Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Photo Stylization Using Painterly Rendering

Bc. Jan Lazarek

Supervisor: prof. Ing. Daniel Sýkora, Ph.D.
May 2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Lazarek** Jméno: **Jan** Osobní číslo: **474599**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačová grafika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Stylizace fotografie pomocí simulace malby

Název diplomové práce anglicky:

Photo Stylization Using Painterly Rendering

Pokyny pro vypracování:

Seznamte se s metodami pro automatickou konverzi fotografie na stylizovanou malbu, jenž využívají pokročilé filtrační techniky [1, 2] a simulaci tahů štětcem [3, 4]. Na základě těchto postupů se pokuste rekonstruoval existující řetězec operací, který navrhl profesionální výtvarník v nástroji Adobe Photoshop. Rozložte řetězec na dílčí kroky a pokuste se reprodukovat jejich chování volbou vhodného algoritmického řešení. Navržený stylizační nástroj rozšiřite tak, aby umožňoval parametrickou či na předloze založenou kontrolu. Uživatel - výtvarník tak bude moci nastavením parametrů či dodáním vhodných alternativních vzorů ladit výsledný vzhled stylizace. Chování implementovaného nástroje průběžně konzultuje s výtvarníkem. Ověřte jeho správnou funkcionalitu srovnáním s výstupy původního sledu operací na několika praktických příkladech, které dodá výtvarník.

Seznam doporučené literatury:

[1] Kyprianidis: Image and Video Abstraction by Multi-scale Anisotropic Kuwahara Filtering, Proceedings of the 9th Symposium on Non-Photorealistic Animation and Rendering, pp. 55-64, 2011.
[2] Semmo et al.: Image Stylization by Interactive Oil Paint Filtering, Computers & Graphics 55:157-171, 2016.
[3] Zeng et al.: From Image Parsing to Painterly Rendering 29(1):2, 2009.
[4] Lindemeier et al.: Artistic Composition For Painterly Rendering, Proceedings of the 21st International Symposium on Vision, Modeling and Visualization, pp. 119-126, 2016.

Jméno a pracoviště vedoucí(ho) diplomové práce:

prof. Ing. Daniel Sýkora, Ph.D. Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **06.02.2022** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **30.09.2023**

prof. Ing. Daniel Sýkora, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání Podpis studenta

Acknowledgements

I am thankful to my master's thesis supervisor Prof. Ing. Daniel Sýkora, Ph.D. and Jakub Javora for continuous support during this project and also to the Czech Technical University in Prague and especially to the Faculty of Electrical Engineering for providing me with an opportunity and environment to work on this master's thesis.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 19, 2022

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 19. května 2022

Abstract

The subject of this work is a research of methods and a proposal of automatization of procedures that enable the stylization of photography into digital painting. The reference to this project is the current work and workflow of a graphics artist who specializes in digital painting. The content of this work consists of an analysis of currently used methods, a proposal, and an implementation of a solution, which enables part automatization of this creative work.

Keywords: stylization, digital painting, image filtering, Kuwahara filter, painterly rendering

Supervisor: prof. Ing. Daniel Sýkora, Ph.D.

Abstrakt

Předmětem této práce je zkoumání metod a návrh automatizace postupů umožňujících stylizaci fotografie do digitální malby. Předlohou tohoto projektu je tvorba a proces vzniku děl grafika specializujícího se na digitální malbu. Obsahem práce je analýza stávajících postupů, návrh a implementace řešení, které umožní částečnou automatizaci tvůrčího procesu.

Klíčová slova: stylizace, digitální malba, filtrování obrazu, Kuwahara filtr

Překlad názvu: Stylizace fotografie pomocí simulace malby

Contents

Figures

Tables

Chapter 1

Introduction

As the possibilities of the use of computers grow, so does the number of industries in which computers are used. These include various human activities, including those that used to be purely handiwork. One such branch is the fine arts.

Fine artists are often skeptical of any involvement of computers in the process of creating their art. They fear that the use of computer-controlled technologies may lead to a devaluation of their creations, therefore, there is often an effort to avoid or even despise these methods. The fine artists often think that the work they create is given by a set of values such as the artist's inner intention, talent, experience in the field and personal ability to specifically perceive the world around them, and then bringing all this into their work.

The process of creating a piece represents a type of communication in which the creator, based on his feelings and impressions, creates a piece of information that he wants to pass on to the viewer. However, adding a computer to this communication channel as an intermediary or even a co-creator may disrupt the initially intended pure communication between the artist and the viewer.

There is already a separate branch of fine arts, known as digital art. It is a set of techniques, tools, and procedures in which the computer is used to create the artwork and mediate the result to the viewer. However, this does not change the fact that the artist remains the main one who makes art and the computer is only his tool.

An example of this art is the so-called digital painting. The creative process here can be very similar to creating a classical painting. The only difference is that the brush used by the artist is electronic and that the canvas is moved to a monitor display. One of the fields where digital painting is applied is concept art.

These days, there are already artists, especially from the younger generation, who have practical experience with the use of computer technology in their daily personal lives. Thus, they are able to realize and project the positives that computer technology can bring to their artistic work. As a part of this project, there was an opportunity to meet and work with Jakub Javora, a concept artist who has already mastered a computer as his working tool.

In this work, we explore and research how Jakub can use automation and image processing methods in his creative process. This project aims to analyze and understand his current workflow of digital painting and based on the gathered knowledge, we will propose and develop image-processing methods that would reduce the time that the artist devotes to less creative and repetitive tasks helping him to focus on the creative work itself. To be able to improve the current workflow, it was first needed to understand individual steps of the workflow.

1.1 Analysis of the current state

The idea behind this project was born based on the long-term cooperation of Prof. Daniel Sýkora, a supervisor of this project and Jakub Javora, a concept artist who, in his work, uses digital painting methods. An integral part of his work is to research and test new approaches that he could incorporate into his creative process. In this work, we decided to focus on a digital painting created based on a template, which can be either a photograph or a rendered image. These resources have in common that they faithfully capture reality from a visual standpoint. At the same time, the photos are not just inspirations in this particular process, but are directly modified to create the final art piece, which can be seen in Figure 1.1.

The process of creating an art piece could be called the process of stylizing photography into painting. At the input, there is a photograph to which several adjustments are applied, and the result is a digital painting. With the current creative process, the artist already uses a computer and several partially automated procedures and operations performed on the picture. We decided on such a solution proposal which would automate more of these operations, and thus shorten the creation of the final work. The crucial prospect of these solutions is that they do not significantly change the current process and do not add new difficulties to an already complicated workflow.

Currently, the artist uses the Adobe Photoshop environment for his work. This tool was originally designed purely for manual image processing and editing. Over the time, Adobe added so-called "actions", in which it is possible to define sequences of actions and adjustments that are automatically applied to an image. Similarly, the ability to write custom scripts has been added to Photoshop. The purpose of these tools is to automate specific repetitive tasks using tools already built directly into the program. However, they are not made exactly for implementing other new procedures.

The current creative process consists of four phases, which will be described below.

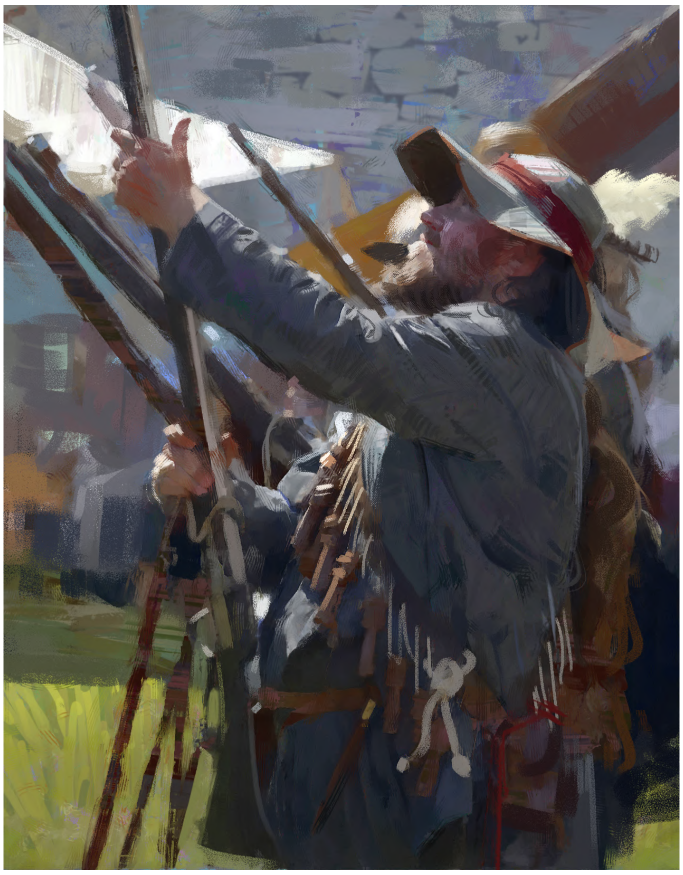

Figure 1.1: Example of a finished digital painting

1.1.1 Photo disruption

The first phase of the creative process can be described as disrupting the perfection of photography. Photography is currently a medium that can perfectly capture the environment around us, which has its advantages and disadvantages. The advantage is that the amount of information and details that we can obtain from the photo is so large that we should not need to invent any additional data in its processing. We should be able to get all the information from the image using just the appropriate methods and operations. However, the perfection of photography is twofold.

On the other hand, it is challenging to artistically grasp the photographed scene while maintaining all its specific content. Meaning that if we want to process the photograph further artistically and visually, we will have to disrupt the perfection and precision of the details in some way. For this purpose, the artist uses mainly manual techniques. In Photoshop, these are primarily tools, such as blur, liquefaction, or brush tool. The main goal of this step is to disrupt the perfect shading and crisp edges of the image. This results in an image containing fewer and less clear visual aids that our brain uses to imagine the original real-life scene or an object that was a template for a given painting. This modification not only helps the artist so that he does not connect the image that much to reality anymore, but it also helps with further editing of the image. Afterwards, it resembles a flat painting and not a photograph that is still cognitively three dimensional.

Furthermore, the artist uses filtering methods and monadic operations applied to the whole image. Optical "flattening" of a photograph is often performed, which is achieved by reducing the brightness of the light parts of the picture and, conversely, increasing the brightness in the dark parts of the photograph, especially in the shadows. Then, a colour adjustment is performed which involves changing the brightness and saturation of each colour. A median filter, for example, is also used for the interference of photograph perfection. Examples of such adjustments can be seen in the Figure 1.2. Although the filtering successfully distorted the original image information in both cases, many valuable details were lost. It is necessary to remember that this step's goal is not to lose or add too much detail to the image. The goal is to transform the image. The main problem in this step are the filters themselves. Even though it is possible to "flatten" the image enough, there is no adequate solution to edit the picture in a way that it would not resemble a photograph anymore to the artist.

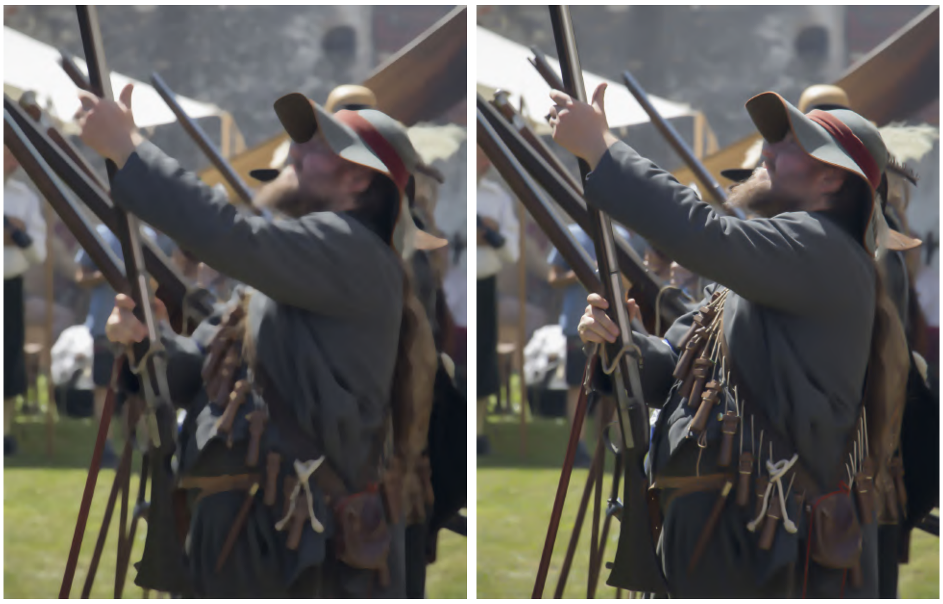

Figure 1.2: Demonstration of image distortion on the left using a median filter and on the right using a similar filter that preserves small structures

1.1.2 Rough painting

The second phase is a basic rough digital painting, which, like the physical one, is made up of individual layers. First, the artist starts with the background and then roughly paints the basic structures and objects in the painting. He is gradually moving to smaller and more complex objects or objects that he wants to come out of the picture visually at the end. Similarly in physical painting, it is possible to work with layers in Photoshop. Moreover, the great advantage of Photoshop is the ability to return to any layer, edit it, change the order of the layers, or mix them as desired.

Rough painting is the first phase that is at least partially automated. The first step is to obtain information about the image's local structure, which is necessary to obtain the prevailing directions in individual parts of the picture. To do this, the artist uses the "oil painting" styling filter that Photoshop offers. Its output is then thresholded, and thus a texture based on Photoshop-generated brush strokes is obtained. Then a vectorization is performed to obtain curves that copy these strokes. The artist applies this action to variously scaled source images and thus gets several layers of differently complex curves. Finally, an action is used, in which Photoshop draws individual brushstrokes along the curves created in this way. The result is several layers of strokes that the artist must manually blend into one final image, called underpainting.

The problem with this approach lies within the Photoshop-generated vectors. It is hardly possible to have any control over their generation. The only possible parameter to manipulate them is their density based on the image resolution. The second problem is that every path that Photoshop generates is a closed loop instead of just an open path. This results in rendered brushstrokes having this closed-loop characteristic, which is not strictly faithful to actual hand-made strokes.

Another method with which the artist works at this stage is displacement mapping. The map itself is a single-channel texture where based on its value at a given pixel, the pixel of the image being edited is moved in a particular direction. This is just the basic idea of this method, and the implementation is a bit more complex in practice. Both methods add many brush-like structures to the image. However, neither of these approaches create visual content that would add any added value to the work from an artistic point of view. In summary, the result seems relatively underwhelming and, in some areas, the image may become unreadable at this stage. The result of the adjustment using displacement mapping can be seen in the image 1.3.

1.1.3 Addition of structures

In the third phase, the artist adds structures that are not present in the original photo. Several methods try to simulate classical painting techniques. The aim is to add some added value to the painting. This phase is purely manual. The artist himself shows his own style and invention in the painting, he paints strokes that may seem random at first glance, but each one of them has its own meaning. In this way, it is often possible to highlight specific

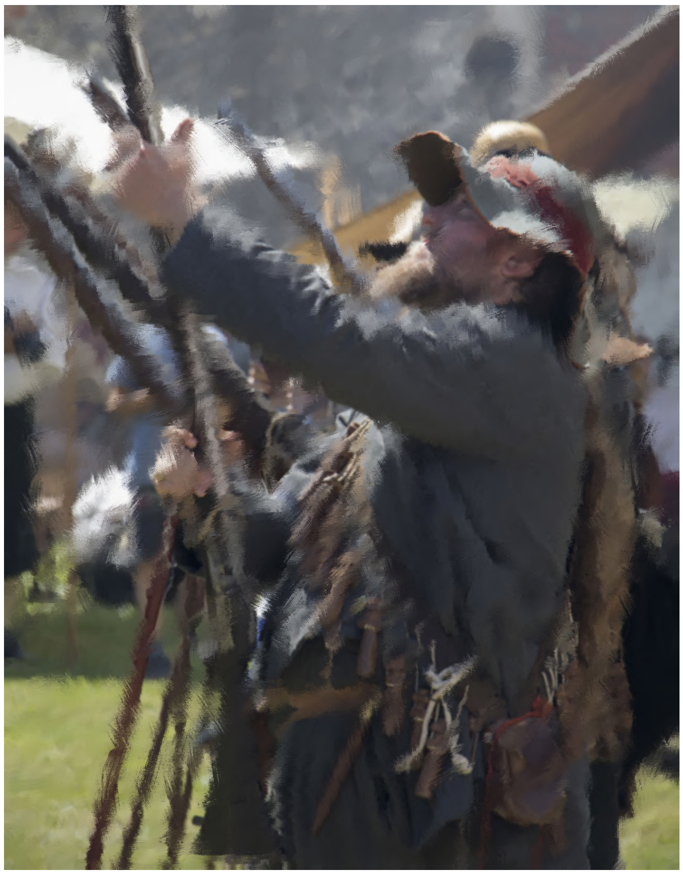

Figure 1.3: Image editing using displacement mapping

structures, either by the particular shape of the strokes or by using colours that may not appear at all in the original image. An example of such precisely adjusted tonality can be seen in the figure 1.4.

One of the techniques performed in this phase corresponds to the dry brush painting used in the real painting. As the name suggests, the painter uses a brush that is just barely soaked in paint or the paint used is relatively thick. This, combined with a canvas structure or previously applied colour, results in an effect when the brush stroke is irregular and randomly discontinuous. Since there is no implementation of physics in most graphics software that would model any interaction between a brush and a canvas, the artist cannot directly replicate this effect. Instead, many different brush textures are part of any program, usually with the option of importing or even creating one's own brush textures. The use of many different brush textures can be seen in the figure 1.5, especially around the edges, for example, around the guns or the beard.

The artist also tried to automate this part of his creative process. He came up with a computer-generated method based on hand painting. The first step is to segment the image into sections. Next, it is possible to apply a

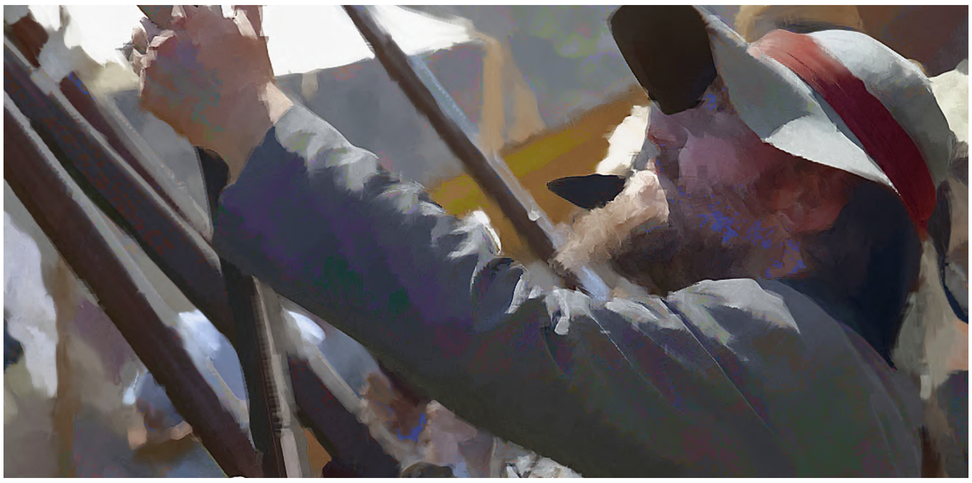

Figure 1.4: Demonstration of adjusted tonality in the beard and on the sleeve of the figure

pre-prepared texture containing relatively random brush strokes to individual parts of the picture. Although some parts of this operation can be done automatically, in practice, each of the described partial steps requires a relatively large degree of manual intervention. Therefore, this method is still relatively time consuming.

1.1.4 Return and creation of details

In the fourth phase, the artist adds structures back into the painting that may not be present in the original photo or have disappeared with gradual adjustments. These can be, for instance, small details on the face, typically the eyes. In addition, different patterns are added to the painting, such as highlighting the original fabric pattern or leaves in the vegetation. The artist tries to improve the previous steps of his creative process to such an extent that he can minimize the amount of these final adjustments. Currently, all these details have to be manually repainted, as shown in figure 1.5 or blended directly from the output of one of the previous steps.

1.2 Conclusion

Based on the analysis of the current state and the knowledge collected about the existing workflow, it was possible to get an idea of which parts of the process could or should be improved. The first is the image disruption, which is not a completely solved problem yet. To that is connected the step of rough painting, where the result of these two steps combined creates a so-called underpainting. The next great topic that we can work on is the brush strokes or, more precisely, their automatic generation. Even though the artist already uses a semi-automatic process which he was mostly capable of tuning to his liking, we both agreed that there was still room for improvement.

Lastly, even though it does not come directly from the analysis, it arises from the painting and artistic composition. It is a fact that creating digital painting often consists of mixing individual, often differently stylized segments of images. This means that the program should support either an image segmentation or operations with semitransparent images.

The following chapters will discuss related works and the theory connected to this problem. We will propose a solution and its implementation, which will be tested in the end, and we will discuss the results of the whole work.

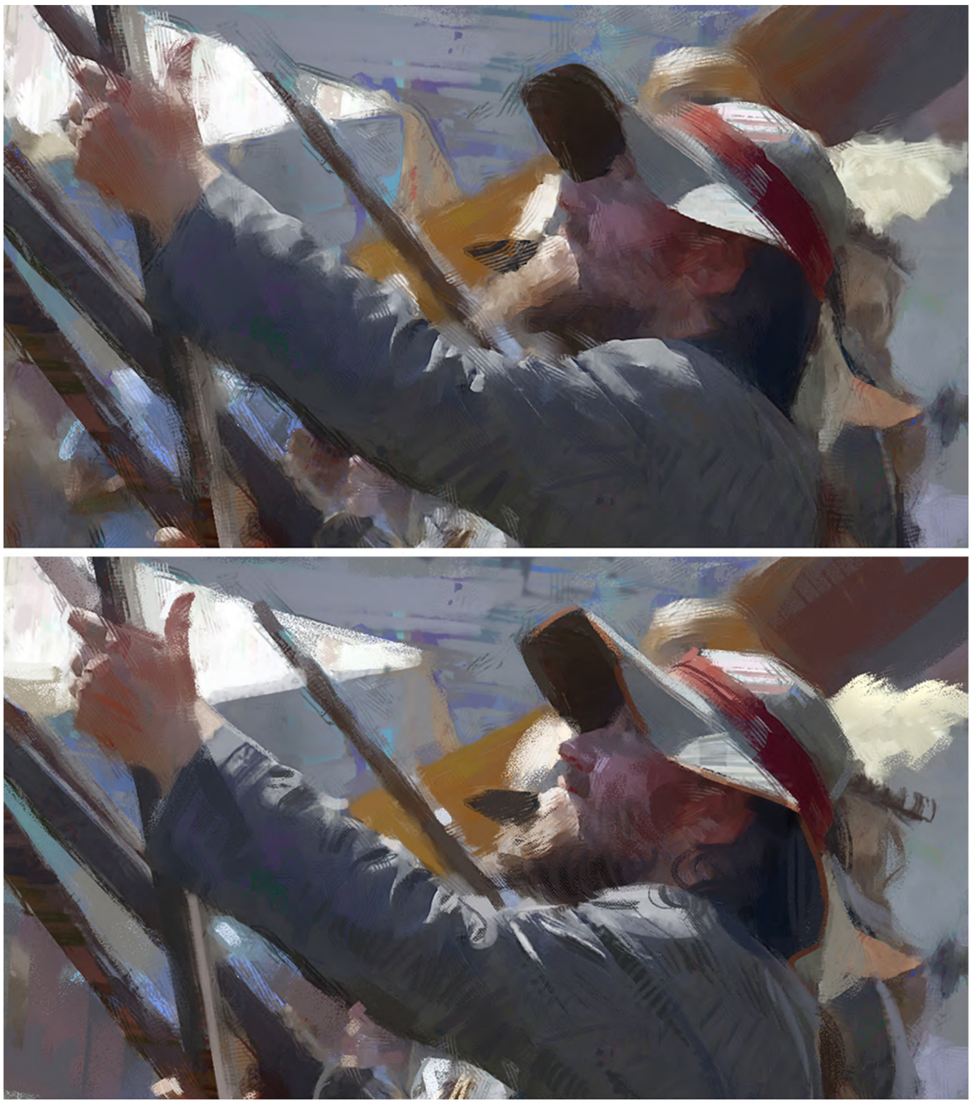

Figure 1.5: Demonstration of additional details on the hat and sleeve of the figure, also the results of use of textured brushes are visible especially around edges in both images.

Chapter 2
Related works

In this chapter, we will write about the basic concepts of this work and, for each of them, mention several previous related works and research on which this project is based or similar topics.

2.1 Painterly Rendering

Brushstrokes are the main building blocks of each painting. Moreover, this holds even in the case of digital painting. However, the approach to stroking and the medium used are different. As described in [vSJ21], in real life, painters use pigment-based colours, which are applied onto a canvas, usually using a brush or another utensil. Several factors, ranging from the type of colour used to the angle at which a painter holds a brush, determine the final characteristics of a brushstroke. In the figure 2.1 below, we can see several different types of brushes where each of them has its purpose of creating slightly different strokes and, therefore, being used under different circumstances. For example, the filbert brush can create a thick, from-the-end rounded stroke, but also a relatively thin one when rotated on a chisel edge. The second example is where a mop brush would usually be used to paint with a thin colour and, as a result, be able to apply just a thin film or glaze onto a previously painted area.

In contrast to this, a digital artist does not have a very straightforward approach, which means that he cannot determine the appearance of a brushstroke just by manually tampering with another brush and making a specific movement with it. To substitute manual control, an artist can use many controls and settings in painting programs that try to emulate nuance that can otherwise be done only by hand. The settings range from those that simulate the physical characteristics of a brush such as its size, hardness, and shape to more advanced options such as brush texture or the type of colour mixing. In Photoshop, there is also a tool called a mixing brush. It tries to represent the feeling of a painting with a real brush soaked in paint, which enables colour mixing even though it is just mixing in an RGB colour space, which differs from the actual pigment mixing. Despite the fact, this may seem like the possibilities of digital painting are drastically limited compared to physical painting, it is not entirely true. In reality, the techniques are just different and adapted to the virtual workspace, meaning that artists can use most methods or achieve visual characteristics with a few workarounds.

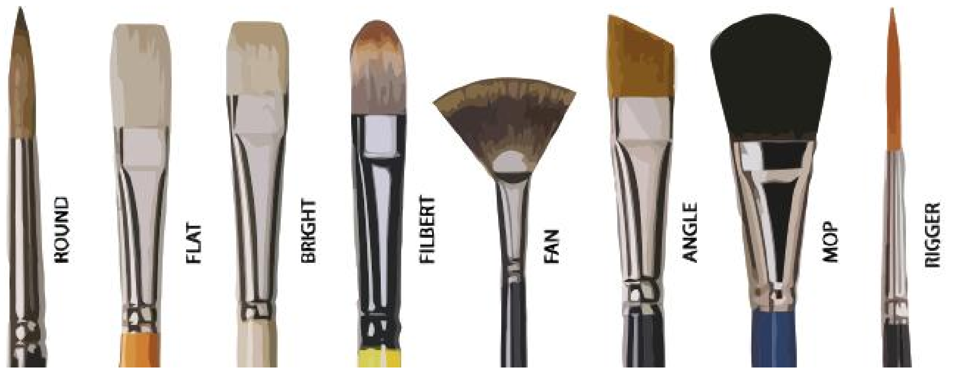

Figure 2.1: Example of brush types.

Brush stroke rendering is an area of research that has been well explored since the graphical user interfaces were added to computers. Also, there are several powerful programs, such as the Adobe Photoshop mentioned above. Therefore, we will not further discuss stroke rendering itself in this project. On the other hand, as was discussed in chapter 1.1, the current artist's process of painting consists of two parts, the second of which is rendering the strokes. This is preceded by the generation of paths along which Photoshop will draw the strokes. This is a nontrivial task that we will discuss in more detail in this work.

The first attempts to capture and create visual art using computers were made in the 1980s. These were mainly rendering methods that tried to replicate the visual impression of the painting, especially a brushstroke. It is worth mentioning the article by Dr. Steve Strassmann presented at the 1986 SIGGRAPH conference [Str86]. In this work, he focused on drawing strokes that looked like someone painted them with a brush, as shown in figure 2.3.

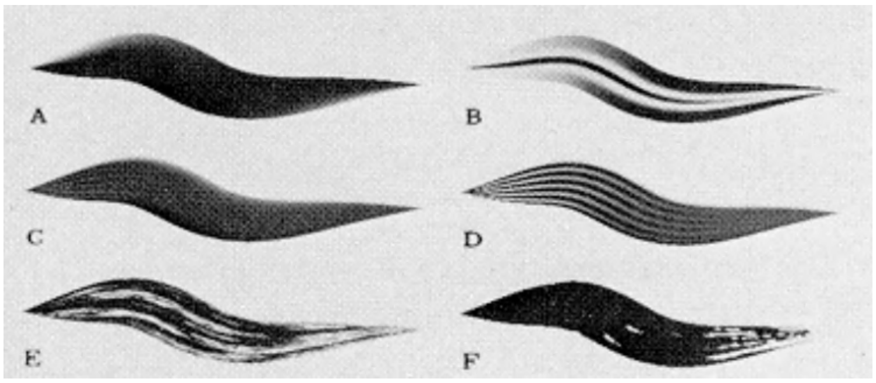

Figure 2.2: An example of brush strokes from [Str86]

In 1990, Paul Haeberli came up with an article on semiautomatic painting systems that can build paintings from strokes of different sizes and shapes [Hae90]. Furthermore, in 1997, Salisbury et al. focused on filling user-defined areas in an image with drawing-like textures. In this method, the user first defines the individual areas of the drawing together with the dominant direction in the given area. The program then colours or rather hatches these areas, as seen in figure 2.3 [SWHS97]. In 1998, Aaron Hertzmann came up with a new method of drawing brush strokes. It allows the renderer to arbitrarily turn the strokes and, in combination with the use of a bump map, achieves a final appearance that is very similar to oil painting [Her98]. So far, these have been works that dealt purely with drawing strokes to achieve a visual perception that will be as similar as possible to drawing or painting.

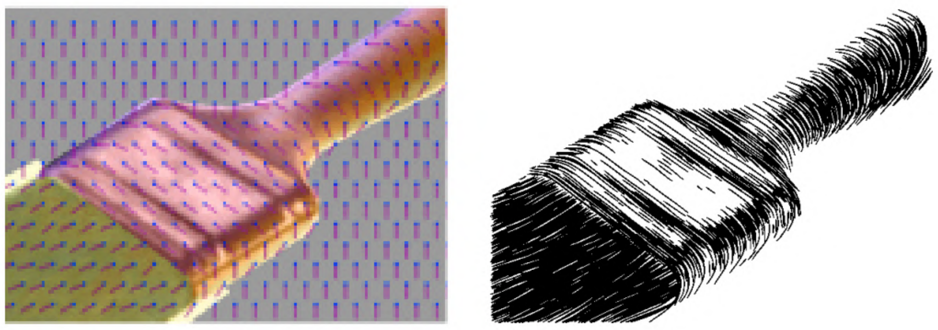

Figure 2.3: An example of the orientation and curvature of strokes from an article by Salisbury et al. [SWHS97]

2.2 Image stylization

Image stylization is an artistic process in which we have a source image that we want to modify so that it looks like the picture has been created with a different technique than it actually was. There are two main approaches to this problem. The first option is to hardcode the style we want to achieve into some type of filter. Most often, we encounter a situation where we have a source photo that we want to modify so that it looks like an oil painting. Most of these techniques try to replicate the first impression of the image, meaning that it uses similar colours or textures as the artwork we want to replicate. But in reality, all these methods usually do is a camouflage that works at first glance. Still, if we look closer, the actual strokes look entirely different from the ones the painter would actually make.

The second approach to this is the so-called style transfer. In this case, the method needs two images. The first one is the one we want to stylize, and the second one contains the art style to which we wish the image to stylize. These methods usually try to understand what actually is in the picture, and then the stylization proceeds based on the objects or shapes in the image. An interesting example of this approach is the article FaceBlit: Instant Real-time Example-based Style Transfer to Facial Videos [TTK+21]. The result of this method can be seen in figure 2.4.

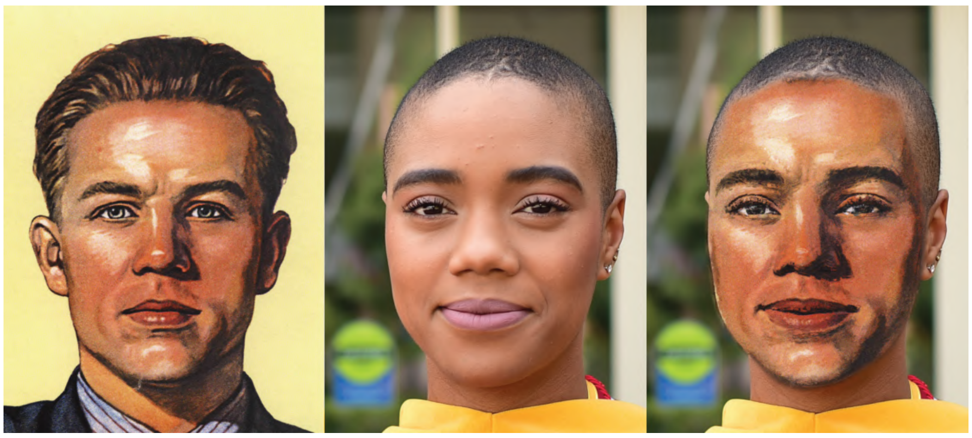

Figure 2.4: An example of style transfer from style source image (left) onto a source photo (middle) and the result (right) [TTK+21]

Another group of related works can be called procedural filtering. These automatic image stylization methods rely only on standard signal processing algorithms. A noteworthy article is Real-time Video Abstraction by Winnemöller et al. of 2006 [WOG06]. The work builds on bilateral filtering and shows how fast a computer can process an image. Multiple pieces of research are based on the so-called image flow, which is based on determining the local orientation in the image and its anisotropy. Visualization of image flow can be seen in the figure 2.5. This procedure is based on the calculation of the image structure tensor [KD08]. There are also several works by Jan Eric Kyprianidis that deal with image filtering using the anisotropic Kuwahara filter [KKD09]. A comparison of filters can be seen in the figure 2.6. Most of the above works were selected from the Artistic Stylization by Nonlinear Filtering by John Collmoss and Jan Eric Kyprianidis [Kyp13].

2.3 Image segmentation

Image segmentation is a process that allows us to divide the input image into regions or objects. Depending on the purpose of the segmentation, there are many different techniques. [GW08] There are two fundamental approaches to this problem. Firstly, we can segment the image based on the information saved in the image itself. Examples of this method is segmentation based on a colour or brightness. The other approach is to use image semantics. In this case, we often try to recognize and segment the foreground from a background or some particular objects in the scene, for example, humans. These two approaches often correspond to a partial, respectively, complete segmentation. The complete segmentation aims for a set of disjoint regions

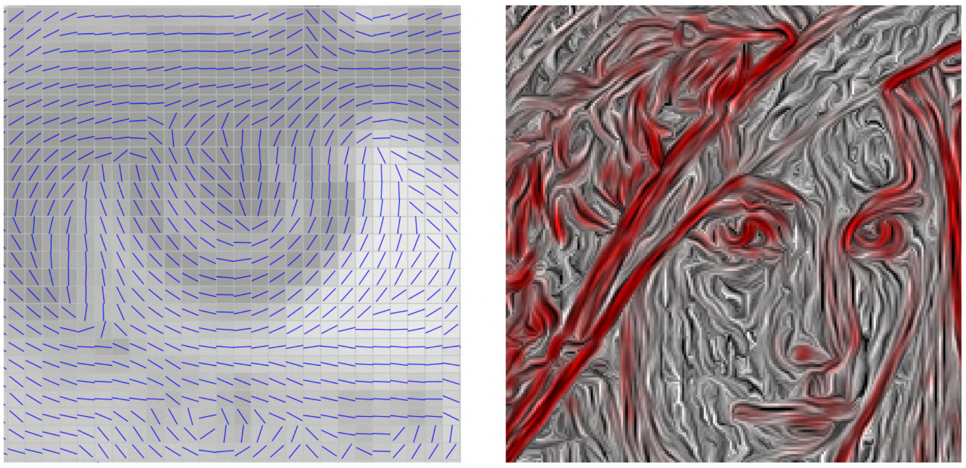

Figure 2.5: Tangent field induced our the smoothed structure tensor. Gradients with high magnitude are highlighted in red [KD08]

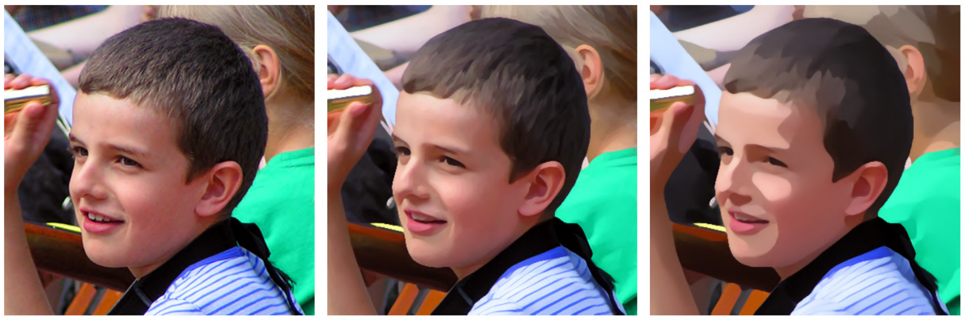

Figure 2.6: Comparison of methods from the article by Jan Eric Kyprianidis [Kyp11]. Gradually from left: original photo, anisotropic Kuwahara filter, multi-scale anisotropic Kuwahara filter.

corresponding uniquely to objects in the input image. On the other hand, the result of a partial segmentation does not necessarily correspond to the actual objects. It can be just a set of colour patches [SHB14].

The most straightforward image segmentation methods are based only on the value of every pixel, that is, on its colour or brightness. This method is called thresholding and, as the name suggests, is based on a threshold value. The idea behind this approach is simple. We perform a transformation of each pixel based on its value $I(x, y)$ compared to the threshold T as follows:

$$J(x,y) = \begin{cases} 1 \; for \; I(x,y) > T \\ 0 \; for \; I(x,y) \leq T \end{cases} \tag{2.1}$$

The above definition considers just a single colour image, but the concept can be applied even to an RGB image.

The next group of segmentation techniques is based on edge detection. There are many approaches to edge detection, but all are based on finding a discontinuity in an image. The most obvious can be a colour discontinuity, but some methods take into account, for example, brightness or texture [SHB14]. The goal of these methods is to search for the edges in the image and, as a second step, to create edge chains that divide different segments of an image.

In contrast to edge detection techniques, region-based segmentation methods are also used. These methods are generally better for noisy input images, in which it can be a challenging task to detect edges. An important parameter used in these techniques is homogeneity, which describes how similar a given area is or how low a change of an observed parameter in a given area is. A notable example of this approach is a region merging method based on the concept of region growth. We can understand an input image as a matrix consisting of one-by-one regions. We can iteratively merge regions until a set condition is satisfied or until a set rule is not broken.

In this project, we need to approach segmentation in a way similar to what a painter would do. The first point of view is the segmentation based on the semantics of the image. The most basic common task of that would be the division of the image into foreground and background. Determining the foreground and background can be done interactively based on user-specified constraints as proposed in the article Graph Cuts and Efficient N-D Image Segmentation [BFL06]. An example of this approach can be seen in the picture 2.7. If we take an image as a graph, where each pixel is a node and each node connects to its four neighbors via an arc. We can express this problem as an optimization task which is defined as follow:

$$x^* = \underset{x}{\operatorname{argmin}} \sum_{\{p,q\} \in N} w_{pq} |x_p - x_q|^\alpha$$

$$subject\, to\colon x_s = 0 \land x_t = 1$$

where x^* is the optimal labelling of each image pixel, which minimizes a weighted sum of edges that will be excluded from the graph to create two separate graph components where one represents the background, and the other represents the foreground. Based on the value and meaning of parameters, we can transform this problem into other well-known problems for which there are already solvers. For example for $\alpha = 1$ and w_{pq} meaning a capacity of an edge, we get formulation of a minimum cut problem, or for $alpha = \infty$ and $w_{pq} = 1/length$ we get shortest path problem.

Although the method mentioned above may be helpful in the painting process, when we can separate and handle the background, in this case it may be necessary to use other techniques, which will be discussed in the chapter 4.

The first article to mention is the Stylization and Abstraction of Photographs by D. DeCarlo and Anthony Stanell [DS02]. In this work, they discuss the possibility of determining the crucial and visually exciting parts of the image and, on the basis of this, building a hierarchical structure of the image, based on which an image stylization is then performed. The second article is an

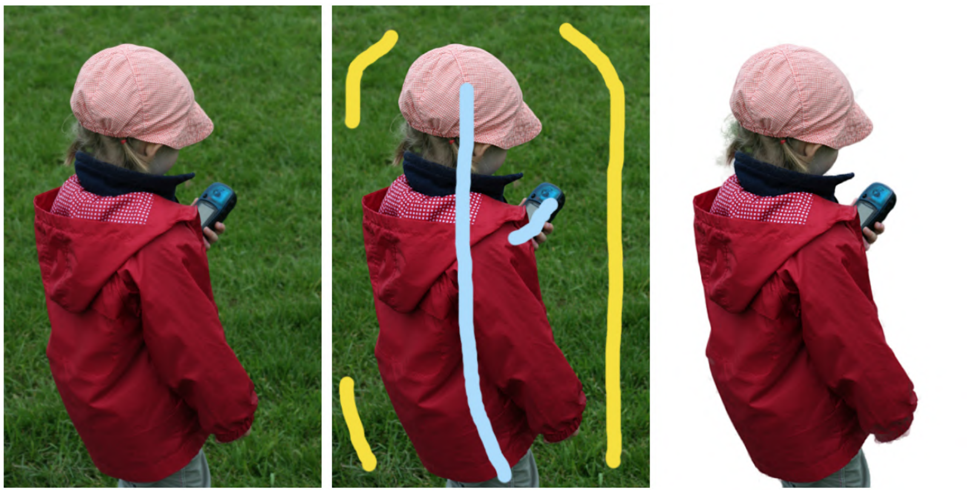

Figure 2.7: Example of segmentation based on user defined constrains.

Artistic Composition for Painterly Rendering by T. Lindemeier, M. Spicker and O. Deussen [LGD18] which discusses the option of decomposition of the canvas into a set of regions and layers. The goal was to replicate the standard painting process, which means that the image is usually painted from the back to the front. It results in smaller, more detailed strokes painted onto courser areas painted before. An example of this method can be seen in the figure 2.8.

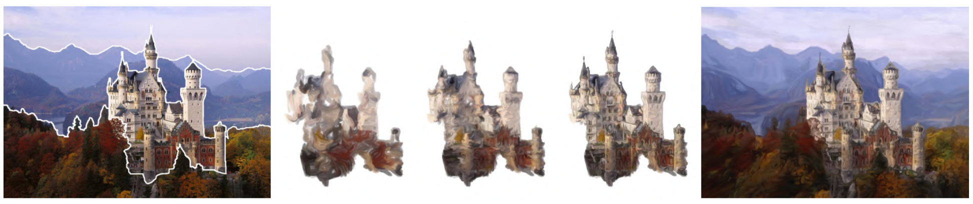

Figure 2.8: An example of image segmentation from [LGD18]

The last group of articles on image segmentation is primarily focused on automated image segmentation. The first is Normalized Cuts and Image Segmentation [SM00], where Jianbo Shi and Malik, J. propose a method that tries to segment an image based on its global impression. The next article is the Spectral Matting by Anat Levin, Alex Rav-Acha, and Dani Lischinski [LRAL07]. Spectral matting is the concept of segmentation of an object from a background, which additionally gives an estimation of how likely every pixel is a part of a given segment. This effectively returns a set of masks that can be further used in a graphical editor to segment and mix individual elements. This approach to segmentation was originally based on A Closed-Form Solution to Natural Image Matting [LLW08].

2.4 Painting stylization and brush stroking

This category of works is in some way a combination of the above. The first is Image Abstraction through Overlapping Region Growth by Rosa Azami and David Mould [AM20]. The goal of this work is to stylize images using image segmentation. Specific of this approach is that the regions are very irregularly shaped, which brings the impression of strokes, but that still respects image structure. An example of generated regions can be seen in the image 2.9.

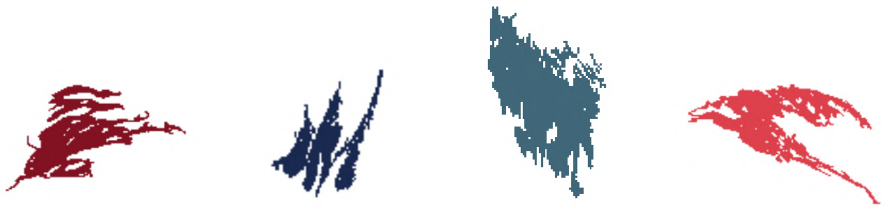

Figure 2.9: An example of irregular regions [AM20]

A new approach was introduced in the article Image Stylization by Oil Paint Filtering using Color Palettes by Semmo, A., Limberger, D., Kyprianidis, J. E., and Döllner, J.[SLKD15], where they propose a method for image transformation into an oil paint look. An example of this approach can be seen in the image 2.10. A big part of this article is dedicated to the work with a colour palette and texture, which should resemble brush strokes and oil paint. However, there is a part devoted to determining the flow in the image. We will discuss the image flow computation in section 3.2.5. To summarise, it is based on a structure tensor calculation determined by partial derivatives.

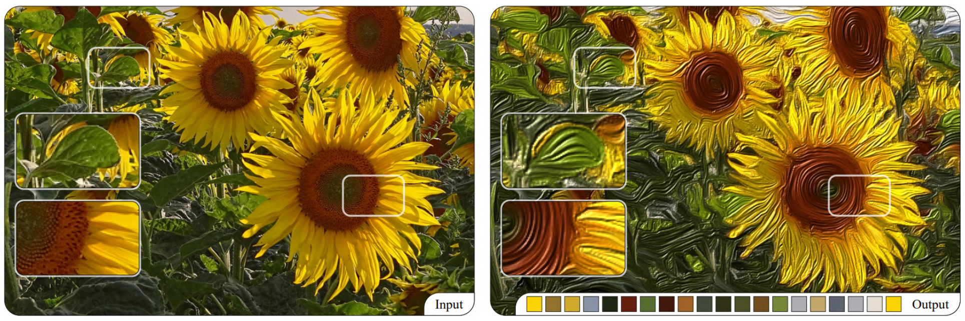

Figure 2.10: Example of oil paint filtering from [SLKD15]

Chapter 3

Background

In this chapter, we describe the basic theory and concepts related to the digital image and the operations that can be performed on it. Based on this theory, the algorithms and procedures listed in the following parts of the work will be defined.

3.1 Digital image

Digital image can be formally defined as a two-dimensional function $z = f(x, y)$ where the planar coordinates x, y determine the position in the image. The functional value of the function f at a given point determines the value of the image information, that is, for a black and white image, it would be the value of the brightness of the picture in a given pixel. By a function, an image is defined in a continuous domain, which is useful in the context of mathematically defined operations that can be performed on the image. However, if we consider the digital image represented per usual standards and viewable on a computer, it is necessary to transition to the discrete domain. Therefore, the coordinates x, y and the function value F have to take on finite discrete values [GRC08]. An image can then be understood in the discrete domain as a projection of a continuous world into the eye of an observer or a camera sensor [vBSF05]. The whole space of a digital image can then be defined as its domain consisting of the Cartesian product of the intervals defining the area of the image.

$$f : (\langle x_{min}, x_{max}\rangle \times \langle y_{min}, y_{max}\rangle) \tag{3.1}$$

Numbers $x_{min}, x_{max}, y_{min}, y_{max}$ denote the interval limits defining the image and therefore are true for $x_{min} \leq x \leq x_{max}$ as well as $y_{min} \leq y \leq y_{max}$.

As we know it, a digital image consists of individual pixels stacked in a usually rectangular grid or a pixel matrix called raster. Each pixel has one or more values that correspond to the value of the image function at that exact point. It can be a brightness level or a colour's intensity in a given pixel. We denote such an image function by I is defined by equation 3.2:

$$I[x, y] : N^2 \Rightarrow R^n \tag{3.2}$$

Where I is a discrete image function that assigns n real numbers to each coordinate pair x, y, the number of values n depends on the type of image. For example, $n = 1$ is enough for a black and white image, that is, we have only one value that determines the brightness value at the point. On the contrary, a standard colour image consists of three colour channels, red, green, and blue. In this case $n = 3$. For each pixel, we get the intensity value of each colour channel.

3.2 Digital image processing

Digital image processing means performing various operations on the image function the matrix in which the image data are stored. These operations can be divided into local and global operations, or in other words, those that affect only part of the image and the whole image, respectively. In this work, we will focus mainly on global operations, especially on filtering. Global operations are further divided into several groups, which will be described below.

3.2.1 Monadic operations

Monadic operations are usually simple operations performed on the image. By definition, the whole image is processed pixel by pixel, and a mathematical operation is applied to each pixel. Ultimately, this operation is the same for every single pixel of the image. The value of a given pixel is stored back to the appropriate position in the raster after transformation by a mathematical or other algorithmic operation. The advantage of this approach, where one pixel is processed in isolation from the rest of the image, is that this operation can easily be parallelized. Monadic operations include image negation, changing brightness, contrast, or image thresholding.

3.2.2 Convolution

The convolution operation is one of the basic filtering operations performed on images. In a continuous domain, it is defined by a relation:

$$(F * G)(s,t) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F(x,y).G(s-x,t-y)dxdy \tag{3.3}$$

for the image function F and the convolution kernel G, where s, t are the displacement parameters of the convolution kernel G. In practice, a gradual shift of the convolution kernel is performed over the entire area of the input image. The convolution kernel and image function are multiplied at each position corresponding to every pixel. These resulting values are summed and stored in the appropriate position in the raster. In this case, we can understand the convolution kernel as a weight function that determines the extent to which each pixel around the calculated pixel will contribute to the resulting value. We then obtain a similar relationship in the case of convolution over the discrete image function F and the kernel G:

$$(F * G)[s,t] = \sum_x \sum_y F[x][y].G[s-x][t-y] \tag{3.4}$$

There are many types of kernel that differ according to the result we want to achieve by convolution. Kernels can be of different sizes. For most of the operations described here, a 3x3 kernel is used by default. Standard kernels allow us to blur or sharpen the image, for example. Examples of kernels are given in Figure 3.1 below.

1	2	1	0	-1	0	1	0	-1	p_1	0	$-p_1$
2	4	2	-1	5	-1	2	0	-2	$1-2p_1$	0	$2p_1-1$
1	2	1	0	-1	0	1	0	-1	p_1	0	$-p_1$

Figure 3.1: Demonstration of convolution kernels, from left gradually Gaussian kernel, image sharpening, Sobel detector, Jähne et al. [JSK+99]

3.2.3 Application of convolution

There are several applications for which convolution is used and will be used in this work. First of all, it is a convolution with the so-called Gaussian kernel. This kernel and especially its shape are derived from the Gaussian function, which is defined by the relation:

$$f(x) = ae^{-\frac{x^2}{2\ sigma^2}} \tag{3.5}$$

for $a, \sigma > 0$ are any real numbers. The Gaussian kernel G itself is then created by combining two functions:

$$G(x,y) = g_x(x).g_y(s), \tag{3.6}$$

where $g_x(x), g_y(y)$ denotes a Gaussian function with one oriented along the x axis and a second one, oriented along the y axis. The lateral projection of which always has a silhouette corresponding to the Gaussian function from which the function G was created. The composition of the functions and its result are shown in figure 3.2.

The Gaussian function, and accordingly the resulting kernel, have one very disadvantageous feature for practical use, which is that its value approaches zero but never actually reaches zero. This feature is unsuitable for calculations because it would be necessary to work with a huge kernel and the contributions of pixels far from the calculated pixel would be almost negligible. For this reason, a truncated kernel is used, mainly to the size of 2σ. By performing the convolution with a Gaussian kernel, we achieve a blurring of the image from a visual point of view. The amount of image blur can be adjusted by changing the value of σ.

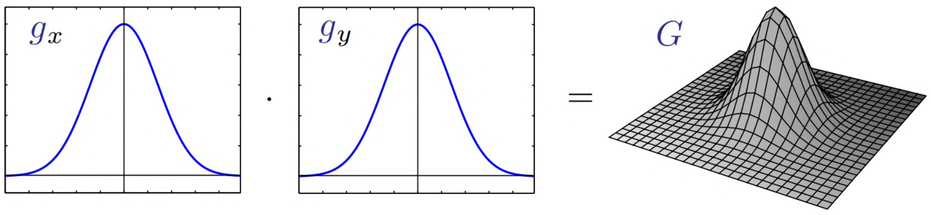

Figure 3.2: Demonstration of Gaussian function and its composition into a Gaussian Kernel [Sý21]

3.2.4 Partial derivatives of an image function

A particular application of convolution is the calculation of the image gradient. In mathematics, gradient magnitude corresponds to the rate of a change of a value of a function in a given direction. This is mainly used to detect areas of an image where the gradient magnitude is large enough, which Irwin Sobel first wrote about in [SF73]. These areas are perceived in the image as edges, hence the naming of the Sobel detector. To obtain partial derivatives, it is sufficient enough to perform an image convolution with the Sobel kernel, which is defined by the relation:

$$G'_x(x,y) \propto \frac{x}{\sigma^2} e^{-\frac{(x^2+y^2)}{\sigma^2}} \tag{3.7}$$

$$G'_y(x,y) \propto \frac{y}{\sigma^2} e^{-\frac{(x^2+y^2)}{\sigma^2}} \tag{3.8}$$

The first kernel is oriented in the direction of the axis x and the second in the direction of y. The Sobel kernel originated as an approximation of the first derivative of the Gaussian kernel along x and y. By applying the Sobel detector in the direction x and y, we obtain a partial derivative of the image in the direction x and y, respectively. The numerical approximation of this convolution kernel can be seen in the figure 3.1. From this computation, we get a pair of numbers for each pixel that denotes the slope of the given image function in both the x and y direction.

3.2.5 Local direction estimation in the image

The local direction describes the visual characteristics of the image in a given location. In other words, the direction in which the image optically flows in a given area. This characteristic usually depends on the shading of the object at a given position. If we look at the process from an artistic point of view, we can then come across the concept of shading by shape in the painting process. If we were to shade along with the shape, we would make strokes in the direction of pixels that have the most similar brightness. Firstly, to be able to paint these strokes, we need to retrieve the orientation information from the image based on the computation of partial derivatives.

The next step is to obtain the direction of the gradient, which, by definition, corresponds to the direction of the greatest growth of the image function. One way to obtain the direction θ of the gradient is

$$\theta = \tan^{-1}\left(\frac{g_y}{g_x}\right), \tag{3.9}$$

where g_x and g_y correspond to the partial derivatives of the image function in the direction x and y, therefore,

$$g_x = \frac{\partial I}{\partial x} \quad g_y = \frac{\partial I}{\partial y} \tag{3.10}$$

This approach to orientation estimation has one significant drawback, which is its great instability, which can be induced by any inaccuracies in the image, such as colour noise contained in the original image.

3.2.6 Structure tensor

The problem of instability of the previous approach is mostly solved by the concept called the structure tensor. This is an extension of the above procedure, which is described in [BY15]. In this case, the estimation of local orientation is based on the structure tensor's eigenvectors. As in the method described above, we obtain approximations of the partial derivatives $\frac{\partial I}{\partial x}$ and $\frac{\partial I}{\partial y}$, to be specific, using the Sobel filtering.

Let us have the Jacobi matrix J in the form:

$$J = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right) \tag{3.11}$$

From there, the structure tensor g_{ij} will have the form [KKD09]

$$g_{ij} = J^T J = \begin{pmatrix} \frac{\partial I}{\partial x}\cdot\frac{\partial I}{\partial x} & \frac{\partial I}{\partial x}\cdot\frac{\partial I}{\partial y} \\ \frac{\partial I}{\partial x}\cdot\frac{\partial I}{\partial y} & \frac{\partial I}{\partial y}\cdot\frac{\partial I}{\partial y} \end{pmatrix} =: \begin{pmatrix} E & F \\ F & G \end{pmatrix} \tag{3.12}$$

Subsequently, it is necessary to calculate the eigenvalues of the matrix g_{ij}, which we obtain by solving the equation $det((g_{ij} - \lambda_i I) = 0$, from which the eigenvalues $\lambda_{1,2}$ can be expressed by a relation:

$$\lambda_{1,2} = \frac{E + G \pm \sqrt{(E-G)^2 + 4F^2}}{2} \tag{3.13}$$

The eigenvectors v_1, v_2, bound to the calculated eigenvalues, are then obtained by the relation:

$$v_1 = \begin{pmatrix} F \\ \lambda_1 - E \end{pmatrix} \quad v_2 = \begin{pmatrix} \lambda_2 - G \\ F \end{pmatrix} \tag{3.14}$$

These two vectors are perpendicular to each other, and the highest rate of change at a given point in the image is in the direction of the vector v_1 and the smallest of the vector v_2. After calculating the eigenvectors for each pixel, we obtain a vector field that describes the local orientations in the entire image. Like the previous methods described in 3.2.5, we can encounter undesirable results and discontinuities caused by noise. The solution to this situation is to smooth the vector field using convolution with the Gaussian filter described in 3.2.3. The comparison of the orientation estimation between the Sobel detector and the structure tensor can be seen in the figure 3.3.

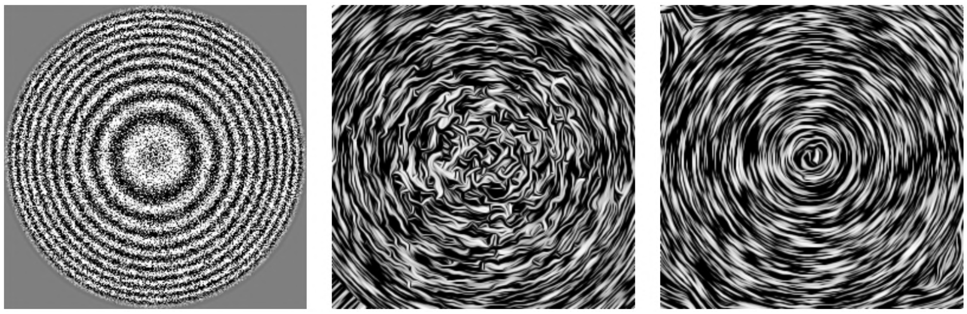

Figure 3.3: Direction estimation done by Sobel detector and smoothed structure tensor [Kyp11]

There may be areas in the image that are so flat that it is impossible to compute the derivatives correctly, or any imperfection of the image can greatly impact the computation. Thus, the resulting structure tensor might be completely incorrect. For these areas, we can perform a simple infill from [KK11] as described in equation 3.15.

$$S'(x,y) = \begin{cases} S(x,y) & if (x,y) \in \partial\Omega \\ \frac{S(x-1,y)+S(x+1,y)+S(x,y-1)+S(x,y+1)}{4} & otherwise \end{cases} \tag{3.15}$$

Here $S(x,y)$ denotes a structure tensor at position x, y and $(x,y) \in \partial\Omega$ denotes a structure tensor the amplitude of which is greater than a set threshold. This infilling iteration is performed many times until we reach a smooth distribution of gradients in a given image.

3.2.7 Curves

As previously mentioned, the rendering of brush strokes will not be directly part of this work. Also, we need to pass the generated stroke paths to a rendering engine, in this case Adobe Photoshop, which will draw the strokes for us. The simplest way is to take a set of curves and import them as "paths" into Photoshop.

There are many different types of curves in computer graphics that differ in their representation and use cases. There are three main categories based on the mathematical representation of curves: explicit 3.16, implicit 3.17, and parametric 3.18. In computer graphics, the most used types of curves are parametric ones, which can be understood as a trajectory of a point based on coordinates that are functions of a parameter t, usually in interval $\langle 0, 1\rangle$ [vBSF05].

$$y = f(x) \tag{3.16}$$

$$F(x, y) = 0 \tag{3.17}$$

$$x = f(t),\ y = g(t),\ t \in \langle t_{min}, t_{max}\rangle \tag{3.18}$$

Each curve is defined by a set of control points. Based on this, there are two main groups of parametric curves: interpolating and approximating. The difference between them is that the interpolation curve passes through each of the control points, whereas the approximation curve usually passes just through a subset of control points, as shown in image 3.4.

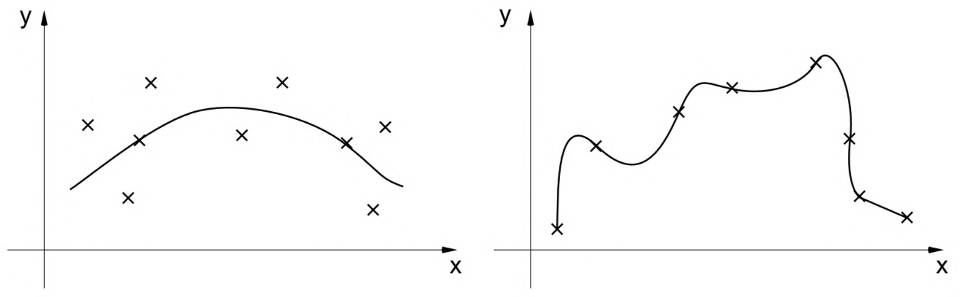

Figure 3.4: Example of aproximating curve (left) and interpolating curve (right) with their control points [vBSF05]

In this work, the Bézier curves will be used. They are the most popular and approximating curves used in 2D spaces. By the Bézier curve, we mean one segment of a curve. When we chain multiple segments together, we create a Bézier spline. The most commonly used form of the Bézier curve is the Bézier cubic. It is a curve segment in which the position and shape are always defined with four points. The start and end point through which the curve passes and two other control points determine the shape and curvature of the segment. Construction, representation, and chaining of Bézier curves will be further described in the chapter 5.

Chapter 4
Our approach

The course of work on the project could be divided into several steps. First of all, we got acquainted with the current state of the creative process. That is, to understand what actions and in what order are they performed by the artist we cooperate with. Subsequently, we got acquainted with the existing image filtering and styling methods. On the basis of these findings, we selected several approaches that we implemented in this project.

We decided to proceed with the work following how the artist himself proceeds in his creative process. In the beginning, we focused on the initial processing and disruption of the input photo. We were looking for methods that preserve the shape and content of the original image and significantly reduce the amount of unnecessary detail that the picture contains.

4.1 Image filtration

Image filtering using the Kuwahara filter and its modification according to the research of Jan Eric Kyprianidis became a relatively clear choice. At first, we still considered the possibility of using bilateral filtering described in [TM98] or the mean shift filtering [CM02] for the first step of image processing. These filtering methods have in common that they preserve the edges in the image to some extent, while smooth areas are filtered, and thus blurred. Unlike the Kuwahara filter, we did not find any documented procedure in the literature that would take into account other shapes and flows in the image beside the edges themselves. This means that these two algorithms would be suitable for the initial image disruption, but it would be hard to create any follow-up step to make the underpainting from this output. As a result, we decided to use the Kuwahara filter.

4.1.1 Kuwahara filter

We proceeded from the simplest forms of algorithms, which we continuously modified and improved with various extensions. As the first and also the most straightforward algorithm, we used filtering using the standard Kuwahara filter [KHEK76]. For each pixel of the image function $I(x, y)$, this filter works with the square neighborhood of the point (x, y) of size $2r + 1$, where r is the filter size. The filter window itself is divided into four parts Q_i, which are defined for position (x, y) as:

$$Q_i(x,y) = \begin{cases} [x, x+r] \times [y, y+r], & if \quad i = 0 \\ [x-r, x] \times [y, y+r], & if \quad i = 1 \\ [x, x+r] \times [y-r, y], & if \quad i = 2 \\ [x-r, x] \times [y-r, y], & if \quad i = 3 \end{cases} \tag{4.1}$$

The arithmetic mean of the brightness of the pixels $m_i(x, y)$ and the standard deviation $\sigma_i(x, y)$ are then calculated for each part Q_i. The resulting value of the pixel $I'(x, y)$ is equal to $m_i(x, y)$, where $i = argmin_j \sigma_j(x, y)$.

4.1.2 Anisotropic Kuwahara filter

The anisotropic Kuwahara filter is an expansion of the standard filter, which is described in [KKD09]. First of all, this filter does not have anything to do with anisotropic filtering, which we might otherwise know from computer graphics. The word isotropic can be paraphrased as uniform in all directions [Dic12]. The prefix "an" denotes the opposite, meaning that anisotropic filtering is a type of filtering that will deform at each point and thus locally adapt to the structure of the filtered image.

In the article Image and Video Abstraction by Anisotropic Kuwahara Filtering [KKD09], Jan Eric Kyprianidis proposes an extension of the version of the Kuwahara filter, which is described in [PPC07]. The primary difference of this approach is the use of a circular filter instead of a square filter. The simple circular filter is further extended by the possible deformation of this circle into an ellipse and rotation of it based on the structure of the image. Another difference is that not only the average colour of the part with the lowest standard deviation is used to calculate the resulting pixel colour. But it also uses the weighted sum of several parts that make up the individual sections of the filter. The filter scheme of the individual methods can be seen in the picture 4.1.

The discussion is now going to revolve around the algorithm itself. Since we need to obtain data according to which we will later transform the filter, it is necessary to calculate the structure tensor, the computation of which is described in 3.2.6. The aim is to preserve the basic shapes of the objects in the image. Therefore, we want to filter the image mainly in the direction of the smallest change in the image, as explained in 3.2.5. The vector t of the orientation of this minimal change is obtained from the structure tensor as:

$$t = \begin{pmatrix} \lambda_1 - E \\ -F \end{pmatrix} \tag{4.2}$$

where E and F correspond to the respective elements of the structure tensor as defined in 3.2.6. Finally, we compute the anisotropy A as:

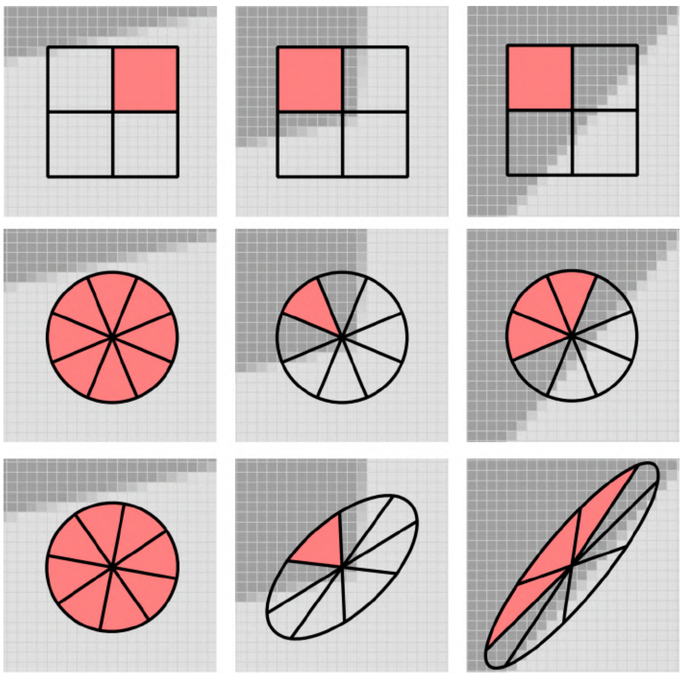

Figure 4.1: Comparison of filter shapes for individual methods. Gradually standard Kuwahara filter, circular Kuwahara filter [PPC07], anisotropic Kuwahara filter[KKD09]

$$A = \frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2} \tag{4.3}$$

The result is in the range $\langle 0; 1\rangle$, where 0 corresponds to a fully isotropic region and 1 to an entirely anisotropic one. Based on the parameters calculated in this way, we can calculate the filter transformation for each pixel. First of all, it is necessary to define the transformation of the circular filter. Let us have the local orientation ϕ and the anisotropy A at the point (x, y). Then the parameters of the eccentricity of the elliptical filter are obtained by the relation:

$$a = \frac{\alpha + A}{\alpha} r \quad b = \frac{\alpha}{\alpha + A} r \tag{4.4}$$

The values of a, b correspond to the lengths of the semi-major and semi-minor axis of the ellipse. From this, we can create matrices S and R, which describe the shape and rotation of the filter.

$$S = \begin{pmatrix} a^{-1} & 0 \\ 0 & b^{-1} \end{pmatrix} \quad R = \begin{pmatrix} cos(\phi) & -sin(\phi) \\ sin(\phi) & cos(\phi) \end{pmatrix} \tag{4.5}$$

Next, we need to define the division of a circular filter into N parts:

$$\chi_i(X) = \begin{cases} 1 & \frac{(2\pi-1)\pi}{N} < arg(x) \leq \frac{(2\pi+1)\pi}{N} \\ 0 & otherwise \end{cases}, i = 0, ..., N-1 \tag{4.6}$$

The parts thus obtained have sharp edges, which is undesirable in this case. Therefore, beyond this division, a smoothed weight function K_i, which is defined by the relation $K_i = (\chi_i \star G_{\sigma_s}).G_{\sigma_r}$. Here G_σ is the Gaussian filter with which the convolution is first performed, and then a multiplication by this filter is executed next. Because $\chi_i = \chi_0 \circ R_{-2\pi i/N}$, so $K_i = K_0 \circ R_{-2\pi i/N}$, where $\circ$ corresponds to the composition of the functions. Next, we define the weight functions w_i:

$$K_0 \circ R_{-2\pi i/N} S R_{-\phi} \tag{4.7}$$

Let us also have the weighted averages of individual sectors:

$$m_i = \frac{1}{k} \int f(x) w_i(x - x_0) dx \tag{4.8}$$

and squares of individual standard deviations:

$$s_i^2 = \frac{1}{k} \int f^2(x) w_i(x - x_0) dx - m_i^2 \tag{4.9}$$

where $k = \int w_i(x - x_0) dx$ is the normalization factor. The resulting filter output is defined as:

$$\frac{\sum_i \alpha_i m_i}{\sum i \alpha_i} \tag{4.10}$$

where α_i is defined as:

$$\alpha_i = \frac{1}{1 + ||s_i||^q}. \tag{4.11}$$

4.1.3 Multi-scale anisotropic Kuwahara filter

Another implemented algorithm is an improvement of the anisotropic Kuwahara filter using the image pyramid, which is described in [Kyp11]. This enhancement addresses one of the most significant shortcomings of the previous approach in actual use. Although it was possible to set the size of the filter used, and thus the degree of filtering, this value remains static for the entire image. The improved method is designed to use an image pyramid that effectively enables the algorithm to use a filter of different sizes depending on the nature of the image part.

As we mentioned above, this approach uses the so-called image pyramid first described in [BA83]. It is a structure composed of several successively filtered and scaled-down images. The process of its creation is explained in [Sch92]. The filtration itself is essentially the same as for the anisotropic filter. The difference is that the algorithm gradually filters the images from the smallest to the largest. That is, from the highest floor of the image pyramid to the lowest. After the filtration on a k+1 floor is finished, the resulting image is combined with the input image on the k-th floor of the pyramid in the manner described below.

For each pyramid floor, the calculation begins by obtaining the structure tensor, followed by the Kuwahara filtration itself. The structure tensor is then resampled and doubled in resolution. In the next level of the pyramid, a new structure tensor is then calculated, which is merged with the structure tensor from the previous level using the relation:

$$\widetilde{J}_p^k = \alpha^k J_p^k + (1 - \alpha^k)\overline{J}_p^{k+1} \tag{4.12}$$

where $\overline{J}^k$ is a merged tensor, J^k is a tensor that is calculated at the level k and J^{k+1} is an upsampled tensor from the previous level. Blending coefficient α is at a pyramid level k calculated for each pixel as:

$$\alpha^k = \frac{A^k}{A^k + \overline{A}^{k+1}} \tag{4.13}$$

where A^k is the anisotropy at the current level, and $\overline{A}^{k+1}$ is upsampled anisotropy from the previous pyramid level. We can see from this relationship that it gives more weight to the structure tensor with a higher corresponding value of anisotropy at a given location.

Similar to the estimation of local orientation, the filtering is performed gradually from the highest floor of the pyramid to the lowest one. On the upper floor, filtering is performed by default on the image stored in the pyramid. At lower levels, the process is similar to the case of structure tensors. Thus, the input image is mixed with the scaled output of the anisotropic Kuwahara filter one level above. The relationship then defines mixing as:

$$\widetilde{f}^k = \beta^k f^k + (1 - \beta^k)\overline{f}^{k+1} \tag{4.14}$$

Here, $\widetilde{f}^k$ denotes the merged image, f^k is the input from image pyramid and $\overline{f}^{k+1}$ is a resampled output from $k+1$ level of the pyramid. Blending coefficient β^k is computed from relation:

$$\beta^k = clamp(s^k.p_s(p_d)^k - \tau_v, 0, 1) \tag{4.15}$$

The numbers p_s, p_d and τ_v are the mixing parameters and according to the article [Kyp11] are set by default at 0.5, 1.25 and 0.1. The combined image is then used to calculate the structure tensor for the k-th level of the pyramid, and it is also being filtered. The calculation of the standard deviation s^k from the definition is relatively computationally demanding, therefore, its approximation is used, given by the relation:

$$s_{max} = \sum_{i=0}^{N-1} max(\tau_w, ||s_i||). \tag{4.16}$$

4.2 Image segmentation

Image segmentation can be used under different circumstances, and as mentioned in 2.3, there are several different approaches to it. In this project, the artist may use segmentation to divide an image into regions, where he may edit each one of them afterwards separately. This use case creates three main criteria that need to be fulfilled. First, some control over the granularity of the segmentation is required, which means how big or small the resulting segments will be. Next, since each region will be edited in one fashion, it must be somewhat visually coherent. Lastly, it would benefit the rest of the process if the algorithm could return the segments themselves as separate objects or images, or at least a mask for each segment.

This part of the project was more of a proof of concept or experiment if we could find a method that would be possible to seamlessly incorporate into the current workflow or that would bring substantial improvement compared to the current approach. Since Adobe Photoshop already has numerous tools for image segmentation, it would be necessary to find a solution that would work differently or give somewhat specific or exciting results.

One of those solutions that provide fairly specific results is based on the article Spectral Matting [LRAL07]. As the name suggests, this method combines two methods: spectral analysis and digital matting. Digital matting is a method of segmenting an object from a background. Formally, this algorithm assumes that each image pixel I_i is a linear combination of a foreground colour F_i and background colour B_i:

$$I_i = \alpha_i F_i + (1 - \alpha_i) B_i \tag{4.17}$$

In the Spectral Matting article, a change in this approach is presented. Instead of just background and foreground, it is assumed that there is N layers $F^1, ..., F^N$:

$$I_i = \sum_{n=1}^{N} \alpha_i^k F_i^k \tag{4.18}$$

where the α^k denotes the contribution of a pixel to a given layer.

The spectral matting algorithm consists of several main steps. The first one is a computation of the Laplacian matrix. Its smallest eigenvectors are then used to initialize the k-means algorithm, which creates the initial matting components. The k-means algorithm was initially a cluster identification algorithm used in data visualization or computer vision, but it can also be used to find clusters in images and thus its segmentation. In image 4.2 we can see image segmentation using k-means.

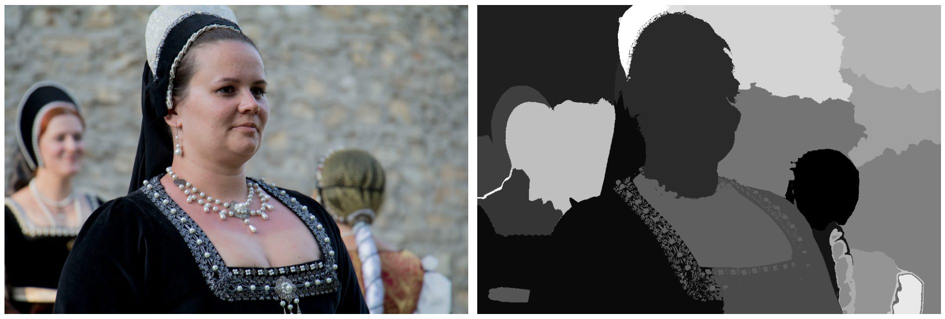

Figure 4.2: Comparison of source image and the resulting clusters after application of k-means algorithm

Based on this, it is finally possible to do the matting. Examples of matted components can be seen in figure 4.3. The final step is to group the matting components in the foreground and background. This can be done in two manners, supervised and unsupervised. Supervised grouping requires user input, which by a stroke determines which part is the background and which part is the foreground. Otherwise, the algorithm itself groups the component into the groups above automatically. The result of the unsupervised grouping can be seen in figure 4.4.

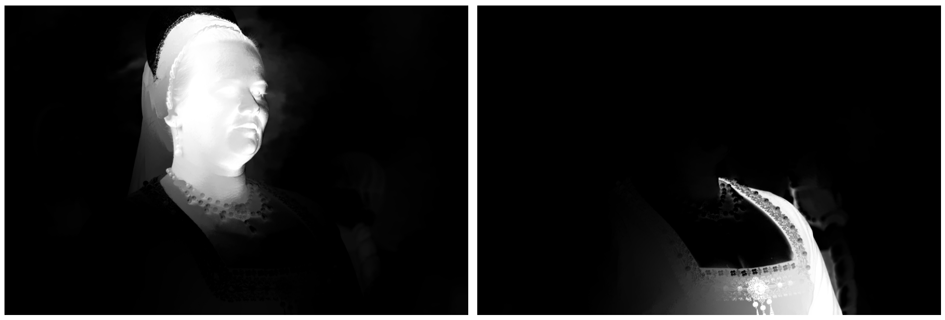

Figure 4.3: Example of matting components

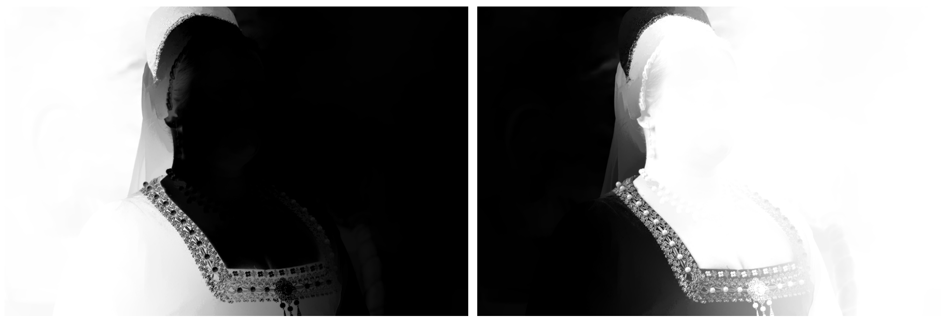

Figure 4.4: Final output of spectral matting algorithm with components merged into a foreground and a background

4.3 Path creation

The last part of our approach is the generation of paths. The paths in this case will be the foundation on which the brush strokes will be based. As described in part 3.2.7, different types of curves serve slightly different purposes. In this project, we selected the Bézier curves based on their prevalence in all computer graphics. They are supported in many programs, and there is a huge amount of information on how to work with them.

In order get the final path creation algorithm, the thought process starts from the very end of the process an ends at the beginning of it. Since implementing a brushstroke renderer felt redundant, we decided to use the artist's older scripts and actions in Photoshop, which can automatically do the rendering for us. The auto-stroking scripts need to be fed with the input image and a set of paths along which it will paint the strokes. The problem is that there is no option to import these paths directly into the Photoshop. The only theoretical option discovered is based on the fact that Photoshop does include the paths directly into a processed image, but that does happen only if the image is in JPEG format. Unfortunately, no documentation was found on this feature, so this approach was a dead end.

Based on our findings, we decided to use Adobe Illustrator as an intermediary. This approach requires creating an SVG file containing the paths that we can copy directly from Illustrator to Photoshop. When we had this part sorted out, it was clear that the type of curve to be used would be the Bézier curves because it is the only type directly supported by the SVG format.

Path creation is somewhat similar to visualizing a fluid flow using stream objects. The most straightforward approach is to create a regular grid of so-called seed points, which are in the image 4.5, shown in the form of small circles. From each of these seeds, a curve is "grown". We can image the curve as a path that the object would take if we were to drop it into a flow field like this.

From the image, we can see that this approach is far from perfect from the visualization point of view. That is, for one reason, which is the changing density of streamlines throughout the visualization, i.e., in some areas, we do not have any streamlines. In contrast, there may be multiple lines along a similar path in other areas.

There are techniques that try to achieve a uniform distribution of the streamlines in the whole area. A group of these approaches is called Farthest point streamline seeding. These are iterative methods that always try to create a new path seed in a position that is farthest away from all previously created paths. The result of this method can be seen in the image 4.6.

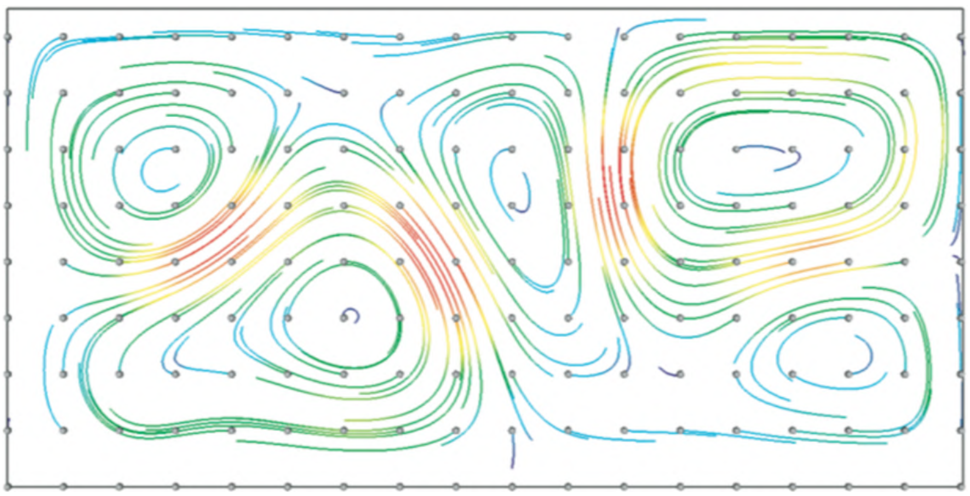
Figure 4.5: Visualization of Streamlines

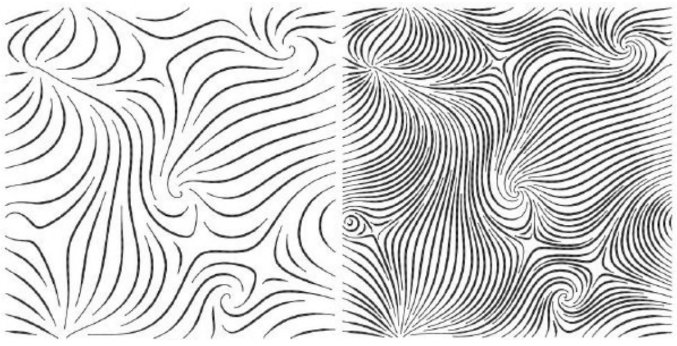
Figure 4.6: Farthest point streamline seeding

Chapter 5
Implementation

For implementation, we chose the C++ programming language and several libraries that made it easier for us to work with images. The first, widely used in implementation, is The CImg Library [Tsc04]. This single-header library contains a fairly extensive set of tools for representing and manipulating various image data formats. We mainly used the library interface to load, save, and represent the image. We also used the image resampling function. We wrote most of the other functionality ourselves on the basis of the literature. We used libpng [Roe00], jpeglib [aut96] and zlib [GR96] libraries to load and save the respective image formats. We also used the linalg.h [ODPJ20] library, which offers a simple and efficient representation of vectors, matrices, and basic arithmetic operations on them. As we progressed towards the GPU implementation, we also used the stb image library for easy image loading, saving, and resizing.

We can divide the implementation process into few parts. We started with image filtration methods based on the Kuwahara filter. These were initially implemented only for the CPU without any parallelization and code optimization. Next, we started optimizing the code, which was further optimized with a GPU implementation using GLSL shaders. This implementation is partially based on the source codes from [KKD09] and [KD08]. Next, we took a detour and experimented with image segmentation and its possible use in the creative process for a while but that did not result in any image segmentation methods being implemented. Instead, we added support for the filtration of semitransparent images. The last major part of this process was the generation of stroke paths. Throughout the implementation of the main features, we also changed the user interface. We added the option of batch processing with either more files or by applying multiple edits on a single file or even by combining these two together.

5.1 Image filtering

5.1.1 Kuwahara filter

We started the project by implementing a standard Kuwahara filter, which is a very simple and relatively straightforward algorithm to implement. The only part that is somewhat nonstandard is the calculation of the average deviation for each section. It is not calculated directly from the colour values of each pixel, but rather from the parameter named "value" that is obtained by converting the input RGB image to the HSV colour profile. We primarily used this implementation to prepare the entire project and test libraries as CImg, which we have not used before.

5.1.2 Anisotropic Kuwahara filter

Kuwahara filter was followed by the implementation of the anisotropic Kuwahara filter according to [KKD09]. There are also source codes for this article. However, they are written to run on a graphics card. We decided at first to reimplement it into standard C++ running only on the processor. The reason for this decision is an easier debugging of the code, which contributes to a better understanding of its exact functionality. This understanding is key to the ability to make further modifications and extend the algorithm. Last but not least, this approach allows for better performance comparisons with algorithms that are written by default for the computer processor only.

Reimplementation itself seemed like a relatively simple task at first. First of all, it was necessary to solve how to appropriately and effectively represent vectors and matrices that are widely used in the OpenGL Shading Language, or GLSL for short. The linalg library, which uses a notation similar to that of GLSL, helped us immensely. Another problem we encountered was the inability to directly debug source code from the article, which was running on a graphics card. Combined with our minimal experience with GLSL programming, countless hours have been spent researching the documentation and image output of the individual GLSL shaders.

Most of the other implementation problems were due to ambiguities in the representation of the edited image. By default, each pixel of a colour image is represented by three colour components, each with an 8-bit space. Therefore, each colour is represented by a triplet of values in the range $\langle 0; 255\rangle$. However, for some operations performed on the image, it is necessary to remap these values to the interval $\langle 0.0; 1.0\rangle$.

Another fact that we discovered after working on the project for some time comes from the CImg library itself. Its data structures are designed to store image data, where the nature of the colour representation implies that no matter which data type is used, all stored values will be clamped to be greater than zero.

Because we have been using only the graphical output to debug and compare the GPU implementation with the CPU implementation, this approach resulted in a situation where the output of both methods yielded somewhat similar partial results. This was caused by GLSL, which, when saving the image, also clamps the values in range $\langle 0; 255\rangle$, even though it internally includes negative values in its calculation process. As a result, the outputs of both approaches look the same, but their actual representation is not the same, which will be reflected in further calculations. Therefore, since CImg cannot store negative values, we were forced to store most of the intermediate calculations in the data structures of the standard C++ library, mainly std::vector. Vectors are as simple to work with as an image in CImg, but the problem is their speed, which is hindered by a different representation in computer memory and data access.

We implemented both versions of the filter division. The difference between them is the division of the circular filter where they divide into either four or eight parts. The difference is that the eight-part division, because of its higher granularity, can better adapt to relatively minor changes corresponding to small structures in the image. This produces images with a higher detail level that seem sharper at first glance.

Next, we will describe a few differences between the definition of the algorithm and our implementation. First, there are the equations 4.8 and 4.9, which are defined for a continuous representation of an image function. In the implementation, the integral is replaced with a sum of pixels. Another change compared to the theoretical definition of the algorithm is the need to use a smoothed array of structure tensors, which is achieved by a convolution of an array of structure tensors with a Gaussian filter. In this implementation, we used the $\sigma = 2$. Without this step, the field may contain discontinuities, creating artifacts in the image.

Furthermore, we deviated from the literature when choosing the size of the filter. It is recommended to leave the original size of six, or, gradually apply the filter in several passes. However, we opted for a user-customizable filter size. We tested this adaptation and no visual artifacts or other side effects of this change were observed. The main difference remained in the increase in computation time with increasing filter size.

5.1.3 Multi-scale anisotropic Kuwahara filter

Subsequently, we implemented a multi-scale version of the anisotropic Kuwahara filter. First, it was necessary to implement the creation of an image pyramid. We created it by gradually filtering the image using the Lancoz3 filter recommended in the article [Kyp11]. Compared to the standard Gaussian or box filter, this filter should better maintain low frequencies in the image and filter out mainly high frequencies.

The filtering itself takes place, except for one modification, in the same way as with the anisotropic Kuwahara filter, which is described in section 4.1.2. In the previous method, the Sobel filter was used to calculate the derivatives, which was replaced by the filter developed by Jähne et al. [JSK+99]. In this particular implementation, the filter constant $p_1 = 0.183$ was chosen. We described both of these methods for calculating partial derivatives in the section 3.2.2.

Furthermore, it was necessary to implement methods for merging images and fields of structure tensors according to the relationship 4.14 and 4.12. For the first equation, it is still necessary to define the mixing parameters p_s, p_d and τ_v. The recommended values of these parameters are according to the article $0.5, 1.25, 0.1$. In the end, we also left choice of these values to the user.

One of the adjustments that have been implemented is the colour threshold. This feature adjusts the algorithm's behavior in areas where a colour gradient occurs. By default, this gradient is maintained. However, if this function is enabled, the gradient colours are quantized. This means that a particular part of the gradient colour spectrum is always represented by one colour. This then creates sharp transitions in the gradient between its unique colours. This feature can be turned on by the user or turned off when the program starts.

5.2 Image segmentation

Image segmentation was more of a proof of concept whether it is even possible to propose a method that would be superior to the methods and options built into Adobe Photoshop. In testing of this method, we used a Matlab implementation from the Spectral Matting article [LRAL07]. It is an easy-to-use program where the user can define the input image with a few parameters. The program then outputs a k-means segmentation, individual matting components, where each is in a separate file, and last but not least, merged matting components.

5.3 Semitransparent images

The stylization of semitransparent images became one of the critical features that allowed image filtration to be used throughout most of the creative process. As mentioned above, digital painting usually consists of numerous different layers of images combined. So, in order to be able to edit just one layer of the image, we need to calculate with transparency.

The simplest way would be to process the RGBA image as a standard RGB image and add the alpha layer at the end. This approach has two significant drawbacks. Firstly, since pixels with $\alpha = 0$ are saved as black, a situation occurred when the black colour of these pixels was mixed in the resulting pixels which had $\alpha > 0$, resulting in partially black silhouettes of the resulting images. The second drawback was that the silhouette, based on the α channel, obviously did not change with filtration, which resulted in not very good looking images.

There are two solutions to the problem aforementioned. The first was to change most of the existing code and add support for the α channel. This seemed like a non-trivial task and, in the GPU implementation, the fourth channel was already used to store other information. The second and more elegant solution, which worked very well, was again splitting the α channel from the colour channels. This results in α being also processsed in the same way as RGB, therefore changing its values and thus the transparency of the resulting image. In figure 5.1, we can see the input and output image, and in figure 5.2, an original alpha layer and the processed one can be seen.

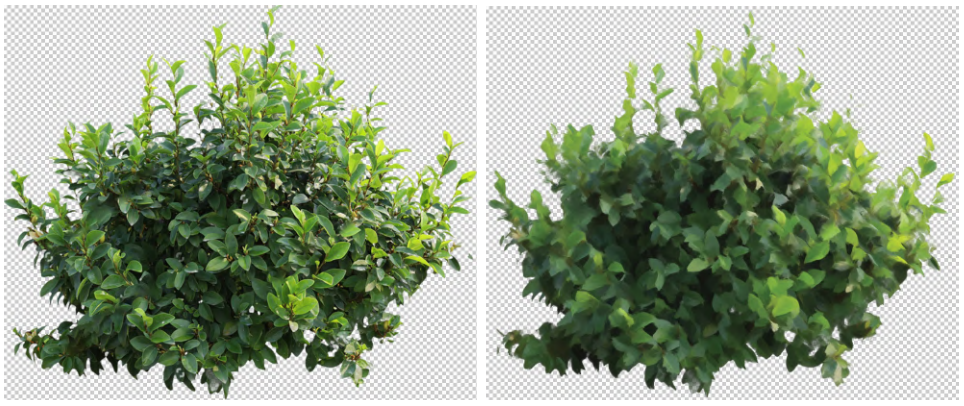

Figure 5.1: Comparison of semitransparent image before and after filtration

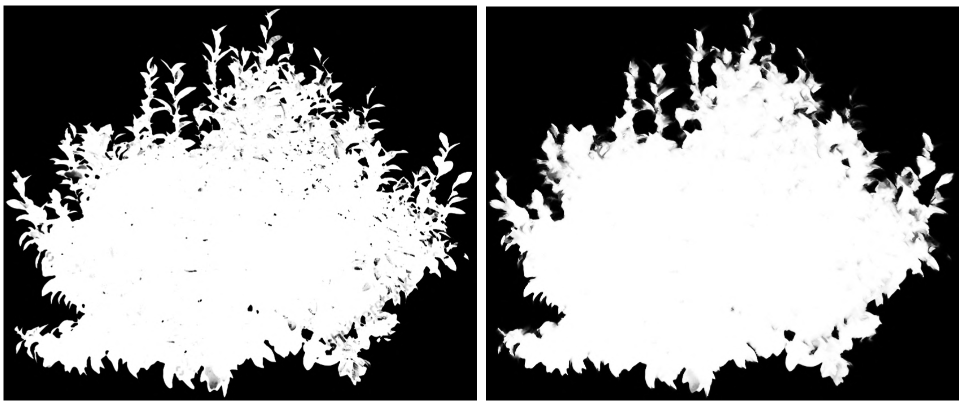

Figure 5.2: Comparison of alpha channel of image before and after filtration

5.4 Path creation

Path creation was the last major part of this project to implement. As described in chapter 4, it is an algorithm that can generate paths, which are then saved in an SVG file. The content of this file can then be imported into Adobe Photoshop through Adobe Illustrator. In Photoshop, the artist already has a set of tools that can automatically stroke the image based on the imported paths.

The computation process starts similarly to the case of the anisotropic Kuwahara filtering process, which was explained in parts 4.1.2 and 3.2. In short, the process starts with the partial derivatives being computed and then the local orientation is calculated in each pixel of the image. Next, the infill of the structure tensor is performed, which is described in 2.1. For this step, border conditions based on anisotropy also need to be set, i.e. pixels whose value remains the same throughout the infilling.

From this, it is possible to generate the paths. This task can be divided into two steps: Path creation and curve fitting. Focusing on the path creation, it consists of creating a seed from which it is walked along the generated orientation, thus creating a path. The movement itself is based on a first-order Euler method, which is defined as

$$x(t + dt) = x(t) + v(x(t))dt. \tag{5.1}$$

In other words, this equation says that to obtain the position of an object in $t + dt$, we need to move from the position in time t in the direction of vector v. If we return to our application, the easiest version of this approach would be the simplification that the step size is equal to the size of one pixel. This means that we can continue to one of the given pixel's neighbours in each position, meaning that we have eight different directions to choose from. As can be seen in the right half of the image 5.3 this approach resulted in more of a polyline than a natural-looking stroke path.

To combat this, we developed an extension that increases the step to two pixels, which gives the option to choose from 16 different directions. This upgrade resulted in a significant improvement in the shape of the generated paths. Therefore, these paths are much closer to representing the actual shapes of brush strokes.

Density mask

An important feature that was added to enhance the art-directability of this procedure is something we call density masking. As shown in part 4.3, the standard approach would be to plant the seeds for the paths uniformly throughout the area of an image. As a request from the artist, we added an option to use a grayscale image, which would determine the density of paths

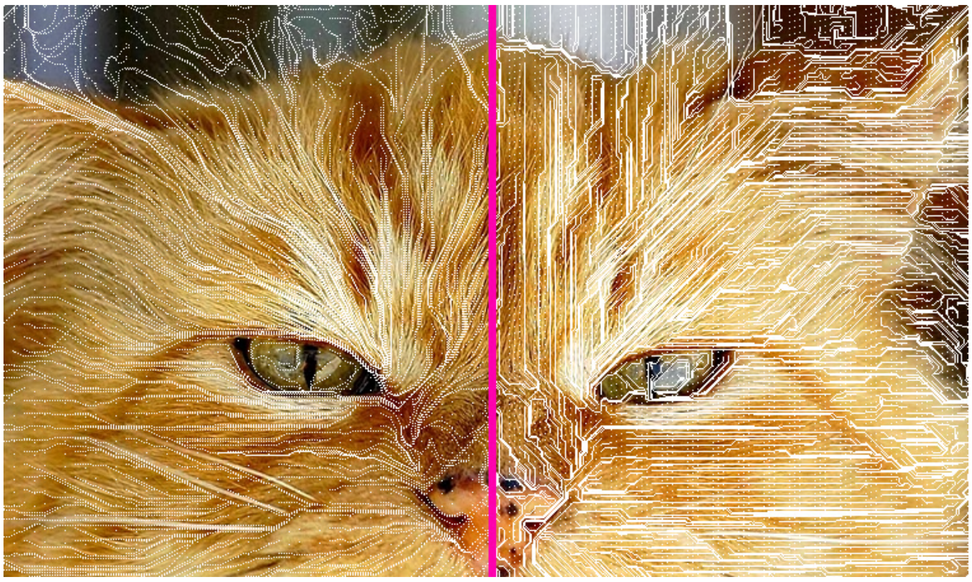

Figure 5.3: Comparison of of path creation using step size equal two (left) and one (right)

in a given part of an image. More precisely, it determines the perceptual decrease of the number of curves in a given area. The interesting part of the implementation was determining whether the curve would spawn or not. we ended up using a probability based on the mask. When the mask is white, that is, has a value of 255, the path will start at that point, and if the value is 0, the path will surely not begin there. First, we tried a linear dependence of the mask value on the probability. This resulted in a distribution of paths in which the change in density was just not perceptually very noticeable. After further testing, we used a cubic dependency where for the value of mask 128, the resulting probability of spawning a curve is just 0,125. This result is then combined with the likelihood of spawning based on the user-defined spacing parameter.

5.5 Vector creation

After creating the paths, it is needed to create vectors based on those. The easiest way would be to create a polyline that gradually passes through all the points on every path. Although the polyline is far from the optimal solution for capturing the brushstroke, it might be possible to get away with this approach because the control points are so close together. However, this approach would be suboptimal based on the sheer number of control points. As was indicated in previous parts of this work, the optimal solution for path description are curves, especially Bézier curves. Each segment of a Bézier curve is defined by two endpoints through which the curve passes and two control points, which define its curvature. There are many algorithms capable of fitting a curve onto a previously generated set of points. In this work, we opted for a reasonably simple solution which does not guarantee a perfectly fitted curve, but does not require nearly any additional computations. Regarding the precision of the curve fitting, it might even be beneficial in this application that this would again slightly distort the original image.

The curve approximation is based on the definition of the Bézier curve. Since four points unequivocally define the curve segment, it is possible to obtain the position of control points if we know the position of the four points through which the curve passes. In practice, we need two of the points to be the end points of the curve segment, and the other two are in 1/3 and 2/3 of the segment. Since the points we obtain from path creation are nearly equidistantly spaced along the curve, picking the four needed points for the computation is a trivial task. Below, we can see the pseudocode snipped for calculation of the control points.

```
var calculateControlPoints(P1, P2, P3, P4){
    var b1 = P2.y - P1.y * (1-1/3)^3 - P4 /27.0;
    var b2 = P3.y - P1.y * (1-2/3)^3 - (2/3)^3 * P4.y

    C1.y = (-2 * b1 + b2) / -0.6666666666666
    C2.y = (b2 - 0.22222222 * C1.y) / 0.444444444444;

    C1.x = P1.x + (P1.x - P4.x)/3
    C2.x = P4.x - (P1.x - P4.x)/3

    return C1,C2
}
```

In figure 5.4 below, we can see a simple Bézier curve with highlighted endpoints and connected control points. The curve is made up of two connected segments. Therefore, there are three control points through which the curve passes and four other control points. The first segment needs to be defined by convention by four points. Interestingly, only two additional pairs of coordinates are required to determine the second segment and any further curve segments.

Regarding the structure of an SVG file, it is necessary to define the image dimensions and the XML namespace. Inside the `<path>` tag, `M` defines the starting coordinates, and after `C`, the following three control points of the initial curve segment are defined. The next curve segments are then the previous one appended by `S` and the definition of the third and fourth control points. Lastly, a stroke and a fill of the curve are defined.

```
<svg width="190" height="160"
xmlns="http://www.w3.org/2000/svg">
  <path d="M 10 80 C 40 10, 65 10, 95 80 S 150 150, 180 80"
  stroke="black" fill="transparent"/>
</svg>
```

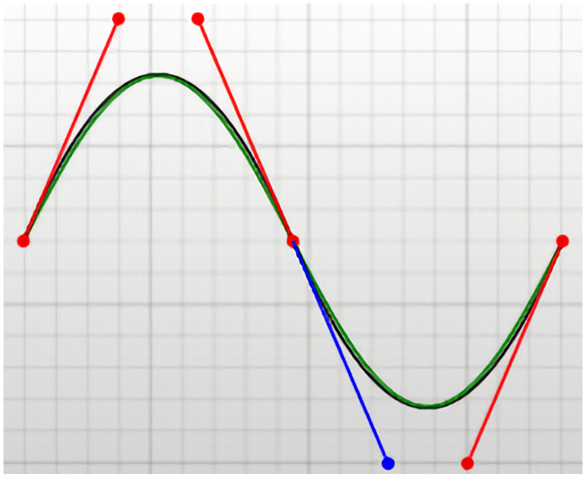

Figure 5.4: Bézier curve with its control points visualization

5.6 Graphics card implementation

Crucial for this project's real-life usage is the implementation running on a graphics card. This is mainly due to the enormous speed-up which was achieved by offloading parallelizable operations from CPU onto GPU. Implementation was done in OpenGL Shading Language (GLSL) mainly in fragment shaders while using OpenGL Extension Wrangler Library (GLEW) [SIM$^+$17]. In the implementation, we inspired ourselves with code distributed alongside the articles on which we based this project on.

Graphics card was used in this project to compute the partial derivatives, structure tensor, structure tensor filling, conversion of the RGB image to CIE Lab, Gaussian blurring and image ant tensor merging, which is used in the multi-scale Kuwahara filtering. The only part of the processing that is not written in shaders is the vector generation after the flow field is created. We will discuss performance and speed improvement in chapter 6.

5.7 User interface

The application was, from the beginning, planned as a console application, which would work in a fairly similar manner. The user would select the image to be edited, select the editing method, and tweak some filter parameters. This fact changed completely by the end of this project that it is now possible to batch edit multiple images and, on each of them, perform multiple edits as well as saving each of the edited results using a naming scheme, which would incorporate all the options and parameters with which the image was edited.

The program in its current state uses some platform-specific functions for file handling. Thus, it has been properly tested, and the full functionality is supported only on the Windows operating system. The program is currently available as an .msi installer which contains all the necessary files for the program. The GPU implementation was tested on integrated graphics cards in Intel processors and dedicated Nvidia graphics cards.

The program is executable by dragging one or more images onto an executable.exe file. In the case of the CPU implementation, only JPEG and PNG formats are supported, but in the case of the GPU implementation, image loading is handled by the stb library. Thus, the limitation of file format lies within this library. Subsequently, as the application starts, the user is asked to select the editing method. There is a selection of nine algorithms: standard Kuwahara filter, anisotropic Kuwahara filter N = 4, anisotropic Kuwahara filter N = 8, anisotropic Kuwahara filter, anisotropic Kuwahara filter with thresholding. Depending on the selected filter, it is possible to define the parameters with which it will run. This procedure is typically required for every loaded file.

Entering the same parameters can become very tedious, which led to the option of defining the operations in a separate file. The structure of this file is simple. On each line, one operation can be defined. The first is the number of operation that will be performed, followed by a set of numbers that are the operation parameters. In lists and tables 5.1 are shown and explained all possible procedures and their respective parameters.

List of all implemented operations

- **Kuwahara filter** - standard Kuwahara filter
- **Adaptive Kuwahara filter** - Kuwahara filter that changes its form based on local orientation
    - **n=4** - Adaptive Kuwahara filter with filter divided into four parts
    - **n=8** - Adaptive Kuwahara filter with filter divided into eight parts
- **Adaptive multi-scale Kuwahara filter** - Adaptive Kuwahara filter
    - **Original Gauss** - Standard Gauss filter used for partial derivatives computation
    - **Original Jähne** - Standard filter by Jähne et al.
    - **Modified Jähne** - Modified version of the filter by Jähne et al. described in 5.7
- **Vector generator** - Flow-based algorithm that generates an svg with a set of curves representing brush strokes.

operation type	filter size	ps,pd,tv	tresholding	derivatives	path setting
Kuwahara filter	yes	-	-	Gauss	-
Anisotropic Kuwahara filter	yes	-	-	Gauss	-
Multi-scale anisotropic Kuwahara filter	yes	yes	yes	Gauss/ Jahne	-
Vector generator	-	-	-	Gauss	yes

Table 5.1: Comparison of possible parameters between operation types

List of all parameters and their explanation

- **Filter size** - Defines the size of the Kuwahara filter or the default filter size in the case of the multi-scale Kuwahara filtering. Determines to what degree the image is filtrated.
- **ps,pd,tv** - Parameters that determine the nature of image merging in the multi-scale Kuwahara filtering method. The merging is done using an equation: $\beta^k = clamp(s^k.p_s(p_d)^k - \tau_v, 0, 1)$, where k is the level of the pyramid and s denoted the standard deviation.
- **Thresholding** - Can be turned on or off, its effect is described in 6.1.2.
- **Derivatives method** - Switch between the Gauss and Jähne filter for partial derivatives computation.
- **Path settings**
  - **Path spacing** - Parameter determining the base density of generated paths.
  - **Path stepping** - Determines the length in pixels of the resulting curve segments.
  - **Minimal, Maximal length** - Limits the minimal and maximal length of the resulting curves.
  - **Color tolerance** - Threshold value that determines over how big of a colour difference a single stroke can continue. The colour difference is measured in the CIE Lab color space.
  - **Normalized image** - When this feature is activated, the path creation algorithm is performed on a scaled image; its longer side is set to 2048 px. Production feature used to bring consistency to images of different sizes.

In the file parametry.txt, it is possible to define with which various settings the image filtering should be performed. The output image is then stored at the exact location where the input image is stored. The parameters with which the image was filtered are then automatically added to the file name.

Our change of the Jahne filter

Convolution kernel by Jähne et al. is defined as:

$$D_x = \frac{1}{2}\begin{pmatrix} p_1 & 0 & -p1 \\ 1-2p_1 & 0 & 2p_1-1 \\ p_1 & 0 & -p_1 \end{pmatrix}, D_y = D_y^T,$$

where parameter $p_1 = 0.183$. Thus the partial derivatives are computed as:

$$\frac{\partial f}{\partial x} \approx D_x \star f \qquad \frac{\partial f}{\partial y} \approx D_y \star f.$$

By default, part of the computation would be $(1-2p_1) * imageValue$ and $(2p_1 - 1) * imageValue$, during the computation we accidentally removed the brackets thus based on the operation precedence we obtain $1 - (2p_1 * imageValue)$ and $2p_1 - (1 * imageValue)$ which completely changed the results. For example, in a mono-colour image with a value of 100, we would expect the partial derivation to equal 0. But in this edited case, the partial derivation equals -67,6. In the image below, we can see the difference that this change makes to the resulting image.

Chapter 6
Results and evaluation

Testing was carried on on three different levels. First, testing the functionality, meaning whether the filtering techniques work as expected, that is whether we were able to achieve similar outputs as described in the literature. Secondly, we evaluated the relative performance of algorithms to each other and the influence of different settings and scaling with input picture size. The last testing level and in this case the most crucial evaluated aspect is the suitability of the chosen methods and their outputs within the artist's creative process.

6.1 Functionality testing

In this part, we will evaluate the individual implemented methods and their modifications. For each of them, we will describe how much our results match the expected outputs, meaning if the achieved result seems in line with the article on which it was based. The first algorithm implemented is the original Kuwahara filter. It was implemented for testing purposes and to prepare the whole program structure, and it will not be used in practice in this basic form. Its output can be seen in the picture 6.1 below.

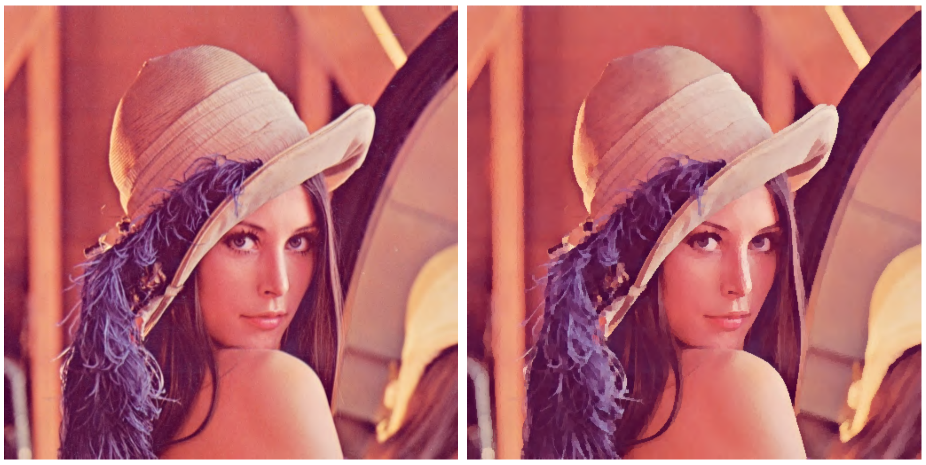

Figure 6.1: Comparison of the input photo and its filtering using a Kuwahara filter of size six. The resolution of the input image is 512x512px.

6.1.1 Anisotropic Kuwahara filter

The second implemented filter was the Anisotropic Kuwahara filter. In this case, we were able to achieve the same output quality as described in the article [KKD09]. The advantage over the previous filter is its ability to respect the shapes in the image. This is best seen in pictures that already contain a distinguishable locally oriented pattern in the original photo, such as animal fur, which can be seen in the figure 6.2.

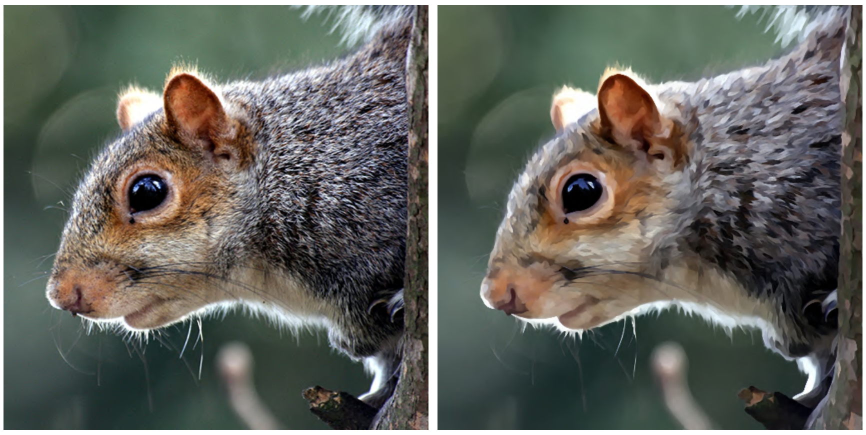

Figure 6.2: Example of animal fur stylization using anisotropic Kuwahara filter

Figure 6.3 shows a comparison of the outputs of the anisotropic Kuwahara filter divided into four and eight parts. We can observe that the first image is less sharp or clear in fine details, such as the eyes that become visually lost in the face. In contrast, the fluff on the hat looks more defined and, therefore, a little better in the first picture. This is a situation that we encountered pretty often in this project. There is no single solution to this problem, hence all the implemented algorithms have their role, and it gives a significant number of possibilities, and thus a control over the result to the artist.

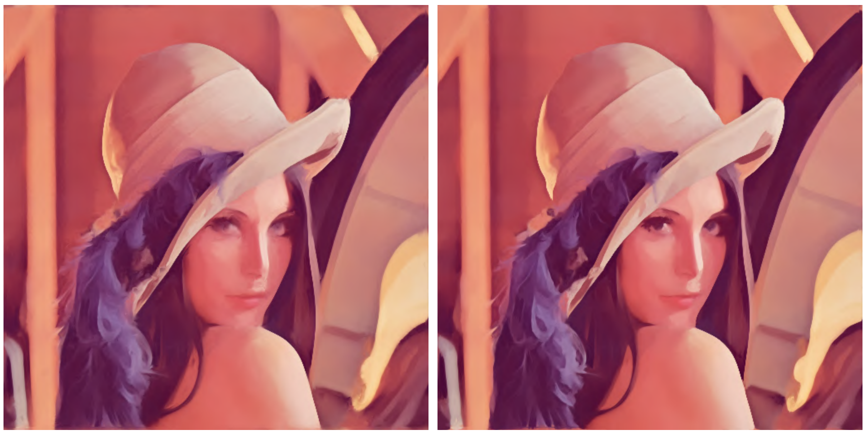

Figure 6.3: Comparison of filtration using an anisotropic Kuwahara filter divided into four and eight parts for filter size 12

6.1.2 Multi-scale anisotropic Kuwahara filter

For this filter, we relied only on its theoretical description in the literature [Kyp11] and a few sample images. With our artist, we tested different parameters to determine the mixing of the partial results. We decided on an approach where users can change them on per-image bases. We implemented three slightly different versions of this algorithm, which differ in the method of partial derivative computation. A comparison of the filter by Jähne et al. and its modified version can be seen in figure 6.4. It is visible that the result of the version from the article (on the left) is clearer and more defined. In contrast, the second image has much more disturbed contours, which look like the painter used an actual dry brush to roughen up the edges. Although the second image was created by mistake in the computation, it ended up being the preferred result by the artist. Also, in figure 6.5 a comparison is shown between filtering with and without thresholding.

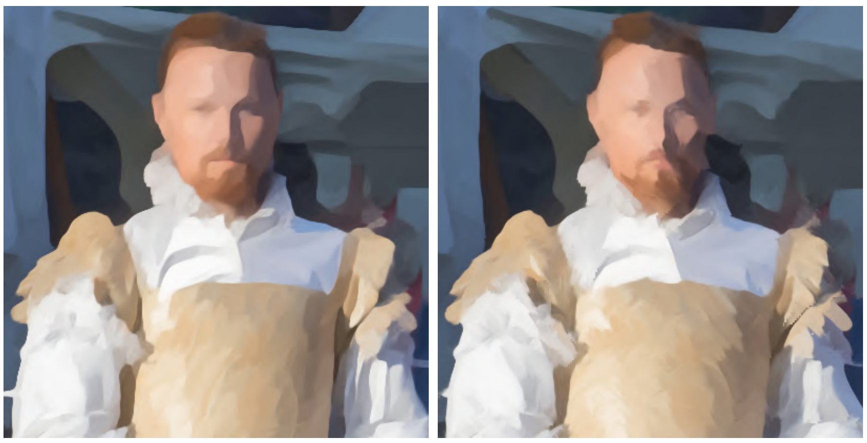

Figure 6.4: Comparison of filtration with multi-scale anisotropic Kuwahara filter using different method of derivatives estimation. In the left filter by Jähne et al. in the right its modified version described in 5.7

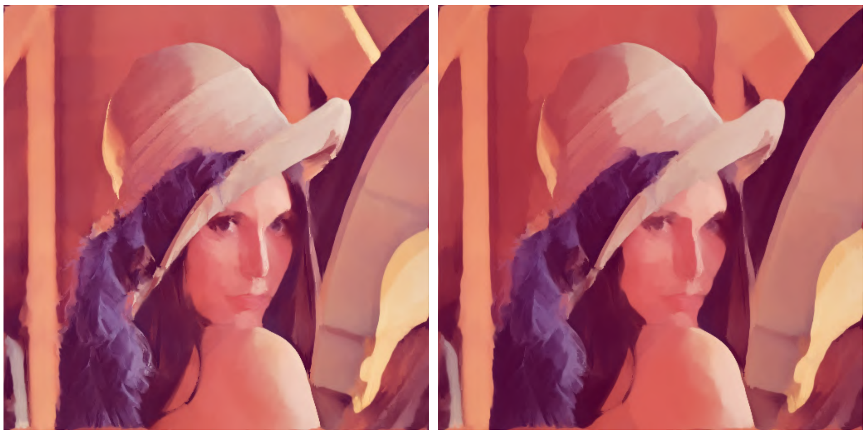

Figure 6.5: Comparison of filtration with multi-scale anisotropic Kuwahara filter divided into four and eight parts and with and without the tresholding enabled

6.1.3 Path creation

The path creation algorithm consists internally of two parts. The first one is the orientation estimation, which is entirely the same as in the case of the anisotropic Kuwahara filter. The second part is the generation of paths based on the local orientation of the image. On the other hand, this part is not entirely concluded anywhere in the literature. It is more of a combination of multiple known concepts, such as walking along the image gradients and the computation of Bézier curves. Therefore, it is hard to compare the results with other literature. Also, the different parameters supported by this operation can completely change the output of this method, as we can see in figure 6.6. For both examples, the artist used a mask, limiting the amount of path generated in the background of the original image, which is the Afghan girl photo. In the first image, we can see that a very small value of the spacing parameter was used, resulting in the paths being very dense. Combined with a pretty limited maximum path length and low stepping, this generated very detailed strokes that respect even the smallest detail in the image, such as lips or eyes. In the second image, longer paths with larger steps were generated. This setting nicely models less detailed areas of the image, such as the girl's shirt.

It is necessary to say that the look of the stroke paths does not convey complete visual information. The crucial part is the resulting strokes painted along these paths, which will be shown and discussed in part 6.3.2.

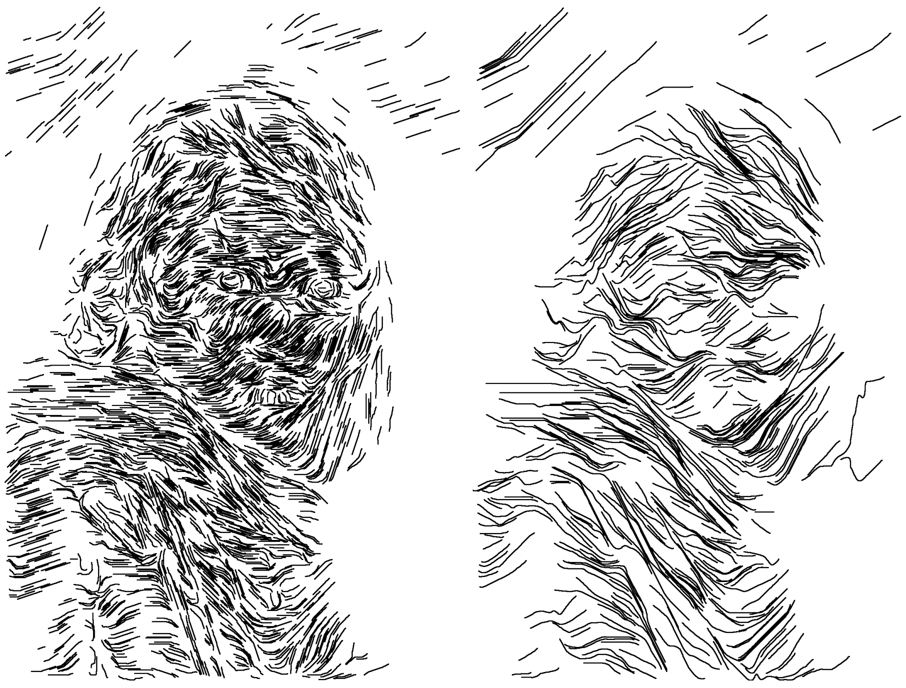

Figure 6.6: Two examples of paths generated using our method.

6.2 Performance comparison

We measured the image processing time using individual methods implemented in this part of the testing. Our goal in this work was to not implement algorithms as efficient and fast as possible, but to explore and test different methods image filtration. The data below shows the calculation time in the context of implemented algorithms. We performed the testing on two computers. The first one is a laptop equipped with an Intel i5-8265U processor, which operates at 3.70 GHz at this type of workload. The notebook also has 16 Gb of RAM running at 2400 MHz. The second is a desktop computer equipped with an AMD Ryzen 5900x processor coupled with 32 Gb of RAM running at 3200MHz. This computer is also equipped with an Nvidia GTX 1070Ti graphics card.

The measured results can be seen in table 6.1. We can observe that the image processing time increases linearly with the image size, or at least for the two smaller images. The 4k image processing on CPU took noticeably longer. The relationship between filtration time and filter size is not so favorable. From the measured data, it can be seen that doubling the size of the filter will roughly triple the computation time.

filter type	filtration time lenna.png [ms]	filtration time gunman.jpg [ms]	filtration time dama.jpg [ms]
Kuwahara filter r = 6	20	110	2283
Kuwahara filter r = 12	29	170	7709
Anisotropic Kuwahara filter N=4, r = 6	1400	11450	95715
Anisotropic Kuwahara filter N=4, r = 12	4320	35090	294295
Anisotropic Kuwahara filter N=8, r = 6	1710	14560	119680
Anisotropic Kuwahara filter N=4, r = 12	5740	49630	360033
Multi-scale anisotropic Kuwahara filter N=8, r = 6	3480	30954	528342

Table 6.1: Comparison of the speeds of individual CPU filtration methods for inputs lenna.png (512 × 512px), gunman.jpg (1280 × 1639px) and dama.jpg (3840 × 2560px), tested on an Intel i5 equipped notebook.

In the second part of performance testing, we compared the processing time of one picture with various types of filtration on different hardware. We tested five different configurations. Firstly, we tested the CPU implementation on the above-mentioned notebook with an Intel processor (CPU 1) and a desktop AMD processor (CPU 2). The second part was testing the GPU implementation, which was run on the Intel processor with its integrated graphics card (GPU 1). Secondly, it was paired with the dedicated Nvidia MX 250 GPU (GPU 2). Lastly, the GPU implementation was tested on a Ryzen desktop computer with Nvidia 1070Ti (GPU 3). Results of these tests can be seen in table 6.2.

filter type	CPU 1 [ms]	CPU 2 [ms]	GPU 1 [ms]	GPU 2 [ms]	GPU 3 [ms]
Anisotropic Kuwahara filter N=4, r = 6	15608	8223	236	81	13
Anisotropic Kuwahara filter N=4, r = 12	48988	26856	742	258	43
Anisotropic Kuwahara filter N=8, r = 6	18430	10471	620	130	21
Anisotropic Kuwahara filter N=8, r = 12	57807	34453	3157	446	63
Multi-scale anisotropic Kuwahara filter N=8, r = 6	28808	49264	1713	336	53

Table 6.2: Comparison of the speeds of individual filtration methods based on different hardware. Tested on the input image which is used in the last part of this work (resolution 1920 × 1080 px)

The last algorithm implemented is path generation. The first half of the computation is identical to the anisotropic Kuwahara filter. Thus, the processing time is the same. On the other hand, the second part of the algorithm is highly output sensitive, meaning that the number of generated curves determines the speed. With smaller images where the number of curves is in the hundreds, the computation time starts at around 10-15 seconds. But in the case of larger images with dense curves, there can be in a neighborhood of ten thousand curves, the computation time rises accordingly.

6.3 Use in the creative process

During the creation of this work, the practical usability testing of the implemented methods was carried out throughout the process. The intended use of individual algorithms has also changed slightly in connection with this. The initial thought was to create a program that would only work as a "black box" in which the image would be inserted, with the option of setting several parameters according to which the result would be created. The artist would then use the modified image in the following stages of the creative process. Gradually, however, we discovered that the implemented algorithms could have a wider use than we originally anticipated. After the initial understanding of the whole concept and the current workflow, we picked three main areas on which we wanted to focus. The first was the initial image disruption. The next part was image segmentation, and the last part was about creating the brush strokes.

The first part of the project initially consisted of implementing a tool that would be able to disrupt the initial perfection and technical purity of a photograph. For this purpose, a Kuwahara filter alone would suffice without any improvements. After showing the artist the outputs of the anisotropic Kuwahara filter, he immediately recognized the potential of this approach. Because in many situations, this filter itself creates structures that, with their shape and layout, resemble brushstrokes in digital painting, therefore creating a pretty good underpainting. This led to the subsequent implementation of both the anisotropic Kuwahara filter and its multi-scale extension. Via tests, we found that no specific setting for one implemented method can produce a perfect result in all respects. Instead, we added the option to generate multiple outputs with different filtering parameters automatically.

Next, we would like to briefly describe the digital painting process, which includes our program, including examples of partial results. The first is the initial filtration of the input image. The filter settings that have worked for us, together with the examples of their outputs, can be seen below in the pictures 6.7 and 6.8. It is an anisotropic Kuwahara filter divided into eight parts and filter sizes 3 and 10. There are also the outputs from a multi-scale anisotropic Kuwahara filter with blending parameters (0.2, 1.2, 0.1), (0.5, 1.2, 0.1), and (4.0, 4.0, 4.0). With these parameters, both the original size image and the half-resolution image are filtered. In the image 6.8, it is possible to see the outputs of the individual filterings and their final composition.

When the underpainting is complete, it is possible to proceed with the actual painting or, as we called it in the beginning, the addition of structures. This consists of the brush painting, which automatization will be discussed in the following part, and the addition of various textures and colour elements not present in the source image. The resulting image after all these manipulations can be seen in the figure 6.9 on the left. We can notice the added textures in the background and grass and numerous areas where purple spots were added.

After the artist makes various manual adjustments, it is time to bring the image to its final form. In this part of the process, we encountered the next possible use case for our implementation. Gradual adjustments made to the image may cause some crucial details of the original photo to degrade or even disappear. The artist can solve this situation in two ways. Firstly, it is possible to use the original input photo and mix selected details back into the output directly from it. However, the details added in this way are relatively easy to recognize, and their drawing and purity are quite different from the digital painting appearance. To not make the contrast with the painting noticeable, this procedure must be performed very carefully, which takes a

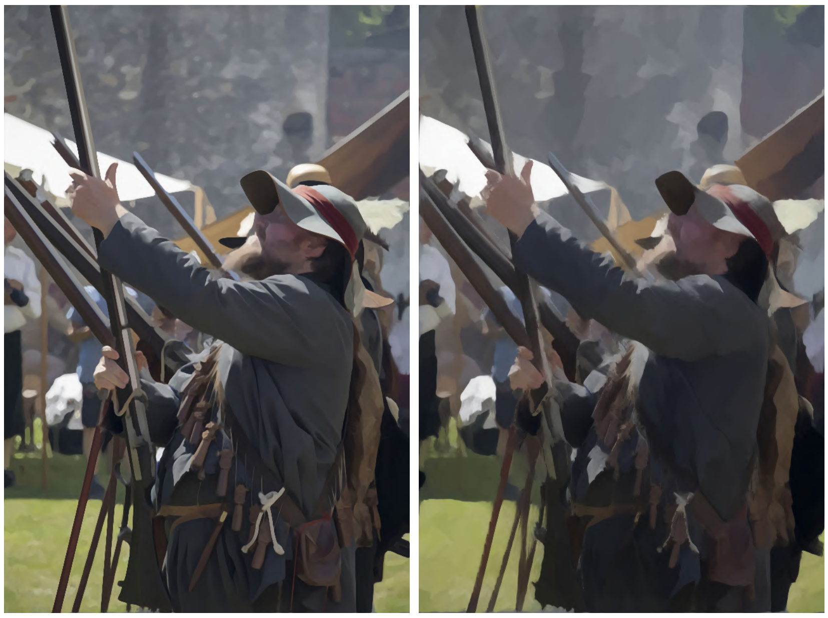

Figure 6.7: Filtration of image gunman.jpg using an anisotropic Kuwahara filter divided into eight parts for filter size of 10 and multi-scale anisotropic Kuwahara filter with color thresholding with mixing parameters $p_s = 0.5$, $p_d = 1.2$, $\tau_v = 0.1$

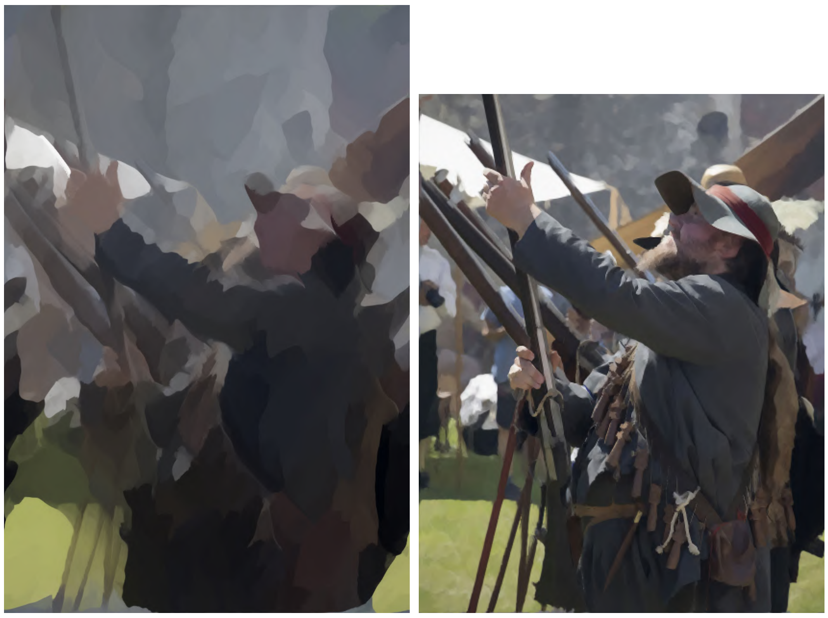

Figure 6.8: Filtration of the image gunman.jpg using a multi-scale anisotropic Kuwahara filter with color thresholding for mixing parameters $p_s = 0.2$, $p_d = 1.2$, $\tau_v = 0.1$ and the resulting composition of individual filtered images

lot of time. The second option is to repaint these details manually. This may be even more time consuming, but there is no need to balance photorealism and painting style.

The solution came from our implementation. If we set a small filter size, we get a resulting image that no longer looks like the photo, but at the same time, even tiny details are preserved. Therefore, it is possible to mix parts of the filtered image back into the resulting image without further editing. The added details will not even resemble the drawing of the original photo. An example of such an adjustment can be seen in 6.9.

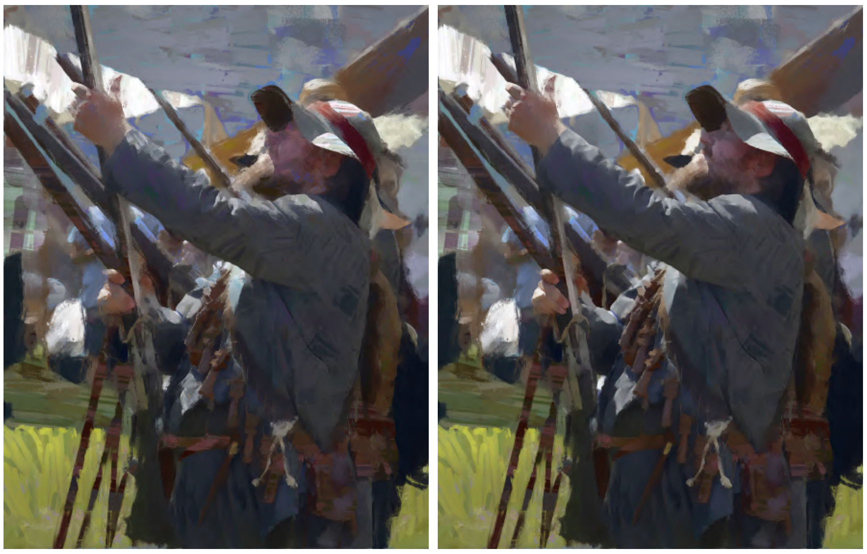

Figure 6.9: Comparison of the output of the creative process without the returned details and with the returned details

6.3.1 Image segmentation

Image segmentation is used in the creative process to divide the image into layers, which can be edited individually. As a proof of concept, we tested the implementation of spectral matting, which was written in Matlab. After testing with the artist, we concluded that the individual matting components output from the program seemed to be useful for further processing. But the limited interactivity and performance, especially compared to the methods included in Photoshop, meant that a lot of effort and research would be needed to make this a viable option, which was out of the scope of this work. In image 6.10, we can see the output of the k-means algorithm, which created the initial matting components and the application where the image was stroked based on the segmentation.

6.3.2 Path creation

The last part of the project was the path creation. The scope of this part is to generate curve-based paths along which Photoshop can subsequently draw actual brush strokes. The artist currently uses his own actions and scripts, which can generate such paths directly in Photoshop. This approach has its own shortcomings. Main problems can be identified as the lack of control

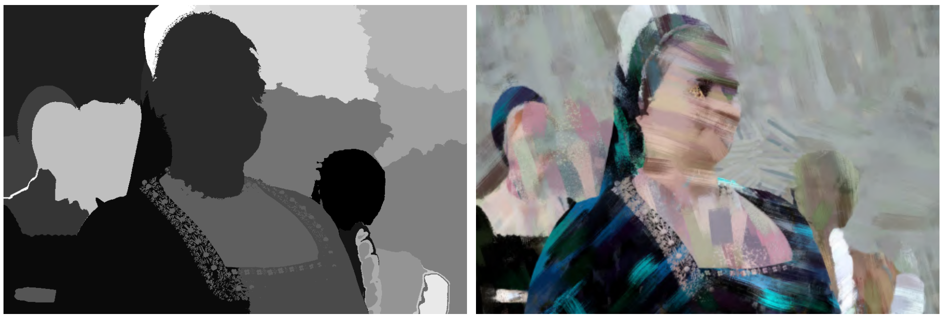
Figure 6.10: Output of K-means clustering and test of strokes based on the image segmentation.

over the process and the closed-loop nature of all the generated paths. An example of the Photoshop generated paths can be seen in figure 6.12. In this project, we have addressed both of these shortcomings. First of all, we can generate any type of path we want with this approach. Thus, we are not constrained by the need to create closed loops. Secondly, several parameters that determine the density, smoothness, and length of the path are completely definable by the user. In addition, based on a grayscale mask, the program dynamically changes the path density throughout various areas of the edited image to enhance user control over the process.

Next, we would like to introduce the complete creative process that uses all the algorithms implemented. First, the image is processed using one of the implemented Kuwahara filters. As was said in 6.3, there is no algorithm and its setting that would perfectly fit the whole image, which means that often multiple filtrations are performed, which are then manually combined. In this example, we worked with the input image 6.11. This was filtered using an anisotropic filter $n = 4$ with a filter size of 12. The second filtration was carried out using the multi-scale Kuwahara filter with setting $ps = 1.0$, $pd = 1.0$ and $tv = 0.0$ and filter size of 12 with activated thresholding. The combined output of these filtrations with additional manual colour grading is also shown in figure 6.11. We would describe this stage as the creation of an underpainting for the following steps.

The next step is path creation, which is performed on the input image because it contains the most accurate information, since it is not modified. In figure 6.12 we can see the paths which were generated with the original artist's approach directly in Photoshop. Next to it, we can see the output of our new method, which was again generated using a mask, which focused the path generation only on the woman's face and skipped the background. From the images, we can deduct that the nature of the paths is entirely different. The Photoshop version looks more coherent throughout the whole area of the image, and it seems as it goes by that the main shapes of the image and the "flatter" insides of these shapes are more affected by these main shapes.

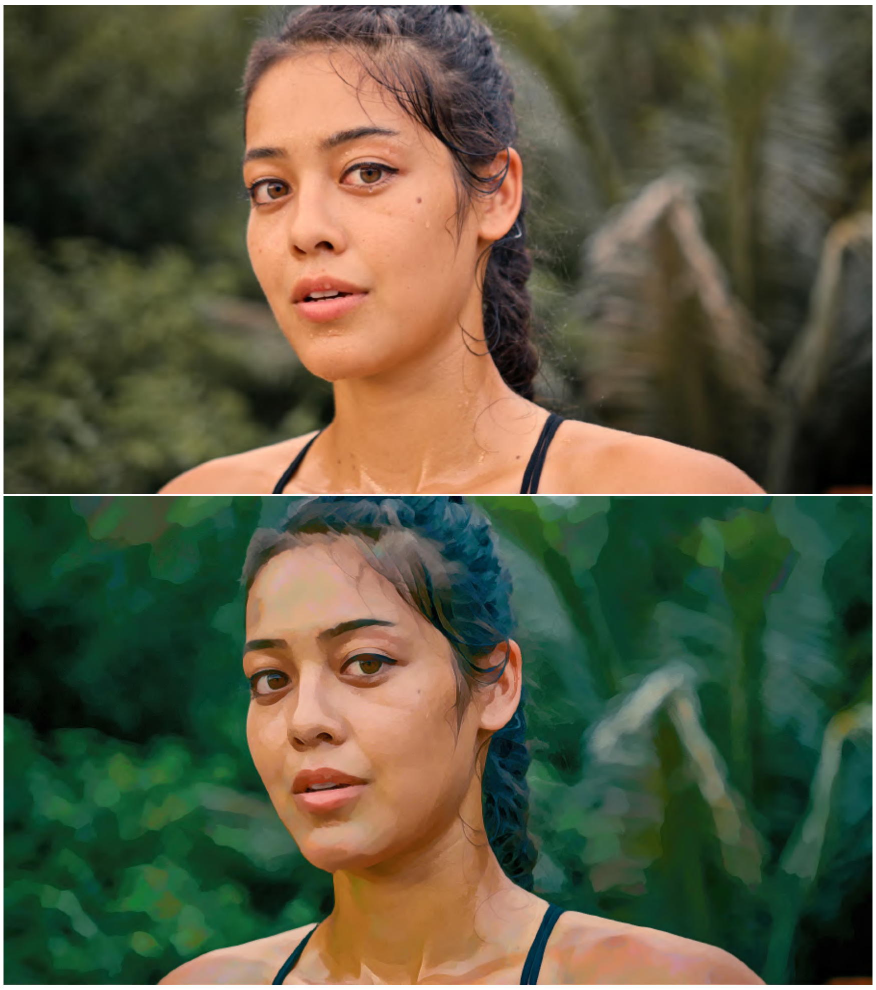

Figure 6.11: Comparison of input image and Kuwahara filtrated and color adjusted image.

In contrast, our approach is more detailed and tries to incorporate all minor details into the result. Even though it is possible to set the parameters in a way that makes the result look more similar to the Photoshop approach, the result will still be different. This is probably the most significant benefit of our approach, that it will always keep a lot of the details from the input image. This makes the result more diverse, which gives the impression that it could have been handcrafted and that it is not a computer program work.

The last part of this process is then performed in Photoshop, where the paths are imported as described earlier in 5.4. As in the initial filtration, the artist will generate multiple layers of paths and each will be used for a different part of the image. Then the automatic stroke drawing procedure is applied. An example of the resulting strokes themselves and their final combination with the underpainting can be seen in figure 6.13.

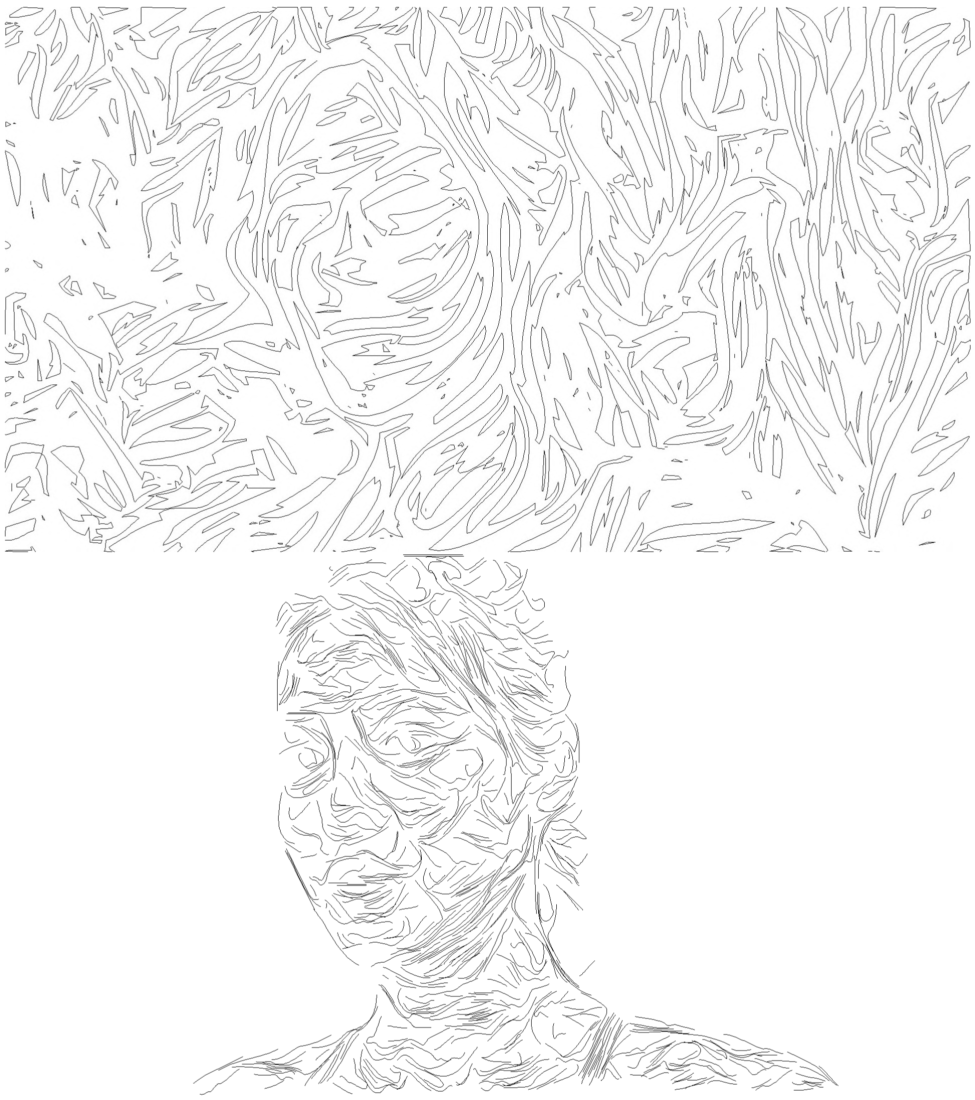

Figure 6.12: Comparison of the Photoshop generated stroke paths compared to our method.

6.3.3 Artist's feedback

"One of the main purposes of the "Paintbot" project was to automatize overload of labor during the specific style - realistic, digital painting process. In the end, Paintbot can reduce this even by 50% and more. This means that the artists could potentially save 50% of time. Paintbot was designed to generate a stylistically acceptable result that could be used as a base layer (underpaint) during the process. Another reason was to achieve a certain level of consistency with this approach. This could have a huge benefit if we could transfer the workflow into an animated sequence of images, where consistency is the key. The challenge was to make output solid and consistent but also allow some level of art direction for the user/artist. This was achieved not only by the optional Kuwahara filter setup but also by the strategy of introducing another

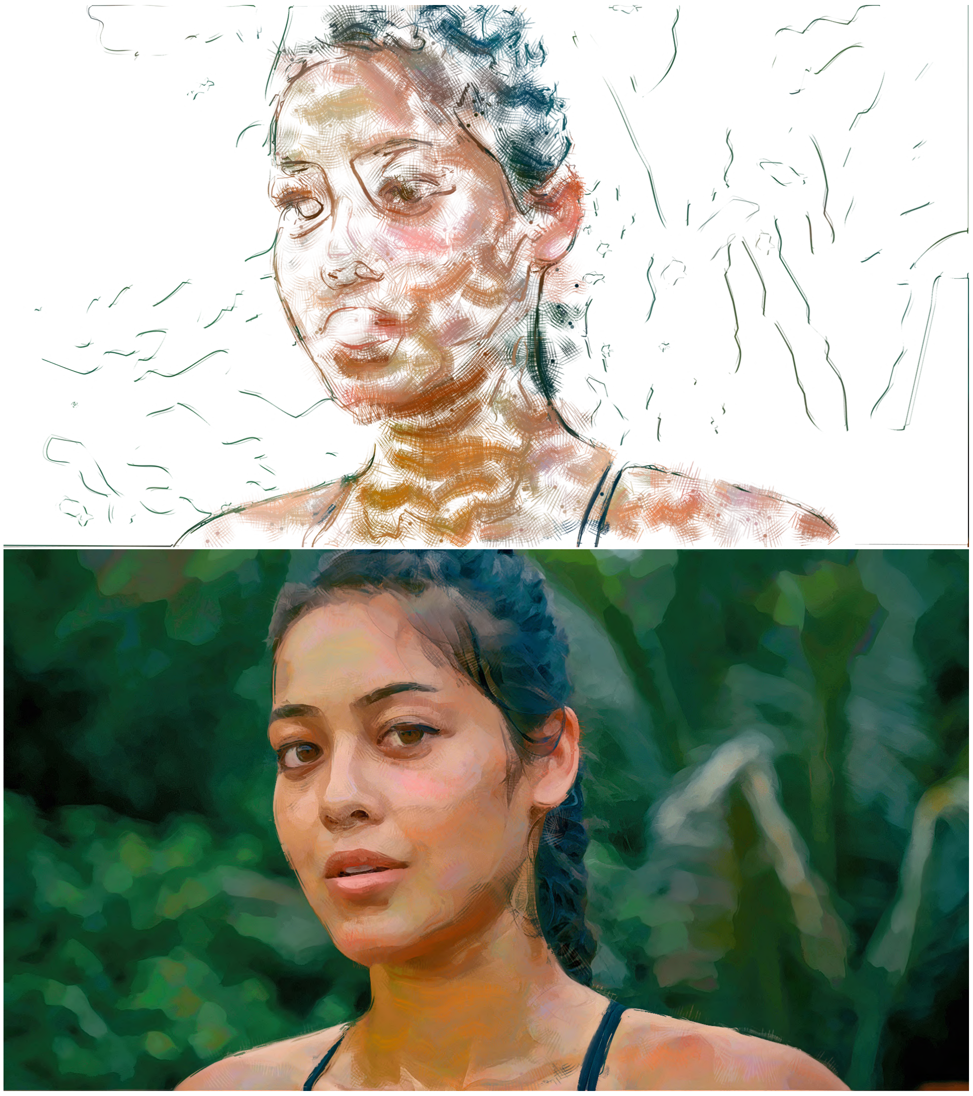
Figure 6.13: Example of Photoshop generated strokes based on our paths and combination of generated strokes with the underpainting.

stylization layer simulating paint strokes. These vector-based generated lines could hold any stroke style and, by that, offer a broad variety of potential looks. With this option, the paintbot underpainting layer is capable of saving even more than 50% of labor. In some scenarios, it could even be 60-70% of timesave.

Because the whole pipeline was design based on my own painting process, the result workflow follows my personal, paiting logic, and I could work with in naturally. Still, there are many manual steps that I would need to do after the paintbot layer, to make a solid painting. But Paintbot can offer me an amazing base to start with and keep the consistency in the case of animation. But there is much more that could be implemented to push it even further!"

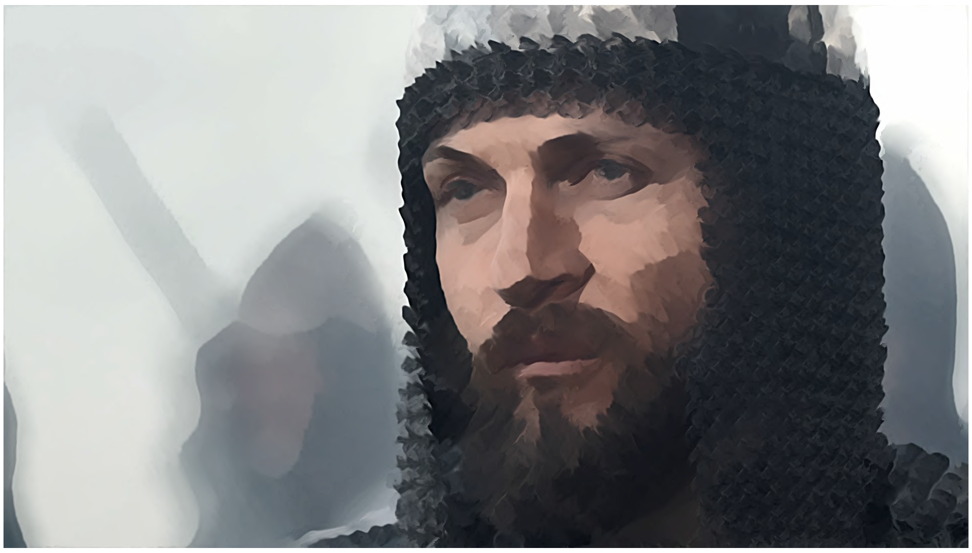

Figure 6.14: Example of keyframe generated by EbSynth based on input images stylized with our method.

As mentioned above in the feedback, a great feature of our implementation is its consistency. The artist is then capable of creating short videos or animations where the image consistency is a key factor. He tested two main approaches. The first is frame-by-frame stylization of a video, which is actually feasible thanks to the processing speed of the GPU implementation. But the second approach might be even more interesting. In this process, the artist uses a program called EbSynth [JvST+19], which can stylize a whole video sequence based on only a few keyframes. In image 6.14 we can see an animation frame that is generated by EbSynth from keyframes that were generated with our method. Figures 6.15 and 6.16 show example results of our approach with zoom-in on various details. In the first column, there is the original input image. In the second column, there is an image filtrated with our approach and slightly color calibrated. In the last column we can see the final image with brushstrokes that are based on our approach.

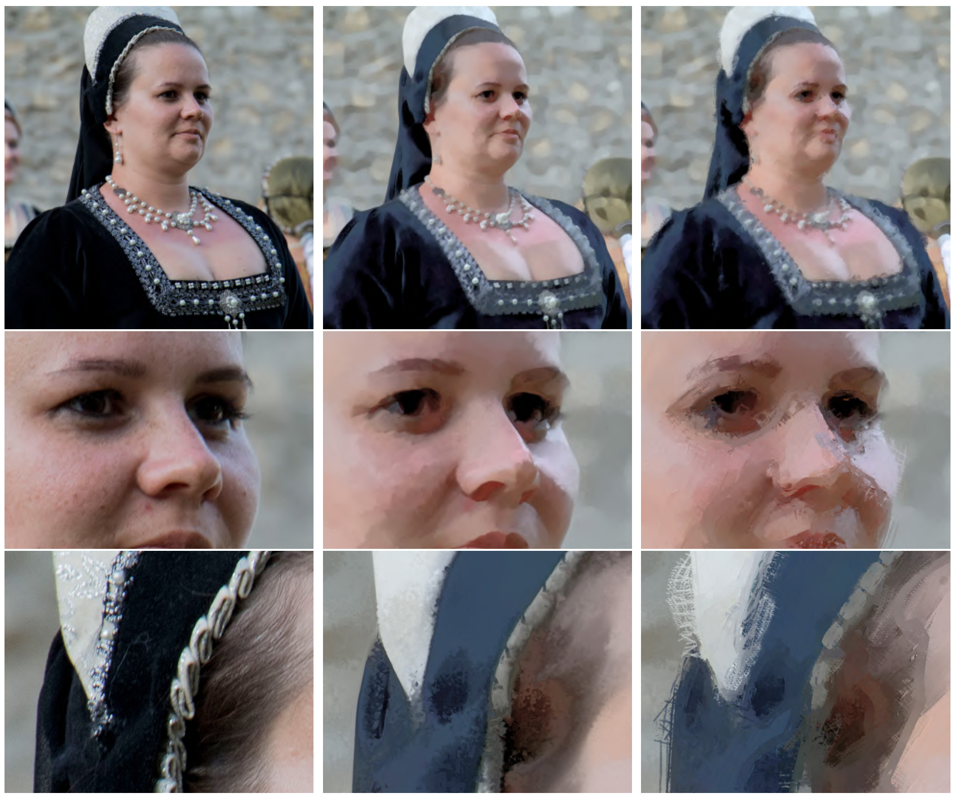
Figure 6.15: Example of original image and the stylized version with our approach.

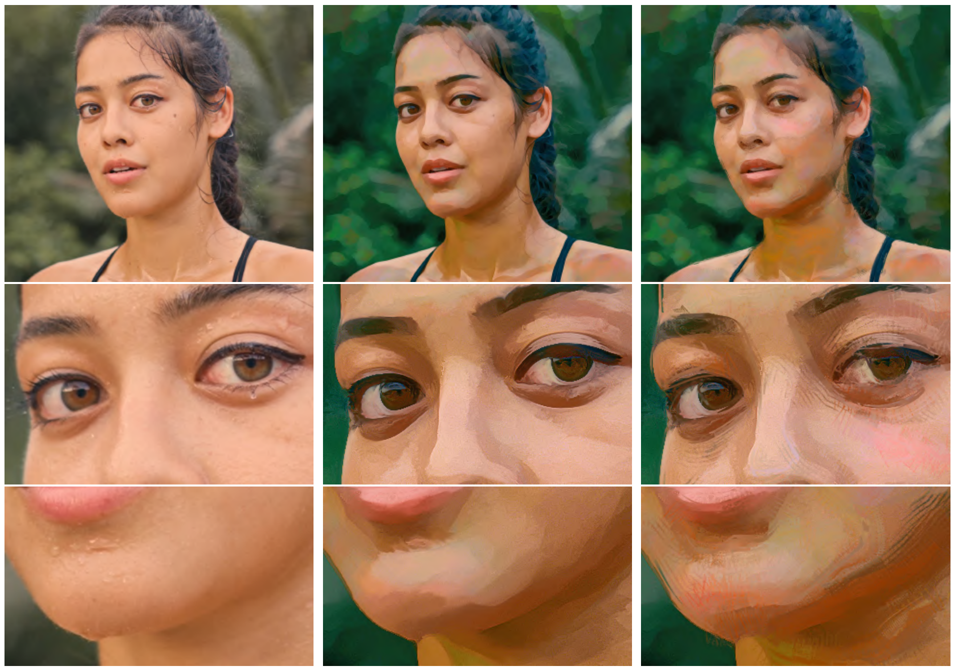
Figure 6.16: Example of original image and the stylized version with our approach.

Chapter 7

Conclusion

The master's thesis on Photo Stylization Using Painterly Rendering focused on discovering the possibilities of using automated image processing methods in the creative process of manual stylization of photography into a digital painting of a digital painter Jakub Javora.

In the beginning, we worked with the artist to analyze the current state of his creative process. Part of the analysis was the need to understand the meaning and content of the process as a whole and then understand the visual properties of individual steps and their outputs, including an understanding of the artistic perception of the artist in the creation of the work. Then we were able to break down the existing process into smaller, partial tasks that corresponded to the individual adjustments made to the image.

However, many actions performed with a painting depend on the aesthetic perception and feelings of the artist, so it would not be easy to find methods that would be able to represent his intuition and distinctive expression. After identifying meaningfully automatable parts of the creative process, we searched for existing approaches and algorithms used in practice. Then came the discussion of the usability of the procedures found. After discussing the usability of the methods found, it was possible to proceed with their implementation. As part of the ongoing testing of each procedure, we made several minor adjustments to the algorithms used.

The result is a program that implements several image filtering methods, all of which are parameterizable in order to achieve results exactly according to the user's preferences.

The benefit of this work is that we managed to create a practical tool that can be used to automate various parts of the creative process, from the initial preprocessing and distortion of the input photo to the creation of the final digital painting. At the same time, the output of the program can be used in other phases of the creative process, such as returning the original details to the final image. The second part of the implementation focuses directly on painting the picture, where we have implemented a tool which can generate paths along which actual digital brush strokes are rendered. As part of this work, we managed to make part of the artist's workflow more efficient while introducing new methods for controlling these semiautomatic steps. But the whole process of the artist's work is significantly more complex and longer than just the part we worked on.

By combining the creative invention of the artist and the sensitive and meaningful use of computer tools, a work of art can be created. It is already up to the discerning v iewer to decide whether he will pass the artwork without significant interest or whether the piece will hit him and enrich his inner world.

Appendix A

Bibliography

[AM20] Rosa Azami and David Mould, *Image abstraction through overlapping region growth*, 05 2020.

[aut96] Contributing authors, `https://ijg.org/`, 1996, Accessed: 2022-01-11.

[BA83] P. Burt and E. Adelson, *The laplacian pyramid as a compact image code*, IEEE Transactions on Communications **31** (1983), no. 4, 532–540.

[BFL06] Yuri Boykov and Gareth Funka-Lea, *Graph cuts and efficient nd image segmentation*, International Journal of Computer Vision - IJCV **70** (2006), 109–131.

[BY15] Ahmadreza Baghaie and Zeyun Yu, *Structure tensor based image interpolation method*, AEU - International Journal of Electronics and Communications **69** (2015), no. 2, 515–522.

[CM02] D. Comaniciu and P. Meer, *Mean shift: a robust approach toward feature space analysis*, IEEE Transactions on Pattern Analysis and Machine Intelligence **24** (2002), no. 5, 603–619.

[Dic12] Dictionary.com, *isotropic*, `https://www.dictionary.com/browse/isotropy`, 2012, Accessed: 2022-01-07.

[DS02] Doug DeCarlo and Anthony Santella, *Stylization and abstraction of photographs*, ACM Trans. Graph. **21** (2002), no. 3, 769–776.

[GR96] Mark Adler. Greg Roelofs, Jean-loup Gailly, `https://zlib.net/`, 1996, Accessed: 2022-01-11.

[GRC08] Woods R. E Gonzalez R. C., *Digital image processing (3rd edition*, Prentice Hall, 2008.

[GW08] Rafael C. Gonzalez and Richard E. Woods, *Digital image processing*, Pearson, 2008.

[Hae90] Paul Haeberli, *Paint by numbers: Abstract image representations*, SIGGRAPH Comput. Graph. **24** (1990), no. 4, 207–214.

[Her98] Aaron Hertzmann, *Painterly rendering with curved brush strokes of multiple sizes*, Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (New York, NY, USA), SIGGRAPH '98, Association for Computing Machinery, 1998, p. 453–460.

[JSK⁺99] Bernd Jähne, Hanno Scharr, S. Körkel, Bernd Jähne, Horst Haußecker, and Peter Geißler, *Principles of filter design*, vol. 2, Academic Press, 1999, pp. 125–151.

[JvST⁺19] Ondřej Jamriška, Šárka Sochorová, Ondřej Texler, Michal Lukáč, Jakub Fišer, Jingwan Lu, Eli Shechtman, and Daniel Sýkora, *Stylizing video by example*, ACM Transactions on Graphics **38** (2019), no. 4.

[KD08] Jan Eric Kyprianidis and Jürgen Döllner, *Image abstraction by structure adaptive filtering*, Proc. EG UK Theory and Practice of Computer Graphics, 2008, pp. 51—-58.

[KHEK76] M. Kuwahara, K. Hachimura, S. Eiho, and M. Kinoshita, *Processing of ri-angiocardiographic images*, pp. 187–202, Springer US, Boston, MA, 1976.

[KK11] Jan Eric Kyprianidis and Henry Kang, *Image and video abstraction by coherence-enhancing filtering*, Computer Graphics Forum **30** (2011), no. 2, 593–602.

[KKD09] Jan Eric Kyprianidis, Henry Kang, and Jürgen Döllner, *Image and video abstraction by anisotropic kuwahara filtering*, Computer Graphics Forum **28** (2009), no. 7, 1955–1963.

[Kyp11] Jan Eric Kyprianidis, *Image and video abstraction by multi-scale anisotropic kuwahara filtering*, Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering (New York, NY, USA), NPAR '11, Association for Computing Machinery, 2011, p. 55–64.

[Kyp13] ______, *Artistic stylization by nonlinear filtering*, pp. 77–101, Springer London, London, 2013.

[LGD18] T. Lindemeier, J. M. Gülzow, and O. Deussen, *Painterly rendering using limited paint color palettes*, Proceedings of the Conference on Vision, Modeling, and Visualization (Goslar, DEU), EG VMV '18, Eurographics Association, 2018, p. 135–145.

[LLW08] Anat Levin, Dani Lischinski, and Yair Weiss, *A closed-form solution to natural image matting*, IEEE transactions on pattern analysis and machine intelligence **30** (2008), 228–42.

[LRAL07] Anat Levin, Alex Rav-Acha, and Dani Lischinski, *Spectral matting*, 2007 IEEE Conference on Computer Vision and Pattern Recognition, 2007, pp. 1–8.

[ODPJ20] Sterling Orsten, Dimitri Diakopoulos, Juan Patten, and Wojciech Jabłoński, `https://github.com/sgorsten/linalg`, 2020, Accessed: 2022-01-11.

[PPC07] Giuseppe Papari, Nicolai Petkov, and Patrizio Campisi, *Artistic edge and corner enhancing smoothing*, Ieee transactions on image processing **16** (2007), no. 10, 2449–2462 (English), Relation: http://www.rug.nl/informatica/organisatie/overorganisatie/iwi Rights: University of Groningen. Research Institute for Mathematics and Computing Science (IWI).

[Roe00] Greg Roelofs, `http://www.libpng.org/pub/png/libpng.html`, 2000, Accessed: 2022-01-11.

[Sch92] Dale Schumacher, *1.2 - general filtered image rescaling*, Graphics Gems III (IBM Version) (DAVID KIRK, ed.), Morgan Kaufmann, San Francisco, 1992, pp. 8–16.

[SF73] Irwin Sobel and Gary Feldman, *A 3×3 isotropic gradient operator for image processing*, Pattern Classification and Scene Analysis (1973), 271–272.

[SHB14] Milan Sonka, Vaclav Hlavac, and Rodger Boyle, *Image processing, analysis, and machine vision*, Cengage Learning, 2014.

[SIM$^+$17] Nigel Stewart, Milan Ikits, Marcelo Magallon, Aaron Lefohn, Joe Kniss, and Chris Wyman, *The opengl extension wrangler library*, `http://glew.sourceforge.net/`, 2017, Accessed: 2022-02-24.

[SLKD15] Amir Semmo, Daniel Limberger, Jan Kyprianidis, and Jürgen Döllner, *Image stylization by oil paint filtering using color palettes*, 06 2015.

[SM00] Jianbo Shi and J. Malik, *Normalized cuts and image segmentation*, IEEE Transactions on Pattern Analysis and Machine Intelligence **22** (2000), no. 8, 888–905.

[Str86] Steve Strassmann, *Hairy brushes*, Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (New York, NY, USA), SIGGRAPH ’86, Association for Computing Machinery, 1986, p. 225–232.

[SWHS97] Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin, *Orientable textures for image-based pen-and-ink illustration*, Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (USA), SIGGRAPH ’97, ACM Press/Addison-Wesley Publishing Co., 1997, p. 401–406.

[Sý21] Daniel Sýkora, *Separable kernel*, `https://dcgi.fel.cvut.cz/home/sykorad/aim/slides/l03.pdf`, 2021, Accessed: 2022-01-10. Slide 7.

[TM98] C. Tomasi and R. Manduchi, *Bilateral filtering for gray and color images*, Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271), 1998, pp. 839–846.

[Tsc04] David Tschumperlé, `https://www.dictionary.com/browse/isotropy`, 2004, Accessed: 2022-01-11.

[TTK$^+$21] Aneta Texler, Ondřej Texler, Michal Kučera, Menglei Chai, and Daniel Sýkora, *Faceblit: Instant real-time example-based style transfer to facial videos*, Proceedings of the ACM in Computer Graphics and Interactive Techniques (Proceedings of I3D 2021) **4** (2021), no. 1, 14.

[vBSF05] Jiří Žára, Bedřich Beneš, Jiří Sochor, and Petr Felkel, *Moderní počítačová grafika (2. vydání)*, Computer Press, 2005.

[vSJ21] Šárka Sochorová and Ondřej Jamriška, *Practical pigment mixing for digital painting*, ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2021) **40** (2021), no. 6, 234.

[WOG06] Holger Winnemoeller, Sven Olsen, and Bruce Gooch, *Real-time video abstraction*, ACM Trans. Graph. **25** (2006), 1221–1226.