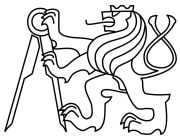
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Example-Based Stylization of Navigation Maps on Mobile Devices
Student:	Bc. Ondřej Texler
Supervisor:	doc. Ing. Daniel Sýkora, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2018/19

Instructions

Explore the state-of-the-art in guided texture synthesis [1, 2] and adopt selected techniques to implement example-based style transfer into navigation maps. Focus both on the quality of the resulting synthesis and on the processing speed. Identify time-consuming parts of the implemented algorithm that are suitable for parallelization and consider their parallel implementation using std::threads or OpenCL. Consider also the reduction of computational overhead by reusing parts that were already synthesized or parts for which faster bitmap operations can replace texture synthesis. Try to achieve interactive response during navigation or map exploration on currently available mobile devices.

References

[1] Fišer et al.: StyLit: Illumination-Guided Example-Based Stylization of 3D Renderings, ACM Transactions on Graphics 35(4):92, 2016.
[2] Kaspar et al.: Self Tuning Texture Optimization, Computer Graphics Forum 34(2):349-360, 2015.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague September 9, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering)

Master's thesis

Example-Based Stylization of Navigation Maps on Mobile Devices

Bc. Ondřej Texler

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

8th January 2018

Acknowledgements

First of all, I would like to express my gratitude to doc. Ing. Daniel Sýkora, Ph.D. for his phenomenal guidance throughout the process of creating this thesis, for his enthusiasm, frequent and continuous advice and comments and for motivating me. Further thanks to my family for supporting me during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 8th January 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Texler, Ondřej. *Example-Based Stylization of Navigation Maps on Mobile Devices.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato diplomová práce pojednává o vedené syntéze textury na základě předlohy. Syntéza textury má za cíl co nejvěrněji napodobit umělecký styl i použitý umělecký nástroj. V moderní počítačové vědě je syntéza textury častým předmětem výzkumu a má široké uplatnění. Obsahem této diplomové práce jsou definice a popis problému vedené syntézy textury a jejího řešení. V práci je k nalezení přehled základních i pokročilých technik a algoritmů používaných k řešení problému syntézy textury. Implementace algoritmu syntézy textury popsána v této práci byla integrována do existující mobilní navigace. Detaily implementace a integrace lze v této práci taktéž najít. Možnost paralelizace problému byla vzata v potaz a její realizace je rovněž vysvětlena. Dále práce zahrnuje výsledky, porovnání, experimenty a měření, které byly na algoritmu syntézy textury provedeny.

Klíčová slova stylizace, syntéza textury, na základě předlohy, mapa výskytů, paralelizace, mobilní zařízení

Abstract

This thesis deals with example-based guided texture synthesis. The goal of texture synthesis is to reproduce artistic style and used art tool as faithfully as

possible. In modern computer science texture synthesis is an active research problem and has wide use. The thesis includes a definition and description of the guided texture synthesis problem and its solution. An overview of both basic and advanced methods and algorithms used to solve the texture synthesis problem are provided. Implementation of the texture synthesis algorithm described in this thesis was integrated into an existing mobile navigation application, implementation and integration details are included. Parallelization was considered and implemented and is described in this thesis as well. Additionally, the thesis contains results, comparisons, experiments and measurements which were performed on the texture synthesis algorithm.

Keywords stylization, texture synthesis, example-based, occurrence map, parallelization, mobile device

Contents

List of Figures

List of Tables

Introduction

If we have a painting of a certain artistic style, an artist is able to draw a new image of different content in the same style as his inspiration. The problem is when the new image should be significantly larger and would consume too much of the artist's time, that is where computer texture synthesis becomes really helpful. For example, a detailed map of Europe or of the whole world is much bigger than an image which could be drawn by a human artist, so texture synthesis is needed. Furthermore, if we do not know the content of the new image in advance, computer texture synthesis is the only option. The goal of this thesis is to explore the state-of-the-art in texture synthesis and implement example-based style transfer into navigation maps.

Example-based stylization is a complex problem. In modern computer science, there has been a lot of research in this field and there has been remarkable progress in quality and speed of texture synthesis. At present time, texture synthesis is still a very active researcher topic, meaning there is a lot of researchers and research groups.

Motivation

Computer-generated navigation maps are often confusing and it is hard to find important information for orientation. That is especially the problem when a driver drives a car and he has only a few seconds to look at the navigation map and understand where he should drive. A thesis, Visualizing Route Maps [3], has been published which describes how valuable, useful and easy to follow are hand-drawn maps instead of classical computer-generated maps. To the best of our knowledge, there is not any mobile navigation, which uses handcrafted or stylized maps.

Many state-of-the-art stylization approaches work well with complex images, for example, photos but fail on simple images like a screenshot from a map. In a complex scene and a complex artistic style, it is easy to hide some glitches without notice. However, these glitches are very apparent in a simple

scene and a simple style. The survey showed that StyLit [4] is currently the best and the most suitable technology for map stylization. This thesis uses methods from StyLit.

Goal of this Thesis

The goal of this thesis is to implement guided texture synthesis to stylize navigation maps on mobile devices by using a given style example. Requirements for this stylization are both the speed and quality of a result. Texture synthesis implementation, described in this thesis, is integrated into an existing mobile navigation application Dynavix [5] as the prototype.

Thesis Structure

The thesis is divided into seven chapters: Introduction, Background, Method, Algorithm, Implementation, Results and Comparison and Conclusion. Next is the summarized content of each chapter.

Introduction. At first, the texture synthesis problem and the motivation to solve it are briefly presented in the Introduction chapter. It is followed by related work where many past and present methods are presented.

Background chapter contains a general description of texture synthesis and its types. There, the texture energy and global optimization approach used to solve texture synthesis are defined. The chapter also includes a definition of the Image Analogies [6] concept.

Method. In this chapter, basic terms and methods related to texture synthesis are defined. The texture synthesis problem and energy function are defined in a more formal way.

Algorithm chapter contains a detailed description of methods and algorithms as well as images with examples. Pseudocodes of methods are provided.

Implementation chapter describes the implementation of algorithms and methods with more technical details and with more pseudocode examples. Additionally, parallel implementation as well as a description of the integration into Dynavix [5] is presented in this chapter.

Results and Comparison chapter compares quality and speed of our implementation with other approaches and implementations. A comparison with nowadays popular convolutional neural networks is presented.

Conclusion. Last chapter Conclusion contains a summarization and future work.

Related work

In recent years manny different approaches has been published, dealing with the creation of a digital painting. The algorithmic compositon of exemplar strokes proposed by Salisbury et al. [7] and Zhao and Zhu [8]. Decomposition of the stylized image into a set of meaningful parts to better preserve semantic of individual regions by Zeng et al. [9]. The physical simulation was proposed by Curtis et al. [10] and by Haevre et at. [11]. Procedural approach by Bousseau et al. [12] and Benard et al. [13]. Advanced image filtering by Winnemöller et al. [14] and by Lu et al. [15]. All mentioned approaches can produce impressive results but only on the particular type of data since they are limited by the used algorithm or by the library of applied brush strokes. The appearance of the result is defined only by the algorithm so it can produce only one or limited amount of styles.

Generic example based technique The Lit Sphere was introduced by Sloan et al. [16]. By artist painted shaded sphere is used as the style example and pixels from this exemplar are transferred to the target 3D model using texture mapping. Patch-based synthesis with more complex illumination guidance was proposed in StyLit [4].

Image Analogies is a concept proposed by Hertzman et al. [6]. There are two pairs of images, source images and target images (one image in the pair is filtered-stylized and one is unfiltered). Stylization is described by the source image pair. The algorithm iterates over unfiltered target pixels and finds the most similar location in the unfiltered source image and transfers look from the filtered counterpart. However, this approach suffers from introducing visible seams, repetition and other artefacts and does not preserve visual appearance of used art tool. Hashimoto et al. [17] and Benard et al. [18] extended this approach for animation.

Another example-based region-growing approach is by Efros et al.[1], where the new texture is generated by one pixel at one time. Given a sample texture image, a new image is initialized by a patch from the sample texture and this patch in the new texture is grown pixel by pixel. Another approach is proposed also by Efros et al. [2]. The new texture is not growing one pixel at a time but one patch at a time. New pixels or patches in both previous approaches are determined based on similarity with the sample texture.

Kwatra et al. [19] and Wexler et al. [20] introduced a texture optimization technique. They defined similarity metric for measuring the quality of synthesized texture with respect to a given input sample by Markov Random Field. Then they formulated the synthesis problem as the minimization of an energy function by using expectation-maximization like algorithm. In contrast to the region-growing approaches, it is not synthesized one pixel at a time or one patch at a time, but the whole new texture is refined many times in iterations, from randomly initialized, through coarse texture to the sharp, fine and faithfully-looked texture.

In recent work of Fiser et al. [21] and of Barnes et al. [22] there was a replaced original Hertzman [6] algorithm with texture optimization technique. This new approach is ideally suited for the controllable synthesis of textures and other adjustments, but it suffers from the so-called wash-out effect proposed by Newson et al. [23] and closely described by Jamriška et al. [24]. Wash-out effect makes some parts of the new texture to be smoothed or blurry. This undesired effect is caused by an extensive use of small amount of same or similar patches from which a new texture is composed. Many strategies have been developed to deal with the wash-out effect, e.g. discrete solver by Han et al. [25], feature masks by Lefebvre and Hoppe [26], color histogram matching by Kopf et al. [27], and bidirectional similarity by Simakov et al. [28] and Wei et al. [29]. These approaches work well in case of the source being mostly stationary without many nearly-homogeneous patches. Unfortunately, it does not work well on realistic style examples. More robust mitigation of the wash-out effect by encouraging uniform patch usage was recently published by Kaspar et al. [30] and Jamriška et al. [24] and before by Chen and Wang [31] and Benard et al. [18]. In our scenario, some patches need to be used multiple times, so that uniform patch usage is also not robust enough. Pure uniform patch usage is suitable for normal texture synthesis but not for guided texture synthesis.

Recently, few alternative approaches to achieve computer-assisted stylization by using Neural Networks were developed. [32] uses a deep convolutional neural network VGG [33], trained for object recognition. In this approach the VGG-Network is used for extracting information from a texture, it can extract information about both style and content. Texture synthesis is represented as the minimization problem solved by an iterative gradient descent. The minimization problem is defined by the similarity between VGG style response from Sample style and Output texture and VGG content response from Output guide and Output texture. This approach has impressive results in some cases. Since the VGG network is trained on natural images, it does not work well on images with synthetic appearance - like computer-generated maps. In complex images, it is easier to hide some artifacts and glitches, so this method does not work well on simple images. The additional drawback of this approach is that there is no intuitive way to control the transfer process. Extension of this approach, along with the original Hertzman Image Analogy [6] concept results in the Deep Image Analogy [34], that uses VGG as well. This has very promising results, but both input images have to be the same in content, which is the big limitation.

Chapter 1

Background

In this chapter we will talk about texture synthesis in general and types of texture synthesis, the difference between example-based and procedural synthesis, we will define and describe in detail region-growing and the global optimization method and the difference between guided and non-guided texture synthesis. The Image Analogy concept is also defined in this chapter.

1.1 Types of Texture Synthesis

The research in texture synthesis was quite active in the last two decades, [1], [2], [6], [19], [4], [30]. Many methods and algorithms facilitating texture synthesis, accelerating texture synthesis [35] or dealing with the guidance channels different way [4] have been proposed. Since there are many texture synthesis approaches, we can classify them into the following categories. Example-based or Procedural, Region-Growing or Global Optimization and Guided or Non-Guided texture synthesis. In the next three subsections, we will talk about these categories in more detail.

1.1.1 Example-based / Procedural

From the texture generating point of view, we can distinguish between example-based and the procedural approach. In case of procedural texture synthesis, the new texture is generated using predefined parts and methods. For example, texture is composed of the predefined set of brush strokes, filters and effects. The typical example of procedural approach is shown in the Interactive watercolor rendering with temporal coherence and abstraction [12] where the result is obtained using the static pipeline. At first, a paper effect (dry brush or wobbling) is applied, then an edge darkening effect and in the final step, there is a turbulent flow effect, pigment dispersion and a paper variation effect. This leads to the well looking watercolor image, but since this is a procedural approach, it can produce only watercolor images.

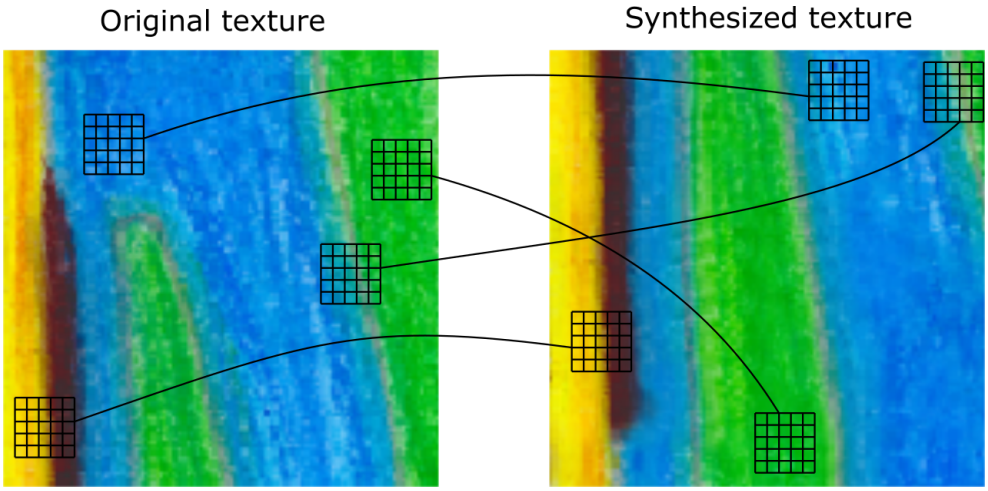


Figure 1.1: *Example of four similar patches in two textures.*

The Example-based approach is more generic, meaning it does not depend on the specific algorithm. An example of a style has to be provided and the new image is created based on this example. There is no limitation of styles and the current state-of-the-art example-based approaches (e.g. StyLit [4]) can reproduce almost every possible style very faithfully.

1.1.2 Region-Growing / Global Optimization

We can also categorize texture synthesis approaches into the Region-Growing category, where new texture grows one pixel or one patch at a time, or into the Global Optimization category, where the whole texture is refined in multiple iterations, from coarse initialization to a finite and faithfully looking texture. Both categories are described in more detail in the next two subsections.

1.1.2.1 Region-Growing

In the Region-Growing method [1], [6] the new texture is generated one pixel at a time or one patch at a time. Given the sample texture, the new image is initialized by a patch from the sample texture. This patch in the new texture is grown pixel by pixel. New pixels which are added to the new image are determined based on similarity with the sample texture. Meaning the part of the new image where the new pixel is added should look as similar as possible as any part of the sample image. It means that if you choose an arbitrary patch from the new image there should exist a very similar or almost the same patch in the original sample texture. See demonstration on 1.1.

Efros and Leung [1] described the Region-Growing algorithm so that given a sample texture image (left) 1.2, a new image is being synthesized one pixel at a time (right) 1.2. To synthesize a pixel, the algorithm first finds all the

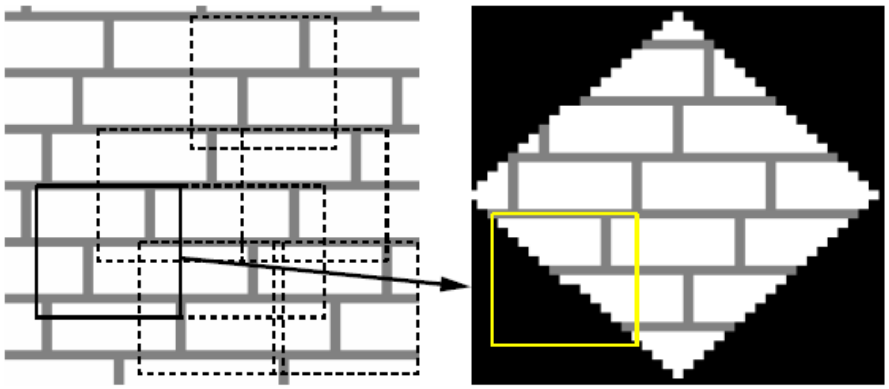


Figure 1.2: *Texture Synthesis by Non-parametric Sampling algorithm Overview. [1]*

neighborhoods in the sample image (boxes on the left) that are similar to the pixel's neighborhood (box on the right) and then randomly chooses one neighborhood and takes its center to be the newly synthesized pixel.

A little different approach is called the Image Quilting and was proposed by [2]. This approach still falls into the Region-Growing category but the new texture is not growing one pixel at a time but one patch at a time. The new image is initialized by one random patch from the sample texture. The sample texture is then searched for patches which best follow on the new image with respect to the defined overlap. The founded patch is "quilted" next to the new image with respect to this overlap. We can see the results of this algorithm on figure 1.3. The most left image shows a case with no overlap, it means that blocks are placed next to each other based on no information, so there are strong horizontal and vertical artefacts. The image in the middle shows a case with overlap and each new block is chosen with respect to this overlap, the boundary between blocks is horizontal or vertical, so it results in little horizontal and vertical artefacts in the image. The most right image shows a case with overlap and the boundary between blocks is not straight and is computed as a minimum cost path through overlap region, which results in a really well "quilted" image.

1.1.2.2 Global Optimization

This approach is built on a totally different basis than the Region-Growing algorithms. In this approach, we do not synthesize one pixel at a time or one patch at a time, but we refine a whole new texture many times in iterations, from randomly initialized, through coarse texture to the sharp, fine and faithfully-looking texture. This approach was published by [19] and texture synthesis is there defined as a problem of energy minimization using the

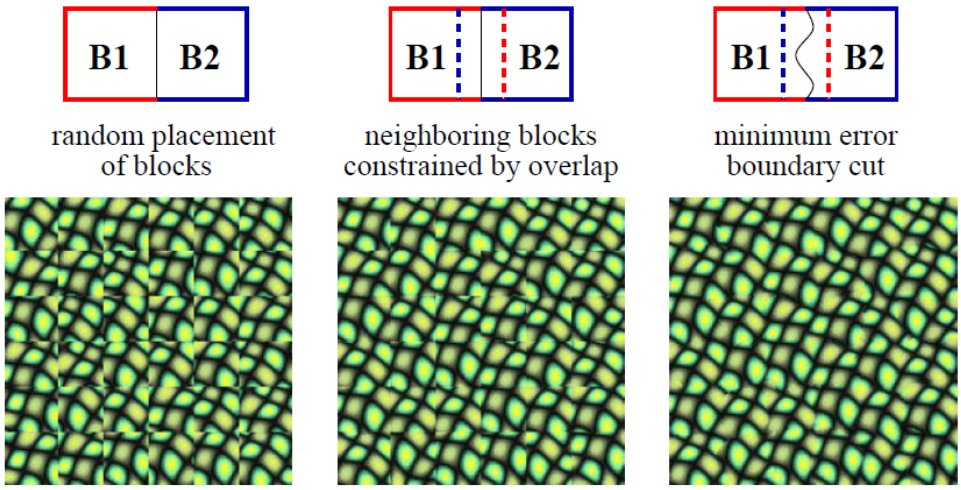


Figure 1.3: *Quilting texture. [2]. On the left image, blocks are placed next to each other randomly, there is no quilting. On the middle image, there is overlap and straigth quilting. On the right image, there is overlap and minimal boundary cut quilting.*

expectation-maximization-like algorithm. Given the sample texture, the new texture is synthesized. As long as the new texture does not look similar as the sample texture, the new texture has high energy and it is changed in a way to look more similar as the sample texture, so it is changed in such a way to have lower energy. In each iteration, each pixel from the new image is evaluated as the average of several pixels from the sample texture. Generally, more iteration leads to better results. The energy function E of a texture X with respect to the sample style S is defined as follows:

$$E = \sum_{x \in X} \min_{s \in S} [SSD(x, s)]$$

In other words, energy E is the sum of distances of each patch from texture X to its closest patch in texture S. The distance between two patches is computed using the SSD similarity measure method 2.2.1.

As can be seen, energy E would be zero if for each patch from texture X could be found a perfect match in texture S. We will see later, that this definition of energy is not robust enough.

1.1.3 Guided / Non-Guided

We can distinguish between guided and non-guided synthesis. If guide images are provided and the new image is synthesized based on the content of the output guide, we call this guided texture synthesis [4]. Guided texture synthesis is needed if our sample style image is more complex.

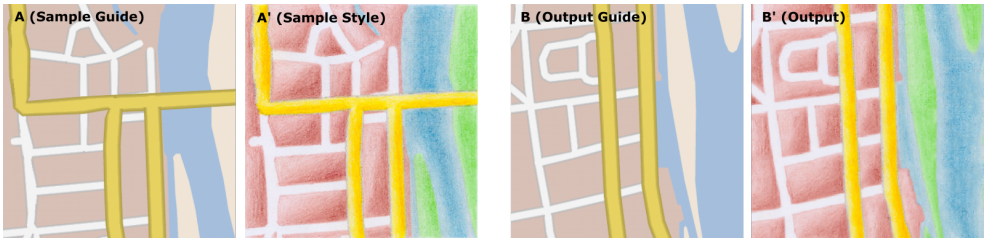


Figure 1.4: *Real demonstration of stylization of a navigation map. Both Sample Guide and Output Guide are screenshots from Dynavix. Sample Style is painted by a human and scanned. Output is a result of the synthetization algorithm. The filter or transformation used here is crayon drawing.*

If there is only sample style texture and we want to synthesize the same texture just bigger it is called normal texture synthesis or non-guided texture synthesis [19].

1.2 Image Analogies

Image Analogies is a concept which was originally defined by Aaron Hertzman [6]. It describes relation (analogy) between two pairs of images, shown on figure 1.4. We have two pairs of images, first pair is image A and image A', second pair is image B and B'. There is an analogy between the images A and B and between A' and B', meaning image A' has to image A the same relation as image B' to image B. Hertzmann in Image Analogies [6] likens the problem of synthesis to the filtering and defines that given a pair of images A and A' (where A is unfiltered and A' is a filtered source image), along with some additional unfiltered target image B, a new filtered target image B' can be synthesized. This Hertzman definition means that we have an example of some unknown filtration (in some literature the word "transformation" is used rather than "filtration") in case image A and A', we also have another unfiltered image B, given the example of filtration we can modify image B to obtain image B', which completes the analogy. In this thesis we will also refer to example image A as Sample Guide, stylized image A' as Sample Style, image B as Output Guide and stylized image B' will be called Output.

Chapter 2

Method

In this chapter, we will describe the synthesis problem and based on the previous chapter, we will define methods and terms which are used to solve the texture synthesis problem. We will also define the texture synthesis problem and its energy more formally. First, we need to describe and clarify some problems, methods and terms which we will use to formulate the problem of texture synthesis. It is important to define even really basic terms with which the reader is surely very acquainted. Because in each publication these terms may be defined a little differently.

2.1 Basic terms

Some of these terms are defined more deeply or described from the implementation point of view in the chapter Algorithm or Implementation.

Texture T, often referred to as image, is a rectangular shaped area consisted of pixels. A texture is defined by its width w and height h. A certain pixel from the texture can be referred to by its position x, y in the texture. See 2.1 for illustration. Suppose $x \leq w$ and $y \leq h$, then $T_{x,y}$ refers to the pixel.

Patch P is a square region of pixels of some width. The smallest possible patch is a patch of size 1 which is one pixel. Most of patches mentioned in this thesis are small (5, 7 or 15 pixels) and have odd width. If a patch has odd width and belongs to the texture T, we can define patch P by its width wp and its center pixel (yP, xP) in texture T. Certain pixel from the patch can be referred by its position x, y in the patch. See 2.1 for illustration. Suppose $x \leq wp$ and $y \leq wp$, then $P_{x,y}$ refers to the pixel. Later in the thesis we will define many operations with patches.

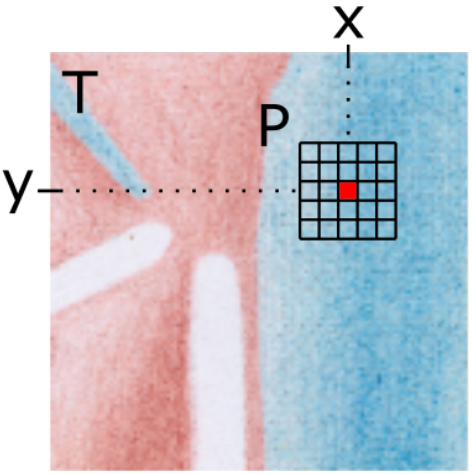


Figure 2.1: *Example of patch P in texture T. Size wp of the patch P is 5 in this case and the patch is centered around the red pixel (y, x)*

2.2 Measures of patch similarity

In computer graphics, a measure of patch similarity is a common and not a trivial problem. We will often need to measure how two patches are similar or we will need to find the most similar patch from a set of patches referring to one particular patch. The problem of similarity measuring is described in detail in [36]. We are more likely to encounter dissimilarity measuring than similarity measuring. This means we define a dissimilarity function which reaches high values if patches are different and low or zero value if patches are very similar or the same respectively. In the most literature, the term similarity is used instead of dissimilarity even if it refers to dissimilarity. In this thesis, we will not distinguish between similarity and dissimilarity and we will use only the term similarity. There exists a lot of ways to measure similarity, most common are the Sum of Absolute Differences, Sum of Squared Differences and Normalized Cross-Corelation. In the implementation of this thesis, only the Sum of Squared Differences measure method is used, so we will define only this method.

2.2.1 SSD - Sum of Squared Differences

Sum of Squared differences, often referred to as SSD, is a similarity measure between two patches of the same size. Suppose we have a patch P and a patch Q, both patches have size wp. The SSD function is defined as follows:

$$SSD(P,Q) = \sum_{x=0}^{wp-1} \sum_{y=0}^{wp-1} (P_{x,y} - Q_{x,y})^2$$

2.3 Wash-out Effect

Wash-out effect [23] is an undesired effect which occurs when a small amount of same or similar patches from the sample texture (A' in our case) are used too excessively when a new output texture (B' in our case) is synthesized. If particular patches are used too often, the output texture loses its diversity and some parts of the texture may look smoothed or blurry. Many strategies have been developed to deal with this undesired phenomenon. Between 2006 - 2008 many approaches was published, discrete e.g. solver by Han et al. [25], feature masks by Lefebvre and Hoppe [26], color histogram matching by Kopf et al. [27], and bidirectional similarity Simakov et al. [28] and Wei et al. [29]. In past few years more robust approaches was published, Kaspar et al. [30] and Jamriška et al. [24].

In this thesis, we will mitigate wash-out phenomenon using the Kaspar et al. [30] approach. It means we will penalize patches which were already used too much and prevent them from being used again. In case of guided synthesis, the content of guide images needs to be taken into account when the patch is penalized.

See 2.2 figure, there is an example of the wash-out effect on the road and a strong wash-out effect on the water area.

2.3.1 Occurrence Map

Occurrence map Ω is a structure used while creating texture *B'* from texture *A'*. Occurrence map is same in the size as image *A'* and for each pixel from *A'* it stores how many times was this pixel used in the new texture *B'*. In other words, it stores how many times each pixel from image *A'* was used in the patches that make up the output image *B'*. Let $N(s_i)$ denote the set of the pixels in a patch s_i, Kaspar et al. [30] defines the occurrence map Ω as follows:

$$\Omega(x, y) = |\{s_i | (x, y) \in N(s_i)\}|$$

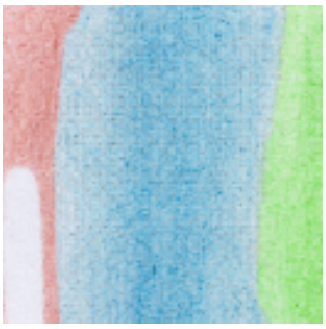
In case of not-guided texture synthesis, it is ideal if each exemplar pixel is used the same number of times. Further ω_{best} is defined and later it will be extended for guided synthesis.

$$\omega_{best} = \frac{|B'|}{|A'|} * |P|$$

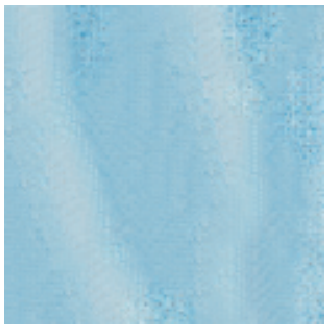
where $|P|$ is a number of pixels in patch P. This will encourage uniform distribution of patches which is suitable for normal texture synthesis but not for guided texture synthesis. In our case, the content of guide images needs to be taken into account. Suppose that there is a water area in both images A and B and $|A_{water}|$ is the size of the water area in image A and $|B_{water}|$ is the size of the water area in image B. We can define ω_{best} for water as follows:

(a) Without patch penalization

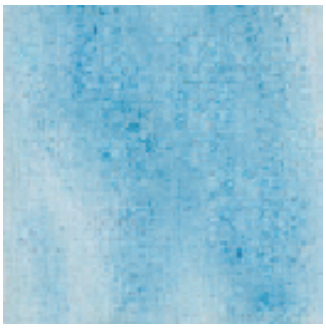
(b) With patch penalization



(c) Water style



(d) Water without patch penalization



(e) Water with patch penalization

Figure 2.2: *The wash-out effect Demonstration on the navigation map. Look at the images in the top row and focus on the yellow road of the image (a), there are visible vertical lines which are not part of the style. The yellow road at the image (a) is composed of the small amount of patches which causes the road to look vertically blurred. The bottom three images are an example of a strong wash-out effect on water. Image (c) is a sample of water style, on the image (d) a significant blur is visible. Image (e) was created using patch penalization, so there is almost no wash-out effect.*

$$\omega_{best\_water} = \frac{|B_{water}|}{|A_{water}|} * |P|$$

For each other areas (e.g. forest, road ...) ω_{best} is defined in the same way, see more details in the Algorithm chapter.

Additionaly we can also define $\Omega(s_i)$ for the entire patch as follows:

$$\Omega(s_i) = \frac{\sum_{(x,y)\in s_i} \Omega(x,y)}{|P|}$$

2.4 Nearest-neighbor Field

Once we have defined the similarity between two patches, we can define the term Nearest-neighbor Field (NNF), see illustration 2.3. It describes the relation between two textures. Let *A'* and *B'* be textures (e.g. *A'* can be our sample style texture and *B'* can be our Output texture). Let φ be some patch similarity metric (e.g, Sum of Squared Differences 2.2.1). We can define *NNF* as follows:

$$NNF_{B'\to A'}(P) = \min_{Q\in A'}[\varphi(P,Q)]$$

P is an arbitrary patch from image *B'*. *NNF* is the mapping of all patches from image *B'* to some patches of image *A'*. This mapping is not injective, multiple patches from image *B'* can be mapped to one particular patch from image *A'*.

2.4.1 Nearest-neighbor Field with Patch Penalization

In order to deal with the wash-out effect described above, we need to extend the *NNF* by occurrence map Ω and penalize and prevent particular patches to be used too often. To achieve this we extend the *SSD* similarity metric used when constructing *NNF* by patch penalization.

$$d(t_i, s_i) = ||t_i - s_i||^2 + \lambda_{occ} * \frac{\Omega(s_i)}{\omega_{best}}$$

λ_{occ} controls the relative contribution of uniformity enforcement and is set experimentally.

Later in this thesis, we will use the term NNF_Ω to refer to this extended NNF which deals with patch penalization.

2.5 Problem Formulation

The texture synthesis solution, described in this thesis, is based on [19] texture optimization approach. Texture synthesis is treated as an energy minimization

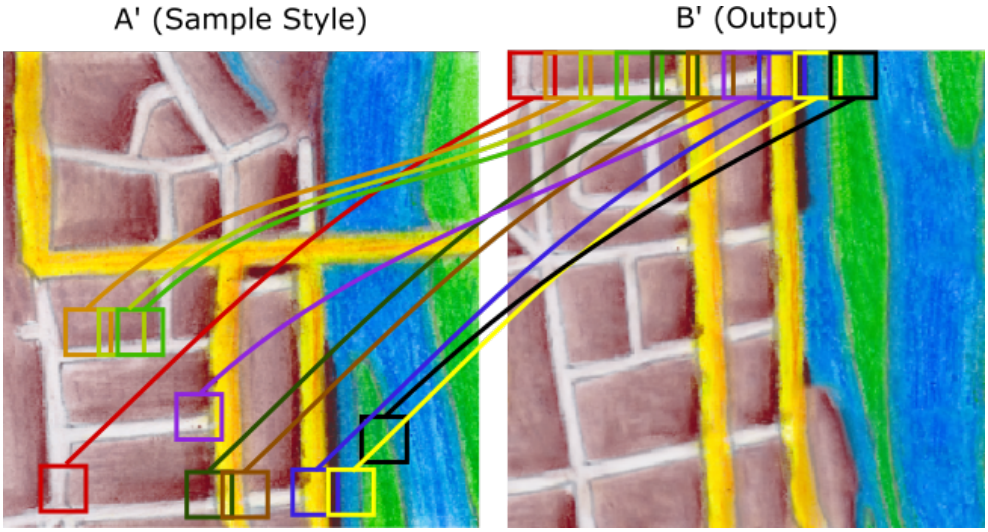


Figure 2.3: *Illustration of the Nearest-neighbor Field, in case of our texture synthesis it maps all patches from the output image B' to the sample style image A'. For simplification, the figure shows only few patches.*

problem. We will define energy exactly how Kwatra [19] defines it and we will extend it two times. At first, we will augment this definition by guide images. Second, the definition will be augmented to mitigate the wash-out effect.

Suppose we have textures denoted as in 1.4, φ is some similarity measure method (in our case it is SSD 2.2) and once we have defined NNF 2.4 we can define energy E as follows:

$$E = \sum_{s \in B'} \varphi(s, NNF^{\varphi}_{B' \to A'}(s))$$

This is the energy definition by Kwatra [19]. Since Kwatra's approach is not guided synthesis it does not deal with sample guide (A) and output guide (B). This means we have to extend this energy definition by guide images. Note that both $NNF^{\varphi}_{B' \to A'}$ and $NNF^{\varphi}_{B \to A}$ exist in a pair and contain the same information, $NNF^{\varphi}_{B' \to A'}$ returns style patch and $NNF^{\varphi}_{B \to A}$ returns guide patch. For simplification we will use NNF_s instead of $NNF^{\varphi}_{B' \to A'}(s)$ for style Nearest-neighbor Field and NNF_g for guide Nearest-neighbor Field respectively.

$$E = \sum_{s \in B} \alpha \cdot \varphi(s, NNF_s) + \beta \cdot \varphi(s, NNF_g)$$

Parameters α and β are optional and can be used to make synthesis follow more guide than style or follow more style than guide.

Kwatra's original definition of energy is not robust enough and suffers from the wash-out effect. In section 2.3 is explained what is the wash-out effect and

why it occurs. To mitigate the wash-out phenomenon we need to extend the current definition of energy in order to suppress the excessive use of particular patches.

$$E = \sum_{s \in B'} \alpha \cdot \varphi(s, NNF_s) + \beta \cdot \varphi(s, NNF_g) + \lambda \cdot \Omega(NNF_g)$$

Our energy definition is now a sum of squared differences in style plus a sum of squared differences in guide plus patch penalization. Patch penalization Ω will encourage that patches from sample style will be used with respect to areas in guide images. Parameter λ determines how big is the penalization for patches which are used more often than they should be.

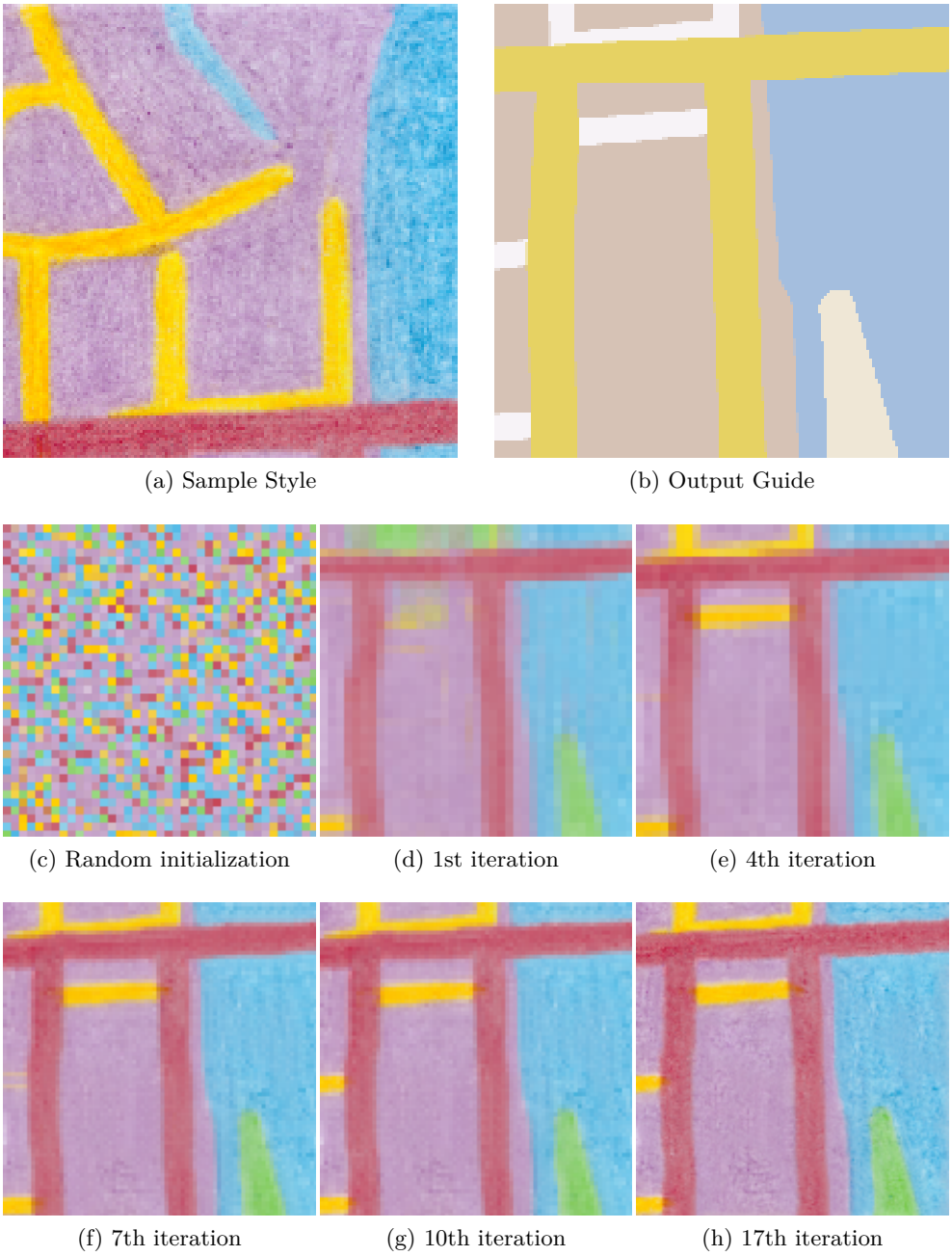
Chapter 3

Algorithm

Based on the terms, methods and problem formulation we have already defined, we can now construct an algorithm, solving the guided texture synthesis problem. This algorithm is based on Kwatra's approach [19] extended by Kaspar's [30] patch penalization. We will construct the basic algorithm for texture synthesis and extend it by the Multi-resolution approach. The chapter contains also a description of guidance on navigation maps and pre-synthesized texture improvement.

3.1 Texture Synthesis Algorithm

This is a global optimization problem solved in multiple iterations. At first, our Output image is initialized by random colors and then is refined in multiple iterations to a fine and faithfully looking texture, see image 3.1, this iterative approach is in some literature called "coarse-to-finite". In each iteration, *NNF* is computed and each pixel of the new output texture is evaluated based on this *NNF* using the Voting method 3.2. During iterations, each next subsequent output image should have lower energy (defined in chapter above) therefore should look more like style sample but with content defined in the output guide. See the algorithm TextureOptimization 1. There are four input parameters, sample guide A, output guide B, sample style A' and number of iterations N. In this algorithm, function *findNNF* takes four parameters, both guide images, sample and output image. It means that the distance between two patches is computed using two *SSD* computations, *SSD* distance between image A and A' plus *SSD* distance between image B and B'. To deal with the undesired wash-out effect we were discusing in 2.3, the function *findNNF* uses extended *NNF*, described in 2.4.1.



(a) Sample Style (b) Output Guide

(c) Random initialization (d) 1st iteration (e) 4th iteration

(f) 7th iteration (g) 10th iteration (h) 17th iteration

Figure 3.1: *Coarse-to-finite illustration on real map synthetization example. The result is initialized by random colors and then it is refined in multiple iterations. It can be seen that the content of the output guide image is captured in the first few iterations. Then the style of Sample Style and the texture appearance is captured.*

Algorithm 1 *Texture Synthesis Basic Algotrithm*

```
function TextureOptimization(A, B, A', N)
for n = 0 : N do
  if n == 0 then
    NNF = random NNF
  else
    NNF = findNNF(A, B, A', B')
  end if
  B' = voting(NNF, A')
end for
return B'
end function
```

3.2 Voting method

For simplification, suppose we are solving non-guided texture synthesis, therefore, we only have images *A'* (sample style) and *B'* (output). *NNF* describes the relation between these two images - for every patch in image *B'*, *NNF* gives a patch in image *A'* which is similar the most. Let *NNF* be created using a patch of size *wp*, then each pixel in image *B'* is contained in Wp^2 patches in image *A'*, so this amount of patches (its center pixels respectively) will "vote" to assess the value of the new pixel. Such voting is used to evaluate every pixel in the new texture *B'*. See example 3.2 of voting for one pixel.

3.3 Multi-resolution approach

Multi-resolution approach gives us a significant improvement in both speed and quality of texture synthesis. Sample texture, as well as both guide textures, are downsampled to half of their original resolution a few times. Synthetization then starts from the lowest resolution. See figure 3.3. Multiple iterations are performed on each resolution level. When transiting to higher resolution level, output texture is upsampled. See pseudocode of texture synthesis augmented by the multi-resolution approach 2. There are five input parameters, sample guide *A*, output guide *B*, sample style *A'*, number of iterations on each resolution level *N* and number of resolution levels *LVLS*. The advantage of having sample textures in multiple resolutions is that we can use Example-based Upsample method 4.5 for upsampling, see details in the section below. Next, we will discuss details of speed and quality improvement which the Multi-resolution approach offers.

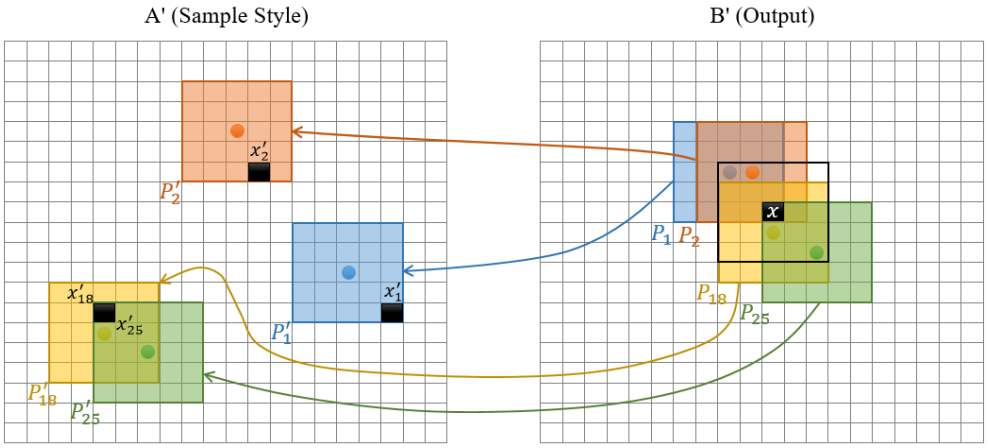


Figure 3.2: *Example of pixel evaluation using the Voting method. Let NNF between output texture B' and sample style texture A' be computed, see arrows in the figure. Patches have size 5 so pixel x in image B' is evaluated based on 25 patches (pixels) from image A'. Pixel x is evaluated as an arithmetic mean of pixels* x'_1 *...* x'_{25}*, see the simplified example with four patches and four pixels* x'_1*,* x'_2*,* x'_{18} *and* x'_{25} *at this figure.*

Multi-resolution approach - Speed improvement

Finding *NNF* is a very time-consuming problem, it has complexity $O(|A| \cdot |B|)$ where $|A|$ is a number of pixels of the sample image and $|B|$ is a number of pixels of the output image. In each iteration, *NNF* has to be computed. Performing n iterations on m cascading resolution levels and upsampling the output image between resolution levels is significantly less time-consuming than performing $n \cdot m$ iterations on the original resolution level.

Multi-resolution approach - Quality improvement

Beside speed improvement, the multi-resolution approach gives us also a significant quality improvement. Suppose using the same size of the patch during constructing NNF at each resolution level. On lower resolution levels, texture synthesis is able to capture bigger image features, because the ratio of image size to patch size is lower. It also decreases the impact of the randomly generated initial *NNF*.

3.4 Upsampling

There is a significant advantage to performing texture synthesis on multiple resolution levels and transitioning from a lower resolution level to a higher resolution level needs upsampling. The inherit problem of upsampling is that

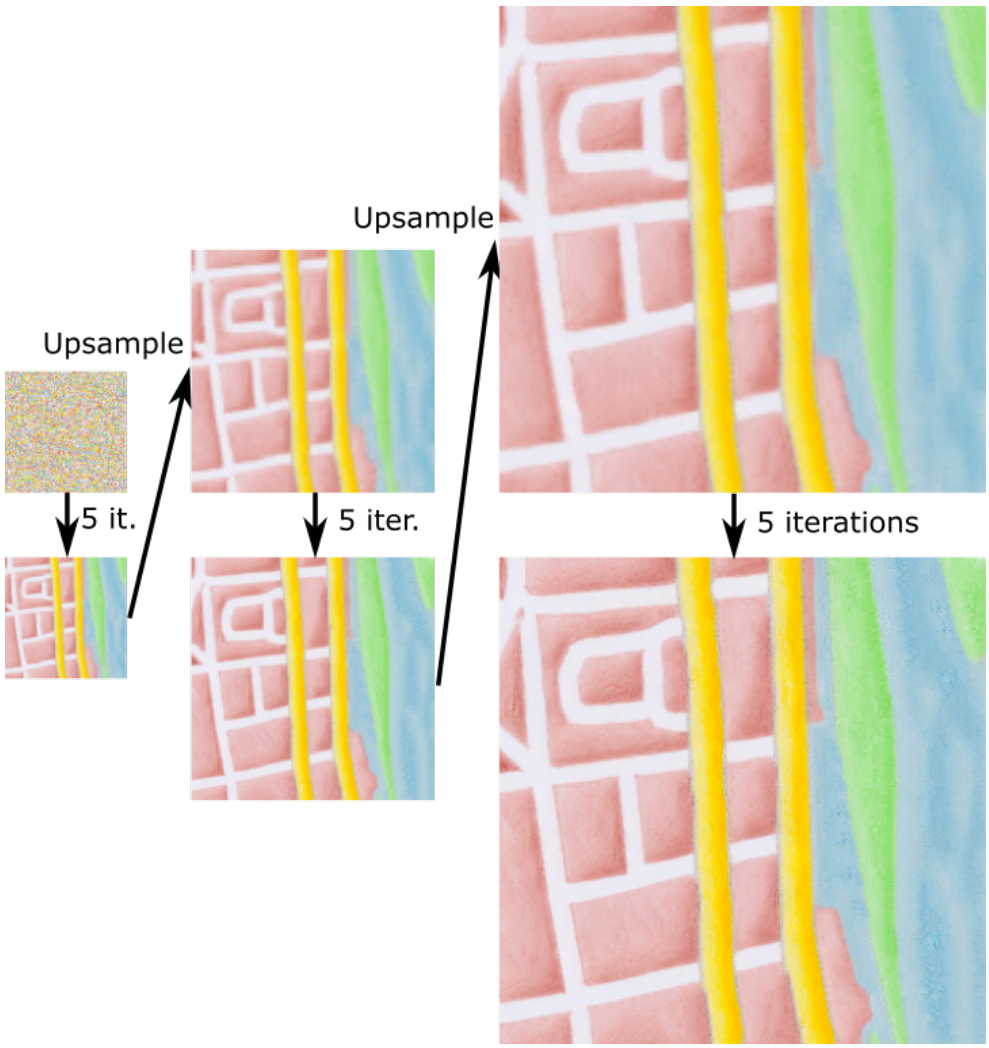


Figure 3.3: *The multi-resolution approach illustration on a real map example, there are three resolution levels and five iterations on each level. Synthesis starts from random initialization on quarter resolution, five iterations are performed and the result is upsampled, five iterations on half resolution are performed and the image is upsampled once more and the final five iterations on full resolutions are performed.*

Algorithm 2 *Texture Synthesis - Multi-Resolution*

function TextureOptimizationMultiResolution(A, B, A', N, LVLS)
for lvl = 0 : LVLS **do**
$A_{\downarrow} = Downsample(A, 2^{LVLS-lvl})$
$B_{\downarrow} = Downsample(B, 2^{LVLS-lvl})$
$A'_{\downarrow} = Downsample(A', 2^{LVLS-lvl})$
for n = 0 : N **do**
if n == 0 **then**
NNF = random NNF
else
$NNF = findNNF(A_{\downarrow}, B_{\downarrow}, A'_{\downarrow}, B')$
end if
$B' = voting(NNF, A'_{\downarrow})$
end for
if lvl < LVLS **then**
$B' = Upsample(B', NNF)$
end if
end for
return B'
end function

we need to add new pixels into the image to make it bigger; so we need to determinate the color of those pixels in order to preserve the natural appearance of the image. There are multiple common upsampling methods such as the Nearest-Pixel Interpolation, Linear Interpolation or Example-based Upsampling. In case of Nearest-Pixel Interpolation, the image is enlarged, which results in empty pixels and the color of these pixels is set to the color of the nearest original pixel. But upsampled image looks coarse. A similar method is Linear Interpolation, the image is enlarged so that there are empty pixels as in the previous method. The color of these empty pixels is computed as an aritmethic average of two or four nearest original pixels. In this case, an upsampled image does not look coarse but is blurry. The different and more sophisticated approach is the Example-based Upsample, which yields results of significantly better quality than the previous two methods. Since this method requires additional information about the image, it is not suitable in all cases, but in case of multi-scale texture synthesis, we have this information. Next is a subchapter describing the Example-based upsample method more in depth.

3.4.1 Example-based Upsample

Assume that we have two in content and appearance same sample textures. One sample texture of the same resolution as a smaller image and one sample texture of the same resolution as is our desired size. These sample textures

must capture all the important elements and characteristics of the texture we want to upsample. Then we can match the smaller image to the lower resolution sample texture using NNF 2.4. Once *NNF* is computed it can be upsampled - all positions in *NNF* are multiplied by two. Based on this upsampled *NNF* and a higher resolution sample texture, a bigger image can be reconstructed. This upsampling method has significantly better results than the Nearest-Pixel Interpolation or Linear Interpolation, but it requires sample textures which is the big limitation. This upsample approach is also often referred to as the NNF-upsampling.

3.5 Map Segments Guidance

Since we do not synthesize only one texture but the entire image with more complex content, guidance is needed to distinguish between different parts of the synthesized image. A computer-generated map is usually consisted only from a small amount of regions and therefore only from a small amount of colors. Image 1.4 is consisted from five different colors - blue color for water, yellow color for bigger roads, white color for smaller roads, pink color for built-up areas and sand color for grass areas. This inherit segmentation of computer-generated maps is ideally suited to be used as the guide for synthesis. Later in this section, the term "segment" is used to refer to a map component like water, road, built-up area, etc.

In section 2.4.1, we have defined the distance between two patches using an occurrence map and ω_{best}. Finding a good value of ω_{best} in general case is a very complex problem. In our case, ω_{best} can be computed separately for each segment as follows:

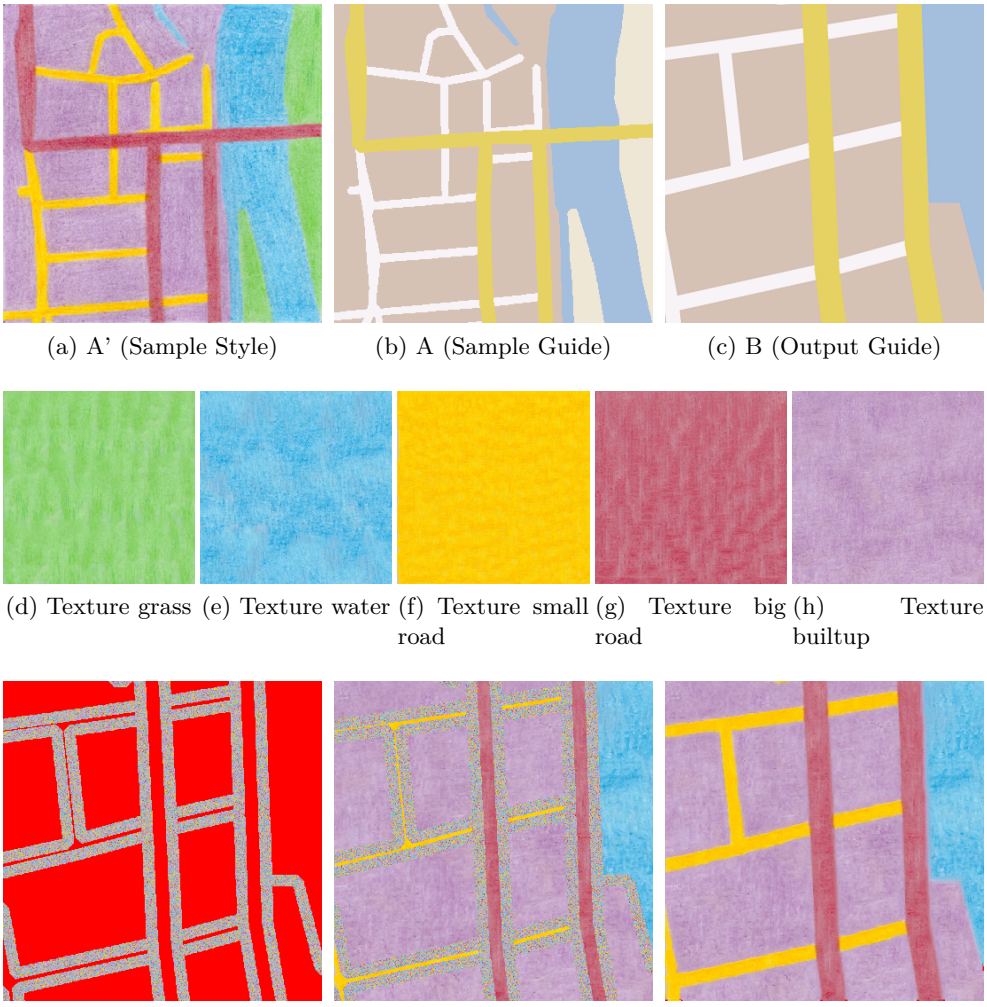
$$\omega_{best}(segment) = \frac{output[segment].size}{sample[segment].size}$$

If ω_{best} is set correctly, function *findNNF* will probably, for most of the water patches from output guide, also find water patches in sample guide. But it might not work every time, suppose that in our guide images A and B, the color of one segment is similar to the color of a different segment, for example, water has a light-blue color and road has a dark-blue color. Also, assume that *findNNF* already assigned a lot of water patches to water patches thus all the water patches in sample guide already have a big occurrence value. Even if ω_{best} is correct for each segment, it can happen that for a light-blue water patch from output guide *findNNF* will assign a patch from the dark-blue road segment in the sample guide because this dark-blue patch has little different color but it has low occurrence. We can get significant quality and also speed improvement by setting a strong requirement for the *findNNF* function to match patches only between the same segments. It means, for example, for

a patch from output guide located on water, the *findNNF* function will find the closest patch only from water patches in the sample image.

3.6 Pre-Synthesized Textures Speed-Up

As we describe in the section above, computer-generated maps consist of a small amount of segments, therefore, there is a possibility to pre-generate the texture of each segment and use these textures to speed-up the synthetization. The output image is initially composed of pre-generated textures which results in visible seams and sharp borders between different segments. Since the texture synthesis approach used in this thesis is the global optimization method, this coarse composed image is then optimized in multiple iterations in order to remove seams and make borders between segments faithfully-looking. Synthesis is applied only around seam areas and borders. See detailed illustration on 3.4.



(a) A' (Sample Style) (b) A (Sample Guide) (c) B (Output Guide)

(d) Texture grass (e) Texture water (f) Texture small road (g) Texture big road (h) Texture builtup

(i) Mask (j) Initialization (k) Final result

Figure 3.4: *Illustration of the Pre-Synthesized Textures Speed-Up on the real example. At the top row, input images A', A and B are shown. The middle row contains pre-synthesized textures. At first, a mask around the edges of the Output Guide B is computed, see image (i). Areas near to the edges are initialized by random colors, areas far from the edges are initialized by pre-synthesized textures based on the content of Output Guide B, see image (j). Only areas around the edges are then synthesized. See (k) for the final result.*

Chapter 4

Implementation

In this chapter, we will describe the implementation of the previously mentioned algorithms more in depth and with more technical details, pseudocode examples are provided. We will start from the basic functions and compose a texture synthesis algorithm step-by-step. The chapter also contains one section dealing with graphic card acceleration improvement. Pseudocodes of kernels written in the OpenCL [37] standard are provided as well.

4.1 SSD function

In this implementation, we use the SSD - Sum of Square Differences 2.2.1 method for the measure of patch similarity. Since this is guided synthesis, the sum of squared differences is computed based on two style patches (A and B) and two guide patches (A' and B'). While constructing NNF, the SSD function is called multiple times in order to find the best mapping between the patches. A significant speed-up is achieved if the optional parameter BestSSD is provided to early terminate SSD comparison if a better match is already known. Additional optional parameters α and β are used to make the SSD comparison function more sensitive to the difference in style or in guide respectively. The ideal setting of α and β is a complex problem and depends on the particular style, it was empirically found that the ideal ratio is $\frac{\alpha}{\beta} = \frac{0.3}{0.7}$ for most of the styles.

See pseudocode of SSD function 3. There are seven input parameters, a patch from image A, B, A' and B', optional parameter BestSSD, α and β which are described above.

4.2 Find nearest neighbour

Given a patch from output image B and output guide image B', this function finds its nearest patch in image A, A' respectively. See pseudocode 4. The

Algorithm 3 *SSD Similarity method for guided texture synthesis*

function SSD(A, B, A', B', BestSSD = inf, $\alpha = 0.3$, $\beta = 0.7$)
SSDTotal = 0
for [x, y] in A.pixels.position **do**
  SSDStyle = $(A_{x,y} - A'_{x,y})^2$
  SSDGuide = $(B_{x,y} - B'_{x,y})^2$
  SSDValue = α·SSDStyle + β·SSDGuide
  if BestSSD $\leq$ SSDTotal **then**
    return bestSSD
  end if
end for
return SSDTotal
end function

occurrence map Ω is taken into account. SampleMask is a mask over the sample guide image, describing where to look for patches. Using this mask, a significant speed-up and quality improvement is achieved. For example, if we have a patch of a water region, using this mask we will search in the sample image only where the water is. See section Map Segments Guidance 3.5 for details. The function, takes six parameters, patch from output guide *B* and output *B'*, sample guide *A* and sample style *A'*, sample mask and omega for patch penalization computing.

Algorithm 4 *Find nearest neighbour*

function FindNearestNeighbour(PatchB, PatchB', A, A', SampleMask, Omega)
bestOccurrence = find best occurrence for PatchB
bestSSD = -inf
bestPosition = NULL
for [x,y] in SampleMask **do**
  PatchA = create patch around pixel $A_{x,y}$
  PatchA' = create patch around pixel $A'_{x,y}$
  currentSSD = SSD(PatchA, PatchB, PatchA', PatchB', bestSSD, 0.3, 0.7) + (Omega[x,y] bestOccurrence)
  if currentSSD < bestSSD **then**
    bestSSD = currentSSD
    bestPosition = [x,y]
  end if
end for
return bestPosition
end function

4.3 Find NNF

Function *findNNF*, see pseudocode 5, finds the nearest neighbour field between two images. In our case, it is used to find the NNF mapping from output guide B and output guide B' to sample style A and sample guide A'. NNF is found with respect to patch occurrence penalization. OutputMask specifies the region for which we need to find NNF.

Algorithm 5 *Find nearest neighbour field*

```
function FindNNF(A, A', B, B', SampleMask, OutputMask)
NNF = NULL
Occurrence = init with zero
for [x,y] in OutputMask do
  PatchB = create patch around pixel B_{x,y}
  PatchB' = create patch around pixel B'_{x,y}
  NNF[x,y] = FindNearestNeighbour(PatchB, PatchB', A, A',
  SampleMask, Occurrence)
  Occurrence[NNF[x,y]] = Occurrence[NNF[x,y]] + 1
end for
return NNF
end function
```

4.4 Voting method

Based on the nearest neighbor field, the new image is composed using the Voting method. Once we have computed NNF which maps each patch from output image B' to the closest patch in sample style image A', we take these patches from A' and merge them together. See pseudocode 6. At first, the StackedImagePixels structure is computed, it is a structure similar to an image but with multiple pixels at each position. Provided NNF does not have to cover the entire output image if there was an Output mask during NNF computing so StackedImagePixels is initialized with output image B'. All pixels of all patches from NNF are added to this StackedImagePixels structure so there is up to $PatchSize^2$ pixels at a single position. Then a new OutputImage is computed as an average of stacked pixels. The Voting method function takes a sample style image A', nearest neighbour field and patch size *pSize* with which NNF was computed. Function *avg* computes an arithmetic mean from the provided list of values.

Algorithm 6 *Voting method*

```
function Voting(A', B', NNF, pSize)
StackedImagePixels = init with B'
for x = 0 : NNF.size.width do
  for y = 0 : NNF.size.height do
    for innerX = NNF[x,y].x - (pSize/2) : NNF[x,y].x + (pSize/2) do
      for innerY = NNF[x,y].y - (pSize/2) : NNF[x,y].y + (pSize/2) do
        stackedX = x + innerX - NNF[x, y].x
        stackedY = y + innerY - NNF[x, y].y
        StackedImagePixels[stackedX, stackedY].add(A'[innerX, innerY])
      end for
    end for
  end for
end for
OutputImage = allocate
for x = 0 : StackedImagePixels.size.width do
  for y = 0 : StackedImagePixels.size.height do
    OutputImage[x,y] = avg(StackedImagePixels[x,y])
  end for
end for
return OutputImage
end function
```

4.5 Texture Synthesis

Given *A*, *B* and *A'*, the output image *B'* is created in the TextureSynthesis function, see pseudocode 7. OutputImage is initialized using pre-synthesized textures based on the output guide. Output image is filled with pre-synthesized textures and only seams are synthesized, SampleMask and OutputMask describe the location of mentioned seams. Argument *N* is the number of iterations on each resolution level *LVLS*.

At first, images *A*, *B* and *A'* are downsampled to the desired resolution level. Based on guide *A* and *B*, a mask is created so that it covers only the location where segments change, e.g. transition from a water segment to a grass segment. Output image *B'* is initialized with pre-synthesized textures in the first iteration or reinitialized with pre-synthesized textures of higher resolution. This initialization respects output guide *B*, so that space around seams is not reinitialized again once it have been synthesized. *FindNNF* and the *Voting* function are applied in each iteration and *Upsample* refers to the Example-based upsampling .

Based on consideration and experimental evaluation, we found that the ideal value for most of the styles is 5 for a number of iterations on each level and 3 for a number of resolution levels. We also consider size 5 of a patch to be

Algorithm 7 *Texture Synthesis*

```
function TextureSynthesis(A, B, A', N, LVLS)
for lvl = 0 : LVLS do
  A↓ = Downsample(A, 2^(LVLS−lvl))
  B↓ = Downsample(B, 2^(LVLS−lvl))
  A'↓ = Downsample(A', 2^(LVLS−lvl))
  SampleMask = CreateMask(A)
  OutputMask = CreateMask(B)
  B' = InitWithPre-SynthesizedTextures(B, OutputMask)
  for n = 0 : N do
    NNF = FindNNF(A, A', B, B', SampleMask, OutputMask)
    B' = Voting(A', B', NNF, 5)
  end for
  if lvl < LVLS then
    B' = Upsample(B', NNF)
  end if
end for
return B'
end function
```

ideal. If the patch size is bigger, synthesis is able to preserve larger structures, but since we do not change the size of a patch for lower resolutions, large-scale structures are captured on lower resolution levels, e.g. on quarter resolution a patch of size 5 has a relative size 20, in comparison to the original image.

4.6 Parallel acceleration

From algorithms described in this and previous chapters, it is obvious that the most time-consuming part of texture synthesis is finding the Nearest-Neighbor Field. Texture synthesis runs in multiple iterations on multiple resolution levels and in each iteration, NNF is computed. NNF matches each patch from output texture *B'* to the closest patch (with respect to the patch penalization) in sample style *A'*. In our case NNF is computed exactly by using a brute-force-like algorithm so for each patch from *B'* the entire image *A'* is searched. Suppose that the width and height of both images *A'* and *B'* is n, that means that complexity of the NNF computing is $O(n^4)$. Other parts of the texture synthesis algorithm have significantly less complexity. By accelerating the NNF, the entire texture synthesis will be accelerated.

For simplification, suppose we are not dealing with patch penalization, meaning this brute-force-like NNF algorithm is ideally suited for parallelization. For each patch from image *B'*, texture *A'* can be searched independently. During computing of the NNF, texture *A'* is accessed only for reading, so there

is no need for any explicit synchronization and finding NNF can be naturally parallelized. Next in this section, we will deal with parallelization of the NNF.

4.6.1 OpenCL

At first, we will introduce the OpenCL [37] standard briefly. Then, we will follow with parallel functions used in texture synthesis.

OpenCL stands for Open Computing Language. It is an open standard for parallel programming of heterogeneous systems and is maintained by a non-profit technology consortium Khronos Group. It runs on central processing units (CPUs), graphic processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and many other hardware accelerators. In OpenCL terminology, by *device* or *OpenCL device* is meant a hardware accelerator (e.g. CPU, GPU) where a parallel program is executed. By term *host* is meant a processor, which calls a parallel program on the *OpenCL device*. *Kernel* is a term for functions executed on an *OpenCL device*. OpenCL defines a C-like programming language for writing kernels. Kernels are compiled at run-time thus OpenCL code is ideally portable between implementations for various host devices. In the OpenCL standard, there are host APIs only for C and C++, but there also exist third-party implementations or wrappers for Python, Java, .NET and others. Nowadays, the OpenCL standard is adopted and implemented by various companies such as AMD, Apple, ARM, Intel, Nvidia, Samsung and many others, meaning OpenCL programs can run on Windows, Linux, OSX and even on some Android devices.

4.6.2 SSD OpenCL Kernel

Following is the kernel for computing sum of squared differences. For simplification, this pseudocode deals only with one image, it can be used for non-guided synthesis as is or naturally extended by another image and patch. Input arguments are image I, which can be in our case sample the image A', patch P is a patch which will be found in image I, width *imgWidth* and height *imgHeight* of image I and size of patch *pSize*. Last is the output parameter *outputSSDData*, which stores all the computed SSDs. See pseudocode 8. The input image I is serialized to a one-dimensional array. Function *get_global_id(0)* is an OpenCL API function returning an index of the thread, this index is used to determine which data should this particular thread take care of.

4.6.3 Find NNF OpenCL Kernel

Another kernel used in the implementation of this thesis is a kernel for finding the nearest neighbour field, see pseudocode 9.

Algorithm 8 *SSD kernel function*

```
kernel function SSD_KERNEL(I, P, imgWidth, imgHeight, pSize, outputSSDData)
patchHalf = patchsize / 2
row = get_global_id(0) / (imgWidth - 2·patchHalf)
col = get_global_id(0) % (imgWidth - 2·patchHalf)
ssd = 0
for innerRow = 0 : patchSize do
  for innerCol = 0 : patchSize do
    imgIndex = [(innerCol + col) + (innerRow + row)·imgWidth] ·3
    patchIndex = innerCol·3 + innerRow·pSize·3
    diffB = I[imgIndex] - P[patchIndex]
    diffG = I[imgIndex + 1] - P[patchIndex + 1]
    diffR = I[imgIndex + 2] - P[patchIndex + 2]
    ssd = ssd + diffB·diffB + diffG·diffG + diffR·diffR
  end for
end for
outputSSDData[get_global_id(0)] = ssd
end kernel function
```

4.7 Dynavix Integration

Implementation of this thesis was integrated into an existing mobile navigation Dynavix [5] as the prototype. Dynavix offers GPS navigation and maps for Android devices. In this section, we will describe the details of this integration.

During the startup of Dynavix, an instance of the class TextureSynthesis is created. Two methods used to initialize the TextureSynthesis object are called, i.e. *loadSampleAndSampleGuide* and *loadPreSynthesizedTextures*. These methods load input images, needed by texture synthesis, into memory and prepare them to be used. Method *loadSampleAndSampleGuide* takes the path to sample and the guide image as an argument, it loads sample style image *A'* and sample guide image *A*, downsamples them to half and quarter resolution and stores them into the TextureSynthesis object as OpenCV [38] images. Method *loadPreSynthesizedTextures* loads presynthesized textures, downsamples them and store them into the TextureSynthesis object as well.

Dynavix uses OpenGL technology for map and map widgets rendering and the Android Framework for the rest of its user interface. In the first phase, a map using OpenGL is rendered. In the second phase, map widgets are rendered over the previously rendered map, also by OpenGL. In the third phase, the remaining Android UI is rendered over it. The prototype of map stylization was inserted between the first and the second phase.

See pseudocode 10 of initializing and running a map stylization in Dynavix.

Algorithm 9 *FindNNF kernel function*

```
kernel function FindNNF_KERNEL(B', A', pSize, A, B, outNNFRow,
outNNFCol, outImgSize, sampleSize)
pHalf = pSize / 2
outImgRow = (get_global_id(0) / (outImgSize - 2 * pHalf)) + pHalf
outImgCol = (get_global_id(0) % (outImgSize - 2 * pHalf)) + pHalf
bestSSD = +inf
for [sampleRow, sampleCol] : sample.pixels.position do
  SSD = 0
  for innerRow = 0 : patchsize -1 do
    for innerCol = 0 : patchsize -1 do
      currOutRow = outImgRow + innerRow - pHalf;
      currOutCol = outImgCol + innerCol - pHalf;
      currSmpRow = sampleRow + innerRow - pHalf;
      currSmpCol = sampleCol + innerCol - pHalf;
      indexOutImg = currOutRow * outImgSize * 3 + currOutCol * 3;
      indexSampleImg = currSmpRow * sampleSize * 3 + currSmpCol *
      3;

      diffB = A'[indexSampleImg] - B'[indexOutImg];
      diffG = A'[indexSampleImg+1] - B'[indexOutImg+1];
      diffR = A'[indexSampleImg+2] - B'[indexOutImg+2];
      squareSum = diffB * diffB + diffG * diffG + diffR * diffR;

      diffB = A[indexSampleImg] - B[indexOutImg];
      diffG = A[indexSampleImg+1] - B[indexOutImg+1];
      diffR = A[indexSampleImg+2] - B[indexOutImg+2];
      squareSumGuide = diffB * diffB + diffG * diffG + diffR * diffR;

      SSD = SSD + squareSum + squareSumGuide;
    end for
  end for
  if SSD < bestSSD then
    bestSSD = SSD;
    index = (outImgRow - pHalf) * (outImgSize - 2 * pHalf) + (outImgCol
    - pHalf);
    outNNFRow[index] = sampleRow;
    outNNFCol[index] = sampleCol;
  end if
end for
end kernel function
```

Algorithm 10 *Dynavix Main*

```
function DynavixMain()
textureSynthesis = create TextureSynthesis object
textureSynthesis.loadSampleAndSampleGuide(...)
textureSynthesis.loadPreSynthesizedTextures(...)
...
while Dynavix is running do
  Render Map
  B = Read pixels from OpenGL bufffer
  B' = textureSynthesis.synthesis(B)
  Write B' to OpenGL buffer
  Render Map Widgets
  Render Android Framework UI
end while
end function
```

Chapter 5

Results and Comparison

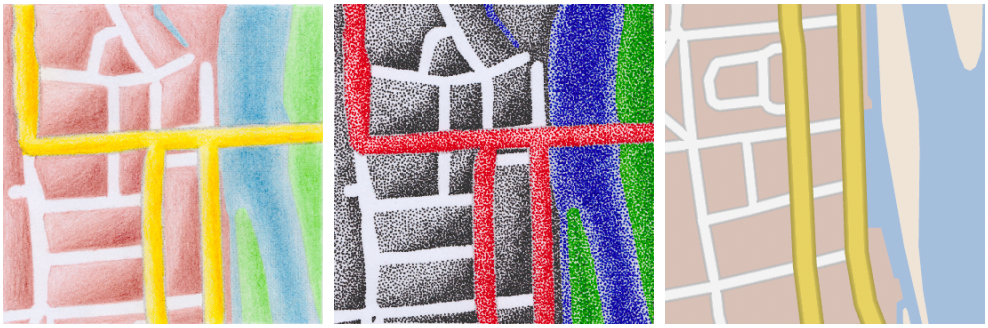
At first, in this chapter, we will compare the quality of this synthetsis implementation with nowadays very popupar convolutional neural networks. Next are some experiments with the Pre-Synthesized texture improvement, OpenCL acceleration speed-up and experiments with loss-less zooming. Many images are presented here.

5.1 Comparison with CNN approaches

In past years, there has been a big expansion of the machine learning, deep learning and neural networks. These technologies expanded into almost all of the corners of computer science, including texture synthesis and stylization. Convolution neural networks have impressive results if they are used for stylization of complex paintings. In case of simple style and simple guide, most of neural network approaches do not work well.

Next, in this section, we will compare the quality of the results of our synthesis implementation with three neural approaches. First is the "Semantic Style Transfer and Turning Two-Bit Doodles into Fine Artwork" [39] (later reffered to only as Neural Doodle), second is "A Neural Algorithm of Artistic Style" [32] (later reffered to only as Artistic Style) and third is Deepart.io [40]. Implementation of the Neural Doodle and the Artistic Style is available as python code and the Deepart.io has a web page application.

Figure 5.1 shows input images used in comparison. There is the output guide, a crayon style sample and a pen style sample. The first is a comparison of three previously mentioned convolutional neural network approaches and is performed on the crayon style, see 5.2. As can be seen, our result is significantly better than any other result. According to the Deepart.io web application [40], it has good results when the sample style and output guide are more complex, however in our simple map scenario, Deepart.io results are insufficient. Artistic Style completely failed to capture colors, while Neural Doodle failed to capture content, showing the sample style content instead of



(a) A' Crayon (b) A' Pen (c) B output guide

Figure 5.1: *Comparison with CNN - input images for comparison, results are shown on figure 5.3 and 5.2*

the output guide content. See another comparison 5.3 performed on the pen style. Results are similar as on the previous comparsion. Moreover, on this style Neural Doodle failed to preserve the content, the bottom third of the image has red roads same as output guide, but the rest of the image is wrong. Our result shows three little visible green glitches on the white roads. It can be fixed by setting α and β parameters of the SSD measure to make synthesis follow more guide than style.

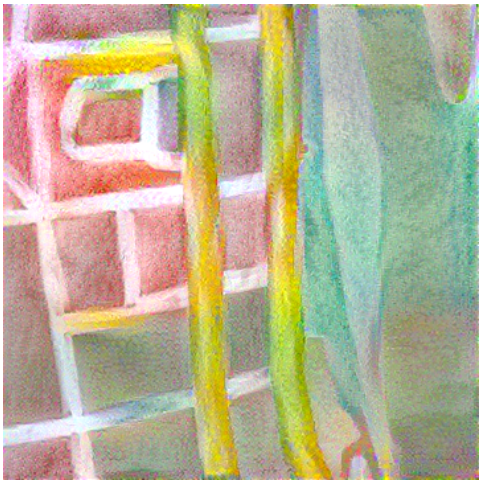
Comparison of two more styles of our implementation with only Deepart.io is shown on figure 5.4. The chalk style is a scan of the real image painted with chalk. MS Paint style is a style, where some parts of the image were repainted by a brush in the Microsoft Paint software.

5.2 Pre-Synthesized textures experiments

In this section, some results and experiments performed on Pre-Synthesized textures improvement are presented.

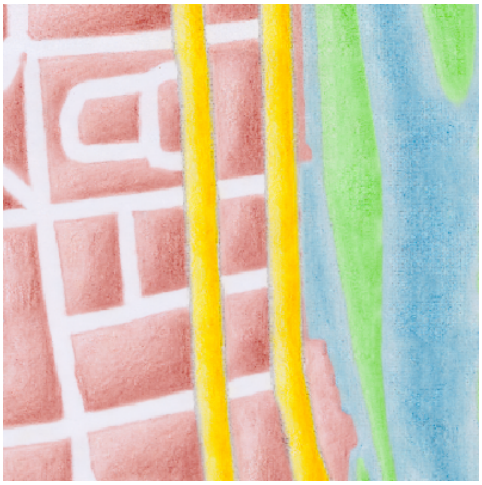
In case of the Pre-Syntheiszed improvement, the output image is initialized with pre-generated textures and only areas around seams (e.g. edges in the output guide) are synthesized. Quality of the result is given by the size of the synthesized area around the edges. If the area around the edges has the size of 0, nothing is synthesized and the resulting image is composed only from pre-synthesized textures. If the area around the edges has a size bigger than width or height of the image, the whole texture is synthesized and it is not Pre-Synthesized improvement anymore. At figure 5.5, see multiple results for different sizes of the synthesized area around the edges. As can be seen, results (a), (b) and (c) look almost the same, meaning we need to synthesize only a really small area around the edges to get a faithfully-looking result.

(a) Deepart.io



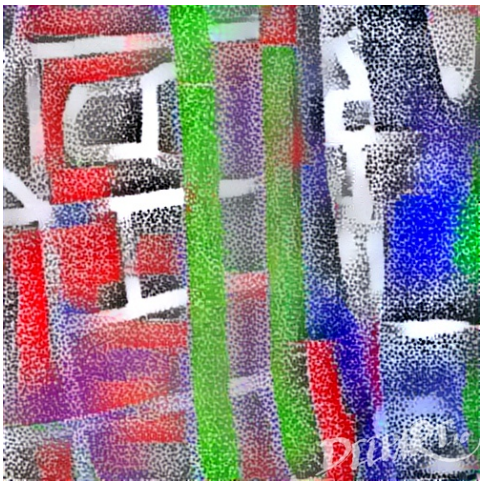
(b) Artistic Style

(c) Neural Doodle

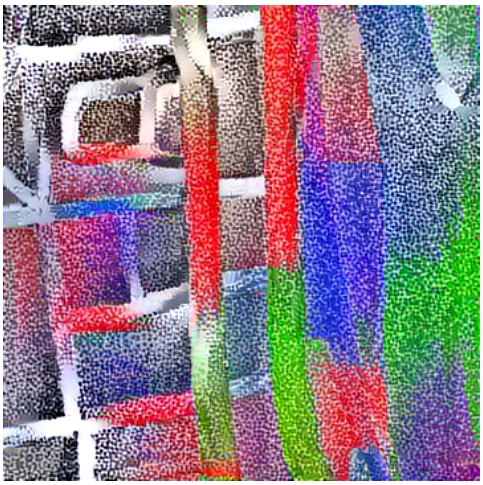


(d) Ours

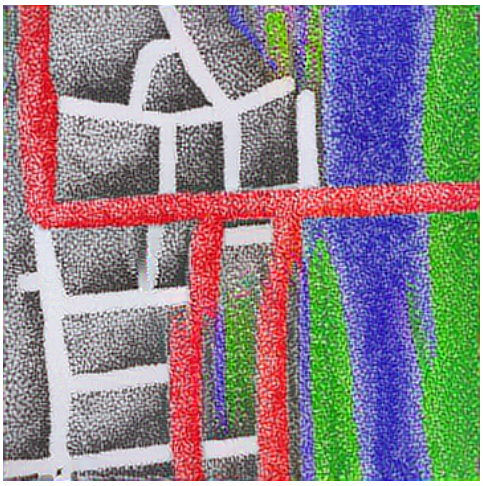
Figure 5.2: *Comparison with CNN - crayon style, input images are shown on figure 5.1. The image (a) shows result of Deepart.io, image (b) result of Artistic Style, image (c) result of Neural Doodle and the last image (d) is our result.*



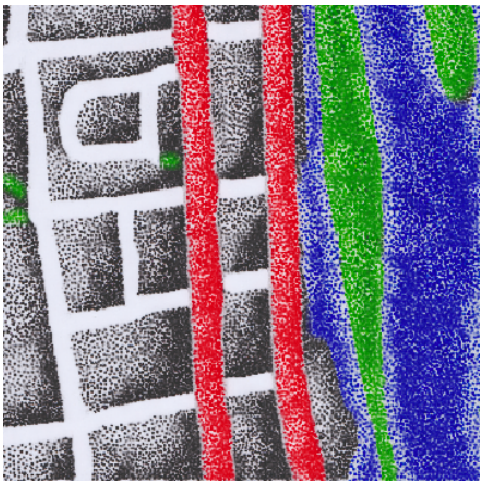
(a) Deepart.io



(b) Artistic Style



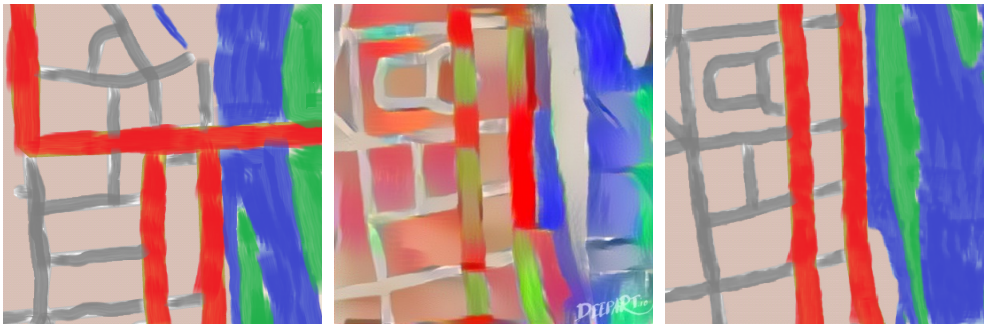
(c) Neural Doodle



(d) Ours

Figure 5.3: *Comparison with CNN - pen style, input images are shown on figure 5.1. The image (a) shows result of Deepart.io, image (b) result of Artistic Style, image (c) result of Neural Doodle and the last image (d) is our result.*

(a) Chalk style (b) Deepart.io (c) Ours



(d) MS Paint style (e) Deepart.io (f) Ours

Figure 5.4: *Deepart.io comparison with ours. Chalk style and style created using MS Paint.*

Table 5.1: *OpenCL NNF speed-up*

	Single thread CPU Intel i7-4700MQ	**GPU GeForce GT 745M (ms)**
360 x 360 pixels	14 000 ms ($\pm 200ms$)	1 750 ms ($\pm 50ms$)
160 x 160 pixels	2 180 ms ($\pm 30ms$)	460 ms ($\pm 20ms$)

5.3 OpenCL acceleration

In our implementation, finding the NNF is parallelized using OpenCL. Table 5.1 contains the speed-up results. Time of finding one NNF on a certain resolution was measured. First is the time of single thread NNF computing on the CPU, second is the time of NNF computing on the GPU. As can be seen, GPU implementation is approximately eight times more effective.

(a) Full synthesis area

(b) Bigger synthesis area

(c) Smaller synthesis area

(d) No synthesis area

Figure 5.5: *Pre-Synthesized improvement - different sizes of the synthesized area around the edges. Image (a) shows a case of full synthesis case, size of area around the edges is larger than size of the image. Image (b) shows a case where an area of size 2 was synthesized around the edges on quarter resolution, 4 on half resolution and 8 on full resolution. Image (c) shows a case where an area of size 1 was synthesized around the edges on quarter resolution, 2 on half resolution and 4 on full resolution. On the image (d) area around the edges is 0, meaning nothing was sythesized and the image is all composed of pre-synthesized textures.*

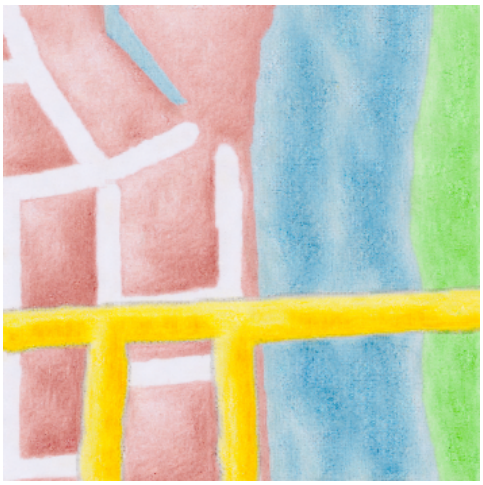
5.4 Zoom experiments

Texture synthesis, as described in this thesis, has a wide use. It can be used, for example, to do ”loss-less” zooming on a bitmap texture. Given the sample style, texture is synthesized on each zoom level, meaning zoomed texture has the same texture appearance and quality as the original texture. See examples on figure 5.6. Zoom is performed without loosing the quality and resolution of the texture. Light-blue areas on the water may look like glitches, but they are a part of the style. In the original texture, there is light-blue water near the shore and dark-blue water in the center of the river. However, guide textures do not distinguish between them, meaning the texture synthesis algorithm cannot differentiate between them.

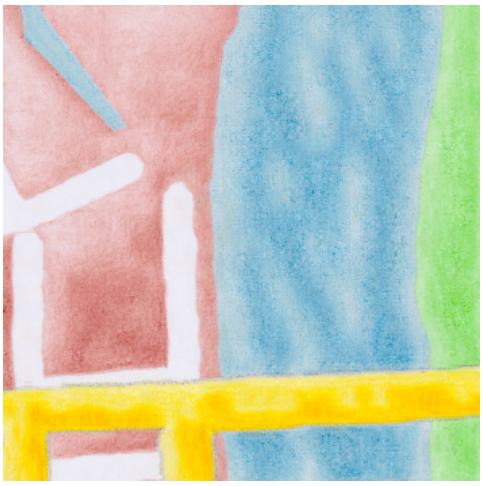
5.5 More results

Figure 5.7 and 5.8 show more results of two different crayon styles. Sample style can be also a very simple filter, experiments with basic filters are shown on figure 5.9.

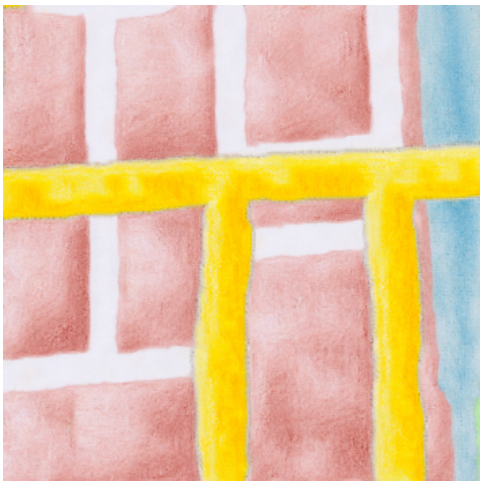
(a) Original zoom



(b) Zoom to 150%



(c) Zoom to 200% water



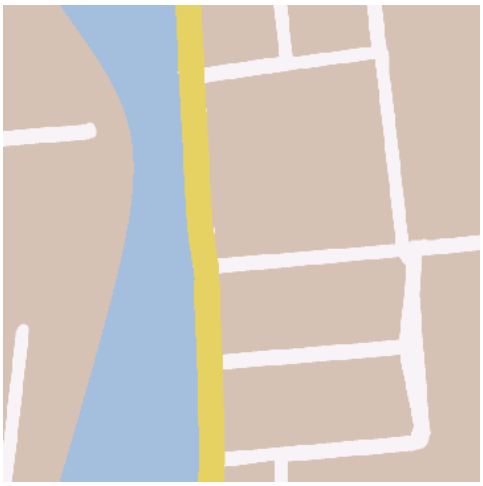
(d) Zoom to 200% road

Figure 5.6: *Zoom experiments. The image (a) shows original texture. Zoom to 150% is on the image (b). Image (c) shows zoom to 200% on the water region and the image (d) shows 200% zoom on the road.*

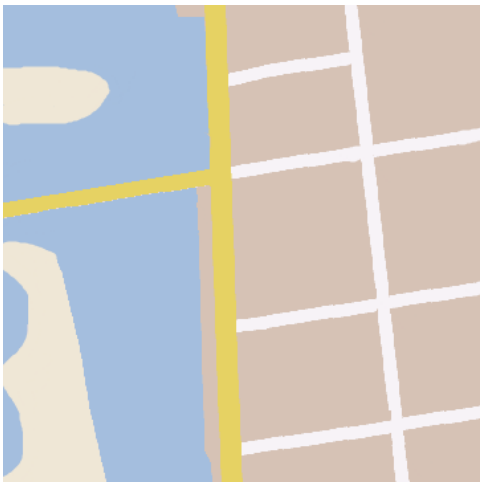
(a) Guide 1 (b) Guide 2

(c) Result 1 (d) Result 2

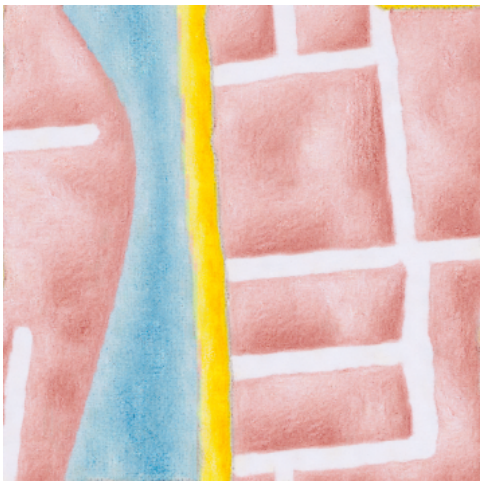
Figure 5.7: *Crayon drawing example. The top row shows guide images, bottom row shows the results.*



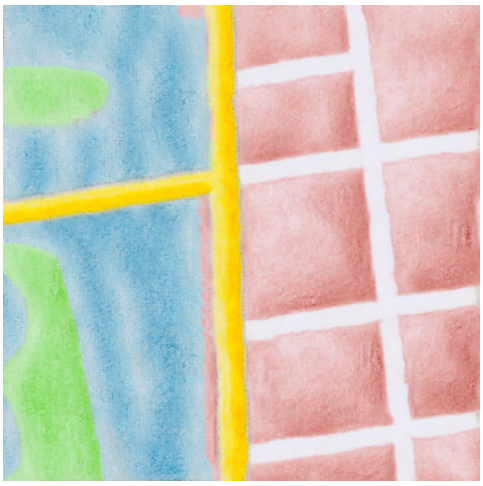
(a) Guide 1



(b) Guide 2

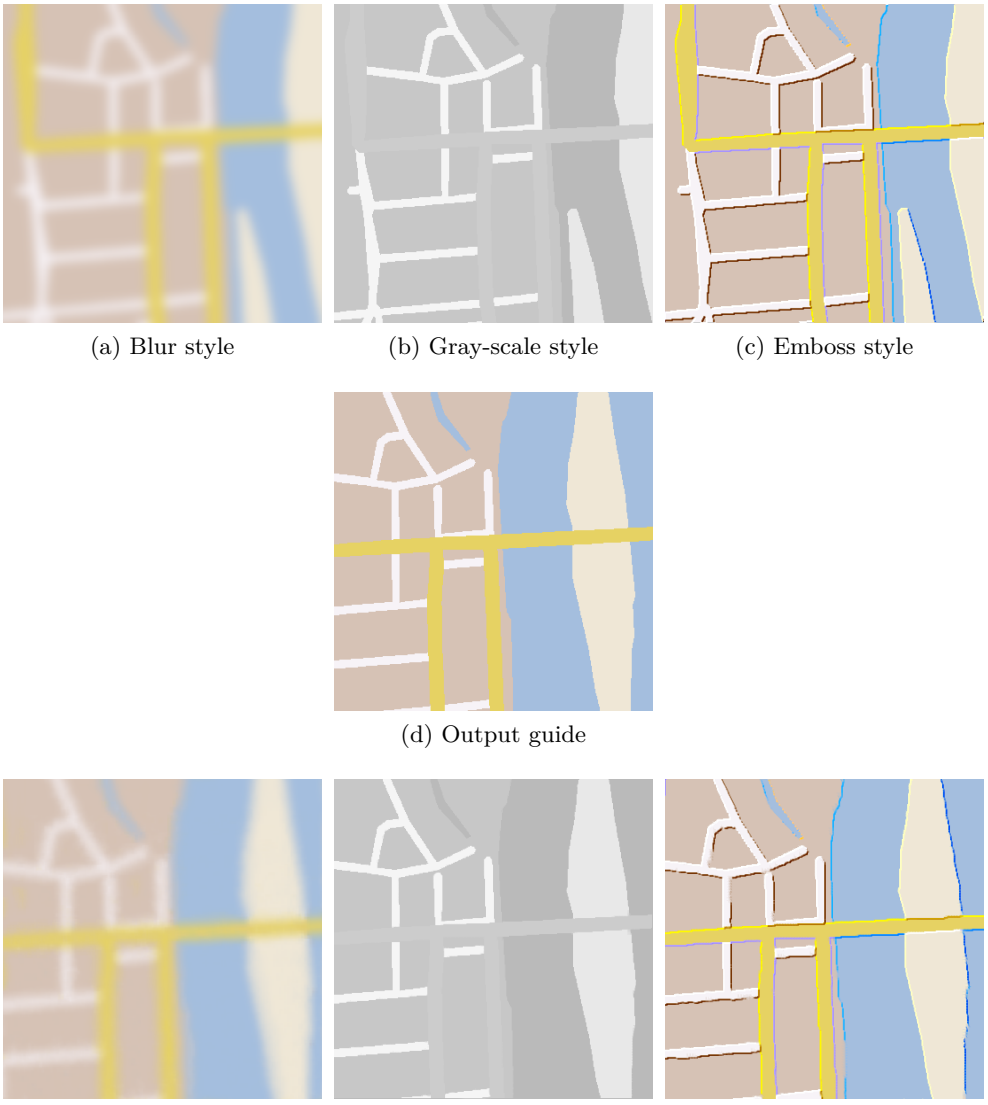


(c) Result 1



(d) Result 2

Figure 5.8: *Crayon drawing example. The top row shows guide images, bottom row shows the results.*



(a) Blur style (b) Gray-scale style (c) Emboss style

(d) Output guide

(e) Blur result (f) Gray-scale result (g) Emboss result

Figure 5.9: *Example of three basic filters. To obtain image (e), image (a) was used as a sample style and image (d) was used as an output guide. Image (f) was created using grayscale style (b) and image (g) was created using emboss style (c).*

Conclusion

In the course of this thesis, we have begun with a description of the stylization problem in general, and a motivation to solve such problem. Many related publications were mentioned, and the most important of them were described in detail. Numerous terms and approaches used to solve the texture synthesis problem were explained. Kwatra's [19] optimization approach and definition of texture energy were formulated and extended two times. At first, from non-guided texture synthesis to guided texture synthesis. Second, it was extended by Kaspar's [30] occurrence map Ω in order to mitigate the wash-out effect [23].

Based on this previous formulation of the problem, we then presented the basic method to solve the guided texture synthesis problem. This basic method was extended by the multi-resolution approach. Additionally, extended guidance, based on the content of the stylized image and computing of Ω_{best}, was introduced. Provided is also an explanation of speed-up using pre-synthesized textures.

Furthermore, detailed implementation with many pseudocodes is provided. Starting with basic functions and its pseudocodes and continuing with a description of the texture synthesis algorithm. Parallel implementation and OpenCL kernels of the most time-consuming parts of an algorithm are also provided. The texture synthesis algorithm was integrated into the Dynavix GPS navigation as a prototype. Details of integration are presented as well.

Finally, the results are presented and compared with other stylization approaches and implementations. Our results have significantly better quality in map stylization in most cases. Moreover, computational time is substantially lower and quality is still within a satisfactory range. Our implementation does not have any dependencies on additional resources like neural networks, databases, etc., meaning it is ideally suited for integration into mobile navigation applications or any other pipeline or device. The goal of this thesis was therefore successfully fulfilled.

After reading this thesis, the reader should be familiar with the texture

synthesis problem in general along with its past as well as with the state-of-the-art methods that used to solve this problem. The reader should also be able to implement all the methods and algorithms described in this thesis.

Future work

Still, some options for future improvement are available. The algorithm can be improved in multiple ways and its implementation can be more effective. Although finding NNF is, in the case of the Pre-Synthesized improvement computed only around the edges, there is still a possibility to achieve further speed improvement by using the approximative method [35]. Since this implementation targets mobile devices and not all mobile phones support OpenCL, another future development could be an implementation of an OpenGL version of finding NNF. Moreover, working with a mask during NNF computing could be done more efficiently. Although this implementation uses OpenCV, only a few and very basic structures and functions from this library are actually used. By removing OpenCV, implementation can become even more independent. Many other minor things could also be improved, but mentioned were the most fundamental.

Bibliography

[1] Efros, A. A.; Leung, T. K. *Texture Synthesis by Non-Parametric Sampling.* International Conference on Computer Vision, 1033-1038, 1999.

[2] Efros, A. A.; Freeman, W. T. *Image Quilting for Texture Synthesis and Transfer.* SIGGRAPH '01, Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, 341-346, 2001.

[3] Agrawala, M.; et al. *Visualizing Route Maps.* Stanford University, Computer Science Dept, 2001.

[4] Fišer, J.; Jamriška, O.; Lukáč, M.; Shechtman, E.; Asente, P.; Lu, J.; Sýkora, D. *StyLit: Illumination-Guided Example-Based Stylization of 3D Renderings.* ACM Transactions on Graphics, 35(4), 2016.

[5] Dynavix - GPS Navigation and Maps for Android. Available from: `http://www.dynavix.com`

[6] Hertzmann, A.; Jacobs, C. E.; Oliver, N.; Curless, B.; Salesin, D. H. *Image Analogies.* SIGGRAPH Conference Proceedings, 327-340., 2001.

[7] Salisbury, M. P.; Wong, M. T.; Hughes, J. F.; Salesin, D. H. *Orientable textures for image-based pen-and-ink illustration.* SIGGRAPH Conference Proceedings, 401-406., 1997.

[8] Zhao, M.; Zhu, S. C. *Portrait painting using active templates.* International Symposium on Non-Photorealistic Animation and Rendering, 117-124, 2011.

[9] Zeng, K.; Zhao, M.; Xiong, C.; Zhu, S. C. *From image parsing to painterly rendering.* ACM Transactions on Graphics 29, 1, 2., 2009.

[10] Curtis, C. J.; Anderson, S. E.; Seims, J. E.; Fleischer, K. W.; Salesin, D. H. *Computer-generated watercolor.* SIGGRAPH Conference Proceedings, 421-430., 1997.

[11] Haevre, W. V.; Laerhoven, T. V.; Fiore, F. D.; Reeth, F. V. *From Dust Till Drawn: A real-time bidirectional pastel simulation.* The Visual Computer 23, 9-11, 925-934., 2007.

[12] Bousseau, A.; Kaplan, M.; Thollot, J.; Sillion, F. *Interactive watercolor rendering with temporal coherence and abstraction.* International Symposium on Non-Photorealistic Animation and Rendering, 141-149, 2006.

[13] Bénard, P.; Lagae, A.; Vangorp, P.; Lefebvre, S.; Drettakis, G.; Thollot, J. *A dynamic noise primitive for coherent stylization.* Computer Graphics Forum 29, 4, 1497-1506., 2010.

[14] Winnemöller, H.; Kyprianidis, J. E.; Olsen, S. C. *XDoG: An extended difference-of-gaussians compendium including advanced image stylization.* Computers & Graphics 36, 6, 740-753., 2012.

[15] Lu, C.; Xu, L.; Jia, J. *Combining sketch and tone for pencil drawing production.* Proceedings of International Symposium on Non-Photorealistic Animation and Rendering, 65-73., 2012.

[16] Sloan, P. P. J.; Martin, W.; Gooch, A.; Gooch, B. *The Lit Sphere: A model for capturing NPR shading from art.* Proceedings of Graphics Interface, 143-150., 2001.

[17] Hashimoto, R.; Johan, H.; Nishita, T. *Creating various styles of animations using example-based filtering.* Proceedings of Computer Graphics International, 312-317, 2003.

[18] Bénard, P.; Cole, F.; Kass, M.; Mordatch, I.; Hegarty, J.; Senn, M. S.; Fleischer, K.; Pesare, D.; Breeden, K. *Stylizing animation by example.* ACM Transactions on Graphics 32, 4, 119., 2013.

[19] Kwatra, V.; Essa, I. A.; Bobick, A. F.; Kwatra, N. *Texture optimization for example-based synthesis.* ACM Transactions on Graphics 24, 3, 795-802., 2005.

[20] Wexler, Y.; Shechtman, E.; Irani, M. *Spacetime completion of video.* IEEE Transactions on Pattern Analysis and Machine Intelligence 29, 3, 463-476., 2007.

[21] Fišer, J.; Lukáč, M.; Jamriška, O.; Gingold, Y.; Asente, P.; Sýkora, D. *Color Me Noisy: Example-based rendering of hand-colored animations with temporal noise control.* Computer Graphics Forum 33, 4, 1-10., 2014.

[22] Barnes, C.; Zhang, F. L.; Lou, L.; Wu, X.; Hu, S. M. *PatchTable: Efficient patch queries for large datasets and applications.* ACM Transactions on Graphics 34, 4, 97, 2015.

[23] Newson, A.; Almansa, A.; Fradet, M.; Gousseau, Y.; Pérez, P. *Video inpainting of complex scenes.* SIAM Journal of Imaging Science 7, 4, 1993-2019, 2014.

[24] Jamriška, O.; Fišer, J.; Asente, P.; Shechtman, E.; Sýkora, D. *LazyFluids: Appearance Transfer for Fluid Animations.* ACM Transactions on Graphics 34, 4, 92., 2015.

[25] Han, J.; Zhou, K.; Wei, L. Y.; Gong, M.; Bao, H.; Zhang, X.; Guo, B. *Fast example-based surface texture synthesis via discrete optimization.* The Visual Computer 22, 9-11, 918-925., 2006.

[26] Lefebvre, S.; Hoppe, H. *Appearance-space texture synthesis.* ACM Transactions on Graphics 25, 3, 541-548, 2006.

[27] Kopf, J.; Fu, C. W.; Cohen-Or, D.; Deussen, O.; Lischinski, D.; Wong, T. T. *Solid texture synthesis from 2D exemplars.* ACM Transactions on Graphics 26, 3, 2., 2007.

[28] Simakov, D.; Caspi, Y.; Shechtman, E.; Irani, M. *Summarizing visual data using bidirectional similarity.* Proceedings of IEEE Conference on Computer Vision and Pattern Recognition., 2008.

[29] Wei, L. Y.; Han, J.; Zhou, K.; Bao, H.; Guo, B.; Shum, H. Y. *Inverse texture synthesis.* ACM Transactions on Graphics 27, 3, 2008.

[30] Kaspar, A.; Neubert, B.; Lischinski, D.; Pauly, M.; Kopf, J. *Self Tuning Texture Optimization.* Computer Graphics Forum 34, 2, 349-360., 2015.

[31] Chen, J.; Wang, B. *High quality solid texture synthesis using position and index histogram matching.* The Visual Computer 26, 4, 253-262., 2010.

[32] Gatys, L. A.; Ecker, A. S.; Bethge, M. *A Neural Algorithm of Artistic Style.* Computing Research Repository, abs/1508.06576, 2015.

[33] Simonyan, K.; Zisserman, A. *Very Deep Convolutional Networks for Large-Scale Image Recognition.* arXiv preprint, arXiv:1409.1556, 2014.

[34] Liao, J.; Yao, Y.; Yuan, L.; Hua, G.; Kang, S. B. *Visual Attribute Transfer through Deep Image Analogy.* Computing Research Repository, abs/1705.01088, 2017.

[35] Barnes, C.; Shechtman, E.; Finkelstein, A.; Goldman, D. B. *PatchMatch: A randomized correspondence algorithm for structural image editing.* ACM Transactions on Graphics 28, 3, 24, 2009.

[36] Goshtasby, A. A. *Similarity and Dissimilarity Measures*. Image Registration. Advances in Computer Vision and Pattern Recognition., 2012.

[37] OpenCL - Open Computing Language. Available from: `https://www.khronos.org/opencl/`

[38] OpenCV - Open Source Computer Vision Library. Available from: `https://opencv.org/`

[39] Champandard, A. J. *Semantic Style Transfer and Turning Two-Bit Doodles into Fine Artwork*. Computing Research Repository, abs/1603.01768, 2016.

[40] Bethge, M.; Ecker, A.; Gatys, L.; Kidziński, L.; Warchol, M. Deepart.io. Available from: `https://deepart.io`

Appendix A

Acronyms

SSD Sum of Squared Differences

NNF Nearest-neighbor Field

OCC Occurrence, refers to the Occurrence Map

OpenCV Open Computer Vision

OpenCL Open Computing Language

CPU Central Processing Unit

GPU Graphic Processing Unit

DSP Digital Signal Processor

FGPA Field-Programmable gate arrays

API Application Programming Interface

GPS Global Positioning System

CNN Convolutional Neural Network

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `impl` ..... implementation sources
- `res` ..... the directory of resource images
- `text` ..... the directory with thesis PDF
- `thesis` ..... the directory of LaTeX source codes of the thesis