FAKULTÄT FÜR !NFORMATIK

Texture Synthesis for Urban Data

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

ausgeführt von

Wolf Reitsamer
Matrikelnummer 9825486

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Wien, 22. 11. 2008

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Abstract

Although there has been a lot of research in synthesizing textures, it is still difficult to design an algorithm that is both: efficient and capable of generating high quality results.

Today it is impossible to imagine 3d rendering without using textures. Textures add details to models without increasing their complexity. Textures also have a wide range of application. They represent color, geometry, material properties, light and shadow and even natural phenomena like rain, smoke, fire, etc.

The creation of visually convincing textures always is a tightrope walk between performance and quality. Depending on whether one is doing implicit or explicit texture synthesis, performance might or might not be the decisive criterion. Furthermore factors like the type of texture being synthesized as well as given constraints may have an important impact on both: performance and quality of the synthesized output.

Synthesized textures however should never look as synthesized to the observer.

Various techniques for synthesizing textures along with several acceleration methods are investigated in this thesis.

Kurzfassung

Obwohl im Bereich der Texture-Synthese bereits großer Forschungsaufwand betrieben wurde, ist es nach wie vor schwer einen Algorithmus zu finden der sowohl effizient als auch im Stande ist qualitativ hochwertige Ergebnisse zu erzeugen.

Heutzutage ist es undenkbar 3d rendering ohne Texturen zu betreiben. Texturen fügen Modellen Detailinformationen hinzu ohne dabei deren Komplexität zu erhöhen. Texturen haben darüber hinaus ein breites Anwendungsspektrum. Sie repräsentieren Farben, Geometrie, Materialeigenschaften, Licht und Schatten und sogar natürliche Phänomene wie Regen, Rauch, Feuer. etc.

Die Erzeugung von visuell ansprechenden Texturen ist ein Drahtseilakt zwischen guter Performance und Qualität. Je nachdem ob jemand implizite oder explizite Texture-Synthese betreiben möchte, spielt die Performance selbst eine entscheidende oder aber untergeordnete Rolle. Davon abgesehen können Faktoren wie die Art der zu synthetisierenden Textur oder definierte Rahmenbedingungen entscheidenden Einfluß sowohl auf die Performance als auch die Qualtität der synthetisierten Ausgabe haben.

Synthetisierte Texturen sollten in keinem Fall synthetisiert aussehen.

In dieser Diplomarbeit werden mehrere Techniken zum synthetisieren von Texturen zusammen mit einer Vielzahl an Beschleunigungsmethoden untersucht.

Acknowledgments

First and foremost I would like to thank Michael Wimmer who has been my supervisor for this diploma thesis. While giving me the freedom to explore the subject, he provided me with useful hints and information when things seemed to be stuck.

I also want to thank Peter Wonka who invested time and energy to review my approaches and statements and who introduced a lot of good ideas.

I particularly want to thank my grandfather Franz Reitsamer who bought my very first computer and who always has been a good friend and role model to me.

I want to thank my godmother Friederike Kotz, who always believed in me and who has always been there for me. She died way too early while i was working on this thesis.

I want to thank my parents Gerhard and Gerlinde for their endless support and patience at all times.

Special thanks go to Christina Ipser, who underwent all my ups and downs during the time working on this thesis.

I also want to thank Matthäus Müller for being a fast friend through all the years of my studies and for lifting my spirits more than once.

I want to thank my friends Andreas Doulos, Thomas Illetschko, Peter Schöbel and many more as well as the rest of my family for their encouragement, the layout tips, the evenings playing FIFA and the numerous sushi sessions.

Last but not least i would like to thank Alexander Burker who always kept me informed about deadlines and other organizational issues.

Inhaltsverzeichnis

Chapter 1

Introduction

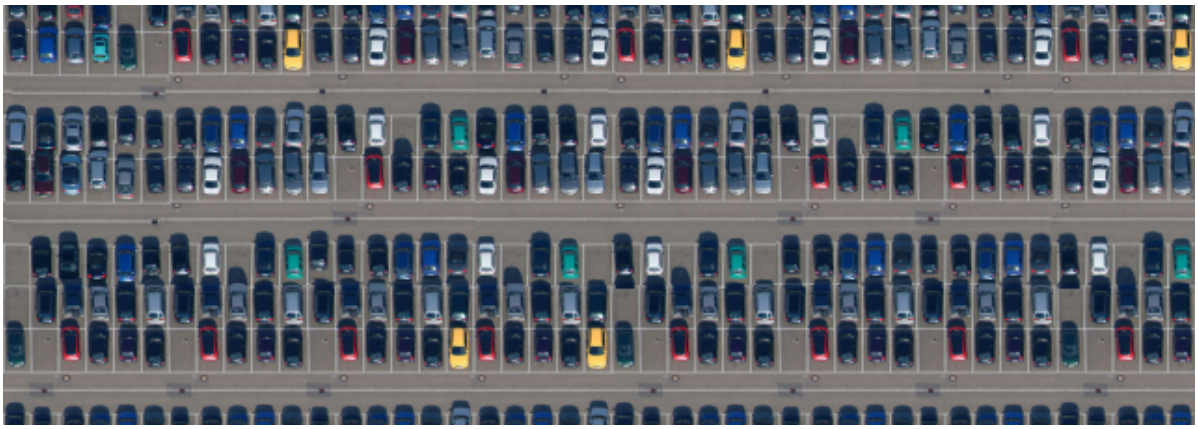


Figure 1.1: Example of a synthesized texture using our implementation of [KSE+03]. More information about this image can be found in Chapter 7

One of the oldest tricks in 3d computer graphics is texture mapping. Pioneered 1974 by Edwin Catmull in his Ph.D. thesis [Cat74], textures add details to models without increasing their complexity. Texture mapping itself is straightforward and important for a wide variety of applications in computer graphics and image processing. Textures can be subdivided into five categories: *regular*, *near-regular*, *irregular*, *near-stochastic* and *stochastic*. Figure 1.2 shows examples for all five categories.

Depending on the type of input texture, there are different approaches to synthesize a larger output, where all algorithms at least aim the following requirements:

1. Similarity between the input and the output image

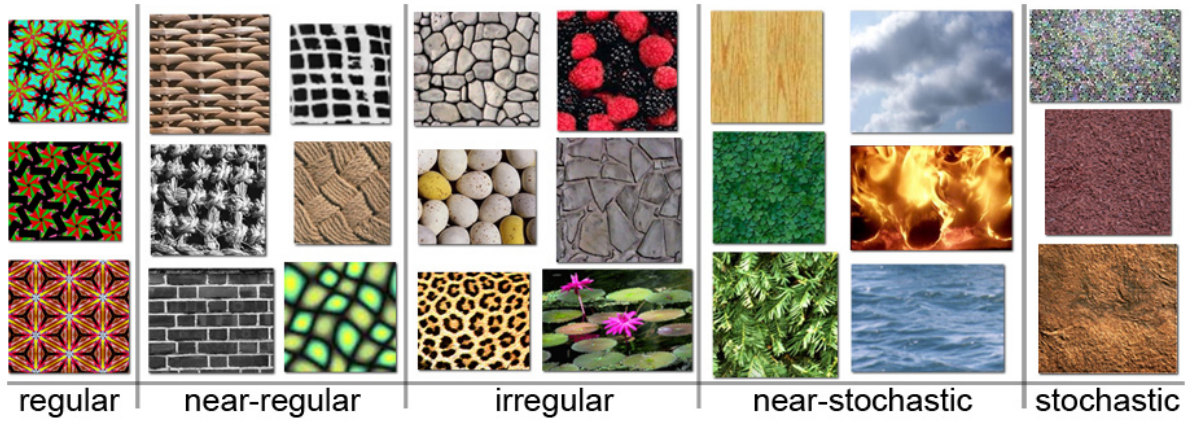


Figure 1.2: Classification of textures. The image was created as part of [LLX+01].

2. Non-repetitive production
3. No visible artifacts (like seams, blocks, etc.)

While earlier approaches in texture analysis and synthesis tried to directly simulate the physical generation process (e.g. weathering and mineral phenomena) or by modeling biological patterns (e.g. fur, skin, etc.), all algorithms dealt within this thesis start with a given input sample and synthesize a texture that is sufficiently different from the given input. Nevertheless as stated above, the output texture has to appear as generated by the same underlying stochastic process.

Preserving local features of the input image while at the same time avoiding repetitive structure in the output image is a key aspect of synthesis strategies.

Various different texture synthesis techniques evolved over the years. There are a number of distinctive features that can be used to categories the different approaches. Some of those features are listed in Chapter 1.2.

The main differentiating factor this work concentrates on is the sampling method. The two main categories we concentrate on are pixel- and patch-based algorithms. We will have a closer look at different approaches in those two categories.

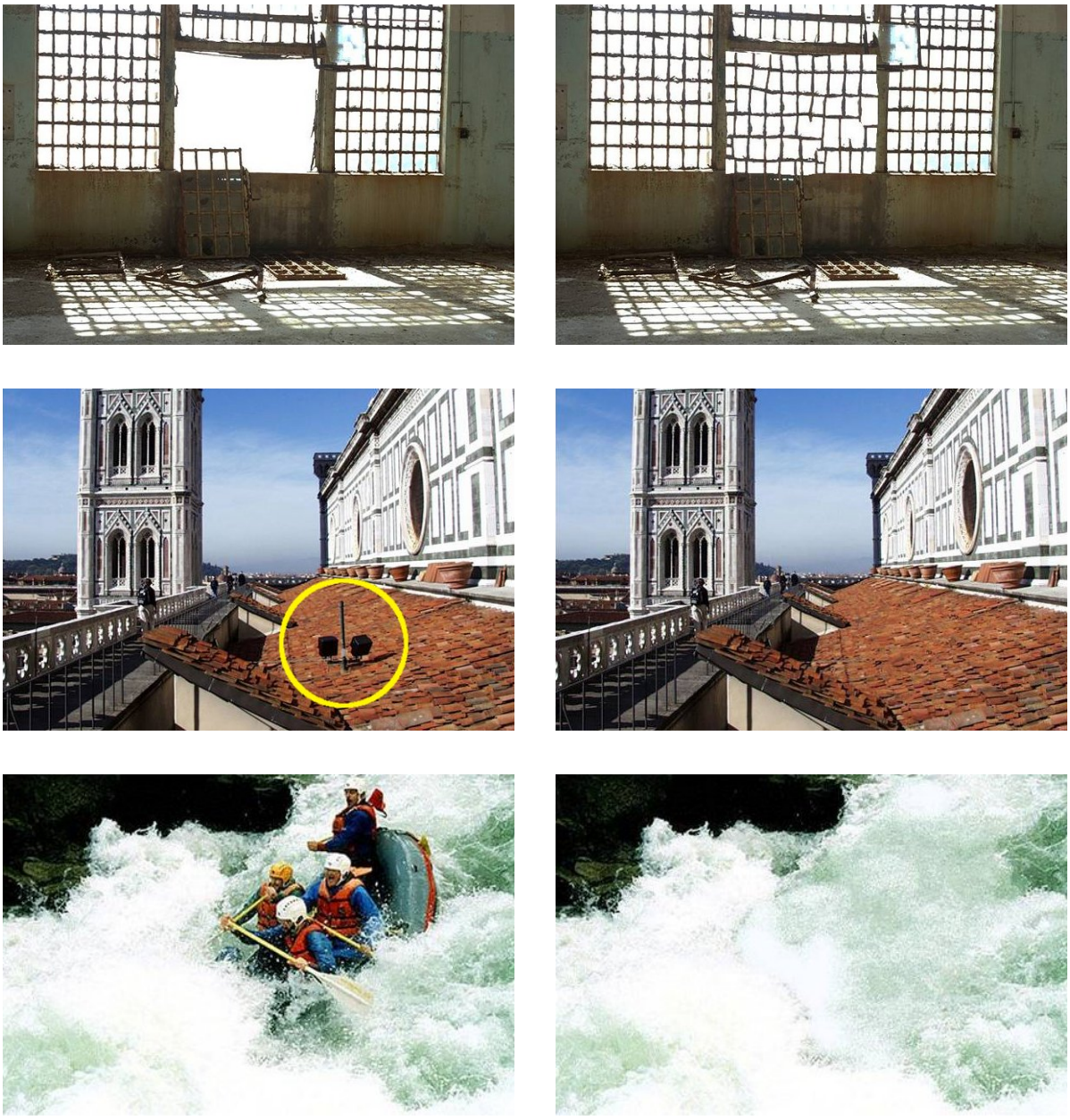


Figure 1.3: Detail Hallucination, Artifact Removal and Image Editing. The images above can be found on Vivek Kwatra's homepage at http://www.cs.unc.edu/ kwatra/.

1.1 Applications

There is a wide range of application areas for texture synthesis. The first that springs to mind is game programming with ample levels and huge objects like mountains, rows of houses, alleys made of cobblestone, ...

As a matter of fact only a fraction of texture synthesis techniques meet the real-time requirements crucial to real-time 3-D games. Nonetheless in the range of image processing real-time performance is not the decisive factor. The following examples just give an impression of what can be done with texture synthesis.

1.1.1 Hole Filling or Image Editing

As the name already suggests, *Hole filling* is a process where missing information can be generated. Hole filling can be understood as general term for a variety of other techniques like *Detail Hallucination*, *Object and Artifact Removal* or *Image Extrapolation*.

While *Object and Artifact Removal* is aimed at erasing data from a given image followed by reassembling appropriate content, *Detail Hallucination* is directed at augmenting images at settled locations. In other words the reason for performing *Object and Artifact Removal* is that you have an object in your image you want to remove, whereas your motivation for *Detail Hallucination* is missing information.

Another sub-category of *Hole Filling* is *Image Extrapolation*, where an image is enlarged at its borders by sampling from itself. This can be of interest for handling the boundary problem when convolving images. Typically this is handled using methods like zero-fill, tiling or reflection. Those workarounds, however, might introduce discontinuities not present in the original image.

1.1.2 Image Combining

Combining two (or more) images with each other is a technique, that naturally arises from the way patch-based texture synthesis works. The challenge to combine several

Figure 1.4: Image Combining: the left image and the image in the middle are combined to form the image to the right. The images above are part of the original publication [KSE+03].

blocks with each other is very similar to the functioning of [EF01] and [KSE+03]. *Image Combining* is an interactive operation, where the user selects a region of image $\boldsymbol{A}$ that should be preserved and combined with image $\boldsymbol{B}$. The selected algorithm (e.g. *MEBC* or *GraphCut*) the tries to find an optimum transition between $\boldsymbol{A}$ and $\boldsymbol{B}$ within a preset overlap region.

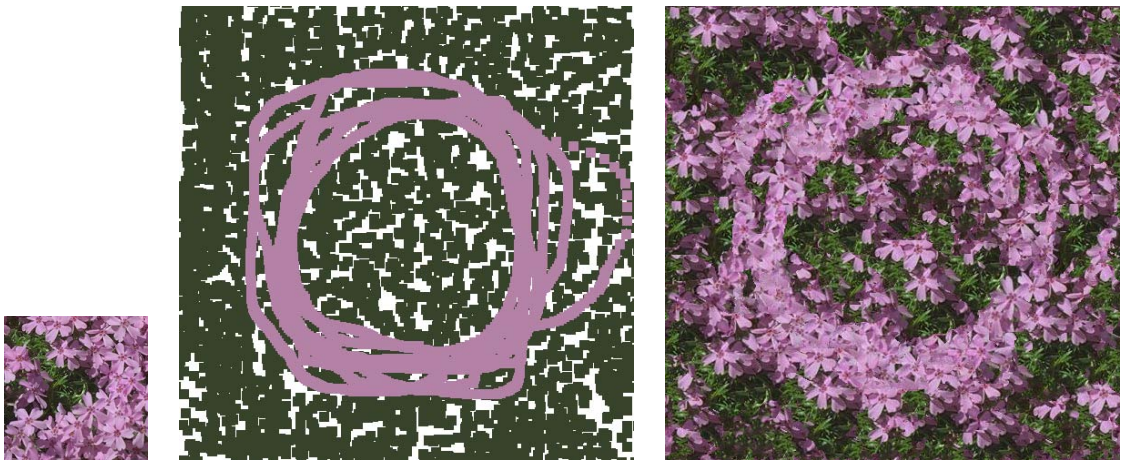
In the majority of cases it will make sense that the position of the adopted patch is specified by the user. After all there is no reason why an algorithm would not be able to find the best matching offset for a new patch, however the result would rarely be satisfactory for photographs as for instances for the images in Figure 1.1.2.

1.1.3 Texture Transfer

Texture transfer is the name of a technique introduced in [EF01]. It takes two images and combines their properties into one final synthesized output. A more detailed explanation how Efros & Freeman achieve their final results is given in Section 2.2.2.

Unlike image combining, texture transfer is not a typical application of patch-based texture synthesis approaches.

The idea of transferring one texture into another or in other words *constraining* the output to possess certain properties or patterns can be found in publications based on pixel-based synthesis methods as well. [Ash01] for instance describes a technique for creating user controlled results by providing an additional image, called *target image*. Ashikhmin's approach is described in further detail in Section 2.1.2.



a) input b) target image c) output image

Figure 1.5: User controlled texture synthesis by Michael Ashikhmin. The input image in a) along with the target image in b) forms the output image c). The target image affects the color balance in c). The images are part of the original publication [Ash01].

Figure 1.5 shows an example of the user controlled synthesis process described in [Ash01].

1.2 Terminology

1.2.1 Markov property

One of the most elementary insights in texture synthesis is the fact, that textures can be understood as product of *Markov chains*.

Markov chains are a special case of stochastic processes. The key aspect of a Markov chain is, that the description of the present state of a process fully captures all the information that could influence its future evolution. Such systems are said to have the *Markov property*.

In regards to synthesizing textures this property in the broader sense means, that each pixel of a texture is influenced only by its immediate neighborhood, but not by pixels

outside this isolated region. Such systems are also refered to as *Markov random fields (MRF)* or *Markov networks*. The coherence between the primary definition and its usage in texture synthesis (or general image processing) is the time limitation first and the regional interaction second.

The Markov property is the fundament for virtually all texture synthesis approaches, since all (stochastic) texture synthesis methods strive to preserve local features of a sample texture in the synthesized output image.

1.2.2 The L_2 norm

Most approaches in texture synthesis use the L_2 norm, also called Euclidean norm or L_2 distance. Given a vector $x = (x_1, x_2, ..., x_n)$ the Euclidean norm determines the ordinary distance from the origin to the point x, as a consequence of the Pythagorean theorem. The norm is the intuitive notion of the length of a vector and is defined as follows:

$$\|x\| = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

The set of all vectors with norm 1 define the unit circle in $\mathbb{R}^2$ and the unit sphere in $\mathbb{R}^3$. Assigned to our problem of comparing color information, given an RGB color space and two neighborhoods N_1 and N_2 the similarity between N_1 and N_2 can be calculated like this:

$$D(N_1, N_2) = \sum_{p \in N} (R_1(p) - R_2(p))^2 + (G_1(p) - G_2(p))^2 + (B_1(p) - B_2(p))^2 \qquad (1.1)$$

where R, G and B are the red, green and blue color components at position p. The Euclidean norm however better applies to the L*a*b color space, since that color space was designed to better approximate human vision. In CIELAB the length of a vector v pointing from one color to a second color determines the perceived similarity between the two colors. CIELAB will be described in Section 3.3.1 in more detail.

1.3 Thesis Objectives

There are three objectives pursued in this thesis.

Objective one is to investigate several available algorithms and existing synthesis strategies and analyze the advantages and drawbacks of the different approaches over each other. Furthermore we are interested in the acceleration methods used in the examined papers.

Beyond that we want to go into further detail on two advanced approaches, [KSE+03] and [LH05] and deliver insights into our implementations. We will summarize our attempts to slightly modify their functioning, provide run-time behavior and talk about adjusting the input parameters to our needs.

Last but not least we want to analyze the aptitude of our implementation of [KSE+03] and [LH05] to synthesize textures for urban data.

1.4 Examined Papers

A lot of work has been dedicated to the topic of synthesizing textures during the last decade. The list of interesting approaches is long. As it is impossible to cover all the literature on this topic, only a handful of publications that appear to be most relevant will be examined.

As this work concentrates on pixel- and patch-based approaches, we will briefly touch on Alexei A. Efros and Thomas K. Leung's work on "Texture Synthesis by Non-parametric Sampling" [EL99], an early publication in the field of order-dependent pixel-based texture synthesis.

Based on their work the very popular publication by Li-Yi Wei and Marc Levoy about "Fast texture synthesis using tree-structured vector quantization" [WL00] will be covered in more detail for the following reasons: Due to its simplicity it is well-suited to outline the general functioning of pixel-based texture synthesis methods. It is one of

the pathbreaking, early algorithms that is cited by most publications on this topic. Additionally it describes two ways to improve and accelerate the basic algorithm, which are also part of this thesis.

An interesting modification of [WL00] is given by Michael Ashikhmin's approach called "Synthesizing natural textures" [Ash01] in which the author concentrates on improving the quality of the result for input samples with arrangements of small objects that usually occur in natural textures.

Sylvain Lefebvre and Hugues Hoppe conclude the pixel-based approaches that are covered here with their order-independent, real-time synthesis strategy called "Parallel controllable texture synthesis" [LH05]. This publication is also covered in more detail in Chapter 6 about special aspects of our implementation.

The second main category examined in the course of this thesis are patch-based synthesis methods.

Again we start by summarizing simple ways of doing this, beginning with an approach by Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo and Heung-Yeung Shum called "Real-time texture synthesis by patch-based sampling" [LLX+01], which performs simple blending to hide visible artifacts along patch transitions.

In addition we will have a closer look on "Image Quilting for Texture Synthesis and Transfer" by Alexei A. Efros and William T. Freeman [EF01] and go into some detail on their way to find optimum patch borders using an algorithm called *Minimum Error Boundary Cut (MEBC)*, a similar approach as used in "Graphcut textures: image and video synthesis using graph cuts" by Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk and Aaron Bobick [KSE+03].

[KSE+03] is the highlight of the patch-based approaches covered in this work. The key difference to *MEBC* is that [KSE+03] can "remember" old seams and therefore is able to correct bad transitions in later iterations of the algorithm.

Chapter 4 is dedicated to algorithms that neither use pixel- nor patch-based techniques exclusively.

"Hybrid texture synthesis" by Andrew Nealen and Marc Alexa [NA03] take up the idea of [XGS00] and present a combination of pixel- and patch-based texture synthesis.

Michael F. Cohen, Jonathan Shade, Stefan Hiller and Oliver Deussen finally conclude our survey with their paper about "Wang Tiles for image and texture generation" [CSHD03].

1.5 Structure of this thesis

This work is structured as follows:

Chapter 2 introduces several basic algorithms, pixel- and patch-based. The purpose of this chapter is to demonstrate the way the two sampling methods work.

Chapter 3 concentrates on two more advanced synthesis techniques, based on concepts elaborated in Chapter 2. In addition, implementation details and issues are appended to both approaches.

Chapter 4 outlines the ideas of two publications that neither use pure pixel- nor pure patch-based texture synthesis.

Chapter 5 provides several optimizations for quality improvements and performance enhancements that are used in the papers elaborated in this work.

Chapter 6 gives an insight in special aspects of our implementations of the publications presented in Chapter 3, [KSE+03] and [LH05].

Chapter 7 covers various aspects of synthesizing textures for urban data. It shows a variety of synthesized outputs and gives a brief overview on input parameters and run-time behavior.

Chapter 8 summarizes the findings of this thesis and gives a brief outlook.

Chapter 2

Early Approaches

To demonstrate the general functioning of pixel- and patch-based texture synthesis, some of the most popular, yet simplest algorithms will be presented in this chapter. We will start each section by only briefly touching on a publication that explains the basic idea behind the presented synthesis method and expand the model by having a closer look on an approach based on the initial paper. Each algorithm covered in this chapter is order-dependent and therefore not state-of-the-art. By providing a brief overview of the domain and gaining insight into special aspects of synthesizing textures, pixel- and patch-based, this chapter is meant as a prequel for the techniques presented in Chapter 3.

2.1 Pixel-based Sampling

Unlike patch-based algorithms, where whole areas of the input texture are copied to the output texture (see Chapter 2.2), pixel-based algorithms fill the output pixel-wise by comparing their neighborhood informations. While most earlier approaches were order-dependent, there now exist more advanced algorithms that exploit parallelism with runtime properties close to real-time. One such order-independent pixel-based synthesis approach, [LH05], will be described later in Section 3.2.

One of the earliest and at the same time simplest pixel-based algorithms is the approach made by Alexei A. Efros and Thomas K. Leung in 1999 [EL99]. The underlying idea is inspired by an article published in 1948 by Claude Shannon [Sha01] about

modeling language as a generalized Markov chain. Shannon's idea was that by reading in enough data and therefore using large samples of the language, one could find a matching word when building sentences by using probability distributions.

Broadly speaking Efros & Leung's approach works like this:

Starting with an initial seed, the algorithm "grows" the texture in an onion skin order one pixel at a time. It therefore uses partial neighborhood search, where the size of the neighborhood can be chosen by the user, depending on how stochastic the user believes the sample texture to be. To grow an output texture from scratch, Efros and Leung use a 3×3 seed randomly taken from the sample image.

Although Efros & Leung's algorithm works well for a wide variety of textures ranging from regular to stochastic, it breaks down for others. Actually the authors admit themselves that their algorithm runs the risk of occasionally "slipping" into a wrong part of the search space and thereupon starts to grow "garbage".

2.1.1 Fast texture synthesis using tree-structured vector quantization

Wei and Levoy's approach [WL00] is similar to [EL99], but instead of working with neighborhood sizes chosen by the user, it works with fixed-sized neighborhoods, which is essential for their acceleration methods as will be seen later. Furthermore it works in a scan-line order.

[WL00] starts with a small input texture, the so called *exemplar*, and generates a (usually larger) similar output texture. It generates textures through a deterministic search process which is derived from Markov Random Field models. The search process is accelerated using tree-structured vector quantization (TSVQ). In Section 5.1 we will have a closer look at TSVQ.

[WL00] however suffers from the same problem as [EL99], i.e. their algorithm breaks down for some textures.

The two major components of the algorithm are a multi-resolution pyramid data structure and a simple search algorithm.

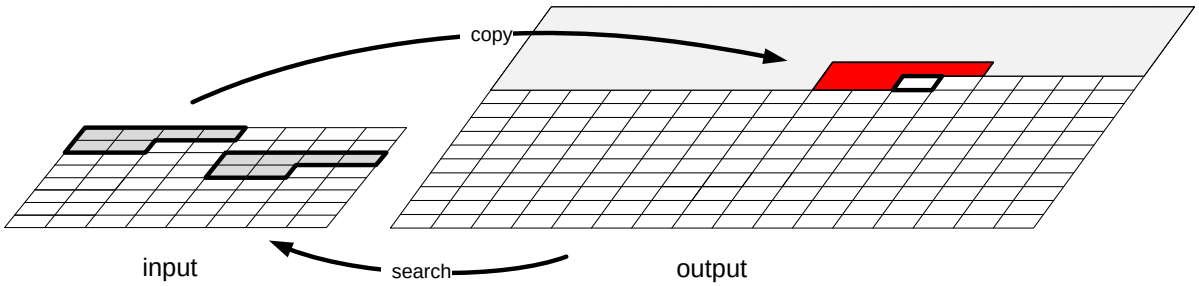


Figure 2.1: Basic neighborhood search as used in most order-dependent pixel-based synthesis approaches. The L-shaped region is used to compare the pixel-neighborhoods of the input and output images with each other.

The new texture is generated pixel by pixel, where each pixel is determined in a way that local similarity is preserved between the example texture and the resulting image. The synthesis process is completely deterministic.

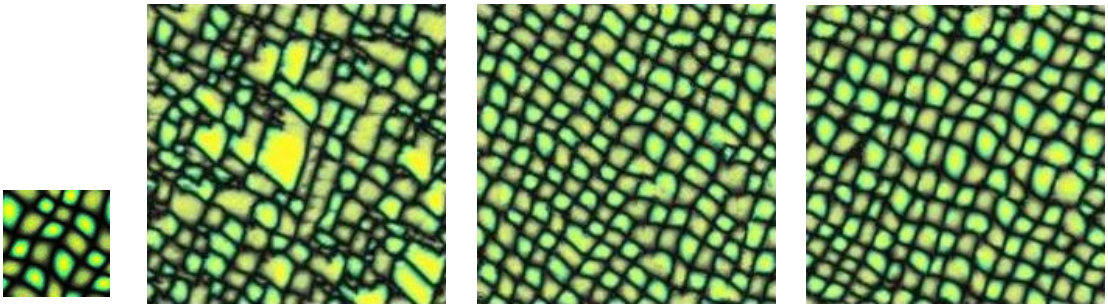
The basic algorithm

The algorithm starts with an input sample image I_{in} and a random noise output image. By transforming the output image I_{out} pixel by pixel, I_{out} is forced to look like I_{in}. The transformation takes place in a raster scan order.

To determine the value of a pixel p in I_{out} its spatial neighborhood - that is the L-shaped region in Figure 2.1 - is compared against all possible neighborhoods in the input image. The input pixel with the most similar neighborhood is assigned to p.

To measure the similarity between two neighborhoods the sum of squared differences is used as a cost function. The goal of this synthesis process is to ensure that the newly assigned pixel will maintain as much local similarity as possible. This process is repeated for each output pixel until all pixels are initialized.

The authors compare this process to putting together a puzzle where the pixels are the individual pieces and the differences of the surrounding neighborhood pixels form the fitness between those pieces.



a) input b) 5×5 (single res.) c) 11×11 (single res.) d) 5×5 (multi res.)

Figure 2.2: Results with different sizes of neighborhoods. While the overall structure is completely destroyed in b), the multi-resolution approach used to synthesize d) yields even better results than the single-resolution 11×11 pixels wide neighborhood in c). The images above can be found on Vivek Kwatra's homepage at http://www.cs.unc.edu/ kwatra/.

The quality of the synthesized texture depends on the size and shape of the neighborhood. The size of the neighborhood should be (at least) the size of the largest regular texture structure. Otherwise the structure of the input image may be lost and the resulting output image may look too random. Figure 2.2 shows this effect.

The shape of the neighborhood will directly determine the quality of the output image. It may only contain those pixels preceding the current output pixel (in a raster scan ordering). The reason for this is to ensure, that each output neighborhood will include only pixels, that are already initialized.

As one can simply tell, that condition is infeasible for the first few rows and columns. Those pixels may contain unassigned noise pixels, but as the algorithm processes all other neighborhoods will be completely valid.

If we would include pixels that were uninitialized, in other words not only use preceding pixels for calculating the neighborhood, the output image would never look like the input image.

Thus the noise image is only used when generating the first few rows and columns of the output image. After this it is ignored.

The Multi-Resolution Approach

One way for both, accelerating the synthesis process as well as improving the output quality of the algorithm, is by using a multi-resolution pyramid. This technique is also essential for tileablity and edge-handling in general, since the singular resolution causal neighborhood only contains pixels above a pixel p in a scan-line order. That is why the vertical tileability may not be enforced. A multi-resolution neighborhood, which contains symmetric regions at lower resolution levels, avoids this problem.

Another difficulty that can be solved using multi-resolution synthesis is edge handling. Therefore the image edges are treated toroidally. This means that the first pixel in a row is treated as the right neighbor of the right-most pixel of the very same row. The pixel below a pixel in the last row is the pixel in the first row of the same column.

Using multiple resolutions of the input and synthesized output also increases the chance that the pattern of the input texture will be preserved, while using small neighborhoods.

The problem with the singular resolution approach is that input textures that contain large scale input structure need to use large neighborhoods. Large neighborhoods demand more expensive computations. Using multi-resolution pyramids solves this problem, because large scale searches can be represented more compactly by less pixels in a lower resolution of the pyramid. Figure 2.2 shows this effect by opposing two output images created with the single resolution approach to one generated using multiple resolutions. Notice that the image synthesized with the multi-resolution strategy uses the same neighborhood size as the first image does. Nevertheless the produced image looks even smoother than the output synthesized using the relatively large 11×11 neighborhood.

The extended algorithm

Gaussian pyramids are built by successive filtering and downsampling the original input image. This is done by convolving the image using the bell-shaped gaussian curve. Each level of the so created image pyramid is a blurred and decimated version of the original image.

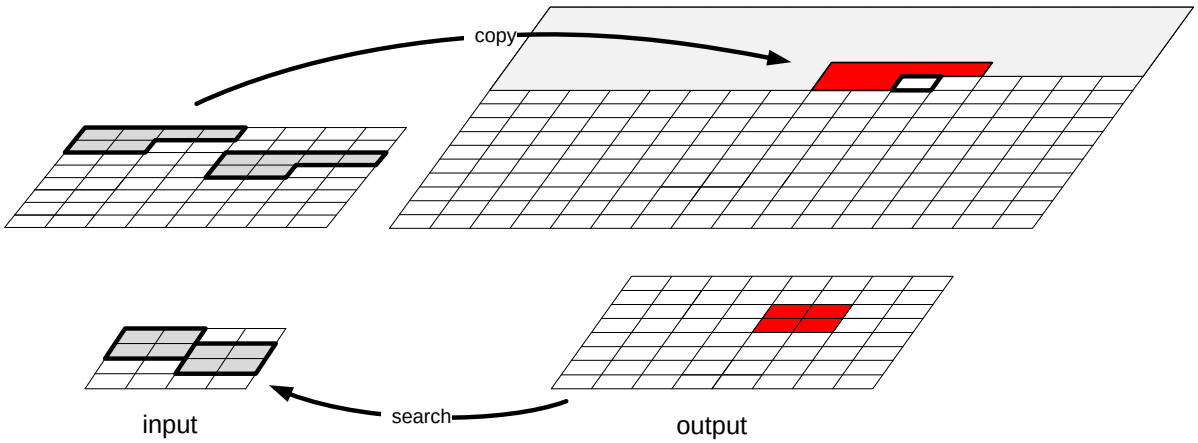


Figure 2.3: Neighborhood search in multi-resolution extension. Not only is the L-shaped neighborhood of the current level of the image pyramid compared to the neighborhoods of the input image, but also the square pixel-neighborhood of the next coarser level.

Two image pyramids are built for the multi-resolution approach: one from the input, one from the output texture respectively. The pyramid of the output is transformed from lower to higher resolution, such that each higher resolution level is constructed from an already synthesized lower level. Apart from that, the pixels are synthesized in a way similar to the single resolution case. The only modification is that in the multi-resolution algorithm each neighborhood contains pixels in the current resolution as well as pixels in a lower resolution. Figure 2.3 illustrates this process.

Wei and Levoy accelerate their algorithm furthermore by using tree-structured vector quantization (TSVQ), which will be described in Section 5.1.

Conclusion

Since the algorithm is order-dependent, a different order leads to a different result. There is no chance for parallelism. Using multiple-resolution pyramids and TSVQ speeds up the algorithm somehow. The paramount benefit for this thesis however is the simplicity of the algorithm, that depicts how synthesizing textures using pixel-based methods works.

Wei and Levoy, aware of the limitations of their order-dependent algorithm, improved their initial work and published a new approach named "Order-Independent Texture Synthesis" [WL03] which was submitted to, but rejected by SIGGRAPH in 2003.

2.1.2 Synthesizing natural textures

Ashikmin's publication, "Synthesizing Natural Textures" [Ash01], is a modification of Wei and Levoy's basic algorithm. It enhances their strategy at two points: first by improving the quality of the output for specific types of textures, second by enabling direct control over the texture synthesis process by accepting a user defined target image.

Ashikmin's synthesis strategy not only yields better results for quasi repeating patterns of irregular size, like flower-fields, bushes, etc., but is also significantly faster than Wei and Levoy's basic algorithm.

The difference between two neighborhoods is expressed using the L_2 distance norm, which already has been defined in Section 1.2.2.

The size of the neighborhood is crucial to the quality of the generated output. On the other hand it critically effects the run-time performance of the algorithm.

Textures created with Ashikhmin's approach consist of irregular shaped patches of the input sample that are not so obvious to an observer. The key difference between [WL00] and [Ash01] is that Wei and Levoy start a new neighborhood search for each new pixel, while Ashikhmin keeps track of similar neighborhoods that are used as valid candidates. By starting the search process from scratch for each new pixel, it is garantueed that the copied pixel matches the given neighborhood best.

Ashikhmin's way to find candidate pixels is to re-use the neighborhoods of already synthesized pixels. This is done by storing the offsets of pixels within the input sample along with their color information. When synthesizing a new pixel, the information of the already synthesized pixels is extracted and the new neighborhood is compared with the neighborhoods of the stored offsets in the input image. While other approaches treat input images toroidally and thereby solve the problem of border pixels, Ashikhmin only stores the offsets of pixels thats neighborhoods completely lie inside

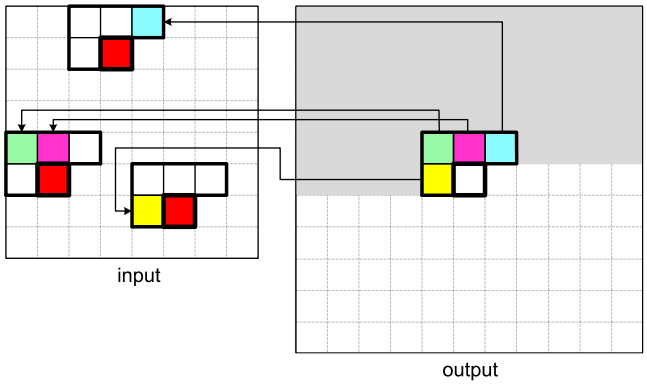


Figure 2.4: Basic Ashikhmin search. In addition to the color information Ashikhmin stores the original offsets of the synthesized pixels and uses those offsets when synthesizing subsequent pixels to find the new candidates.

the input image. If the stored candidate has a neighborhood not completely in the input image, the candidate will be replaced by a new pixel at a random position. The algorithm therefore works somehow similar to k-coherence search, which is described in Section 5.4. Figure 2.4 shows the process using the stored offsets to find the best matching neighborhoods.

In the beginning of the algorithm the array holding the offset locations of the pixels already used is initialized to random positions before the synthesis process begins. Extending the algorithm to use the multi-resolution synthesis strategy of Wei and Levoy is of limited use in Ashikhmin's approach. While the multi-resolution extension in Wei and Levoy allow us to use smaller neighborhoods to obtain the same or better quality than with a large neighborhood, Ashikhmin's publication hardly benefits from using multiple resolutions.

The algorithm

1. Initialize the array of pixel offsets to random positions inside the input image
2. Apply an L-shaped neighborhood to the next uninitialized pixel of the output texture
3. Read the offset from the offset array for each pixel of the neighborhood to generate candidate pixel locations

4. Remove duplicate candidates
5. Compare the neighborhoods of each candidate and choose the candidate with the most similar neighborhood
6. Copy the just calculated candidate pixel to the output image and store the offset in the offset array
7. If the output image is not fully initialized, jump to step 2

Ashikhmin furthermore describes an interesting way of manipulating the generated output texture by including a user defined image mask in the initial algorithm. Image masks can be hand-drawn images that outline general properties the resulting texture should have. The beauty of Ashikhmin's approach for giving the user control over the synthesis process lies in its simplicity.

To realize user control, we first need to change the shape of each pixel neighborhood. That means instead of using the L-shaped neighborhood known from [WL00], we now use a complete square neighborhood. Finding a candidate pixel now is done in two steps: step 1 equals the process described before, where the L-shaped neighborhood in the upper part of the neighborhood were compared pixel-wise using the L_2 norm. In step 2, however, we now add the L_2 difference between the bottom L-shaped neighborhood of our candidate pixels in the input texture and valid pixels of the bottom half of the L-shaped neighborhood of our output texture. For this second step of the algorithm we only accept pixels created by the user. Hence the algorithm does exactly the same things as before for regions without user input. In other words: using an empty image for the now modified algorithm results in the same synthesized texture as before without the modification. Figure 2.5 illustrates the process of using a target image to control the synthesis process.

The degree of similarity between the user input mask and the created output image can be controlled by using multiple iterations. Instead of initializing the array of candidates in each new pass as we did before, the offsets stored in these arrays are now used for each new iteration. After running the algorithm one time, each pixel in the output image is treated as a valid pixel. Thus the selection process for candidate pixels is solely based on square neighborhoods in each successive pass.

Figure 1.5 shows a result of this procedure.

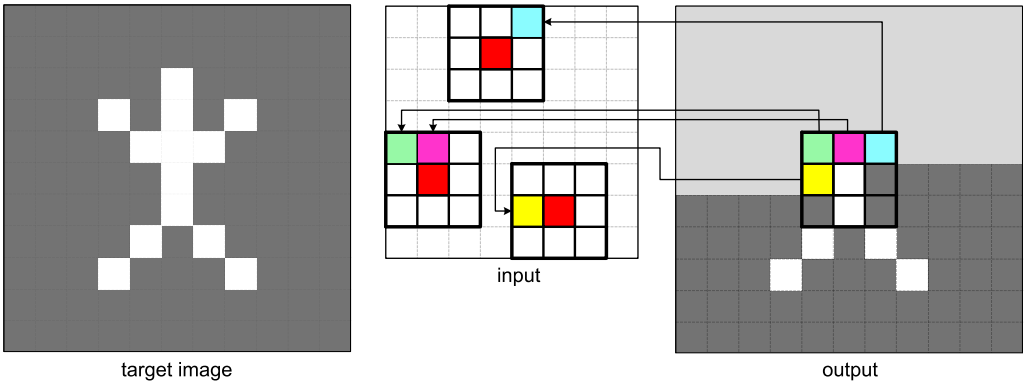


Figure 2.5: User controlled sampling uses a square neighborhood instead of the common L-shaped neighborhood. The thereby added pixel neighborhood is compared against the corresponding parts in the input image, whereas only valid user input is considered.

Conclusion

The goal of Ashikhmin's publication was to develop a special-purpose algorithm that yields better results for input textures consisting of arrangements of small objects of familiar but irregular shapes, often found in "natural" textures.

As any other synthesis method, Ashikhmin's technique to synthesize textures is far from being a general-purpose tool for texture synthesis. Considering the special purpose of his publication, however, it is a very interesting, yet very simple modification of the very popular algorithm of Wei & Levoy. Furthermore it shows impressively that even small and sometimes simple modifications can lead to much better results provided that one knows for what domain he wants to use an algorithm for.

2.2 Patch-based Algorithms

All patch-based algorithms have one thing in common: instead of finding one perfectly matching pixel within the input sample texture I_{in}, whole texture patches are used as building blocks for synthesizing an output texture I_{out}. The major difference between

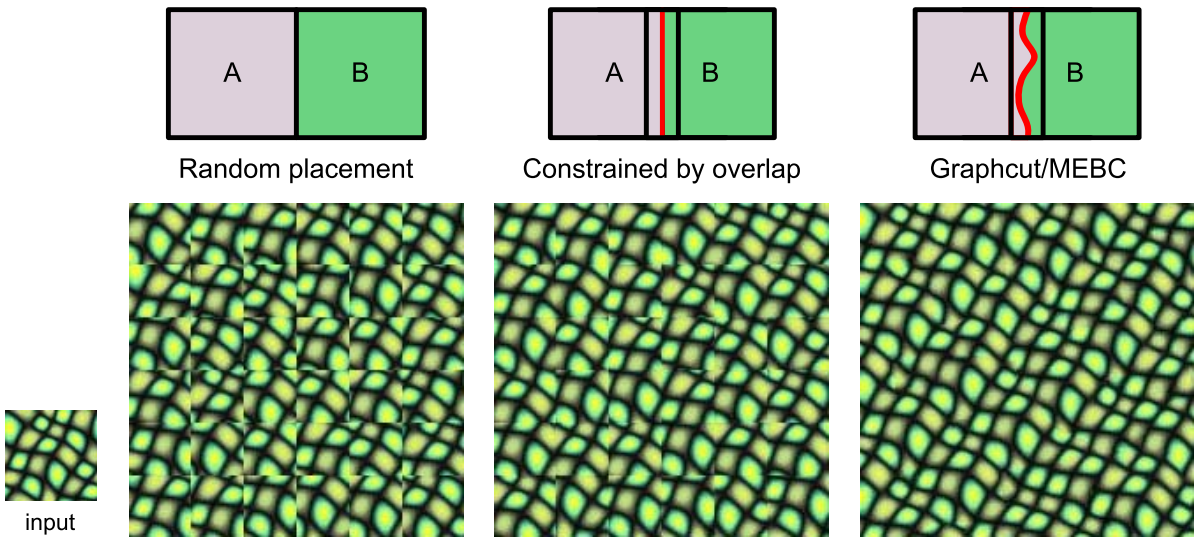


Figure 2.6: Evolution of patch placement. While the first output image uses a random arrangement of patches from the input image, the second output already choses patches that fit best when slightly overlapping. The rightmost image extends this strategy by finding an optimum cut in the overlapping area.

the different approaches lies within the placement strategy as well as the treatment of the occurring patch-boundaries.

The problem with most patch-based algorithms is that seams primarily run horizontally or vertically, which makes the artifacts more noticeable. Thats why using textures showing brick walls, as done in many publications, is not really a prove for the strength of a patch-based algorithm.

The "evolution" of patch-based algorithms could be thought of as follows:

Given an input texture one could copy square blocks of fixed size randomly from an input sample to the output image. As a first improvement, the blocks chosen from the input sample could be restricted to blocks that somehow match a cost function for an overlap region. In other words: only patches with a good match are chosen. As a second refinement step instead of just shifting the selected input patch over the existing output, one could try to find a optimal transition between the two overlapping patches. These three ideas for placing and selecting patches are illustrated in Figure 2.6.

[EF01] and [KSE+03] try to contribute to this third placement method, with their "Minimum Error Boundary Cut" strategy and the "Graphcut" algorithm respectively.

We will start with two basic algorithms and go into further detail on [KSE+03] later.

2.2.1 Real-time texture synthesis by patch-based sampling

Liang et al. [LLX+01] try to avoid mismatching features across patch boundaries by using square patches of a prescribed size w_B x w_B, where each new patch is simply blended over existing ones. The new patch B_k is chosen based on the patches $B_0, ..., B_{k-1}$ already pasted into I_{out} and constrained by a user defined maximum distance δ_{max}.

Formally expressed, this means:

Let I_{R_1} and I_{R_2} be two texture patches of the same shape and size and let d define a distance function that measures the difference between those patches. We then say, that I_{R_1} and I_{R_2} match if and only if $d(I_{R_1}, I_{R_2}) < \delta_{max}$ where δ_{max} is a prescribed constant.

Each patch has an overlap region of size w_{OR}. When placing a new patch B_k into I_{out} the boundary zones of the already existing patches $(B_0, ..., B_{k-1})$ partly overlap with the boundary zone of the new patch. Two boundary zones are said to match if they match in their overlapping region. For the sake of randomness a set of all matching overlap regions is formed before one patch is randomly chosen to be pasted into I_{out}.

The algorithm

Let $B_{(x,y)}$ denote a patch at position (x,y) in an input image I_{in} to be copied to an output image I_{out}. Let further be O_{B_k} be the overlapping region of B_k, the k-th patch selected from the input image with a width of w_{OR}. Each patch already pasted into I_{out} also has a boundary zone, identified as O^k_{out}, where k again denotes the k-th patch. Using a distance function d that calculates the Euclidean distance as described in in Section 1.2.2, the following definition forms a set of patch candidates to be copied to I_{out} in the next iteration:

$$\Psi_B = \left\{ B_{(x,y)} | d(O_{B_{(x,y)}}, O^k_{out}) < \delta_{max}, B_{(x,y)} \subseteq I_{in} \right\}$$

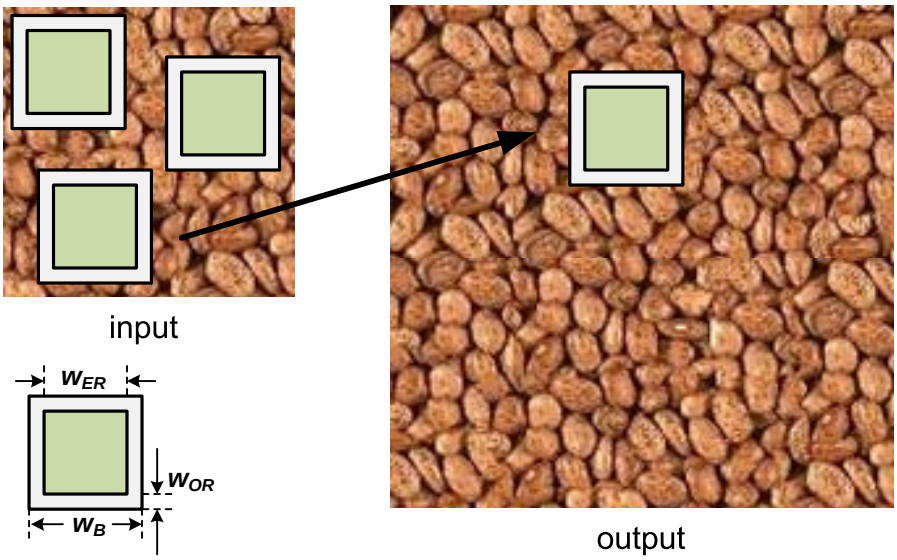


Figure 2.7: Three patches of the input image meet the distance tolerance condition. One of them is randomly selected to be copied to the output.

where δ_{max} is the predefined maximum distance between two boundary zones. Having this set of possible patches, one element of the set is chosen randomly and copied to I_{out} as the k-th patch.

Depending on δ_{max}, Ψ_B might also be an empty set. In that case the patch with the smallest distance $d(O_{B_k}, O^k_{out})$ is chosen as B_k from the input sample I_{in}.

Step-by-step the algorithm can be summarized as follows:

1. Copy a randomly selected patch from I_{in} to I_{out} and place it to the upper left corner
2. Calculate Ψ_B
3. If no patches have been found that meet the condition in Ψ_B, add B_{min}, the patch with the smallest distance $d(O_{B_k}, O^k_{out})$ to the set Ψ_B
4. Randomly select an element of Ψ_B and copy it to I_{out}
5. If I_{out} is not fully initialized yet, continue at step 2
6. Perform a blending operation for all overlap regions

The blending of the overlap region can also be performed immediately after copying the new patch to I_{out}.

In the algorithm above the order of arranging patches has been changed to better match the other patch-based algorithms covered in this work. Liang et al. start in the lower left corner and work their way up the image, which actually makes no difference to our modified version. The new order has just been introduced to be able to compare the different approaches with each other more easily.

Figure 2.7 shows an example of Ψ_B containing three patches.

2.2.2 Image Quilting for Texture Synthesis and Transfer

In "Image Quilting for Texture Synthesis and Transfer" [EF01] Efros and Freeman not only describe a technique for patch-based texture synthesis, but also for performing texture transfer as mentioned in Section 1.1.3.

The main idea of the algorithm is to synthesize a new texture by stitching together small patches of a given source texture in a consistent way. Efros and Freeman believe that during the synthesis process pixel-based approaches waste a lot of time for neighborhood search for pixels that are determined by what has been synthesized so far. They suggest to enlarge the unit of synthesis to whole patches and thereby transforming the challenge of synthesizing the new texture into a jigsaw puzzle like problem.

As in [EL99] the user has only one parameter to control the synthesis process. While it is the size of the neighborhood in [EL99] it is the size of the block in this publication. The balancing act is to make the block big enough to capture the relevant structures in the texture while keeping it small enough so that parts between these structures can be varied by the algorithm.

Efros and Freeman's way of stitching together several patches to form a larger output is a very naive, as the authors admit themselves.

Their algorithm calculates the transition path between patch A and patch B by using a method called "Minimum Error Boundary Cut" (MEBC) to reduce artifacts. The MEBC can easily be calculated using dynamic programming. As most texture synthesis approaches, [EF01] calculates the similarity between two patches using the L_2 norm.

To find the optimal cut through the overlap region, the following recursive formula is used:

$$E_{i,j} = e_{i,j} + \min(E_{i-1,j-1}, E_{i-1,j}, E_{i-1,j+1}) \qquad \text{with } i = 2, ..., N$$

where N is the user-defined size of the patch copied from the sample in each iteration of the algorithm and $e_{i,j}$ is the squared difference of pixel (i, j) in the overlap region of patch A and patch B. When the algorithm reaches the last row, the pixel with the minimum value is one end of the MEBC. Tracing back, the whole transition between A and B can be found by always selecting the minimum value. A similar procedure can be applied for horizontal cuts. If the patch overlaps vertically and horizontally, a combination of both approaches is used.

The algorithm

The algorithm is quite simple and rather similar to other patch-based synthesis sampling strategies.

1. In each iteration go through the output image in raster scan order in steps of one block, minus the size of the overlap region
2. Search the input image for a set of blocks, that differ in their overlap region only by a preset amount and finally choose one of those blocks randomly
3. Use the MEBC to find an optimum cut and make that cut the boundary of the new block
4. Copy the block to the output texture
5. If the output is not fully initialized, jump to step 1

Similar to pixel-based algorithms, where the user usually can control the size of the neighborhood, Efros and Freeman engage the user to set the size of the block. As with the size of the neighborhood in pixel-based approaches, the block size in patch-based algorithms is critical for the success of the algorithm, as the block must be big enough to capture the relevant structure in the given sample. Efros and Freeman state that using an overlap region that was $1/6$ of the block size worked best for them.

Texture transfer

A common way to extend user control over synthesis algorithms is to accept a target image as user input. To account for this input image, Efros and Freeman modify their cost function as follows: they introduce a weighting factor α to calculate the weighted sum of α times block overlap matching error (as before), but now they add the amount of $(1-\alpha)$ times squared error between the pixels of the source texture and those of the given user image mask. α thereby defines the trade-off between the regular texture synthesis process and the likeness to the user image mask. Because a cost function is only applied to the overlap region while initializing the output, several iterations need to be applied to yield good results. In each iteration Efros and Freeman reduce the size of a block and in addition now calculate the error between two patches not only for the overlap region, but also for the inner region. Efros and Freeman claim that at most 5 iterations are sufficient to get good results, where in each iteration the block size is reduced by $1/3$. They calculate the weighting factor α in each iteration i as:

$$\alpha_i = 0.8 * \frac{i-1}{N-1} + 0.1$$

where N is the number of total iterations.

2.3 Summary

In this chapter the differences between pixel- and patch-based approaches have been elaborated. The theoretic framework built in Chapter 1 has been emphasized by some basic, yet very popular and path-breaking algorithms for both kinds of texture synthesis strategies.

We have seen early approaches of:

Efros & Leung who describe a naive method that uses exhaustive search for each synthesized pixel

Wei & Levoy with a technique based on Efros & Leung that has been advanced by using a multi-resolution strategy and performing TSVQ for runtime acceleration

Ashikhmin with a clever modification of Wei & Levoy's publication that not only reduces search space, but also yields better results albeit just for natural textures

Liang et al. as a first patch-based texture synthesis approach, where new patch offsets are chosen to slightly overlap already existing patches to avoid artifacts by cross-fading

Efros & Freeman and their innovative *Minimum Error Boundary Cut* method to define the transition between two patches as well as their popular approach concerning texture transfer

Chapter 3

State-of-the-art Algorithms

The algorithms covered here build upon the concepts and publications presented in Chapter 2. The following sections examine an interesting patch-based approach by Kwatra et al. as well as an order-independent way to synthesize a texture pixel-based by utilizing parallelism of modern GPUs.

3.1 Graphcut textures

In this approach patch regions from a sample image are transformed and copied to the output, where they are stitched together along optimal seams that are found using a maximum-flow/minimum-cut algorithm. The patches that were stitched together form a new and (particularly) larger output.

Kwatra et al. [KSE+03] however first search for an appropriate location in the output image I_{out} to place the patch, then select a patch from the input image I_{in} and finally find the best transition between the patch taken from I_{in} and already initialized pixels in I_{out} by creating a graph that is cut into two parts. This operation is also responsible for the name of the algorithm: Graphcut. The calculated minimum-cut determines the parts copied from I_{in} over I_{out} and the parts kept in I_{out}. Using additional blending like feathering as in [LLX+01] is possible of course. See Section 3.1.5 for further information on this topic.

Again, we want the generated texture to be perceptually similar to the sample texture.

3.1.1 Definitions

There are some terms and definitions that need to be introduced before the algorithm can be examined in detail.

Error Region. The *error region* is the inner part of a patch. It meets two requirements. It is the region of the input patch that is *always* copied to the output, on the one hand, and is used on the other hand during the correction steps to define the next position in the output image that needs to be updated. The size of the error region has a huge impact of the synthesized texture. Not only it is responsible for maintaining patterns of the input sample, but also for finding regions that most urgently need to be revised after the output texture has been initialized. The size of the error region not only is important for the performance of the algorithm, but also for the quality of the result.

Overlap Region. The *overlap region* is the outer part of a patch. Unlike the error region, the overlap region is *not* always copied to the output texture. Instead it defines the area that is represented by the undirected graph used to find the optimal transition between two or more patches. The overlap region has a great impact on the quality of the synthesized texture. Having a large area for a transition between patch A and patch B, increases the chance of finding an inconspicuous seam. The size of the overlap region, however, is decisive for the performance of the algorithm. The larger the overlap region, the longer it takes to find the minimum cut.

Patch Offset. The *patch offset* declares where the new patch selected from the input image should be placed in the final output image. The offset highly depends on the chosen placement strategy. During initialization of the output image, the patch offset is chosen differently than during the refinement steps and generally serves a different purpose: it is chosen in a way that it contains both, initialized and uninitialized pixels of the output image. The reason for this is, that a new patch should do both, add new pixels to the output and be consistent with already existing, initialized pixels. After the whole output image has been initialized, the patch offset should be chosen in a way,

that a new patch covers the error region with the largest error, i.e. the region with the most expensive patch transitions.

Patch Seam. *Patch seams* define what parts of the input texture should be copied to the output texture. They denote the cut in the created undirected graph that causes the minimum costs. The algorithm used to calculate patch seams is the max-flow min-cut theorem, a statement in optimization theory about maximum flows in flow networks. It states that "the maximum amount of flow is equal to the capacity of a minimal cut".

One of the benefits of the Graphcut approach is, that it can "remember" where patch seams were defined and therefore correct suboptimal crossovers in later iterations. Furthermore Kwatra et al. suggest two methods to blur patch seams, which are described in Section 3.1.5.

3.1.2 How to find the perfect cut

The core of this approach is an algorithm, that computes the best seam between sample patches after finding the desired patch offset.

To find the optimum intersection between existing pixels of the output image A and the new Patch B the transition problem must be formulated into a common graph problem. In this (indirected) graph each node represents a pixel of the overlap-region, while edges hold the transition costs between those pixels. In other words: the higher the edge costs between pixel p_1 and pixel p_2 are, the less you want this edge to be cut by the graphcut algorithm. To be able to create our transition graph we need a cost function that calculates the edge costs.

A simple cost function could look like this:

$$M(s,t,A,B) = \|A(s) - B(s)\| + \|A(t) - B(t)\| \tag{3.1}$$

where A and B are the two patches and s and t are adjacent positions in A and B, respectively.

The success of the algorithm highly depends on the quality of this cost function, i.e. how good it represents the given transition costs. Equation 3.1 obviously is not a perfect cost function, as it only measures color differences (as, however, most algorithms do by simply using the L_2 distance norm). Nevertheless showed Ashikhmin in his publication about synthesizing "natural textures", that even a simple cost function can lead to impressive results as long as you know the domain you use an algorithm for.

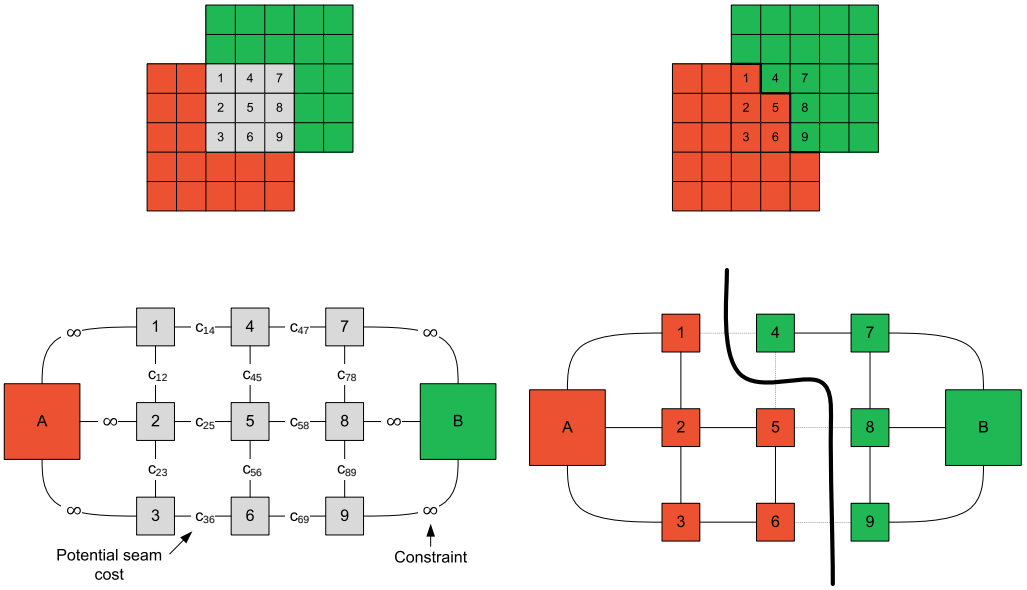
With our simple cost function in Equation 3.1 we can now build our indirected graph to find the transition between an existing patch $\boldsymbol{A}$ and a new patch $\boldsymbol{B}$. This is done by seeking the minimum cut of the graph that separates node $\boldsymbol{A}$ from node $\boldsymbol{B}$. The algorithm for finding this cut is a classical graph problem called min-cut or max-flow [Sed90].

The only constraint we have is, that after running the min-cut/max-flow algorithm, some pixels must be connected to patch $\boldsymbol{A}$ while other pixels must still be connected to patch $\boldsymbol{B}$. The cuts may only run between those constrained pixels. In Figure 3.1 we force pixels 1, 2, 3 to keep connected to patch $\boldsymbol{A}$ while we constrain that the pixels 7, 8, 9 have to be taken from the new patch $\boldsymbol{B}$. That means that in the output image the pixels represented by the nodes 1, 2, 3 will not be overwritten by pixels from patch $\boldsymbol{B}$, while, no matter what, the pixels represented by the nodes 7, 8, 9 will always be copied from patch $\boldsymbol{B}$ over the output image.

Accounting for old seams

One of the advantages of the graphcut approach over algorithms using dynamic programming, as for instance [EF01] does, is that the costs for old seams can be remembered and thereby corrected in one of the following iterations. To build a model that accounts for old seams however, tends to result in a less intuitive graph representation. Old cuts are now introduced as seam nodes between regular pixel nodes. The previously outlined algorithm therefore has to be extended as follows:

1. Introduce seam nodes where a seam has been drawn earlier
2. Connect each seam node s_i with an arc to the new patch node $\boldsymbol{B}$
3. Assign old transition costs along with the new costs for patch $\boldsymbol{B}$



a) Initial situation b) Result

Figure 3.1: After applying edge costs to the graph that is formed by the pixels overlapping each other the new cut can be found using the max-flow min-cut theorem.

There are three scenarios for this extended graph now:

Scenario A The edge between the old seam node an the new patch is cut. In this case the old seam remains in the output image

Scenario B The edge between the old seam node an the new patch is *not* cut: in this case the old seam needs to be overwritten by new pixels, thus the old pixel values are replaced and the old seam costs are discarded

Scenario C One of the edges between the old seam node and the pixel node has been cut: in this case a new seam is introduced, the new pixel values copied to the output and the costs for the new seam are added to the already existing costs of the old seam

Figure 3.2 illustrates the extended graph representation that is needed to account for old seams.

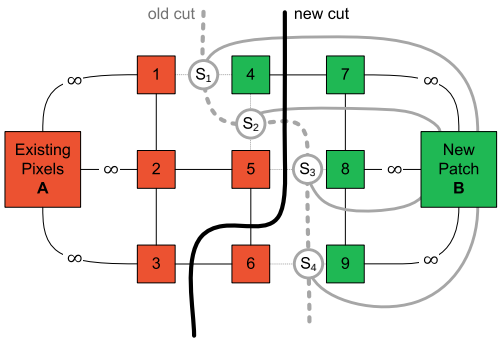


Figure 3.2: When the algorithm accounts for old seams, additional nodes need to be inserted into the graph. Those nodes represent the preceding cut. Including this information in subsequent passes might eliminate former suboptimal transitions.

3.1.3 Increasing the quality

Apart from placing new patches over existing pixels and placing new patches over old seams, there is a third configuration one must bear in mind, namely placing patches over a fully initialized output image. The main purpose to continue placing patches over initialized output images is to overwrite potentially visible seams with the new patch. The number of patches applied like this can either be automatically determined by a calculated error inside a defined error region or interactively by the user. In either way patches are placed as before when we defined how to deal with old seams. The only difference now is, that all pixels in the border region of the new patch are constrained to come from pixels from the output image. To guarantee that at least one pixel is taken from the new patch, an edge with infinite costs is inserted between an arbitrary pixel somewhere in the middle of the graph and the new patch B.

3.1.4 Placement strategies

Choosing the offset within the sample image and finding the best new cut in the created graph highly depends on the chosen placement strategy as well as whether or not the output is fully initialized, yet. There are three different placement strategies: *Random placement*, *Entire patch matching* and *Sub-patch matching*. We will now have a closer look at those placement strategies.

Random placement. In this approach the entire input image is taken as the input patch and is translated to a random offset location. The only constraint is, that during initialization the new patch must overlap initialized and uninitialized pixels (except for the very first patch, of course). After initializing is done, any offset is as good as the other. This means, that no pixel comparison needs to be done which makes *random placement* the fastest placement strategy by far. As for the other methods, the graphcut algorithm tries to find the optimal seam, which in this approach naturally only works satisfactory for non-regular textures.

Entire patch matching. As random placement, *entire patch matching* also uses the entire input image as the input patch. The offset within the output texture however is chosen by calculating the sum-of-squared differences of all pixels in the overlap region of the input image I_{in} overlapping exiting pixels of the output image I_{out}. The total costs of each translation is calculated as:

$$C(t) = \frac{1}{|A_t|} \sum_{p \in A_t} |I(p) - O(p+t)|^2 \tag{3.2}$$

where A_t is the number of pixels in the overlap region of I_{in} overlapping initialized pixels in I_{out} at translation t. Unlike random placement, entire patch matching also works good on regular textures.

Sub-patch matching. In contrast to random placement and entire patch matching sub-patch placement only uses a small sub-patch that is usually the same size or slightly larger than the error region. The offset in the input texture is chosen by a sum-of-squared differences.

$$C(t) = \sum_{p \in S_O} |I(p-t) - O(p)|^2 \tag{3.3}$$

where S_O is the output-sub-patch. All other variables have the same meaning as in Equation 3.2.

Sub-patch placement is the most general of all 3 techniques. At the same time it is the slowest.

3.1.5 Hiding seams

Graphcut does its best to hide artifacts caused by image transitions along the seams. Provided that the defined cost function and the used color space approximate human vision the calculated minimum cut runs along the optimal path. However, quite often none of the possible cross-fades is smooth enough, because overlap regions are too small or image content simply does not match at one or more borders of the selected patch.

Kwatra et al. suggest to use either multi-resolution splining or feathering in that case.

After implementing both techniques the result was sobering:

Multi-resolution splining would need much larger overlapping regions than usally used. As Kwatra et al. state themselves, the size of a typical overlap region is 8 pixels wide. Moreover multi-resolution splining should not be used excessivly during the synthesis process, since it tends to reduce the contrast of the image.

Feathering is a quite simple and rather cheap operation. After determining all adjacent patches near a seam, a gaussian filter is applied to the pixels close enough to the seam. Feathering works fine, but smudges the image visibly.

Both operations should therefore be handled with care and better be used selective.

3.1.6 Extended cost function.

As mentioned in Section 3.1.2 using a representative cost function to label the edges of the graph is essential for the final result, i.e. for the transitions of the different patches.

Based on the fact that possible discontinuities are less visible in high-frequency regions, a way to extend the trivial cost function defined in 3.1 could be to incorporate the gradient of the image.

The new cost function, introduced by Kwatra et al., takes this into account by scaling the previously calculated costs based on the vertical and horizontal gradients of both patches:

$$M'(s,t,A,B) = \frac{M(s,t,A,B)}{\left\|G_A^d(s)\right\| + \left\|G_A^d(t)\right\| + \left\|G_B^d(s)\right\| + \left\|G_B^d(t)\right\|} \tag{3.4}$$

where $G_A^d(x)$ indicates the gradient in direction d, vertical or horizontal, of patch A at pixel x. To calculate the gradient we used the *Sobel operator* in our implementation.

3.2 Parallel controllable texture synthesis

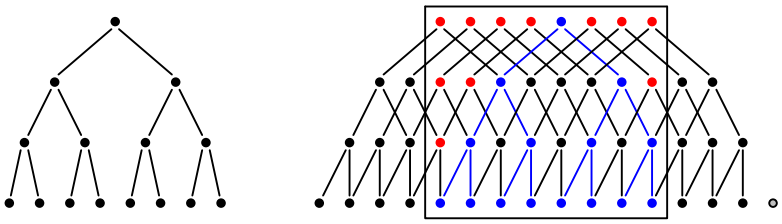
In 2005 Lefebvre and Hoppe published a paper called "Parallel controllable texture synthesis" [LH05]. It concentrates on two things: *efficient, parallel synthesis* and *intuitive user control*. The algorithm is based on [WL03], an extension of [WL00], and thus *order-independent*. One significant change in [LH05] is the use of texture coordinate-over color space.

Given an Exemplar E Lefebvre and Hoppe build a gaussian stack, which is used later to successively upsample the synthesized image. The process used to generate the output is similar to the method used in [WL03]. Starting with a small texture window of size 1×1, successive layers are calculated by applying three steps:

1. Upsampling
2. Jittering
3. Correction

Each of these steps can be performed to all pixels in parallel and thus utilize the parallelism of multi-core CPUs or modern GPUs.

In "Appearance-Space Texture Synthesis" [LH06], Lefevbre and Hoppe extend their own approach by encoding additional information into each pixel. They add *feature distance* and *radiance transfer* along neighborhood and pixel color information into high-dimensional vectors. These vectors are then reduced to lower dimensions using



a) Traditional image pyramid b) Gaussian stack

Figure 3.3: Traditional image pyramid vs. gaussian stack. The red dots in the stack mark invalid pixels. The blue path symbolizes the fact, that a gaussian stack is a set of nested gaussian pyramids. To create the highlighted gaussian stack of size $m \times m$ one needs an input image of size $2m \times 2m$.

principal components analysis (PCA). Section 5.5 is dedicated to how dimensionality reduction can be achieved.

3.2.1 Downsampling the exemplar

As a first step, the given exemplar image is downsampled using a gaussian stack, whereas the word *downsampling* is actually wrong. In fact, Lefevbre and Hoppe introduce a new data structure named *gaussian stack* that actually just blurs the image. The major feature of a gaussian stack is, that it has the same size in each level. Strictly speaking it is a the sum of all possible gaussian pyramids, where each pyramid is shifted by one pixel at the finest level, as can be seen in Figure 3.3.

The gaussian stack is comparable to the *quadtree pyramid* of [LLX+01], which is described in detail in Section 5.3. The main difference lies in the representation of the data. While gaussian stacks consist of one image per level, quadtree pyramids are stored as arrays of images.

To create a gaussian stack with a size of $m \times m$, an image twice as large in both, width and height, is required. For this reason we either have an input image that is $2m \times 2m$ pixels large or need to augment the exemplar at all sides, which can be achieved in one of two ways:

1. by tiling (if the input image is toroidal)

2. by mirroring (if the input image is *not* toroidal)

In case 2, when missing information is supplemented by mirroring the input image, we need to invalidate some of the potential candidates in the candidate set created for *k-coherence search*, which is used later during the image composition. Using this information in the correction step would result in synthesis artifacts.

Figure 3.3 shows the difference between a traditional image pyramid and the gaussian stack. The red dots in Figure 3.3b represent the invalidated pixels mentioned above. The blue path exemplifies the the concept of shifting gaussian pyramids into each other. The gray dot in the lower right symbolizes that this pixel is unused.

For further details on k-coherence search refer to Section 5.4

3.2.2 The algorithm

Synthesizing the final is done by successively upsampling, jittering and correcting the initial 1×1 output texture. The factor by which the upsampled texture is perturbed is a user-specified value and can vary from level to level.

Formally expressed, what the algorithm does is to synthesize an image S that stores the texture coordinates of an exemplar image E. Thus $S[p]$ holds the texture coordinates u of a pixel p in E. In mathematical terms this means $u = S[p]$. The actual color value can be retrieved by $E[u]$ or $E[S[p]]$ respectively.

Starting with an exemplar image E of size $m \times m$ a gaussian stack with L levels is created, where $L = \log_2 m$.

Upsampling

In this step the texture is enlarged and therefore doubled in both, width and height. Upsampling is a simple operation. Depending on the underlying concept, gaussian pyramid or gaussian stack, one of the following two formulas is applied to the output at level l:

$$S_l[2p + \Delta] := (2S_{l-1}[p] + h_l\Delta) \bmod m \tag{3.5}$$

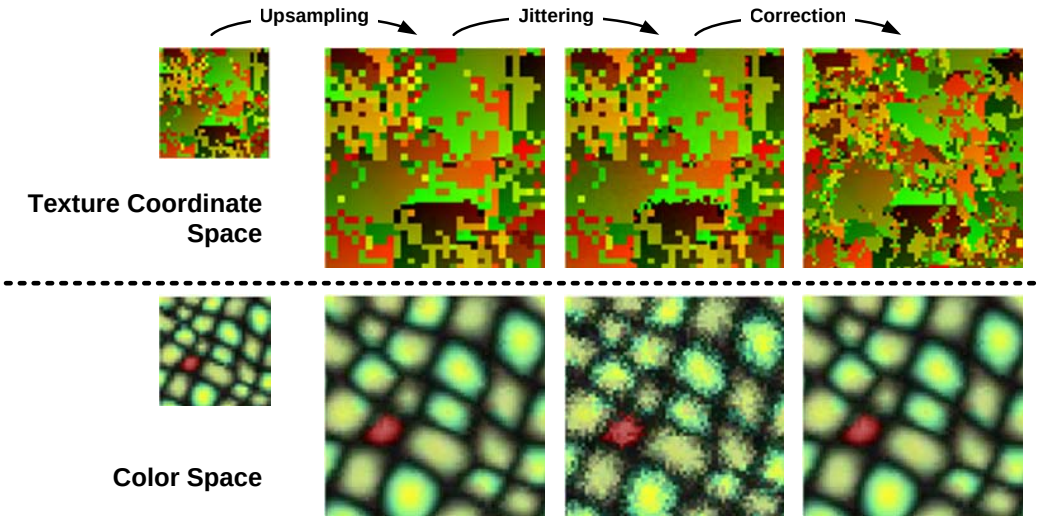


Figure 3.4: Steps performed during synthesis. In each iteration the image in the next coarser level is upsampled, jittered and finally corrected. This operation is done in texture coordinate space.

$$S_l[2p+\Delta] := (S_{l-1}[p] + \left\lfloor h_l(\Delta - \begin{pmatrix}.5\\.5\end{pmatrix})\right\rfloor) \bmod m \tag{3.6}$$

where h_l denotes the regular output spacing of exemplar coordinates and is defined as $h_l = l$ for Equation 3.5 and $h_l = 2^{L-l}$ for Equation 3.6.

Jittering

Jittering introduces spatially deterministic randomness to the upsampled texture. Given a jitter function and a user specified per level randomness parameter, jittering is defined as follows:

$$S_l[p] := (S_l[p] + J_l(p)) \bmod m \qquad \text{where } J_l(p) = \left\lfloor h_l H(p) r_l + \begin{pmatrix}.5\\.5\end{pmatrix}\right\rfloor \tag{3.7}$$

with $H : \mathbb{Z}^2 \to [-1,+1]^2$ defining a hash function and r_l representing the user-defined per-level randomness parameter, where $0 \leq r_l \leq 1$.

Correction

Correction is by far the most complicated of all three steps. For each pixel we gather a 5×5 neighborhood of the upsampled output texture and compare this with all neighborhoods from the exemplar. To accelerate this process we use *k-coherence search*.

The formula for correcting a pixel p in S is defined by Lefebvre and Hoppe as follows:

$$S_l[p] := C^l_{i_{min}}(S_l[p+\Delta_{min}] - h_l\Delta_{min}) \tag{3.8}$$

where $C^l_{1..k}(u)$ is the previously created candidate set of pixel u in level l with k candidates. The first candidate in C is the identity $C^l_1(u) = u$. i_{min} and Δ_{min} are defined as:

$$i_{min}, \Delta_{min} = \text{argmin} \left\| N_{S_l}(p) - N_{E_l}(C^l_i(S_l[p+\Delta] - h_l\Delta)) \right\| \varphi(i)$$

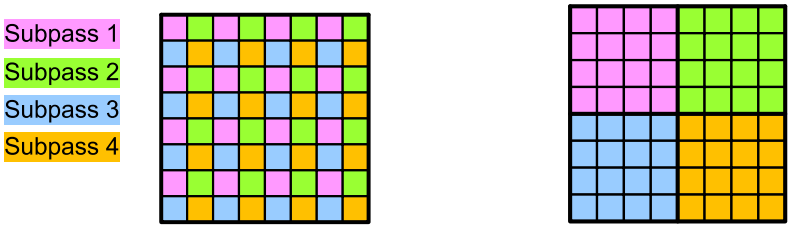
$$\text{with} \quad i \in 1..k,\ \Delta \in \left\{ \begin{pmatrix} -1 \\ -1 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \end{pmatrix}, ..., \begin{pmatrix} +1 \\ +1 \end{pmatrix} \right\} \quad \text{and} \quad \varphi(i) \begin{cases} 1 & i = 1 \\ 1+\kappa & i > 1 \end{cases}$$

where $N_{S_l}(p)$ is the neighborhood of pixel p in level l of image S and κ is a parameter that penalizes jumps for all candidate sets, except for C^l_1.

Basically this quite complicated formula finds the best matching neighborhood in the exemplar E for pixel p of the synthesized texture S, while favoring patch formations by using κ to penalize jumps.

An important issue of the correction pass is the order in which pixels are adjusted. Calculating pixel values based on neighborhoods changing simultaneously can lead to slow convergence or even cyclic behavior.

Lefebvre and Hoppe therefore suggest a clustering into a sequence of subpasses, each subpass correcting only non-adjacent pixels. This clustering is illustrated in Figure 3.5a. Although increasing the number of correction passes along with the number of subpasses per correction pass leads to better results, there is a limit for both. While too many subpasses are simply worthless as the quality does not further improve, doing



a) Fragmentation of the output into 4 subpasses b) Pixels resorted by subpasses

Figure 3.5: To avoid branches in the shader, the pixels of the four subpasses are resorted into four quadrants.

to many correction passes might result in an output texture, that looks less like the exemplar.

3.2.3 Implementation Details

Several optimizations recommended by Lefebvre and Hoppe have been adopted into our implementation. Most of them result from the architecture and limitations of modern GPUs.

Avoiding branches. As Lefebvre and Hoppe state in their publication, it turns out that a setup of two correction passes, each consisting of four subpasses, yields best results. In a naive approach one would correct the pixels of a subpass by using if-statements to reject the pixels not being updated in the current subpass of the pixel shader. As modern GPUs (still) do not support efficient branching, in such an implementation subpasses would take as long as a full pass, since all branches would have to be evaluated by the GPU.

In our implementation we therefore use an optimization suggested by Levebvre and Hoppe, that devides the subpasses into several sub-textures. This way the four subpasses together take not much longer than one full pass except for the calls to SetRenderTarget, which are insignificant compared to the naive approach however.

Dimensionality reduction of pixel neighborhoods. With a neighborhood of 5×5 pixels, each neighborhood would have a dimension of 75, i.e. 25 pixels, each consisting of R, G and B channel. With the introduction of a feature distance as Lefebvre and Hoppe outline in [LH06] the dimension would grow to 100. To speed up the process of neighborhood comparison, all neighborhoods of the sample texture are explored in each level and a matrix P_6 holding the 6 principal components is generated. Using P_6 each neighborhood can be projected as $\widetilde{N}_{E_l}(u) = P_6 N_{E_l}(u)$, where $N_{E_l}(u)$ is the calculated neighborhood at position u with a dimension of 75, respectively 100. This projection can be done as a preprocess.

During the correction step each pixel's neighborhood $N_{S_l}(p)$ then is projected using $\widetilde{N}_{S_l}(p) = P_6 N_{S_l}(p)$ likewise. Similarity once again is measured using the Euclidean distance, now in 6-d instead of 3-d as we did so far however.

Bandwidth optimization

While Levebvre and Hoppe suggest an abundance of data compression and therefore bandwidth optimization operations, our implementation only concentrates on the following two aspects:

Color caching: since operations in this algorithm are not carried out in color but texture coordinate space, each color fetch in the pixel shader would result in two texture lookups, namely one for getting the texture coordinate of the pixel in the exemplar $S_l[p]$ and one for actually looking up the color value $E_l[u]$. This can be avoided by storing a tuple for each pixel $S_l[p]$, that is $(u, E_l[u])$ instead of just u. Unfortunately this tuple would require 5 channels (2 for u, 3 for $E[u]$), so we make use of PCA once more and reduce the color information to its two principal components.

Channel quantization: since we always use k=2 for our k-coherence search, we only have $C_1^l(u)$ and $C_2^l(u)$ for each pixel in each level. Furthermore we calculated the projected neighborhood $\widetilde{N}_{E_l}(u)$ for each pixel. Overall we have 10 channels to store, which can be stored as an RGBA and two RGB textures. Beyond that, it is possible to spare one texture lookup by directly storing the neighborhoods of the candidate sets, similar to the technique used for the tuple we used for color

caching. This quadruple would look like this: $(C_1^l(u), \widetilde{N}_{E_l}(C_1^l(u)), C_2^l(u), \widetilde{N}_{E_l}(C_2^l(u)))$ and could be stored using four RGBA textures, two for each pair of candidate set and neighborhood.

3.3 Novel extensions

Apart from the optimization ideas mentioned in the publication of Kwatra et al. we experimented primarily with three ideas.

One idea was to avoid the widespread and very popular RGB color space, since RGB does not match the human perception of colors and luminance very well. Especially comparing two or more colors with each other is not conclusive in RGB.

Our second effort was to modify the algorithm to create tileable textures. Tileable textures have great benefits for a wide range of applications, not least they create better results when used as input to our implementation of [LH05].

Our third novelty contributes to eliminate the shortcoming of simply comparing color values to find optimal transitions between patches, as used in the default approach outlined in [KSE+03]. The method for determining seams between patches described by Kwatra et al. does not (and cannot) account for any kind of structure within the sample. This situation, however, leads to unsatisfactory behavior in many cases. Our attempt adds a simple way of prioritizing marked regions over other, untagged blocks.

3.3.1 Using L*a*b*

Our first approach was to verify the impact of the used color space, by converting RGB into grayscale and thus compare the image values mainly based upon their luminance. The results were, hardly surprising, unsatisfactory.

Thanks to the extended cost function introduced in Section 3.1.6, the shapes and objects of the textures, i.e. the shapes and objects crossing the seams, more or less fit together. Their colors however did not match at all.

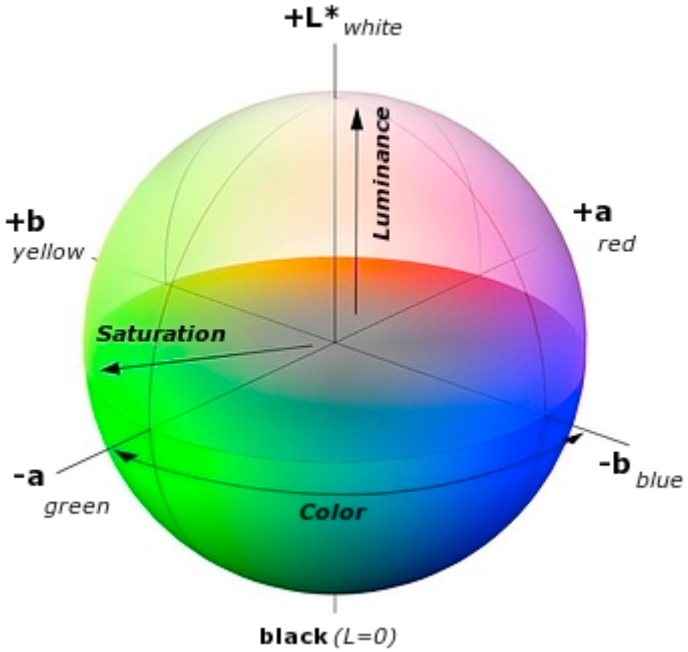


Figure 3.6: CIELAB color space

The common way to measure similarities is by using the L_2 norm, which certainly is not a good measurement of perceptual similarity in RGB. Our visual system recognizes edges and other high-frequency features very easily. This behavior is not reproduced by simply calculating the euclidean distance between two RGB values.

CIE L*a*b, also known as CIELAB or simply Lab, was specified in 1976 by the *Commission Internationale d'Eclairage (CIE)*. CIELAB, unlike RGB, attempts to approximate human perception. As a result perceptual differences between any two colors can easily be measured. For this purpose each color is treated as a point in 3-d space. Using the Euclidean distance between two or more points, as formulated in Equation 1.1 in Section 1.2.2, can be used as a degree of relative perceptual discrepancy.

CIELAB better maps the way we perceive things, but due to our imperfect understanding of the human visual system, it is not completely reliable either.

The positive impact of our modification was rather limited.

3.3.2 Tileable textures

The second modification we gave a try was to create tileable textures. Formally expressed, this means if $I_{out}(x,y)$ denotes the pixel at position (x,y) in the output image I_{out}, then $I_{out}(x,y) \equiv I_{out}(x \bmod M, y \bmod N)$, where M and N are the number of rows and columns, respectively.

While having textures that tile seamlessly is advantageous in many situations, extending the original approach to generate tileable textures is also problematic. To be precise, forcing the algorithm to synthesize textures that tile seamlessly might propagate and therefore increase errors with each iteration of the algorithm.

The reason for this behavior is intuitively obvious: imagine a near-regular sample texture as the basket image in Figure 3.7. The sample itself is not tileable. Highly depending on the placement strategy used and the parameters for the size of the overlapping and error region, a situation in which the left part of the patch might fit perfectly and the right part does not fit at all might occur. In this case the near-regular patterns of the basket in Figure 3.7 are displaced along (at least) one seam of the patch.

In the original version of the algorithm this is no big deal, since the error can be shifted stepwise to finally be no longer within the boundary of the output image.

In our modified version of the algorithm this self-correcting property does not exist. As an input patch is wrapped around both output image boundaries, the introduced error, i.e. the displacement of the near-regular texture patches, is too high to be corrected in following iterations. As a matter of fact in our test series the visible artifacts got even worse.

There are configurations where creating tileable textures work better, of course. It turned out that when using a large overlapping region and an irregular, near-stochastic or stochastic texture, the synthesized result is mostly satisfying. Increasing the overlapping region, however, extremely slows down the overall performance of the algorithm as discussed in Section 7.9.

As texture synthesis is far from being a fully automated process today and input properties along with parameter setup form the most crucial factors for each individual case, it can be said that extending Graphcut to produce tileable textures was a success.

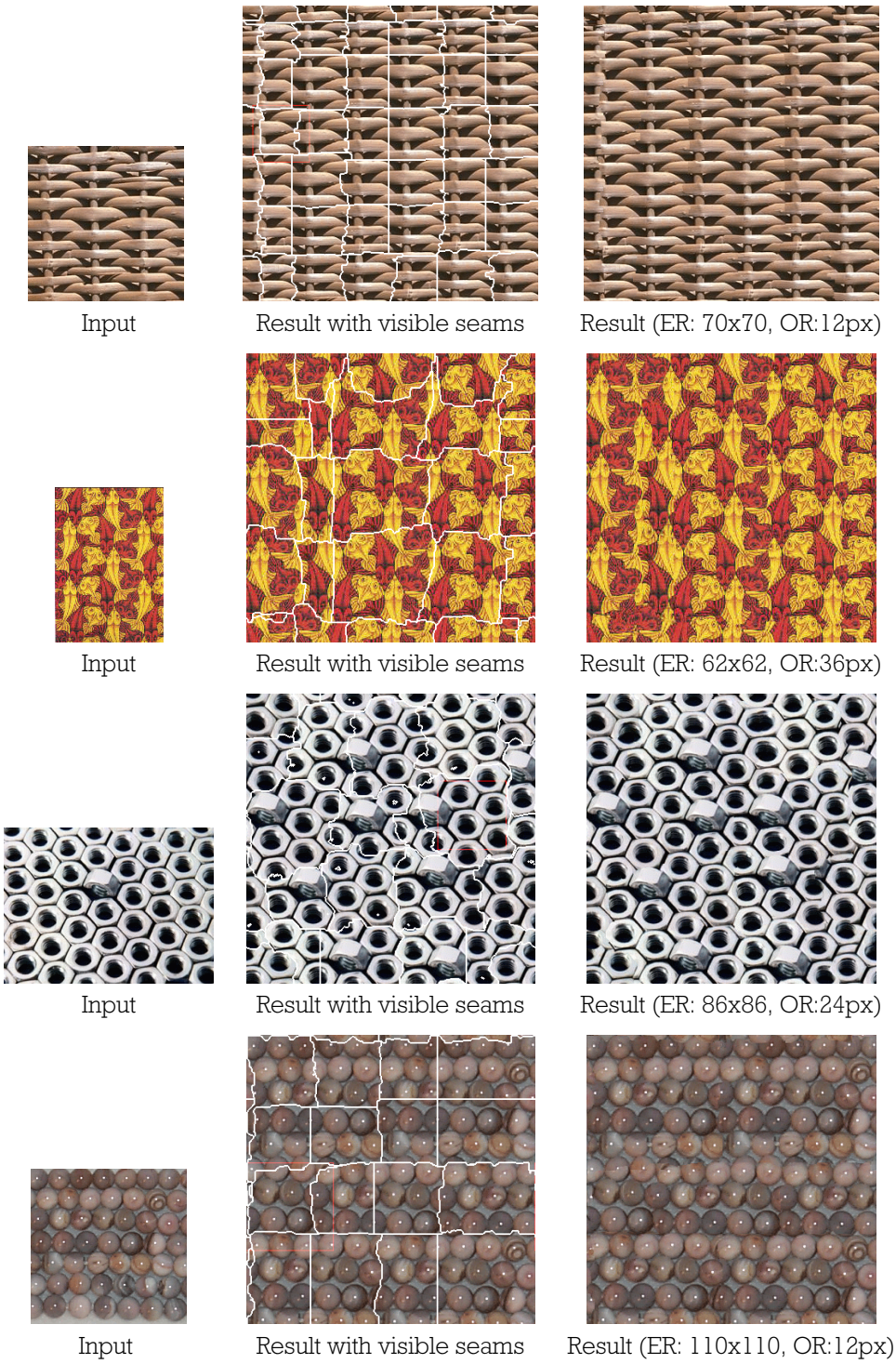


Figure 3.7: Results of our implementation with different error regions (ER) and overlap regions (OR). The images show the results of our extension to create tileable outputs for different types of sample textures. All input images can be found at the graphcut project's webpage at http://www.cc.gatech.edu/cpl/projects/graphcuttextures/.

Figure 3.7 shows some examples of synthesized textures with different types of input samples.

3.3.3 Weighting factors

One of Graphcut's major shortcomings is the missing information that goes beyond the color difference of currently compared pixel-pairs. The algorithm cannot deduce between more and less important regions or deduce where objects begin or end. In other words, it does neither have the ability to preserve the structure of simple objects or dominant regions nor does it provide a way for the user to prioritize certain pixels, objects or regions manually.

In addition to extending the cost function as described in Section 3.1.6, it turned out that there are situations where it is useful to weight selected areas in the input image.

To introduce weights to our implementation we provide an optional grayscale image as second input texture. The brighter a pixel in that image is, the more important it is to have neat transitions at the corresponding pixel of the input texture. In other words, color differences between patches are penalized more the brighter their corresponding regions in the grayscale image are.

In formal mathematical terms, this means that the cost function changes as follows:

$$M'(s,t,A,B) = \left[\frac{M(s,t,A,B)}{\left\|G_A^d(s)\right\| + \left\|G_A^d(t)\right\| + \left\|G_B^d(s)\right\| + \left\|G_B^d(t)\right\|} \right] * W_A(s) * W_B(t)$$

where $W_A(s)$ and $W_B(t)$ identify the weighting factors in patch A at position s and in Patch B at position t respectively.

The idea behind this extension is, that by penalizing bad transitions at some positions, the user has the ability to brand regions that could lead to bad results in a very straightforward and intuitive way.

The effect of introducing weighting factors is first and foremost to reduce the negative impact of dominant regions from the input. By applying weights to regions where erro-

neous transitions would be particularly apparent, we instruct the algorithm to do one of three things:

1. Find a nearly perfect seam for the prioritized region
2. Copy the whole region as it is to the output
3. Avoid copying the region (or parts of it) at all

Each of the three options above is acceptable. The major difference to the input modification described in [LH05] is, that the algorithm can decide which option best fits the current situation, while Lefebvre and Hoppe's extension forces their algorithm to keep the tagged blocks as they are to be kept in the output by constraining the jitter amplitude for that regions. This constraint however is predestined to lead to visible repetitive patterns. Depending on the size of the labeled areas, the modification might even produce outputs that are similar to simply tiled versions of the input as will be seen later in Section 7.

Furthermore Lefebvre and Hoppe's extension does not guarantee the preservation of marked regions as their approach only affects the amount of modulation caused by jittering the image after upsampling took place. During correction, tagged areas might be scrambled nonetheless. Figure 7.21 gives an example of this situation.

In Chapter 7 the effect of introducing weighting factors to Graphcut as well as modulating the input of PCTS is evaluated by synthesizing outputs with and without those extensions.

3.4 Summary

In this chapter we have analyzed two state-of-the-art texture synthesis methods:

Method 1 was a patch-based method by Kwatra et al. that handles patch transitions similar to the *MEBC* strategy used by Efros & Freeman in Chapter 2. The main difference between the two strategies, however, is that the algorithm based on Graphcuts "remembers" old seams and therefore has the ability to correct former

suboptimal seams that might have caused visual artifacts. Moreover three placement strategies have been introduced: *Random placement*, *Entire patch placement* and *Sub-patch placement*.

Method 2 a pixel-based method by Lefebvre and Hoppe that is order-independent and therefore enables outsourcing expensive calculations to the GPU. To achieve texture variations several steps of the algorithm are performed in texture coordinate- rather than color space. The functioning is encapsulated into three parts: *Upsampling*, *Jittering* and *Correction*. A new data structure, the gaussian stack, has been introduced for exemplar matching. We have seen several methods for packing information into vectors and textures to facilitate data transfer between the pixel shader and the CPU.

Moreover the impacts of our novel ideas have been outlined. Using CIELAB, producing tileable textures and introducing weighting factors to [KSE+03] improved the original attempt of Kwatra et al. to different extents. None of our novelties can replace the burden of analyzing the input texture before running the algorithm. Nonetheless it gives the user more control over the synthesis process and thereby leads to better results.

More details about special aspects of our implementation of both approaches examined above will be given in Chapter 6.

An elaborate evaluation of both algorithms is given in Chapter 7.

Chapter 4

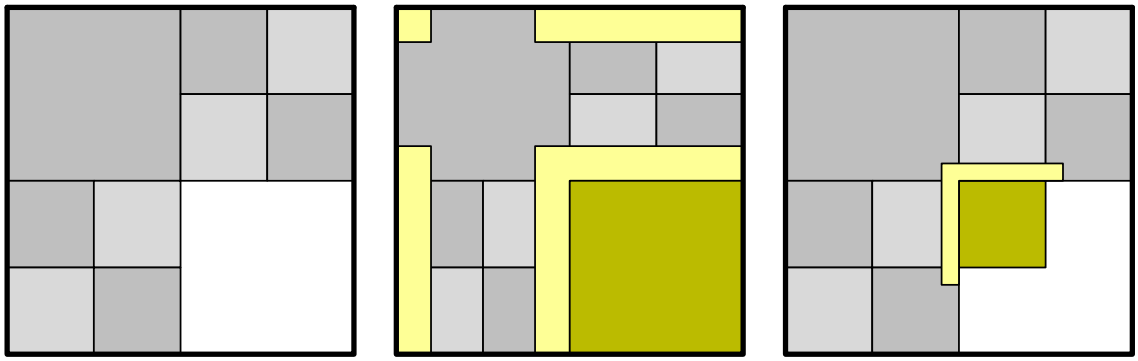
Alternative Sampling Methods

So far the most popular ways to synthesize textures is by either using pixel- *or* patch-based approaches. However while patch-based texture synthesis algorithms are good in preserving global structure, they often produce visible seams along patch boundaries. On the other hand pixel-based algorithms often tend to lose small texture patterns and therefore although producing a consistent impression do not resemble the input texture.

To improve the quality of the resulting texture, there are approaches that are based on pixel- or patch-based algorithms, but extend those ideas. Although pixel- and patch based algorithms are most popular at the moment, reasoning that textures synthesis can only be done in this manner is wrong. To emphasize that, two different approaches shall be presented subsequent in a nutshell.

4.1 Hybrid Texture Synthesis

Andrew Nealen and Marc Alexa from the University of Technology in Darmstadt, Germany, try to combine the benefits of pixel- and patch-based algorithms into one technique called "Hybrid Texture Synthesis" [NA03]. They try to exploit the speed and structure preserving properties of patch-based algorithms and switch to the blurring kind of pixel-based approaches when errors in the overlapping regions become too obvious.



a) Synthesize white region b) $\Delta_p > \Delta_{max} \rightarrow$ split patch a) $\Delta_p <= \Delta_{max}$

Figure 4.1: The figure shows the patch-based synthesis part of the algorithm where the new patch does not meet the condition $\Delta_p < \Delta_{max}$ and therefore needs to be split

The algorithm starts by placing patches as all patch-based algorithms do. Instead of blending, feathering, etc., however, it computes the error for each pixel in the overlap region and uses a pixel-based synthesis strategy to synthesize erroneous pixels.

4.1.1 The algorithm

The algorithm works as follows: in each step a patch fitting the target region best is selected from the sample texture and placed to the output image with an overlapping region of ov pixels in each direction. If the overlap error Δ_p exceeds the maximum overlap error Δ_{max}, the search process is repeated using smaller patches. This process, which the authors call *splitting*, is repeated until the overlap error in the overlap region is below the maximum overlap error Δ_{max} or the size of the patch reaches a lower limit.

A second constraint δ_{max} is used to measure each pixel error within the overlap region. If the pixel error is above the threshold δ_{max}, it is marked as invalid and the pixel is re-synthesized afterward.

The exact formula for calculating the pixel error is of no further interest for this work as this section is just meant to give a gist of alternative texture synthesis methods.

Figure 4.1 shows the process of placing and thereby splitting a new patch to the output.

4.2 Wang Tiles for Image and Texture Generation

Wang Tiles are not limited to be used in texture synthesis. They may also contain patterns or geometry to create new tiles in a non-periodic way. This creation process can be done very efficient once the tiles are filled with data.

Wang Tiles are squares where each edge is encoded by a color. Only when all edges of a tile match with the colors of the edges the tile is connected to, the tiling is said to be valid. Before Wang Tiles can be used for texture synthesis, the tiles need to be filled with 2D-texture information. This can be achieved in two ways, manually or automatically.

Cohen et al. show that using 8 tiles is sufficient to non-periodically tile the plane using Wang Tiles. However just encoding the North, South, East and West edge, is not satisfactory when features cross more than one edge. To maintain coherent patterns, additional informations need to be encoded into the tiles. This is known as the *Corner Problem* and will be covered in a later section.

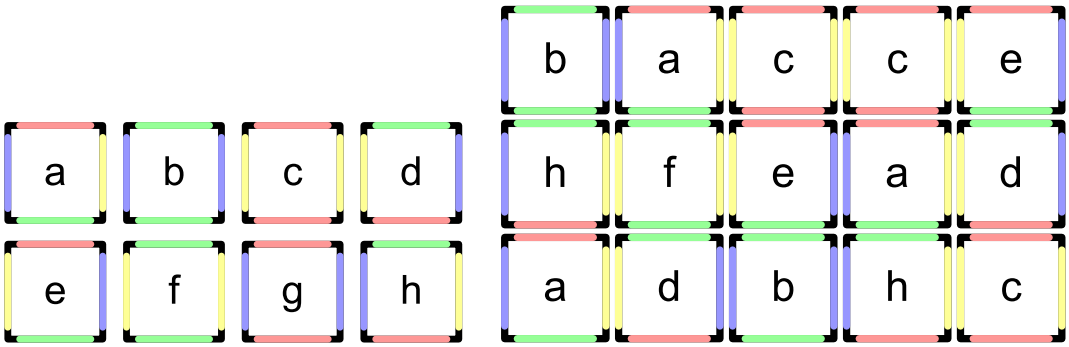
The idea of using Wang Tiles for texture synthesis first came up in [Sta97]. Wang Tiling itself was first proposed by Hao Wang, a chinese mathematician, in 1961 and therefore is named after him.

The elements in a Wang tile-set can not be rotated. Wang tiles are always squares.

4.2.1 The algorithm

The stochastic tiling algorithm designed by Cohen et al. works as follows:

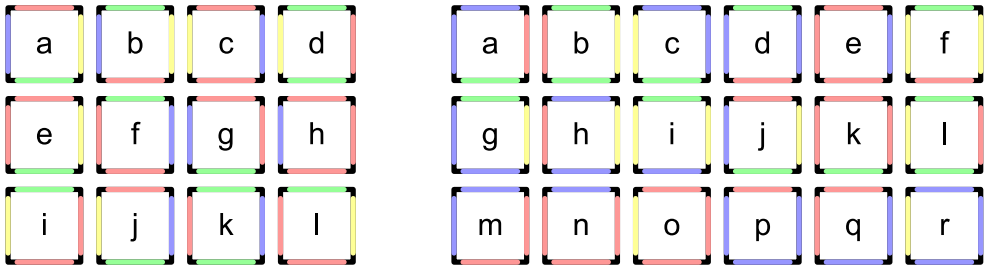
1. Select any tile for the upper-left corner
2. Fill the first row by always selecting tiles thats left edge matches the color of the right edge of the previously placed tile



a) 8 Wang Tiles can stochastically tile the plane

b) Sample tiling constructed using the set 8 Wang Tiles

Figure 4.2: Example of a set consisting of 8 Wang Tiles



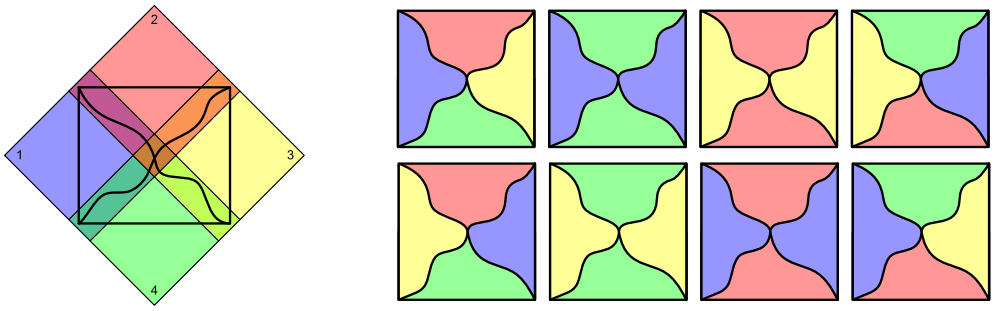
a) 12 Wang Tiles constructed from two horizontal and three vertical colors

b) 18 Wang Tiles constructed from three horizontal and three vertical colors

Figure 4.3: Extended Wang Tiles sets

3. For the first tile in a new row select a new tile where the upper edge matches the lower edge of the tile placed above
4. Fill the row with tiles where the upper edge matches the lower edge of the tile above and the left edge matches the right edge of the tile to the left
5. Go to step 3 to generate as many rows as desired

As long as there is at least one tile for each upper/left edge combination, the procedure above creates a valid tiling. Since our set of Wang Tiles contains all combinations of upper/left and lower/right edges this is always true. The algorithm described above assumes that the tiles are placed from top to bottom and left to right.



a) Four subimages are combined to form each Wang Tile

b) Construction of an eight tile set

Figure 4.4: Creating the tiles

If we use K-colors to encode the edges, we get K^2 color combinations for two adjacent edges of a tile. If our set contains at least two tiles for each such combination at each step we have at least two choices. If this choice is made with uniform probability each step reduces to an independent random process (i.e. a coin flip) and therefor the plane will always be tiled non-periodically.

Using a larger set of tiles has two advantages:

1. it reduces repetition artifacts
2. larger patterns can be introduces to the tiling of the plane

4.2.2 Tile filling

Cohen et al. introduce two methods for filling the tiles within the Wang Tile set, whereas the first method, *Interactive Tile Design*, is limited to simple geometry and thus of no further interest for this work.

The more advance technique to fill the tiles is done automatically. It is based on [EF01]. New tiles are created by putting together four patches. This is done by slightly overlapping the samples and finding a cutting path for that overlapping region. Thereafter the now combined four samples are rotated by 45° so that they form a diamond shape now. The new tile is generated by cutting along the diagonals of the new diamond

shape. This forms the final tile. The thereby created tile only uses one half of each sample. This means that the other half is used when two tiles are connected along one edge having the same color. Therefor the tiles will always fit together. Figure 4.4 illustrates this process to create on a set of 8 tiles.

Visible artifacts may occur for two reasons:

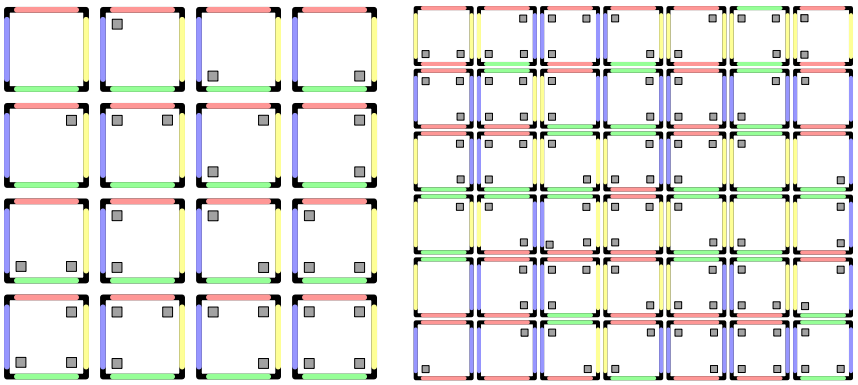
Reason 1: too view sample patches are used: this problem can be overcome by increasing the number of tiles

Reason 2: because the errors created by Efros and Freeman's algorithm (or any other path-finding algorithm) become even more visible in this approach, since they are repeated for each tile created. This means that the quality of Wang Tiles can only be as good as the path-finding algorithm used. To minimize the quiliting errors, Cohen et al. iterate over large sets of samples and sum all errors that occur along the cutting path, i.e. for a set of eight tiles 32 paths. If after this process the commulated error is too high, four new samples are selected, stitched together as before and the new overall error is computed. This process is repeated until the overall error is small enough or a pre-set number of iterations have been completed. The creation of the tiles is a time-consuming process, but has to be done only once before the algorithm starts and therefor can be precomputed. After the tiles are generated, creating large textures is very fast.

4.2.3 The Corner Problem

The problem with the so far presented algorithm is, that is does not account for object that span across the corner of a tile. To cover this case as well, the corners of the tiles need to store additional information. This leads to a combination of 2^4 - corner encoded yes/no for each corner - encodings for each tile. In a set of 8 tiles this means an enlargement to $8 \times 16 = 128$ tiles.

The extended information that is now encoded into the new tiles along with a sample arrangement is shown in Figure 4.5.



a) Extended Wang Tiles set

b) Sample arrangement of the extended Wang Tiles

Figure 4.5: Extended Wang Tile sample set with additional information to account for the corner problem

4.2.4 Further improvement

By using two source images instead of one the synthesis process described above can be further enhanced. The idea behind this is to exploit the extension devised to solve the corner problem. Imagine having two images of a hayfield with a different density of flowers. The previously introduced encoding of the corners can be used to represent regions of different density. The two images used in this process must fit together via the quilting algorithm of course.

4.3 Summary

In this chapter two rather unconventional synthesis methods have been summarized.

First, *Hybrid texture synthesis* by Nealen and Alexa, who try to improve existing texture synthesis algorithms by combining their strength has been summarized in a nutshell. The idea behind this approach is to first apply patch-based texture synthesis by stepwise splitting the input patch until the condition $\Delta_p < \Delta_{max}$ is met. This procedure is followed by a pixel-based correction scheme that corrects single erroneous pixels.

The second alternative approach concerned *Wang Tiles* and has been covered in more detail, since it is a totally different synthesis strategy than all other methods described in this work. Using Wang Tiles to synthesize textures is an interesting idea. Although similar to tiling, having not one, but many tiles, the algorithm creates much better results in a high-performance manner. The weakness, which at the same time is a strength of the introduced method, is the creation process for the tiles. The problem is that the same visible artifacts are repeated over and over again across the output image in a repetitive way. However, creating the tiles is a pre-process and therefore neither time critical nor bound to a special way of finding optimal transitions between the patches.

Chapter 5

Optimization Methods

When performing texture synthesis a lot of time consuming operations may occur. To not waste too much time searching for perfect neighborhoods, overlap regions, etc. a number of optimization algorithms exist. Some of those algorithms and data structures along with quality enhancing methods will be examined in this chapter.

5.1 Tree-structured vector quantization

Tree-structured vector quantization (TSVQ) is a procedure that is usually used as a lossy data compression method. In 1980 Linde, Buzo and Gray (LBG) introduced an algorithm for Vector Quantization (VQ) that was based on a training sequence. Vector Quantizations that use this algorithm, like Wei and Levoy's approach does, are therefore also referred to as LBG-VQ.

In simple terms: vector quantization is a compression method, that clusters large sets of vectors into groups of vectors that are close to each other and approximately have the same number of elements per group. Each group is represented by its centroid point, also called *codevector*. The area assigned to codevectors is called *encoding regions*. The union of all codevectors is called *codebook*, the set of all encoding regions *partition* of the space.

Vector quantization consists of two steps. Step one is responsible for creating the codebook by training the codevectors with a set of sample vectors. Step two assigns the closest codevector to each new vectors.

In formal mathematical terms, this means:

Let T be a training sequence of M vectors:

$$T = \{v_1, v_2, ..., v_M\}$$

where each of the vectors has a dimension of k

$$v_i = (v_{i,1}, v_{i,2}, ..., v_{i,k}) \qquad \text{where } i = 1, 2, ..., M$$

Let N be the number of codevectors with

$$C = \{c_1, c_2, ..., c_N\}$$

where each k-dimensional codevector is defined as

$$c_j = (c_{j,1}, c_{j,2}, ..., c_{j,k}) \qquad \text{with } j = 1, 2, ..., N$$

Let S_j denote the encoding region for the codevector c_j with

$$P = \{S_1, S_2, ..., S_N\}$$

as partition of the space.

The approximation Q of a given vector x_i in the encoding region S_a is then given by

$$Q(x_i) = c_a \qquad \text{if } x_i \in S_a$$

Using the L_2 norm the average error can be defined as:

$$D_{avg} = \frac{1}{Mk} \sum_{m=1}^{M} (x_m - Q(x_m))^2 \tag{5.1}$$

The goal is to find a solution for this minimization problem.

The LBG-VQ algorithm

LBG-VQ uses an initial codebook to iteratively solve the optimization problem defined by the following two conditions:

1. Any encoding region S_n contains all vectors that are closer to c_n than to any other codevector
2. Any codevector c_n is the centroid of all training vectors in S_n

The algorithm works as follows:

1. Choose the size of the codebook by setting the number of desired codevectors N
2. Define an initial codebook by randomly selecting N codevectors
3. Cluster each input vector by determining its Euclidean distance to each codeword
4. Compute the set of codevectors by obtaining the centroid of the values in each cluster
5. Repeat step 3 and 4 until the change in codevectors is below a predefined threshold

The TSVQ algorithm

The algorithm used by Wei and Levoy for their *tree-structured vector quantization* works similar to the LBG-VQ algorithm described above.

It starts by computing a centroid from the set of given training vectors and uses that as its root level codevector. As a second step this codevector is split and the initial codevector along with a perturbed centroid are chosen as initial children of the root. As before in LBG-VQ the two children are then assigned training vectors to and corrected

to find the locally optimal codevectors. The training vectors are thereby divided into two groups based on theses codewords and the algorithm recurses on each of the subtrees. This procedure is repeated until either a number of predefined codevectors have been reached or the average error, as defined in Formula 5.1, is below a certain threshold.

Using the tree

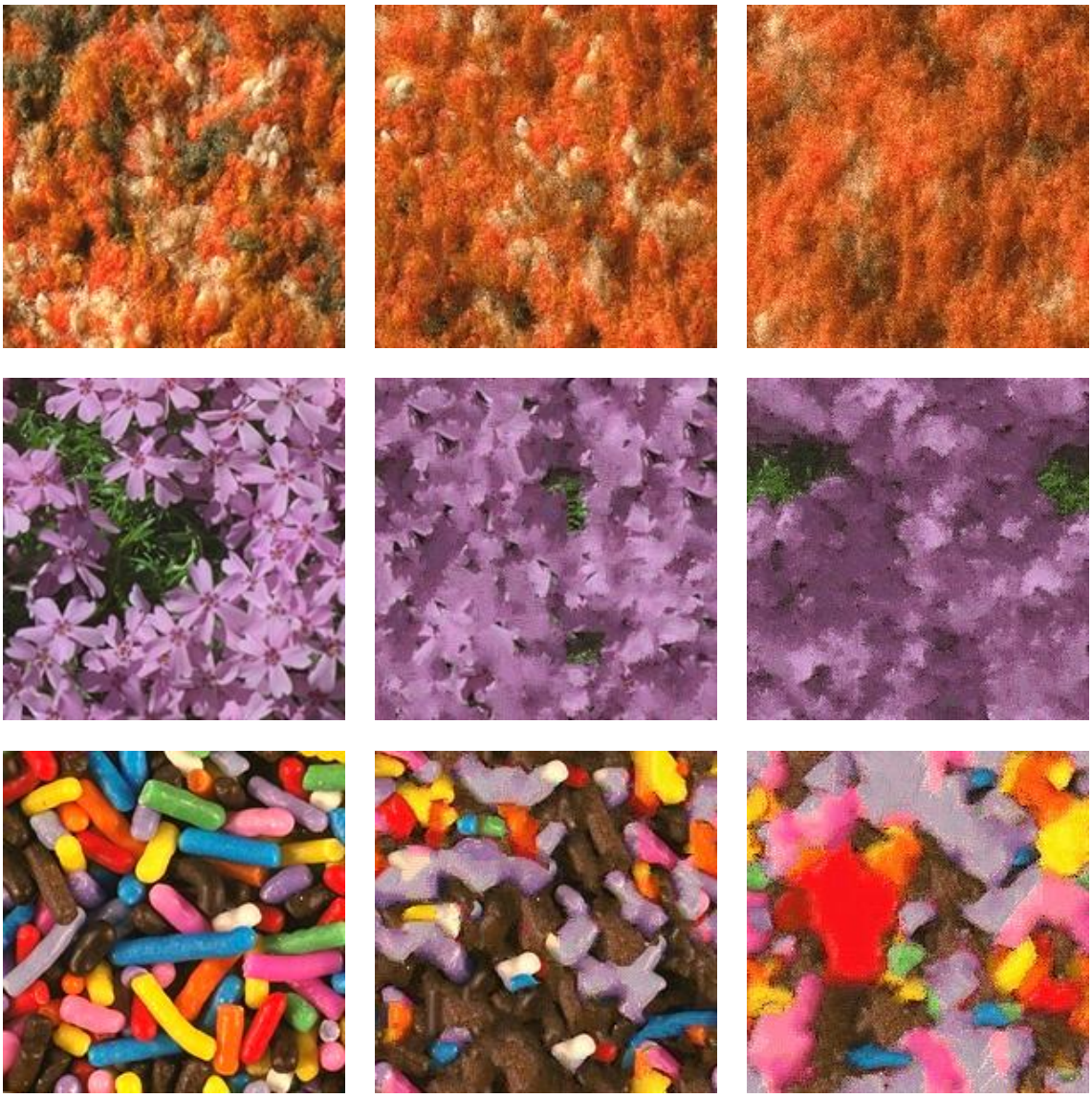
To find the nearest codevector for a given query vector, and thereby the new pixel to be synthesized, the tree is traversed in best-first ordering. Thus in each level of the tree the child node with the codevector closer to the input vector is followed until a leaf node is reached. Although this node in the majority of cases does not hold the best match available for the given vector/neighborhood, the runtime behavior is much better than with the exhaustive search. As training data Wei and Levoy use the neighborhoods of each pixel of the input sample.

A pitfall of TSVQ acceleration is that it tends to blur the output even more as can be seen in Figure 5.1.

5.2 Optimized k-d tree

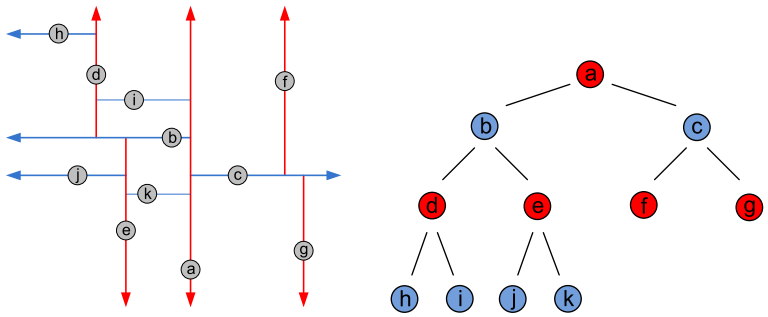
Nearest neighbor search is an optimization problem for finding closest points in metric spaces. Let S be a set of points in M, a d-dimensional metric space, and $q \in M$. We then want to find $p \in S$ that is closest to q.

A k-d tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space. k-d trees are a special case of BSP-trees and can be used for nearest neighbor searches.



a) Original b) Full Searching Synthesis c) TSVQ Synthesis

Figure 5.1: The images above show the results of images created with and without using TSVQ. All images of this figure can be found at the project's web page of Li-Yi Wei and Marc Levoy at http://www.graphics.stanford.edu/projects/texture/



a) Points in 2D space b) k-d tree

Figure 5.2: Sample k-d tree dividing space into two parts at each node. Each new plane is perpendicular to the preceding one.

BSP trees

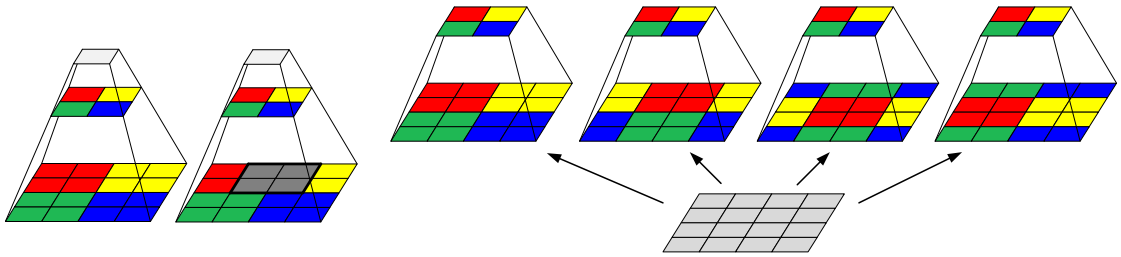
Binary Space Partitioning (BSP) trees are data structures that recursively divide the plane into two parts. BSP trees can either be axis- or polygon-aligned, where only axis-aligned BSP trees will be covered here.

While 3-D rendering for instance uses BSP trees to sort objects from front to back (e.g. for occlusion culling or transparency) the k-d tree used in texture synthesis is utilized to perform an approximate nearest neighbor search.

The property that makes a BSP tree a k-d tree is the used splitting strategy. The most general approach is by starting with a root node that divides the whole space into two parts. Each node at any recursion level generates a perpendicular plane to the one used last and again divides space into two boxes. This splitting operation usually is repeated until the number of data points associated with the node is smaller than a preset quantity, called the *bucket size*.

Splitting strategy

A common strategy for splitting a scene into two halves is by cycling through the axis, i.e. starting by dividing space along the x-axis, its children might be split along the y-, the grandchildren along the z-axis, and so on. After k-splits the cycle repeats. Points



a) Problem of regular image pyramids

b) Quadtree image pyramid

Figure 5.3: Each level of a quadtree pyramid has four children. Thereby each patch of one level has a corresponding patch in the next lower level.

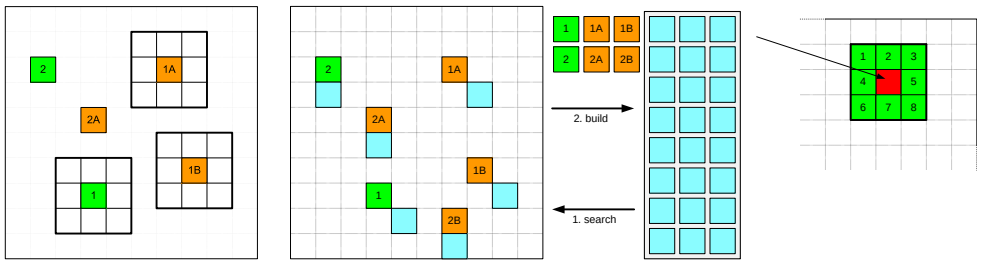
lying on a splitting plane may be associated with either child, according to a predefined splitting rule.

As a splitting rule Liang et al. use the *sliding midpoint rule* for their implementation. The sliding midpoint rule is a simple modification of the midpoint splitting rule. The unmodified midpoint splitting rule simply cuts the current cell through its midpoint orthogonal to its longest side. This simple rule runs the risk of producing so called *trivial splits*, where all points lie to one side of the splitting plane. As a result the depth of the resulting tree can be arbitrary large. To avoid this degeneration the splitting algorithm only accepts the midpoint as splitting plane if points of the data set lie on both sides of the splitting plane.

If this condition is not fulfilled, the splitting plane is moved toward the points, until the first data point is reached. By "sliding" the splitting plane like this, the partitioning of the space can never result in any trivial splits. Thus it can be guaranteed that the tree has a maximum depth of N levels.

5.3 Quadtree Pyramid

Quadtree pyramids (QTP) are data structures similar to gaussian stacks. The idea behind QTPs is to have an extended gaussian pyramid where each patch of one level has a corresponding patch in a lower resolution. In standard gaussian pyramids this is not the case as can be seen in Figure 5.3.



a) Analysis b) Synthesis

Figure 5.4: k-coherence search is divided into two phases. During analysis k similarity-sets are built for each pixel. Those similarity-sets are then used to synthesize the new pixel as shown in b).

The "*quad*" in *quadtree pyramid* just says, that each node has four children, and therefore holds a corresponding pixel for each set of four pixels in its next, lower level. In other words: in each level of the pyramid the four lower resolution images are calculated by shifting by 1-pixel along the x- and y-directions and using a box filter of size 2×2 to downsample the image data.

The resulting image pyramid can be used to get a first approximation of the nearest neighbors in the lowest resolution level. The reduced set of candidates can then be improved or further reduced in each higher level of the tree. The speed-up achieved by this acceleration method naturally becomes more efficient with smaller numbers of initial candidates, i.e. when the number of candidates $m << n$, where n is the number of datapoints.

5.4 k-coherence sets

K-coherence sets establish a basis for k-coherence search, an acceleration algorithm in pixel-based texture synthesis. The basic idea of k-coherence search is to reduce the number of pixels covered in search of similar neighborhoods. The k-coherence algorithm [TZL+02] is divided into two phases:

Analysis. This stage of the algorithm can be pre-computed. When analyzing a given texture a similarity-set is built for each pixel. This similarity-set contains a list

of other pixels, respectively their offsets, that have similar neighborhoods as the current pixel. The size of the set can be chosen arbitrarily, but since k-coherence search is meant as an acceleration, using similarity-sets with a large number of elements k would degrade the effectiveness of the optimization. The lower the number of elements in a similarity-set, the more different the resulting texture looks compared to the full search. Based on experience a good trade-off between quality and performance is reached with a k in the range of 1 to 11.

Synthesis. For each new pixel in the output image a set of candidate-pixels is generated. This is done by adding the similarity sets of all texels in the neighborhood of the pixel thats new value should be determined. In this list of similarity-sets we then search to find the best matching neighborhood in the input image. As can be seen in Figure 5.4 the texels of the similarity-sets need to be shifted to match the position of the particular pixel in the current neighborhood offset of the output image.

5.5 Principal Components Analysis

Principal Components Analysis (PCA) is a frequently used operation to reduce multi-dimensional data sets to their most significant or principal components. Although this dimensionality reduction is not a lossless data compression, it can be shown, that a high degree of information content is obtained even with a highly narrowed input. PCA is often used to accelerate algorithms for pattern recognition. To fully understand what the PCA does and how it can be calculated, some terms and definitions need to be introduced.

Standard deviation and variance

The standard deviation of a data set is a measure of dispersion of the values of the given set. Casually spoken, it is the average distance from the mean to a value of the data set.

The standard deviation is defined as:

$$s = \sqrt{\frac{\sum_{i=1}^{n} (X_i - \overline{X})^2}{(n-1)}} \tag{5.2}$$

The variance of a data set is simply the standard deviation squared.

Covariance

Standard deviation and variance only specify how data varies in one dimension. To determine if and how two or more dimensions are related to each other, a new measurement, called *covariance*, is used. The formula for the covariance is very similar to the formula of variance:

$$cov(X,Y) = \frac{\sum_{i=1}^{n} (X_i - \overline{X})(Y_i - \overline{Y})}{(n-1)} \tag{5.3}$$

As can be seen easily, the only difference to the formula of variance is the second set of brackets, where the Xs have been replaced by Ys. The result of this replacement is, that we now measure how much the dimensions vary from the mean, with respect to each other. Trivially the covariance between one dimension and itself equals the variance of this dimension. Furthermore is $cov(X,Y)$ equal to $cov(Y,X)$, since multiplication is commutative. The result of covariance is interpreted as follows:

- $cov(X,Y) > 0$: both dimensions increase together
- $cov(X,Y) < 0$: one dimension increases, as the other decreases
- $cov(X,Y) = 0$: the given dimensions are independent of each other

Eigenvectors and eigenvalues

The eigenvector of a linear transformation is a non-zero vector that, when applied to the transformation, changes in length, but not direction. Eigenvalues are closely related to eigenvectors. Each eigenvector has a corresponding eigenvalue which determines the amount the eigenvector is scaled under the linear transformation. The eigenvalue is a scalar value.

The following properties apply to eigenvectors:

- eigenvectors can only be found in square matrices
- not every square matrix has eigenvectors
- a $m \times m$ matrix has exactly m eigenvectors, if it has eigenvectors at all
- all eigenvectors of a matrix are orthogonal to each other

PCA step-by-step

Doing a principal components analysis is done performing the following steps:

Acquire data This step is self-explanatory, since we want to use the PCA for dimensionality reduction of data in our textures

Normalize data In this step the deviation of each value in each dimension to the mean value of that dimension is calculated and used for further processing

Ascertain the covariance build the covariance between each given dimension. The so calculated values can easily be stored inside a matrix, called *covariance matrix*. The covariance matrix is a symmetric, square matrix, with each dimension's variance as the main diagonal.

Calculate eigenvectors and eigenvalues as already mentioned before, the eigenvectors are perpendicular to each other. This crucial property allows to express the collected data in terms of these eigenvectors instead of the standard coordinate system. The new coordinate system now resembles a best-fit straight line.

In other words: by using the eigenvectors we are able to extract lines that characterize the data

Forming the feature vector is the next step to reduce the dimension of our initial data set. We now sort the eigenvectors descending by the calculated eigenvalues. This results in a list, that reflects the significance of the components. The first n eigenvectors of the list now form the so called *feature vector*, where each eigenvector forms a column.

Calculating the new data set to finally apply the dimensionality reduction we intended to, the following multiplication must be done:

$$FinalData = FeatureVector^{\mathrm{T}} \times DataAdjust^{\mathrm{T}}$$

where *DataAdjust* is the mean-adjusted data we created in step 2. *FinalData* now holds the data items as its columns and the dimensions along the rows. Thereby the initial data has been transformed into a coordinate system, that better describes the relationship between the given dimensions of the data set.

Chapter 6

Implementation

While the previous chapters provided an overview on theoretical aspects of several different ways for synthesizing textures, this section concentrates on implementation details for the approaches described in Chapter 3.

Several aspects of the software architecture will be described using the Unified Modeling Language (UML) [ZGK04].

6.1 Graphcut textures

Starting with our implementation of the patch-based algorithm of Kwatra et al., we first and foremost want to concentrate on the way perfect seams are found and managed. We start this section by providing a coarse overview over the system's general software architecture. Subsequent sections go into further detail on specific parts of the system. Note that this chapter is not intended to provide an exhaustive specification of our implementation of the algorithm, but only to highlight some special aspects of it.

6.1.1 System overview

Since our implementation does not utilize the GPU for parallel execution, the application is built using a traditional two-tier system, with graphical user interface (GUI) and

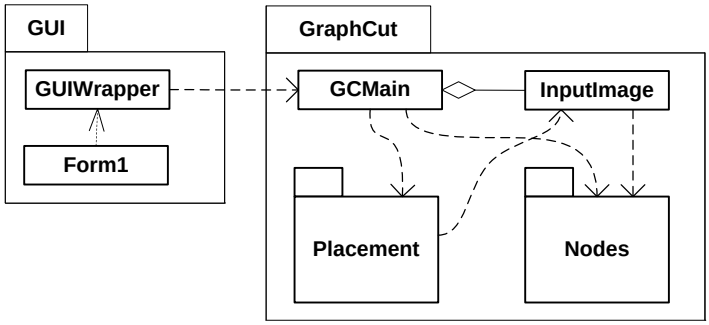


Figure 6.1: System overview

business logic as the two layers. The GUI layer is implemented using Microsoft Windows Forms, while the business logic is written in core C++. Figure 6.1 shows the different modules.

The module `GraphCut` contains all worker classes. It contains the placement classes `RandomPlacement`, `EntirePatchPlacement` and `SubPatchPlacement` and maintains the images in `InputImage` and `OutputImage`. `AbstractNode`, `OldSeamNode` and `RegularNode` represent the three types of nodes that occur during the synthesis process. Section 6.1.3 focuses on the structure of `OutputImage` and provides a detailed view on the classes used to manage the different types of nodes.

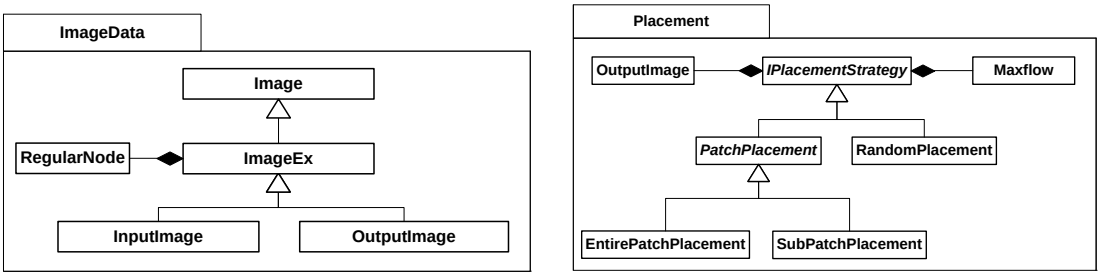
In the second module `GraphCut_GUI` two classes are of particular interest: `GUIWrapper`, which handles the communication between the GUI and the worker classes and `Form1`, the main GUI form, which uses the interface of `GUIWrapper` to interact with the worker classes. `GUIWrapper`, as the name implies, wraps the Microsoft Windows Form class structure around core C++ structures and classes and vice versa.

The interface between `GraphCut_GUI` and `GraphCut` implements the Singleton Design Pattern.

6.1.2 Package *ImageData* and *Placement*

There are three packages that play a major role in our implementation. Two of them, `ImageData` and `Placement` will be described in this section, the third `Nodes` is ana-

lyzed in further detail in Section 6.1.4.



a) Package `ImageData` b) Package `Placement`

Figure 6.2: The diagrams above show the packages for managing image-information and handling patch placement

The package `ImageData` is the simplest of the three packages that will be described here. The reason for its excessive nesting is the overall structure of the application, i.e. the interface between `GraphCut` and `GraphCut_GUI`. The class `Image` just stores color information and thus primarily used to load and save images from and to disc.

`ImageEx` is the most powerful class in the package. It adds the concept of nodes (and consequently edges) to the plain color information of `Image`. It furthermore stores extended image informations like mean value, variance and standard deviation of the represented image.

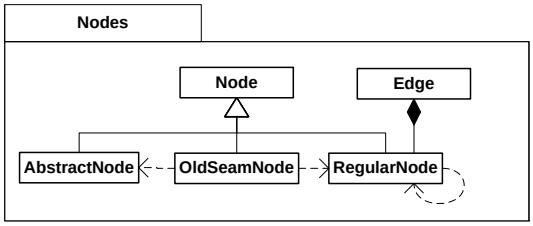
`InputImage` and `OutputImage` are the specialized objects for input and output images. A detailed specification of those two classes is beyond the scope of this work.

The `Placement` package implements the three different kinds of how new patches might be selected and positioned. Refer to Section 3.1.4 for a detailed discussion on all feasible placement strategies. Most of the placement logic is implemented in the `IPlacementStrategy` interface. It holds all methods common to each placement strategy. This includes calculating transition costs, building the input graph for the `Maxflow` class, the class that finds the optimal cut for the previously created transition-graph, interacting with the `Maxflow` class, updating the output, etc.

The specialized placement strategy classes `RandomPlacement`, `EntirePatchPlacement` and `SubPatchPlacement` determine the translation and shape of the selected input patch based on the specification in Section 3.1.4.

`PatchPlacement` is another interface that abstracts functionality, that `EntirePatchPlacement` and `SubPatchPlacement` have in common.

6.1.3 Package *Nodes*



Package `Nodes`

Figure 6.3: The diagram above shows the components that are involved when accounting for old seams

The complex structure of the package `Nodes` results from the ability of the Graphcut approach to "remember" old seams. This ability forces us to introduce an adapted data structure to account for the situation outlined in Section 3.1.2. As shown in Figure 6.4 we now need nodes that do not actually represent a pixel itself, but only an old cut. This kind of nodes are represented by the object `OldSeamNode` that are connected to two `RegularNode` objects and one `AbstractNode`. The `AbstractNode` is used as a placeholder for the new patch. The construct of the three objects in combination with the regular, undirected graph created according to Section 3.1.2 serves as input to the maxflow algorithm.

6.1.4 Defining cuts

Kwatra et al. describe two different scenarios for defining transitions between two or more patches. One for finding a seam between two patches A and B that have not been cut within their overlap region, yet, and one to cover the case, where two or more patches overlap along an old edge. Scenario 2 is the rule as Scenario 1 can only occur during initializing the output texture. Correcting old transitions is by far more

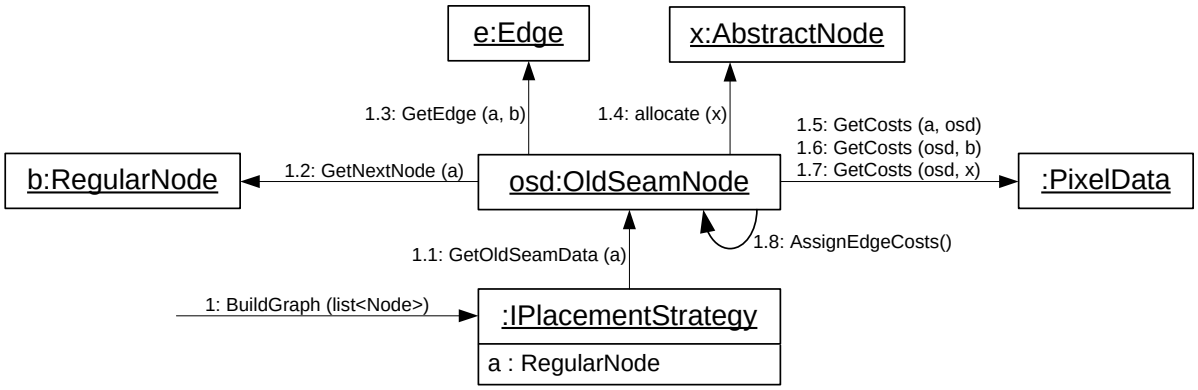


Figure 6.4: Collaboration diagram for building a graph based on existing seams

complicated than just finding a perfect seam between two patches and shall therefore be explained in more detail now.

There are three different types of nodes implemented: `RegularNode`, `OldSeamNode` and `AbstractNode`.

`RegularNode` objects are the most common nodes in the system. Unlike the other two node types, they represent real image data like color, position, owner, neighborhood information, etc. An `OutputImage` therefore is based upon the informations stored in these objects.

An `OldSeamNode` on the other hand is a node that is inserted between two `RegularNodes` to account for old seams. It is inserted before the mincut/maxflow algorithm is started. It is connected to two `RegularNodes` and one `AbstractNode` to symbolize the old seam.

An `AbstractNode` is used to establish a relation between old input patches and the new patch without loosing information about transition costs for old seams.

Building the graph for finding new transitions is somewhat difficult. At best there are two patches overlapping without an old seam running through one them. In this case the graph consists solely of `RegularNode` and `Edge` objects. As the algorithm proceeds, more and more previously defined cuts have to be taken into account. Figure 6.4 shows the procedure for setting up all required objects.

6.2 Parallel controllable texture synthesis

Our implementation of [LH05] differs quite a bit from the implementation described above. This is mainly due to the fact, that it highly exploits the parallelism provided by the GPU. It is based upon the *DirectX Utility Toolkit* to interact with the user and uses *Microsoft DirectX 10* and *HLSL* to calculate major parts during the synthesis process. It is obvious, that this architecture by far leads to a better performance than the model we used to implement [KSE+03]. However, the chosen configuration, the choice of *Microsoft DirectX 10* over *OpenGL* in particular, leads to a less generalized implementation. In other words: the implementation only works on Microsoft Windows Vista with GPUs supporting DirectX 10.

6.2.1 User Interface

The user interface of our implementation consists of a single class, that for one thing initializes DirectX and for another thing handles the interplay between the user input and the business logic. The user interface is kept simple and realized using *DirectX Utility Toolkit* to display some buttons for loading the images and some sliders for controlling the jitter factor for each of at most eight pyramid levels.

Communication between the user interface and the time consuming tasks in `ExampleUtil` are implemented as separate thread to prevent the application from not responding while the calculations take place. Furthermore the singleton design pattern is used for realizing some kind of console that renders status messages to the main application window.

6.2.2 Business logic

The business logic primarily consists of two classes, `ExemplarUtil` and `ResultUtil`, and two libraries, `newmat10` and `dt`.

ExemplarUtil as the name implies, accomplishes all tasks related to the exemplar image. This includes building the gaussian stack, as described in Section 3.2.1,

reducing color information and feature mask and building similarity sets, mentioned in Section 5.4.

ResultUtil is responsible for generating the output pyramid and therefore for controlling the three major steps: *Upsampling*, *Jittering* and *Correction* found in Section 3.2.2.

newmat10 is a math library intended for scientists and engineers that supports a variety of matrix operations. Our implementations uses `newmath10` to calculate the PCA for dimensionality reduction. The library can be found at: http://www.robertnz.net/nm10.htm.

dt is an image library that implements the fast distance transform algorithm described in [FH]. It was distributed under the terms of the GNU General Public License as published by the Free Software Foundation. In our implementation it is used to calculate the distance for the given feature mask that is stored together with the color information in the PCA reduced version of the exemplar image.

6.2.3 Rendering

Rendering is, as already mentioned, done via *DirectX Utility Toolkit*, *Microsoft DirectX 10* and *HLSL*. The time-consuming steps *Upsampling*, *Jittering* and *Correction*, that form the main real-time algorithm, are implemented in the pixel shader and thus are outsourced to the GPU.

Chapter 7

Evaluation

In this chapter an evaluation of our implementations of [KSE+03] and [LH05] is given. According to the focus of this thesis, this chapter covers a wide range of all kinds of urban datasets as input textures.

As Michael Ashikhmin criticizes in [Ash01], newly developed algorithms are often corroborated with textures possessing certain properties, gracious to the weaknesses and strengths of the particular synthesis method.

Using our implementations that are described in further detail in Chapter 3 and 6, we show the limitations of the algorithms, demonstrate the improvement of introducing weighting factors and outline the differences between our implemented sampling methods.

We walk a tightrope by synthesizing facades from extremely regular input textures and create aerial textures with very limited input data as well as street systems and river flows.

In Section 7.7 we create walls and pavements, a process exemplified in nearly each paper on texture synthesis. We will show, that a slight perspective distortion is oblivious to an observer and see the significance the input texture has on the overall synthesis process.

Section 7.8 deals with the problem of prominent features.

The chapter is concluded by a runtime analysis of our implementations in Section 7.9.

For all results presented in this chapter that were generated using our implementation of [KSE+03] we used sub-patch matching described in Chapter 3.1.4. In addition, all textures created by that implementation tile seamlessly, as announced in Chapter 3.3.

7.1 Typical parameters

Naturally, a variety of factors contribute to the successful choice of parameters that define the functioning of the algorithm. As the synthesis process in [LH05] takes place in real-time, most parameters can be chosen by trial and error. Setting up the basic conditions for [KSE+03] is a bit more tricky as it is crucial to both: runtime *and* quality. Changing values like the dimension of the overlap- or error-region requires the synthesis process to start from scratch.

Some properties, however, apply to both approaches and are decisive factors for the synthesis process. Those factors usually embrace the following attributes first and foremost:

1. the size of the input
2. the size of the output
3. the size of the patterns within the input
4. the degree of regularity within the input texture

7.1.1 Graphcut

Thinking that only runtime is the limiting factor for selecting endlessly large transition zones, i.e. overlap regions, is not true. Imagine an error region with a size of 32 pixels and an overlap region spanning 64 pixels. Such a setup would require an input image with a dimension of at least 160×160 pixels $(32 + 2 * 64)$.

As already stated above, the placement strategy used in this chapter is sub-patch matching, which only makes sense if the sub-patch can be chosen from a large set of possible offsets. That means that the ratio between the size of the input texture and

size of the new patch must be large enough to test the new patch at various different offsets of the input texture. The more different offset locations exist in the input image, the better the final result will be, since that offset is crucial to the final selection of objects along which the new seam will be drawn in the output texture.

The other extreme are error- and overlap regions that are too small to surmount certain patterns of the input image. In that case the algorithm gets stuck and starts to produce garbage.

Both situations are illustrated in Figure 7.1.

Finding the best offset in the input image is a very time-consuming task as will be outlined in Section 7.9. Setting up large overlap regions in rather big input textures leads to unsatisfactory runtime behavior.

In our test series we mostly used input images with sizes between 128 and 256 pixels in both width and height. The usual size used for overlap regions is 16 to 32 pixels. We also experimented with overlap areas that were 48 pixels wide and even with very low values as can be seen in Section 7.6, but only to demonstrate some special case behavior.

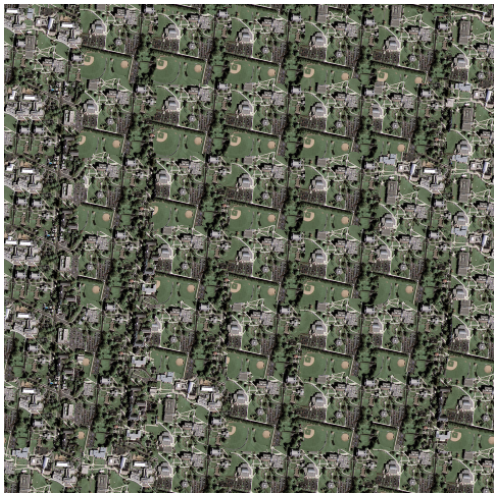
The largest error region used had a dimension of 48×48 pixels. Again, the most common value was a square of 16 to 32 pixels.

7.1.2 Parallel controllable texture synthesis (PCTS)

PCTS depends on a lot of parameters, that in contrast to our implementation of [KSE+03] can be set on-the-fly however and therefore do not have to be analyzed before starting the algorithm. As a matter of fact, there are no typical parameters for PCTS. The reason for that is the different mode of operation and the fact that PCTS works in real-time, which eliminates runtime as a limiting factor.

PCTS at its heart is based on simple tiling, due to the nature of the upsampling step, where the texture of the active layer is upsampled from the texture of the previous iteration, that is only half as wide and half as high. Therefore to eliminate any repetitive pattern one needs to disarrange the tiles in the first place. This can be achieved by

a) Input (128×128)



b) Output (480×480, ER: 64, OR: 32)

c) Input (192×192)

d) Output (320×240, ER: 8, OR: 16)

e) Input (192×192)

f) Output (320×240, ER: 8, OR: 16)

Figure 7.1: The images above show the effect of choosing awkward parameters. The texture in b) was synthesized using too large error- and overlap regions, which precluded the algorithm to find an optimal offset for the next patch. d) and f) illustrate the problem where the algorithm gets stuck, because of too small error- and overlap regions. The input texture in a) originates from a news article at http://derstandard.at. The images in c) and e) can be found at http://www.aceaerialphoto.us.

low κ high κ low κ high κ

Figure 7.2: The images above demonstrate the impact of penalizing jumps within the similarity sets. The higher the value of κ, the more jumps are penalized. Low values of κ blur the image, but smooth out errors along the borders of the different tiles.

independently setting the jitter amplitudes for up to $\log_2(m)$ levels, one for the each upsampling step, where m is the resolution of the exemplar.

Aside from the decisive factors mentioned above that affect the quality of the synthesized output, a key aspect of PCTS is covered by the value κ that penalizes jumps in the similarity sets calculated from the input texture.

This value, that can be set online during the synthesis process as well, defines the similarity of the output texture with the input sample by favoring pixels that are spatially close to each other over pixels with similar colors. In other words, it discriminates the second candidate in the created similarity sets.

In our implementation the value of κ ranges from 0 to 2,5.

Figure 7.2 demonstrates the impact of κ.

A big drawback of PCTS is that it only works with image dimensions that are multiples of 2.

Since the size and ratio of the in- and output is no limiting factor for the Graphcut approach, the texture sizes throughout this chapter used for each implementation were chosen to best fit the capabilities of the respective algorithm.

7.2 Synthesizing aerial images

The task of synthesizing textures from aerial images works quite well with both of our implementations.

As a rule of thumb it can be said that the smaller the single objects in the input texture, the better the final result. In other words, the more the objects naturally blur by moving away from the observer, the less we recognize seams between image patches and distortions. As objects in images move to the distance, details become less prominent, the object consumes fewer and fewer pixels until it can be represented by a single pixel in the extreme case. This behavior is usually utilized when working with different levels of detail in 3-d computer graphics and is extremely supportive for our pixel-based approach.

Beside the fact that object details become less observable, images holding small objects work better as input due to the fact, that the objects can be clipped as a whole. Imagine an input image with an object that consumes 50×50 pixels of the texture. With typical sizes of overlap regions between 16 and 32 pixels, as defined in Section 7.1, the algorithm would most likely not preserve the object as a whole and therefore introduce an error that is hard to correct in subsequent passes. Handling large prominent features will be discussed in Section 7.8 in further detail.

Aerial images with a lot of small, varying objects are optimal for producing convincing results. Even perspectively distorted images can be used as input for synthesizing larger outputs as long as the angle between the viewer and the ground is not too acute and the image is taken from a reasonable distance. Section 7.6 addresses this subject.

The smaller the objects in the input image, the smaller the input image might be.

As can be seen in Figure 7.3 to 7.5 both algorithms generate pretty good results out of the given input textures. While Graphcut even manages to synthesize the skyscraper in 7.4f out of a relatively small input texture, the result in Figure 7.5b looks a little static. The problem here is, that Figure 7.5b was created from an input texture that is only a fourth in size and features relatively large objects.

a) Input (192×192)

c) Input (128×128)

b) Output (320×240, ER: 8, OR: 16)

d) Output (256×256, medium κ)

e) Input (192×192)

g) Input (128×128)

f) Output (320×240, ER: 8, OR: 24)

h) Output (256×256, high κ)

Figure 7.3: The images above show examples of aerial images synthesized with Graphcut (left) and PCTS (right). The used input textures can be found at http://www.aceaerialphoto.us.

a) Input (192×192)

c) Input (128×128)

b) Output (320×240, ER: 8, OR: 16)

d) Output (256×256, low κ)

e) Input (158×192)

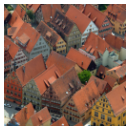
g) Input (128×128)

f) Output (320×320, ER: 16, OR: 16)

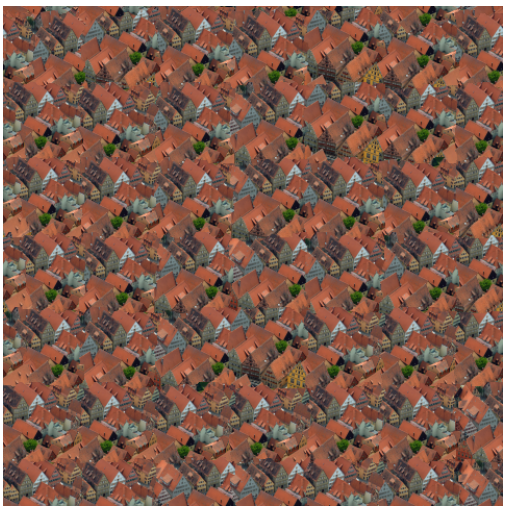
h) Output (256×256, medium κ)

Figure 7.4: The figure shows how Graphcut (left) and PCTS (right) synthesize aerial images containing relatively large objects. The input images are appropriate to our implementation of Graphcut. PCTS has a tough time with the stadium in c). The input textures can be found at http://www.aceaerialphoto.us.

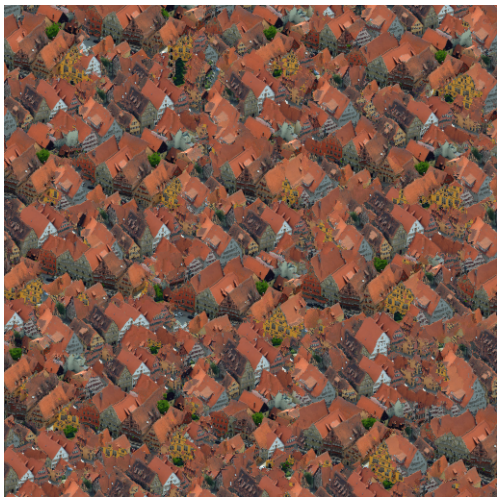
a) Input (128×112)



c) Input (128×128)



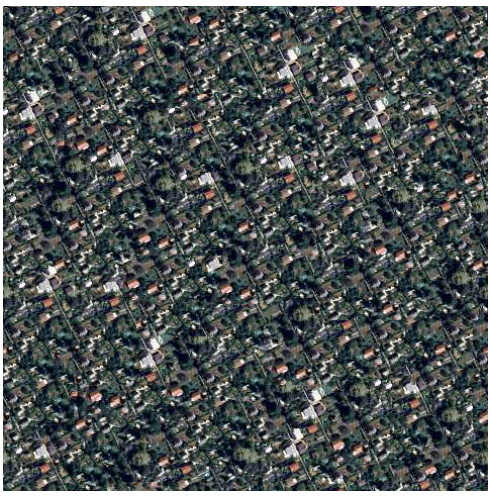
b) Output (480×480, ER: 32, OR: 16)



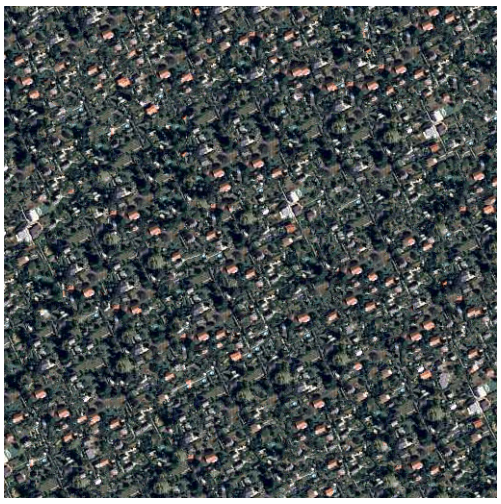
d) Output (512×512, medium κ)

e) Input (128×128)

g) Input (128×128)



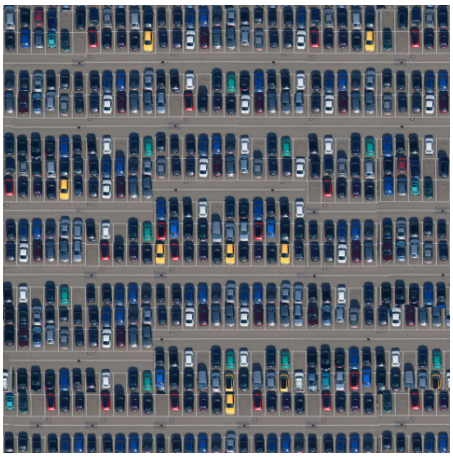
f) Output (480×480, ER: 8, OR: 8)



h) Output (512×512, high κ)

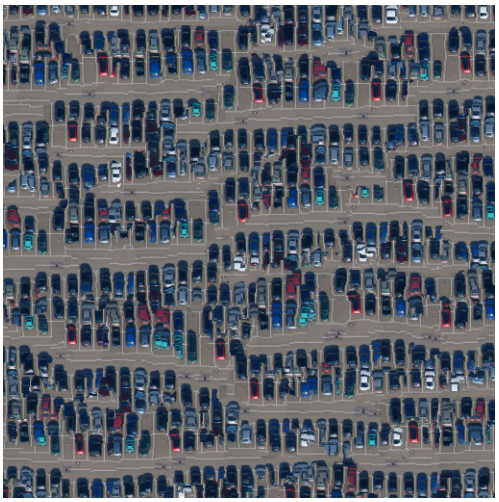
Figure 7.5: The input texture shown in a) (and c)) features rather large objects in a relatively small image. Image e) (and g) on the contrary includes tiny objects. The input textures were downloaded from http://leidorf.blogspot.com and http://www.wien.gv.at respectively

a) Input (192×192)



b) Output (480×480, ER: 32, OR: 16)

c) Input (128×128)



d) Output (512×512, medium κ)

Figure 7.6: The figure shows an image with semi-regular structure. While Graphcut (left) handles the task well, PCTS (right) cannot keep the structure up. The input texture can be found at http://leidorf.blogspot.com.

While the input texture used to synthesize Figure 7.5f is not significantly larger, the major difference is the size of the object in the input texture. This allows for smaller error- and overlap regions and therefore gives more opportunities to arrange the patches.

PCTS on the other hand has a tough time to sustain the structure of the stadium in Figure 7.4c that consumes almost one fourth of the input texture, as preserving structure is not the strength of pixel-based algorithms. This fact is also illustrated in Figure 7.6d. While the alignment and structure of the cars is too regular too generate pleasing results, PCTS does a better job on the houses in Figure 7.5d than Graphcut did. Although the output is not completely coherent, the objects are jumbled nicely while keeping the overall impression of the input.

7.3 Synthesizing facades

Synthesizing facades is a challenging task. Due on the fact that most facades have an extremely regular structure with objects, mostly windows, of the same size, synthesizing facades using a pixel-based algorithm is leading nowhere. Variations in height, width, structure, etc. are easily perceived by an observer.

Therefore synthesizing regular (or near-regular) textures that are intended to stay regular (near regular), as is the case for facades, is not possible with PCTS.

PCTS introduces variety by jittering the previously tiled input texture. Therefore the blocks are jumbled and cannot reconstruct a previously existing overall regular structure as the size of the windows and their alignment exceeds the size of the pixel neighborhood used for correcting the output. As with the example in Figure 7.6 that shows the parking cars, this sort of input is suboptimal for PCTS.

Synthesizing textures that add enough variations to not look like simply tiled textures while preserving the overall structure of the regular input image is a task that can only be done by a patch-based algorithm.

To increase the chance of synthesizing a reasonable output from a regular facade input image, one should choose the parameters in a way, that the size of the error region plus the size of the overlap region are at least as large as the largest prominent feature of the facade.

Figure 7.9d shows the benefit of applying weights to the input by using our novel extension introduced in Section 3.3. By penalizing errors that occur along the windows, the algorithm is aimed to find optimal transitions in these areas at the expense of other, less dominant regions. As can be seen in Figure 7.9d, bad transitions within the brickwork are by far not as eye-catching as along the windows.

Small features, as the one in Figure 7.7 and 7.8, do not need special weights to keep there initial structure.

a) Input (128×128)

c) Input (128×128)

b) Output (320×240, ER: 32, OR: 16)

d) Output (256×256, medium κ)

e) Input (128×144)

g) Input (128×128)

f) Output (320×240, ER: 48, OR: 24)

h) Output (256×256, low κ)

Figure 7.7: The images above show the results of synthesizing different types of facades using Graphcut in the left and PCTS in the right column. The input textures are freely available at http://www.imageafter.com/.

a) Input (123×108)

c) Input (64×64)

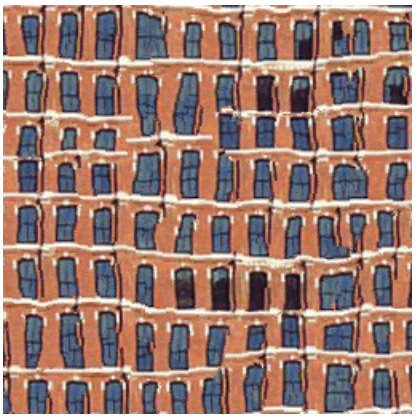
b) Output (320×240, ER: 32, OR: 16)

d) Output (256×256, medium κ)

e) Input (128×128)

g) Input (128×128)

f) Output (320×240, ER: 32, OR: 16)



h) Output (256×256, low κ)

Figure 7.8: The images above show the results of synthesizing different types of facades with a regular structure. While the input is appropriate to Graphcut (left), PCTS (right) loses the overall structure of the input. The input textures are freely available at http://www.imageafter.com/.

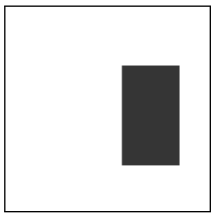
a) Input (128×128)

b) Weights

c) Unweighted output (320×240, ER: 48, OR: 16)

d) Weighted output (320×240, ER: 48, OR: 16)

e) Input (128×128)



f) Input modulation

g) Regular output (256×256, medium κ)

h) Modulated output (256×256, medium κ)

Figure 7.9: Image c) illustrates the result of the default Graphcut algorithm. d) shows the impact of our extension. g) is the output of PCTS without input modulation, while h) modulates the jitter amplitudes using the mask shown in f). The input texture is freely available at http://www.imageafter.com/.

7.4 Synthesizing land use maps

The purpose of land use maps is to provide a guide to future growth and development of a city. Land use maps help to ensure that growing populations will have adequate housing, employment and recreation opportunities. They can be used as a tool for the designation of areas.

Land use maps are typically simplified illustrations of reality with different colors identifying different areas. Synthesizing land use maps can be done with both our implementations. Depending on whether the output is intended to vary a little or to be rather regular, one approach might be better suited than the other.

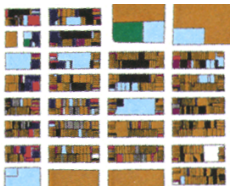
Figure 7.10 exemplify this situation. While 7.10b looks more like a typical American city designed on a drawing-board, 7.10d resembles a naturally grown old town structure.

The key benefit of using a patch-based synthesis approach again lies within the fact, that one can restrict the generated output to feature input areas of specified size from the sample image. Graphcut can be tuned to preserve patterns by defining error regions large enough to keep up an overall structure. Therefore it can be guaranteed, that the synthesized texture will not switch between the declared areas too frequently. This advantage, along with another demonstration of our Graphcut extension, is illustrated in Figure 7.11c and 7.11d.

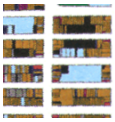
In contrast, land use maps that look like scatter-plots as the one in Figure 7.12, can be covered much better by PCTS. For this kind of input texture Graphcut tries to preserve a structure where no structure exists which results in unwanted repetitive patterns that can be seen in Figure 7.12b in contrast to 7.12d.

7.5 Synthesizing street systems and rivers

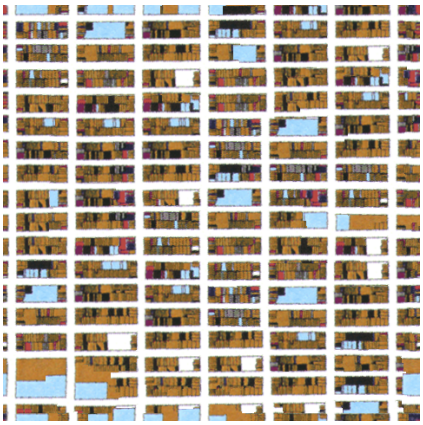
Synthesizing street systems and rivers from regular satellite or aerial images is one of the hardest things to do.



a) Input (270×215)



c) Input (128×128)

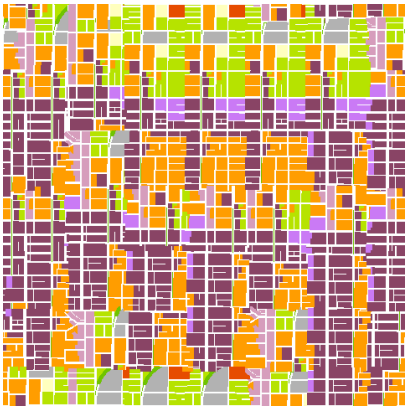


b) Output (480×480, ER: 8, OR: 16)

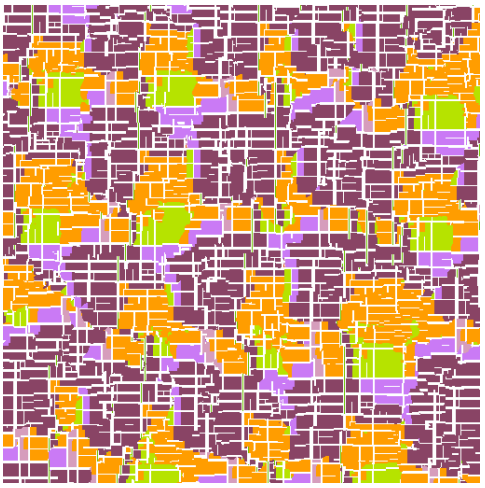
d) Output (512×512, medium κ)

e) Input (123×234)

g) Input (128×128)

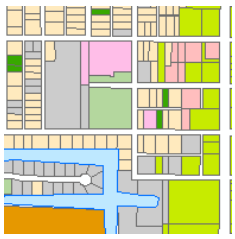


f) Output (480×480, ER: 16, OR: 32)



h) Output (512×512, medium κ)

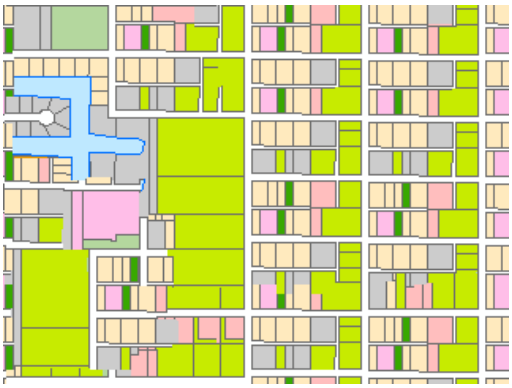
Figure 7.10: The synthesized textures above show two proponents of land use maps. While Graphcut (left) keeps the orthogonal structure of the input, PCTS (right) introduces little variation which causes the output in d) to look more like a naturally grown city. The input images can be found at http://www.east-harlem.com and http://www.mapwise.com respectively.



a) Input (192×192)

b) Weights

c) Unweighted output (320×240, ER: 8, OR: 16)



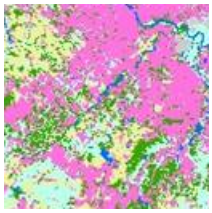
d) Weighted output (320×240, ER: 8, OR: 16)

e) Input (128×128)

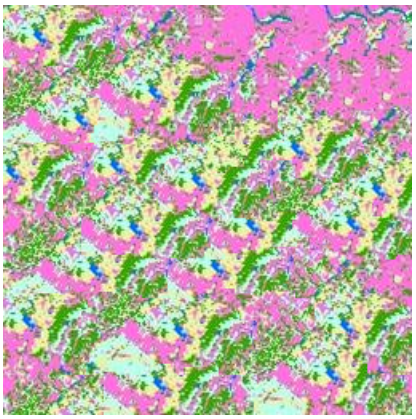
f) Output (256×256, medium κ)

Figure 7.11: The images above show another example of our Graphcut extension. While c) was synthesized using the default Graphcut approach, d) was generated using the weighting factors shown in image b). f) is the output created by PCTS. The used input texture can be found at http://www.mapwise.com.

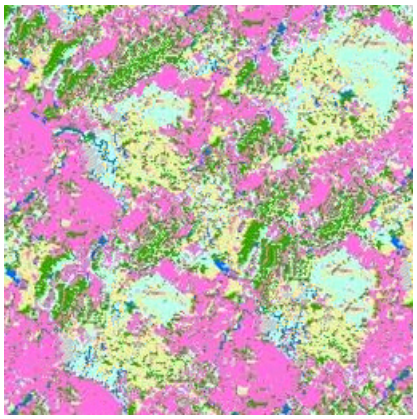
a) Input (128×128)



c) Input (128×128)



b) Output (256×256, ER: 16, OR: 8)



d) Output (256×256, high κ)

Figure 7.12: The synthesized land use map in b) does, due to the structure preserving nature of Graphcut, not look as promising as its counterpart in d) generated by PCTS. The input texture can be downloaded at http://www.nj.nrcs.usda.gov

Even with minor perspective distortion, street systems and rivers form a special case of synthesizing textures. The reason for that is that our attention is immediately drawn to roads and streams that suddenly end or take an unnatural twist.

Synthesizing an output from a texture as illustrated in Figure 7.13e is an exhausting task and takes multiple attempts to find parameters that do not produce disconnected parts of the river all over the town.

We experimented with our extension and tried to apply different weights to the image, but the quality of the output did not turn for the better.

Figure 7.13 to 7.15 again show some results we achieved with both of our implementations for similar input images. It can be seen, that PCTS produces images that can keep up with the results of Graphcut.

a) Input (192×192)

c) Input (128×128)

b) Output (320×240, ER: 48, OR: 24)

d) Output (256×256, high κ)

e) Input (239×208)

g) Input (128×128)

f) Output (320×240, ER: 48, OR: 24)

h) Output (256×256, medium κ)

Figure 7.13: The images above show the attempts to synthesize street systems and rivers using Graphcut (left) and PCTS (right). The input textures originate from http://www.wien.gv.at.

a) Input (128×128)

c) Input (128×128)

b) Output (320×240, ER: 5, OR: 16)

d) Output (256×256, low κ)

e) Input (192×192)

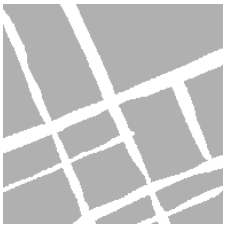
g) Input (128×128)

f) Output (320×240, ER: 8, OR: 16)

h) Output (256×256, medium κ)

Figure 7.14: The input texture shown in a) (and c) is similar to the one illustrated in Figure 7.6, however not as regular. Thus not only Graphcut (left) but also PCTS (right) creates promising results. Image f) and h) demonstrate the attempt to synthesize street systems by using a higher level of abstraction. The used input textures can be downloaded at http://www.bl.uk and http://www.wien.gv.at respectively.

a) Input (192×192)



b) Weights

c) Unweighted output (320×240, ER: 4, OR: 8)

d) Weighted output (320×240, ER: 4, OR: 8)

e) Input (128×128)

f) Output (256×256, high κ)

Figure 7.15: The image in d) is another attempt to improve the quality of the output by using our extension of Graphcut. c) shows the output of the standard Graphcut algorithm. f) was synthesized using PCTS. Applying input modulation to e) makes no sense in this case. The used input textures can be found at http://www.townofplainfield.com.

7.6 Synthesizing perspectively incorrect images

Apart from images that have an orthogonal view to the ground, like pictures taken by satellites, photographs like the one in Figure 7.6 and 2-d illustrations as used in Figure 7.14e, all input images possess a noticeable distortion caused by perspective. Simply put this means that objects close to an observer appear larger than objects in the distance.

Although this is an obvious and well-known fact, perspectively incorrect rendering is surprisingly not evident to our visual perception. Of course there is a threshold where this statement is not true any more.

Unfortunately using images that have little perspective distortion can sometimes smooth out unsightly transitions or crossovers that actually do not make sense, like streets suddenly ending or houses overlapping each other.

To verify this theory, Figure 7.16 has been synthesized with an overlap region of just 3 pixels, by far the lowest value in our test series (see Section 7.1.1 for more information about typical parameters for Graphcut). However, due to the perspective of the image (and other supporting factors like the multitude of small and quite similar object) the resulting image is satisfactory nevertheless.

7.7 Synthesizing walls and pavements

Synthesizing output images from samples with unbalanced lighting and coarse, irregularly shaped or jumbled patterns as illustrated in Figure 7.17 to 7.19 is a rewarding task. The algorithm can normally find optimal cuts between the various patches more easily and possible erroneous transitions are understated. This certainty is somewhat similar to what we observed in Section 7.2 where images from a distance yielded better results than close-ups.

Although the input texture in Figure 7.17a has a regular pattern that is out of square, the cobblestones enable nearly invisible seams when synthesized using Graphcut.

a) Input

b) Output (ER: 6, OR: 3)

Figure 7.16: [Graphcut] The synthesized result above demonstrates the impact of the input image on the output image. Although we used an overlap region of just 3 pixels, the result is quite pleasing. The error introduced by wrong perspective distortion is not distracting. There are no eye-catchers. Errors in the road system are nearly invisible.

PCTS on the other hand has a hard time correcting the errors that occur due to jittering the upsampled portions of the input. To still get an output like the one seen in Figure 7.17d, one has to reduce the value of κ and thereby accept a loss of structure for the sake of smooth transitions. See Section 7.1.2 for further information about κ.

All other images used in Figure 7.17 to 7.18 are a little more suitable for PCTS than for Graphcut, as PCTS has the ability to introduce more variation. The strength of Graphcut to preserve structures is rather counterproductive as it makes way for visible seams and repetitive patterns, which happened in Figure 7.19f to some extent.

a) Input (192×192)

c) Input (128×128)

b) Output (320×240, ER: 16, OR: 16)

d) Output (256×256, low κ)

e) Input (192×192)

g) Input (128×128)

f) Output (320×240, ER: 16, OR: 16)

h) Output (256×256, high κ)

Figure 7.17: The images above show road fragments synthesized with Graphcut (left) and PCTS (right). The lost structure seen in d) is caused by a small value of κ, that allows the algorithm to choose the second value of the candidate set. All input images are freely available at http://texturez.com.

a) Input (229×243)

c) Input (128×128)

b) Output (320×320, ER: 16, OR: 16)

d) Output (256×256, medium κ)

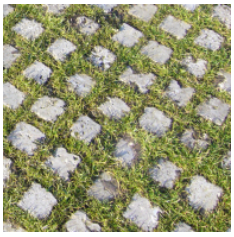
e) Input (192×192)

g) Input (128×128)

f) Output (320×240, ER: 24, OR: 16)

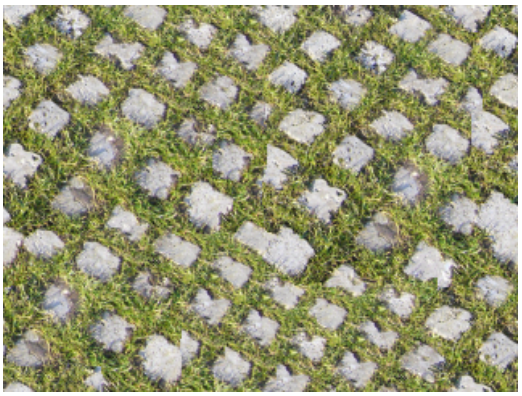
h) Output (256×256, medium κ)

Figure 7.18: Synthesizing images from input textures as shown above is a rewarding task with both approaches, Graphcut (left) and PCTS (right). While PCTS slightly blurs the images, they appears a little smoother than the results generated with Graphcut. The input images are freely available at http://texturez.com.

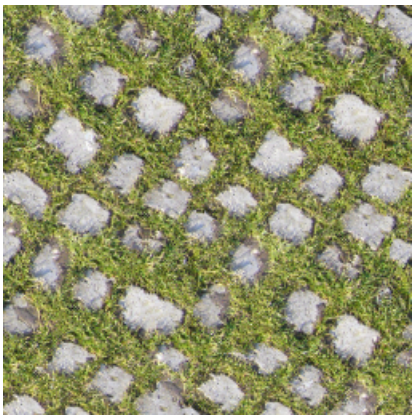


a) Input (192×192)

c) Input (128×128)



b) Output (320×240, ER: 16, OR: 16)



d) Output (256×256, high κ)

e) Input (192×192)

g) Input (128×128)

f) Output (320×240, ER: 16, OR: 16)

h) Output (256×256, high κ)

Figure 7.19: Each of the input images above lead to satisfying results for Graphcut (left) and PCTS (right). Image f), however, suffers from the introduction of repetitive patterns to some extent. All input images are freely available at http://texturez.com.

7.8 Problems with eye-catchers

Prominent features of an input texture most often lead to unsatisfying results. Although one might think, that eye-catchers could distract an observer from shortcomings in the final output, they rather tend to make the repetitive pattern of the output very obvious.

As an example, have a look at the church in Figure 7.20c. It is the first thing that grabs the attention of an observer. You can tell from the very first look that those churches do not look the same although they should. As a result an overall good output becomes useless because of such objects.

Another common eye-catcher in urban data sets is formed by apparent structures that are connected to each other like major streets, rivers, telephone lines, electrical towers, etc. As Graphcut normally assigns the same priority to all features in the overlap region, the algorithm pays no attention to such objects.

7.9 Runtime behavior

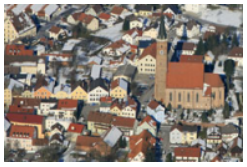
The following system configuration has been used to synthesize the pictured textures:

Operating System	Windows Vista 32-bit SP1
CPU	Intel Core 2 6600 @ 2.4GHz
RAM	4 GB
GPU	NVIDIA GeForce 8800 GTS 640MB
API	Microsoft DirectX SDK (March 2008)

Kwatra et al. specify the size of the overlap region in [KSE+03] to typically range from 4 to 8 pixels. They mostly use input textures being 64×64 or 128×128, as this is enough to synthesize textures with simple, artificial patterns.

As our focus lies on synthesizing textures for urban data, we usually need slightly larger inputs and transition areas. However, there is a limit where execution time gets out of hand while results will not get significantly better.

As we used sub-patch matching to select and arrange patches, two aspects are crucial to the overall execution time:



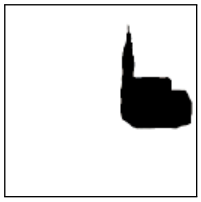
a) Input (192×128)

b) Weights

c) Unweighted output (320×240, ER: 8, OR: 16)

d) Weighted output (320×240, ER: 8, OR: 16)

e) Input (128×128)

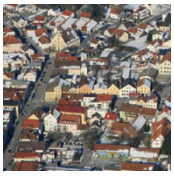


f) Input modulation

g) Regular output (256×256, medium κ)

h) Modulated output (256×256, medium κ)

Figure 7.20: The results above demonstrate the problem arising from input images with dominant objects. While our extension using weighting factors works quite well, using the input modulation for PCTS leads to a more or less simply tiled texture in h). The input image can be found at http://leidorf.blogspot.com.



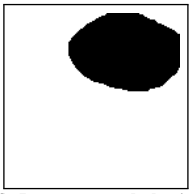
a) Input

b) Weights

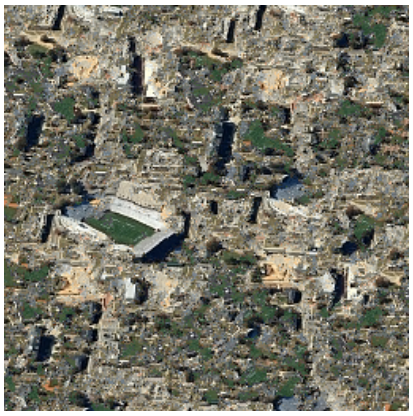
c) Unweighted output

d) Weighted output

e) Input



f) Input modulation



g) Regular output (256×256, low κ)

h) Modulated output (256×256, medium κ)

Figure 7.21: Image d) shows another example of applying weighting factors to the input (Graphcut). Image g) and h) illustrate the attempt to synthesize an output with and without input modulation using PCTS. The input images can be found at http://leidorf.blogspot.com and http://www.aceaerialphotos.us respectively.

Resolution of input texture	OR: 8px	OR: 16px	OR: 24px	OR: 32px
128×128	< 10 sec	< 15 sec	< 20 sec	< 20 sec
192×192	< 30 sec	< 50 sec	< 1:15 min	< 1:30 min
256×256	< 1:00 min	< 2:30 min	< 3:30 min	< 4:30min

Table 7.1: Run-time approximations for one iteration of Graphcut. This includes finding the best offset within the input texture, copying the patch to the output and calculating the optimal transition between the new and existing patches. The used error-regions span between 8px and 32px.

1. the size of the overlap region
2. the ratio between the input texture and the patch size

It is quite obvious that the larger the overlap region, the more pixel comparisons need to be performed for each possible translation within the input texture. Formally expressed, this means that for each possible offset $4(OR \times ER + ER^2)$ values must be compared with each other.

The second aspect is less obvious, but has an impact on both, quality and performance. Patch sizes, i.e. error- plus overlap region, that are small compared to the size of the input texture in general improve the quality of the synthesized texture, since the algorithm has more options to find perfect matches. However, the larger the ratio, the slower the performance.

In formal mathematical terms, this means that the number of comparisons necessary to find the new offset for a patch equals

$$4(OR \times ER + ER^2)(I_{width} - (ER + 2OR) + 1)(I_{height} - (ER + 2OR) + 1)$$

where ER and OR denote the size of the error- and overlap region respectively. I_{width} is the width of the input texture, I_{height} its height.

Table 7.1 lists typical values for the settings used to synthesize the images of this chapter. The larger the input texture gets, the more the duration for finding a new offset varies. The values in the table are therefore only coarse benchmarks to illustrate the abrupt rise in runtime.

Resolution of input texture	(Approximated) duration
32×32	< 2 sec
64×64	< 30 sec
128×128	~9:30 min
256×256	~3:00 h

Table 7.2: Approximated durations for creating similarity sets for various input texture sizes using exhaustive search.

PCTS on the other hand works in real-time. The only time consuming task is the creation of the similarity sets. Although the time consumed to create this information rises with larger input textures rapidly as well (see Table 7.2), it can be precomputed and stored along with the texture. Similarity sets only need to be calculated once for each input texture and can therefore not be compared with the runtimes listed in Table 7.1. Loading precomputed similarity sets and computing neighborhood information takes less than a minute, even for textures on the scale of 256×256.

Chapter 8

Conclusion

The goal of this thesis was to verify the suitability of synthesizing textures for urban data using state-of-the-art synthesis algorithms. For that purpose we decided to implement one promising candidate for each of the two most popular synthesis techniques nowadays, i.e. pixel- and patch-based approaches.

We started by structuring the field of activity, starting with early, pathbreaking algorithms. We summarized the two publications we finally chose to implement, [KSE+03] and [LH05], and outlined two alternative ways to synthesize textures.

In the course of this thesis we furthermore extended the approach of Kwatra et al. to better match our requirements. Finally we verified the benefits of [KSE+03] over [LH05] and vice versa. We created a test series for a whole range of urban textures possessing all kinds of different properties.

In general, we have learnt that synthesizing urban data with today's synthesis methods is not an easy task. Inputs including objects that draw the attention of the observer should be avoided as far as possible since they most often reveal the repetitiveness of the created output.

Significant patterns like street systems or rivers need special treatment. It appears reasonable to extend the basic implementation by some sort of feature map, where one could weight selective regions of the input texture.

Choosing inadequate parameters or feeding the algorithm with huge input images lets the performance drop dramatically.

Graphcut	PCTS
patch-based	pixel-based
calculations done on CPU	calculations done on GPU
offline	real-time
structure-preserving	correction based on immediate neighbors

Table 8.1: Differences between Graphcut and PCTS

Having an input image of 128×128 to 256×256 pixels and a sub-patch between 48×48 and 96×96 pixels worked best for [KSE+03] in most cases.

Generally speaking, it is a good idea to use an overlap region that is almost as large or even larger than the biggest prominent feature in the texture. However the most crucial factor for a successful synthesis process is and remains the input image. Figure 7.16b for instance uses an error region of just 3 pixels to synthesize a formidable output texture.

While our implementation of [KSE+03] operates on the CPU and therefore allows for arbitrary input sizes, *Parallel Controllable Texture Synthesis* (PCTS) working on the GPU only accepts multiples of 2 as dimension for in- and output textures. This must be kept in mind when comparing the outputs as well.

As we extended Graphcut to prioritize tagged blocks, we decided to also implement an extension of [LH05] to constrain the algorithm to preserve objects in marked regions.

Although the two extensions of the algorithms seem similar, they do fundamentally different things. On the one hand, our extension of Graphcut assigns weighting factors to the input image and thereby forces the algorithm to take special care of regions with higher weighting factors. This does, opposed to the extension of PCTS, not mean, that regions with large weighting factors will be used more often than others, but that errors in that areas are worse than elsewhere.

The input modification of PCTS on the other hand forces the shader to keep marked regions together by reducing the value of the jitter amplitude. Since PCTS works without the jittering step like conventional tiling, using input modification leads to repetitive patterns and therefore to less satisfactory results in general.

Our evaluation and direct comparison of [KSE+03] with [LH05], outlined in Chapter 7, showed that [KSE+03] creates better results for urban datasets in the majority of cases. This is due to the fact, that a lot of textures in our chosen domain include rather large objects, that are not infrequently regular or at least semi-regular. That is a property that the algorithm used in *Parallel Controllable Texture Synthesis* is not designed for at all.

PCTS however is great for generating textures with a lot of small elements that are irregularly arranged. Satellite images, close-ups of pavements, streets, or walls as well as abstractions like the one seen in Figure 7.12 are situations that best fit PCTS. With that type of input the algorithm creates better results than Graphcut, since it introduces a higher degree of variation.

Graphcut on the other hand preserves structures. It is able to keep the regularity of input samples and, using our weighting factor extension, capable of prioritizing and thereby keeping up regions of importance. Therefore our results not least in Figure 7.7 to 7.9 or Figure 7.6 are far better than the according outputs generated by PCTS.

Graphcut however is really slow. While PCTS synthesizes textures in real-time and thereby offering the user a high degree of control and transparency, the runtime behavior of Graphcut rises explosively with larger input images and broader overlap regions. Sometimes it even gets stuck for no obvious reason and starts to produce garbage due to awkwardly chosen parameters after doing a great job till then.

However, accepting longer runtimes does not automatically lead to better results. Simply choosing larger overlap regions or increasing the size of pixel neighborhoods often even worsens the perceptual coherence as it decreases the number of possible offsets in the input to choose the new patch from. Knowing the properties of the input sample is the most important aspect for synthesizing textures.

One of the problems of the algorithm described in [KSE+03] is the fact, that similarity is only measured by color difference. By introducing weighting factors to the approach of Kwatra et al. we tried to compensate that shortcoming and thereby provide a simple way for the user to prioritize objects or whole regions of the input texture.

While our first attempt, changing color space from RGB to CIELAB, was not the expected success, introducing weighting factors to prioritize special regions of interest

was. Although our novelty cannot solve the problem of ensuring the overall structure, like lined up windows for facades, it supports the coherence of tagged areas.

Some problems can be solved by twiddling the parameters. Combinations of keeping a regular structure *while* taking care of dominant objects is, without our extension, impossible in the majority of cases.

The beauty of using weighting factors as described in Section 3.3 also lies in its intuitive treatment. While the user can apply different levels of distinctions, i.e. not only a binary important/unimportant mask bit, it is mostly sufficient to mark single blocks as regions of particular importance.

Our third novelty, enforcing tileability for the textures synthesized by our implementation of Graphcut, can also be considered as major enrichment. In many cases, textures that tile seamlessly have benefits over those that do not. Our second implementation, that is based on [LH05], for instance needs tileable textures to create similarity sets for the pixels at the borders of the texture. If the input texture does not tile seamlessly, the texture needs to be mirrored to get an approximation of the pixel's neighborhood. The latter does not produce as good results, however, by far.

8.1 Outlook

The future definitely belongs to order-independent algorithms, that highly exploit the parallelism of modern GPUs or multicore CPUs. Real-time performance and memory efficiency are probably the most essential keywords in this context.

Approaches like [LH05] and [LH06] by Lefebvre and Hoppe already head for that direction. Moreover, Hybrid methods like [NA03] or [XGS00] seem to be promising, as they may combine best aspects from different approaches.

Future research will have to go beyond the scope of synthesizing pure color information, however. The more the algorithms reach real-time performance, the more additional information such as normal, displacement or radiance maps will play a major role. [LH06], as an example, already includes calculations like this.

Other interesting ideas include the synthesis of multiscale textures like in [HRRG08]. [HRRG08] is based on [LH06] and uses a tree-like graph data structure to represent different scalings as well as self-similarity.

Texture Synthesis is a recent field of research. As creating content becomes more and more labor intensive and time consuming, and therewith expensive, having suitable algorithms that are able to automate the process of creating general structure preserving, infinite textures would be of great value.

List of Figures

Appendix A

Bibliography

[AMH02] Tomas Akenine-Moller and Eric Haines. *Real-Time Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[Ash01] Michael Ashikhmin. Synthesizing natural textures. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 217–226, New York, NY, USA, 2001. ACM.

[BA83] Peter J. Burt and Edward H. Adelson. A multiresolution spline with application to image mosaics. *ACM Trans. Graph.*, 2(4):217–236, 1983.

[Cat74] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, 1974.

[CSHD03] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 287–294, New York, NY, USA, 2003. ACM.

[EF01] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. *Proceedings of SIGGRAPH 2001*, pages 341–346, August 2001.

[EL99] Alexei A. Efros and Thomas K. Leung. Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision*, pages 1033–1038, Corfu, Greece, September 1999.

[FH] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Distance transforms of sampled functions. Technical Report TR2004-1963, Cornell Computing and Information Science. http://people.cs.uchicago.edu/ pff/dt/.

[HRRG08] Charles Han, Eric Risser, Ravi Ramamoorthi, and Eitan Grinspun. Multiscale Texture Synthesis. *SIGGRAPH (ACM Transactions on Graphics)*, 27(3):51, Aug 2008.

[KSE+03] Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: image and video synthesis using graph cuts. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 277–286, New York, NY, USA, 2003. ACM.

[LH05] Sylvain Lefebvre and Hugues Hoppe. Parallel controllable texture synthesis. *ACM Trans. Graph.*, 24(3):777–786, 2005.

[LH06] Sylvain Lefebvre and Hugues Hoppe. Appearance-space texture synthesis. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 541–548, New York, NY, USA, 2006. ACM.

[LHZ+04] Xinguo Liu, Yaohua Hu, Jingdan Zhang, Xin Tong, Baining Guo, and Heung-Yeung Shum. Synthesis and rendering of bidirectional texture functions on arbitrary surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 10(3):278–289, 2004.

[LLH04] Yanxi Liu, Wen-Chieh Lin, and James Hays. Near-regular texture analysis and manipulation. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 368–376, New York, NY, USA, 2004. ACM.

[LLX+01] Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. Real-time texture synthesis by patch-based sampling. *ACM Trans. Graph.*, 20(3):127–150, 2001.

[Mao98] David M. Maount. Ann programming manual. Technical report, 1998.

[NA03] Andrew Nealen and Marc Alexa. Hybrid texture synthesis. In *EGRW '03: Proceedings of the 14th Eurographics workshop on Rendering*, pages 97–105, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[PFH00] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 465–470, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[Sed90] Robert Sedgewick. *Algorithms in C*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[Sha01] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, 2001.

[Smi97] Steven W. Smith. *The scientist and engineer's guide to digital signal processing*. California Technical Publishing, San Diego, CA, USA, 1997.

[Smi02] Lindsay I. Smith. *A tutorial on Principal Components Analysis*, February 2002.

[Sta97] Jos Stam. Aperiodic texture mapping. Technical report, European Research Consortium for Informatics and Mathematics (ERCIM, 1997.

[TZL+02] Xin Tong, Jingdan Zhang, Ligang Liu, Xi Wang, Baining Guo, and Heung-Yeung Shum. Synthesis of bidirectional texture functions on arbitrary surfaces. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 665–672, New York, NY, USA, 2002. ACM.

[WL00] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 479–488, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[WL03] Li-Yi Wei and Marc Levoy. Order-independent texture synthesis, 2003. http://graphics.stanford.edu/papers/texture-synthesis-sig03/.

[XGS00] Ying-Qing Xu, Baining Guo, and Harry Shum. Chaos mosaic: Fast and memory efficient texture synthesis. Technical Report MSR-TR-2000-32, Microsoft Research, April 2000.

[ZGK04] Wolfgang Zuser, Thomas Grechenig, and Monika Köhle. *Software Engineering mit UML und dem Unified Process*. Pearson Studium, 2. aufl. edition, 2004.