Universidad Polytécnica de Valencia
Departamento de Comunicaciones

Texture Synthesis Using Patch Based Sampling

A Student Research Project,
directed by
Josep Prades-Nebot

Markus Multrus

Valencia,
April 10, 2003

Acknowledgments

This project work has been accomplished at the Universidad Politécnica de Valencia, Valencia, Spain, in the frame of a Socrates study and will be submitted as a "Studienarbeit" in the study course Elektrotechnik at the Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany.

I would like to thank Josep Prades-Nebot for an excellent tutoring in Valencia. He always was an encouraged and inspiring advisor with his support over all my time in Valencia.

Further on I would like to thank Dr.-Ing Gerd Schaller and Prof. Dr.-Ing. Andre Kaup for their tutoring and their advises from my home university Friedrich-Alexander-Universität Erlangen-Nürnberg.

Contents

List of Figures

Chapter 1

Introduction

The upcoming of modern, powerful computers has enabled new possibilities for graphical work. But naturally it has also entailed new problems. Great advances have been made in creating graphical objects artificially. Surfaces of such computated objects are often filled with texture mapping. But a problem arises when the surface is larger than the available texture. The textures can not, in general, simply be tiled, because boundaries would become visible. And what if the available texture contains undesired parts? The generation of a new texture, larger and only with desired parts, would solve this problem.

Artificial generations of this kind have to be done with texture synthesis. But there are other application areas for texture synthesis. It can not only be applied to create textures without undesired regions, but also to replace these parts directly within the given texture [17]. In the same direction goes the approach to correct transmission errors, e.g. caused by packet based transmissions of images. The loss of single packets could cause (without error protection at network side) erroneous parts in the received image. These parts could be reconstructed via texture synthesis [16]. Other approaches were made to use texture synthesis for digital painting [12]. Also an application in image compression could be imagined. Textures within the images would not have to be transmitted, but could be generated from a very small texture sample.

But as many applications in graphical processing, texture synthesis also suffers from its high computational cost. Only with recent computers it is possible to solve problems of texture synthesis, as it can be seen in the further work.

1.1 Definitions

1.1.1 Texture

Although it was already talked about, until now no definition of the term *texture* was given. There are different approaches to this definition. In some publications the statement can be found that there does not exist a clear definition of texture (e.g. [7], p. 414). But there has to be a distinction from any surface pattern.

A basic feature of a texture is the periodic, aperiodic or random repetition of certain small, elementary patterns. These patterns, whose the texture consist of, are called *texels*. Natural textures consist normally of random texel placements, whereas periodic or deterministic texel placements often can be found in artificial textures ([11], p. 394). With this knowledge, a texture can be described only with these texels and rules for their repetition and placement.

1.1.2 Texture Synthesis

Texture synthesis creates a new, generally not deterministic, texture from a given, finite texture sample. The following demands are made on result and process (cp. [19], p. 6):

Visual Fidelity. Visual fidelity describes the quality of the created texture. Therefore we expect certain demands to be fulfilled. First of all from the output texture a *similarity* to the given texture sample is expected. The produced texture should have the same look as the input sample, it should contain the same structure. Texels should be combined in a comparable way. Further on it is expected from the produced texture not to look artificial. Remarkable *repetitions* have to be avoided, structures should be continued with natural *transitions*.

Effectiveness. The demand on the synthesis process is to be efficient in time and resources. It should have a small computational effort and use little memory.

1.2 General Approaches to Texture Synthesis

As already described in the definition of the term texture, it consist of a certain placement of the basic texture elements. Texture synthesis has to synthesize these texels in appearance, repetition and placement as true as possible to the original. For this primarily a fitting model has to be found, which gives a clearer characterization of the texture. There are two main categories of models, which can be found in the literature ([1], p. 108 et seqq.).

Statistical Models. The statistical models try to characterize the texture globally, whereby statistical properties of the spatial distribution of gray levels are used as texture descriptors. Thereby the description is only dependent on the statistical properties of the points, without explicit usage of texture elements like texels or subregions. In this category fall e.g. *time series models* ([1], p. 108) and *Markov Random Field models* ([1], p. 108 seqq., [4], p. 45 seqq.).

Structural Models. The structural models regard a texture as an arrangement of a set of sub patterns, positioned with certain placement rules. This is continued recursively, so that the sub pattern themselves are again made of sub pattern, positioned according to certain placement rules. With this recursive approach the hierarchical structure of natural scenes should be captured. Although a very reasonable approach, until today very little effort has been devoted to this approach.

With the help of one of these models finally a new texture has to be synthesized. Great advances in synthesizing textures have been made with the statistical models. Above all, very interesting papers have been published over the last years in the field of the Markov Random Field models [5], [6], [13], [19], [20]. In the following work we follow this approach.

1.3 Problem Formulation

As already mentioned, many different texture types with different characterization of the texel placement can be found. Goal of this work was to adapt texture synthesis using the Markov Random Field, i.e. pixel and patch based texture synthesis, to highly stochastic textures.

Textures like this can be found in the nature, e.g. textures of stone, ground, water, wood, etc.. Whenever necessary for comparison to other approaches or for demonstration purpose, the limitation to natural textures is broken. Demand on the resulting algorithm was a high effectiveness and a high visual fidelity. The algorithm should be implemented in C++.

In the last few years great advances have been made with patch based texture synthesis. Our main attention lies therefore in this approach, because it produces textures of a high visual quality with a comparative small computational effort. Nevertheless also its predecessor pixel based texture synthesis should be presented and the results compared.

1.4 Proceeding

In the following, primarily pixel based texture synthesis is presented (Chapter 2). Basing on this, in Chapter 3 patch based texture synthesis is introduced. Its free parameters are determined and the results are compared to the pixel based method. Propositions to enhance visual quality and computational cost and its effect to the synthesized texture are made and discussed in Chapter 4. Finally, Chapter 5 presents modified applications of the enhanced patch based texture synthesis algorithm.

1.5 Conventions

As far as not mentioned otherwise, the following conventions are made:

- All here used lengths are in pixels.
- All here presented images are in color, 24 bit/pixel. Synthesized images have the same color scale as their input samples.
- All here presented results were produced with our implementation of the here introduced algorithms. Examples taken from other source are marked. The implementation is based on an image processing software, developed by the Universidad Politécnica de Valencia for Microsoft Windows computers.
- Measured times were produced on a middle class desktop computer with an Intel Pentium 4, 1.70 GHz CPU and 256 MBytes RAM with Microsoft Windows XP Professional.
- All measured times are only published exemplarily for demonstration purpose.

Chapter 2

Pixel Based Texture Synthesis

In this chapter various pixel based synthesis algorithms are presented. Several papers have been published since 1999, e.g. papers from Efros and Leung [6] and Wei and Levoy [20]. They both follow in general the same approach, but with different implementations. For that we treat them as far as possible commonly. Whenever differences appear, these are mentioned explicitly. In the beginning the theoretical motivation of the algorithms is presented. It is followed by an approach to the practical implementation, in whose context also open parameters are discussed. The chapter ends with a presentation of some results. A more detailed presentation of results is not in the aim of this paper, reference is made to [6] and [20]. A detailed discussion of the results in comparison with the patch based algorithms can be found in Chapter 3.

2.1 Theoretical Approach

In the pixel based synthesis the texture is modeled as a Markov Random Field (MRF), assuming that the brightness values of a pixel are highly correlated to the brightness values of its spatial neighbors, but independent on the rest of the image [6]. The neighborhood is modeled as a window around that pixel, with size and shape of the window as free parameters.

In the following, let I be an image that is synthesized. Let I_{real} be an infinite texture, from which pixels are sampled. Further let $p \in I$ be a pixel and $w(p) \subset I$ be a neighborhood around p. The approach consists in estimating all sources of p in I_{real}. This is done by considering the stochastic dependencies in the MRF on the basis of comparing the pixel neighborhoods. From the set of pixels, which contains all supposed sources of p in I_{real}, the pixel $p' \in I_{\text{real}}$ is finally sampled randomly to I_{sample}. The estimation is done by calculating the conditional probability distribution function (pdf) $P(p|w(p))$ in I_{real}, which can be approximated by the histogram of the set $\Omega(p) = \{p' \in I_{\text{real}} | d(w'(p'), w(p)) = 0\}$, where $w'(p') \subset I_{\text{real}}$ is the neighborhood of p' in I_{real} and $d(w_1, w_2)$ is an appropriate distance between two neighborhoods w_1, w_2.

In the real case, only a finite texture sample $I_{\text{sample}} \subset I_{\text{real}}$ is available. I_{real} therefore has to be substituted by I_{sample}. In this case it is possible, that no appropriate neighborhood can be found ($\Omega(p) = \{\}$), because no distance $d = 0$ exists. For this reason, in the following it is not sampled from the pdf any longer, but from its approximation $\Omega'(p) = \{p' \in I_{\text{sample}} | d(w'(p'), w(p)) < d_{\max}\}$, where $w'(p') \subset I_{\text{sample}}$ is the neighborhood of p' in I_{sample}, and $d_{\max}$ is an appropriate distance tolerance (Figure 2.1).

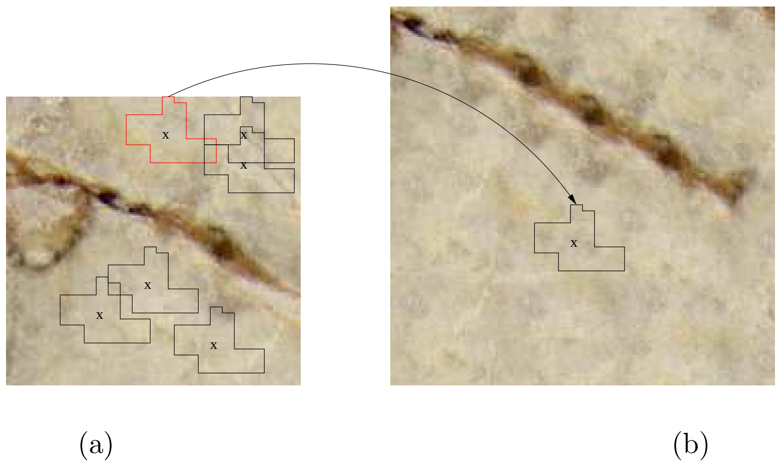


Figure 2.1: Overview pixel search process. Given a texture input sample (a) and an output image (b), in which one pixel (x) is synthesized. From all neighborhoods, which match the criterion Ω' (painted in the input sample), one is randomly selected (red), and the pixel x copied in the output image.

2.2 Free Parameters

In the theoretical approach from Section 2.1, some parameters remain undetermined. In the following these parameters are determined.

2.2.1 Neighborhood $w(p)$ and Processing

Size and shape of the neighborhood $w(p)$ are the main parameters, that determine the quality of the synthesized texture. The size should be on the scale of the largest regular structure, that should be synthesized, to catch sufficiently the stochastic constraints of the texture. Its shape is strongly related with the processing of the synthesis process, because it is recommended to use only already known pixels as neighborhood (causality). In the following, we summarize choice for size and shape made in [6] and [20].

Efros et al. [6] initialize the output texture with a 3×3 patch (*seed*), randomly taken from the input sample. Processing is done in layers outward from the already synthesized pixels and/or from the seed. The neighborhood is modeled as a square window. To match the causality criterion, only already processed pixels within this window are considered for the distance calculation (Figure 2.2).

Another approach is used by Wei et al. [20]. First, the output image is totally initialized with white noise. Then the output image is processed in raster scan order (from top to bottom, left to right). For the processing a L-shaped neighborhood is used, which ensures in general causality, apart from the edge regions. Because of the initialization with white noise, the neighborhood contains in the beginning noise, which affects the randomness of the output texture. Edges are handled in the following manner. In I_{sample} only those neighborhoods w' are considered, which are completely inside I_{sample}. To guarantee causality and tileability of the output image I, it is regarded toroidally. As soon as the neighborhood $w(p)$ exceeds I, it is expanded toroidally (Figure 2.3).

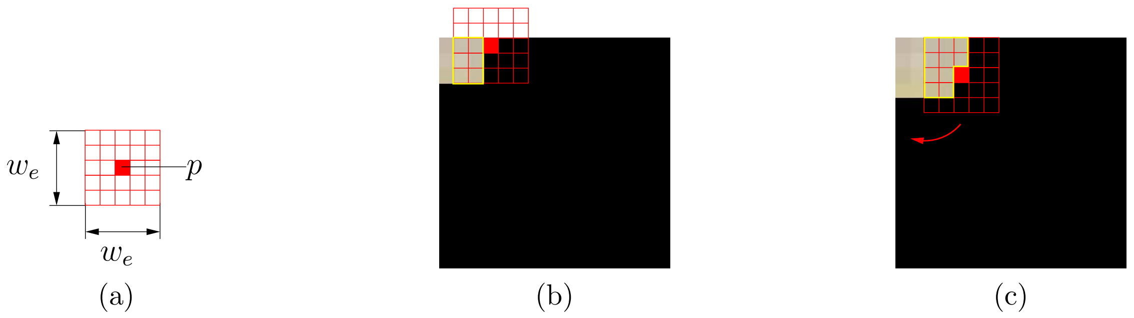


Figure 2.2: Pixel based synthesis according to Efros et al. [6]. (a) Pixel p with neighborhood $w(p)$ of size $w_e \times w_e$ (in this example $w_e = 5$), formed as square window. (b) The output image I is initialized with a seed of 3×3 pixels. Afterwards the synthesis is started. (c) The output texture is grown in layers from the seed. Only the yellow marked regions of the neighborhood are used for the distance calculation.

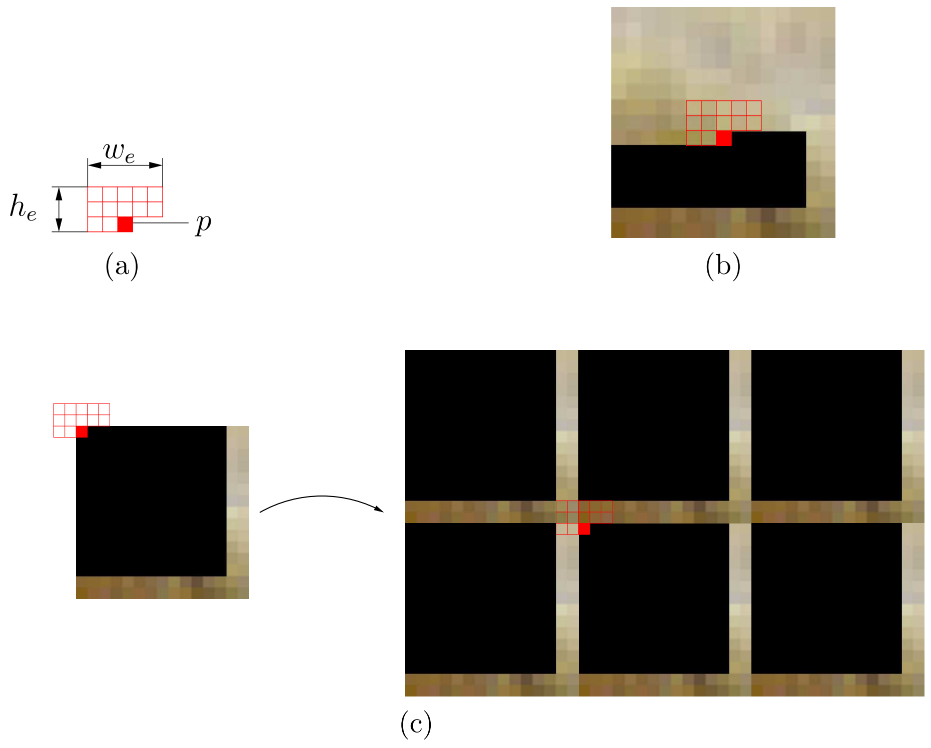


Figure 2.3: Pixel based synthesis according to Wei et al. [20]. (a) Pixel p with Neighborhood $w(p)$, $w_e = 5, h_e = 3$. (b) Synthesizing a middle pixel. (c) Start of synthesis process. To guarantee causality and tileability, the completely with noise initialized image I is expanded toroidally. Only the noise in the last two rows and columns is used, all other pixels are overwritten in the following synthesis process before they are used. For clarity, unused noise pixels are painted black.

2.2.2 Distance d and Distance Tolerance $d_{\max}$

To obtain useful results, a reasonable distance d and a distance tolerance $d_{\max}$ have to be found. In [6] a normalized L_2 norm is used to measure the difference:

$$d' = A^{-1} \sum_{k=1}^{A} (w'_k - w_k)^2, w'_k \in w', w_k \in w,$$

where A is the number of processed pixels and w' and w are two, in size and shape identical, neighborhoods. Normalization is done to compensate the different numbers of pixels used for the distance calculation during the synthesis of the whole texture. Further on, to give the pixels near p a higher weight than the outer pixels, d is set to $d = d' * G$, where G is a two-dimensional Gaussian kernel. The selection is done with a variation of the nearest neighbor technique:

$$d_{\max} = (1+\epsilon) d(w(p), w_{\text{best}}),$$

where $w_{\text{best}} = \text{argmin}_{w'} d(w', w(p)), w' \subset I_{\text{sample}}$. In [6], ϵ is set to $\epsilon = 0.1$. We obtain a resulting

$$\Omega'(p) = \{p' \in I_{\text{sample}} | d(w'(p'), w(p)) < (1+\epsilon) d(w(p), w_{\text{best}})\}$$

from which it is sampled randomly.

Wei et al. use the L_2 norm for distance calculation, but without any convolution. Afterwards the neighborhood w' with the smallest distance d is selected. The distance tolerance $d_{\max} = \text{argmin}_{w'} d(w', w(p)), w' \subset I_{\text{sample}}$.

$$\Omega'(p) = \{p' \in I_{\text{sample}} | d(w'(p), w(p)) = \text{argmin}_{w'} d(w', w(p))\}.$$

2.3 Computational Efforts

Pixel based synthesis has a high computational cost, which consist mainly of the high number of gray value differences of the single pixels, that has to be calculated. Let w_{sample} and h_{sample} be the width and the height of the input sample. According to [20] a L-shaped neighborhood is assumed. To synthesize one pixel, $w_{\text{sample}} \cdot h_{\text{sample}}$ search steps are needed. Further on for each search step $(h_e - 1)w_e + \lceil (w_e/2) \rceil$ pixel comparisons (difference operations) have to be made. Consequently the total computational cost consists of

$$n_{op} = w_{\text{sample}} \cdot h_{\text{sample}}((h_e - 1)w_e + \lceil (w_e/2) \rceil)$$

difference operations for the synthesis of one pixel. The computational cost of Efros' algorithm is in the same order of magnitude. Figure 2.4 shows that the number of operations increases exponentially with the size of the input sample.

2.4 Results

We implemented Wei's algorithm. The algorithm produces good results, as already Wei et al. show in [20]. But despite of that it is too slow. To create an output image of 200×200 from an input sample of 128×128 (with 256 gray levels), more than 2100 s (more than 35 minutes) are needed by our implementation with a neighborhood of height $h_e = 3$ and width

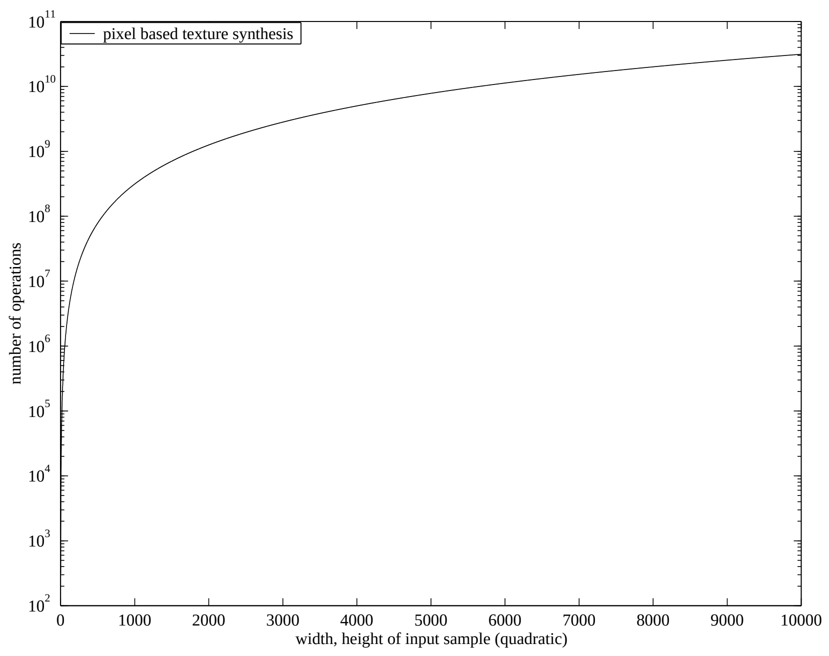
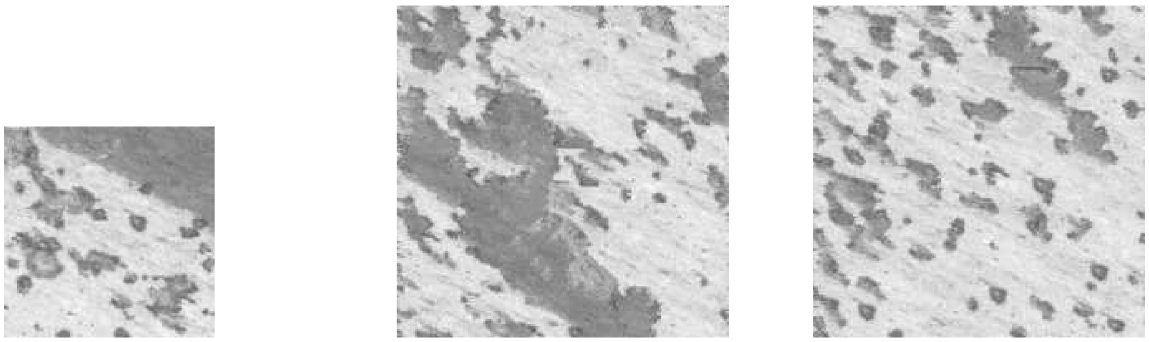


Figure 2.4: Number of operations/pixel depending on the size of the input sample ($w_e = 25$, $h_e = 13$). A quadratic input sample is assumed.

$w_e = 7$, and more than 2700 s (more than 45 minutes) with a neighborhood of height $h_e = 3$ and width $w_e = 9$. But one has to remember that with neighborhoods of these sizes, only small stochastic constraints can be captured. To reproduce bigger texels sufficiently, larger neighborhoods are needed, which increase the computational cost enormously.



(a) (b) (c)

Figure 2.5: Pixel based synthesis. Results. (a) Input sample (128×128), gray scale 8 bit/pixel (b) Synthesized image, $w_e = 7$, $h_e = 3$. (c) Synthesized image, $w_e = 9$, $h_e = 3$.

Chapter 3

Patch Based Texture Synthesis

Pixel based texture synthesis has lead to good results. But despite of that it is still too slow. One thing, that can be seen with pixel based synthesis, is that neighbored pixels are normally highly correlated. Imagine a circle on a plane: Once a part of the circle has been synthesized, all other pixels are determined [5]. As conclusion of the upper assumption can be drawn, not to synthesize single pixels, but larger texture parts at once. In this chapter various approaches to this are presented. First, the algorithm is motivated theoretically. Second, the practical implementation is approached and free parameters are discussed. Finally, we end with a presentation of the results in comparison to the pixel based method (computational efforts, visual fidelity). The various approaches are treated as far as possible commonly. Whenever differences appear, these are mentioned explicitly.

3.1 Theoretical Approach

Various similar papers have been published, which deal with the approach, not to sample single pixels, but patches from an input sample [5],[13]. In the beginning the approach of pixel based synthesis (Chapter 2) is followed. Again the texture is modeled as a MRF, the brightness value of a pixel being highly correlated to the brightness values of its spatial neighbors.

Let I be an image that is synthesized from an infinite texture I_{real}. Let further be $R \subset I$ be a square *patch* of pixels with the neighborhood (*seam*) $\delta R(R) \subset I$ modeled around R. Finally we define a *block* $B = (R \cup \delta R(R)) \subset I$ as the combination of patch and seam

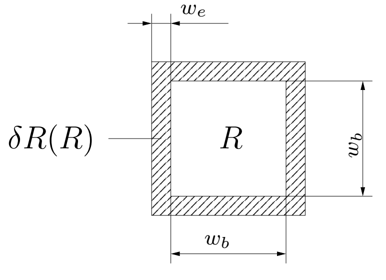


Figure 3.1: Patch based sampling. Block B consisting of patch R with surrounding neighborhood $\delta R(R)$.

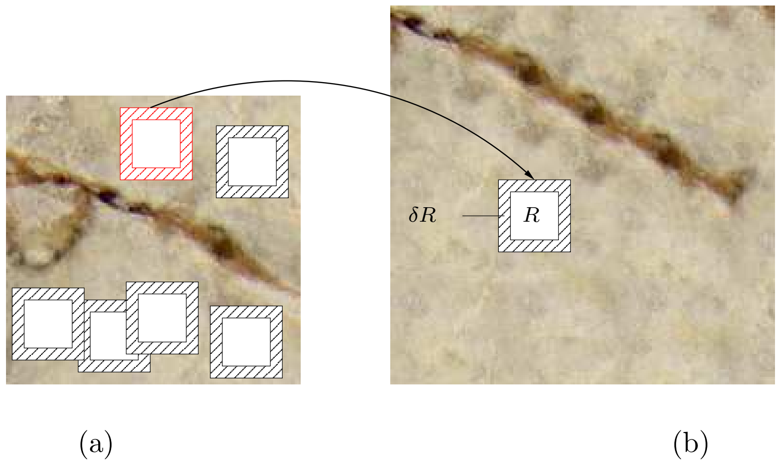


Figure 3.2: Overview patch search process. Given a texture input sample (a) and an output image (b), in which one patch R is synthesized. From all neighborhoods $\delta R'$, which match the criterion Ω' (painted in the input sample), one is randomly selected (red), and the corresponding patch is copied to the output image.

(Figure 3.1). The approach consists - like in the pixel based texture synthesis from Chapter 2 - in estimating a set containing all sources of a certain patch R in I_{real}. This estimation is done in consideration of the stochastic dependencies in the MRF by a comparison of the neighborhoods. From the set of patches, which contains all supposed sources of R in I_{real}, the patch $R' \subset I_{\text{sample}}$ is finally sampled randomly to I. Like in Section 2.1, the estimation is done by calculating the conditional pdf $P(R|\delta R(R))$ in I_{real}. This can be empirically approximated by the histogram of the set $\Omega(R) = \{R' \subset I_{\text{real}} | d(\delta R'(R'), \delta R(R)) = 0\}$, where $\delta R'(R') \subset I_{\text{real}}$ is the neighborhood of R' in I_{real} and $d(\delta R_1, \delta R_2)$ is an appropriate distance between two neighborhoods δR_1 and δR_2.

In real case, once again an image with finite texture $I_{\text{sample}} \subset I_{\text{real}}$ is introduced, and so I_{real} is substituted by I_{sample}. Consequentially, Ω has to be adapted to an $\Omega'(R) = \{R' \subset I_{\text{sample}} | d(\delta R'(R'), \delta R(R)) < d_{\text{max}}\}$, where $\delta R'(R') \subset I_{\text{sample}}$ is the neighborhood of R' in I_{sample} and d_{max} is an appropriate distance tolerance.

3.2 Further Approach to the Algorithm

3.2.1 Edge Handling

Only patches are considered for sampling, whose blocks $B' \subset I_{\text{sample}}$ are completely in I_{sample}. Further on patches are copied *with* the neighborhood as whole blocks to the output image I (see Section 3.2.3). For all blocks B', which are sampled directly to the boundary of the output image I, the patches R' are placed exactly at the boundary of I, with no neighborhood $\delta R'(R')$ between R' and and the boundary of I (Figure 3.3). If a block B' exceeds the output image boundaries, only the pixels within the boundaries would be processed for distance calculation.

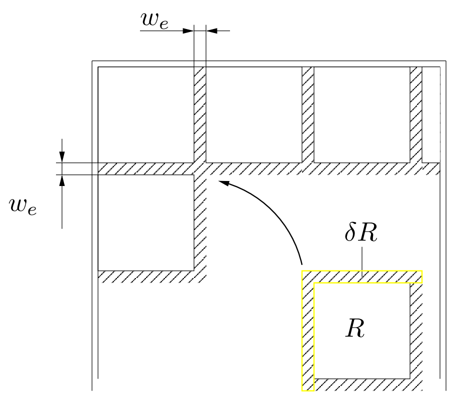


Figure 3.3: Arrangement of blocks in the output image. The blocks are copied with their neighborhood in a manner that these seams overlap. Note that for blocks at the image boundary no seam is copied. Only the yellow marked part of the neighborhood δR is used for distance calculation.

3.2.2 Processing and Causality Criterion

The output image is initialized with a randomly copied block from I_{sample}, which is placed in the upper left corner of the output image. The remaining blocks are sampled in raster scan order, i.e. from top to bottom and from left to right.

To match the causality criterion, not the whole neighborhood $\delta R(R)$ is used for distance calculation, but only the upper and left part of this neighborhood. If R touches an image boundary, the according part of the neighborhood is neglected because of above mentioned edge handling (Section 3.2.1, Figure 3.3).

3.2.3 Arrangement of Blocks

It could be suggested, just to copy the selected patch R' to the output image. As it can be seen in Figure 3.4 (b), block boundaries can easily be seen, and this method does not satisfy a high visual fidelity. In [5] and [13] two ways to solve this problem are proposed. Both efforts use the seam to advance the results. The blocks B' are copied to the output image in a manner, that the seams overlap. (Figure 3.3)

In [5] the overlapping process is done by calculating the minimum cost path in the seams between the newly chosen block and the already sampled blocks. Finally the new block is pasted along this path. We do not follow this approach. Instead, to reduce computational efforts, another approach is followed. In [13], p. 7, a simple blending (*feathering*, [18], p. 252) between the two seams is proposed. It weightens the pixels in each block proportionally to their distance to the edge of B' (Figure 3.5). A disadvantage of this solution is, that the algorithm smoothes the texture along the boundaries, which could have a negative effect to the sharpness of the texture.

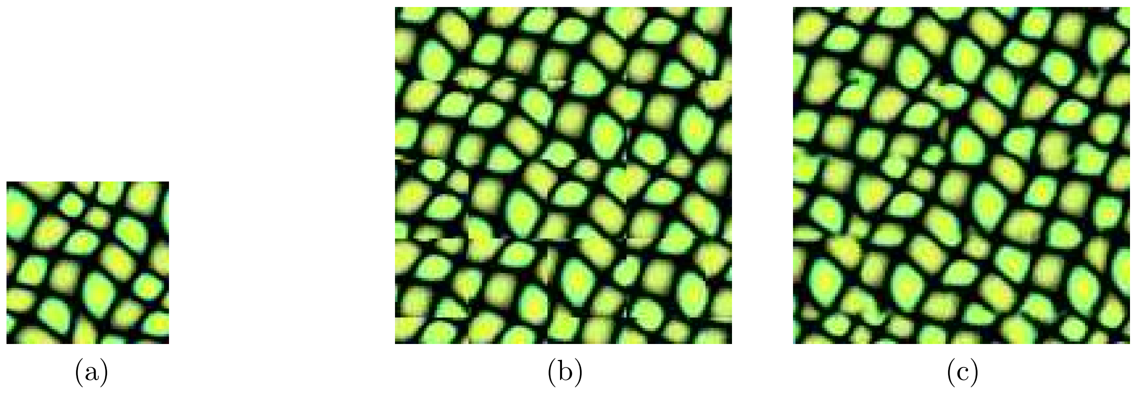


Figure 3.4: Block arrangement with and without blending. (a) Input texture [2]. (b) Patch based synthesis without blending. Block boundaries can easily be recognized. (c) Patch based synthesis with blending.

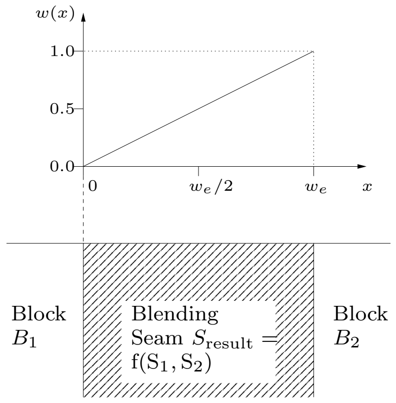

$w(x)$
1.0
0.5
0.0
0
$w_e/2$
w_e
x
Block B_1
Blending Seam $S_{\text{result}} = \mathrm{f}(\mathrm{S}_1, \mathrm{S}_2)$
Block B_2

Figure 3.5: Feathering of the blocks B_1 (with seam S_1, left) and block B_2 (with seam S_2, right). The image in the seam region S_{result} results from a blending of the brightness levels of S_1 and S_2, linear dependent on the distance from the block edges. $S_{\text{result}}(x) = (1 - w(x))S_1(x) + w(x)S_2(x)$.

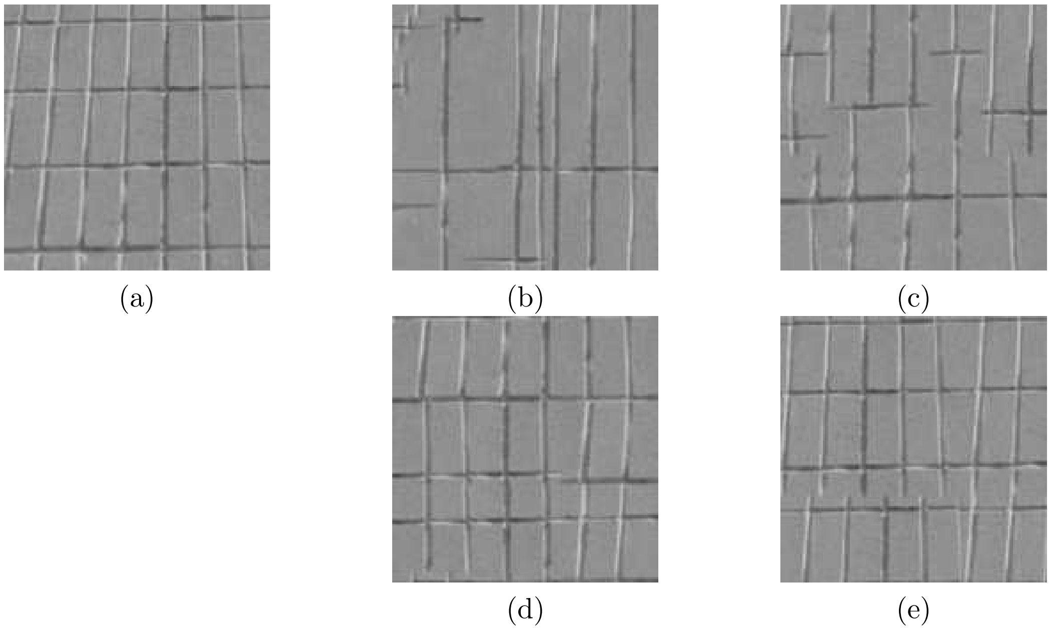


Figure 3.6: Comparison of different patch sizes w_b. (a) Input texture [3], gray scale 8 bit/pixel. Size of the characteristic bricks about 20×40 (width $\times$ height). (b) Synthesized image, $w_b = 5$, $w_e = 4$: In general the texture structure is not reproduced correctly. (c) Synthesized image, $w_b = 25$, $w_e = 4$: Horizontal structure is recognizable. (d) Synthesized image, $w_b = 45$, $w_e = 4$: Horizontal and vertical structures are reproduced. (e) Synthesized image, $w_b = 100$, $w_e = 4$: Structures are reproduced, but only little interaction between these structures is left over to the algorithm.

3.3 Free Parameters

3.3.1 Patch Size w_b

The patch size is the most critical parameter of this algorithm, and can be compared to the neighborhood in the pixel based method. The patch has to capture the statistical constraints of the input texture and transfer them to the output image, which has in the pixel based synthesis the neighborhood to do. A smaller w_b means more randomness in the output image and vice versa. The patch size should be big enough to capture the biggest regular structure in the texture. But it should not be too big, so that interaction between these structures is left over to the algorithm ([5], Figure 3.6).

3.3.2 Seam Size w_e

The seam size should be big enough to capture statistical constraints across patch boundaries: A large w_e catches strong statistical constraints, which forces a natural transition of texture features across boundaries. In our tests the width of the seam is largely independent on other parameters, and - depending on the texture - good results *can* be reached with a very small seam (e.g. for very smooth textures). A large effect to the visual fidelity of the output image has also the blending, which must not be neglected. Good results have been obtained with seams of widths $w_e \approx 5$. It is important not to make the seam too big, to avoid introduced errors and a loss of sharpness because of the blending, and to reduce computational efforts. (Figure 3.7)

3.3.3 Distance d

For distance calculation we use

$$d(\delta R', \delta R) = [A^{-1} \sum_{k=1}^{A} (\delta R'_k - \delta R_k)^2]^{1/2}, \delta R'_k \in \delta R', \delta R_k \in \delta R,$$

where A is the number of processed pixels and δR and $\delta R'$ are two, in size and shape identical, neighborhoods.

For color images the distance is calculated for each RGB component separately. The resulting distance d is calculated by the quadratic mean of the RGB distances.

$$d = \sqrt{\frac{1}{3}(d_R^2 + d_G^2 + d_B^2)}$$

3.3.4 Distance Tolerance $d_{\max}$

In [13], p. 9 the distance tolerance is dependent on a quality parameter ϵ and the neighborhood $\delta R(R)$ of the actually to be synthesized patch R:

$$d_{\max} = d_{\max,\ \epsilon} = [A^{-1} \sum_{k=1}^{A} (\epsilon \delta R_k)^2]^{1/2},$$

where A is the number of processed pixels and $\delta R_k \in \delta R(R)$ are the brightness values of the pixels in the neighborhood of R. If none of the processed block neighborhoods matches this distance tolerance, the block with the minimum distance is selected. A value of $\epsilon = 0.2$ is proposed in [13], p. 9.

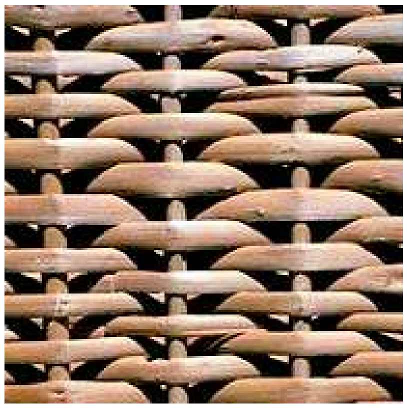
In contrast to that, we just set the distance tolerance dependent on the n best blocks:

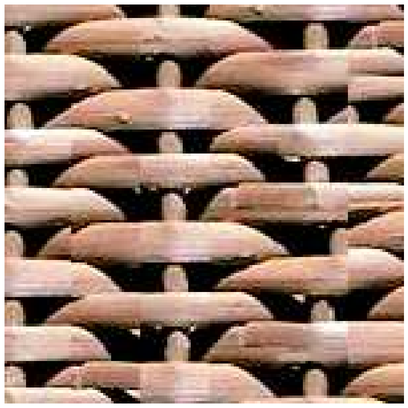
$$d_{\max} = d_{\max,\ \mathrm{n}} = d(\delta R'_{n+1}, \delta R(R)),$$

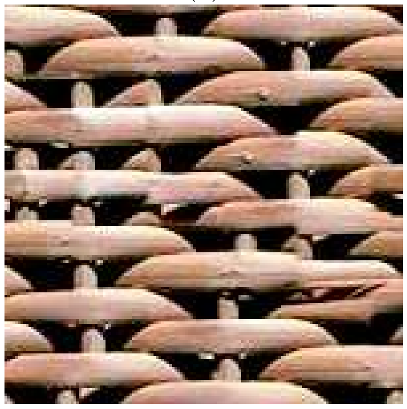
where $\delta R'_{n+1}$ is the (n+1)-th best matching neighborhood (neighborhood with the (n+1)-th smallest difference $d(\delta R', \delta R(R)), \delta R' \subset I_{\text{sample}}$) to the to be sampled patch $R \subset I$. So

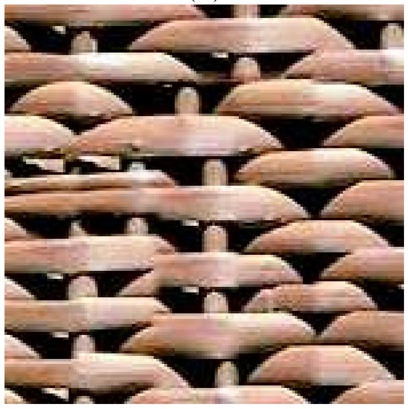
$$\Omega'(R) = \{R' \subset I_{\text{sample}} | d(\delta R'(R'), \delta R(R)) < d(\delta R'_{n+1}, \delta R(R))\},$$

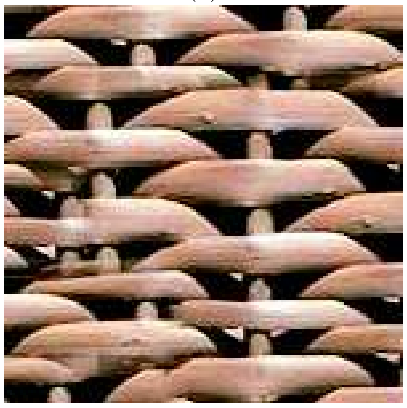
where $\delta R'(R') \subset I_{\text{sample}}$ is the neighborhood of R' in I_{sample}. This choice has the advantage to determine directly the randomness of synthesis process by the parameter n. The quality is ensured because of the selection of the n *best* blocks. Good results have been obtained with $n \approx 5$. With this choice of a distance tolerance, a with the factor A^{-1} normalized distance d is not necessary. Despite of that we leave it to enable an easy comparison with Liang's distance tolerance $d_{\max,\ \epsilon}$.


(a)


(b)


(c)


(d)


(e)

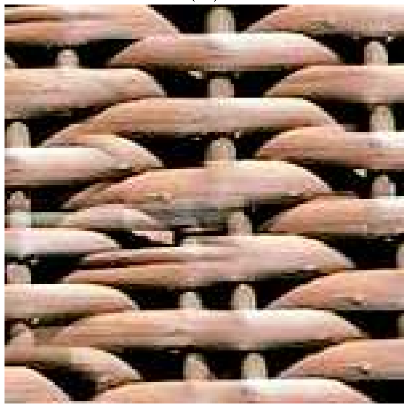

(f)

Figure 3.7: Comparison of different seam sizes w_e. (a) Input texture [14]. Patch size $w_b = 25$. (b) Synthesized image, $w_e = 1$. Problems with the feature matching can be seen. (c) Synthesized image, $w_e = 3$. Smoother transitions can be observed. (d) Synthesized image, $w_e = 5$. Best result. Good feature matching, smooth transitions. (d) Synthesized image, $w_e = 10$. Smooth transitions, good feature matching, but first errors introduced by blending. (e) Synthesized image, $w_e = 20$. Strong errors introduced by blending.

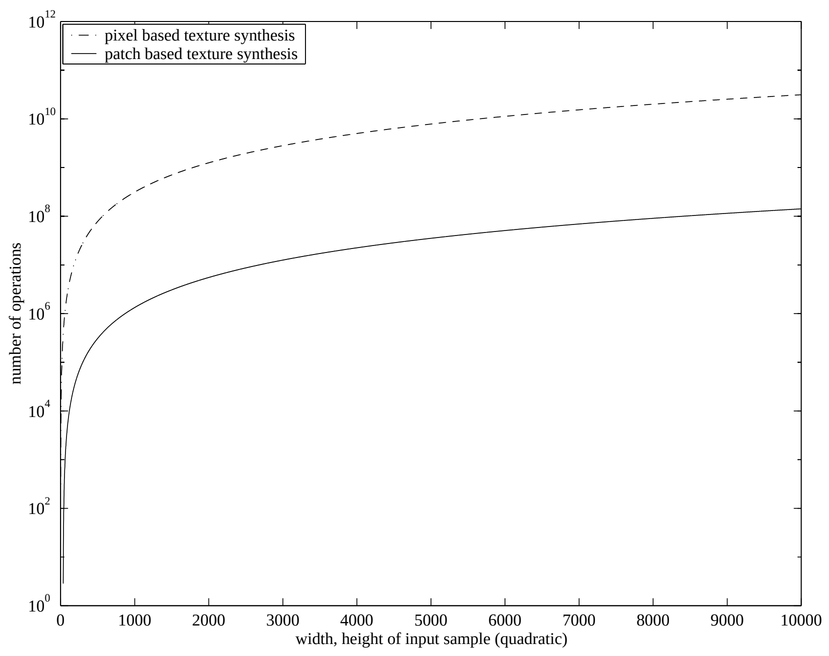


Figure 3.8: Number of difference operations/pixel depending in the size of I_{sample} (assuming a quadratic I_{sample}). Patch based synthesis: $w_b = 25$, $w_e = 5$. Pixel based synthesis: $w_e = 25$, $h_e = 13$.

3.4 Computational Efforts

Patch based synthesis has in comparison to the pixel based synthesis process in common a reduced computational effort. Cause of the reduction is the sampling of whole blocks instead of single pixels. Let w_{sample} and h_{sample} be the width and the height of the input sample. The number of search steps for the synthesis of one block is $(w_{\text{sample}} - w_b - 2w_e + 1)(h_{\text{sample}} - w_b - 2w_e + 1)$. For each search steps $2w_e(w_b + 2w_e)$ pixels in the neighborhood are processed (note the overlap in the upper left corner), and for each a difference of gray values is calculated. The resulting effort has to be be divided by the number of pixels in a block, to obtain the values for one pixel. The resulting number of difference operations for one pixel is

$$n_{op} \approx \frac{(w_{\text{sample}} - w_b - 2w_e + 1)(h_{\text{sample}} - w_b - 2w_e + 1) \cdot 2w_e(w_b + 2w_e)}{(w_b + 2w_e)(w_b + 2w_e)}.$$

Please note that in this approximation a precise edge handling has been neglected.

Both algorithms, pixel and patch based approaches, show a parallel exponential increasing number of difference operations depending on the size of the input sample (Figure 3.8). But a clear enhancement in effectivity can be seen using the patch based texture synthesis.

3.5 Results

As the Figures 3.9, 3.10 and 3.11 show, patch based sampling produces better results with less temporal efforts than pixel based sampling. Sampling of whole blocks conserves well the

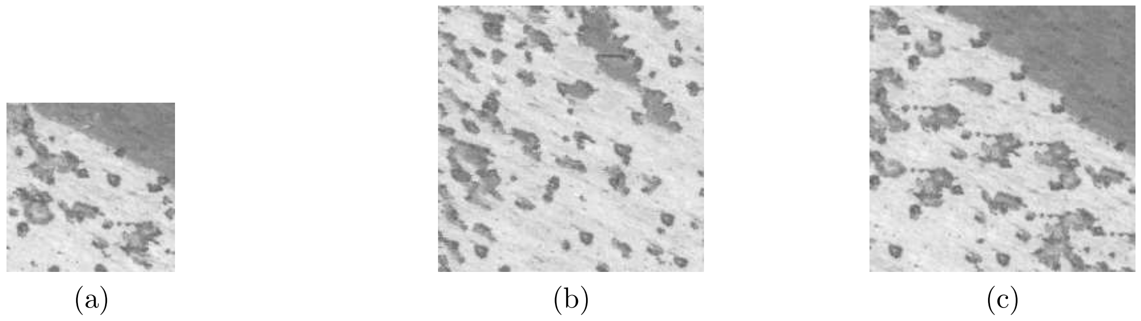


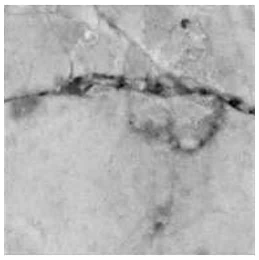
Figure 3.9: Patch based synthesis. Results (1). Size of the input texture 128×128, size of the synthesized images 200×200. In brackets time in seconds for synthesis process. (a) Input image, gray scale 8 bit/pixel (b) Synthesized image, pixel based synthesis, $w_e = 9$, $h_e = 3$ (2780 s). (c) Synthesized image, $w_b = 25$, $w_e = 5$ (10 s).

character of the texture and leads to naturally looking output textures. Main advantage in respect to pixel based sampling is the capture of bigger stochastic constraints by patch and seam than in pixel based sampling with comparable computational cost. Until today pixel based sampling suffers from the constricted neighborhood because of a too high computational effort. This can easily be seen in (Figures 3.10 (b), (e), (h)). Bigger, strongly from the background varying structures are not reproduced. These structures are better reproduced by the patch based synthesis, because of the copying of whole blocks to the output image. This approach captures automatically the structures of the foreground. In common no block boundaries are visible because of the blending, which though has a low computational cost.

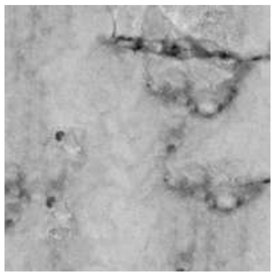
Figure 3.12 shows exemplarily a map of distance values during the block search process. The next synthesized block B is chosen from the blocks with the n lowest values. It can be seen that for the highly stochastic texture (a) exist only few local minimums, with a slowly increasing environment. In contrast to that many, periodic local minimums can be seen in the highly deterministic texture (d), with rapidly increasing environment. Also clearly the structure of the texture (texels) can be recognized in (f).

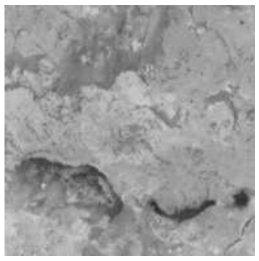
However various disadvantages of the block based method can be seen. It can not be avoided, that sometimes block boundaries become visible. Cause of this is normally a bad continuation of existing structures (Figure 3.10 (i), 3.11 (f)). Another problem is that patch based synthesis tends - other than pixel based synthesis - to visible repetitions of certain blocks (and neighbors) in the output image (Figure 3.9 (c) at the boundary to the dark part). This problem depends on the noticeability of the repeated block. So disturbs the repetition of very smooth blocks less than the repetition of very eye-catching structures. The problem gets worse when synthesizing large images from very small input samples. Nevertheless a higher stochastic variability would be desirable. Further on as in Section 3.4 can be seen, the computational efforts are still too high, especially when synthesizing from bigger input samples. So the synthesis of an image of the size 200×200 ($w_b = 25$, $w_e = 5$) from a color input image I_{sample} of the size 200×200 needs more than 93 s (128 s from 300×300, 440 s from 400×400, 716 s from 500×500 etc.). The effort in time is far away from real time synthesis.

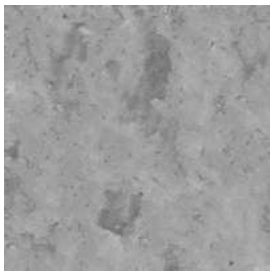
As final result can be drawn that patch based synthesis produces - especially for highly

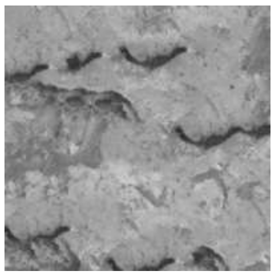

(a)

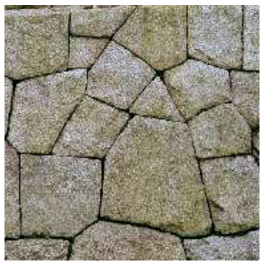
(b)

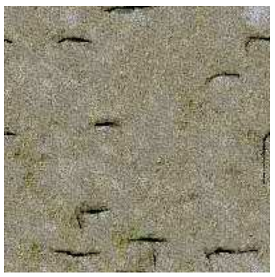

(c)


(d)


(e)


(f)


(g)


(h)

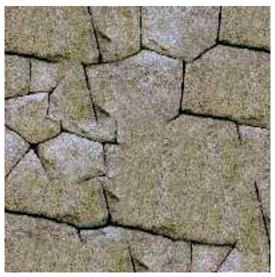

(i)

Figure 3.10: Patch based synthesis. Results (2). Size of the input textures 192×192, size of the synthesized images 200×200. In brackets time in seconds. Image (h) was downloaded from Wei's Web page [21]. (a) Input image, gray scale 8 bit/pixel. (b) Synthesized image, pixel based synthesis, $w_e = 9$, $h_e = 3$ (6447 s). (c) Synthesized image, patch based synthesis, $w_b = 25$, $w_e = 5$ (32 s). (d) Input image, gray scale 8 bit/pixel. (e) Synthesized image, pixel based synthesis, $w_e = 9$, $h_e = 3$ (6488 s). (f) Synthesized image, patch based synthesis, $w_b = 25$, $w_e = 5$ (33 s). (g) Input image [14]. (h) Synthesized image, pixel based synthesis [21]. (i) Synthesized image, patch based synthesis, $w_b = 40$, $w_e = 5$ (64 s).

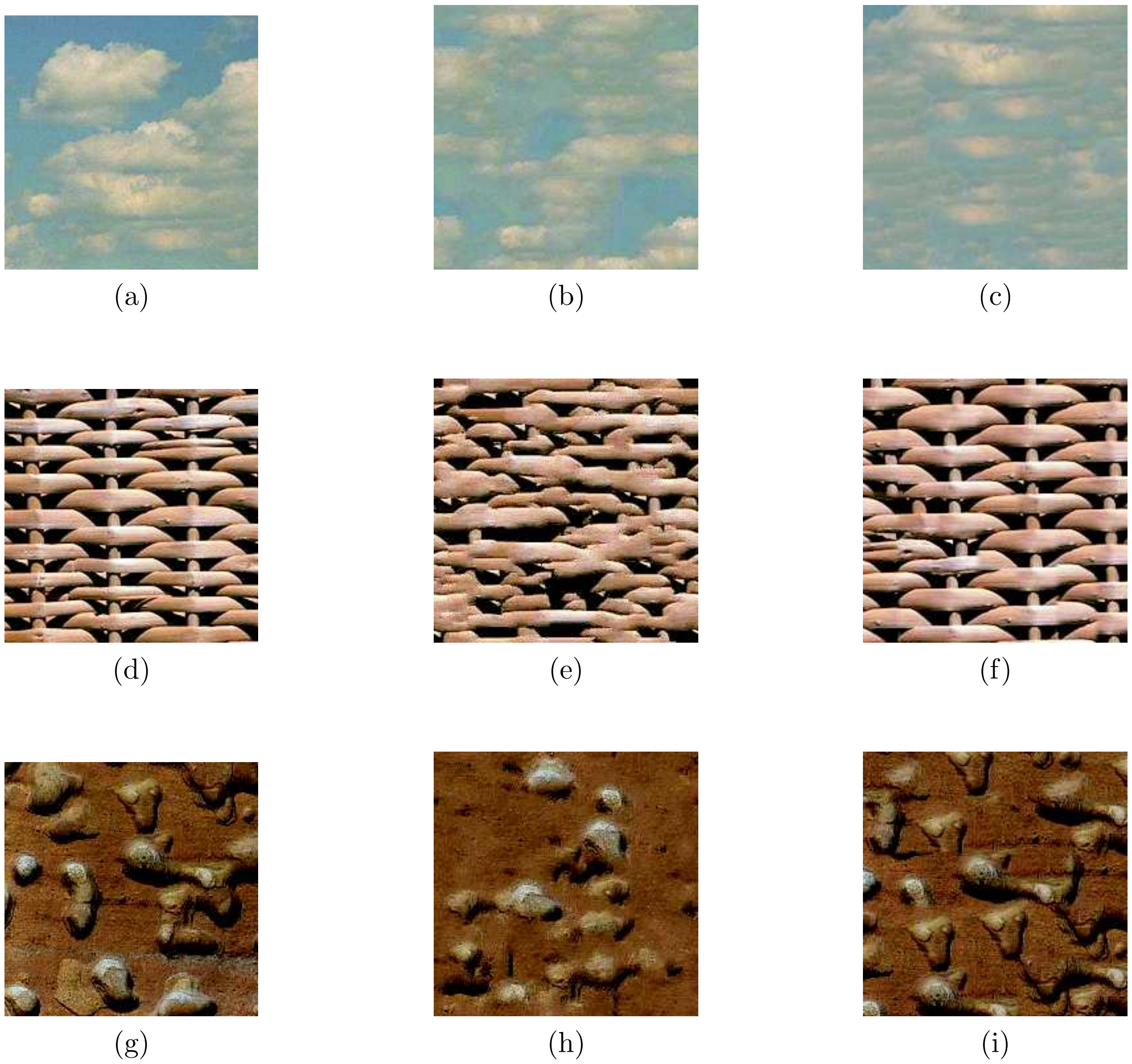
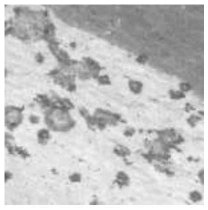
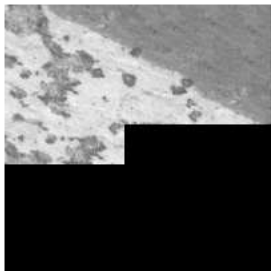
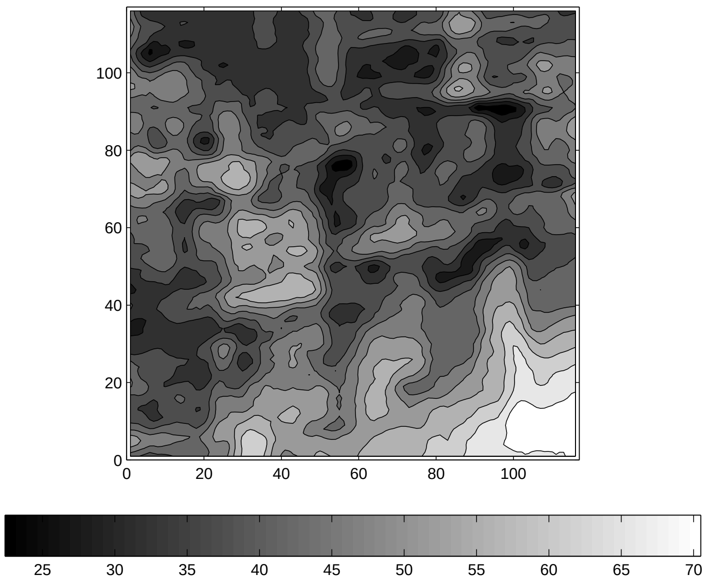


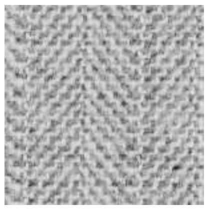
Figure 3.11: Patch based synthesis. Results (3). Size of the input images 192×192, size of the synthesized textures 200×200. In brackets time in seconds for synthesis process. Images (b), (e), (h) were downloaded from Wei's Web page [21]. (a) Input image [14]. (b) Synthesized image, pixel based synthesis [21]. (c) Synthesized image, patch based synthesis, $w_b = 20$, $w_e = 5$ (88 s). (d) Input image [14]. (e) Synthesized image, pixel based synthesis [21]. (f) Synthesized image, patch based synthesis, $w_b = 25$, $w_e = 5$ (83 s). (g) Input image [14]. (h) Synthesized image, pixel based synthesis [21]. (i) Synthesized image, patch based synthesis, $w_b = 35$, $w_e = 5$ (57 s).

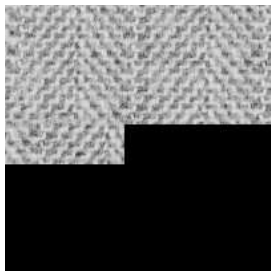

(a)

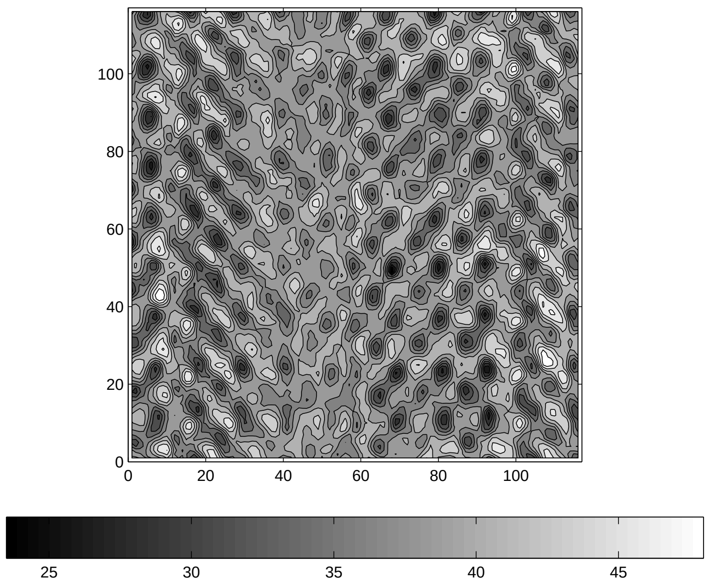

(b)



(c)


(d)


(e)



(f)

Figure 3.12: Distance values of the search for the next block, which is synthesized in (b) resp. (e) from (a) resp. (d). Boundary zones, in which no valid blocks can be found (Section 3.2.1) are not displayed in (c) and (f). (a) Input texture (150×150), gray scale 8 bit/pixel. (b) Synthesized image (200×200), in progress. (c) Distance values of all valid blocks ($w_b = 25, w_e = 5$) of the input texture (a). (d) Input texture [3] (150×150), gray scale 8 bit/pixel. (e) Synthesized image (200×200), in progress. (f) Distance values of all valid blocks ($w_b = 25, w_e = 5$) of the input texture (d).

stochastic, natural textures with bigger structures inside (Figures 3.9 (a), (d), (g)) subjectively better results in less time. But there still are points to advance the algorithm.

Chapter 4

Improvements

Despite of an improvement of the visual fidelity and computational cost in comparison to pixel based methods, still remain possibilities to advance the results. In the following chapter firstly possibilities to maximize the visual fidelity, and secondly a possibility to minimize the computational cost for the synthesis of highly stochastic textures are introduced.

4.1 Application of Isometries

One of the critics in Section 3.5 was the moderate variability in the output image, which could be improved. Nevertheless we hold on the block based sampling, without making the number n of selected blocks too big, which could introduce errors caused by a bad matching of the synthesized blocks to the image. We extend the variability by increasing effectively the number of available blocks in I_{sample}, without making I_{sample} larger.

In former approaches, blocks are sampled without modifications from the input image. But imagine an arc in the source image, running from the top to the right edge. The algorithm starts at a certain position at the left boundary of the output texture with the most horizontal part of this arc. A continuation to the top is impossible, because of the processing in raster scan order. A horizontal continuation to the right would signify a steady usage of the same part of the arc, and cause visual infidelity, because of visible repetitions (Figure 4.1 (b)). Desirable in terms of an increased stochastic variability would be instead another continuation of the arc, either an arc to the down, or another characteristic. This is possible by an application of transformations either to the input sample I_{sample} or to single blocks $B \subset I_{\text{sample}}$. Last but not least exactly these modifications of sampled texture blocks are needed for the synthesis of segmented textures (Section 5.3).

4.1.1 Approach

In the following various transformations of single square blocks B are introduced. We call these transformations with reference to [9], p. 20, isometries. Let B be a block of $N \times N$ pixels, and $B_{i,j} \in B$ be a single pixel of this block, with $i, j \in \{0, 1, \ldots, N-1\}$. So the following isometries are defined:

0. Identity:

$$I_0(B_{i,j}) = B_{i,j}.$$

1. Orthogonal reflection about mid-vertical axis ($j = (N-1)/2$) of block:
$$I_1(B_{i,j}) = B_{i,N-1-j}.$$
2. Orthogonal reflection about mid-horizontal axis ($i = (N-1)/2$) of block:
$$I_2(B_{i,j}) = B_{N-1-i,j}.$$
3. Orthogonal reflection about first diagonal ($i = j$) of block:
$$I_3(B_{i,j}) = B_{j,i}.$$
4. Orthogonal reflection about second diagonal ($i = N-1-j$) of block:
$$I_4(B_{i,j}) = B_{N-1-j,N-1-i}.$$
5. Rotation around center of block, through +90°:
$$I_5(B_{i,j}) = B_{j,N-1-i}.$$
6. Rotation around center of block, through +180°:
$$I_6(B_{i,j}) = B_{N-1-i,N-1-j.}$$
7. Rotation around center of block, through -90°:
$$I_7(B_{i,j}) = B_{N-1-i,j}.$$

Now the above introduced isometries are applied to all blocks $B \subset I_{\text{sample}}$. The transformed blocks B are considered for patch based sampling.

The application is done by splitting the isometries into three categories, which can be applied alone or together to all blocks B:

Category 0 Contains the isometry 0 (equivalent to processing in Chapter 3).

Category 1 Contains the isometries 0, 1, 2, 6. Only transformations, which change the block character approximately about 180° are applied.

Category 2 Contains the isometries 0, 3, 4, 5, 7. Also transformations, which change the block character approximately about 90° and reflections about diagonals are applied.

This split is done with respect to the increasing computational effort and to the application area. The increasing of the computational cost consists not only in calculating a higher number of distance values, because of effectively more available blocks, but also in application of the isometries to the blocks. Further on, a category with no additional isometries, and categories with isometries, which change the block character through (approximately) 180°(e.g. to mirror a semi-circle) and 90°(e.g. to rotate a quarter-circle) are regarded as reasonable.

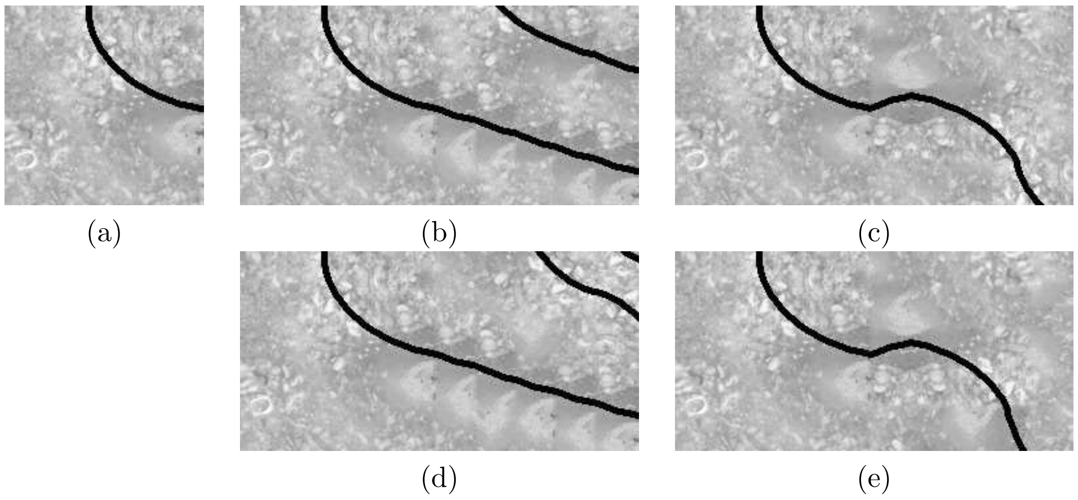
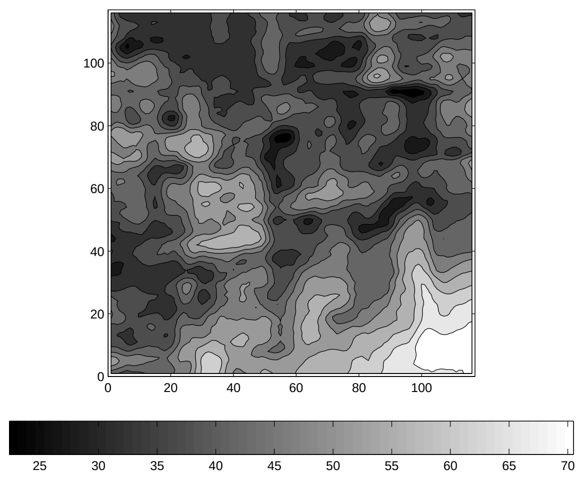


Figure 4.1: Patch based synthesis with application of isometries. Parameters: $w_b = 25$, $w_e = 5$, $n = 2$. The synthesized pictures (300×150) were created with the input image (150×150) as initialization (left), which is continued with patch based sampling to the right. In brackets time in seconds. (a) Input image, gray scale 8 bit/pixel. (b) Synthesized image, application of isometries of the category 0 (previous processing) (9 s). (c) Synthesized image, application of isometries of the category 1 (23 s). (d) Synthesized image, application of isometries of the category 2 (30 s). (e) Synthesized image, application of isometries of the categories 1 and 2 (45 s).

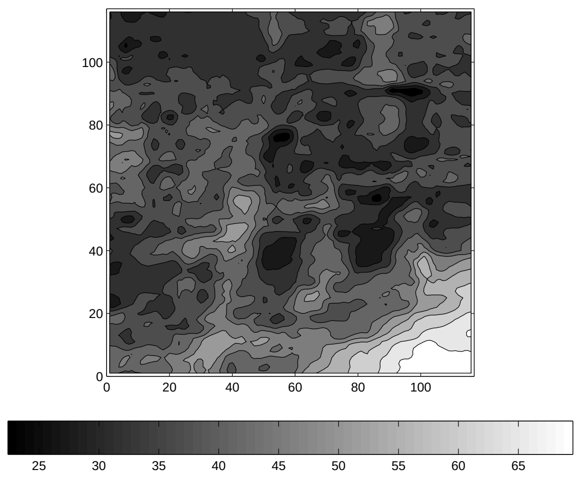
4.1.2 Results

The introduction of isometries leads to an improved stochastic variability of the sampled blocks. With additional isometries an effectively larger number of blocks is available for the synthesis process, because of block transformations, without damaging the visual quality of the blocks. In general, smaller distance values d can be obtained for many blocks B by the additional application of isometries (Figure 4.2). For this reason other blocks could match the distance tolerance $d_{\max}$ and taken into account for sampling. This enables a higher variability of the blocks sampled to I, and therewith a less deterministic continuation of already sampled blocks is possible.

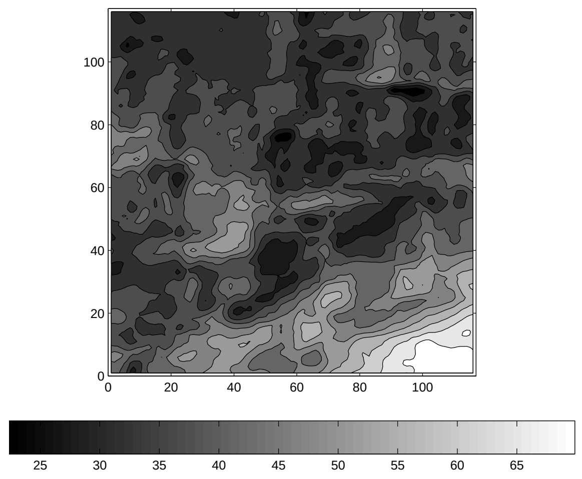
As can be seen in the Figures 4.1 and 4.3, an application of additional isometries to the blocks *can* optimize the visual fidelity of the output image - whenever an improved stochastic variability is desired. In Figure 4.3 (a) - (d) can be seen, how different the synthesized results can be, in regard to the handling of the dark zone. Without additional isometries (category 0), this dark zone is likely continued endlessly. With isometries of the categories 1 and 2 another course is possible. Though it is visible in Figure 4.3 (c) - (e) that repetitions of certain blocks can not always be avoided, but they are less visible, because of previous transformations. But also Figure 4.3 (f) - (j) shows that for highly deterministic textures the application of isometries can even lead to worse results. In this case the Herringbone Wave is not always continued correctly, and the results appear worse than without application of additional isometries (Figure 4.3 (g)). For these cases a higher variability of the output image is normally not desired. So we see that the application of isometries always depends on the application area.



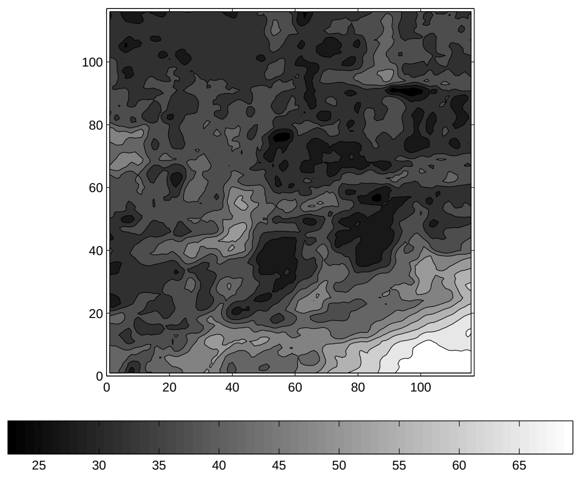
(a)



(b)



(c)



(d)

Figure 4.2: Distance values of the search for the next block (continuation from Figure 3.12 (a) - (c)). For each block the noted isometries are applied. The best result is presented in the graphic. (a) Distance values without additional isometries (category 0). (b) Distance values with isometries category 1. (c) Distance values with isometries category 2. (d) Distance values with isometries category 1 and 2.

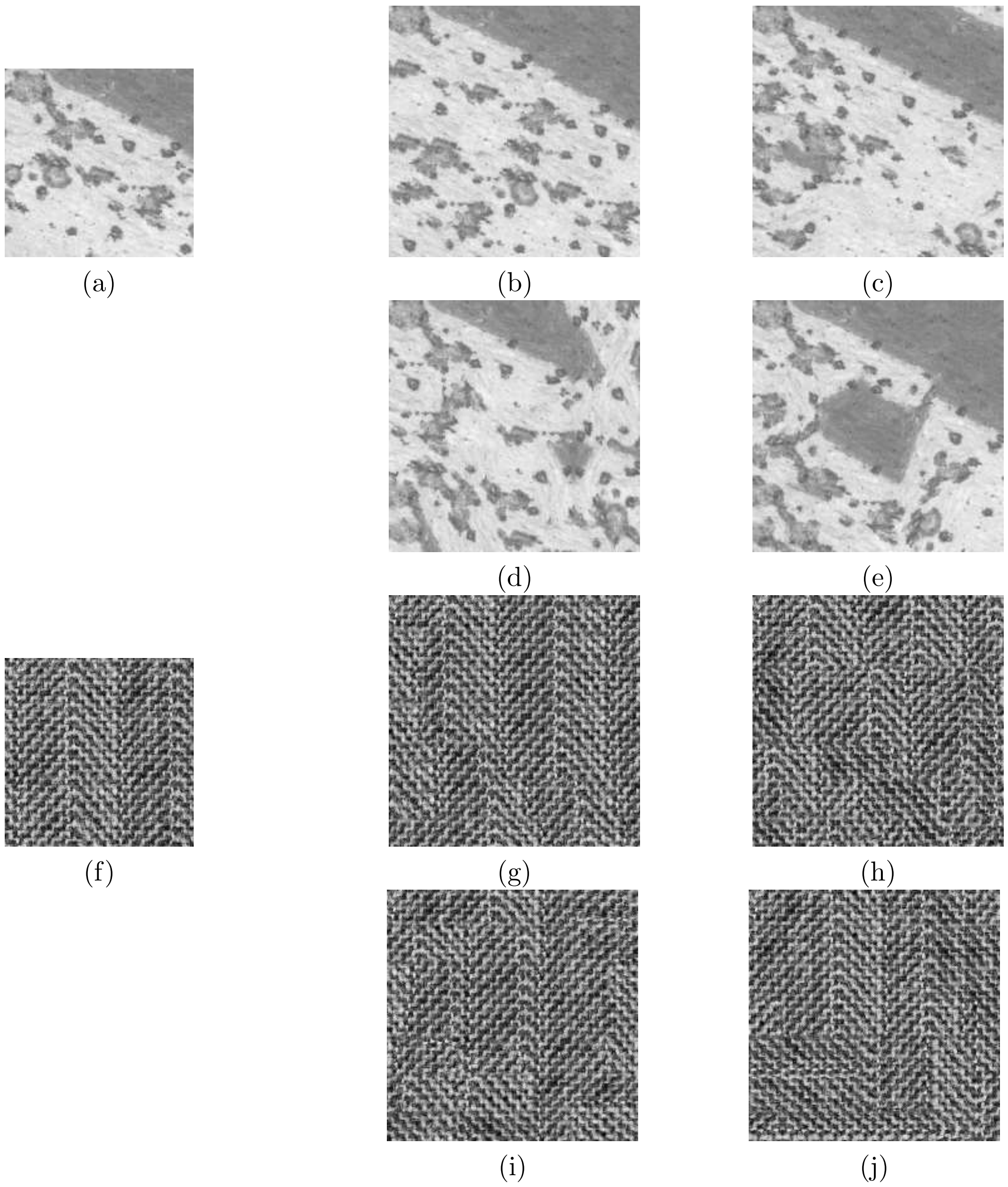


Figure 4.3: Patch based synthesis with application of additional isometries. Results. For all synthesized images the parameters $w_b = 25$, $w_e = 5$, $n = 5$ are used. All synthesized images are initialized with the same, deterministic block B, to enable a better comparison. Size of the input textures 150×150, size of the synthesized images 200×200. In brackets time in seconds. (a) Input texture. (b) Synthesized image, application of isometry category 0 (17 s). (c) Synthesized image, application of isometry category 1 (46 s). (d) Synthesized image, application of isometry category 2 (57 s). (e) Synthesized image, application of isometry categories 1 and 2 (83 s). (f) Input texture [3]. (g) Synthesized image, application of isometry category 0 (17 s) (h) Synthesized image, application of isometry category 1 (43 s). (i) Synthesized image, application of isometry category 2 (57 s). (j) Synthesized image, application of isometry categories 1 and 2 (84 s).

As disadvantages have to be seen that the selection, if isometries should be applied, depends on the characteristic of the input sample. Until today this estimation is done by the user in a subjective manner. Further on the application of isometries entails an increasing of the computational cost. First, the computational cost is increased by applying the transformations to the blocks. But this only has to be calculated once, and can be done before the synthesis process. Second, the computational cost is increased by calculating more distance values d, which has to be done for each synthesized block. So roughly estimated - neglecting the increased effort by applying the isometries to the blocks - the computational cost is multiplied by the number of applied isometries.

Concluding it can be said that an application of isometries is only recommended for highly stochastic textures, where also a highly variable output texture is desired. Because of an increasing of the computational cost and possible undesired transformations, the application of isometries should be restricted to all necessary isometries.

4.2 Prevention of Repetitions

As further mentioned in Section 3.5, patch based sampling leads from time to time to annoying, visible repetitions of certain image parts. So we are further looking for a possibility to prevent repetitions as far as possible. Because of the restricted size of the input sample I_{sample}, repetitions can not be avoided, when the size of the synthesized texture exceeds the size of the input sample. A first approach would be an equal distribution of all pixels from taken the input sample to the output texture. In respect of the quality of the synthesized texture, a strict equal distribution could lead to worse results. An approach with a consideration of the block neighborhoods suggests itself.

4.2.1 Approach

In general we follow the approach, to modify the block distance calculation. We do this by introducing an additional block *weight*, depending on the number of repetitions of the whole block or of parts of it. Different modes of marking the blocks can be imagined. One way could be, to mark each single pixel of the sampled block as used. The block weight would have to be calculated by an in general weighted sum of the pixel marks of the analyzed block. Although this approach would be very accurate, it suffers from the high introduced computational cost because of the sum calculation for each analyzed block.

Another approach is implemented by us. Let c_i be a counter for each block $B_i \subset I_{\text{sample}}$. In the beginning all blocks B_i are marked as unused, meaning $c_i = 0$. For each usage of a block B_i, this block and all blocks which are located within the spatial distance of $w_b + 2w_e - 1$ pixels from B_i are marked as used (Figure 4.4). This is done by increasing c_i by one. The consideration of neighbored blocks is done, because the in the distance of $w_b + 2w_e - 1$ pixels surrounding blocks also contain pixels of the actually sampled block B_i.

In the following the number of marks, represented by c_i of each block B_i, is considered in the further synthesis process. We do this by introducing a new distance $d_{\text{repetition},i} = d_i + c_i d_i$, where d_i is the in Section 3.3.3 introduced distance. So a higher distance to the block B_i is assigned, whenever this block or parts of it could be sampled repeatedly. But also the block B_i is in the case of repetition not automatically pushed out of range and all other blocks given a preference, even if they have a very bad matching and therefore a very high distance d_i. So further a block, which could be repeated once, with a very small distance is preferred to a

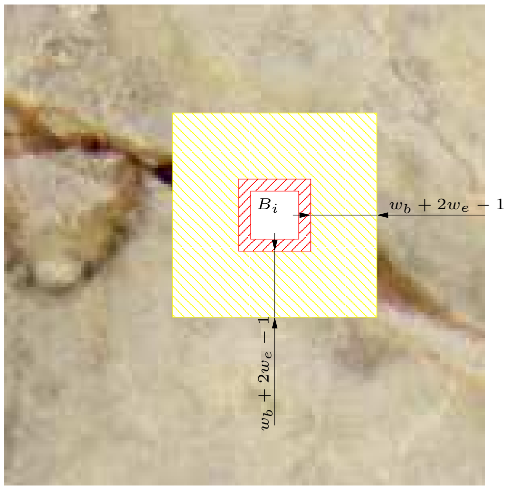


Figure 4.4: Prevention of repetitions. All blocks within the distance $w_b + 2w_e - 1$ from B_i from the sampled block B are marked as used.

block, whose distance is very large. This is done with respect to the high visible corruption, that a worse block could cause. Always a high visual fidelity with repetitions is preferred to a strict equal distribution.

4.2.2 Results

As in Figure 4.5 can be seen, the application of the above presented repetition prevention algorithm can produce good results. Figure 4.5 (c) and (f) and (h) show significantly that the synthesized image has less noticeable repetitions than the result produced without that prevention. Very deterministic textures are synthesized without bigger problems, although the result is slightly worse than without the application (Figure 4.6 (b), (c)). However application to such textures is normally senseless, because very deterministic textures normally consist in repeating regularly certain texels. The algorithm fails for not uniform input samples, as Figure 4.6 (f) shows. Problems in the transition between the dark and bright part can be seen. The transition should be continued with dark blocks, but these are already repeated too often. A solution for this problem could be found in a segmentation of the textures, and in a further application of the algorithm to the single segments. As also can be seen in Figure 4.6 (i), the preventions fails for very small input samples with striking elements. Although Figure 4.6 (h) contains a lot of repetitions, these are subjectively less significant than in Figure 4.6 (i). Cause for this is that the in (h) repeated blocks belong to the very smooth image area. The visual fidelity of the image (i) is worse, because of the multiple repetition of a striking element. The computational effort of the algorithm is imperceptible in comparison to the distance calculations during the search process, as the measured values show.

As result can be drawn that this algorithm for prevention of repetitions works well for large, uniform and highly stochastic input samples. As above showed, it fails for very small and for not uniform input samples. An application to highly deterministic textures appears senseless. In respect to the computational cost it is neutral compared to the cost of the search process. However there are many possibilities to improve the algorithm. As already above mentioned, for not uniform textures a segmentation of the textures and a following

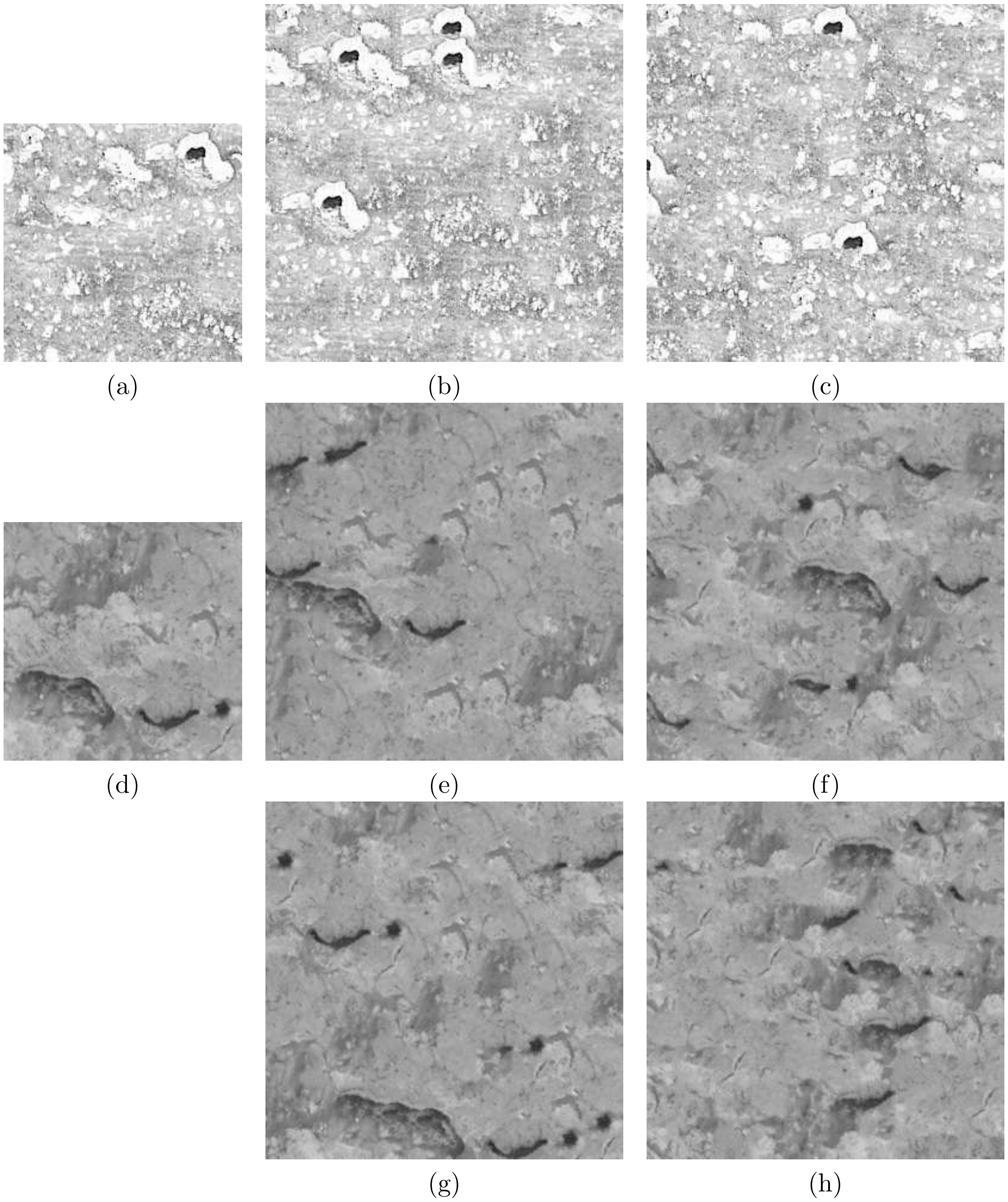
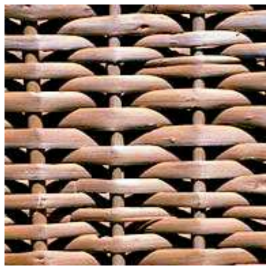
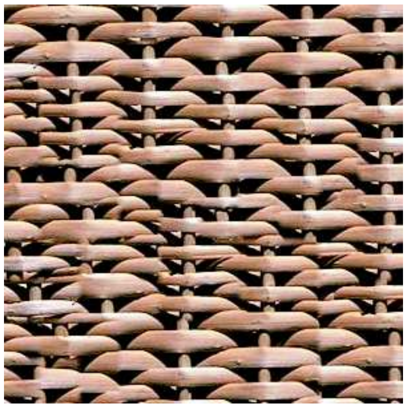
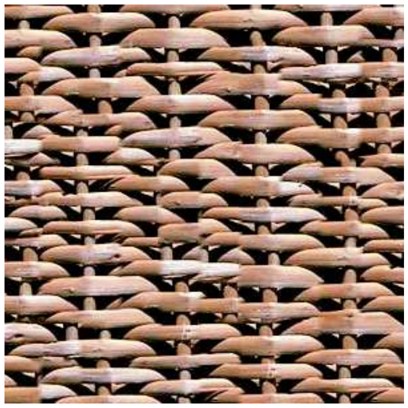
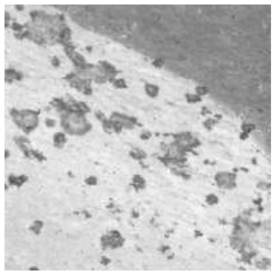


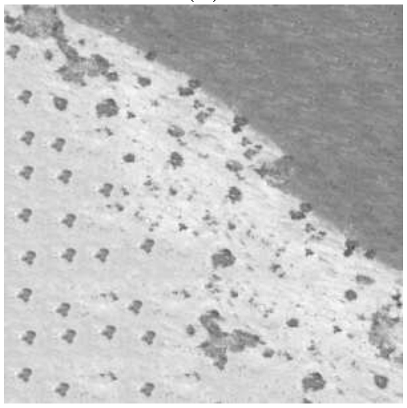
Figure 4.5: Prevention of repetitions. Results (1). Size of the input images 200 × 200. Size of the synthesized images 300 × 300. For demonstration purpose, synthesized images are initialized identically. In brackets synthesis time in seconds. (a) Input image, gray scale 8 bit/pixel. (b) Synthesized image, $w_b = 25, w_e = 5, n = 1$. No prevention of repetitions applied (73 s). (c) Synthesized image, $w_b = 25, w_e = 5, n = 1$. Prevention of repetitions applied (73 s). (d) Input image, gray scale 8 bit/pixel. (e) Synthesized image, $w_b = 25, w_e = 5, n = 1$. No prevention of repetitions applied (72 s). (f) Synthesized image, $w_b = 25, w_e = 5, n = 1$. Prevention of repetitions applied (73 s). (g) Synthesized image, $w_b = 25, w_e = 5, n = 5$. No prevention of repetitions applied (73 s). (h) Synthesized image, $w_b = 25, w_e = 5, n = 5$. Prevention of repetitions applied (73 s).

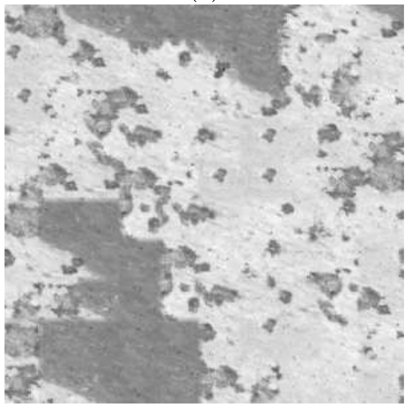

(a)

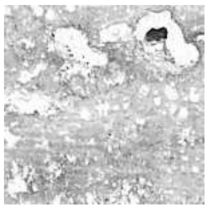

(b)

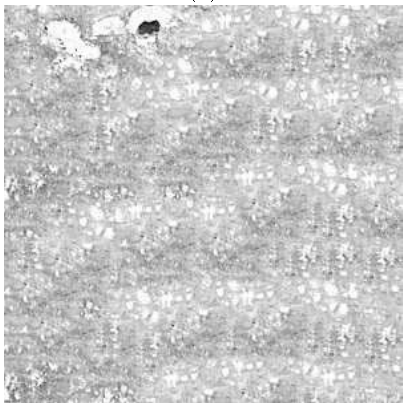

(c)


(d)


(e)


(f)


(g)


(h)

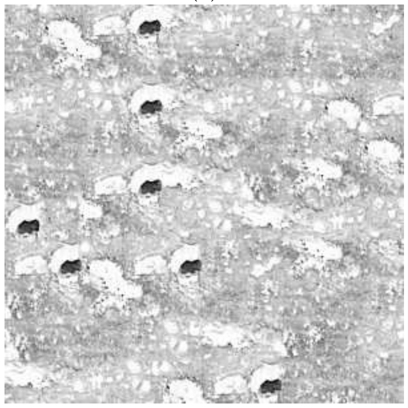

(i)

Figure 4.6: Prevention of repetitions. Results (2). Size of the input images (a) and (d) 200×200, (g) 150×150. Size of the synthesized images 300×300. For demonstration purpose, the synthesized images are initialized identically. In brackets synthesis time in seconds. (a) Input image. (b) Synthesized image, $w_b = 25, w_e = 5, n = 1$. No prevention of repetitions applied (183 s). (c) Synthesized image, $w_b = 25, w_e = 5, n = 1$. Prevention of repetitions applied (182 s). (d) Input image, gray scale 8 bit/pixel. (e) Synthesized image, $w_b = 25, w_e = 5, n = 1$. No prevention of repetitions applied (72 s). (f) Synthesized image, $w_b = 25, w_e = 5, n = 1$. Prevention of repetitions applied (73 s). (g) Input image, gray scale 8 bit/pixel. (h) Synthesized image, $w_b = 25, w_e = 5, n = 1$. No prevention of repetitions applied (34 s). (i) Synthesized image, $w_b = 25, w_e = 5, n = 1$. Prevention of repetitions applied (34 s).

application to the segments could be imagined. Problem further is that the algorithm does not take in account the relative position of repeated blocks to each other. The algorithm is dependent on the number of sampled blocks between possible repetitions. Because of this and a processing in raster scan order, repetitions in horizontal direction are more improbable than in vertical direction. So it still could be, if all blocks are marked again between the processing of a certain block and the block just below, that this block is repeated directly below. This could be improved by regarding the local distance between possibly repeated blocks.

4.3 Application of a Multiresolution Pyramid

One of the main critics to the existing algorithm is the high computational cost. As can be seen in Section 3.4, the computational effort is highly dependent on the sizes of input sample and neighborhood. An acceleration of the algorithm could be reached, if these parameters can be reduced. One approach, to do this, is a processing in various resolutions, as already proposed by [13], [20] and [22].

4.3.1 Approach

Multiresolution analysis of images is already highly common. We do this by applying a multiresolution pyramid (MRP) to the input sample I_{sample} and the already processed part of the output texture I. The pyramid has one base ($l = 0$) and two reduced ($l = 1, 2$) levels l (Figure 4.7 (a)). Each reduced level l contains the filtered and with the factor 2 subsampled image of the lower level $l - 1$. Filtering is done with a mean filter with a mask of 3×3. The filtering and subsampling, firstly to level $l = 1$ and secondly to level $l = 2$, are applied to the input sample I_{sample} and the already processed output texture I to obtain $I_{\text{sample,level l}}$ and $I_{\text{level l}}$. Afterwards, the patch based search algorithm, as presented in Chapter 3, is applied to level $l = 2$ of the MRP. Note that the sizes of the patch R and neighborhood δR have to be adapted to $\lfloor w_b/2^l \rfloor$ and $\lfloor w_e/2^l \rfloor$. After having chosen a block $B_{\text{level l}}$, which matches the criterion Ω', this block is projected onto the next lower level $l - 1 = 1$ to receive $B_{\text{level l}-1}$. An area $A \subset I_{\text{sample,level l}-1}$ is cut of $I_{\text{sample,level l}-1}$ with the size of $(2(\lfloor w_b/2^l \rfloor + 2\lfloor w_e/2^l \rfloor)) \times (2(\lfloor w_b/2^l \rfloor + 2\lfloor w_e/2^l \rfloor))$ centered around $B_{\text{level l}-1}$ (Figure 4.7 (b)). Within this area A the search process is continued by another application of the block search algorithm. This is analogically repeated until a block $B_{\text{level 0}}$ from level 0 is chosen. Finally this block $B_{\text{level 0}}$ is sampled to the output image I.

4.3.2 Results

The number of difference operations/block at level 2 is

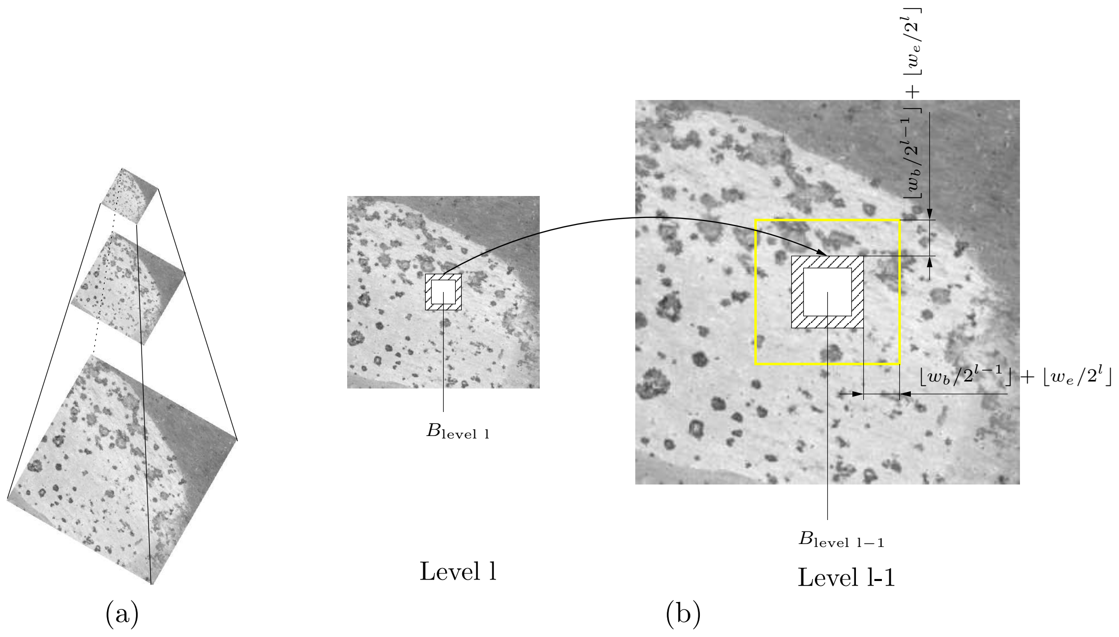
$$n_{\text{op,level 2}} \approx (\lfloor w_{\text{sample}}/4 \rfloor - w_{b,2} - 2w_{e,2} + 1)(\lfloor h_{\text{sample}}/4 \rfloor - w_{b,2} - 2w_{e,2} + 1) \cdot 2w_{e,2}(w_{b,2} + 2w_{e,2}),$$

where $w_{b,l} = \lfloor w_b/2^l \rfloor$ and $w_{e,l} = \lfloor w_e/2^l \rfloor$. At level 1 are

$$n_{\text{op,level 1}} \approx ((2w_{b,1} + 4w_{e,1} - w_{b,1} - 2w_{e,1} + 1)((2w_{b,1} + 4w_{e,1}) - w_{b,1} - 2w_{e,1} + 1) \cdot 2w_{e,1}(w_{b,1} + 2w_{e,1})$$

operations necessary, and finally remain for level 0

$$n_{\text{op,level 0}} \approx ((2w_{b,0} + 4w_{e,0}) - w_{b,0} - 2w_{e,0} + 1)((2w_{b,0} + 4w_{e,0}) - w_{b,0} - 2w_{e,0} + 1) \cdot 2w_{e,0}(w_{b,0} + 2w_{e,0})$$



$B_{\text{level l}}$
$B_{\text{level l}-1}$
Level l
Level l-1
$\lfloor w_b/2^{l-1} \rfloor + \lfloor w_e/2^l \rfloor$
$\lfloor w_b/2^{l-1} \rfloor + \lfloor w_e/2^l \rfloor$
(a)
(b)

Figure 4.7: Multiresolution processing. (a) 3 level MRP. Image I_{sample} is represented in original resolution $I_{\text{sample, level0}}$ (300×300, level 0) and two reduced resolutions $I_{\text{sample, level1}}$ (150×150, level 1) and $I_{\text{sample, level2}}$ (75×75, level 2). (b) Block $B_{\text{level l}}$ is chosen from pyramid level l. It is projected to the next lower level l-1. A new search area A (yellow) is selected around $B_{\text{level l}-1}$.

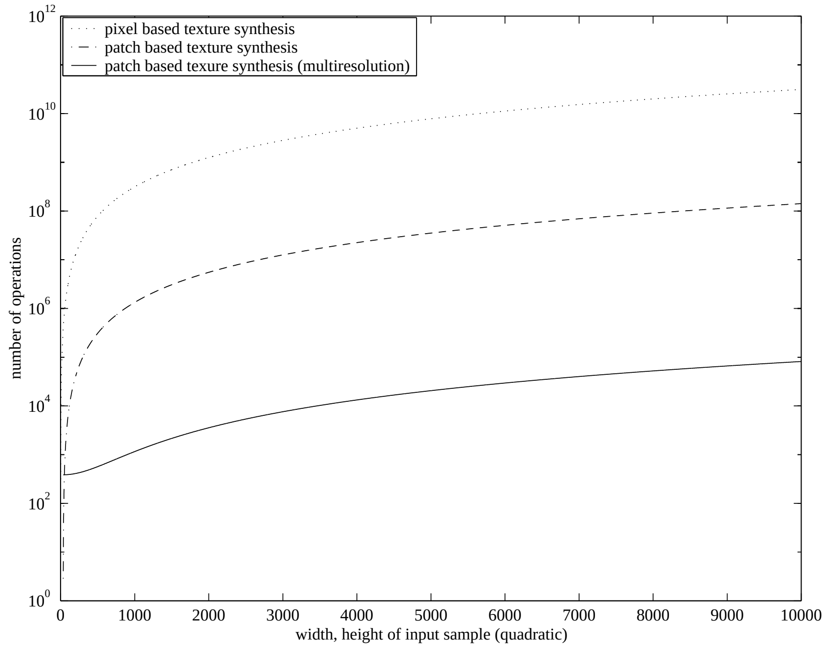


Figure 4.8: Number of difference operations/pixel depending on the size of I_{sample} (assuming a quadratic I_{sample}). Pixel based synthesis: $w_e = 25$, $h_e = 13$. Patch based synthesis: $w_b = 25$, $w_e = 5$. Patch based synthesis (MRP): $w_b = 25$, $w_e = 5$, application of a 3-level MRP.

operations. We obtain

$$n_{\text{op,pyramid}} \approx \frac{(n_{\text{op,level 2}} + n_{\text{op,level 1}} + n_{\text{op,level 0}})}{(w_b + 2w_e)(w_b + 2w_e)}$$

difference operations for the processing of one pixel. As in former approaches, a precise edge handling has been neglected. Also neglected in above calculations was the effort for filtering and subsampling. We regard this cost as small in comparison to the cost of the search process. Filtering and subsampling of I_{sample} has only to be done once. From the processed I only the processed parts of the neighborhood have to be reduced in their resolution. So the number of operations/synthesized pixel can be clearly reduced by the application of a multiresolution analysis, as Figure 4.8 shows. A clear gain in effectiveness compared to patch based texture synthesis without MRP and to pixel based texture synthesis could be made.

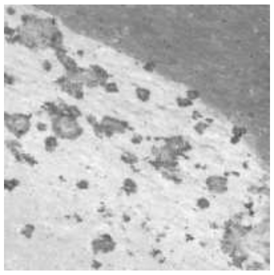
Despite of the gain in computational cost, the algorithm produces good output results for highly stochastic textures, as Figure 4.9 shows. The quality of the synthesized texture is equal for highly stochastic textures in comparison to patch based synthesis without MRP. Sometimes a trend to very smooth image areas can be observed (Figure 4.3 (i)). For highly deterministic textures with many similar blocks the algorithm with MRP produces worse results than without MRP (Figure 4.10 (c)). It can be seen, that the block transitions do not match exactly. Because of the high similarity of the input texture can be assumed that the algorithm already chooses a wrong block $B_{\text{level 2}}$ at the highest pyramid level $l = 2$. At this point the algorithm could be advanced, e.g. by application of a quad tree pyramid ([13], p. 15 seqq.) instead of the here introduced MRP. Also an application of other filter (e.g. Gaussian kernel) could be considered.

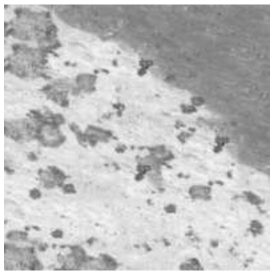
Concluding it can be said, that the application of a MRP leads - for highly stochastic textures - to a massive shortage of synthesis time and produces images with the same quality. Only for highly deterministic textures with many similar blocks a slight reduction of the quality can be observed. Although of great improvements in the computational efforts, the algorithm still depends highly on the size of the input texture.

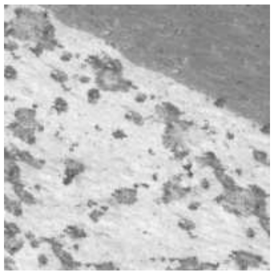
4.4 Summary

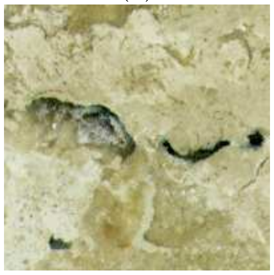
The in this chapter introduced methods advance the in Chapter 3 presented patch based texture synthesis. The introduction of isometries (Section 4.1) improves the stochastic variability of the synthesized image by application of transformations to the synthesized blocks. Unfortunately it brings along an increasing of the computational cost by a comparison of more blocks. Also it could help to reduce visible block boundaries by a better transition of structures. Visible repetitions could be prevented with the algorithm for prevention of repetitions (Section 4.2). It is in its computational cost neutral, but could lead to worse synthesized images for not uniform textures and very small input samples. Finally the application of a MRP (Section 4.3) leads to a massive improvement in computational cost, and for stochastic textures no loss in quality can be observed. But still a high dependency on the size of the input sample remains. Still not sufficiently solved is that as a matter of principle block bounds could become visible. This could be improved by the application of isometries, but not totally avoided. We do not think that this can be solved by a block based algorithm.

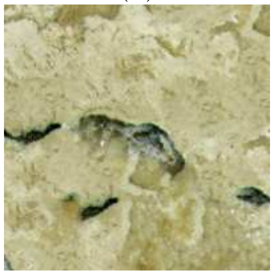
The in this chapter introduced improvements do in general not work well for all types of textures. They were introduced for the work with highly stochastic textures, and especially

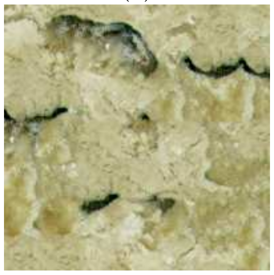

(a)

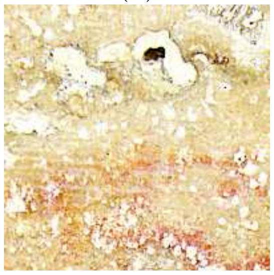

(b)

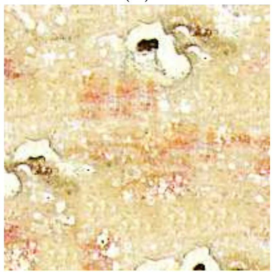

(c)

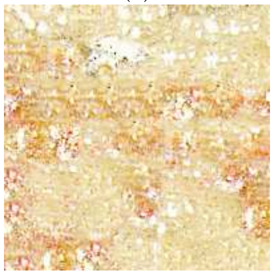

(d)

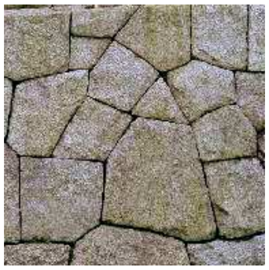

(e)

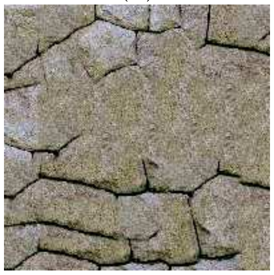

(f)


(g)


(h)


(i)


(j)


(k)

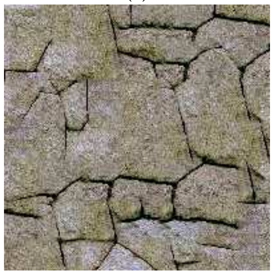

(l)

Figure 4.9: Patch based texture synthesis with application of a MRP. Results (1). Size of the input images 200×200, size of the synthesized images 200×200. For demonstration purpose, images are initialized identically. In brackets time in seconds. (a) Input image, gray level 8 bit/pixel. (b) Synthesized image, $w_b = 25$, $w_e = 5$, $n = 5$. Patch based synthesis without MRP (33.4 s). (c) Synthesized image, $w_b = 25$, $w_e = 5$, $n = 5$. Patch based synthesis with MRP (1.0 s). (d) Input image. (e) Synthesized image, $w_b = 25$, $w_e = 5$, $n = 5$. Patch based synthesis without MRP (87.1 s). (f) Synthesized image, $w_b = 25$, $w_e = 5$, $n = 5$. Patch based synthesis with MRP (2.9 s). (g) Input image. (h) Synthesized image, $w_b = 25$, $w_e = 5$, $n = 5$. Patch based synthesis without MRP (86.4 s). (i) Synthesized image, $w_b = 25$, $w_e = 5$, $n = 5$. Patch based synthesis with MRP (2.8 s). (j) Input image. (k) Synthesized image, $w_b = 25$, $w_e = 5$, $n = 5$. Patch based synthesis, without MRP (84.2 s). (l) Synthesized image, $w_b = 25$, $w_e = 5$, $n = 5$. Patch based synthesis with MRP (2.8 s).

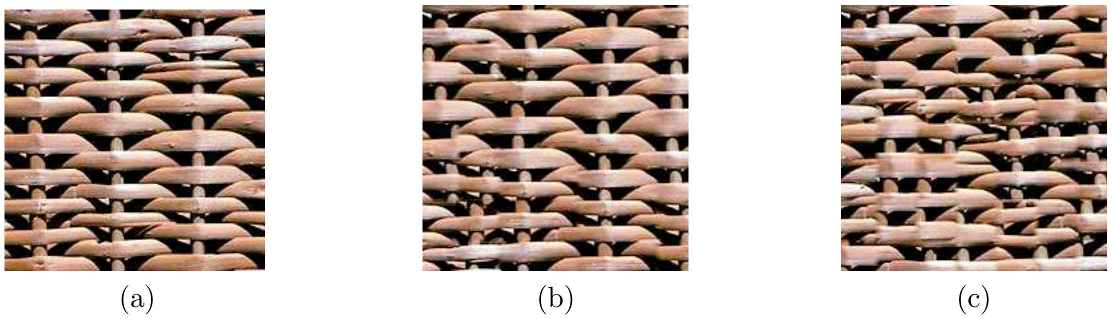


Figure 4.10: Patch based texture synthesis with application of a MRP. Results (2). Size of the input images 200 × 200, size of the synthesized images 200 × 200. For demonstration purpose, images are initialized identically. In brackets time in seconds. (a) Input image. (b) Synthesized image, $w_b = 25$, $w_e = 5$, $n = 5$. Patch based synthesis without MRP (85.6 s). (c) Synthesized image, $w_b = 25$, $w_e = 5$, $n = 5$. Patch based synthesis with MRP (2.8 s).

problems with highly deterministic textures can be seen. As already pointed out in the introduction, our efforts are mainly based in synthesizing highly stochastic textures, so no further efforts to solve these problems were done.

Chapter 5

Modifications in the Application of the Patch Based Synthesis

In the chapters from above, patch based texture synthesis was introduced and advanced. Thereby in all those efforts the initial, basic proceeding to synthesize a completely new texture from an input sample (Chapter 3) was followed. The synthesized texture I was initialized with a randomly sampled block from I_{sample}. Afterwards, a processing in raster scan order was applied. All in I_{sample} available blocks could be used for the synthesis.

But other applications can be imagined. This chapter introduces and discusses the by us made modifications to the application of the algorithm. The basic concepts of the algorithm consist further.

5.1 Initialization of the Synthesized Image with the Input Sample

In the previous approach (Section 3.2.2) the output texture I is initialized with a randomly sampled block from I_{sample}. For certain application areas a variation of this could be preferred. So it is sometimes desired, not to generate a completely new texture I, but to expand the input sample I_{sample} to a certain size. We do this by initializing the output texture I with I_{sample} (Figure 5.1 (a)) at the left, and enlarge I_{sample} with synthesized blocks to the right (Figure 5.1 (b)). For this enlargement a processing in raster scan order is applied, where the neighborhoods δR of the already sampled blocks and the zone of the width of w_e pixels at the very right of of I_{sample} serve as neighborhoods. The limitation to an expansion to the right side can easily be bypassed by a rotation of the synthesized texture after a first synthesis through 90°and a following expansion of the synthesized texture with a new synthesis.

5.2 Avoidance of Marked Pixels

For certain applications it could be desired that the synthesized image I does not contain certain image areas or single pixels of I_{sample}. We solve this problem by neglecting all blocks $B \subset I_{\text{sample}}$ for synthesis, which contain a pixel $p \in B$ of a certain, user defined color C. So the synthesis process is further applied to the set $\mathcal{B} = \{B \subset I_{\text{sample}} | p \in B \neq C\}$ instead of the whole input sample I_{sample}. Undesired regions or pixels have to be defined manually by

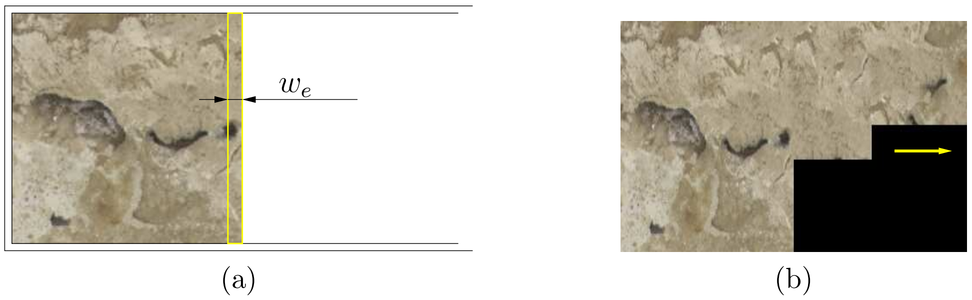


Figure 5.1: Initialization of the output texture I with I_{sample}. (a) The output texture I is initialized with I_{sample}. The yellow marked region of width w_e is used as neighborhood for the following synthesis. (b) Synthesis in progress. The image is synthesized in raster scan order, from top to bottom and from left to right.

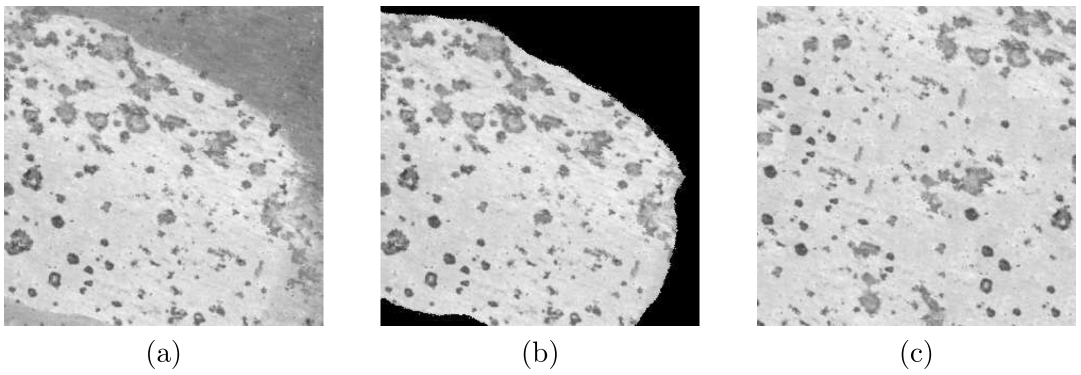


Figure 5.2: Avoidance of marked pixels. (a) Input sample (300×300), gray scale 8 bit/pixel. Dark region at the image boundary should be avoided. (b) The undesired region is marked with C = black by the user. (c) Synthesized image (300×300). Elapsed time for synthesis 243 s.

the user. This is done by painting them in the color C (Figure 5.2).

5.3 Segmentation of Input Sample and Output Texture

Consequently not only certain blocks should be avoided, but the user should be able to define the segmentation of the synthesized texture. We have implemented a way to define a segmentation of the output texture I, consisting of two segments, and synthesize I according to the defined segments.

5.3.1 Approach

We partition manually the texture sample I_{sample} and the desired synthesized texture I into two segments. This is done by using two segmentation schemes $I_{\text{sample, seg}}$ and I_{seg}, signaling the partitions of the textures. The segmentation schemes consist of a segmentation into two different regions S_1 and S_2 (Figure 5.3). The segmentation schemes must assign each pixel $p \in I_{\text{sample}}, I$ uniquely to a region S_1, S_2. Further on for each region S_1, S_2 must exist at least one block $B' \subset I_{\text{sample}}$, which can completely be assigned to this region S_1, S_2.

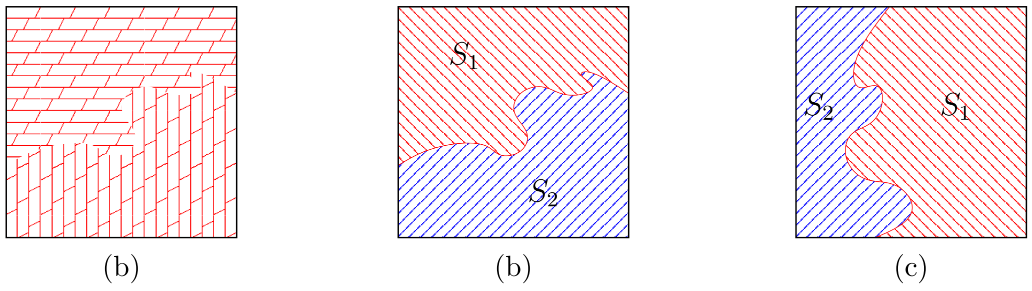


Figure 5.3: Partitioning of I_{sample} and I into two segments S_1 and S_2, using two segmentation schemes. (a) Input sample I_{sample}. The segmentation can easily be seen. (b) Segmentation scheme $I_{\text{sample, seg}}$ of I_{sample}. (c) Segmentation scheme I_{seg} of I.

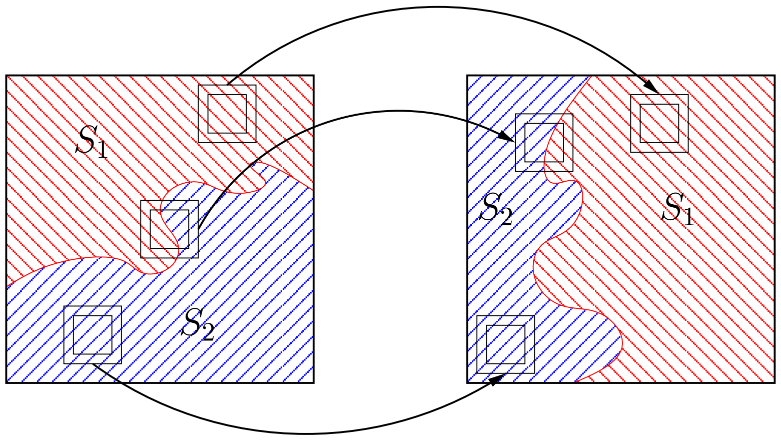


Figure 5.4: Sampling of blocks dependent on the segmentation. The regions, to which the block in I_{sample} (left) belongs, must be identical to the region, to which the block is sampled in I (right).

In the following all blocks $B' \subset I_{\text{sample}}$ and all to be sampled blocks $B \subset I$ are divided into three sets G_1, G_2, G_3. Set G_1 contains all blocks which are fully located in region S_1. Set G_2 contains all blocks which are fully located in region S_2. All other blocks, which are partially located in region S_1 and region S_2 are allocated to G_3. During the synthesis process, only blocks B' are taken in consideration for sampling, which belong to the same set as the to be sampled block B (Figure 5.4).

To enable an as exact as possible matching of the synthesized blocks to the segmentation scheme at the boundary between S_1 and S_2, a new distance d_{seg} is introduced. It considers the different segmentation characteristics of the to be sampled block B' and the scanned block B at the boundary.

$$d_{\text{seg}} = d + \alpha \tilde{d}(B, B'), \alpha = \text{const.},$$

where $d = d_{\text{max, n}}$ (Section 3.3.3) and $\tilde{d}(B, B')$ is an appropriate distance of the segmentations of B and B'.

In the following the distance $\tilde{d}(B, B')$ is defined. Let $B_{\text{seg}} \subset I_{\text{seg}}$ and $B'_{\text{seg}} \subset I_{\text{sample, seg}}$ be two blocks, containing the segmentations of B and B', in size and shape identical to the blocks B, B'. Further let $B_{\text{seg,k}} \in B_{\text{seg}}$ and $B'_{\text{seg,k}} \in B'_{\text{seg}}$ be two elements of these blocks, representing, to which region S_1, S_2 the according pixel of B and B' belongs. We define the operation $\ominus$ to the elements $B_{\text{seg,k}}$ and $B'_{\text{seg,k}}$, so that

$$B_{\text{seg,k}} \ominus B'_{\text{seg,k}} = 0 \text{ if } B_{\text{seg,k}} \equiv B'_{\text{seg,k}}$$

and

$$B_{\text{seg,k}} \ominus B'_{\text{seg,k}} = 1 \text{ if } B_{\text{seg,k}} \not\equiv B'_{\text{seg,k}}.$$

So

$$\tilde{d} = (A^{-1} \sum_{k=1}^{A} (B_{\text{seg,k}} \ominus B'_{\text{seg,k}})^2)^{1/2},$$

where A is the number of processed elements $B_{\text{seg,k}}$, $B'_{\text{seg,k}}$ of a block. Note that the distance $\tilde{d} = 0$ for all blocks B, B', which are totally located in S_1, S_2.

Moreover two different requirements could be found to synthesize blocks, which fully belong to a region S_1, S_2 or which belong to the boundary zone. For synthesizing blocks at the boundary zone it could be desired, that these blocks match as good as possible the segmentation boundary, with very less stochastic variability, whereas for blocks contained in a region S_1, S_2 a high stochastic variability could be desired. We meet these requirements by introducing two different distance tolerances $d_{\text{max, n}}$ for all blocks B in G_1 and G_2, and $d_{\text{max, n}'}$ for all blocks B in G_3. So directly and independently on the other synthesized blocks the variability of the sampled blocks at the boundary can be defined by the parameter n'.

5.3.2 Free Parameters

Some parameters remain undeclared. The parameter α regulates the influence of the exact boundary characteristic to the synthesis. Figure 5.5 shows examples for various values of α. We made good experiences with an $\alpha = [1.5; 2.5]$. In our following approaches we set therefore $\alpha = 2$. A too small α does not take sufficiently in account the boundary characteristic. As Figure 5.5 (d) and (e) show, a smooth block transition of the sampled blocks is preferred to an exact matching of the boundary, because the distance of the seams is weighted too strong. An $\alpha = 2$ produces good results. In Figure 5.5 (f) a good matching of the boundary characteristic can be seen. Problems with the boundary matching can only be observed for the block at the top of the boundary. In contrast to that in the Figures 5.5 (g), (h) the distance for the boundary characteristic is weighted too strong. A missing smooth transition of the sampled blocks can be observed, single blocks can easily be recognized. As also can be seen, for very high α the synthesis algorithm tends to produce steps at the segmentation boundary.

For the parameters n and n' we refer to Section 3.3.4. The parameter n regulates the stochastic variability of the output texture in the regions, which do not belong to the segmentation boundary. We recommend $n \approx 5$, but this value should be adapted to the desired characteristic of the synthesized texture. Whereas the parameter n' defines the variability of the segmentation boundary, this variability has to be seen critically in aspect to the visual quality of the synthesized boundary. Therefore we recommend a smaller n' that the variability of the chosen blocks does not have any negative impact to the segmentation boundary. Good results have been made with an $n' \approx 1$.

5.3.3 Results

The algorithm produces segmented textures of a good visual fidelity. Essentially for a good characteristic of the synthesized boundary between the two segments is the application of

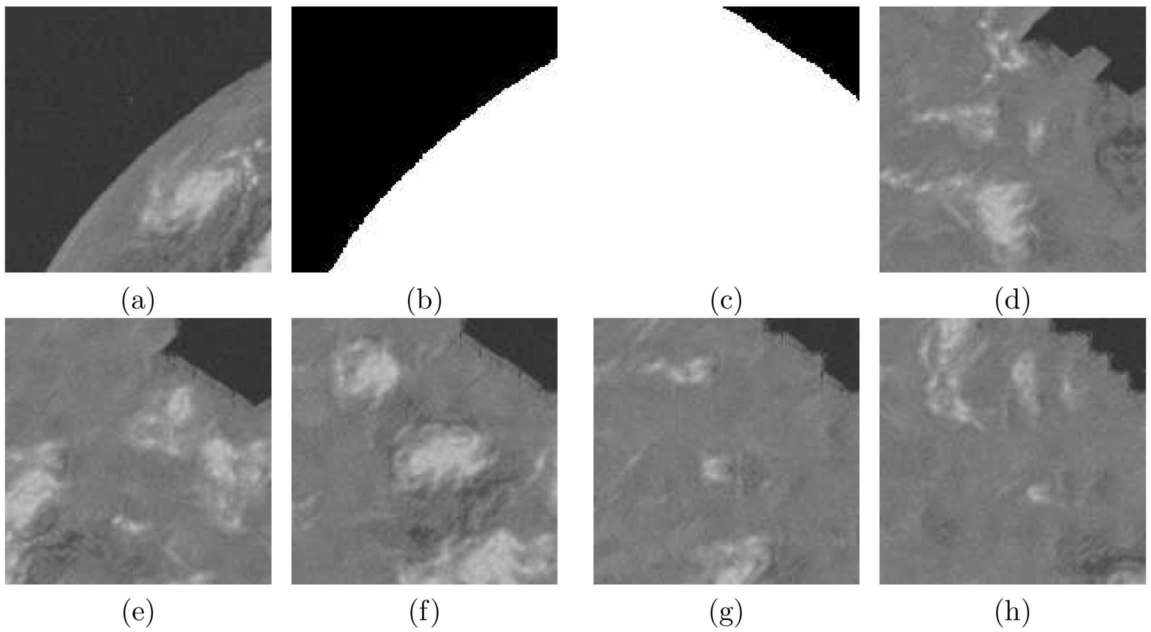


Figure 5.5: Segmentation of input sample and output texture. Variations of α. Size of the images 150×150, gray scale 8 bits/pixel. $w_b = 7$, $w_e = 4$, $n = 5$, $n' = 1$. Isometries of the categories 1 and 2 applied. In brackets time in seconds. (a) Input texture. (b) Segmentation of the input texture. (c) Segmentation of the output texture. (d) Synthesized image, $\alpha = 0.5$ (50 s). (e) Synthesized image, $\alpha = 1.0$ (51 s). (f) Synthesized image, $\alpha = 2$ (50 s). (g) Synthesized image, $\alpha = 3$ (49 s). (h) Synthesized image, $\alpha = 5$ (51 s).

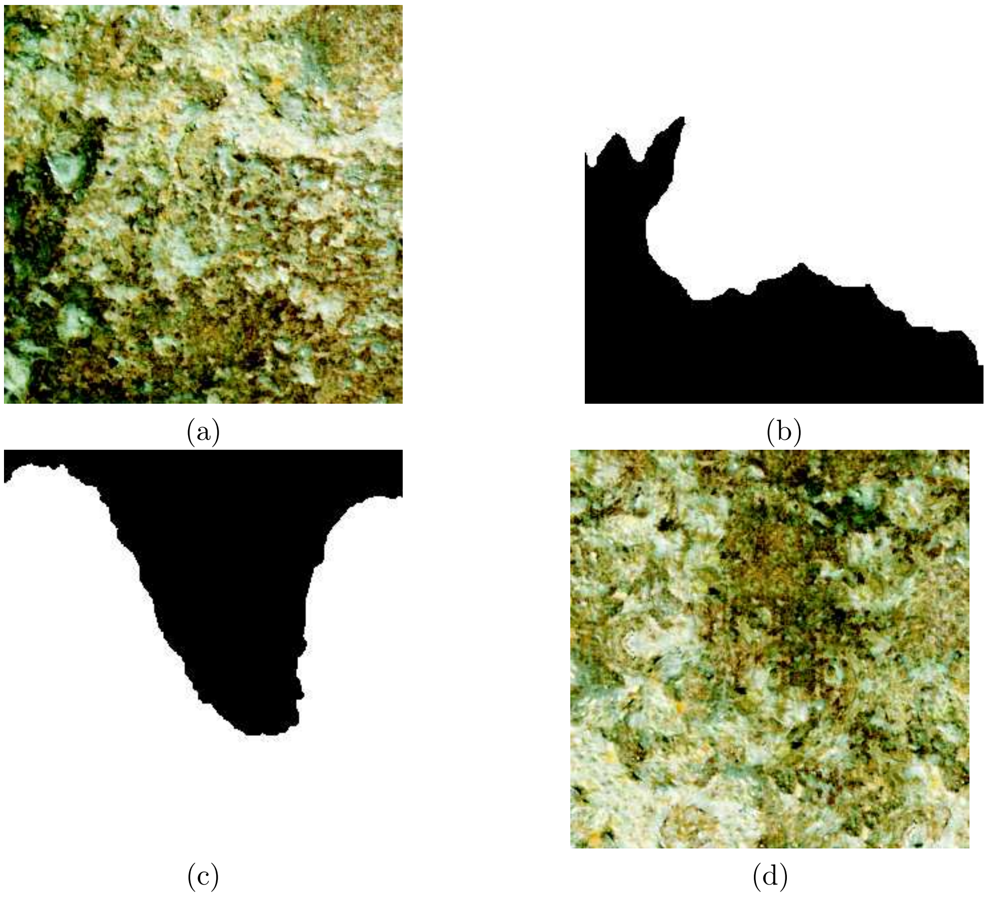


Figure 5.6: Segmentation of input sample and output texture. Results (1). Size of the images 300×300. $w_b = 25$, $w_e = 5$, $n = 5$, $n' = 1$, $\alpha = 1$. Isometries of the categories 1 and 2 applied. Application of MRP. (a) Input texture. (b) Segmentation of the input texture. (c) Segmentation of the output texture. (d) Synthesized image. Elapsed time for synthesis 35 s.

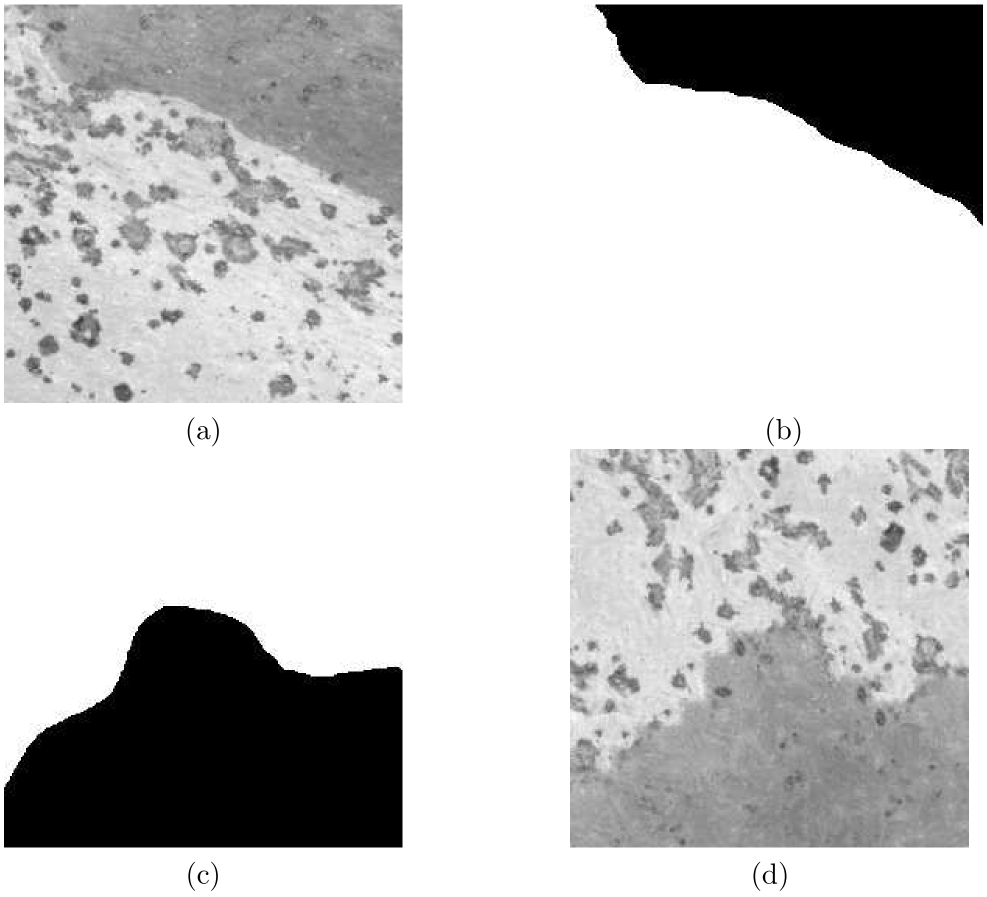


Figure 5.7: Segmentation of input sample and output texture. Results (2). Size of the images 300 × 300, gray level, 8 bit/pixel. $w_b = 10$, $w_e = 5$, $n = 5$, $n' = 2$, $\alpha = 2$. Isometries of the categories 1 and 2 applied. (a) Input texture. (b) Segmentation of the input texture. (c) Segmentation of the output texture. (d) Synthesized image. Elapsed time for synthesis 598 s.

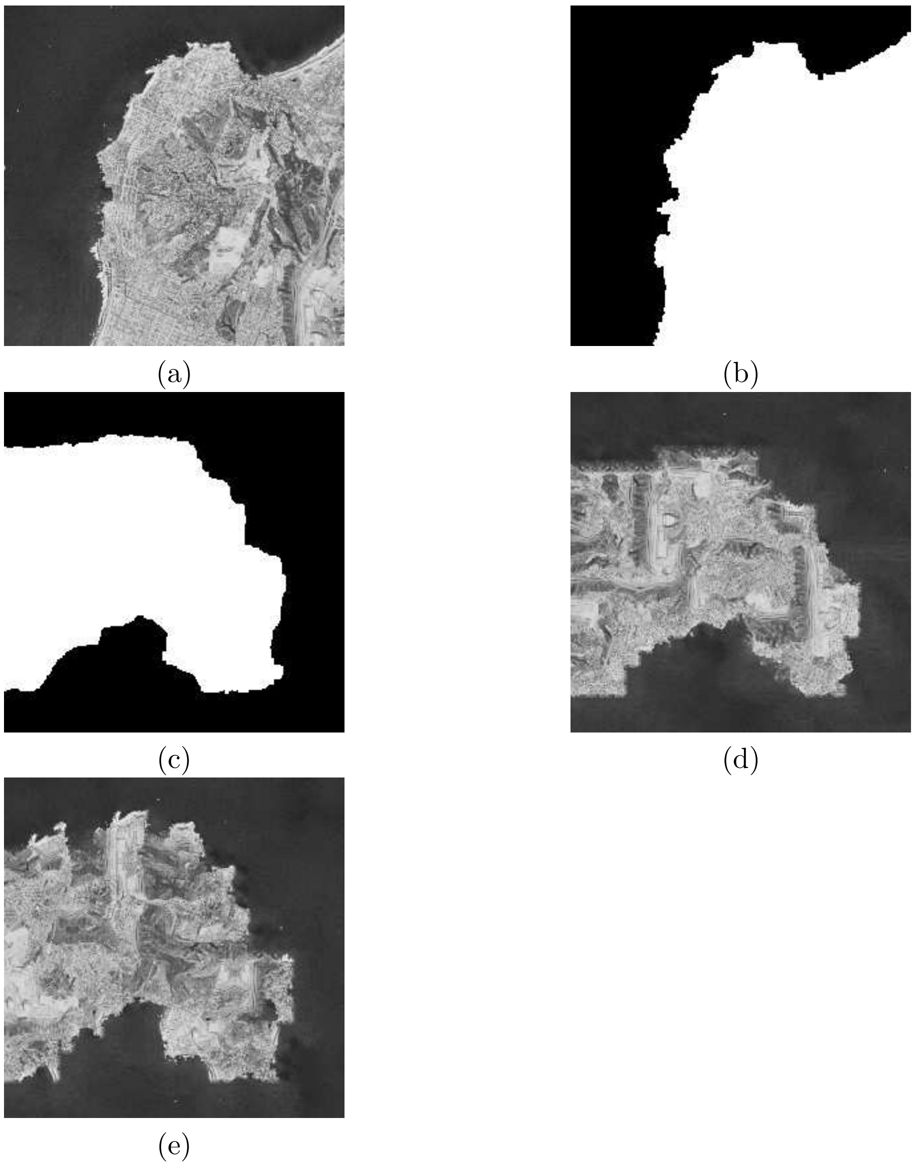


Figure 5.8: Segmentation of input sample and output texture. Results (3). Size of the images 256×256, gray scale 8 bits/pixel. $w_b = 7$, $w_e = 4$, $n = 5$, $n' = 1$. Isometries of the categories 1 and 2 applied. (a) Input texture. (b) Segmentation of the input texture. (c) Segmentation of the output texture. (d) Synthesized image, $\alpha = 1.5$ (559 s). (e) Synthesized image, $\alpha = 0$ (560 s).

isometries to the algorithm. Only with this application it is possible to synthesize segmentation boundaries of not identical characteristics as the segmentation boundaries of the input samples.

The synthesis algorithm for segmented textures was developed for highly stochastic textures with very smooth transitions between the segments. For these cases results with a high visual fidelity are produced (Figure 5.6), and this is done with application of a MRP with a low computational effort. If the transitions have a boundary with a strong contrast, good results only could be gained for these parts of the synthesized boundary, which also can be found (e.g. transformed) in the boundary of the input sample. For all other parts of the boundary the algorithm fails (Figure 5.7). E.g. a failing of the algorithm can be observed for very fine structures at the segmentation boundary (Figure 5.8). Further on for a satisfying synthesis of the segmentation boundary, the sizes of patch and neighborhood have to be chosen very small. This leads to a lack of capturing bigger texels of the input sample and of transferring them to the output texture. Also an increased computational effort is caused by this. Finally for images with a high contrast between the two segments, visual infidelities are introduced by the blending (Figures 5.5, 5.8).

The visual infidelities, introduced by blending, are a problem of the algorithm and can only be advanced by a very small neighborhood w_e. Alternatively another block arrangement method (e.g. [5]) could be considered. All other above mentioned problems have their source in the limited number of blocks at the image boundary of the input sample I_{sample}. A block based method can only sample a limited number of output characteristics from a limited number of input characteristics. One solution to avoid visual infidelities in the output texture would be an adaptation of the desired segmentation boundary to the segmentation boundary of the input sample. Whenever a strict characteristic of the output segmentation boundary is not necessary, an $\alpha = 0$ has also lead to good results. In this case, the exact boundary characteristic is not regarded, and the highest priority of the algorithm is given in generating smooth transition between the sampled blocks. Figure 5.8 (e) has subjectively a higher visual fidelity than Figure 5.8 (d).

Synthesis with very small w_b is done to obtain a good matching to the segmentation boundary. But it increases the synthesis time and it is not able to capture bigger texels. Some methods could be considered, to avoid these problems. So it could be imagined to use further on large w_b for texture synthesis of the blocks, which are not part of the segmentation boundary. At the boundary the blocks could be scaled down and synthesis of the boundary zone could be done with the down scaled blocks. Another approach could be, to combine pixel and patch based synthesis. So the image parts, which do not belong to the boundary, firstly could be synthesized with patch based texture synthesis. The parts, which belong to the segmentation boundary, then could be synthesized with the pixel based synthesis method (Chapter 2). So an exact matching of the segmentation boundary could be reached.

Chapter 6

Conclusions and Perspective

What conclusions can be drawn from this work? It was demonstrated that patch based texture synthesis is an appropriate method to synthesize textures. Good results have not only been obtained for highly stochastic textures, but also for very deterministic ones. Texel placement of the input sample can easily be caught by the patch, whereas the seam transfers the constraints of synthesized blocks. So a high visual fidelity of the texture can be reached. The computational effort of the patch based texture synthesis is largely improved in comparison to the pixel based method.

Nevertheless the patch based method has still problems. First, a trend to markable repetitions is observable. Second, the synthesized texture has an insufficient variability. Further on for some structured textures the block boundaries can be seen, and the applied feathering leads to an smoothing along these boundaries. Finally, the computational effort is still too high.

Advances in the computational cost could be gained by the application of a multiresolution image pyramid to the texture analysis. With that the computational effort could be minimized in some order of magnitude. But it is still highly dependent on the size of the input sample. Further approaches should try to decrease this dependency further.

Our approaches to a prevention of repetitions only make sense for highly stochastic textures. The algorithm works well for sufficiently large input samples. With respect to a very low additional computational cost caused by this algorithm, the results are satisfying. But in general a higher visual quality could be desired. Approaches in this direction could consider the spatial distance of the potentially repeated blocks. Also approaches, which exclude only blocks with a striking feature, could be imagined.

A higher variability of the synthesized image could be reached by the application of isometries. The approach seams reasonable and produces results of an undiminished, high visual quality. Also it could help to reduce the visibility of markable repetition by block transformations. But it causes a by the number of applied isometries multiplied computational effort. With this disadvantage the application of isometries has to be considered very well.

The synthesis algorithm is not only useable to synthesize completely new textures from a given texture sample. By marking certain pixels in the sample as undesired, effectively the input sample only was reduced. But this could be used to create new, clean textures from input samples without disturbing artifacts. Another step in this direction was the application of texture synthesis to segmented textures. So new segmented textures could be created. The algorithm works well for highly stochastic textures with no clear segmentation boundary. It fails for all strict deterministic boundaries, which do not have a complementary in the input sample. At this point the combination of patch and pixel based texture synthesis could lead to better results.

Further applications of the algorithm can be imagined. Logical consequence of the disregarding of certain image parts for the synthesis of a completely new image is, to replace these image parts directly within the image (cp. [8], [17]). We propose for this case a combination of pixel and patch based texture synthesis. The patch based method would be suited for synthesis of larger areas, whereas boundary parts could be synthesized with pixel based synthesis. The same considerations are valid for the synthesis of missing blocks in textures, e.g. caused by transmission errors.

A direct application could be imagined for the introduced and further advanced synthesis of segmentations. Pictures, consisting of various textures, could be created by a few samples. Typical examples for this are landscapes. So e.g. varying landscapes could be synthesized and used in trick films or video games. In these two application areas a trend to high realistic pictures can be observed.

Open question is still, if texture synthesis can make it into the domain of conventional lossy source coding, although the proposed application to trick films and video games could lead us to this assumption. Until today it suffers from a too less deterministic output texture, with a too varying visual fidelity and a too high computational cost. So no satisfying solutions e.g. for the source coding of motion pictures can be produced.

Concluding it can be said that texture synthesis has great chances for application in the future. Patch based texture synthesis has firstly shown a way to generate quickly textures of a high quality. With the here introduced modifications to the algorithm, some defects could be remedied, and already some application areas could be found. But there still remain possibilities to improve the algorithm and enable so a further distribution of this technique.

Bibliography

[1] Narendra Ahuja, Azriel Rosenfeld. Mosaic Models for Textures. IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume PAMI-3, Number 1, January 1981. In Rama Chellappa. Digital Image Processing, p. 107 - 117. IEEE Computer Society Press, Los Alamitos, California. 1992.

[2] Jeremy S. De Bonet. Multiresolution Texture Synthesis. http://www.debonet.com/Research/TextureSynthesis. February 2003.

[3] P. Brodatz. Textures: A Photographic Album for Artists and Designers. Dover Publications, New York. 1966.

[4] George R. Cross, Anil K. Jain. Markov Random Field Texture Models. IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume PAMI-5, Number 1, January 1983. In Rama Chellappa. Digital Image Processing, p. 44 - 58. IEEE Computer Society Press, Los Alamitos, California. 1992.

[5] Alexei A. Efros, William T. Freeman. Image Quilting for Texture Synthesis and Transfer. Proceedings of SIGGRAPH 2001. Los Angeles, California. August 2001.

[6] Alexei A. Efros, Thomas K. Leung. Texture Synthesis by Non-parametric Sampling. IEEE International Conference on Computer Vision, Corfu, Greece. September 1999.

[7] Rafael C. Gonzales, Paul Wintz. Digital Image Processing. Second Edition. Addison-Wesley, Reading, Massachusetts. 1987.

[8] Homan Igehy, Lucas Pereira. Image Replacement through Texture Synthesis. Stanford University. International Conference on Image Processing (ICIP'97). 1997.

[9] Arnaud E. Jacquin. Image Coding Based on a Fractal Theory of Iterated Contractive Image Transformations. In IEEE Transactions on Image Processing, Vol. 1, No. 1, p. 18 - 30. January 1992.

[10] Bernd Jähne. Digitale Bildverarbeitung. 5. überarbeitete und erweiterte Auflage. Heidelberg 2002.

[11] Anil K. Jain. Fundamentals of Digital Image Processing. Prentice Hall International. 1989.

[12] John Peter Lewis. Texture Synthesis for Digital Painting. Massachusetts Institute of Technology. In Computer Graphics Volume 18, Number 3 July 1984.

[13] Lin Liang, Ce Liu, Yingqing Xu, Baining Guo, Heung-Yeung Shum. Real-Time Texture Synthesis by Patch-Based Sampling. Technical Report MSR-TR-2001-40. Microsoft Research, March 2001. ftp://ftp.research.microsoft.com/pub/tr/tr-2001-40.pdf

[14] Rosalind Picard, Chris Graczyk, Steve Mann, Josh Wachman, Len Picard, Lee Campbell. Vision Tex Database. Media Laboratory, Massachusetts Institute of Technology (MIT) Cambridge, Massachusetts 1995.
http://www-white.media.mit.edu/vismod/imagery/VisionTexture/

[15] Wiliam K. Pratt. Digital Image Processing. John Wiley & Sons, Inc., New York. 1991.

[16] Shantanu D. Rane, Guillermo Sapiro, Marcelo Bertalmio. Structure and Texture Filling-In of Missing Blocks in Wireless Transmission and Compression Applications. University of Minnesota, Minneapolis. Accepted IEEE Trans. Image Processing, 2002.
http://www.stanford.edu/ srane/sdr_pubs_files/blockinpaint.pdf

[17] Jonathan Marshall Rowlett. Extracting Occlusions from Images Based on Texture-Synthesis Methods. Carnegie Mellon University. 1999.
http://www.cs.cmu.edu/ ph/869/results/rowlett/project/occ.doc

[18] Richard Szeliski, Heung-Yeung Shum. Creating Full View Panoramic Image Mosaics and Environment Maps. Computer Graphics (SIGGRAPH'97), pages 251-258, August 1997.

[19] Texture Synthesis by Fixed Neighborhood Searching. Dissertation. Stanford University. November 2001.

[20] Li-Yi Wei, Marc Levoy. Fast Texture Synthesis using Tree-structured Vector Quantization. Stanford University. In Proceedings of SIGGRAPH 2000.

[21] Li-Yi Wei. Texture Analysis and Synthesis. February 2003.
http://graphics.stanford.edu/projects/texture/.

[22] Steve Zelinka, Michael Garland. Towards Real-Time Texture Synthesis with the Jump Map. Thirteen Eurographics Workshop on Rendering. 2002.

Appendix A

User Manual

In the following, an user manual for the implemented functions of patch based texture synthesis is provided. The implementation was done by adding the functions to the already existing software PdiWin32, which was created by the Departamento de Comunicaciones of the university Universidad Politécnica de Valencia. The software was developed for Microsoft Windows operating systems. It is assumed that the reader is familiar in the usage of the basic concepts of the windows operating systems.

Two menu entries have been added to the pull down menu **VENIS**. The entry **Patch Based Synthesis** provides the possibilities, to synthesize new images from existing textures samples and to enlarge existing input samples to the right by patch based texture synthesis. The entry **Segmentation Synthesis (Patch Based)** provides the possibility to synthesize a defined segmented texture from an input sample and two segmentation schemes using patch based texture synthesis (Figure A.1). The entries are explained in the according sections below.

A.1 Patch Based Synthesis

The menu entry **Patch Based Synthesis** can be found in the pull down menu **VENIS**. It provides all in this work presented possibilities to synthesize textures with patch based texture synthesis, except for synthesis of segmentations. Please note that the texture sample, from which should be sampled, has to be opened and selected before choosing the menu entry. The opening of graphic files can be done in the pull down menu **Archivo**, entry **Abrir**. The file then can be opened with the common windows dialog.

After having chosen the menu entry, the dialog box **VENIS: Patch Based Texture Synthesis** appears (Figure A.2 (a)). This dialog box serves for choosing different applications and to modify parameters. In the following the entries are presented:

Patch size: Sets the patch size w_b, which is applied in the synthesis process. The default value is set to 25. The input box expects an integer value $1 \leq w_b \leq 99999$.

Seam size: Sets the size of the neighborhood (seam) w_e, which is applied in the synthesis process. The default value is set to 5. The input box expects an integer value $1 \leq w_e \leq 99999$.

Number of eval. Blocks: Sets the number of blocks n used as distance tolerance $d_{\text{max, n}}$. From the n blocks with minimum distance d the sampled block is chosen randomly.

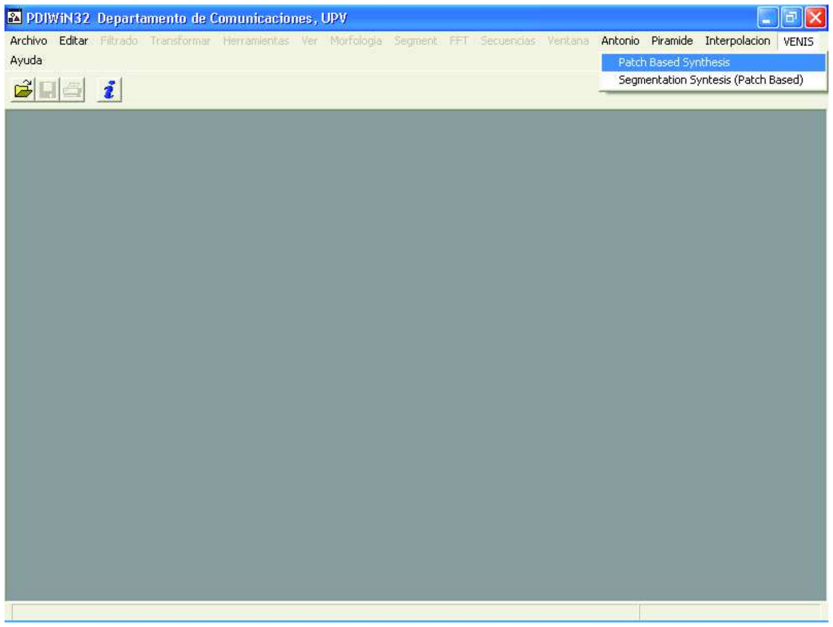
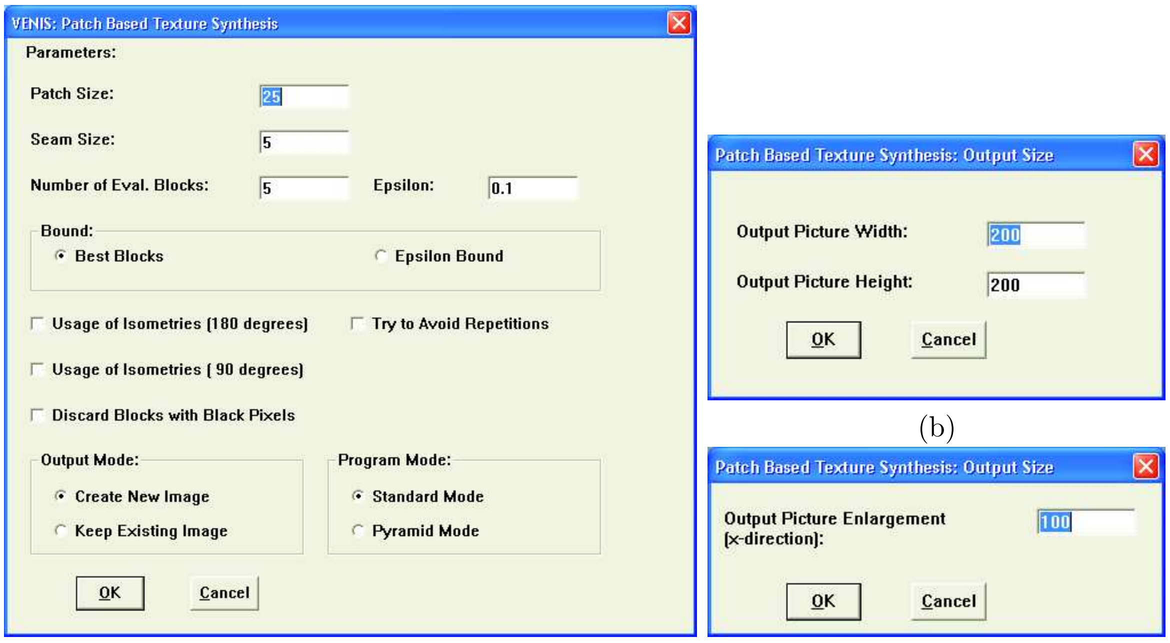

PDIWiN32 Departamento de Comunicaciones, UPV
Archivo Editar Filtrado Transformar Herramientas Ver Morfologia Segment FFT Secuencias Ventana Antonio Piramide Interpolacion VENIS
Ayuda
Patch Based Synthesis
Segmentation Syntesis (Patch Based)

Figure A.1: PdiWin32 shows after starting an empty desktop. The two entries **Patch Based Synthesis** and **Segmentation Synthesis (Patch Based)** were added to the menu **VENIS**.


VENIS: Patch Based Texture Synthesis

Parameters:

Patch Size: 25

Seam Size: 5

Number of Eval. Blocks: 5 Epsilon: 0.1

Bound: [✓ Best Blocks] Epsilon Bound

☐ Usage of Isometries (180 degrees) ☐ Try to Avoid Repetitions

☐ Usage of Isometries (90 degrees)

☐ Discard Blocks with Black Pixels

Output Mode: [✓ Create New Image] Keep Existing Image

Program Mode: [✓ Standard Mode] Pyramid Mode

Patch Based Texture Synthesis: Output Size

Output Picture Width: 200

Output Picture Height: 200

Patch Based Texture Synthesis: Output Size

Output Picture Enlargement (x-direction): 100

(a) (b) (c)

Figure A.2: (a) The dialog box **VENIS: Patch Based Texture Synthesis**. (b) The dialog box **Patch Based Texture Synthesis: Output Size**. (c) The dialog box **Patch Based Texture Synthesis: Output Size**.

Note that this distance tolerance would only be applied, if the field **Bound: Best Blocks** is selected. The default value is 5. The input box expects an integer value $1 \leq n \leq 99999$.

Epsilon: Sets the ϵ as distance tolerance $d_{\text{max, }\epsilon}$ according to Liang [13]. Note that this distance tolerance would only be only applied, if the field **Bound: Epsilon** is selected. The default value is 0.1. Expects a positive float value.

Button group Bound: In this button group the choice between the two presented distance tolerances $d_{\text{max, n}}$ and $d_{\text{max, }\epsilon}$ can be set (see also Section 3.3.4):

Best Blocks: Selects the distance tolerance $d_{\text{max, n}}$. The sampled block is chosen from the n best blocks. The parameter n can be specified in the input box **Number of eval. blocks**.

Epsilon Bound: Selects the distance tolerance $d_{\text{max, }\epsilon}$. The parameter ϵ can be specified in the input box **Epsilon**. If no block matches the ϵ-criterion, the best block is sampled.

The distance tolerance is by default set to **Best Blocks**.

Usage of isometries (180 degrees): Isometries of the category 1 (compatible 180°) are applied to the blocks considered for sampling (Section 4.1), if the check box is selected.

Usage of isometries (90 degrees): Isometries of the category 2 (compatible 90°) are applied to the blocks considered for sampling (Section 4.1), if the check box is selected.

Discard Blocks with Black Pixels: All blocks of the input sample, which contain at least one black pixel (all RGB components 0) are discarded for synthesis process, if the check box is selected (Section 5.2).

Try to Avoid Repetitions: The in Section 4.2 introduced algorithm to prevent repetition of sampled blocks is applied, if the check box is selected.

Button group Output Mode: In this button group, the desired output mode can be selected.

Create New Image: The output texture is initialized with a randomly chosen block from the input sample. Any user defined sizes can be chosen as output size. This is the basic processing of the algorithm, introduced in Chapter 3.

Keep Existing Image: The output texture is initialized with the input sample. The output texture consists of an to the right enlarged input sample (Section 5.1).

The output mode is by default set to **Create New Image**.

Button group Program Mode: With this button group the mode of the texture synthesis can be selected.

Standard Mode: Texture synthesis is done without application of a MRP.

Pyramid Mode: Texture synthesis is done with application of a MRP (Section 4.3).

The program mode is by default set to **Standard Mode**.

The pressing of the **OK**-button accepts the made settings, whereas a pressing of the **Cancel**-buttons discards them and returns to the PdiWin32 desktop.

In the following dialog the user has to specify the output size of the texture. Depending on the setting made in the button group **Output Mode**, one of the following dialogs appears.

Create New Image selected: The dialog Figure A.2 (b) appears. Within this dialog width and height of the synthesized image can be defined within the input boxes **Width** and **Height**. Both default values are set to 200. Both input boxes expect integer values $1 \leq value \leq 99999$.

Keep Existing Image selected: The dialog Figure A.2 (c) appears. Within this dialog the enlargement of the input sample to the right can be specified within the input box **Output Picture Enlargement (x-direction)**. The default value is set to 100. The input box expects an integer value $1 \leq value \leq 99999$.

By confirming this dialog with **OK**, the synthesis process is started. When completed, the output picture is copied to the desktop of PdiWin32. Note that the synthesis process can last up to several hours, depending on the size of the input sample, size of the output sample and applied isometries. Synthesized images can be saved using the **Guardar** and **Guardar como...** functions from the pull down menu **Archivo**.

A.2 Segmentation Synthesis (Patch Based)

Also the menu entry **Segmentation Synthesis (Patch Based)** can be found in the pull down menu **VENIS**. It provides the possibilities to synthesize defined texture segmentations (of max. two segments) by patch based texture synthesis. Before applying the function, three input images are needed to be opened. These images have to be in size and color format identical (i.e. all have to be color or gray scale). The needed files are:

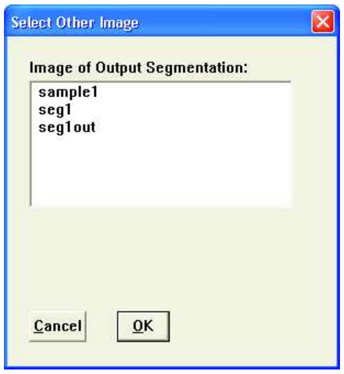
Input sample: The sample image, from which is sampled.

Segmentation of the input sample: The Segmentation of the input sample has to consist of two colors, black and white. It has to represent the segmentation of the input sample, painting one segment black, and the other segment white. Note that the segmentation scheme has to have the same color format as the input sample, also if containing only two colors.

Segmentation of the output texture: This segmentation scheme has to represent the desired segmentation of the output texture. As the segmentation scheme of the input sample, it has to consist of two different colors, black and white. Please note that the segmentation scheme has to be in size and color format identical to the input sample.

These images have to be opened in the PdiWin32 desktop. This can be done with the entry **Abrir** from the pull down menu **Archivo**. Further on the *input sample* has to selected before continuing.

After having chosen the menu entry **Segmentation Synthesis (Patch Based)** from the pull down menu, the dialog box **Select Other Image** (Figure A.3 (a)) appears. In this dialog box the segmentation of the input sample has to be selected. After confirming this with **OK**, another dialog box **Select Other Image** Figure A.3 (b)) is presented, in which



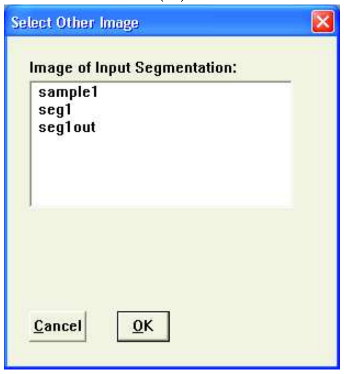
Select Other Image

Image of Output Segmentation:

sample1
seg1
seg1out

Cancel OK

(a)



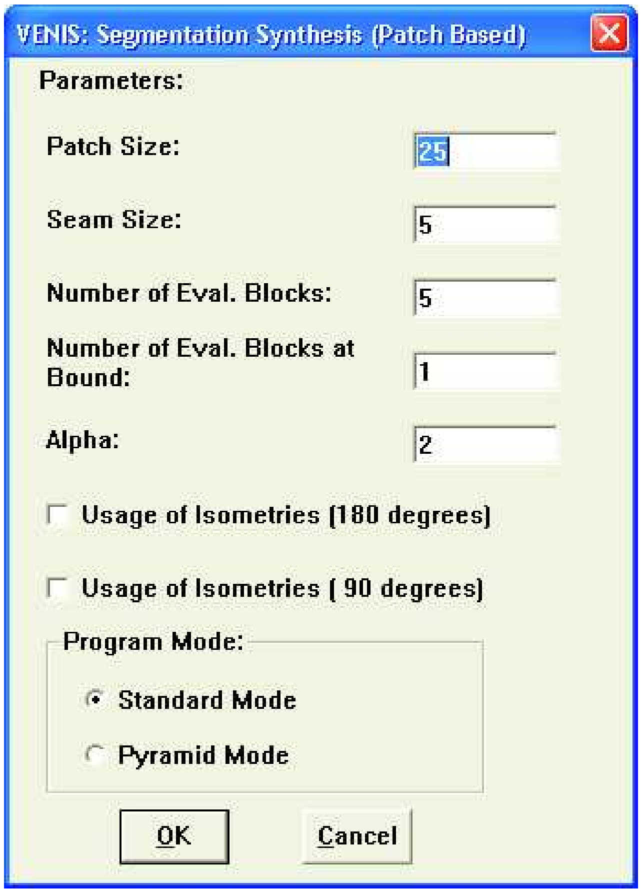
Select Other Image

Image of Input Segmentation:

sample1
seg1
seg1out

Cancel OK

(b)



VENIS: Segmentation Synthesis (Patch Based)

Parameters:

Patch Size:	25
Seam Size:	5
Number of Eval. Blocks:	5
Number of Eval. Blocks at Bound:	1
Alpha:	2

☐ Usage of Isometries (180 degrees)

☐ Usage of Isometries (90 degrees)

Program Mode:

[✓ Standard Mode]

Pyramid Mode

OK Cancel

(c)

Figure A.3: (a) Dialog box **Select Other Image** to select the segmentation of the input sample. (b) Dialog box **Select Other Image** to select the segmentation of the output texture. (c) Dialog box **VENIS: Segmentation Synthesis (Patch Based)**

the segmentation of the output sample has to be chosen. Note, if the segmentation schemes do *not* appear in these boxes, please ensure that they are in size and color format identical to the input sample. This can easily be done with the function **Informacion** from the pull down menu **Ver**.

After selecting the appropriate images and confirming the selection with **OK**, the dialog box **VENIS: Segmentation Synthesis (Patch Based)** (Figure A.3 (c)) is shown to the user. Within this dialog box all relevant parameters for segmentation synthesis can be modified.

Patch Size: Sets the patch size w_b, which is applied in the synthesis process. The default value is set to 25. The input box expects an integer value $1 \leq w_b \leq 99999$.

Seam Size: Sets the size of the neighborhood (seam) w_e, which is applied in the synthesis process. The default value is set to 5. The input box expects an integer value $1 \leq w_e \leq 99999$.

Input box Number of Eval. Blocks: Sets the number of blocks n used for the distance tolerance $d_{\text{max, n}}$. From the n blocks with minimum distance d the sampled block is chosen randomly. The default value is 5. The input box expects an integer value $1 \leq n \leq 99999$. This distance tolerance is only applied to all blocks, which are sampled fully to a segment of the segmentation.

Number of Eval Blocks at Bound: As above, this input box sets the number of blocks n' used for the distance tolerance $d_{\text{max, n}}$. From the n' blocks with minimum distance d the sampled block is chosen randomly. The default value is 1. The input box expects an integer value $1 \leq n \leq 99999$. This distance tolerance is applied to all blocks, which are sampled to the segmentation boundary.

Alpha: The parameter α regulates the influence of the the boundary segmentation to the distance calculation (Section 5.3). This is only valid for all blocks, which are sampled to the boundary of the segments. The default value is 2.0. Expects a positive float value.

Usage of Isometries (180 Degrees): Isometries of the category 1 (compatible 180°) are applied to the blocks considered for sampling (Section 4.1), if the check box is selected.

Usage of Isometries (90 Degrees): Isometries of the category 2 (compatible 90°) are applied to the blocks considered for sampling (Section 4.1), if the check box is selected.

Button group Program Mode: With this button group the mode of the texture synthesis can be selected.

Standard Mode: Texture synthesis is done without application of a MRP.

Pyramid Mode: Texture synthesis is done with application of a MRP (Section 4.3).

The program mode is by default set to **Standard Mode**.

By confirming the chosen settings with the **OK** button, the synthesis process is started. An image according to the output segmentation scheme, sampled from the input sample is synthesized. The output texture is in size and color information identical to the input sample. It is finally copied to the PdiWin32 desktop. Please note that the synthesis process might last up to several hours, depending on the size of the input sample and the chosen settings. Synthesized images can be saved using the **Guardar** and **Guardar como...** functions from the pull down menu **Archivo**.

Appendix B

Technical documentation of the implemented Software

This appendix describes the implemented code. First, general informations are given. Second, the two main functions are presented, and their usage is described. Finally, all visible functions are documented.

The code description is grouped by files. For the application of each function the according header file ("filename.h") has to be included.

All in this appendix presented code is pseudo code. It follows the C++ code syntax, but i.e. it is not complete. At the beginning of each code example, the to be included header files are presented:

```
#include <stdio.h>
#include "headerfile.h"
```

So it is showed that the header files "stdio.h" "headerfile.h" have to be included. Further on it is signaled that the file "stdio.h" is part of the C++ standard library by the usage of `< >`.

Afterwards, brackets show the beginning and ending of the function.

```
{

}
```

As far as not otherwise mentioned, the presented (pseudo code) functions return an integer value `int`.

All in the function used variables are declared as in the C++ syntax. The usage of ... signals the leave out of certain function parts, i.e. the memory allocation or initialization of earlier declared variables. This is only done with respect to a compact description. The left out parts are mentioned in the text and are already described earlier.

B.1 Documentation

B.1.1 General

The code is written in C++ for Microsoft Windows (32 bit) systems. It is provided as source code. The integration into the existing software PdiWin32 is done via the Borland C++

development environment in the file `jppatchtexturesynthesis.cpp`, whose description is not part of this documentation. All other files are independent on this project and are written in standard C++.

The code was written with respect to a safe and easy error handling. Philosophy behind the written functions is, to give back an error code as an integer value. In all cases of an error free, normal termination of the function, the function returns `NO_ERROR`. In all other cases an error code according to the Victor Image Processing library is returned. With respect to this, the function should be called in the following manner:

```
int errorHandler;

if(NO_ERROR != (errorHandler = function(...))){
     errorHandling(errorHandler);
     return errorHandler;
}
```

The used error codes are:

- `NO_ERROR` Function is terminated normally. No error occurred during execution.
- `BAD_MEM` Error with memory management occurred. The function is not able to allocate memory or insufficient memory for execution allocated.
- `BAD_RANGE` Variable contains invalid value.

All the code was written for a Microsoft Windows (32 bit) operating system. In general it is not portable, mainly because the memory allocation was done with appropriate windows functions.

B.1.2 jptexturepatch.h, jptexturepatch.cpp

These files provide the main patch based texture synthesis function:

```
int texture_patch(imgdes *srcimg, imgdes *desimg, datapatch varpatch)
```

- **Description:** The function synthesizes an image `imgdes desimg` using patch based texture synthesis from the input sample `imgdes srcimg` with the given parameters `datapatch varpatch`. The parameters are the following:

  ```
  struct datapatch{
    long patch;
    long seam;
    long numEvalBlocks;
    float epsilon;
    int isometries_180_true;
    int isometries_90_true;
    int createNewImageTrue;
    int discardBlackBlockTrue;
    int useModePyramidTrue;
    int avoidRepetitionsTrue;
    findBestBound bound;
  };
  ```

  - `long patch` Size of the patch (w_b), quadratic. Expects a long integer value > 0.
  - `long seam` Size of the seam (w_e). Expects a long integer value > 0.
  - `long numEvalBlocks` Number of n best blocks, which are regarded for synthesis (distance tolerance $d_{\text{max, n}}$, Section 3.3.4). Expects a long integer value > 0. Only valid if `findBestBound == BOUND_NUMBER`, else set to 0.
  - `float epsilon` ϵ of distance tolerance $d_{\text{max, }\epsilon}$ (Section 3.3.4). Expects a float value > 0. Only valid, if `findBestBound == BOUND_EPSILON`, else set to 0.
  - `int isometries_180_true` Application of isometries compatible to 180°(isometries category 1, Section 4.1), if `TRUE`. Expects `TRUE` or `FALSE`.
  - `int isometries_90_true` Application of isometries compatible to 90°(isometries category 2, Section 4.1), if `TRUE`. Expects `TRUE` or `FALSE`.
  - `int createNewImageTrue` A completely new image is created, if `TRUE`. If `FALSE`, the existing image is enlarged. Expects `TRUE` or `FALSE`.
  - `int discardBlackBlockTrue` If `TRUE`, all blocks containing black pixels are discarded for synthesis process (Section 5.2). Expects `TRUE` or `FALSE`.
  - `int useModePyramidTrue` If `TRUE`, a MRP is applied to the synthesis process. Please note that in this case the size of patch and seam have to be chosen large enough (≥ 4). If `FALSE`, the synthesis is done without MRP (Section 4.3).
  - `int avoidRepetitionsTrue` If `TRUE`, additionally the algorithm for repetition prevention is applied (Section 4.2). Expects `TRUE` or `FALSE`.

  - `findBestBound bound` Selects the distance tolerance, which is used for synthesis process. A setting to `findBestBound BOUND_EPSILON` selects the distance tolerance $d_{\max,\ \epsilon}$, whereas a setting to `findBestBound BOUND_NUMBER` selects the distance tolerance $d_{\max,\ \mathrm{n}}$ (Section 3.3.4). Expects `BOUND_EPSILON` or `BOUND_NUMBER`.

  Sizes and color information of the output images have to be provided by the initialized `imgdes srcimg` and `imgdes desimg`. The function returns an error code.

- Usage: In the following example, a new texture `imgdes outTexture` of 300×300 is synthesized from `imgdes inTexture`. Patch size is set to 25, seam size to 5. Distance tolerance $d_{\max,\ \mathrm{n}}$ is applied, n is set to 4. No additional isometries are applied, but MRP. No application of repetition prevention, or discarding of certain pixels (blocks containing black pixels) is used.

```
#include <stdio.h>
#include "vicdefs.h"
#include "vicdef.h"
#include "jpmyerror.h"
#include "jptexturepatch.h"
{
int errorHandler;
imgdes inTexture;
...

// read out, if the input texture is color or gray scale
// value, how many bits/pixel are needed is saved in tmpBpps
// needed to allocate buffer for outTexure
int tmpWidth, tmpHeight, tmpBpps;
CalcularParametros(&inTexture, &tmpWidth, &tmpHeight, &tmpBpps);

// image outTexture declared and initialized
imgdes outTexture;
if(NO_ERROR != (errorHandler =
     allocimage(&outTexture, 300, 300, tmpBpps))){
     fprintf(stderr, "Unable to alloc for outTexture");
     return BAD_MEM;
}


// allocation of values to datapatch synthesisData
datapatch synthesisData;
synthesisData.patch                 = 25;
synthesisData.seam                  = 5;
synthesisData.numEvalBlocks         = 4;
synthesisData.epsilon               = 0; // not used
synthesisData.isometries_180_true   = FALSE;
synthesisData.isometries_90_true    = FALSE;
synthesisData.createNewImageTrue    = TRUE;
synthesisData.discardBlackBlockTrue = FALSE;
synthesisData.useModePyramidTrue    = TRUE;
synthesisData.avoidRepetitionsTrue  = FALSE;
synthesisData.bound                 = BOUND_NUMBER;

if(NO_ERROR != (errorHandler =
     texture_patch(&inTexture, &outTexture, synthesisData))){
     fprintf(stderr, "Error at texture_patch()");
     return errorHandler;
}
}
```

B.1.3 jpsegmentsynth.h, jpsegmentsynth.cpp

These files provide the main segmentation synthesis function, using patch based texture synthesis:

```
int segmentSynth(imgdes *origImage, imgdes *inputSegmentation, imgdes
*outputSegmentation, imgdes *outputImage, dataSegmentSynth *varSegmentSynth)
```

- **Description:** The function provides the possibility to synthesize a segmented texture `imgdes outputImage` with a segmentation `imgdes outputSegmentation` from an input sample `imgdes origImage` with according segmentation `imgdes inputSegmentation`. The segmentation must consist of two different segments, marked with black and white. Further on the segmentations must be in size and color mode identical to `imgdes origImage`. As distance tolerance d_{max}, the distance tolerance $d_{\text{max, n}}$ is used (3.3.4). The parameters of the synthesis can be set in `dataSegmentSynth varSegmentSynth`, which is in the following described.

  ```
  struct dataSegmentSynth{
    long patch;
    long seam;
    long numEvalBlocks;
    long numEvalBlocksAtBound;
    float alpha;
    int isometries_180_true;
    int isometries_90_true;
    int usePyramidModeTrue;
  };
  typedef struct dataSegmentSynth dataSegmentSynth;
  ```

  – `long patch` Size of the patch (w_b), quadratic. Expects a long integer value > 0.

  – `long seam` Size of the seam (w_e). Expects a long integer value > 0.

  – `long numEvalBlocks` Number n best blocks, which are regarded for synthesis (Section 3.3.4). Expects a long integer value > 0. This value is only valid for all synthesized blocks, which do not touch the segmentation boundary.

  – `long numEvalBlocksAtBound` Number n' best blocks, which are regarded for synthesis (Section 3.3.4). This value is only valid for all synthesized blocks, which touch the segmentation boundary. Expects a long integer value > 0.

  – `float alpha` Parameter α, moderates the influence of the segmentation to the synthesis process at the segmentation boundary (Section 5.3). Expects a float value ≥ 0.

  – `int isometries_180_true` Application of isometries compatible to 180°(isometries category 1, Section 4.1), if `TRUE`. Expects `TRUE` or `FALSE`.

  – `int isometries_90_true` Application of isometries compatible to 90°(isometries category 2, Section 4.1), if `TRUE`. Expects `TRUE` or `FALSE`.

  – `int useModePyramidTrue` If `TRUE`, a MRP is applied to the synthesis process. Please note that in this case the size of patch and seam have to be chosen large enough (≥ 4). If `FALSE`, the synthesis is done without MRP (Section 4.3). Expects `TRUE` or `FALSE`.

The function returns an error code.

- **Usage:** In the following example, the usage of `segmentSynth()` is demonstrated. The texture `imgdes outTexture`, with segmentation `imgdes outSegmentation` is synthesized from the input sample `imgdes inTexture`, with according segmentation `imgdes inSegmentation`. Patch size is set to 10, seam size is set to 4. Isometries of all two categories are applied. No MRP is applied. α is set to 2.0. n is set to 5, and n' (at the segmentation boundary) is set to 1.

```
#include <stdio.h>
#include "vicdefs.h"
#include "vicdef.h"
#include "jpsegmentsynth.h"
{
int errorHandler;
imgdes inTexture, inSegmentation, outSegmentation.
...

// read out, if the input texture is color or gray scale
// write how many bits/pixel to tmpBpps
// needed to allocate right buffer for outTexure
int tmpWidth, tmpHeight, tmpBpps;
CalcularParametros(&inTexture, &tmpWidth, &tmpHeight, &tmpBpps);

// allocate memory to outTexture
imgdes outTexture;
if(NO_ERROR != (errorHandler =
     allocimage(&outTexture, tmpWidth, tmpHeight, tmpBpps))){
     fprintf(stderr, "Unable to alloc for outTexture");
     return BAD_MEM;
}

// initialize dataSegmentSynth synthesisData
dataSegmentSynth synthesisData;
synthesisData.patch               = 10;
synthesisData.seam                = 4;
synthesisData.numEvalBlocks       = 5;
synthesisData.numEvalBlocksAtBound = 1;
synthesisData.alpha               = 2.0;
synthesisData.isometries_180_true  = TRUE;
synthesisData.isometries_90_true   = TRUE;
synthesisData.usePyramidModeTrue   = FALSE;

if(NO_ERROR != (errorHandler =
     segmentSynth(&inTexture, &inSegmentation, &outSegmentation,
                  &outTexture, &synthesisData))){
     fprintf(stderr, "Error at segmentSynth");
     return errorHandler;
}
}
```

B.1.4 jpmyimage.h, jpmyimage.cpp

These files provide a to texture synthesis adapted image representation and handling. Image representation is done with the type `myImage`. Color images thereby are represented in the three RGB components. Switching between color and gray scale images is done via

```
enum colorMode{BLACKWHITE, COLOR};
```

where `BLACKWHITE` represents pictures with 256 gray levels/pixel and `COLOR` represents color images.

Type myImage

The type is `myImage` is defined as followed:

```
struct myImage{
  unsigned char *imageBufferR;
  unsigned char *imageBufferG;
  unsigned char *imageBufferB;
  colorMode      mode;
  unsigned long  maxBufferSize;
  long           width;
  long           height;
  HGLOBAL hImageBufferR;
  HGLOBAL hImageBufferG;
  HGLOBAL hImageBufferB;
  HGLOBAL hMyImage;
};
typedef struct myImage myImage;
```

- `unsigned char *imageBufferR` Pointer to an array of unsigned char, containing the R component of a color image, as image of 256 gray levels. If the image is gray level, this buffer contains the image of 256 gray levels. The array has the size of `maxBufferSize`. The image has the size of `width` $\times$ `height`, and is filled in the array in raster scan order, from top to bottom and from left to right (i.e. the point (0,0) is the upper left corner of the image). So the point (x,y) can be accessed by `imageBufferR[y*width + x]`.

- `unsigned char *imageBufferG` Pointer to an array of unsigned char, containing the G component of a color image. The pointer is NULL for all gray scale images. Processing as above.

- `unsigned char *imageBufferB` Pointer to an array of unsigned char, containing the B component of a color image. The pointer is NULL for all gray scale images. Processing as above.

- `colorMode mode` Color mode of the image. A `colorMode == COLOR` marks a color image in RGB components, whereas `colorMode == BLACKWHITE` marks a gray scale image.

- `unsigned long maxBufferSize` Size of the allocated image buffers `imageBufferR`, `imageBufferG`, `imageBufferB`. Note, if the image is allocated as gray scale (`mode == BLACKWHITE`), the image buffers `imageBufferG` and `imageBufferB` are not allocated, but NULL.

- `long width` Width of the image contained in `myImage`. Value is set when `myImage` is initialized (not when allocated!).
- `long height` Height of the image contained in `myImage`. Value is set when the image is initialized (not when allocated!).
- `HGLOBAL hImageBufferR` Windows memory handler. For allocating and freeing memory. Should not be accessed!
- `HGLOBAL hImageBufferG` Windows memory handler. For allocating and freeing memory. Should not be accessed!
- `HGLOBAL hImageBufferB` Windows memory handler. For allocating and freeing memory. Should not be accessed!
- `HGLOBAL hmyImage` Windows memory handler. For allocating and freeing memory. Should not be accessed!

Allocating and Freeing Memory of myImage

`int createMyImage(myImage **self, long maxWidth, long maxHeight, colorMode mode)`

- **Description:** Allocates memory to a pointer of the image `self` of type `myImage`. Afterwards an image of the maximum width `long maxWidth`, maximum height `long maxHeight` and of the color format `colorMode mode` can be saved in `self`. The function returns an error code.
- **Usage:** In the following example, memory to a color image `testImage` of the max. width of 100 and the max. height of 50 is allocated.

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpmyimage.h"
{
int      errorHandler;// error handler
myImage *testImage;   // declaration of the pointer to myImage testImage

// allocating memory to testImage
// max. width 100, max. height 50, color image
if(NO_ERROR != (errorHandler =
      createMyImage(&testImage, 100, 50, COLOR))){
      fprintf(stderr, "Error when creating testImage.");
      return errorHandler;
}
}
```

`int destroyMyImage(myImage *self)`

- **Description:** Frees the memory allocated to `myImage self`. The function returns an error code.
- **Usage:** In the following, the memory, allocated to testImage, is freed.

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpmyimage.h"
{
int errorHandler;
myImage *testImage;
...

if(NO_ERROR != (errorHandler = destroyImage(testImage))){
      fprintf(stderr, "Error when destroying testImage.");
      return errorHandler;
}
}
```

`int myImageInfo(myImage *image, long *width, long *height, colorMode *mode)`

- **Description:** Provides information about the created image `myImage image`. Writes values of the image width, image height and the color mode to `long width`, `long height`, `colorMode mode`. The function returns an error code.
- **Usage:** In the following example, the width, height and color mode of `testImage` are written to `long testWidth`, `long testHeight`, `colorMode testMode`.

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpmyimage.h"
{
int errorHandler;
myImage *testImage;
...

long      testWidth, testHeight;
colorMode testMode;

if(NO_ERROR != (errorHandler =
     myImageInfo(testImage, &testWidth, &testHeight, &testMode))){
     fprintf(stderr, "Error at myImageInfo.");
     return errorHandler;
}
}
```

Converting images to/from the Victor Image Library from/to myImage

Two functions are provided to convert images to the format `imgdes` from the Victor Image Library to `myImage` and vice versa.

`int copyImage2MyImage(imgdes *inImage, myImage *outImage, long width, long height, colorMode mode)`

- **Description:** Copies an image `imgdes inImage` of the width `width`, height `height` and color mode `mode` from the format `imgdes` (Victor Image Library. See there for details) to `myImage outImage`. The user has to initialize `width`, `height` and `mode`. The function returns an error code.

- **Usage:** In the following, the color image `imgdes vicImage` of the width 100, height 50 is copied to `myImage testImage`. Before copying, memory is allocated to `testImage`.

```
#include <stdio.h>
#include "vicdefs.h"
#include "jpmyerror.h"
#include "jpmyimage.h"
{
int errorHandler;
imgdes *vicImage;
...

myImage *testImage;

// allocating memory to testImage
// max. width 100, max. height 50, color image
if(NO_ERROR != (errorHandler =
      createMyImage(&testImage, 100, 50, COLOR))){
      fprintf(stderr, "Error when creating testImage.");
      return errorHandler;
}

// copying imgdes vicImage to testImage
if(NO_ERROR != (errorHandler =
      copyImage2MyImage(vicImage, testImage, 100, 50, COLOR))){
      fprintf(stderr, "Error when copying testImage.");
      return errorHandler;
}
}
```

`int copyMyImage2Image(myImage *inImage, imgdes *outImage, long *width, long *height, colorMode *mode)`

- **Description:** Copies an image `myImage inImage` to `imgdes outImage`. Width, height and color mode are written to `long width`, `long height`, `colorMode mode`. The function returns an error code. Please note, that sufficient memory has to be allocated to `outImage`.
- **Usage:** In the following, the copying of the image `myImage testImage` to the image `imgdes vicImage` is demonstrated.

```
#include <stdio.h>
#include "vicdefs.h"
#include "jpmyerror.h"
#include "jpmyimage.h"
{
int errorHandler;
myImage *testImage;
...

long      tmpWidth, tmpHeight;
colorMode tmpMode;

if(NO_ERROR != (errorHandler =
     myImageInfo(testImage, &tmpWidth, &tmpHeight, &tmpMode))){
     fprintf(stderr, "Error at myImageInfo");
     return errorHandler;
}

// an image of Victor Image Libary has to be allocated.
// See Victor Image Libary for details
int vicBits = 8;
if(tmpMode == COLOR){
     vicBits = 24;
}
imgdes vicImage;
if(NO_ERROR != (errorHandler =
     allocimage(&vicImage, tmpWidth, tmpHeight, vicBits))){
     fprintf(stderr, "Error allocating memory to vicImage.");
     return errorHandler;
}

long      tmpWidth, tmpHeight;
colorMode tmpMode;

if(NO_ERROR != (errorHandler =
      copyMyImage2Image(testImage, &vicImage, &tmpWidth,
                        &tmpHeight, &tmpMode))){
      fprintf(stderr, "Error copy image to testImage");
      return errorHandler;
}
}
```

Copying and Pasting Blocks from/to myImage

`int getBlock(myImage *srcImage, myImage *block, long index_i, long index_j, long size_i, long size_j)`

- **Description:** Copies a block from `myImage srcImage` to `myImage block`. The upper left corner of `block` in `srcImage` is `index_i`, `index_j`). The block has a width of `size_i` and a height of `size_j`. Note that the block has to be fully in `srcImage`. The function returns an error code.
- **Usage:** In the following, a block `myImage srcBlock` of width 40 and height 20 is copied from `myImage srcImage`. The upper left corner of the block in `srcImage`, from which it is taken, is at (15,10).

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpmyimage.h"
{
int errorHandler;
myImage *srcImage;
...

long      tmpWidth, tmpHeight;
colorMode tmpMode;

if(NO_ERROR != (errorHandler =
     myImageInfo(srcImage, &tmpWidth, &tmpHeight, &tmpMode))){
     fprintf(stderr, "Error at myImageInfo.");
     return errorHandler;
}

myImage *srcBlock;
if(NO_ERROR != (errorHandler =
     createMyImage(&srcBlock, 40, 20, tmpMode))){
     fprintf(stderr, "Error allocating for srcBlock.");
     return errorHandler;
}

if(NO_ERROR != (errorHandler =
     getBlock(srcImage, srcBlock, 15, 10, 40, 20))){
     fprintf(stderr, "Error copying block from srcImage.");
     return errorHandler;
}
}
```

`int fillPatchInImage(myImage *block, myImage *destImage, long patchWidth, long patchHeight, long seam, long destI, long destJ, int blendingTrue)`

- **Description:** Fills `myImage block` into `myImage destImage`. The block consists of a patch of a width of `long patchWidth` and a height of `long patchHeight`. It is covered with a seam of size `long seam`. The upper left corner of the patch is filled to (`destI`, `destJ`). If `blendingTrue == TRUE`, the feathering is used to for the upper and left seam region. If `blendingTrue == FALSE`, the outer `seam/2` pixels are kept from `destImage`, and the other pixels are overwritten with pixels from `block`. This behavior does not affect the pixels in the patch. Note that the function `fillPatchInImage()` does automatically consider a correct edge handling. So only (`destI`, `destJ`) has to be inside of `destImage`. Whenever a seam or parts of the patch would exceed `destImage`, this is taken into account, and these parts are not copied. The function returns an error code.

- **Usage:** In the following, it is demonstrated, how to fill a block `myImage testBlock`, consisting of a patch of 25×25, with a seam of 5 into the image `myImage *outImage`. The upper left corner of the patch is (0,0).

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpmyimage.h"
{
int errorHandler;
myImage *testBlock;
myImage *outImage;
...

if(NO_ERROR != (errorHandler =
     fillPatchInImage(testBlock, outImage, 25, 25, 5, 0, 0, TRUE))){
     fprintf(stderr, "Error filling block in image.");
     return errorHandler;
}
}
```

B.1.5 jpisometries.h, jpisometries.cpp

`int getIsometry(myImage *inBlock, myImage *outBlock, isometType isometry)`

- **Description:** The function applies the isometry `isometType isometry` to the image `myImage inBlock` and copies the result to `myImage outBlock`. The images have to be quadratic. The function returns an error code. The following isometries can be applied (in brackets equivalents to Section 4.1):
  - `isometType IDENT` Applies the isometry *identity* (0. isometry) to the block.
  - `isometType REFLECT_VERTICAL` Applies an *orthogonal reflection about mid-vertical axis* (1. isometry) to the block.
  - `isometType REFLECT_HORIZONTAL` Applies an *orthogonal reflection about mid-horizontal axis* (2. isometry) to the block.
  - `isometType REFLECT_1ST_DIAGONAL` Applies an *orthogonal reflection about first diagonal of block* (3. isometry).
  - `isometType REFLECT_2ND_DIAGONAL` Applies an *orthogonal reflection about second diagonal of block* (4. isometry).
  - `isometType ROTATE90` Applies a *rotation around center of block, through +90°* to the block (5. isometry).
  - `isometType ROTATE180` Applies a *rotation around center of block, through +180°* to the block (6. isometry).
  - `isometType ROTATE270` Applies a *rotation around center of block, through -90°* to the block (7. isometry).
- **Usage:** The following example demonstrates the allocation of sufficient memory to `myImage rotatedTestBlock`, and afterwards the application of a rotation around the center of the block through +180°to `myImage testBlock`. The result is copied to `myImage rotatedTestBlock`.

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpisometries.h"
#include "jpmyimage.h"
{
int errorHandler;
myImage *testBlock;
...

long      tmpWidth, tmpHeight;
colorMode tmpMode;

if(NO_ERROR != (errorHandler =
     myImageInfo(testBlock, &tmpWidth, &tmpHeight, &tmpMode))){
     fprintf(stderr, "Error at myImageInfo.");
     return errorHandler;
}

myImage *rotatedTestBlock;

if(NO_ERROR != (errorHandler =
     createMyImage(&rotatedTestBlock, tmpWidth, tmpHeight, tmpMode))){
     fprintf(stderr, "Error allocating for rotatedTestBlock.");
     return errorHandler;
}

if(NO_ERROR != (errorHandler =
     getIsometry(testBlock, rotatedTestBlock, ROTATE180))){
     fprintf(stderr, "Error applying isometry to testBlock.");
     return errorHandler;
}
}
```

B.1.6 jpfindblocks.h, jpfindblocks.cpp

Functions to Search the Next Sampled Block

`int findBestBlock(databest *varbest, databestOut *outbest, int extSwitch);`

- **Description:** The function searches the next block, which has to be sampled for patch based texture synthesis. Input parameters are given via `databest varbest`, output parameters are written to `databestOut outbest`. `int extSwitch` signals to the function, if the next block has to be placed at the upper border of the output image (`extSwitch = 1`), at the left border (`extSwitch = 2`) or at another place (`extSwitch = 0`). In the following, the structures `databest` and `databestOut` are explained:

  - `struct databest`

    ```
    struct databest{
      findBestMode  opMode;
      findBestBound opBound;
      float epsilon;
      long  numEvalBlocks;
      long  seam;
      long  patch;
      long  numBlocks;
      int   isometries_180_true;
      int   isometries_90_true;
      int   isometriesFactor;
      myImage *rightColumn;
      myImage *lowerRow;
      float    alpha;
      float   *error;
      float   *sort_error;
      myImage *origImage;
      myImage *inputSegmentation;
      myImage *outputSegmentation;
      myImage *origBlock;
      myImage *mutantBlock;
      myImage *schemeBlock1;
      myImage *schemeBlock2;
      myImage *schemeBlock3;
      regions findRegion;
      regions *classifiedBlocks;
      int     normValue;
      int     discardBlackBlockTrue;
      int     avoidRepetitionsTrue;
      imageEvaluation *origImageEval;
    };
    ```

    * `findBestMode opMode`: Two operation modes can be applied to `findBestBlock`. `findBestMode MODE_TEXTUREPATCH` applies the mode for patch based texture synthesis, without segmenting the in- and output images.

`findBestMode MODE_SEGMENTSYNTH` applies a mode, where the in- and output image can be segmented with usage of two additional segmentation schemes (cp. Section 5.3). Expects either `MODE_TEXTUREPATCH` or `MODE_SEGMENTSYNTH`.

∗ `findBestBound opBound`: Two different distance tolerances can be applied to the patch based texture synthesis. `findBestBound BOUND_EPSILON` applies the distance tolerance $d_{\text{max}, \epsilon}$, with a to be defined parameter ϵ, whereas `findBestBound BOUND_NUMBER` applies the distance tolerance $d_{\text{max, n}}$, with a to be defined parameter n (cp. Section 3.3.4). Expects `findBestBound BOUND_EPSILON` or `findBestBound BOUND_NUMBER`.

∗ `float epsilon` defines the ϵ of the distance tolerance $d_{\text{max}, \epsilon}$. Set `epsilon = 0`, if `opBound != BOUND_EPSILON`. Expects a positive float value.

∗ `long numEvalBlocks` defines the number n of the distance tolerance $d_{\text{max, n}}$. Set `numEvalBlocks = 0`, if `opBound != BOUND_NUMBER`. Expects a long integer value > 0.

∗ `long seam` Width of the seam. Correspondents with w_e. Expects a long integer value > 0.

∗ `long patch` Width, height of a quadratic patch. Correspondents with w_b. Expects a long integer value > 0.

∗ `long numBlocks` Number of all valid blocks for synthesis in the input sample. It can be calculated by $(inputWidth - patch - 2 * seam + 1) * (inputHeight - patch - 2 * seam + 1)$, where $inputWidth$ and $inputHeight$ are width and height of the input sample. Expects a long integer value > 0.

∗ `int isometries_180_true` If `isometries_180_true == TRUE`, isometries of the 1st category are applied (Section 4.1). The number of the additional applied isometries is stored in `NUM_ISOMETRIES_180`. Expects `TRUE` or `FALSE`.

∗ `int isometries_90_true` If `isometries_90_true == TRUE`, isometries of the 2nd category are applied (Section 4.1). The number of the additional applied isometries is stored in `NUM_ISOMETRIES_90`. Expects `TRUE` or `FALSE`.

∗ `int isometriesFactor` Has to be set to the total number of applied isometries. It is minimum 1 (because isometry *Identity* is always applied). If `isometries_180_true == TRUE`, `isometriesFactor += NUM_ISOMETRIES_180`. If `isometries_90_true == TRUE`, `isometriesFactor += NUM_ISOMETRIES_90`. Expects an integer value > 0.

∗ `float *error` Pointer to an array of float with $isometriesFactor * numBlocks$ entries. The memory has to be allocated before calling the function. The array contains the distances d of all blocks.

∗ `float *sort_error` Pointer to an array of float with $isometriesFactor * numBlocks$ entries. The memory has to be allocated before calling the function. The array contains the sorted distances of all blocks.

∗ `float alpha` Factor α for synthesis of segmentations (Chapter 5.3). Only valid, if `opMode == MODE_SEGMENTSYNTH`, else set to `alpha = 0`. Expects a float value ≥ 0.

∗ `myImage *rightColumn` Pointer to `myImage`, contains the image part, that the left part of the block seam is compared with(Figure B.1).

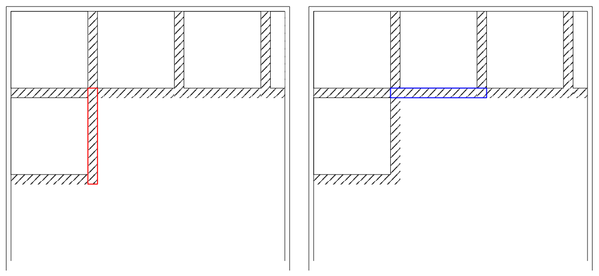


Figure B.1: The two marked seam parts are copied separately to the images `myImage rightColumn` (red), `myImage lowerRow` (blue), for distance calculation.

* `myImage *lowerRow` Pointer to `myImage`, contains the image part, that the upper part of the block seam is compared with (Figure B.1).
* `myImage *origImage` Pointer to `myImage`, containing the input sample.
* `myImage *inputSegmentation` Pointer to `myImage`, containing the segmentation of the input sample. Only necessary, if `opMode == MODE_SEGMENTSYNTH`, else set to `NULL`. The segmentation has to consist of two colors, black and white. The color mode has to be gray scale.
* `myImage *outputSegmentation` Pointer to `myImage`, containing the segmentation of the output texture. Only necessary, if `opMode == MODE_SEGMENTSYNTH`, else `NULL`. The segmentation has to consist of two colors, black and white. The color mode has to be gray scale.
* `myImage *origBlock` Pointer to `myImage`. Enough memory for one block (patch +2seam)×(patch + 2seam) has to be allocated. The color mode has to be identical to the color mode of the input sample.
* `myImage *mutantBlock` Pointer to `myImage`. Enough memory for one block (patch +2seam)×(patch + 2seam) has to be allocated. The color mode has to be identical to the color mode of the input sample.
* `myImage *schemeBlock1` Pointer to `myImage`. Enough memory for one block (patch +2seam)×(patch + 2seam) has to be allocated. The color mode has to be identical to the color mode of the input segmentation. Only necessary if `opMode == MODE_SEGMENTSYNTH`, else `NULL`.
* `myImage *schemeBlock2` Pointer to `myImage`. Enough memory for one block (patch +2seam)×(patch + 2seam) has to be allocated. The color mode has to be identical to the color mode of the input segmentation. Only necessary if `opMode == MODE_SEGMENTSYNTH`, else `NULL`.
* `myImage *schemeBlock3` Pointer to `myImage`. Enough memory for one block $(patch + 2seam) \times (patch + 2seam)$ has to be allocated. The color mode has to be identical to the color mode of the input segmentation. Only necessary if `opMode == MODE_SEGMENTSYNTH`, else `NULL`.
* `regions findRegion` Has to be set to the region, the next synthesized block should belong to. Valid regions are:
  · `REGION_1` Block is located totally in the region of the input segmentation marked white.

        · `REGION_2` Block is located totally in the region of the input segmentation marked black.
        · `BOTH_REGIONS` Block is partially located in both regions.

      Only necessary if `opMode == MODE_SEGMENTSYNTH`, else set to 0.

    * `regions *classifiedBlocks` Has to contain a classification of all `numBlocks` blocks of the input sample, to which region they belong. Only necessary if `opMode == MODE_SEGMENTSYNTH`, else set to `NULL`.
    * `int normValue` Has to contain a value for a scaling of the distance between the color schemes. Should be the difference between the two colors of the input segmentation. Only necessary if `opMode == MODE_SEGMENTSYNTH`, else set to 0.
    * `int discardBlackBlockTrue` Set this `TRUE` to discard all blocks containing black pixels for texture synthesis (Section 5.2). Expects `TRUE` or `FALSE`.
    * `avoidRepetitionsTrue` Set this `TRUE` to try to avoid repetitions of blocks (Section 4.2). Expects `TRUE` or `FALSE`.
    * `imageEvaluation *origImageEval` Pointer to `imageEvaluation`. Is created by `int createImageEvaluation()`. Only necessary, if `avoidRepetitionTrue == TRUE`, else set to `NULL`.

  – `struct databestOut`

```
struct databestOut{
  long  i_opt;
  long  j_opt;
  isometType isometriaOpt;
  long  badBlockTrue;
};
```

    * `long i_opt` i-coordinate of the chosen patch.
    * `long j_opt` j-coordinate of the chosen patch.
    * `isometryOpt` Isometry of the chosen block.
    * `badBlockTrue` `TRUE`, if no block matches the ϵ distance tolerance, and the best block has to be chosen, else `FALSE`. Only valid if using the ϵ bound (`opBound == BOUND_EPSILON`).

  The function returns an error code.

- **Usage:** In the following example, the usage of `findBestBlock` is demonstrated. We assume a correct initialized `varbest`. A detailed example how to initialize `varbest` can be found in texture_patch() (jptexturepatch.cpp).

```
#include <stdio.h>
#include "jpmyimage.h"
#include "jpmyerror.h"
#include "jpfindbest.h"
{
int errorHandler;
struct databest    *varbest;
struct databestOut *outbest;
int extSwitch;
...

if(NO_ERROR != (errorHandler =
     findBestBlock(varbest, outbest, extSwitch))){
     fprintf(stderr, "Error at findBestBlock.")
     return errorHandler;
}

long i_opt        = outbest->i_opt;
long j_opt        = outbest->j_opt;
int  isometryOpt  = outbest->isometryOpt;
int  badBlockTrue = outbest->badBlockTrue;
}
```

`int pyramidFindBestBlock(hPyramidFindBestBlock *self, databest *varbest, databestOut *outbest, int extSwitch)`

- **Description:** The function searches the next block, which has to be sampled by patch based texture synthesis. In contrast to `findBestBlock`, this is done with application of a multiresolution pyramid. Input parameters are given via `databest varbest`, output parameters are written to `databestOut outbest`. `int extSwitch` signals to the function, if the next block has to be placed at the upper border of the output image (`extSwitch = 1`), at the left border (`extSwitch = 2`) or at another place (`extSwitch = 0`). The structures `databest` and `outbest` are explained above, and have to be applied in the same manner. The handler `hPyramidFindBestBlock` has to be initialized before the first usage of `pyramidFindBestBlock` and destroyed after the last usage. This can be done with the functions `int createPyramidFindBestBlock(hPyramidFindBestBlock **self)` and `int destroyPyramidFindBestBlock(hPyramidFindBestBlock *self)`. The function returns an error code.

- **Usage:** In the following example, the usage of `pyramidFindBestBlock` is demonstrated. We assume a correct initialized `varbest`. First the handler `hPyramidFindBestBlock` is initialized. The function is called and finally the handler destroyed.

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpmyimage.h"
#include "jpfindbest.h"
{
int errorHandler;
struct databest    *varbest;
struct databestOut *outbest;
int extSwitch;
...

hPyramidFindBestBlock *hPyramidFind;

if(NO_ERROR != (errorHandler =
     createPyramidFindBestBlock(&hPyramidFind))){
     fprintf(stderr, "Unable to create hPyramidFind");
     return errorHandler;
}

if(NO_ERROR != (errorHandler =
     pyramidFindBestBlock(hPyramidFind, varbest, outbest, extSwitch))){
     fprintf(stderr, "Error at pyramidFindBestBlock.")
     return errorHandler;
}

long i_opt        = outbest->i_opt;
long j_opt        = outbest->j_opt;
int  isometryOpt  = outbest->isometryOpt;
int  badBlockTrue = outbest->badBlockTrue;


if(NO_ERROR != (errorHandler =
     destroyPyramidFindBestBlock(hPyramidFind))){
     fprintf(stderr, "Unable to destroy hPyramidFind");
     return errorHandler;
}
}
```

Functions to Evaluate Block Repetitions

The following functions were written with respect to an evaluation, how often a certain block is sampled (Section 4.2).

`int createImageEvaluation(imageEvaluation **self, long width, long height)`

- **Description:** Creates `imageEvaluation self` for an input sample of width `long width` and height `long height`. The function returns an error code.
- **Usage:** In the following, for the image `myImage testImage` an appropriate image evaluation structure `imageEvaluation testEvaluation` is created. The structure consists of a counter for each block, which is increased by each repetition of this block.

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpmyimage.h"
#include "jpfindbest.h"
{
int errorHandler;
myImage *testImage;
...

long      tmpWidth, tmpHeight;
colorMode tmpMode;

if(NO_ERROR != (errorHandler =
     myImageInfo(testImage, &tmpWidth, &tmpHeight, &tmpMode))){
     fprintf(stderr, "Error at myImageInfo.");
     return errorHandler;
}

imageEvaluation *testEvaluation;

if(NO_ERROR != (errorHandler =
     createImageEvaluation(&testEvaluation, tmpWidth, tmpHeight))){
     fprintf(stderr, "Unable to createImageEvaluation.");
     return errorHandler;
}
}
```

`int destroyImageEvaluation(imageEvaluation *self)`

- **Description:** Destroys a created structure `self` of type `imageEvaluation`. The function returns an error code.
- **Usage:** In the following, the structure `imageEvaluation testEvaluation` is destroyed.

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpfindbest.h"
{
int errorHandler;
imageEvaluation *testEvaluation;
...

if(NO_ERROR != (errorHandler = destroyImageEvaluation(testEvaluation))){
     fprintf(stderr, "Unable to destroy ImageEvaluation.");
     return errorHandler;
}
}
```

`int imageEvaluationInfo(imageEvaluation *inImageEvaluation, long *width, long *height)`

- **Description:** Writes width and height of `imageEvaluation inImageEvaluation` to `long *width`, `long *height`. The function returns an error code.
- **Usage:** In the following, the width and height of the initialized `imageEvaluation testEvaluation` is written to `long tmpWidth`, `long tmpHeight`.

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpfindbest.h"
{
int errorHandler;
imageEvaluation *testEvaluation;
...

long tmpWidth, tmpHeight;

if(NO_ERROR != (errorHandler =
     imageEvaluationInfo(testEvaluation, &tmpWidth, &tmpHeight))){
     fprintf(stderr, "Unable to info about testEvaluation.");
     return errorHandler;
}
}
```

`int writeImageEvaluation(imageEvaluation *self, long i, long j, long patchWidth, long patchHeight, long seam)`

- **Description:** Writes the evaluation entry for the patch (i,j) with patch width `long patchWidth`, patchHeight `long patchHeight` and seam `long seam`. The patch (i,j) is marked and all $patch + 2 * seam - 1$ neighbored blocks. This is done by increasing the counter for each block by one. The function returns an error code.

- **Usage:** In the following, the marking of a block with patch at (100,200) is demonstrated. We assume `imageEvaluation testEvaluation` as already initialized. Patch width and height are assumed as to be `patch`, seam as `seam`.

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpfindbest.h"
{
int errorHandler;
imageEvaluation *testEvaluation;
long patch, seam;
...

if(NO_ERROR != (errorHandler =
     writeImageEvaluation(testEvaluation, 100, 200,
                          patch, patch, seam))){
     fprintf(stderr, "Unable to write image evaluation.");
     return errorHandler;
}
}
```

int getImageEvaluation(imageEvaluation *self, long i, long j, float *evaluation)

- **Description:** Writes out the image evaluation, i.e. how often a block (i,j) is marked as repeated, to `evaluation`. The function returns an error code.

- **Usage:** We demonstrate in the following, how to get the image evaluation for the patch (100,200) of the structure `imageEvaluation testEvaluation`. The value is saved in `float tmpRepeated`.

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpfindbest.h"
{
int errorHandler;
imageEvaluation *testEvaluation;
...

float tmpRepeated;
if(NO_ERROR != (errorHandler =
     getImageEvaluation(testEvaluation, 100, 200, &tmpRepeated))){
     fprintf(stderr, "Unable to get image evaluation.");
     return errorHandler;
}
}
```

`int reduceImageEvaluation(imageEvaluation *inImageEvaluation, imageEvaluation *outImageEvaluation, int factor)`

- **Description:** Subsamples `imageEvaluation *inImageEvaluation` by `factor` and copies the result to `imageEvaluation *outImageEvaluation`. The function returns an error code.
- **Usage:** In the following, `imageEvaluation testEvaluation` is subsampled by the factor 2 and copied to `imageEvaluation subTestEvaluation`.

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpfindbest.h"
{
int errorHandler;
imageEvaluation *testEvaluation;
...

imageEvaluation *subTestEvaluation;
long tmpWidth, tmpHeight;

if(NO_ERROR != (errorHandler =
     imageEvaluationInfo(testEvaluation, &tmpWidth, &tmpHeight))){
     fprintf(stderr, "Error at imageEvaluationInfo()");
     return errorHandler;
}

if(NO_ERROR != (errorHandler =
     createImageEvaluation(&subTestEvaluation, tmpWidth/2,
                           tmpHeight/2))){
     fprintf(stderr, "Unable to createImageEvaluation.");
     return errorHandler;
}


if(NO_ERROR != (errorHandler =
     reduceImageEvaluation(testEvaluation, subTestEvaluation, 2))){
     fprintf(stderr, "Unable to reduce image evaluation.");
     return errorHandler;
}
}
```

`int getBlockImageEvaluation(imageEvaluation *inImageEvaluation, imageEvaluation *outImageEvaluation, long startI, long startJ, long width, long height)`

- **Description:** Copies a region of `imageEvaluation *inImageEvaluation` and copies it to `imageEvaluation *outImageEvaluation`. The region is characterized by its upper, left corner (startI, startJ) and its width `long width` and height `long height`. The function returns an error code.
- **Usage:** In the following example, a region, beginning at (0,0) of size 50×50 is copied from `imageEvaluation testEvaluation` to `imageEvaluation blockTestEvaluation`.

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpfindbest.h"
{
int errorHandler;
imageEvaluation *testEvaluation;
...

imageEvaluation *blockTestEvaluation;

if(NO_ERROR != (errorHandler =
     createImageEvaluation(&blockTestEvaluation, 50, 50))){
     fprintf(stderr, "Unable to createImageEvaluation.");
     return errorHandler;
}

if(NO_ERROR != (errorHandler =
     getBlockImageEvaluation(testEvaluation, blockTestEvaluation,
                              0, 0, 50, 50))){
     fprintf(stderr, "Unable to get block from testImageEvaluation.");
     return errorHandler;
}
}
```

Other

`int blockContainsColor(myImage *block, unsigned char colorValueR, unsigned char colorValueG, unsigned colorValueB, int *trueFalse)`

- **Description:** Writes `TRUE` to `int trueFalse`, if the `myImage block` contains the color specified with its RGB components (`colorValueR`, `colorValueG`, `colorValueB`), else `FALSE`. If the image is gray scale, only the R component is evaluated. The function returns an error value.
- **Usage:** In the following example is demonstrated, how to prove, if the image `myImage testImage` contains a black (0,0,0) pixel. The result is written to `int containsBlack`.

```
#include <stdio.h>
#include "jpmyerror.h"
#include "jpmyimage.h"
#include "jpfindblocks.h"
{
int errorHandler;
myImage *testImage;
...

int containsBlack;
if(NO_ERROR != (errorHandler =
     blockConainsColor(testImage, 0, 0, 0, &containsBlack))){
     fprintf(stderr, "Error at blockContainsColor()");
     return errorHandler;
}
}
```

B.1.7 jpclassifyBlocks.h, jpclassifyBlocks.cpp

`int classifyBlocks(myImage *inImage, regions *classifiedBlocks, long maxClassifiedBlocks, long patchWidth, long patchHeight, long seam, unsigned char color1, unsigned char color2)`

- **Description:** Analyzes all valid blocks of `myImage *inImage`, which has to be gray level (`colorMode BLACKWHITE`) and consist of 2 gray values `unsigned char color1` and `unsigned char color2`, what region they belong to. The following regions are defined:

  – `REGION_1` Block is located totally in the region marked with color1.

  – `REGION_2` Block is located totally in the region marked with color2.

  – `BOTH_REGIONS` Block is partially located in both regions.

  The result is written to `regions *classifiedBlocks`, an array of `regions` with `maxClassifiedBlocks` entries. The blocks consist of a patch of width `patchWidth`, height `patchHeight` and surrounding seam of size `seam`. Only blocks, which are completely in `inImage` are evaluated, and the result is written in raster scan order to `classifiedBlocks`. I.e. the first evaluated block is (`seam`, `seam`), where (x,y) defines the upper left corner of the patch of a block placed in `inImage`. The function returns an error code.

- **Usage:** In the following example, the image `myImage testImage`, consisting of the two gray values 0 and 255 (black and white) is evaluated. The result is written to `regions *testRegions`. Patch width and height are 25, seam 5.

```
#include <stdio.h>
#include <stdlib.h>
#include "jpmyerror.h"
#include "jpmyimage.h"
#include "jpclassifyBlocks.h"
{
int errorHandler;
myImage *testImage;
long patchWidth  = 25;
long patchHeight = 25;
long seam        = 5;
...

long      tmpWidth, tmpHeight;
colorMode tmpMode;
if(NO_ERROR != (errorHandler =
     myImageInfo(testImage, &tmpWidth, &tmpHeight, &tmpMode))){
     fprintf(stderr, "Error at myImageInfo.");
     return errorHandler;
}

long blocksHorizontal = tmpWidth  - patchWidth  - 2*seam + 1;
long blocksVertical   = tmpHeight - patchHeight - 2*seam + 1;
long maxBlocks = blocksHorizontal*blocksVertical;

// either maxBlocks == 0 or < 0
// in both cases testImage too small for patch and seam
if(maxBlocks < 1){
     fprintf(stderr, "maxBlocks < 1");
     return BAD_RANGE;
}

regions *testRegions;
if(NULL == (testRegions = (regions)calloc(maxBlocks, sizeof(regions)))){
     fprintf(stderr, "Unable to calloc for testRegions.");
     return BAD_MEM;
}

if(NO_ERROR != (errorHandler =
     classifyBlocks(testImage, testRegions, maxBlocks, patchWidth,
                    patchHeight, seam, 0, 255))){
     fprintf(stderr, "Unable to classify testImage.");
     return errorHandler;
}
}
```

B.1.8 jpmyfilter.h jpmyfilter.cpp

`int filterAndSubsampleImage(imgdes *inImage, imgdes *outImage, filtertype FILTER)`

- **Description:** Applies a filter and subsamples the image `imgdes *inImage` by the factor 2. Finally the result is copied to `imgdes *outImage`. The following `filtertype` can be applied to the function:
  - `NO_FILTER` No filter is applied. The image is only subsampled.
  - `MEAN_FILTER` A mean filter with the mask of 3×3 is applied to the image.

  The function returns an error code.

- **Usage:** In the following, the filtering and subsampling of `imgdes testImage` is presented. `testImage` is filtered with a mean filter. The result is subsampled by the factor 2 and copied to `imgdes subsampledTestImage`.

```
#include <stdio.h>
#include "vicdefs.h"
#include "myError.h"
#include "jpmyfilter.h"
{
int errorHandler;
imgdes *testImage;
...

// get parameters of testImage
int tmpBpp, tmpWidth, tmpHeight;
CalcularParametros(testImage, &tmpWidth, &tmpHeight, &tmpBpps);

imgdes subsampledTestImage;
if(NO_ERROR != (errorHandler =
     allocimage(&subsampledTestImage, tmpWidth/2, tmpHeight/2,
                tmpBpps))){
     fprintf(stderr, "Error alloc for subsampledTestImage.");
     return errorHandler;
}

if(NO_ERROR != (errorHandler =
     filterAndSubsampleImage(&testImage, &subsampledTestImage,
                             MEAN_FILTER))){
     fprintf(stderr, "Unable to filter and subsample
                      testImage to subsampledTestImage.")
     return errorHandler;
}
}
```

B.1.9 jpmyerror.h, jpmyerror.cpp

The following functions provide an easy possibility to print out error and info messages. It was implemented with respect to a later easier portability of the functions. All in this chapter introduced functions use internal these these two functions to print out messages.

`void myErrorMessage(char buffer[MAXERRORMSG])`

- **Description:** Prints out the error message contained in `char buffer[MAXERRORMSG]`. `MAXERRORMSG` is defined in jpmyerror.h. The function returns void.

`void myInfoMessage(char buffer[MAXERRORMSG])`

- **Description:** Prints out the info message contained in `char buffer[MAXERRORMSG]`. `MAXERRORMSG` is defined in jpmyerror.h. The function returns void.

B.1.10 jpmyrand.h, jpmyrand.cpp

`long myRand(long n)`

- **Description:** The function returns a random variable of long in the range of [0;n[.