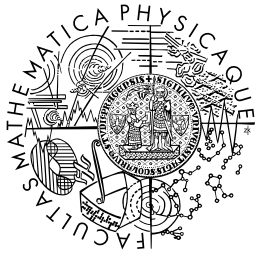


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Petr Laitoch

Procedural Modeling of Tree Bark

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	Mgr. Jan Beneš
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2018

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Procedural Modeling of Tree Bark

Author: Petr Laitoch

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jan Beneš, Department of Software and Computer Science Education

Abstract: Even though procedural modeling of trees is a well-studied problem, realistic modeling of tree bark is not. However, the more general techniques of texture synthesis, including texture-by-numbers, may be helpful for modeling tree bark. Texture synthesis is a process of generating an arbitrarily large texture similar to an input image. This method is capable of generating homogeneous textures. This is not enough as many types of bark are inhomogeneous. Texture-by-numbers improves texture synthesis by further guiding the process with provided label maps to allow the generation of even inhomogeneous textures. Many texture-by-numbers algorithms are not currently implemented. In this thesis, we implement a promising texture-by-numbers algorithm along with algorithms for generating the required label maps. This combination of algorithms creates a pipeline for synthesizing realistic tree bark textures based on a single small input image. We test out the pipeline on samples of multiple types of real-world tree bark images and discuss the results. We further suggest multiple directions for improving the employed techniques.

Keywords: procedural modeling modelling tree bark

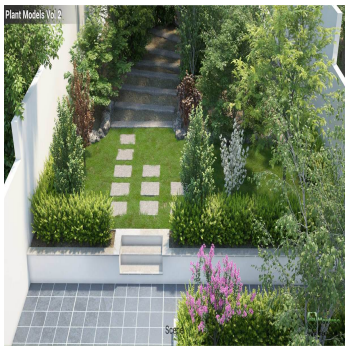
I would like to thank my supervisor, Mgr. Jan Beneš, for the help and guidance he has given me while working on this thesis.

Contents

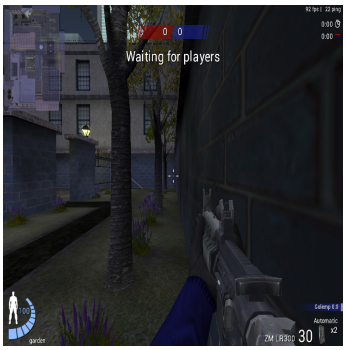
1. Introduction

Much of computer graphics is involved with the creation and rendering of computer-generated scenes. Many of these scenes contain trees. One can find trees in architectural visualizations, see Figure 1.1a, landscape management applications, see Figure 1.1b, simulators, video games, see Figure 1.1c, and movies. The visual quality of trees present in a scene can greatly increase its visual appeal to humans.

(a) Public space reconstruction visualization.



(b) Professional high quality plant models (Maxtree).



(c) Video game (Urban Terror).

Figure 1.1: Tree models in current software applications.

Consequently, many methods for automatic tree generation are being researched. Some of the most significant aspects of trees such as their overall structure and their leaves are a common subject of study in procedural modeling, see Pirk et al. [2012].

Tree bark has a significant influence on tree appearance, especially at close range. However, literature dealing with procedural modeling of bark is limited.

Currently, tree bark is typically only represented by a photograph-based texture. For example, the commercial tree modeling package Xfrog offers tree bark texture scans. When many tree objects are present in a software product such as a computer game, a bark texture is typically repeated many times on all trees of a given species. When viewed from a close distance, this becomes apparent to the viewer. For applications requiring a higher degree of visual quality such as high-end architectural visualizations or movies, trees including bark are designed utilizing a significant amount of manual labor of computer graphic designers.

Sometimes, when a tree or a group of trees are very important components of a scene, a designer might want to control a tree's visual appearance by manually or automatically modifying its bark. Specific patterns in the bark could be needed in a specific location on the tree. For example, one could want to grow moss from the same direction on all tress. Or maybe tree bark should be damaged or ripped off on logs where crossing forest paths. The ability to easily and intuitively add more complex structure to tree bark would lead to procedural trees with a much higher quality, especially when seen from a close distance.

Nowadays, for scenes demanding high quality, trees are generated using commercial procedural modeling software such as Xfrog or they are bought and imported as complete 3D models from companies like, for instance, Maxtree. The obtained models are then used by scene designers, who are responsible for the creation of whole scenes. It is therefore beneficial to create tools that will allow computer graphic designers to create trees more easily and without the use of commercial software.

In this thesis, we combine existing approaches and algorithms for generating textures to devise a new approach for automatic generation of tree bark. As input, we require an existing tree bark texture sample. Using clustering, as described in Section 3.5, we create the input texture's label map which will define similar regions in it. We then use Rosenberger et al. [2009], a label map synthesis algorithm, to synthesize a new semi-random output label map. We then provide these two new maps as input to a texture-by-numbers algorithm, Fan et al. [2012], in order to synthesize new bark textures. A diagram of the whole pipeline is shown in Figure 1.2.

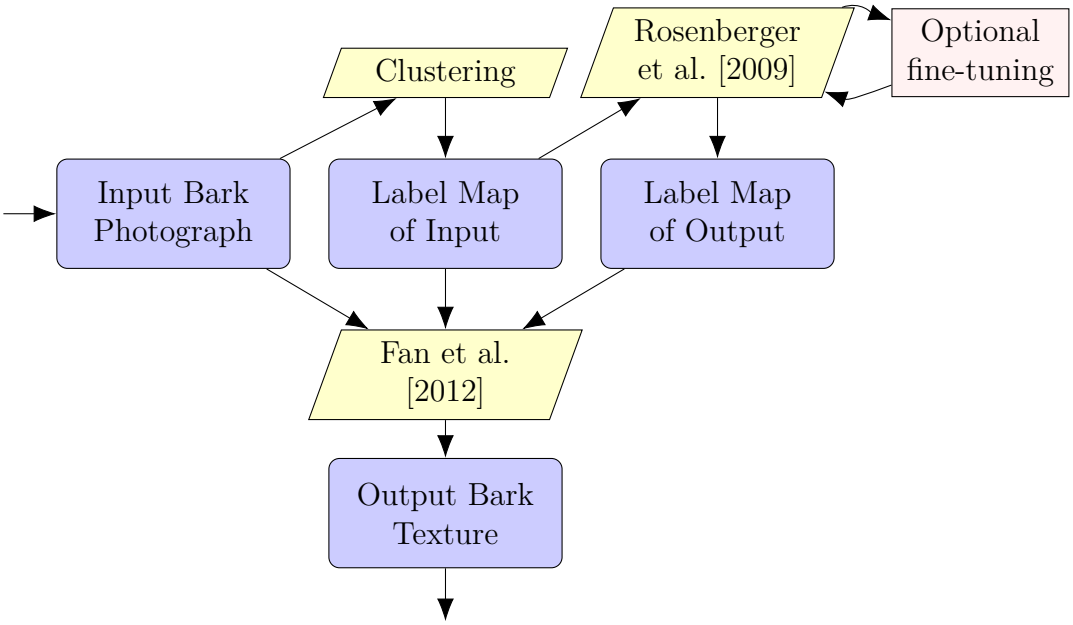


Figure 1.2: Diagram of our bark generation pipeline.

This approach provides results surpassing existing techniques in realism and is general enough to cover multiple types of bark of real-world tree species. We discuss the achieved results in Chapter 4, where we also point out difficulties and suggest improvements both in the texture synthesis algorithms in general and with respect to the problem of modeling tree bark.

The input images (and possibly label maps) could be collected into a library and provided to a scene designer. They could either choose a bark image based on a species or upload their own and generate the bark with one click, only setting the required output size. To fine-tune the result, if need be, they would have the possibility to manually modify the intermediate results of the process generating the output label map in order to influence the large scale patterns of the bark.

Our main contribution is the design of a new method for procedural generation of tree bark. As there exist no implementations of the chosen algorithms by Fan et al. [2012] and Rosenberger et al. [2009] used in this method, a significant portion of the work will be the reimplementation of these known algorithms.

2. Previous Work

In this Chapter, we first present a brief overview of existing techniques for procedural modeling of trees in Section 2.1. We then talk about tree bark and classify it in into multiple categories in Section 2.2. In Section 2.3, we discuss previous attempts at procedural modeling of tree bark. Since we will use methods of texture synthesis to model and generate tree bark in this thesis, we present a classification of textures in Section 2.4.1 and then provide an overview of existing texture synthesis algorithms in Section 2.4.2. Lastly, in Section 2.4.3, we briefly discuss the possibility of utilizing neural networks.

2.1 Procedural Modeling of Trees

The creation of botanical tree computer models is a complex task requiring specialized modeling methods, Shek et al. [2010]. One of the first methods introduced for tree modeling were Lindenmayer-systems (also called L-systems), Prusinkiewicz and Lindenmayer [1990]. L-systems are a formal grammar with which one can model trees and plants by defining special rules and by hand-tuning parameters. Barron et al. [2001] later introduced a method for procedural modeling of tree movement in the wind and in the rain.

When a high-quality tree model is needed, a valid possibility is recreating an actual real-world tree by the means of three-dimensional laser scanning, see Xu et al. [2007]. Livny et al. [2011] devised a method of encoding an input tree model using an intermediate representation called lobes that represent the individual subsections of three-dimensional space taken up by leaves and branches to capture overall tree shape.

TreeSketch, introduced by Longay et al. [2012], is an approach to tree modeling with which artists may produce trees of varying shapes in just a few brush strokes. Often, the structure of a tree does not need to be designed. It only needs to match its surroundings. When growing, trees accommodate for nearby obstacles such as buildings, walls or other trees. Pirk et al. [2012] created a tree model that self-adapts and changes shape when presented with a nearby obstacle.

2.2 Types of Tree Bark

Tree bark is the protective outer layer of a tree completely covering its trunk, branches and roots. The process of bark formation is very complex - even more complex than that of wood. Many patterns found on tree bark are created by the interaction of the two different tissues that play a role in bark formation. In comparison, wood is composed of only one tissue. Bark is also exposed to external factors such as weather, injury or disease that all contribute to its visual appearance. The physical and other processes in bark formation differ among bark types.

Dendrologists often classify tree bark into various categories. These tree bark types are largely arbitrary and differ from author to author. Nevertheless, having such a classification helps with the recognition of tree species. It is however

important to note that bark structure changes significantly during the lifespan of a tree. It is common, that a tree may be assigned to multiple bark categories depending on its age. Bark category may even differ among different branches of the same tree.

Vaucher [2003] classifies bark into 18 types. There exist more classifications and Vaucher himself warns, that his classification is entirely arbitrary. In Figure 2.1, you can see an incomplete classification heavily inspired by his work.

(a) 1: Bark smooth, bumpy; thin. *Carpinus betulus.*

(b) 2: Bark with shallow, vertical fissures; thin. *Acer capillipes.*

(c) 3: Bark with deep, vertical furrows; thick. *Castanea sativa.*

(d) 4: Bark with interlacing ridges; thick and hard. *Fraximus excelsior.*

(e) 5: Bark divided into rectangular blocks by deep fissures; thick and hard. *Diospyros melanoxylon.*

(f) 6: Corky bark wirh raised rhytidomes; thick and hard. *Phellodendron amurense.*

Figure 2.1: Types of bark.

2.3 Procedural Modeling of Tree Bark

Only a handful of papers were published on the topic of modeling tree bark. Bloomenthal [1985] created one of the first works concerned with the modeling of trees. He modeled tree bark using a digitized bump-map of actual maple bark using plaster and an x-ray. The bump-map was then overlapped at its sides for a seamless mapping.

Trees are commonly being covered by bark textures using simple texture mapping. Most often, a single bark image is taken, used multiple times and blended. An overview of texture mapping approaches specifically suited for covering an entire tree with a bark texture, including a list of several blending approaches, can be found in Maritaud [2003].

Some of the first bark models were those of Federl and Prusinkiewicz [1996] and Federl and Prusinkiewicz [2002]. They used a mass-spring model and created a physics-based simulation of fractures in bi-layered materials such as tree bark or drying mud. Inspired by their work, some of the most promising results of tree bark synthesis are likely due to Lefebvre and Neyret [2002]. Their model is also based upon the physics of fractures. It is applicable to all fracture-based barks, such as the bark of a fir tree. The appearance of the generated bark is adjustable by the user through parameters of their algorithm. These include among others bark stiffness, fracture density and fracture shape. The generated fractures were mapped onto a three-dimensional tree model and mapped with textures containing fine-grain details of both the fracture interior and the outer bark layer. These textures were to be created by a skilled artist - once for a given tree species. Desbenoit et al. [2005] later built up upon this method and modeled cracks and fractures on a large variety of surfaces, including wood. Methods for creating realistic-looking bark using GPU shaders can be found in Ricklefs [2005].

Wang et al. [2003] produced models of tree bark of very high quality. However, they were not concerned with generating tree bark. They proposed a sequence of methods and implemented a software tool for the end user who could then construct a height-map of a provided bark image. They then used this height-map to filter out shadows from the original bark photograph by an estimated position of a single light source. Finally, they proposed an appropriate rendering method.

Another notable model of tree bark is that of Dale et al. [2014]. Their goal was not to procedurally generate bark for use in computer animated scenes. Instead, it was to describe the biological process of bark creation. They created a bio-mechanical model of a single tree, namely *xanthorrhoea*, commonly known as the grasstree. Parameters adjust the overall appearance. Their simulation captured the growth of the tree along with the aging of its bark.

To further improve the quality of tree bark, one may add additional features such as mold on top of the bark. See Joshua [2005] for mold modeling. J. et al. was capable of deforming tree bark around objects such as street signs or fences.

2.4 Generating Textures

Texture synthesis is the process of creating (usually large) digital images from a small sample image. It aims to create images of a given size as similar to the original image as it can. Visual artifacts, such as repeating copy-pasted structures or seams, should be kept at a minimum.

2.4.1 Types of Textures

According to Lin et al. [2004], textures may be classified into a spectrum of regular, near-regular, irregular, near-stochastic and stochastic textures as seen in Figure 2.2 [1]. Stochastic textures are completely random textures and look almost like noise. These are the easiest to synthesize, since they exhibit no large

[1]Reused from `https://commons.wikimedia.org/wiki/File:Texture_spectrum.jpg` on 18th May 2018

scale patterns visible to a human observer. On the opposite extreme of the spectrum are regular textures, which can be created by periodically repeating a small shape or pattern by tiling it in equal intervals. Near-regular textures include for example weaved baskets, cloth or brick walls. Near-stochastic textures are images generated by nature in a random fashion such as grass, clouds or water surfaces. Irregular textures are the ones in between these two extremes. They include most day-to-day photographs of objects.

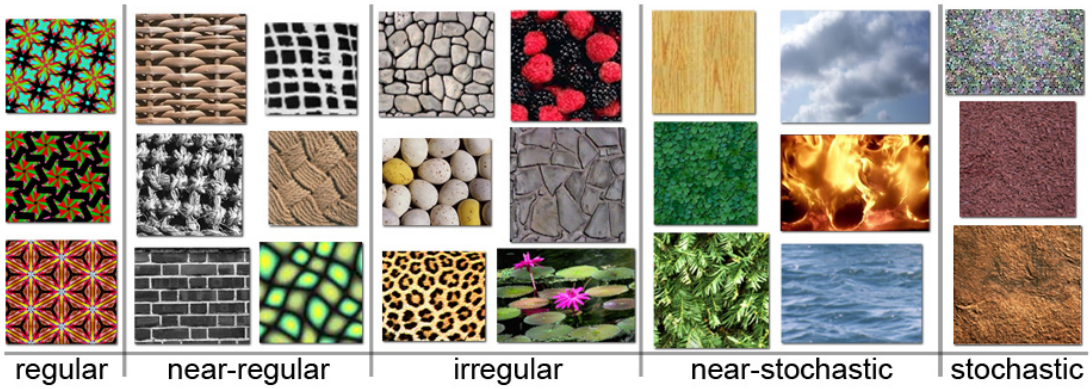


Figure 2.2: Texture spectrum by James Hays of Lin et al. [2004].

Often, a simpler terminology is used. Non-homogeneous textures refer to irregular textures of the texture spectrum, while homogeneous textures are either regular or stochastic textures. Homogeneity may be described from a statistical point of view. Homogeneous textures have equivalent statistical properties in all regions. At least when the region size is chosen appropriately.

2.4.2 Overview of Texture Synthesis Algorithms

Texture synthesis algorithms can be classified into pixel-based, patch-based and optimization-based methods. Recently, also into methods based on neural networks.

First, let us discuss some of the simplest, most specific, texture synthesis methods. They usually work on a very narrow set of textures. These include stochastic texture synthesis, single purpose structured texture synthesis and others. There are also other techniques based upon tiling, blending and seam reduction.

Pixel-based methods are based on copying pixels from the input texture to the output texture one at a time. Some of the most notable pixel-based methods include Efros and Leung [1999], Wei and Levoy [2000] and Hertzmann et al. [2001]. These methods may not only be used for synthesizing two-dimensional textures, but also for synthesizing textures directly on three-dimensional objects.

Patch-based methods copy whole coherent patches from the input texture to the output texture. These methods include Liang et al. [2001], Efros and Freeman [2001] and Kwatra et al. [2003]. These methods are susceptible to discontinuities along the edges of individual patches. Wu and Yu [2004] introduced Feature Matching to prevent breaking of features important to the human visual system, such as edges. They first create feature maps that locate these features. Then, they modify previous algorithms to increase continuity along the feature map.

Optimization-based methods first define an energy function that corresponds to the similarity between the input texture and the output texture. An initial output texture is generated. This initial output texture is iteratively modified, which causes the optimization of the energy function. One of the most notable optimization methods is that of Kwatra et al. [2005]. The two algorithms we implement in Section 3.3 and in Section 3.6 are based on this algorithm.

A more detailed overview of texture synthesis methods can be found in Wei et al. [2009]. They also state that patch-based and optimization-based methods are more successful at synthesizing two-dimensional images than pixel-based methods. Meanwhile, pixel-based methods perform well when synthesizing textures directly on three-dimensional objects.

2.4.3 Neural Networks for Texture Synthesis

Recently, neural network-based approaches are being introduced. These include Gatys et al. [2015], Gatys et al. [2016], Ustyuzhaninov et al. [2016] and Wilmot et al. [2017]. These approaches mostly use convolutional neural networks to synthesize textures. This may be due to the fact, that convolutional neural networks closely resemble Gaussian image pyramids often used in classic texture synthesis algorithms.

When beginning work on this thesis, we decided against the use of neural networks due to a limited amount of literature and few available out-of-the box implementations. We were aiming to create a method capable of modeling tree bark based on a single small input bark photograph. We were concerned by the fact, that neural networks in general require large datasets and computational resources for training. We later learned that this is not always the case. For more information, see Ulyanov et al. [2017].

During the work on this thesis, the following implementations of image analogy and texture syntheses algorithms were being created independently of our work: Neural Doodle (based on Li and Wand [2016]) Deep Textures, and Image Analogies. These implementations might be a good alternative to the algorithm we implemented and described in Section 3.3.

3. Tree Bark Generation

In this Chapter, we describe our proposed tree bark texture synthesis pipeline along with all implemented algorithms. We start with Section 3.1, where we describe the Image Analogy framework. We then give an overview of our whole pipeline in Section 3.2, where we describe how the individual pipeline components fit together and how they fit into the Image Analogy framework. A major component of the pipeline, texture-by-numbers by Fan et al. [2012], is described in Section 3.3. In Section 3.4, we describe a texture-by-numbers initialization method for increased quality of the resulting bark texture. Aside from an input tree bark photograph, texture-by-numbers requires two additional inputs - namely the input label map and the output label map. In Section 3.5, we present a clustering-based image segmentation method for creating an input label map from an input tree bark photograph. In Section 3.6, we present the algorithm of Rosenberger et al. [2009] for generating an output label map from an input label map.

3.1 Image Analogies

Image Analogies is a framework for processing images by example by Hertzmann et al. [2001]. Using this method, when given an input image A, a filtered version of the input image A' and an unfiltered version B of the wanted output image, we are able to create a filtered output image B'. B' should visually relate to (be analogous to) B in the same manner as A' relates to A.

Applications of image analogies include creating arbitrary image filters, texture synthesis, super-resolution, texture transfer, artistic filters and texture-by-numbers.

The simplest application is texture synthesis. In this case, the images A and B are not used or left blank. Since there is nothing else to guide the analogy process except for the input image A', the output image B' is created as if the whole framework were just a texture synthesis algorithm.

When performing texture-by-numbers with the image analogy framework, the images A and B are used as discrete label maps, where pixels of A' and B' are grouped into regions marked as having the same color at their respective positions in A and B. Changing the color of a region in both A and B to a different unused color should ideally have no impact. This property is not true for the remaining applications of image analogy. Due to this reason, one can design algorithms taking advantage of this specific property and implement a texture-by-numbers algorithm only instead of the whole image analogy framework.

3.2 Method Overview

Now that we understand the image analogies framework (Section 3.1), we can proceed with the explanation of the pipeline we use to generate tree bark. The input of our algorithm is a small photograph capturing tree bark details. The input must contain examples of all patterns we want to generate on the resulting

tree bark. Our aim is to be able to generate many different output images of tree bark at random, given a desired output size. Of course, we want the output images to be as visually close to the input as possible.

First, we generate the input label map A from the input image A', as described in Section 3.5. Then, using the algorithm described in Section 3.6, we generate the output label map B from the generated input label map A. Since now we have the images A', A and B, we can use texture-by-number (Section 3.3) to generate the output image B', which is the output of our whole pipeline. Notice the correspondence to the diagram in Figure 1.2. In addition, Section 3.4 describes a sophisticated method of providing an initial input to the texture-by-numbers algorithm, greatly increasing quality of the final output.

A nice feature of our approach is that we are using multiple algorithms in a sequence and that these algorithms themselves have intermediate results, corresponding to finer and finer details of synthesis. This means, that if the user wants more control over the resulting image, they can stop the computation at any time, manually modify the image corresponding to an intermediate result, and let the pipeline run its course and complete all the fine-grained detail the user does not want to deal with anymore.

3.3 Texture-by-Numbers

Texture-by-numbers is an integral part of our pipeline. This method is capable of synthesizing non-homogeneous images with global-varying patterns. We use this algorithm to generate the final resulting tree bark texture based on the user-provided input and additional images generated specifically for it using all other algorithms of our pipeline. Those are described in the following Sections of this Chapter. Existing texture-by-numbers algorithms bordering with the state-of-the-art (not considering neural-network based approaches) include Busto et al. [2010], Sivaks and Lischinski [2011] and Fan et al. [2012].

For our implementation, we have chosen Fan et al. [2012]. Their algorithm builds upon the research of Sivaks and Lischinski [2011]. Choosing to implement an improved version of a given algorithm made more sense to us.

The chosen algorithm offers a more complete pipeline, improving the synthesis process in several parts of the framework. The core idea behind the algorithm is based upon texture optimization as introduced by Kwatra et al. [2005]. The chosen framework improves previous results by utilizing a sophisticated initialization step based on Bonneel et al. [2010] instead of random initialization, see Section 3.4 for details. To further improve synthesis quality, a distance metric can be specified based on texture type, see Section 3.3.4. For structural textures, Fan et al. [2012] define a distance metric utilizing a feature distance map inspired by the work of Wu and Yu [2004], as previously used in Lefebvre and Hoppe [2006]. Also, the chosen algorithm is parallelizable on a GPU using CUDA.

Busto et al. [2010] is mainly concerned with one of the most important steps of the algorithm - the nearest neighbor search strategy - which we discus further in Section 3.3.6. They aim to improve upon parallel k-coherence search

by introducing their own search strategy called parallel randomized correspondence. However, in comparison to Fan et al. [2012], the approach of Busto et al. [2010] was not yet implemented and tested on a GPU. Our chosen framework also claims, that their improvements provide a similar quality increase to that of randomized correspondence. In addition, it should be possible to improve our implemented texture-by-numbers framework by simply substituting k-coherence search by parallel randomized correspondence. This is left as future work.

3.3.1 Algorithm Overview

Our goal is to synthesize the output texture B' while keeping it visually similar to the input image A' and matching the labels A and B. We employ discrete optimization to achieve this goal. An energy function is defined to reflect synthesis quality taking into consideration the previously mentioned conditions:

$$E_{TBN} := w * \sum_{q \in B} \|A_p - B_q\|^2_{N_{\text{lmr}}(\text{L2})} + \sum_{q \in B'} \left\|A'_p - B'_q\right\|^2_{N_{\text{lmr}}(\text{L2})} \tag{3.1}$$

Where for each pixel q in label B and image B', p specifies a pixel in A and A' whose input neighborhood is the closest to q's output neighborhood. The distance $N_{\text{lmr}}(\text{L2})$ is the Euclidean distance of the square pixel neighborhoods. More specifically:

$$p := \text{argmin}(w * \|A_p - B_q\|^2_{N_{\text{lmr}}(\text{L2})} + \left\|A'_p - B'_q\right\|^2_{N_{\text{lmr}}(\text{L2})}) \tag{3.2}$$

Energy functions will be further discussed in Section 3.3.4.

The algorithm employs a hierarchical image pyramid. We give a high-level overview of the algorithm in Algorithm 1. We begin at the coarsest image pyramid level $L_{initLevel}$ of the input and label images A'_L, A_L and B_L. First, we generate the initial output B' as discussed in Section 3.4. This is done only at the coarsest level. Then, we modify the image B' in order to minimize the energy function E_{TBN} in the optimization step. We perform the optimization step $maxIterTime$ times, see Section 3.3.5. Once optimization is complete, we move onto images A'_{L+1}, A_{L+1} and B_{L+1} in the image pyramid. We perform the upsampling step to increase the resolution of the intermediate result B'_L to match the dimension of B_{L+1}, see Section 3.3.2. The upsampling step is not performed at the finest level.

Algorithm 1 Texture-by-Numbers(A',A,B)

1: **procedure** TBN(A', A, B)
2: Bulid image pyramid for A', A, B
3: $B'_{initLevel} \leftarrow$ InitializeTBN(A', A, B)
4: **for** $L = initLevel$:-1:0 **do**
5: **for** $n = 0$:$maxIterTime$ **do**
6: $Match_L^{n+1} \leftarrow \underset{j \in cand(i)}{\text{argmin}}(w * \left\|A_{Lj} - B_{Li}\right\|^2_{N_{\text{lmr}}(\text{L2})} + \left\|A'_{Lj} - B'_{Li}\right\|^2_{N_{\text{lmr}}(\text{L2})})$ ▷ M-step
7: $B'^{n+1}_L \leftarrow \text{argmin}(E_{TBN})$ ▷ E-step
8: **if** $L \neq 0$ **then**
9: $B'_{L-1} \leftarrow \text{Upsample}(B'_L)$ ▷ Upsample

Pixel Neighborhood

When referring to a pixel neighborhood in this thesis, we always mean its square neighborhood. A neighborhood $N(p)$ centered on a pixel p in an image at a given index is further defined by the neighborhood size. This is an algorithm parameter that needs to be fine-tuned by the user. A neighborhood's size denotes the length of a side of the square defining the neighborhood. Only odd sizes are acceptable.

$N_{\mathrm{lmr}}(p)$ denotes of full square neighborhood. The individual letters stand for left, middle and right. The number of pixels in this type of neighborhood of $size$ is $size^2$.

$N_{\mathrm{lr}}(p)$ denotes of a square neighborhood with the middle pixel p left out. The individual letters stand for left and right. The number of pixels in this type of neighborhood of $size$ is $size^2 - 1$.

Image of References

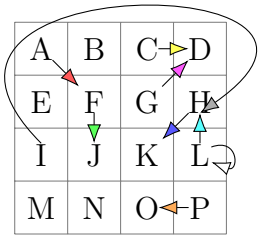
The optimization works by copying pixels from the input image A' to the output image B'. The algorithm requires us not only to keep a color value of all pixels of the synthesized image B', but we must also keep a reference to the source image pixel from which the pixel is copied over. The algorithm will use the knowledge of the neighborhood of that pixel to its advantage in k-coherence search and during forward-shifting. Since we already have the information about pixel positions, we will also use it later during upsampling when increasing the synthesized image resolution. B' will thus be an image of references.

Using this fact and the fact that the energy function essentially only uses the squared Euclidean distance, we can change the number of channels of the input image A' arbitrarily. We can for example add a height-map corresponding to A' as a fourth channel and the algorithm we generate an output height-map corresponding to B' without any extra effort. As a plus, this may actually increase synthesis quality, as if in were a feature distance map. See Section 3.3.4 for more details.

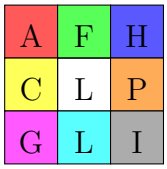
Forward-Shifted Pixel Neighborhood

For each referenced pixel in the input neighborhood, we compute an appropriately forward shifted reference based on relative pixel positions, as visually explained in Figure 3.1. The illustration is using *wrap*.

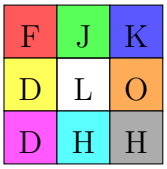
When using *wrap*, forward-shift pixels across neighborhood boundaries.
When not using *wrap*, output *NaN* instead of crossing neighborhood boundaries.



(a) Referenced image



(b) Neighborhood of references



(c) Forward-shifted neighborhood of references

Figure 3.1: Illustration of forward-shifting.

3.3.2 Hierarchical Image Pyramid

Images are kept in an image pyramid. This way, we can first optimize images at a low resolution to focus on large scale patterns. When $maxIterTime$ optimization steps were performed at a given pyramid layer, we move on to the next level. We must also increase the resolution of the synthesized image. This is done by upsampling. The pyramid itself is created by first calculating image sizes at individual pyramid levels and then downsampling the original images to these sizes. We must take care that the indices used in the downsampled and upsampled images match at each pyramid level. These two operations must be inverse to each other.

Pyramid Level Size Computation

Given an input image size and a number of levels, we will calculate the image sizes at individual pyramid levels. We always want to keep both the number of pyramid levels and the lowest resolution size at a minimum. Somewhat arbitrarily, we will adopt the strategy of always doubling both the image width and hight when going up the pyramid when possible. This may be impossible for the level at the finest resolution, where we may have to increase the sizes by less then a factor of two.

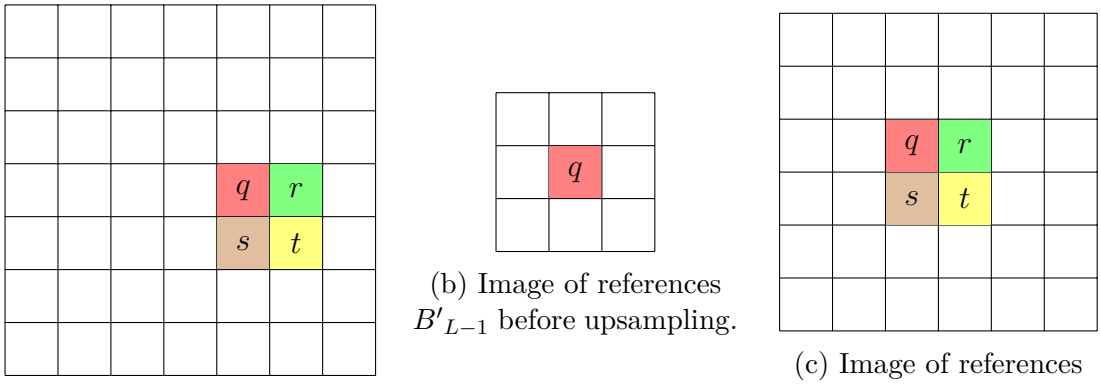
Downsampling

We will usually downsample by a factor of two. This means to simply removing every other row and column and recomputing references. However, during our first downsampling, we may have to downsample by less than a factor of two. Then we downsample the images by uniformly removing rows and columns from the input image. We never remove two rows or columns that are next to each other. This will allow us to perform upsampling. We however need to make sure

that we add back the exact same rows and columns during upsampling as we removed during downsampling.

Upsampling

The upsampling step is performed in order to increase the resolution of an image of references by a factor of 2 or less. Pixel references must first be recomputed. We now add rows and columns matching the positions from which downsampling removes rows and columns. Each newly created pixel inherits an appropriately shifted coordinate of its left, top or top-left neighbor. An illustration of upsampling pixel q in B'_{L-1} when generating B'_L is shown in Figure 3.2.



(a) Referenced image A'_L.

(b) Image of references B'_{L-1} before upsampling.

(c) Image of references B'_L after upsampling.

Figure 3.2: Illustration of upsampling.

3.3.3 Approximate Nearest Neighbor Search

One of the most computationally expensive steps of the algorithms we implement in this thesis is nearest neighbor search. Given a dataset of points in an arbitrary dimension and a testset of points in the same dimension, the goal of k-nearest neighbors is to find for each point in the test set, k points in the dataset, which are closest under a given metric. In this thesis, we will always use the squared Euclidean metric with k-nearest neighbors. There are no known algorithms for solving this problem faster than by linear search. Given our use case, this is time complexity unacceptable. However, an approximate nearest neighbor search gives results comparable to the exact search. They are at least for more than good enough for our purposes.

The only external algorithm library used when implementing the algorithms of Fan et al. [2012] and of Rosenberger et al. [2009] is the FLANN library, Muja and Lowe [2014], for computing approximate nearest neighbors. More possibilities exist. However, we did not pursue them. For example, Fan et al. [2012] suggests the use of Tang et al. [2012] and claims faster computation time on images of sizes $\leq 256 * 256$ pixels. That would cover most of the images we generated.

3.3.4 Distance Metric of Texture Similarity

For images exhibiting structural patterns, such as clearly defined edges and ridges, the L_2 distance of image neighborhood is not sufficient. The human visual system

places an extra value on edges and ridges. To take this into account, we can modify the energy function as follows:

$$E_{TBN} := w * \sum_{q \in B} \|A_p - B_q\|^2_{N_{\text{lmr}}(\text{L2})} + z * \sum_{q \in B'} \|FDM_p - FDM_q\|^2_{N_{\text{lmr}}(\text{L2})} + \sum_{q \in B'} \left\|A'_p - B'_q\right\|^2_{N_{\text{lmr}}(\text{L2})} \tag{3.3}$$

Alternatively, we can keep the energy function the same. We just consider FDM as an additional image channel of A'.

Coefficients w and z determine the importance of the label and feature distance channels respectively. These parameters need to be adjusted properly for good output image quality. The value of these coefficients chosen by Fan et al. [2012] were $w = 1$ and $z = 3$. In our bark generation pipeline, we found good performing values to be $w = 3$ and $z = 8$. It however depends on a given set of inputs.

More information about the calculation of feature distance maps can be found in Lefebvre and Hoppe [2006].

3.3.5 Discrete Solver Based on k-coherence

This discrete optimization solver we employ was introduced by Han et al. [2006] and is an adaptation of the original EM-like texture optimization framework by Kwatra et al. [2005]. The original adaptation utilized a hierarchical tree search for the M-step and least-squares for the E-step. The hierarchical tree search was the computational bottleneck with a time complexity of $O(log(N))$ where N was the number of input pixels. k-coherence 3.3.6 has constant time complexity per search while keeping satisfactory quality, as stated by Lefebvre and Hoppe [2005].

The least squares method originally used in the E-step is incompatible with k-coherence and must be changed as a consequence. In particular, k-coherence stores references to pixels in the exemplar in the synthesized image. This is done because k-coherence needs to know the source pixel location of every synthesized pixel. The least squares method however does direct computation on the RGB values in the synthesized image and thus does not allow for direct pixel copying from the exemplar. To overcome this problem, a different E-step was adopted.

One iteration of the discrete solver:

(Inputs):

- A': Input texture image with the optional feature distance map FDM included as additional channels.
- A: Input label map.
- B: Output label map.
- B': Output reference image generated in the previous iteration. For generating the initial B' before the first discrete solver iteration, see 3.4.

(0) Definitions:

Let X and Y be two images of the same shape. Then $concat(X, Y)$ is an image with all image channels of X and Y concatenated.

Let X be an image (or neighborhood) of references. Then X_{ref} is an image (or neighborhood) with references resolved.

Let X be an image. Then $N(X)$ is a list of all neighborhoods in X. The length of $N(X)$ is the same as the number of pixels in X.

Let X be an image of references. $N^{\text{forward\_shift}}(X)$ is the set of all forward shifted neighborhoods of references in X. The length of $N^{\text{forward\_shift}}(X)$ is the same as the number of references in X.

(1) Construct a candidate set:
A candidate set for each reference in B' is built using k-coherence search 3.3.6.

Let $x \in B'$ be a reference. Then $cand(x)$ is the candidate set of x.

(2) M-step: Find best-matching candidate.
Let $E := concat(A, A')$.

Let $S := concat(B, B'_{\text{ref}})$.

We want to create an image of references $Match_L$ of the same shape as S.

Let $X^\dagger$ be a subset of indices of S. The references in $Match_L$ will be defined only at indices in $X^\dagger$.

$X^\dagger$ is created by uniformly leaving out pixels of $Match_L$.

For every neighborhood in the subset of the synthesized image, find the closest neighborhood to it in the exemplar among the coherence candidates and save its reference into $Match_L$:

$$\forall i \in X^\dagger : Match_{L_i} := \underset{j \in cand(i)}{\text{argmin}}(\|N_{\text{lr}}(S)_i - N_{\text{lr}}(E)_j\|^2_{\text{L2}}) \tag{3.4}$$

(3) E-step: Find candidate minimizing the energy function.
Let $M := N^{\text{forward\_shift}}_{\text{lmr}}(Match_L)_{\text{ref}}$ be an image of neighborhoods.

Create the image Avg by converting each neighborhood into a single pixel by taking the channel-wise average over all neighbors. Undefined neighbors do not take part in the average computation.

Let $T := concat(B, M)$.

Let I be the set of indices in T.

For every pixel in the synthesized image, take the channel-wise average of the corresponding forward shifted neighborhood in $Match_L$, find the closest pixel to it in the exemplar among the coherence candidates and save its reference into B_L:

$$\forall i \in I : B_{L_i} := \underset{j \in cand(i)}{\text{argmin}}(\|T_i - E_j\|^2_{\text{L2}}) \tag{3.5}$$

(Output):

- B'_L: The new reference image. It should be converted to $B'_{L_{\text{ref}}}$ after the last iteration.

3.3.6 k-coherence Search

In many texture-by-numbers algorithms, we repeatedly replace each synthesized pixel by the exemplar pixel with most the similar neighborhood. As this is a frequent and expensive step, multiple search strategies for the most similar pixel exist. They aim at a balance between a fast runtime and a high-quality output.

These nearest neighbor search strategies include:

- Coherence search by Ashikhmin [2001].
- k-coherence search by Tong et al. [2002].
- Patch Match by Barnes et al. [2009].
- Coherent random walk method by Busto et al. [2010].

In this thesis, the texture-by-numbers framework we chose to implement was Fan et al. [2012]. k-coherence search was the method chosen in this framework, which is why we implemented it. Newer methods, such as coherent random walk, show promising results by improving synthesis quality while maintaining fast synthesis times. As future work, it would be beneficial to implement these newer methods to try and improve our bark synthesis pipeline.

Computation of k-coherence search:
Let B' be an image of references to pixels in the image A'.
For each pixel $p_0 \in B'$, we want to compute the candidate set $C(p_0)$.

First, we need a precomputed list of $(k-1)$-nearest neighbors for each pixel in the image. We use the FLANN library. (Both the dataset and the testset contain all image pixels).

Next, we compute coherence candidates $C_1(p_0)$ of pixel $p_0 \in B'$:

- We denote $N_{\text{lmr}}(p_0)$ the set of pixels in the neighborhood of p_0.
- Every pixel $p \in N_{\text{lmr}}(p_0)$ corresponds to a pixel $p' \in A'$.
- We can then compute p'' by forward-shifting p'.
- Forward-shifting means that the difference in coordinates between p' and $p'' \in A'$ is the same as the difference in coordinates between p and $p_0 \in B'$.
- The set of all possible pixels p'' is the coherence candidate set $C_1(p_0)$.

Finally, the candidate set $C(p_0)$ of k-coherence search is created as a union of the coherence candidate set $C_1(p_0)$ and the precomputed lists of $(k-1)$-nearest neighbors for each pixel in $C_1(p_0)$.

3.4 Texture-by-Numbers Initialization

3.4.1 Random Initialization

Originally, the texture optimization framework initialized the algorithm by copying random pixels from the input to the output. A slightly improved technique copies random neighborhoods from the input to the output. A simple texture-by-numbers adaptation would be to only assign pixels or neighborhoods within corresponding label map regions.

While a simple initialization is sufficient for homogeneous textures, synthesis of non-homogeneous textures may be significantly improved by better initialization. In fact, we have found that a high quality result during bark synthesis for certain types of bark may be achieved by simply utilizing texture synthesis (not even texture-by-numbers) with this more sophisticated initialization, see Section 4.1. An illustration of the effect a proper initialization can have on the quality of the result is shown in Figure 4.4. Fan et al. [2012] also notes an increased runtime speed in his implementation due to faster convergence of the later optimization compared to a random initialization.

3.4.2 Texture Synthesis Based Initialization

As a sophisticated initialization for the optimization of texture-by-numbers, we use the first pass of the fast guided texture synthesis method of Bonneel et al. [2010]. In the cited paper, a texture-by-numbers algorithm is also being created. However, their goal is to use this algorithm to quickly and easily synthesize three-dimensional scenes. Fan et al. [2012] decided to only use only the first pass of this other texture-by-numbers algorithm as initialization for our texture-by-numbers algorithm, as it is fast and improves results. It would also be interesting to know, if further increase in synthesis quality would be gained by also implementing the second pass, or if the optimization we perform would renders it unnecessary. This is left as future work.

To initialize the optimization of texture-by-numbers, we create an approximate texture-by-numbers result by using chamfer distance as a metric in a region growth method. The initialization process proceeds as follows:

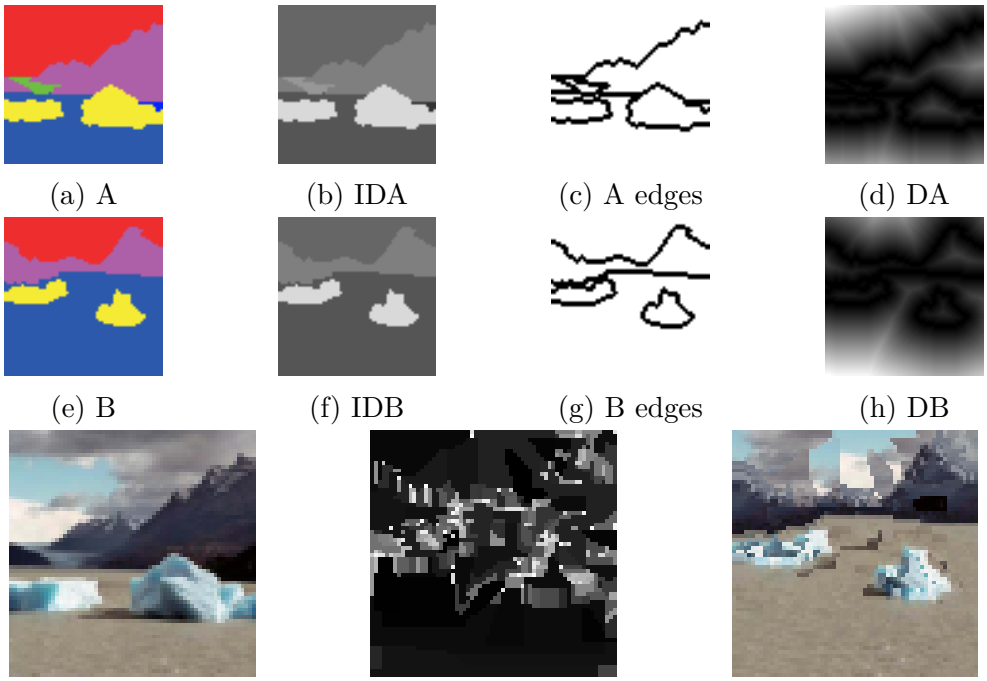
First, discrete label maps must be generated: IDA from A and IDB from B. Pixels of a similar color in the label map must be mapped into one discrete value. In Figure 3.3, this is visualized by gray-scale values. An edge detection algorithm is then used to locate all borders between different discrete label regions. Finally the *3-4 DT method* of Borgefors [1988] is used to produce distance maps DA and DB. Distance maps mark the approximate distance of each pixel from the closest pixel of a different discrete label region.

Now that we have computed the necessary chamfer distance, we can define the following metric:

$$d(p_A, p_B) := w * \|IDA_{p_A} - IDB_{p_B}\|_{N_{\text{lmr}}(\text{chamfer})} + \|DA_{p_A} - DB_{p_B}\|_{N_{\text{lmr}}(\text{L2})} \quad (3.6)$$

Where $w = 90$. In the last step, we first create a blank image $initB$. We randomly select seeds to grow in $initB$ and grow it using a floodfill algorithm. The region growth stop condition is $d(p_B, p'_B) < t$. t is defined as $(d(p_A, p_B) + 5) * 1.25$. $p_B \in IDB$ is the seed of the floodfill algorithm. $p_A \in IDA$ is the pixel nearest to p_B under the Equation (3.6). p'_B is a pixel neighboring the region being seeded by p_B and it is a candidate to join the region if it fulfills the condition specified for the floodfill algorithm.

This method is an example of a patch-based texture synthesis algorithm where the input and output label maps provide extra inputs in addition to the original image. This initialization process could also be parallelized to facilitate GPU synthesis.



(a) A (b) IDA (c) A edges (d) DA

(e) B (f) IDB (g) B edges (h) DB

(i) A' (j) initB (k) colorInitB

Figure 3.3: Illustration of generation of initial output.

3.5 Input Label Map Generation

Generation of an input label map for the texture-by-numbers framework is traditionally left to the user. In general, it is not easy to determine the individual regions present in an image nor the number of regions present. It not only depends on the image itself, but also on our overall goal. In fact, image segmentation is a significant area of research in computer science.

We are however only interested in the specific case of bark textures. We would like to capture large-scale patterns in tree bark by mapping pixels into k regions (usually $k <= 5$). This would correspond to key bark attributes such as fractures, lenticels, furrows, fissures, flaking patches, etc. For this purpose, a clustering-based method into regions of similar colors will be sufficient. The amount of colors corresponds to the parameter k. This parameter will depend

mostly on the type of bark and the artifacts present in the bark. Hierarchical clustering methods could relieve the user from having to specify this parameter. However, there are only few possible choices. Also the performance of the other algorithms in our bark generation pipeline depends heavily on the choice of k, as will be discussed in Chapter 4. For these reasons, k is left to be specified by the user.

This whole process is illustrated in Figure 3.4.

3.5.1 *CIELAB* Color Space

During the generation of the input label map, the denoising and quantization steps will be performed in the *CIELAB* (sometimes called *Lab*) color space, as opposed to the standard *RGB* color space. *CIELAB* was designed as a color space approximating human vision. The three components are L, a and b. L closely matches the human concept of lightness and a and b are green-red and blue-yellow color components. This conversion into *CIELAB* and back greatly increases the quality of our algorithm.

3.5.2 Denoising

As preprocessing of the quantization step, we will denoise the image in order to remove small-scale patterns in the image. Figure 3.4b shows a denoised image. The task of denoising is the opposite of adding random noise to the image. One of the motivations behind denoising algorithms is the removal of noise generated by the process of taking a photograph with a digital camera. When using this method, more coherent regions are created in the quantization step. We have chosen the non-local means denoising algorithm by Buades et al. [2011], which is currently implemented in *OpenCV* as *cv::fastNlMeansDenoisingColored*.

It is important that this step looks at the neighborhood of a given pixel when creating its denoised counterpart. This is useful, since humans do not only consider the absolute pixel color as important when recognizing regions, but also discontinuities such as edges or ridges. In a similar algorithm for creating input label maps by Rosenberger et al. [2009], the quantization is performed using k-means on pixel neighborhoods instead of pixels. For us, quantization of pixels is sufficient thanks to this denoising step.

3.5.3 Quantization

The *MiniBatchKMeans* clustering algorithm is used to cluster the denoised clusters converted into *CIELAB* color space into k clusters. This reduces the number of colors present in an image to k. The individual colors of the result correspond to the k individual regions of the label map. Figure 3.4c shows a quantized image.

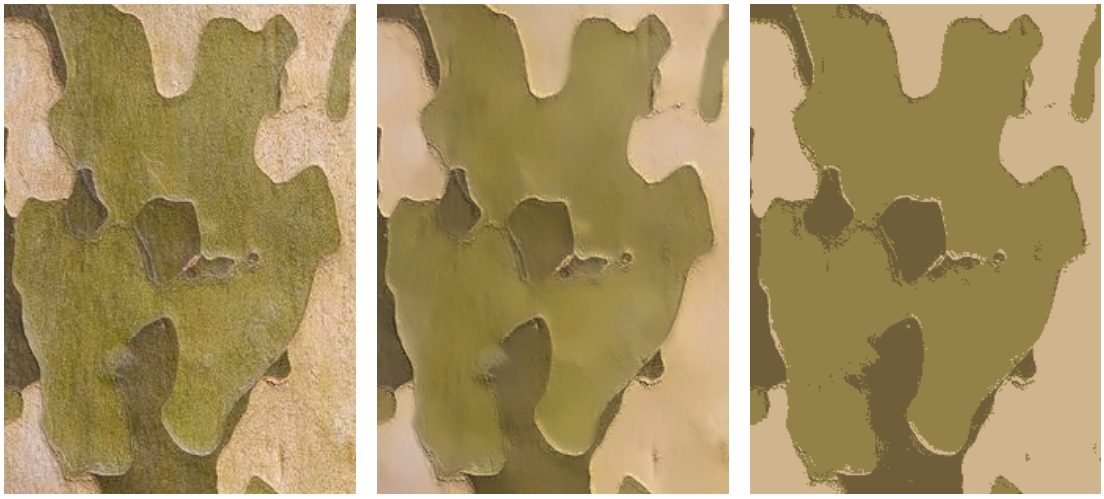
3.5.4 Erosion

Sometimes, the quantization creates very small regions of one or just a few pixels inside a huge monolithic region. Such tiny regions are undesirable. An erosion step further improves the result. A simple heuristic is employed: Each pixel

becomes its most common neighbor. Figure 3.4d shows an image after erosion was applied.

3.5.5 Grayscaling and Edge Detection

Since we only care about designating regions on an images, the actual k-means center colors are unimportant. It is sufficient to keep a grayscale value. We also found that adding a separate region for all edges among the regions found so far often improves quality of the texture-by-numbers framework. However, this postprocessing adds complexity for the algorithm creating the output label map. Therefore, this step should be performed on both input and output label maps after they are both generated. Figure 3.4e shows the final output.



(a) A’ (b) Denoising (c) Quantization

(d) Erosion (e) Grayscaling and Edge Detection

Figure 3.4: Illustration of Label generation.

3.6 Output Label Map Generation

In this Section, we will describe the algorithm by Rosenberger et al. [2009] capable of generating label maps for texture-by-numbers. The algorithm has two parts. The first part is an algorithm capable of synthesizing binary shapes. This algorithm is then used as a basis of an algorithm capable of synthesizing discrete layer maps composed of multiple layers. The concept of layers is inspired by textures consisting of multiple layers in the real world such as peeling paint, height-maps of terrains and surfaces covered by corrosion, rust or lichen.

For the input and output maps to be similar, we want to maintain both local and global similarity. We want the shapes to occupy the same relative area in pixels in both the input and the output. We also want all shape boundaries in the input resemble the boundaries in the output and vice versa. The algorithm will represent these shapes by measuring the similarity of boundary patches. Boundary patches are pixel neighborhoods around pixels at shape boundaries.

The shape synthesis algorithm is based on Kwatra et al. [2005], just like our implemented texture-by-numbers algorithm. Again, the algorithm iteratively optimizes the output with respect to a similarity measure. This similarity measure is defined in Section 3.6.1. The optimization is again composed of two alternating steps. First, there is boundary patch matching introduced in Section 3.6.2. Then, there is shape adjustment introduced in Section 3.6.3.

Optimization proceeds in a Gaussian image pyramid and is initialized using random initialization.

3.6.1 Shape Similarity Measure

Let B_1 and B_2 be the sets of boundary pixels of shapes S_1 and S_2, respectively. Let $x_1 \in B_1$ and $x_2 \in B_2$.

We define the similarity between two boundary pixels as follows:

$$D(x_1, x_2) := \min_{\eta}(\|N_0(x_1) - N_\eta(x_2)\|^2_{\mathrm{L2}}) \tag{3.7}$$

Since the images are binary, the L_2^2 norm is simply the number of different pixels in the two neighborhoods. The subscript of N represents one of four possible 90-degree rotations of the neighborhood. If plausible for the given input image, using these rotations increases the variability of the output by up to a factor of 4.

We define the similarity between a boundary pixel and another shape as follows:

$$D(x_1, S_2) := \min_{x_2 \in B_2}(D(x_1, x_2)) \tag{3.8}$$

We define the similarity between two shapes as follows:

$$D(S_1, S_2) := \frac{\sum_{x_1 \in B_1}(D(x_1, S_2)) + \sum_{x_2 \in B_2}(D(x_2, S_1))}{|B_1 \cup B_2|} \tag{3.9}$$

3.6.2 Boundary Patch Matching

Let B_E and B_S be the set of boundary patches of E and S, respectively.
Compute the boundary patch matching using the following greedy algorithm:

1. Assign each patch in B_E to its nearest neighbor in B_S.
2. For each boundary patch $s \in B_S$, keep only its nearest assignment.
3. Remove all assigned patches from consideration and repeat until all patches have been matched.

Assume, that $|B_S| = Q\,|B_E| + R$.
Create *Choice* as a random choice of R elements from B_E.
Each exemplar patch is to be assigned to a synthesized patch precisely Q times if $e \notin Choice$ and $Q + 1$ times if $e \in Choice$.

3.6.3 Shape Adjustment

We will now increase the similarity of the two shapes by minimizing the energy function $D(S, E)$. That is, we modify the boundary of the synthesized image $B(S)$ by increasing its similarity to the exemplar boundary $B(E)$.

We start by creating weights for pixels near boundaries of the synthesized image. These weights determine pixel changes during an iteration. This weight creation will be done by superimposing each exemplar patch over its counterpart in B_S.

We will assign a score to each pixel x_S in the vicinity of a boundary in the synthesized image $B(S)$. The score predicts if the pixel x_S should belong to the shape or not. A positive value means that it should be a part of the shape S and a negative value means that it shouldn't be a part of S. The absolute value of the score is a measure of certainty of our prediction. We will only change pixels with a prediction higher than a certain threshold.

Let $x_S \in B(S)$ be a synthesized boundary pixel and $x_E \in B(E)$ its corresponding match found in Section 3.6.2. The weights w_{x_S} of the neighborhood $N(x_S)$ are calculated as follows:

$$w_{x_S} := \frac{1}{1 + D(x_S, x_E)} * \mathit{indicator}(x_E) \odot \mathit{falloff} \tag{3.10}$$

Where $\odot$ denotes element-wise matrix multiplication. *indicator* is $+1$ if the pixel belongs the to the given shape and -1 if not. *falloff* is the Gaussian falloff function. This function will make the weight decrease away from the center of the patch - a pixel should have more effect on its immediate surroundings than on the most distant pixels in the neighborhood.

We accumulate the weights of all neighborhoods of pixels in the vicinity of a boundary in the synthesized image $B(S)$ into the weight matrix W. We construct the weight matrix W by adding w_{x_S} to W at the coordinate of x_S for every x_S.

A threshold determines pixel changes resulting from a given shape adjustment step. The threshold is set dynamically to the value in the interval $[10^{-2}, 10^{-7}]$

which will minimize the relative difference in the amount of pixels between the exemplar and the synthesized image. Convergence is reached after a predefined number of iterations or when the number of pixel changes falls below a certain threshold.

3.6.4 Layer Map Generation

The shape optimization algorithm presented in the previous Subsections of Section 3.6 can only synthesize a binary image. However, we want to synthesize a label map which will possibly have more than two colors.

In order to achieve our goal, we will consider label maps to be layer maps. The difference between the two is that a layer map has an additional ordering of its individual regions while a label map does not. A pixel of a given layer is considered to be a part of all lower layers as well. Due to this layer representation, we can decide for every pixel in an image if it belongs to a given layer or not. We can thus create a binary image for each layer. The ordering of layers is left to be specified by user. If the texture is visibly composed of layers, the user should specify this obvious ordering.

The shape optimization algorithm without modification will be used to generate the coarsest layer. We will initialize the algorithm by a randomly generated binary image with the number of foreground pixels matching the corresponding exemplar layer. The shapes of all following layers must be nested inside the previously generated shapes. Also, we will aim at preserving correlation between each pair of successive shapes.

The synthesis of each subsequent layer begins by the creation of a mask. This mask defines the area of the image where the current layer is allowed to synthesize and grow. Each mask is initially set to the shape of the previous layer and then is shrunk according to a weight matrix created based on the last boundary patch matching. We are again superimposing pixels, but this time we are predicting pixels belonging to the subsequent layer. Both the shape of the mask and the layer initialization are determined by the weight matrix using thresholds. At higher resolutions, a similar step is repeated. However, we only modify the upsampled mask and layer shape based on the thresholds used earlier. We do not recreate them.

4. Results and Discussion

In this Chapter, we present experiments we ran with our pipeline under various configurations. We judge the results and discuss the advantages and shortcomings of the utilized methods. In Section 4.1, we discuss the quality of the texture-by-numbers algorithm when used only for texture synthesis. At the same time, we discuss the limits and possibilities of generating bark textures using only texture synthesis. In Section 4.2, we show the importance of proper initialization of the texture-by-numbers algorithm we implemented and state a hypothesis for the reasons behind the need of that particular initialization. In Section 4.3, we use our pipeline to generate tree bark using two-color label maps. The use of multi-color label maps in the pipeline is discussed in Section 4.4

4.1 Texture Synthesis

In the following Section, we use our implemented texture-by-numbers algorithm (Section 3.3) to perform texture synthesis. No label maps are used.

In Figure 4.1, you can see a small detail of an eucalyptus tree bark. Figure 4.1a is the input texture and the two outputs in Figure 4.1b and in Figure 4.1c are different runs of the same algorithm. This bark is clearly composed of three layers. The top two levels are subject to peeling, which gives this bark its characteristic look. It also means that this bark is a good candidate for layered label map generation. Figure 4.7 displays synthesis from the same input bark texture using label maps.

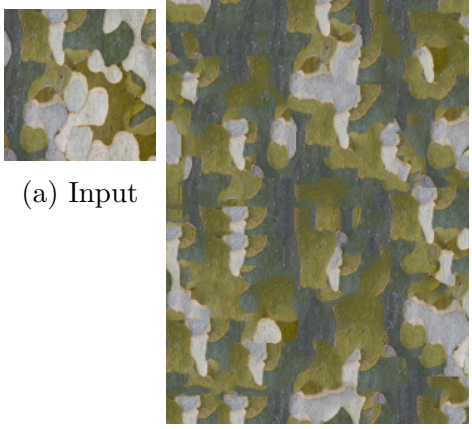
Patterns in the output images are exactly the same as in the input image and they are blended perfectly. However, the overall structure of the texture is not kept. The image histograms do not match. Specifically, one can notice a lager amount of light green and almost no dark green in the outputs in Figure 4.1b and in Figure 4.1c compared to the input in Figure 4.1a.

When not using label maps, we achieve higher resemblance in color and texture. There are no visible seams, nor artifacts of blending. The output texture is perfectly continuous and unless observing the texture very closely at the pixel level, it is hard to tell that the output of our algorithm is not an actual photograph. There are few indicators that the output textures might not be actual photographs. One is reduced sharpness at a few small areas near visible edges in the image. The other is the exact same patterns of pixels being repeated in multiple places of the output image. These pattern became increasingly visible when the output image is significantly larger than the input image, such as in Figure 4.2. That is sadly for us the most common use case of this algorithm. However, for small outputs, this problem is a non-issue.

All in all, texture synthesis is the way to go for generating small homogeneous textures. Texture synthesis is also fine for large output homogeneous textures if the repeating patters do not become too noticeable, such as in Figure 4.3.

(a) Input (b) Output 1 (c) Output 2

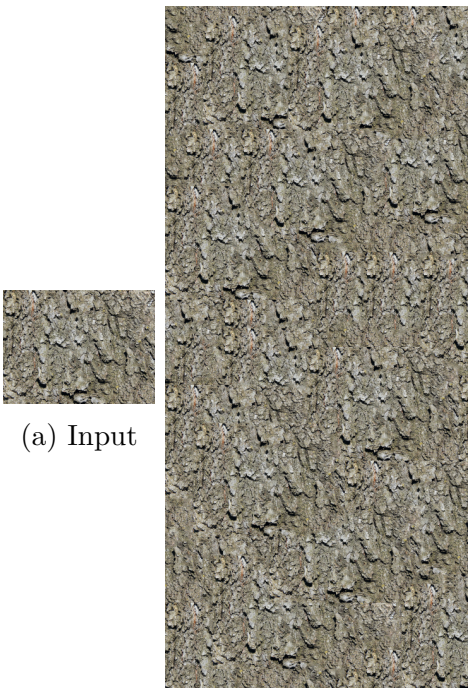
Figure 4.1: Texture synthesis of bark with small output images.



(a) Input

(b) Output

Figure 4.2: Texture synthesis of tree bark with a large bad result.



(a) Input

(b) Output

Figure 4.3: Texture synthesis of tree bark with a large good result.

4.2 Texture-by-Numbers Algorithm Initialization

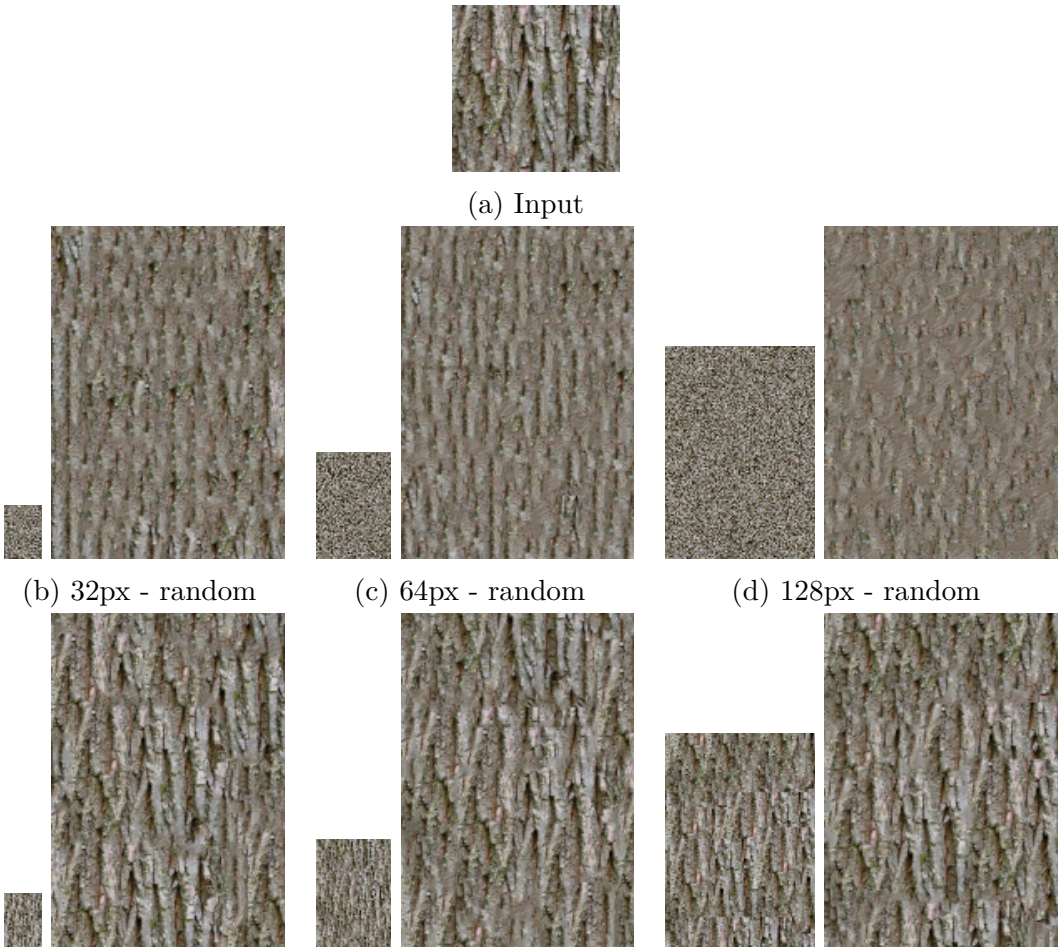
Figure 4.4 shows a comparison of different initialization methods along with the generated results. For each texture, initialization is on the left and the output on the right. For the given input texture, it is clear that random initialization gives poor results. The results for a small initialization size are slightly better. Our intuition is, that this leaves more time for the optimization to recover from bad initialization. On the other hand "*smart*" initialization has close to perfect

results. It is hard to tell which of the initialization sizes produces a higher quality result.

The advantages of "*random*" initialization is its ease of implementation and fast runtime. "*Smart*" initialization has neither. However, as is seen in Figure 4.4, in certain cases the texture-by-numbers algorithm essentially does not work properly unless "*smart*" initialization is used. We believe that this is due to the fact that the initialization and optimization steps complement each other.

During the course of optimization, the output texture is blended. It is being made more continuous. Breaks and seams in the output are being fixed. However, we do not optimize for the coherence of input patches. We do not support pixels close to each other in the input to be in the same relative positions in the output. This is by definition done by patch-based texture synthesis methods. Nor do we optimize so that all pixels in the input are represented in the output. We do not optimize for the input and output histograms to match.

The algorithm we use for "*smart*" initialization is a simple patch-based texture synthesis algorithm that copies coherent patches from the input image and tries to cover all of the input image equally. When this coherence is supplied to the texture synthesis algorithm by the means of initialization, we get a good result.



(a) Input

(b) 32px - random (c) 64px - random (d) 128px - random

(e) 32px - smart (f) 64px - smart (g) 128px - smart

Figure 4.4: Comparison of texture-by-numbers initializations.

We therefore propose as future work a possible change to the energy function being optimized by the texture-by-numbers algorithm to account for this coherence of pixels and equal representation of the input in the output.

4.3 One-Layer Label Maps

Let us first focus on the generated label maps in Figure 4.5. The generated two-color input label map in Figure 4.5b nicely separates the input bark texture (Figure 4.5a) into regions of fracture interiors as the darker color and of the top-most bark layer as the lighter color. We believe that the generated output label map in Figure 4.5c exhibits the same patterns as the input label map. We believe it is of very high quality and that it exhibits no visible defects.

If this label map were used successfully, we believe it would improve synthesis quality by fixing problems described in Section 4.1, such as seams or blending artifacts. It could prevent the repetition of the same input regions over and over again when generating large output textures and could promote the creation of new fracture patterns similar to existing ones by blending them.

The output texture Figure 4.5d generated by our implemented texture by numbers algorithm is very blurry. Some regions are not blurry, but that is because the output label of these regions resembles the input label very closely. An input texture region can thus be copied to the output whole.

This is the same kind of blur as seen in Section 4.2. We thus suppose that this could be fixed by the improved energy function proposed as future work in Section 4.2 or by improving the texture-by-numbers initialization algorithm from Section 3.4 on label maps with many small regions. On the other hand, if we look past the blur, all other problems of generating bark textures we described in previous Sections are fixed.

(a) Input Texture (b) Input Label

(c) Output Label (d) Output Texture

Figure 4.5: Texture-by-numbers with 2-color label maps.

4.4 Multi-Layer Label Maps

In the previous Section, we have praised the results of the shape synthesis algorithm from Section 3.6 when used on a label map with two colors. In this Section, we will show that the result quality decreases dramatically with the number of colors used.

In Figure 4.6, we generate output label maps with three colors and in Figure 4.7, we generate output label maps with five colors. The label map quality achieved for three colors is still acceptable. We had to however manually choose the correct layer ordering as described in Section 3.6.4. With five colors, the output label is completely different from the input label.

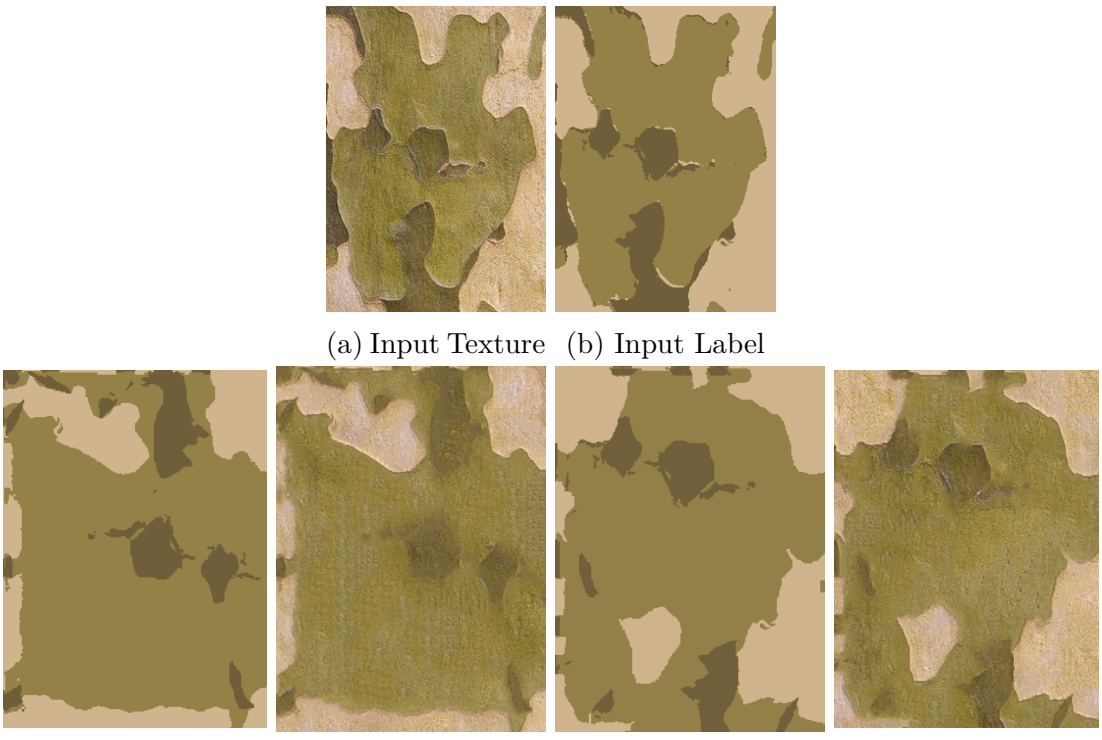
The output quality of the layered shape synthesis algorithm from Section 3.6 decreases rapidly with a growing number of layers. This shows that although the shape optimization algorithm for binary images from Section 3.6 is good, as discussed in Section 4.3, the layer strategy from Section 3.6.4 to generalize the

shape optimization algorithms for multi-colored label maps is not suitable for bark generation.

We would like to state a hypothesis, that the mentioned layer strategy performs well only on a small subset of label maps. The layer ordering must be clearly defined and pixels of a given layer may only border with the preceding or following layer. This is true for discretized terrain height maps, such as the once used directly in Rosenberger et al. [2009].

The problems with the algorithm include bad histogram matching of individual layers. For example, in Figure 4.7c, dark blue covers a majority of the output label. That is however not true for the input label at all. The shapes of individual layers is very distorted. Also, a manual ordering of layers is needed for the algorithm. Most of the time, it is not even clear if a good ordering exists. Nevertheless, badly ordered layers do have a major negative impact on the resulting output label.

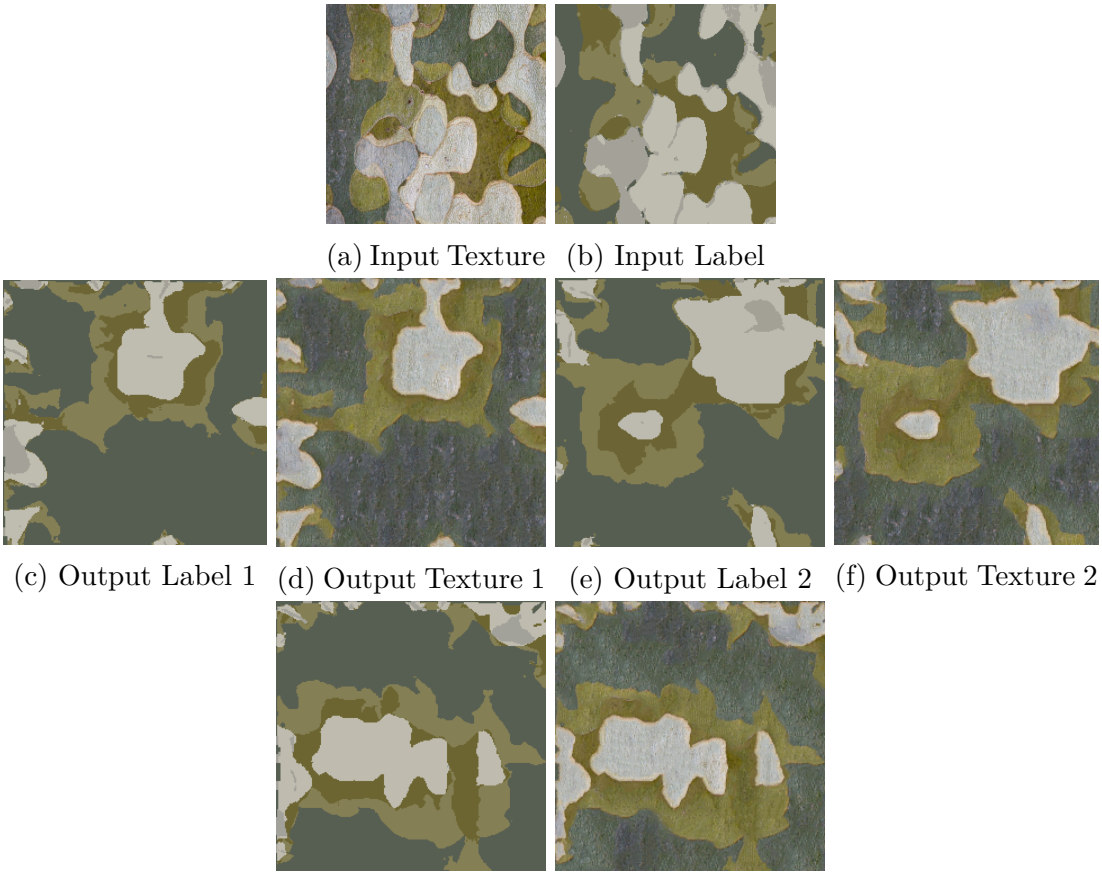
Using texture-by-numbers with automatically generated label maps as showed in Figure 4.6f is a better result than by simply using texture synthesis, as in Figure 4.1b, especially when considering the overall shape of the three color regions in the image and the image histograms. This is probably due to the fact, that this specific bark actually consists of layers.



(a) Input Texture (b) Input Label

(c) Output Label 1 (d) Output Texture 1 (e) Output Label 2 (f) Output Texture 2

Figure 4.6: Texture-by-numbers with 3-color label maps.



(a) Input Texture (b) Input Label

(c) Output Label 1 (d) Output Texture 1 (e) Output Label 2 (f) Output Texture 2

(g) Output Label 3 (h) Output Texture 3

Figure 4.7: Texture-by-numbers with 5-color label maps.

4.5 Rendered Tree Bark Images

(a) With Bump-map

Figure 4.8: Projections of tree bark textures onto a cylinder.

5. Implementation

A significant portion of this thesis is the implementation of a bark generation pipeline. Our implementation was done in Python with performance-critical parts in-lined in C++ using the Python library *weave*. The source code is available on the attached CD. Consider our implementation as a prototype of a tree bark generation pipeline, not an end-user application.

5.1 Installation and Dependencies

The software was developed and tested under Linux. It is written in Python 2, thus the package *python2* is required (Python version 2.7.14 was used). We recommend the Python package manager *pip2* for installing all required Python packages. A required package list is available in the file *requirements.txt*.

Besides standard general-purpose Python packages, we require the use of the *FLANN* library (see Section 3.3.3) available through the python package *pyflann*. At this time, this package is only available in Python 2, not in Python 3.

The package *weave* used to in-line C++ code requires a C++ compiler. By default, it will use the compiler with which Python was installed. *weave* documentation states that any C++ compiler should work, they however recommend *gcc*.

We also used the package *numpy-indexed* in our implementation. As this package is not overly popular, we also discovered that at this moment it is badly packaged. It now requires the package *pyyaml* to be installed beforehand. If this were an end-user application, we would probably avoid use of the *numpy-indexed* package (it is not critical to the software).

To install our implementation of a bark generation pipeline, check that you are running Linux and have installed the following software on your distribution

- C++ compiler (gcc)
- Python 2
- pip

and run the script *setup.sh*.

5.2 Program Structure

Our tree bark generation pipeline can be executed using the script *tree_bark_synthesis/generate_tree_bark.py*. Most important algorithm parameters can be set directly as parameters of this function. The only required input image is the input bark texture A', usually in the file *ai.png*. Also, the output image size must be specified. Optionally, a height-map matching the input image can be specified in order to create an output height-map.

We implemented the texture-by-numbers algorithm of Fan et al. [2012], as described in Section 3.3 in the sub-folder *tree_bark_synthesis/TextureByNumbers*.

The main function is *tbn()*. Important parameters include initialization type, initialization image size, neighborhood sizes used during optimization, optimization iteration count and channel weight coefficients of the energy function.

We implemented the texture-by-numbers initialization as described in Section 3.4 in the sub-folder *tree_bark_synthesis/TextureByNumbers/init*. The main function is *generate_initial_output()*. If you just want to generate "*random*" initial output and not "*smart*" initial output, use the function *random_init()*.

We implemented the texture-by-numbers algorithm of Fan et al. [2012], as described in Section 3.5 in the sub-folder *tree_bark_synthesis/LabelGeneration*. The main function is *color_region_label()*. The most important parameter is k, the number of colors in a label map.

We implemented the control map generation algorithm of Rosenberger et al. [2009], as described in Section 3.6 in *tree_bark_synthesis/ShapeOptimization*. The main function is *shape_optimization()*. Important parameters include Gaussian pyramid size, neighborhood size and how the input image may be rotated for more shape variability.

Intermediate results of the algorithms are being dumped into log folders at runtime.

5.3 Running Experiments

The experiments we ran using our pipeline are included in sub-folders of the *experiments* directory. The input bark image is denoted as *ai.png*. The generated label maps are called *a.png* and *b.png*. The resulting output image is in the file *bi.png*. Intermediate results can be found in subdirectories. The Python function call responsible for synthesis is *run.py*. This script contains the parameters used in the particular experiment. A *makefile* is provided for comfort. To re-run the experiment, run *make*. It will first clean the directory by deleting previously generated output files and intermediate results. It will then synthesize anew. Since we did not seed the random number generator used in our pipeline, the result will thus differ from our previous synthesis.

5.4 Rendering in 3D

The three-dimensional rendering software chosen for the rendering of the final results in Section 4.5 was *VPython*. Scripts launching three-dimensional rendering of tree bark are in the *display_3d* directory. As this package is not needed during bark synthesis, it is only optional to install. We use it, since it is a free and open source package that is very quick to get up and running. Using more sophisticated rendering software would yield higher quality renders. Also, if this software were meant for the end-user, it would be preferable to offer it as a plugin of some popular 3D modeling application rather than to leave it as stand-alone software.

6. Conclusion

In this thesis, we created a pipeline capable of automatically generating a tree bark texture based on a single small bark photograph. In order to do this, we had to implement two texture synthesis algorithms - namely Fan et al. [2012] and Rosenberger et al. [2009] - for which there were no existing implementations. We also implemented a clustering-based image segmentation algorithm to complete the pipeline. The implementation was done in Python and is available on the CD attached to this thesis.

We classified tree bark into multiple types and successfully applied our pipeline to photographs of tree species samples for several types of bark. We discussed the achieved quality of the generated bark texture based on the choice and tuning of pipeline parameters and based on the properties of bark texture itself.

We were able to generate good-looking textures with only texture synthesis where possible. The generated textures however exhibited discontinuities and seams. In order to enhance synthesis, we generated label maps to guide the synthesis process. The quality of two-color label maps for bark textures was very high. The more colors we tried to generate in the label map, the lower was the achieved quality.

When applying these generated label maps, we were able to remove all discontinuities and seams. However, we ran into limitations of the texture-by-numbers algorithm when applying them. A more sophisticated texture-by-numbers initialization method was able to significantly improve performance in some cases. Regrettably, some blur still remained when label maps consisted of many small color regions.

We also succeeded at generating bark not easily synthesizable by texture synthesis algorithms using three-color label maps. With these label maps, we were able to control the large-scale regions and patterns in the bark texture.

Using label maps has the advantage, that artists or designers can use our method to precisely control the results being generated. They can simply alter the output label map during its synthesis at a resolution level of their choice. The resulting bark will then exhibit the large scale patterns of the altered label map.

We pointed out the biggest insufficiencies in the algorithms used when applied to high quality tree bark generation. We further suggested possible improvements as future work. This includes the implementation of other existing nearest neighbor search strategies for texture-by-numbers, as listed in Section 3.3.6. We also suggested an improved energy function taking into account pixel references in Section 4.2. Lastly, the strategy of layered label map synthesis presented in Section 3.6.4 proved mostly ineffective for the purpose of generating tree bark.

Bibliography

Michael Ashikhmin. Synthesizing natural textures. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 217–226. ACM, 2001.

Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan Goldman. Patchmatch: A randomized correspondence algorithm for structural image editing. 28, 08 2009.

Jeremy T Barron, Brian P Sorge, and Timothy A Davis. *Real-time procedural animation of trees*. PhD thesis, Citeseer, 2001.

Jules Bloomenthal. Modeling the mighty maple. In *SIGGRAPH*, 1985.

Nicolas Bonneel, Michiel Van de Panne, Sylvain Lefebvre, and George Drettakis. *Proxy-guided texture synthesis for rendering natural scenes*. PhD thesis, INRIA, 2010.

G. Borgefors. Hierarchical chamfer matching: a parametric edge matching algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10 (6):849–865, November 1988. ISSN 0162-8828. doi: 10.1109/34.9107.

Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. Non-local means denoising. *Image Processing On Line*, 1:208–212, 2011.

Pau Panareda Busto, Christian Eisenacher, Sylvain Lefebvre, Marc Stamminger, et al. Instant texture synthesis by numbers. In *VMV*, pages 81–85, 2010.

Holly Dale, Adam Runions, David Hobill, and Przemyslaw Prusinkiewicz. Modelling biomechanics of bark patterning in grasstrees. *Annals of botany*, 114(4): 629–641, 2014.

Brett Desbenoit, Eric Galin, and Samir Akkouche. Modeling cracks and fractures. *The Visual Computer*, 21(8-10):717–726, 2005.

Alexei A Efros and William T Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 341–346. ACM, 2001.

Alexei A Efros and Thomas K Leung. Texture synthesis by non-parametric sampling. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 1033–1038. IEEE, 1999.

Jing Fan, Xiao-Ying Shi, Zhan Zhou, and Ying Tang. A fast texture-by-numbers synthesis method based on texture optimization. In *Proceedings of the 5th International Symposium on Visual Information Communication and Interaction*, pages 1–10. ACM, 2012.

Pavol Federl and Przemyslaw Prusinkiewicz. A texture model for cracked surfaces, with an application to tree bark. In *Proceedings of Western Computer Graphics Symposium*, pages 23–29, 1996.

Pavol Federl and Przemyslaw Prusinkiewicz. Modelling fracture formation in bi-layered materials, with applications to tree bark and drying mud. In *Proceedings of the 13th Western Computer Graphics Symposium*, 2002.

Leon Gatys, Alexander S Ecker, and Matthias Bethge. Texture synthesis using convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 262–270, 2015.

Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2016 IEEE Conference on*, pages 2414–2423. IEEE, 2016.

Jianwei Han, Kun Zhou, Li-Yi Wei, Minmin Gong, Hujun Bao, Xinming Zhang, and Baining Guo. Fast example-based surface texture synthesis via discrete optimization. 22:918–925, 2006. ISSN 0178-2789. doi: 10.1007/s00371-006-0078-3.

Aaron Hertzmann, Charles E Jacobs, Nuria Oliver, Brian Curless, and David H Salesin. Image analogies. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 327–340. ACM, 2001.

Kratt J., Spicker M., Guayaquil A., Fiser M., Pirk S., Deussen O., Hart J. C., and Benes B. Woodification: User-controlled cambial growth modeling. *Computer Graphics Forum*, 34(2):361–372. doi: 10.1111/cgf.12566. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12566`.

Aimee Joshua. *Modeling and Rendering of Mold on Cut Wood*. PhD thesis, University of Maryland, Baltimore County, 2005.

Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: image and video synthesis using graph cuts. In *ACM Transactions on Graphics (ToG)*, volume 22, pages 277–286. ACM, 2003.

Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. Texture optimization for example-based synthesis. In *ACM Transactions on Graphics (ToG)*, volume 24, pages 795–802. ACM, 2005.

Sylvain Lefebvre and Hugues Hoppe. Parallel controllable texture synthesis, 2005.

Sylvain Lefebvre and Hugues Hoppe. Appearance-space texture synthesis. *ACM Trans. Graph.*, 25(3):541–548, July 2006. ISSN 0730-0301. doi: 10.1145/1141911.1141921. URL `http://doi.acm.org/10.1145/1141911.1141921`.

Sylvain Lefebvre and Fabrice Neyret. Synthesizing bark. In *13th Eurographics Workshop on Rendering Techniques (EGSR'02)*, volume 28, pages 105–116. Eurographics Association, 2002.

Chuan Li and Michael Wand. Combining markov random fields and convolutional neural networks for image synthesis. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2479–2486, 2016.

Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. Real-time texture synthesis by patch-based sampling. *ACM Transactions on Graphics (ToG)*, 20(3):127–150, 2001.

Wen-Chieh Lin, James Hays, Chenyu Wu, Vivek Kwatra, and Yanxi Liu. A comparison study of four texture synthesis algorithms on near-regular textures. In *ACM SIGGRAPH 2004 Posters*, SIGGRAPH '04, pages 16–, New York, NY, USA, 2004. ACM. ISBN 1-58113-896-2. doi: 10.1145/1186415.1186435. URL `http://doi.acm.org/10.1145/1186415.1186435`.

Yotam Livny, Soeren Pirk, Zhanglin Cheng, Feilong Yan, Oliver Deussen, Daniel Cohen-Or, and Baoquan Chen. *Texture-lobes for tree modelling*, volume 30. ACM, 2011.

Steven Longay, Adam Runions, Frédéric Boudon, and Przemyslaw Prusinkiewicz. Treesketch: interactive procedural modeling of trees on a tablet. In *Proceedings of the international symposium on sketch-based interfaces and modeling*, pages 107–120. Eurographics Association, 2012.

Karl Maritaud. *Rendu réaliste d'arbres vus de près en images de synthèse*. PhD thesis, Limoges, 2003.

M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, November 2014. ISSN 0162-8828. doi: 10.1109/TPAMI.2014.2321376.

Sören Pirk, Ondrej Stava, Julian Kratt, Michel Abdul Massih Said, Boris Neubert, Randomir Mech, Bedrich Benes, and Oliver Deussen. Plastic trees: interactive self-adapting botanical tree models. *ACM Transactions on Graphics*, 31(4):1–10, 2012.

Przemyslaw Prusinkiewicz and Aristid Lindenmayer. Modeling of trees, 1990. ISSN 1431-939X.

Hannes Ricklefs. Renderman bark shader, 2005.

Amir Rosenberger, Daniel Cohen-Or, and Dani Lischinski. Layered shape synthesis: automatic generation of control maps for non-stationary textures. *ACM Transactions on Graphics (TOG)*, 28(5):107, 2009.

Arthur Shek, Dylan Lacewell, Andrew Selle, Daniel Teece, and Tom Thompson. Art-directing disney's tangled procedural trees. In *ACM SIGGRAPH 2010 Talks*, page 53. ACM, 2010.

Eliyahu Sivaks and Dani Lischinski. On neighbourhood matching for texture-by-numbers. In *Computer Graphics Forum*, volume 30, pages 127–138. Wiley Online Library, 2011.

Ying Tang, Xiaoying Shi, Tingzhe Xiao, and Jing Fan. An improved image analogy method based on adaptive cuda-accelerated neighborhood matching framework. *The Visual Computer*, 28(6-8):743–753, 2012.

Xin Tong, Jingdan Zhang, Ligang Liu, Xi Wang, Baining Guo, and Heung-Yeung Shum. Synthesis of bidirectional texture functions on arbitrary surfaces. In *ACM Transactions on Graphics (ToG)*, volume 21, pages 665–672. ACM, 2002.

Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Deep image prior. *arXiv preprint arXiv:1711.10925*, 2017.

Ivan Ustyuzhaninov, Wieland Brendel, Leon Gatys, and Matthias Bethge. What does it take to generate natural textures? 2016.

Hugues Vaucher. *Tree bark: a color guide*. Timber Press (OR), 2003.

Xi Wang, Lifeng Wang, Ligang Liu, Shimin Hu, and Baining Guo. Interactive modeling of tree bark. In *Computer Graphics and Applications, 2003. Proceedings. 11th Pacific Conference on*, pages 83–90. IEEE, 2003.

Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 479–488. ACM Press/Addison-Wesley Publishing Co., 2000.

Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. State of the art in example-based texture synthesis. In *Eurographics 2009, State of the Art Report, EG-STAR*, pages 93–117. Eurographics Association, 2009.

Pierre Wilmot, Eric Risser, and Connelly Barnes. Stable and controllable neural texture synthesis and style transfer using histogram losses. *arXiv preprint arXiv:1701.08893*, 2017.

Qing Wu and Yizhou Yu. Feature matching and deformation for texture synthesis. *ACM Transactions on Graphics (TOG)*, 23(3):364–367, 2004.

Hui Xu, Nathan Gossett, and Baoquan Chen. Knowledge and heuristic-based modeling of laser-scanned trees. *ACM Transactions on Graphics (TOG)*, 26 (4):19, 2007.

List of Figures