} < x_{sMin}) \wedge (x_{max} > x_{sMax}) \wedge (y_{min} < y_{sMin}) \wedge (y_{max} > y_{sMax}). \tag{4.6}$$

If after this step the lighter shape is assumed to be surrounded by all the neighbored shapes not an axial but a radial gradient is introduced using Eqs. 4.10 and 4.11.

But if l_x is considered not surrounded only one neighbor shape is needed for introducing an axial gradient. This specific shape d is the one that shares the longest common contour with the lighter shape of all neighbored shapes. The length of a contour is just assumed as the number of common nodes. Finally, after finding d an axial gradient is introduced using Eqs. 4.7–4.9 and assuming l_x as lighter and d as darker shape.

The initial shape l_x holds a gradient: If there is already a gradient introduced to l_x the first thing is to save the lighter color $\boldsymbol{C}_l$ of the already existing gradient, which is always the starting color no matter what type of gradient. After that, the issue of a surrounded lighter shape l_x has to be solved as described in the paragraph before using Eqs. 4.5 and 4.6, after what the shape is either assumed *surrounded* or *not surrounded*.

If the latter assumption holds, a search for the shape with the longest common contour is performed again and the axial gradient is generated using Eqs. 4.7–4.9 but with the only difference that the lighter color for the new gradient is the previously saved lighter color $\boldsymbol{C}_l$ of the already existing

gradient. In this case, it does not matter what kind of gradient already exists in the lighter shape.

If the lighter shape is surrounded by the neighbors, however, it does matter. So if the gradient in l_x is an axial one nothing is done, because the old axial gradient is considered more important in an axial shape as an eventual new radial gradient. But if the existing gradient is a radial gradient this type of gradient is extended by creating a new radial gradient according to Eqs. 4.10 and 4.11 but with the one difference that the previously saved color $\boldsymbol{C}_l$ is used as starting color this time.

Radial Shapes

With radial shapes the whole situation is a little bit easier. If l_x is a radial shape it is also checked if there already is a gradient and if so, the lighter color is saved as $\boldsymbol{C}_l$.

Furthermore, if l_x is surrounded by its neighbor shapes or also if at least three of the assumptions for the surrounding (see Eq. 4.6) are true then a radial gradient is introduced again using Eqs. 4.10 and 4.11 (possibly using the previously saved $\boldsymbol{C}_l$ as lighter color if there was a gradient in l_x before). With this nearly all of the situations for a radial shape can be solved because a radial shape is surrounded by darker neighbors in most cases. If this is not the case and the light shape is not surrounded it is checked if the list of all neighbors D only contains one element. If this is the case this single shape's circularity is calculated and according to the result the last decision is made.

If the only neighbor shape is an *axial* shape also an axial gradient is created according to Eq. 4.7–4.9. If the neighbor shape is a radial shape however the gradient to be introduced is also a radial gradient, which is implemented using Eqs. 4.10 and 4.11.

Generating an Axial Gradient

Creating an axial gradient with iText, as explained in Sec. 4.2.1, only needs two points (a start point and an end point) of the gradient and two belonging colors. Whenever an axial gradient is generated between two shapes in the image the gradient is generated in the lighter shape only and the colors of the gradient are the lighter color as starting and the darker color as ending color. The process of calculating the belonging start and end point is started of by having a lighter colored shape l and a darker shaded shape d, which are sharing a common edge like Fig. 4.7(a) shows. The shape l is colored with a light gray color to make imagination more easy. The blue dots are indicating the common outer contour seeds of the both shapes. For the calculation of the starting point of the gradient this common contour is saved in a list of consecutive seeds $K = (\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_n)$. To make the gradient look even more precisely, the node before $\boldsymbol{p}_1$, which is $\boldsymbol{p}_0$, and the

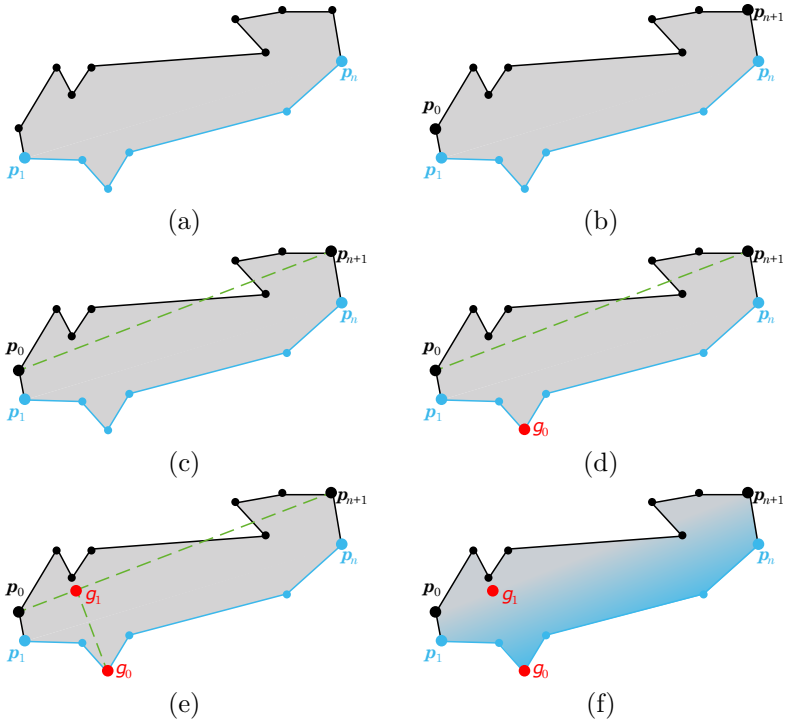


Figure 4.7: Generating an axial gradient. Image (a) indicated a shape l, which is the shape with the lighter color, and the common contour of the two shapes to smooth, which is shown as blue contour. The seed $\boldsymbol{p}_1$ is the first seed and $\boldsymbol{p}_n$ is the last seed of the common contour. Image (b) shows adding the seed $\boldsymbol{p}_0$ before $\boldsymbol{p}_1$ and $\boldsymbol{p}_{n+1}$ after $\boldsymbol{p}_n$ to get a better position for the starting point of the gradient. In (c) the relation of $\boldsymbol{p_0}$ and $\boldsymbol{p}_{n+1}$ is indicated with a imaginary line passing through both seeds. This is important for the next step of calculating the end point of the gradient, which is the farthest seed of the common contour to this line. This end point finally is shown in (d) as red colored seed $\boldsymbol{g}_0$. In (e) now the starting point $\boldsymbol{g}_1$ of the gradient is also printed, which is calculated as the intersection of the original line passing through $\boldsymbol{p}_0$ and $\boldsymbol{p}_{n+1}$ and the perpendicular line passing through $\boldsymbol{g}_0$. The final gradient is then shown in (f).

node after $\boldsymbol{p}_n$, which is $\boldsymbol{p}_{n+1}$ in the closed outer contour of s_l are added in the first and the last position of the list $K = (\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_n, \boldsymbol{p}_{n+1})$, as Fig. 4.7(b) illustrates. The next step is to imagine a line passing through $\boldsymbol{p}_0$ and $\boldsymbol{p}_{n+1}$ (see Fig. 4.7(c)). There is no actual line equation necessary for the next steps but the implementation of the algorithm assumes the two points lying on a straight line. First the endpoint $\boldsymbol{g}_0$ of the gradient is calculated by finding the seed $\boldsymbol{p}_x$ in the common contour, which is the farthest from the imagined line between $\boldsymbol{p}_0$ and $\boldsymbol{p}_{n+1}$. For this the distance $d(p_0, p_{n+1}, p)$ between the line and every seed $\boldsymbol{p} = (x, y)$ (except $\boldsymbol{p}_0$ and $\boldsymbol{p}_{n+1}$ because

their distance would be zero of course) in K is calculated according to the general equation

$$d(\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}) = \frac{|(\boldsymbol{p}_{1,y} - \boldsymbol{p}_{0,y}) \cdot \boldsymbol{p}_x - (\boldsymbol{p}_{1,x} - \boldsymbol{p}_{0,x}) \cdot \boldsymbol{p}_y + \boldsymbol{p}_{1,x} \cdot \boldsymbol{p}_{0,y} - \boldsymbol{p}_{1,y} \cdot \boldsymbol{p}_{0,x}|}{\sqrt{(\boldsymbol{p}_{1,y} - \boldsymbol{p}_{0,y})^2 + (\boldsymbol{p}_{1,x} - \boldsymbol{p}_{0,x})^2}}. \tag{4.7}$$

The seed with the greatest distance to the line is finally taken as the end point of the gradient. An example can be seen in Fig. 4.7(d) showing the end point $\boldsymbol{g}_0$ of the gradient as red colored seed. The final step for the axial gradient is calculating the end point of the gradient. This is actually really easy because this point $\boldsymbol{g}_1$ is assumed to be the intersection point of the line through $\boldsymbol{p}_0$ and $\boldsymbol{p}_{n+1}$ and the perpendicular line passing through g_0 (see Fig. 4.7(e) for a better understanding). The point $\boldsymbol{g}_1$ is defined as

$$\boldsymbol{g}_1 = (\boldsymbol{g}_{0,x} - k \cdot (\boldsymbol{p}_{n+1,y} - \boldsymbol{p}_{0,y}), \boldsymbol{g}_{0,y} + k \cdot (\boldsymbol{p}_{n+1,x} - \boldsymbol{p}_{0,x})) \tag{4.8}$$

with k being calculated as

$$k = \frac{(\boldsymbol{p}_{n+1,y} - \boldsymbol{p}_{0,y}) \cdot (\boldsymbol{g}_{0,x} - \boldsymbol{p}_{0,x}) - (\boldsymbol{p}_{n+1,x} - \boldsymbol{p}_{0,x}) \cdot (\boldsymbol{g}_{0,y} - \boldsymbol{p}_{0,y})}{(\boldsymbol{p}_{n+1,y} - \boldsymbol{p}_{0,y})^2 + (\boldsymbol{p}_{n+1,x} - \boldsymbol{p}_{0,x})^2}. \tag{4.9}$$

With this start point calculated, iText creates an axial gradient starting at $\boldsymbol{g}_1$ with the lighter color and ending at $\boldsymbol{g}_0$ with the darker color. The axial gradient for the example shape is shown in Fig. 4.7(f).

Now it is even more understandable why the seed before the first and the seed after the last common contour seeds are chosen to build up the line. If the first and last common contour seeds $\boldsymbol{p}_1, \boldsymbol{p}_n$ would be taken for calculating the endpoint, the starting point would be calculated nearer to the endpoint and the space for the gradient to develop would be very small in the most cases. Using the seeds before and after the first and the last contour seeds respectively enlarges this space a little bit and makes the gradient look more natural.

Generating a Radial Gradient

Creating a radial gradient in done a little bit different than creating an axial one. The axial gradient is generated in only one of two shapes. The radial gradient, however, considers more than two shapes in the most cases. One of them is the shape with the lighter color and the others (which can be several) are of the darker color. In Fig. 4.8(a) a situation is shown were a shape of the lighter color (shown as light gray shape) and several neighbored shapes (with blue colored outline seeds) are visible. To create a radial gradient with iText as described in Sec. 4.2.1 two circles need to be initialized. These circles are described by two center points, two radii and two colors. The starting color (which is the inner one) is assumed to be the lighter color and the end color

is assumed to be the darker color. The two center points are assumed to be at the same position, which is defined as the position at the average x- and y-coordinates of the outer contour O of the lighter shape. The outer contour again is assumed to be a closed contour $O = (\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_{n-1})$ with a length of n consecutive seeds. Also the last seed $\boldsymbol{p}_{n-1}$ is adjacent to the first seed $\boldsymbol{p}_0$. The center points of the circles are described as $c = (\overline{x}, \overline{y})$ with $\overline{x}$ and $\overline{y}$ being the average x- and y-coordinate of the outer contour that can be calculated as

$$\overline{x} = \frac{1}{n}\sum_{i=0}^{n-1}(\boldsymbol{p}_{i,x}), \qquad \overline{y} = \frac{1}{n}\sum_{i=0}^{n-1}(\boldsymbol{p}_{i,y}). \tag{4.10}$$

The next step, after defining the start and end color and calculating the center point, is to define the radius for the inner, smaller circle. For this the distances d between all nodes in O and the center point c are calculated as

$$d = \sqrt{(\overline{x} - \boldsymbol{p}_{i,x})^2 - (\overline{y} - \boldsymbol{p}_{i,y})^2}, \tag{4.11}$$

with $i \in [0, n-1]$. The smallest as well as the greatest distance are saved as d_0 and d_1 respectively. This step is illustrated in Fig. 4.8(b) with the green dotted lines indicating the smallest distance d_0 and the greatest distance d_1 of an outer contour seed and the center point. According to these two distances the radius of the smaller circle is calculated as $r_0 = (d_0 + d_1)/2$, which is just the medium distance. Figure 4.8(c) shows this step for a better understanding.

Next up is calculating the greater radius, the one for the outer circle. This is done by first storing all outer contour seeds of the darker neighbored shapes, that are not a common node with the lighter shape, in a list $N = (\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_{n-1})$ with n again being the length of this list. These seeds are basically all seeds that are marked with a blue color in Fig. 4.8(d). To get the radius for the greater circle the same steps as for the smaller circle are done. First calculating all distances of these seeds to the center point according to Eq. 4.11, then storing the greatest and smallest one as d_0, d_1 (see Fig. 4.8(d)) and finally calculating the radius as medium distance like $r_1 = (d_0 + d_1)/2$ (see Fig. 4.9(a)). The resulting two circles are shown in Fig. 4.9(b) with both having their center point in c. The smaller circle is indicated via a green colored, dotted outline with radius r_0, the larger one via a blue dotted outline with radius r_1.

The last step for defining the radial gradient is to combine the lighter shape and all neighbored darker shapes into one shape for the gradient to be placed onto. The approach for this is basically the same as described in Sec. 3.5. The common contour seeds of the lighter and darker shapes are deleted and a new outer contour is defined as closed contour of consecutive seeds including all non common seeds of the shapes. The newly created shape can be seen in Fig. 4.9(c) with the black nodes as new outer contour.

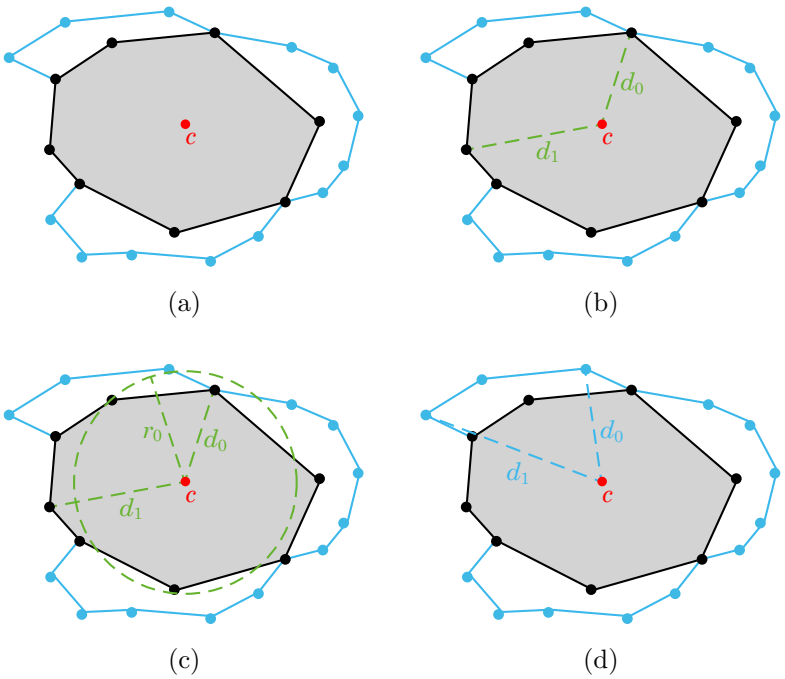


Figure 4.8: The first part of a radial gradient creation. In (a) the single shape with the lighter color (colored in light gray) and the neighbored shapes with the darker color (indicated via the blue outer contour nodes) can be seen. Also the center point for the gradient circles $c = (\overline{x}, \overline{y})$ is illustrated. In (b) the smallest and greatest distances between c and all outer contour seeds of the lighter shape are calculated and stored as d_0 and d_1. The radius r_0 for the smaller circle is calculated as the medium distance of d_0 and d_1 as shown in (c). Figure (d) shows that the steps that were done for the smaller circle are also executed for the larger circle by calculating the greatest and smallest distances d_0, d_1 again but now for the outer contour seeds of the darker neighbored shapes.

The gradient finally happens inside of this new shape and looks like the one pictured in Fig. 4.9(d).

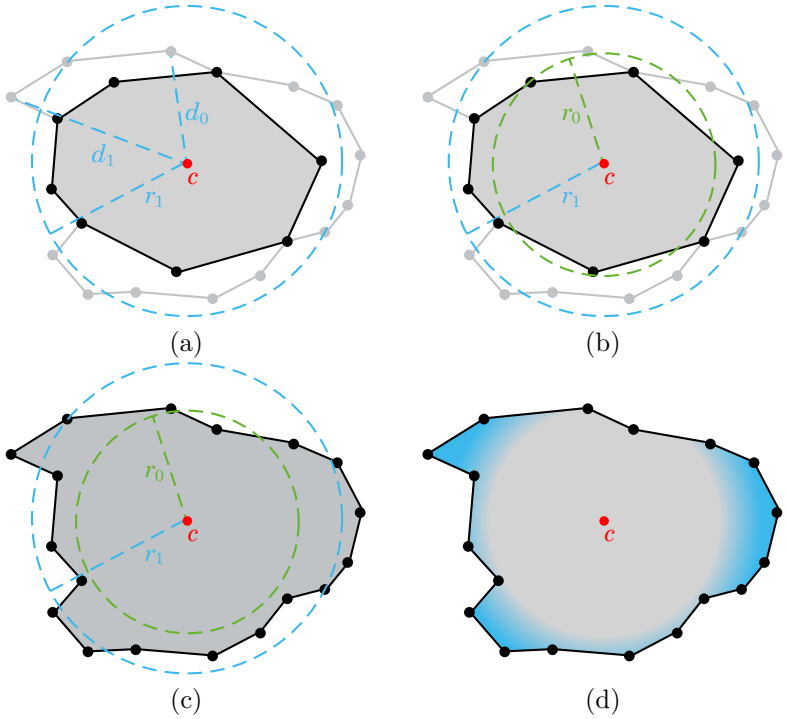


Figure 4.9: The second part of a radial gradient creation. The radius for the larger circle is again calculated as medium distance of the smallest distance d_0 and the greatest distance d_1 of the center and the outer contour seeds for the neighbor shapes in (a). The resulting two circles for the gradient are shown in (b) indicated by differently colored dotted lines together with the belonging radii in the same color. In (c) the result of the lighter and the darker shapes being merged is visible. The new outer contour is visualized as black colored seeds. Also the two circles of the gradient can be seen. The gradient finally takes place in between those two circles. The result of the radial gradient calculation is pictured in (d).

Chapter 5

Examples and Evaluation

In this chapter some example images are shown picturing all the steps that were needed to create the resulting images. All the different images are ordered in the same way to make a comparison between the single images easier. Also some evaluation is done on the different methods that were used and how good they performed.

The first picture for every test image shows the original input image. These original images are pictured in Figs. 5.1(a), 5.2(a), 5.3(a), 5.4(a), 5.5(a) and 5.6(a). The test images vary in size between a 10×10 and a 28×30 size and a color amount between two and fourteen colors. The first steps of generating the similarity graph (see Fig. 5.1(b) for example) and the Voronoi diagram (see Fig. 5.1(c) for example) are the same for all test images used. In this steps the connections of a pixel and its neighbors are introduced and resolved in some special cases of crossing diagonals. Afterwards the pixels are reshaped to make the connections to the neighbors even more visible and more detailed. These steps work perfectly fine for all test images. After the Voronoi diagram has been generated the big shapes, pictured in Fig. 5.1(d) for the "Space Invader" test image, can be created by combining several neighbored shapes with the same color to one big shape. This procedure is also the same for all test images and produces the expected output as well. The next thing to do was extracting splines from this big shaped image. For this some different ways were discovered and tested. First of all, the Catmull-Rom interpolation was introduced. This method produced even bigger shapes as before (see Fig. 5.1(e) for the first test image). With the single shapes now overlapping, the result looks good indeed, but it is not an optimal solution. Some negative effects of this approach can be seen in Fig. 5.3(e) or Fig. 5.4(e) were one or even both of the eyes disappear after being covered by an overlying larger shape. To improve the result and resolve the covering problem the quadratic B-Splines (which can be seen in Fig. 5.1 (f) for the "Space Invader" test image again) were introduced. These splines, however, came out non-optimal again, by generating holes between

the single shapes that use a common outer contour seed. To improve on these holes the next step was to use cubic Bézier curves instead of quadratic B-Splines to make the holes look smaller (see Fig. 5.1(g)) by approximating more towards the shape's outer contour. This worked a little bit better, but could not fully close the holes. To make sure the gaps disappear the seeds that were used by more than two shapes are included into the interpolation process. This led to the result pictured in Fig. 5.1(h) were no gaps appear between the shapes anymore. One can say that this solution would be good enough but with a higher resolution one can still see the small straight line sections on the contours that happen because of the approximation of the Bézier curves. So for a final optimization the cubic Bézier spline was not approximated but generated via PostScript using the iText library that perfectly generated the cubic Bézier curves and exported the whole image as PDF file. Also the color gradients were introduced wherever possible. For the test images in Figs. 5.1(a), 5.2(a) and 5.6(a) no color gradient was generated because either the color distances were to big to introduce a gradient or the overall amount of color in the image was considered to small. For the input images shown in Figs. 5.3(a), 5.4(a) and 5.5(a) a color gradient was generated, which can be seen in the very last picture of the belonging test sets. In general generating the Bézier curves with iText works perfectly fine and the introduction of the color gradient works fine for all test images as well, however the radial gradient tends to produce smoother gradients and works a little bit better than the axial gradients in terms of smoothing the neighbored colors.

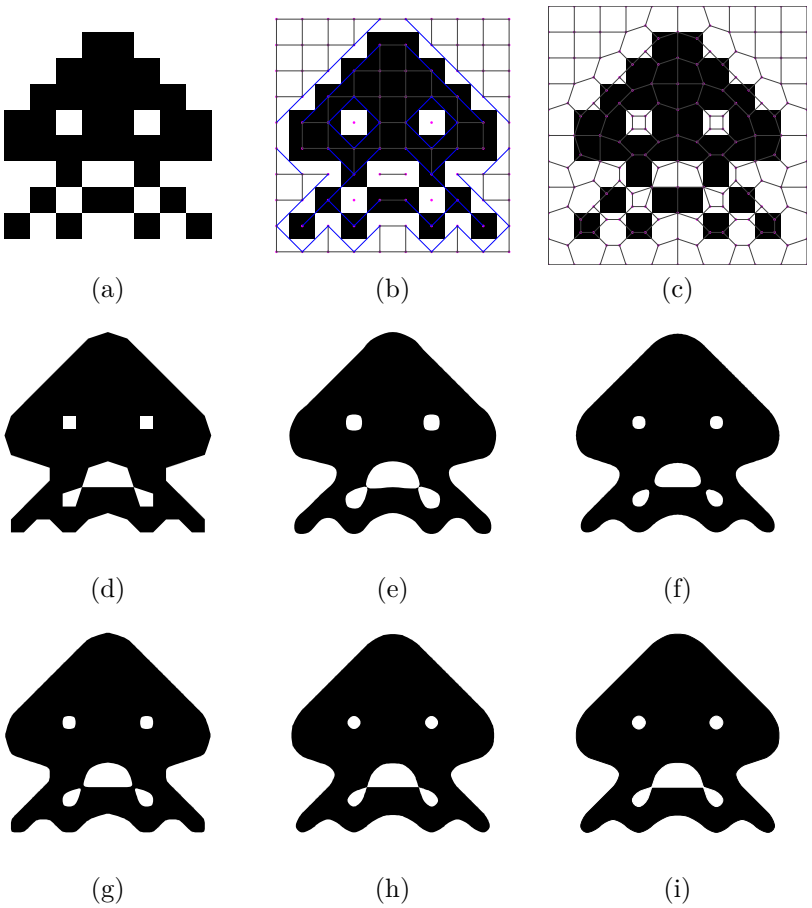


Figure 5.1: All implementation steps for the test image "Space Invader". The original input image is shown in (a). Generating the similarity graph (b), the Voronoi diagram (c) and the big shapes (d) works perfectly fine. The Catmull-Rom interpolation in (e) produces overlapping shapes and the interpolation with the quadratic B-Spline method (f) produces the opposite worst case with the holes. The interpolation with the cubic Bézier splines is shown in (g), which produces smaller holes, and the same method but with resolved multiple seed usage is pictured in (h), which lets the holes disappear. Picture (i) shows the final result with the export to PDF. There was no color gradient introduced, because of the color amount of the input image being too small.

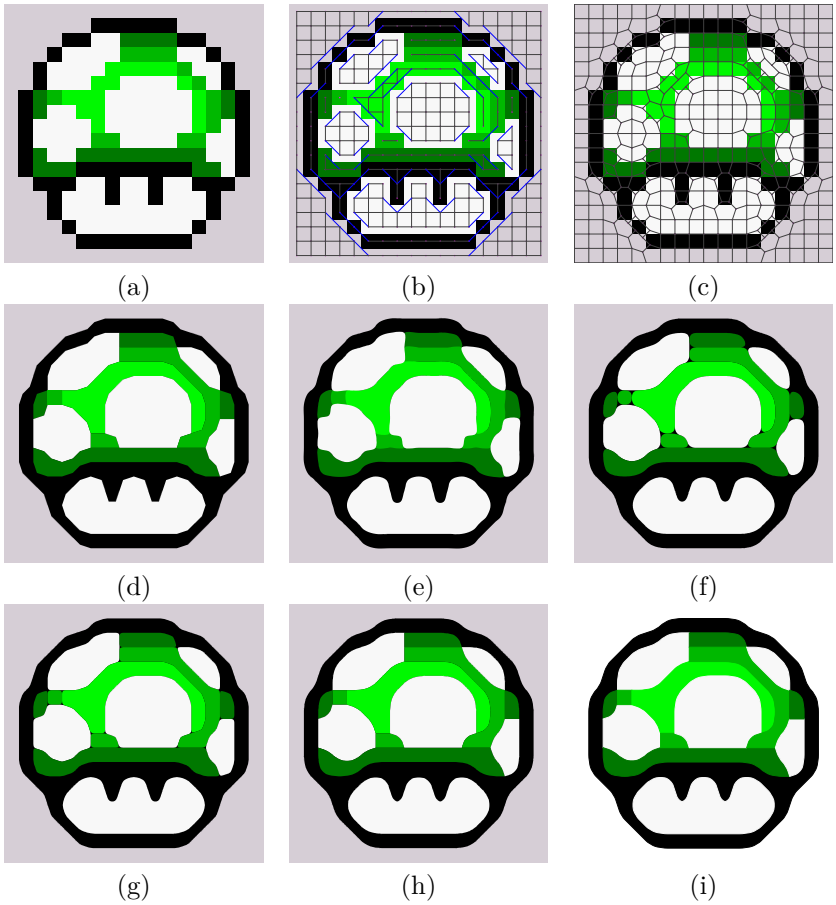


Figure 5.2: All implementation steps for the test image "Mushroom". The original input image is shown in (a). Generating the similarity graph (b), the Voronoi diagram (c) and the big shapes (d) works perfectly fine. The Catmull-Rom interpolation in (e) produces overlapping shapes and the interpolation with the quadratic B-Spline method (f) produces the opposite worst case with the holes. The interpolation with the cubic Bézier splines is shown in (g), which produces smaller holes, and the same method but with resolved multiple seed usage is pictured in (h), which lets the holes disappear. Picture (i) shows the final result with the export to PDF. There was no color gradient introduced, because the color distances were considered too big.

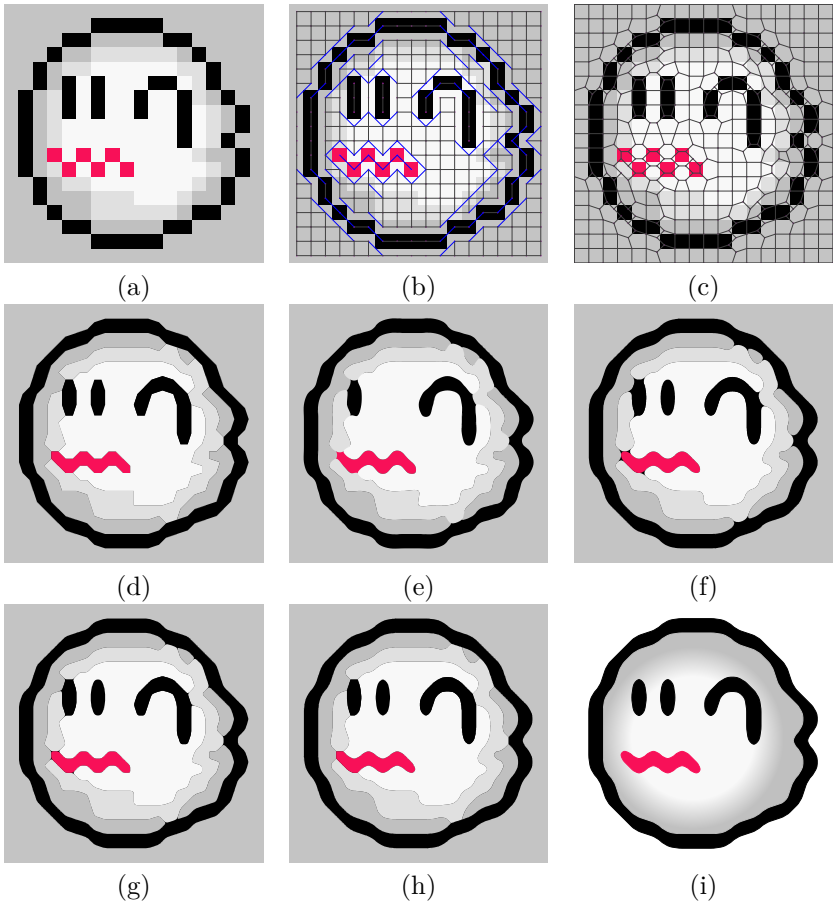


Figure 5.3: All implementation steps for the test image "Boo". The original input image is shown in (a). Generating the similarity graph (b), the Voronoi diagram (c) and the big shapes (d) works perfectly fine. The Catmull-Rom interpolation in (e) produces overlapping shapes and in this case even one shape is completely overlapped by a larger shape, because of which an eye disappeared in the resulting image. The interpolation method with the quadratic B-Spline (f) produces the opposite worst case with the holes showing. The interpolation with the cubic Bézier splines is shown in (g), which produces smaller holes, and the same method but with resolved multiple seed usage is pictured in (h), which lets the holes disappear. Picture (i) shows the final result with the export to PDF. In this specific case, only one radial gradient is produced.

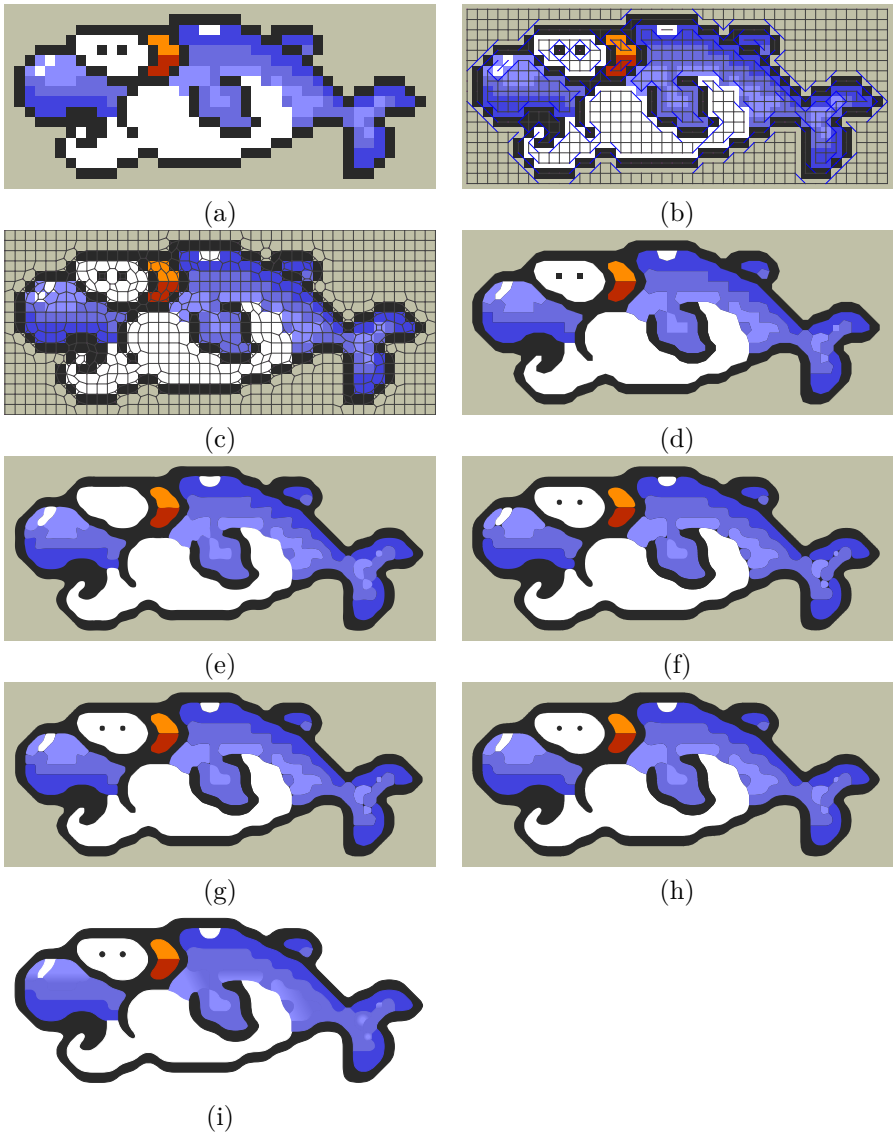


Figure 5.4: All implementation steps for the test image "Dolphin". The original input image is shown in (a). Generating the similarity graph (b), the Voronoi diagram (c) and the big shapes (d) works perfectly fine. The Catmull-Rom interpolation in (e) produces overlapping shapes, because of which both eye shapes are completely overlapped by a larger shape. The interpolation with the quadratic B-Spline method (f) produces the opposite worst case with the holes showing. The interpolation with the cubic Bézier splines is shown in (g), which produces smaller holes, and the same method but with resolved multiple seed usage is pictured in (h), which lets the holes disappear. Picture (i) shows the final result with the export to PDF.

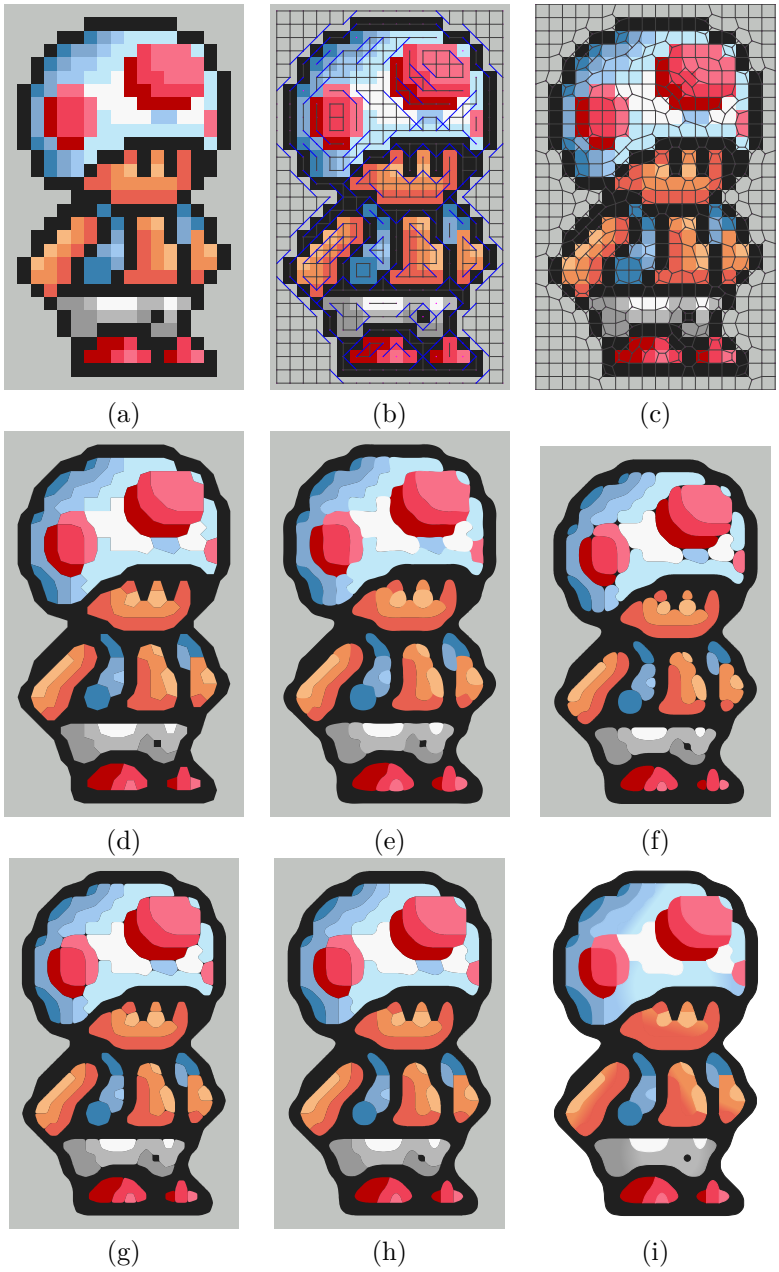


Figure 5.5: All implementation steps for the test image "Toad". The original input image is shown in (a). Generating the similarity graph (b), the Voronoi diagram (c) and the big shapes (d) works perfectly fine. The Catmull-Rom interpolation in (e) produces overlapping shapes and the interpolation with the quadratic B-Spline method (f) produces the holes. The interpolation with the cubic Bézier splines in (g) produces smaller holes and the same method but with resolved multiple seed usage is pictured in (h), which lets the holes disappear. Picture (i) shows the final result with the export to PDF.

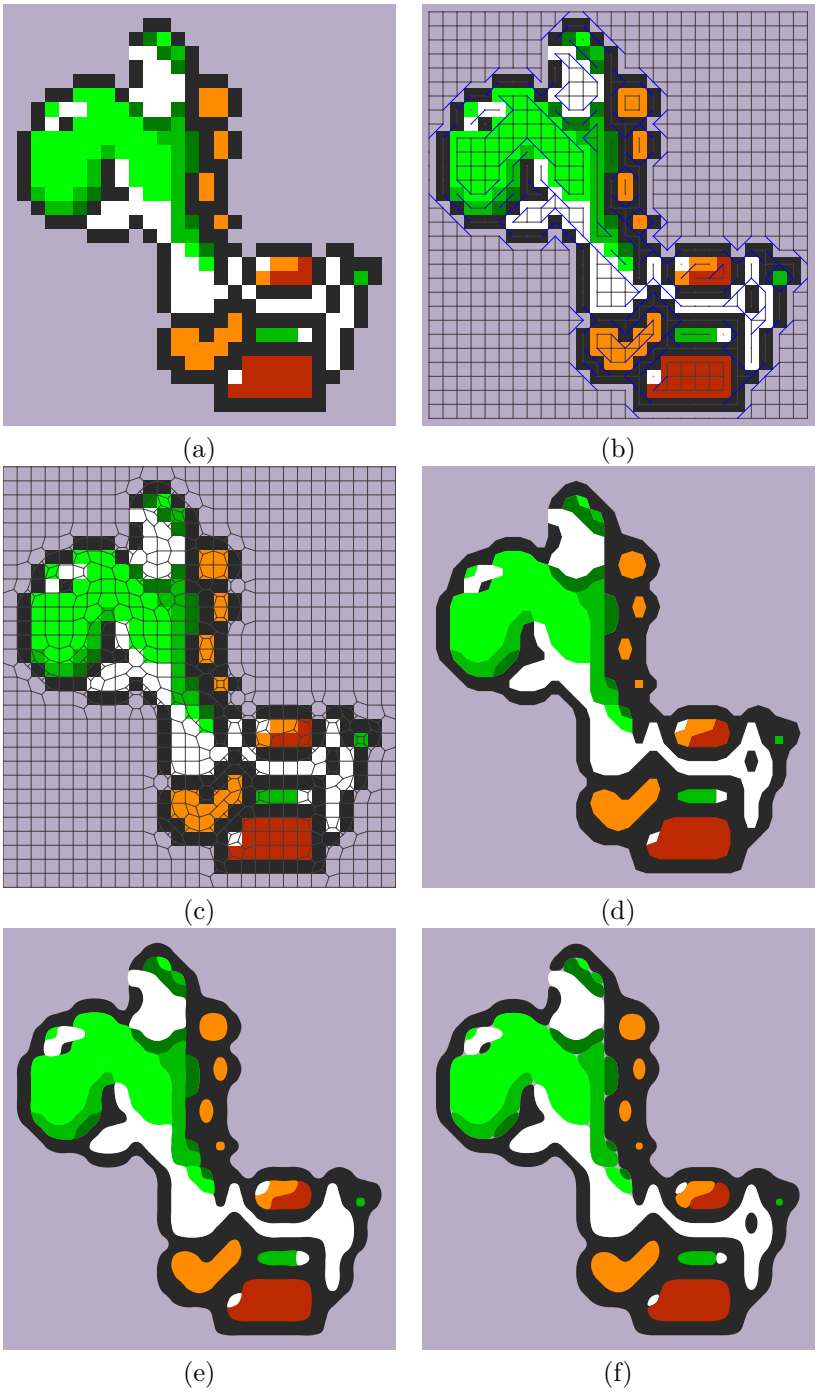
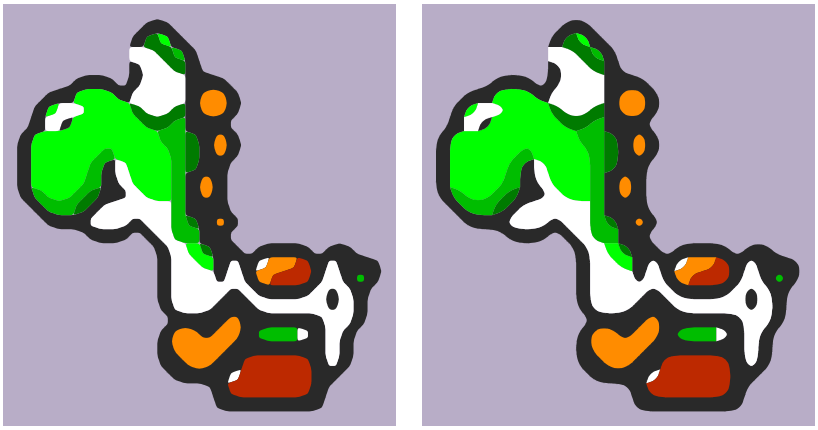


Figure 5.6: The first steps for the test image "Yoshi". The original input image is pictured in (a), the similarity graph in (b) and the Voronoi diagram in (c). Picture (d) shows the big shapes, (e) the Catmull-Rom interpolation and (f) the interpolation with the quadratic BSpline method.



(a) (b)

(c)

Figure 5.7: The last steps for the test image “Yoshi”. The interpolation with the cubic Bezier splines is shown in (g), the optimized method with the resolved multiple seed usage is pictured in (h) and (i) pictures the final result.

Chapter 6

Conclusion

As the example images in Chap. 5 show, the outcome of the implemented algorithm works pretty fine and the results are very smooth looking. Compared to the input images, the results keep the color amount and the beauty of simplicity by being smoother and best of all, scale invariant.

The algorithm works perfectly fine up to this stage and the steps that Kopf and Lischinski used for their original algorithm were successfully transferred to Java/ImageJ. Wherever there were any difficulties, the steps of solving these issues were included into the description. The additional steps and improvements, that were introduced in this thesis, are explained step by step. Additionally, before and after images are pictured to make the changes even more understandable. With the introduction of the gradients the images got their finishing touch in the end.

As the color gradients are a special eye catcher in the resulting images, there are some situation where the color transition may attract some negative attention with coming across too harsh. Unlike the radial gradient, which works perfectly fine in all images, the linear gradient could be improved in further projects. One thing that comes to mind is dividing the two shapes meeting at the common contour into several smaller sections and applying a linear gradient to each one. For this an appropriate method to divide the shapes and apply the color gradients would be necessary. Another potential issue for further projects would be the optimization of the execution time. As all the steps used in this implementation are computationally intensive, the execution time increases with the size of the image. But as this project is intended to work for very small images especially, it did not affect this project in a negative way.

Appendix A

Technical Information and Source Code

A.1 Technical Information

Eclipse Version: 4.5
ImageJ Version: 1.50d–1.51b
iText Version: 5.5.8

A.2 Source Code

In this part of the appendix a helpful code snippet is shown that is mentioned in the thesis. The complete source code will be included in the final upload for the master's project.

The following plugin creates a radial/axial gradient in a 18×18 area. There is no export of a PDF included because it is only a demonstration of the gradient initialization.

```
1 import {...}
2
3 public class TestImagesIText implements PlugIn{
4
5   private static String OutputDirectory = IJ.getDirectory("home");
6
7   public void run(String arg) {
8     drawImage(OutputDirectory);
9   }
10
11  public static String drawImage(String path) {
12    int width = 18;
13    int height = 18;
14
15    BaseColor startColor = new BaseColor(0, 200, 255);
16    BaseColor endColor = new BaseColor(100, 255, 0);
```

```
17
18      Document document = new Document(new Rectangle(width, height));
19
20      try {
21        PdfWriter writer = PdfWriter.getInstance(document,  new
        FileOutputStream(path));
22        document.open();
23        PdfContentByte cb = writer.getDirectContent();
24
25        /* axial shading (writer, startPoint, endPoint, startColor, endColor,
        continue before startPoint, continue after endPoint) */
26        PdfShading axial = PdfShading.simpleAxial(writer, (float)width/2,
        0, (float)width/2, (float)height, startColor, endColor, true, true);
27        PdfShadingPattern axialGradient = new PdfShadingPattern(axial);
28
29        /*radial shading (writer, midPoint1X, midPoint1Y, midPoint2X,
        midPoint2Y, startColor, endColor, continue before startPoint, continue after
        endPoint) */
30        PdfShading radial = PdfShading.simpleRadial(writer, (float)width
        /2, (float)height/2, (float)1.5, (float)height/2, (float)width/2, (
        float)height/2, startColor, endColor, true, true);
31        PdfShadingPattern radialGradient = new PdfShadingPattern(radial);
32
33        cb.setShadingFill(radialGradient);
34        cb.moveTo(0, 0);
35        cb.lineTo(width, 0);
36        cb.lineTo(width, height);
37        cb.lineTo(0, height);
38        cb.closePath();
39        cb.fill();
40      }
41
42      catch (DocumentException de) {
43        IJ.log(de.getMessage());
44      }
45      catch (IOException ioe) {
46        IJ.log(ioe.getMessage());
47      }
48      document.close();
49      return path;
50    }
51 }
```

Appendix B

DVD Content

Format: CD-ROM, Single Layer, ISO9660-Format

B.1 PDF-Files

Pfad: /

_DaBa.pdf . . . . . Masterthesis (whole document)

B.1.1 Literature

Pfad: /literature/printed

*.pdf . . . . . Copies of the used literature as PDF documents.

B.1.2 Online-Literature

Pfad: /literature/online

*.html . . . . . Copies of the used online literature as HTML documents.

*.pdf . . . . . Copies of the used online literature as PDF documents.

B.2 Java-Files

Pfad: /implementation/java/*

*.java . . . . . Java Source Code for the ImageJ PlugIn.

B.3 Miscellaneous

Pfad: /images/*

*.png . . . . . Original Pixel Images
*.pdf . . . . . Original Vector Images

References

Literature

[1] Wilhelm Burger. *Spline Interpolation of 2D contours*. Lecture notes in Computer Vision, University of Applied Sciences Upper Austria. 2008 (cit. on p. 40).

[2] Wilhelm Burger and Mark James Burge. *Digitale Bildverarbeitung*. 2nd ed. Springer, 2015 (cit. on pp. 4, 7–10).

[3] John Canny. "A computational approach to edge detection". *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8.6 (1986), pp. 679–698 (cit. on p. 19).

[4] Tom Fletcher. *Introduction: What is Image Processing?* Lecture notes in Image Processing Basics, The University of Utah. 2012. URL: http://www.coe.utah.edu/~cs4640/slides/Lecture0.pdf (cit. on p. 3).

[5] Eric Johnston. *Eric's pixel expansion (EPX)*. 1992. URL: https://en.wikipedia.org/wiki/Image_scaling (cit. on p. 11).

[6] Johannes Kopf and Dani Lischinski. "Depixelizing pixel art". *ACM Transactions on Graphics (TOG)* 30.4 (2011), pp. 99–106 (cit. on pp. 1, 22).

[7] Bruno Lowagie. *iText in Action*. Manning Publications Co., 2007 (cit. on p. 50).

[8] Tom Lyche and Knut Mørken. *Spline Methods Draft*. Lecture Notes in Spline Methoden, University of Oslo. 2008. URL: http://www.uio.no/studier/emner/matnat/ifi/INF-MAT5340/v05/undervisningsmateriale (cit. on p. 43).

[9] Alexandrina Orzan et al. "Diffusion curves: a vector representation for smooth-shaded images". *Communications of the ACM* 56.7 (2013), pp. 101–108 (cit. on pp. 18, 20, 21).

[10] David Salomon. *Curves and surfaces for computer graphics*. Springer Science & Business Media, 2007 (cit. on pp. 42, 43).

[11] Daniel Silber. *Pixel art for game developers*. CRC Press, 2016 (cit. on p. 2).

[12] George Wolberg. *Digital image warping*. IEEE Computer Society Press, Los Alamitos, CA, 1990 (cit. on pp. 7, 8, 10).

Online sources

[13] Wikipedia the free encyclopedia. Jan. 2016. URL: https://en.wikipedia.org/wiki/Image_processing (visited on 22/02/2016) (cit. on p. 3).

[14] Wikipedia the free encyclopedia. Jan. 2016. URL: https://en.wikipedia.org/wiki/Digital_image_processing (visited on 22/02/2016) (cit. on p. 3).

[15] Wikipedia the free encyclopedia. Jan. 2016. URL: https://en.wikipedia.org/wiki/Image_editing (visited on 22/02/2016) (cit. on p. 3).

[16] Wikipedia the free encyclopedia. Jan. 2016. URL: https://en.wikipedia.org/wiki/Image_scaling (visited on 23/02/2016) (cit. on pp. 7, 8, 10).

[17] Edu Garcia. Jan. 2012. URL: https://github.com/Arcnor/hqx-java (visited on 23/05/2016) (cit. on p. 15).

[18] Alex Handson-White. *Pixel Artist's Beginner Booklet, Chpt. 1*. Jan. 2007. URL: http://www.alexhw.com/pabbc1.pdf (visited on 05/02/2016) (cit. on p. 2).

[19] Adobe Systems Incorporated. Jan. 2016. URL: http://www.adobe.com/products/illustrator (visited on 15/02/2016) (cit. on pp. 5, 20, 43).

[20] Adobe Systems Incorporated. Jan. 2016. URL: http://www.adobe.com/products/postscript/ (visited on 16/02/2016) (cit. on p. 43).

[21] Adobe Systems Incorporated. *Auto Tracing and Resolution*. Jan. 2015. URL: https://helpx.adobe.com/illustrator/how-to/illustrator-auto-tracing-and-resolution.html (visited on 07/03/2016) (cit. on p. 20).

[22] Adobe Systems Incorporated. *Learn the Gradient Mesh tool*. Jan. 2015. URL: https://helpx.adobe.com/illustrator/how-to/illustrator-growing-gradient-mesh-tomato.html (visited on 05/02/2016) (cit. on p. 19).

[23] Adobe Systems Incorporated. *Nachzeichnen von Bildmaterial mit „Interaktiv nachzeichnen" oder Vorlagenebenen*. Jan. 2016. URL: https://helpx.adobe.com/de/illustrator/using/tracing-artwork-live-trace-or.html (visited on 07/03/2016) (cit. on p. 20).

[24] Techopedia Incorporated. *Pixel Art*. Jan. 2016. URL: https://www.techopedia.com/definition/8884 (visited on 05/02/2016) (cit. on p. 2).

[25] Andrea Mazzoleni. *Scale2x*. Jan. 2001. URL: http://www.scale2x.it/algorithm (visited on 10/03/2016) (cit. on p. 12).

[26] Andrea Mezzoleni. Jan. 2015. URL: https://github.com/amadvance/scale2x (visited on 23/05/2016) (cit. on p. 12).

[27] ITWissen: Das große Online-Lexikon für Informationstechnologien. Jan. 2016. URL: http://www.itwissen.info/definition/lexikon/Bildverarbeitung-image-processing.html (visited on 22/02/2016) (cit. on p. 3).

[28] ITWissen: Das große Online-Lexikon für Informationstechnologien. Jan. 2016. URL: http://www.itwissen.info/definition/lexikon/Bildbearbeitung-image-editing.html (visited on 22/02/2016) (cit. on p. 3).

[29] Tamme Schichler. Jan. 2012. URL: https://bitbucket.org/Tamschi/hqxsharp (visited on 23/05/2016) (cit. on p. 15).

[30] Marc-Michael Schoberer. Aug. 2014. URL: http://www.gutenbergblog.de/gestaltung-design/pixel-vs-vektor-4365.html (visited on 05/03/2016) (cit. on p. 4).

[31] Peter Selinger. *Potrace: a polygon-based tracing algorithm*. 2003. URL: ftp://94.198.20.42/pub/Graph/potrace.pdf (cit. on pp. 16–18).

[32] Photography on stackexchange. Jan. 2016. URL: http://photo.stackexchange.com/questions/60983 (visited on 22/02/2016) (cit. on p. 3).

[33] Romy Stein. May 2013. URL: http://www.saxoprint.de/blog/unterschied-pixelgrafik-vektorgrafik (visited on 05/03/2016) (cit. on p. 4).

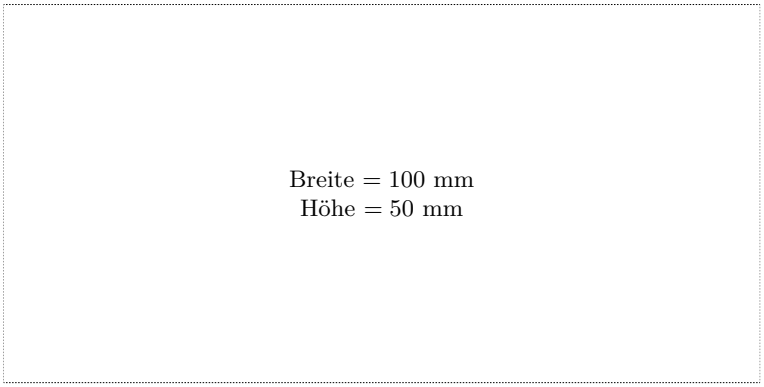
[34] Maxim Stepin. *Demos and Docs - hq2x/hq3x/hq4x Magnification Filter*. Jan. 2003. URL: http://web.archive.org/web/20070624082212/http://www.hiend3d.com/hq2x.html (visited on 10/03/2016) (cit. on p. 15).

[35] Angie Taylor. *Bitmap vs. vector*. Dec. 10, 2013. URL: https://helpx.adobe.com/illustrator/how-to/illustrator-bitmap-vs-vector.html (cit. on p. 4).

[36] Wayne Rasband. *ImageJ*. Version 1.50. 1997. URL: http://imagej.nih.gov/ij/index.html (cit. on p. 26).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —