# Practical Implementation of a Texture Synthesis Algorithm

Johanna Schmidt*

## Abstract

The goal of the project was to implement the texture synthesis developed algorithm by Wei and Levoy [1]. The main focus of this project was to find out which difficulties occur during a practical implementation of this technique, and how they can be solved.

## 1 Introduction

The purpose of texture synthesis is to receive a texture sample as input and to create a new larger one without using replication. The generated textures have a realistic appearance and do not consist of separate tiles. The concrete ideas and concepts of texture synthesis will be discussed in chapter two.

The algorithm of Wei and Levoy [1], which will be described in chapter three, follows a rather simple approach to realise texture synthesis: The new texture is created in scanline direction. At each position it is decided which pixel from the input texture is the best match for its surrounding pixels.

In the following chapters the implementation of the algorithm is described, its advantages and disadvantages are analysed and we suggest possibilities for further work.

## 2 Texture Synthesis

### 2.1 Introduction

Texture synthesis is mainly a helpful tool when texturing objects. It often is the case that a given texture sample is too small for the surface of an object. Then the sample needs to be extended in some way - and that is where texture synthesis comes in. Simple replication of the sample would cause a tiled appearance, but texture synthesis will create a new texture big enough that will still look like the original one. Figure 1 illustrates the idea.

More specific, the goal of texture synthesis is to create a new texture that

- looks different from and is larger than the original sample,
- but appears as if it has been created from the same underlying process as the original one,

*email: e0025558@stud3.tuwien.ac.at

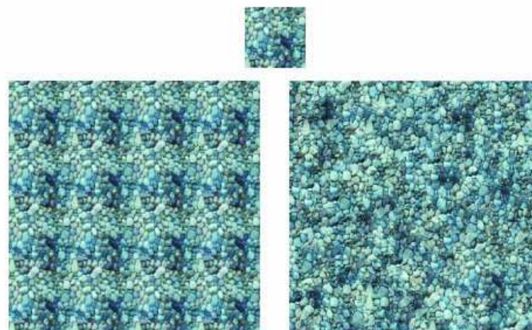

Figure 1: *This is an example to show how texture synthesis works. At the top there is the input texture sample, with which two bigger textures where created: The left one through replication and the right one using texture synthesis. The figure is taken from [2].*

both measured by the standards of human perception. This means that the success of the method can be determined by using our senses.

Jeremy S. De Bonet [2] provides another good explanation of texture synthesis: "Mathematically, the goal of texture synthesis is to develop a function F, which takes a texture image, $I_{input}$, to a new texture sample, $I_{synth}$, such that the difference between $I_{input}$ and $I_{synth}$ is above some measure of visual difference from the original, yet is texturally similar. Formally,

$$F(I_{input}) = I_{synth}$$

subject to the constraints that

$$D^*(I_{input}, I_{synth}) < T_{max}$$

and

$$V^*(I_{input}, I_{synth}) > T_{min}$$

where $D^*$ is a perceptual measure of the perceived difference of textural characteristics, and $V^*$ a measure of the perceived visual difference between the input and synthesized images. [...] The success of a synthesis technique is measured by its ability to minimize $T_{max}$ while maximizing $T_{min}$." [2]

# 3 The Algorithm by Wei and Levoy

Li-Yi Wei and Marc Levoy [1] aimed at implementing an algorithm that is easy-to-use, efficient, and which produces high quality textures. They succeeded with respect to user friendlyness, because the algorithm only needs a texture sample and an image containing white noise as input and all further calculations are executed automatically. However, to ensure efficiency and high quality for all possible textures, some adjustments on the basic algorithm have to be made.

## 3.1 The Algorithm

The Algorithm consists of the following procedures:

**Initialisation:**
A texture sample and an image containing white noise (which is of the planned output size) are needed as input. White noise is additive noise which is evenly distributed over the frequency domain.

**Processing:**
The algorithm now generates the new texture pixel by pixel based on the white noise. It visits each pixel position in scanline direction and decides each time, which pixel from the texture sample would fit best. This decision is made on the basis of the current pixel's neighbourhood. Wei and Levoy have based their algorithm on a theory that is used when creating textures with Markov Random Fields, saying that each pixel in a texture can be identified by its surrounding neighbour pixels. The algorithm by Wei and Levoy identifies the neighbourhood of the current pixel and searches for an equivalent neighbourhood in the texture sample. When it has found one, the pixel corresponding to the retrieved neighbourhood is filled in at the current position in the new texture.

### 3.1.1 The Neighbourhood

First we examine how two such neighbourhoods can be compared. Wei and Levoy use the following procedure: "The similarity of two neighborhoods $N_1$ and $N_2$ is computed according to the $L_2$ distance between them which is a sum over all pixels in the neighborhood of squared differences of pixel values at a particular position:"

$$D(N_1, N_2) = \sum_{p \, in \, N} \{(R_1(p) - R_2(p))^2 +$$

$$(G_1(p) - G_2(p))^2 + (B_1(p) - B_2(p))^2\}$$

[3]. N defines the number of neighbourhood pixel. The functions $R_i(p)$, $G_i(p)$ and $B_i(p)$ get the red, green and blue channel of the pixel number p belonging to the neighbourhood i. The smaller the result of the function D, the more similar the neighbourhoods are.

Neighbourhoods always have the shape of a square with the corresponding pixel lying in the centre. The full square of the neighbourhood is used only at the beginning where only white noise is available. As soon as the first lines of the new texture have been generated, the neighbourhood is reduced to contain only already synthesised pixels - it changes from square to L-shaped (to specify this, see figure 2). The reason why doing so ist that the algorithm does not produce the right results if the white noise is included in the calculation - this can be seen in figure 3.

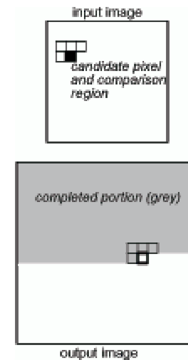


Figure 2: *The new texture is generated pixel by pixel in scanline order. On each position, the algorithm identifies the corresponding L-shaped neighbourhood and searches for an equivalent one in the texture sample. As soon as it finds one, the pixel belonging to it is copied to the current position. The figure is taken from [3].*

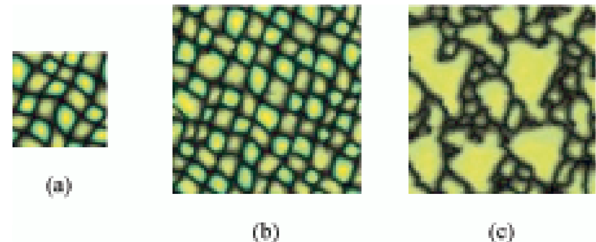


Figure 3: *(a) shows the texture sample and (b) and (c) are textures generated by Wei and Levoy's algorithm. To get (c) the white noise was included in the calculation, but not so for (b). For both (b) and (c) the same neighbourhood size (9x9) was used. It is clearly recognizeable that it is not possible to generate the right results with an algorithm that includes the white noise in its calculations. The figure is taken from [1].*

### 3.1.2 Edge Handling

Pixels on the outer edge of the texture need a special treatment, because their neighbourhood exceeds the borders of the image. Wei and Levoy suggest to treat the edges toroidally so that the neighbourhoods reach again into the texture from the opposite side.

## 3.2 Results

Despite its simple structure the algorithm produces very nice results; three of them can be viewed in figure 4.

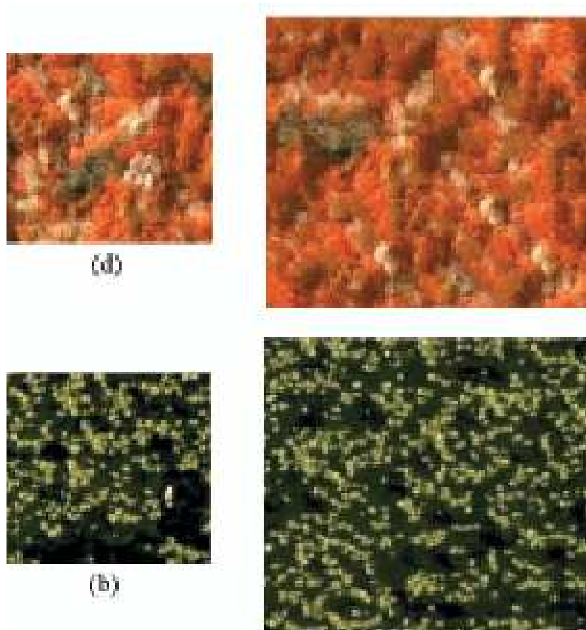


Figure 4: *On the left there are the texture samples (128x128 pixels) and on the right side there are the synthesised textures (200x200) produced with the algorithm by Wei and Levoy. The figure is taken from [1].*

## 3.3 Problems and Improvements

The examples from figure 4 match the type of textures that are very well suited for the algorithm. However, there are many textures for which only defective results can be produced. The main reason is that the algorithm tends to blur edges and corners. Unfortunately the human visual system is very sensitive to these features in an image, so these results do not look convincing (for examples see figure 5). Textures for which edges and corners are important are mainly textures with natural objects (like small flowers, blades of grass, bark, ...). Michael Ashikhmin has modified the algorithm by Wei and Levoy to eliminate the fact that natural textures cannot be processed - for further information see [3].

Another important point when discussing the algorithm is the influence of the neighbourhood. Its size influences the quality of the results. This correlation is rather simple: the larger the neighbourhood, the better the quality of the synthesized texture. Actually, good results can only be produced from a certain neighbourhood size upwards, where the exact size depends on the particular texture used. Too small neighbourhoods will lead to bad synthesising results - for illustration see figure 6. The problem is that a larger

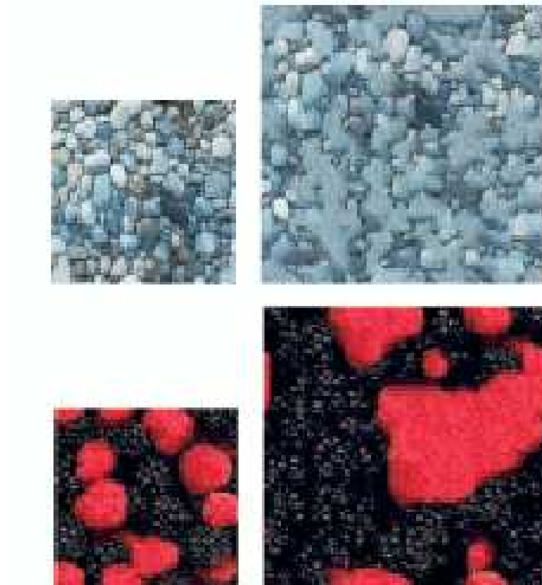

Figure 5: *On the left side there are the texture samples and on the right side there are the synthesised textures. It is easily noticeable how the algorithm blurs edges and corners, which leads to bad synthesising results. All textures have natural motifs because the effect is best visible then. The figure is taken from [1].*

neighbourhood leads to a longer runtime because significantly more pixels need to be processed.

This leads us to the main disadvantage of the algorithm

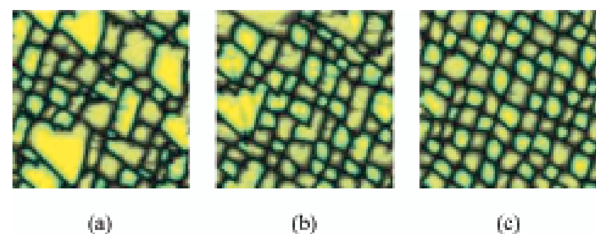


Figure 6: *These three synthesised texture show the influence of different neighbourhood sizes. For (a) the neighbourhood was 5x5 pixels, for (b) 7x7 and for (c) 9x9. It can be noticed that only from a size of 9x9 on the algorithm produces expedient results. The figure is taken from [1].*

- its long runtime. For each pixel in the new texture its neighbourhood is calculated and then compared to each neighbourhood in the texture sample, which is rather costly. One method to improve this is to implement the algorithm with Gaussian pyramids, because smaller neighbourhoods can be used. For a more specific description of this method see Wei and Levoy's paper [1]. The second improvement - which we used in our experiment - is described in the next chapter.

### 3.3.1 TSVQ

The main - and most costly - task in the algorithm is the search for the right neighbourhood. To improve this part, Wei and Levoy suggest to use TSVQ (Tree-structured vector quantization) which can be used as a data structure for efficient nearest-point queries.

"It takes a set of training vectors as input and generates a binary-tree-structured codebook. The first step is to compute the centroid of the set of training vectors and use it as the root level codeword."[1] The other vectors are divided into two groups (the first one represents the left children and the other represents the right children of the root node) and the algorithm recurses on each subtree. "The tree generated by TSVQ can be used [...] for efficient nearest-point queries. To find the nearest point of a given query vector, the tree is traversed from the root [...] by comparing the vector with the two children codewords, and then follows the one that has a closer codeword."[1]

When using TSVQ with the texture synthesis algorithm, all neighbourhoods from the texture sample can be treated as vectors. They are then used as the training data from which the corresponding binary TSVQ tree is constructed. Each neighbourhood from the new texture is also treated as a vector and the (approximate) best matching one is found by doing a best-first traversal in the TSVQ tree. The traversal always ends in a leaf, and the neighbourhood stored there is taken as the best match.

Implementing the algorithm using TSVQ brings a major acceleration in speed. For example, the generation of a particular texture took 6 minutes with the basic algorithm, which was reduced to 30 seconds using this method. However, a disadvantage using TSVQ is that not necessarily the best matching neighbourhood will be found. Unlike when using full search, not all given vectors are searched, because at each level in the tree a set of vectors is ignored. So it may be the case that the best matching neighbourhood lies in one of the ignored parts. Nevertheless, the returned vector is always very similar to the best matching one, so the results produced by the modified algorithm can be compared to those from the basic algorithm - see figure 7 for examples. The fact that not all given vectors are searched can therefore be accepted, because using full search will lead to the same long run time that occurs with the basic algorithm.

A big problem using TSVQ is the increased memory requirement. Another disadvantage when using TSVQ is that the blurring of corners and edges is even stronger than when using the basic algorithm. How strong, this depends on the structure of the texture. To counteract the effect, backtracing can be allowed in the tree. Then the algorithm is allowed to visit more than one leaf to select the best one among them.

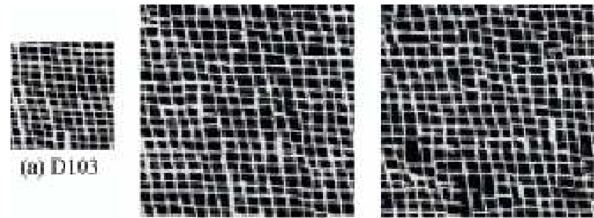


Figure 7: *On the left side there are the texture samples, in the middle there are the results produced with the basic algorithm and the textures on the right side have been synthesised using the modified algorithm with TSVQ. A little difference in quality is visible compared to the textures in the middle, but considering the major acceleration in speed these deviations can be accepted. The figure is taken from [1].*

## 3.4 Summary

The algorithm by Wei and Levoy has a rather simple structure but even so produces textures that compare favourably to results from other texture synthesis algorithms (like the ones by Heeger and Bergen [5], De Bonet [2] or Harrison [6]). With some modifications the run time can be adapted so that it would only last seconds to get a result qualitatively compareable to results from the basic algorithm. The only restriction that cannot be eliminated very easily is that texture samples with natural objects do not lead to proper results because of the characteristic of the algorithm to blur edges and corners.

# 4 Implementation - First Steps

## 4.1 A Prototype

To get an overview over the project, a prototype was implemented in Python 2.3. It only contained the basic algorithm without any improvements concerning speed. In this form the runtime turned out to be unacceptable, which was not only due to the programming language used, but also because of the structure of the algorithm. Using the basic algorithm means that for each pixel from the new texture all neighbourhoods from the texture sample must be searched, and this is a rather costly procedure.

## 4.2 Implementation in Objective-C

The first step to improve the prototype was to use another programming language. It had to be fast and allow the handling of images. Both requirements combined were found in the rendering package ART (Advanved Rendering Toolkit, [4]) by the Institut of Computer Graphics on the Technical University of Vienna. ART is implemented in Objective-C and provides modules for 2D image handling.

# 5 Usage

The executable is called Imagequilter and is used as follows:
`imagequilter SAMPLE -x X -y Y -n N`

`SAMPLE` ist the texture sample which should be a `.tiff` file (this is because we are using ART [4]). `X` and `Y` define the size of the output texture, both values are independent from each other and independent from the size of the texture sample. `N` is the size of the neighbourhood which should always be an odd number. That is because the neighbourhood must always be a square which can have the corresponding pixel in the centre. A greater value of `N` leads to a longer computation time. Using an input sample with wrong parameters will lead to an error message.

The algorithm then creates the new texture and stores it as a `.quilt.tiff` file at the same place as the texture sample.

# 6 The Algorithm

In this chapter the implementation of the main parts of the algorithm is described. The goal was to implement the algorithm by Wei and Levoy [1] using TSVQ to improve the run time while even so getting results of good quality.

At first proper data structures for storing the neighbourhood vectors and the TSVQ tree had to be found which is described in chapter 6.1 and 6.2. In 6.2 we discuss the construction of the TSVQ tree.

In chapter 6.3 the search function where the right pixels are found for the current positions in the new texture is explained.

## 6.1 Initialisation

The first step after loading the texture sample is that all neighbourhoods within the image need to be determined. They are stored in an array, because in this way they can be treated as vectors later on. An array element represents a neighbourhood and contains all belonging neighbourhood pixels. When initialising the array it is necessary to know how many pixels there will be. This cannot be done calculating `neighbourhoodsize`$^2$, because not all neighbourhood pixels are used. Neighbourhoods are L-shaped, so allocating memory for the whole square would be a waste. It was necessary to determine a formula that will calculate the size for the L-shape:

$$\frac{(N+1)*(N-1)}{2}*3$$

The fraction represents the number of neighbourhood pixels and it is multiplied by 3 because the RGB channels of each pixel are stored separately. See figure 8 for further information.

Now that it has been determined how many pixels each neighbourhood will have, it is necessary to find out how many neighbourhoods (array elements) there will be. Each pixel in the sample has a neighbourhood, but not all of them will be used for texture synthesis. Neighbourhoods that exceed the borders of the image should not be included in the calculation. This implies that the number cannot be determined by `X*Y`, but by:

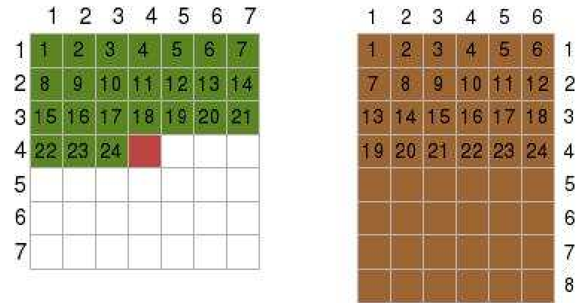


Figure 8: *On the left side it is shown how a neighbourhood of 7x7 pixels looks like - a square with the corresponding pixel in the centre. Only the pixels that are marked green are used for calculations because these only contain already synthesised values. The graphic on the right side is to show the evidence that the formula mentioned in chapter 6.1 calculates the right number of neighbourhood pixels.*

$$(Y-\frac{N-1}{2})*(X-(N-1))$$

In x-direction the neighbourhoods exceed the borders of the image on both sides, so the complete neighbourhood size must be subtracted from X. In y-direction only the neighbourhoods at the top of the image exceed the borders because of L-shape. So only half of the neighbourhood size is subtracted from Y.

Now the array is ready to be filled with neighbourhoods from the texture sample. This is done the following way (i is the number of the current neighbourhood): In [i][0] the red channel, in [i][1] the green channel and in [i][2] the blue channel of the first neighbourhood pixels is stored. [i][3] contains the red channel of the second pixel, and so on. In the last array entry the current index of the loop is stored to retrieve the pixel that corresponds to the neighbourhood. There is a second array where the x- and y-coordinates of this pixel are stored at the same index.

The last thing to do during initialisation is to create a future output image and to fill it with white noise.

## 6.2 Creation of the TSVQ Tree

A data structure was constructed to represent the nodes in the tree. It consists of the following components:

- a pointer to the left and a pointer to the right child
- a pointer to the parent treeNode construct
- a vector (array) `data` where the neighbourhood vector belonging to the node is stored
- an array `vectors`, where the neighbourhood vectors are stored, that should be devided among the node's children
- a variable `count` where the number of neighbourhoods in vectors is stored
- a variable `dim` where the dimension of the neighbourhood vectors is stored
- a variable `depth` which states how deep the node lies in the tree (root node: depth=0)
- a variable `designed` which is 1 when the node has been initialised (is only needed when the tree is created)

First of all the root node is initialised. In its array `data` a zero vector and in its array `vectors` all available neighbourhood vectors are stored. In the method `makeNode` the left and the right child of the root node are created and the neighbourhood vectors are divided among them. This happens as follows:

1. From all neighbourhood vectors two are chosen: The one which is nearest and the one which is farthest from the root's `data` vector (both measured with the $L_2$-distance). The first vector is stored in the left child's, the second in the right child's `data` vector.
2. Now the remaining neighbourhood vectors are devided among the children. For each one we calculate, whether its $L_2$-distance to the left child or to the right child is smaller. As the case may be, the neighbourhood vector is stored in the array `vectors` of the left or the right child.

After this method call the root node has two children and the neighbourhood vectors are devided among them. The procedure is iterated for all other children until the whole tree has been created.

## 6.3 Search for suitable Neighbourhoods

Now the actual texture synthesis can start. On each position of the new texture it is decided which pixel from the texture sample should be copied there.

To start the process the neighbourhood of the current pixel is stored in an array. Now all pixels of the new texture have to be processed, so there are also neighbourhoods which exceed the image borders. This problem is solved by treating the image toroidally so that the neighbourhoods can reach again into the texture from the opposite side.

The search is done as follows:

- At each node the $L_2$-distance of the vector to the left and the right child's data vector is calculated. Whichever $L_2$-distance is smaller, the search is continued to the left or the right side.
- The traversal will end when a leaf is reached. The data vector stored there is returned as the search result.
- If the $L_2$-distance is 0 at some node in the tree, the search will be aborted and the current node's data vector will be returned as search result, because in this case the vectors are equal.

When the search has finished, not the whole result vector, but only its last digit is returned. As we explained in chapter 6.1 this is enough to find out the coordinates of the corresponding pixel.

# 7 Results

The result section figures in this paper all show examples from textures that have been created by our implementation of the algorithm.

Figure 16 and 17 are useful samples to show how the algorithm tends to blur small artefacts in the images. In addition, this disadvantage gets even worse when TSVQ is used.

The last image is a photo of natural objects, and it is clearly visible that the algorithm has some problems to handle them properly.

# 8 Summary

The goal of the project was to implement the algorithm of Wei and Levoy and to find out which difficulties occur when doing so.

The first problem occured when only using the basic algorithm. Then the results had the best quality, but the execution times were unacceptable. The key problem was the structure of the main algorithm.

During initialisation some considerations had to be made that were not mentioned in the paper by Wei and Levoy. A proper data structure for the neighbourhoods was needed, and formulas for calculating the right numbers of neighbourhoods and neighbourhood pixels had to be derived. It was also necessary to find a proper way of visiting all neighbourhood pixels using two loops, because not all pixels of the square were valid (neighbourhoods are L-shaped).

The essential goal was to make the algorithm faster, and

this succeeded using TSVQ. It was not really difficult to create data structures for the nodes and to create the tree, but the main problem with TSVQ was the increased memory requirement (see also next chapter).

An effect that occured because of using TSVQ was increased noise appearing in result textures, especially when using natural textures. To soften this effect the algorithm was allowed to search for two more results and to choose from these three. The difference to the first result is that the new search processes have some random steps at the beginning. They also do not start from the root node but from its children - one from the left, the other from the right one. From the three results the best one is chosen. The additional expenses are barely noticeable because the search in the binary tree is quick enough. The selection of the best results does not slow down the algorithm noticeably as well, so there was no need for further optimization in this area.

# 9 Future Work

The runtime and the results for many types of input images are already very good. However, the algorithm has some points that need improvement.

One of them is the memory requirement. The creation of the TSVQ tree needs such a large amount of memory that it can even lead to the process being killed by the system. It depends on the size of the texture sample and the size of the neighbourhood - a sample of 200x300 pixels together with a neighbourhood of 25x25 pixels will lead to problems. Two experiments were been attempted to solve the problem: To eliminate all duplicates from the array of neighbourhoods, or either to build only half of the TSVQ tree - but both did not work. The first took too long, and the second produced inferior results. The memory requirement for large texure samples is still a problem which should be a subject to further work.

Another problem is the noise which is visible to a greater or lesser extend depending on the texture. We have tried to eliminate it by allowing the algorithm to find two additional results, but it only has decreased it, but did not completely remove it. This is another point that should be improved.

The last point is that the algorithm can be extended with the method by Michael Ashikhmin [3] so that it can handle natural textures without blurring the edges.

# 10 References


[1] LI-YI WEI and MARC LEVOY
2000. Fast texture synthesis Using Tree-Structured Vector Quantization. Siggraph 2000, Computer Graphics Proceedings, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, [479-488].

[2] JEREMY S. DE BONET
1997. Multiresolution Sampling Procedure for Analysis and Synthesis of Texture Images. Computer Graphics Vol. 31, Annual Conference Series, [361-368].

[3] ASHIKHMIN, MICHAEL
2001. Synthesizing Natural Textures. Proceedings of 2001 ACM Symposium on Interactive 3D Graphics, Research Triangle Park, NorthCarolina, March 19-21, [217-226].

[4] ALEXANDER WILKIE, ROBERT F. TOBLER
Institute of Computer Graphics, Technical University of Vienna. ART (Advanced Rendering Toolkit). http://www.artoolkit.org

[5] HEEGER, D.J., and BERGEN, J.R.
1995. Pyramid-based texture analysis/ synthesis. In SIGGRAPH, [229-238].

[6] HARRISON, P.
2001. A non-hierarchical procedure for re-synthesis of complex textures. In WSCG 2001 Conference Proceedings, V. Skala, Ed.


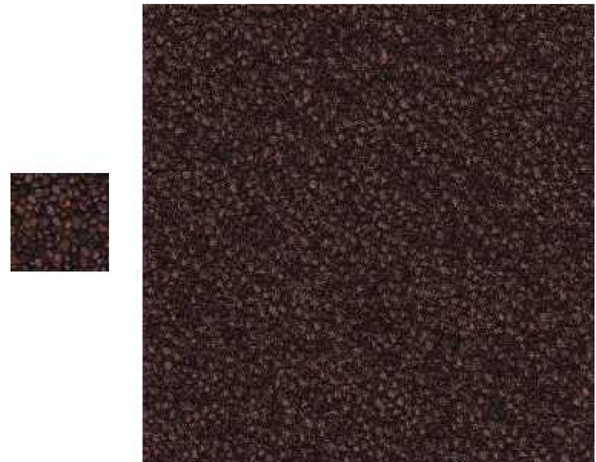

Figure 9: *The texture sample on the left is 88x88 pixels and the new texture is 400x400 pixels. The sample has a structure which suits the algorithm very well.*

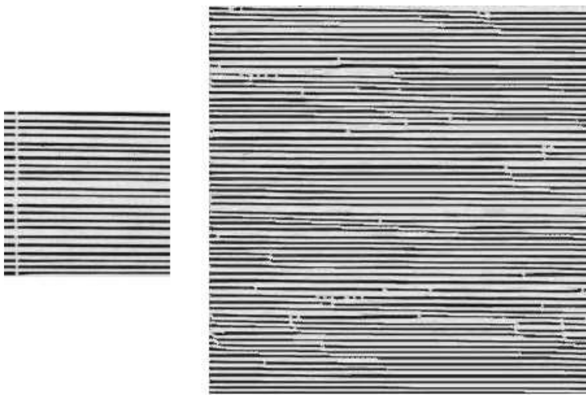

Figure 10: *The texture sample on the left is 177x177 pixels and the result is 400x400 pixels. The artefacts in the new texture arise from the white line in the sample.*

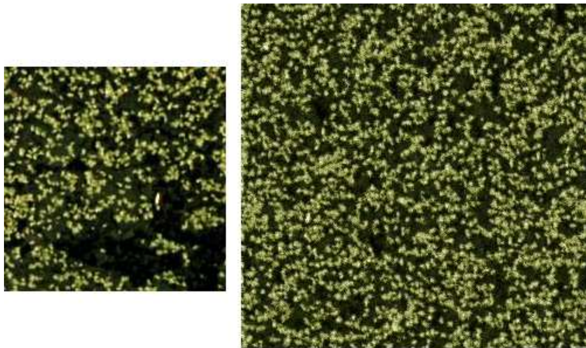

Figure 11: *The texture sample on the left is 264x264 pixels and the new texture is 400x400x pixels. The texture has a structure which the algorithm can handle very well.*

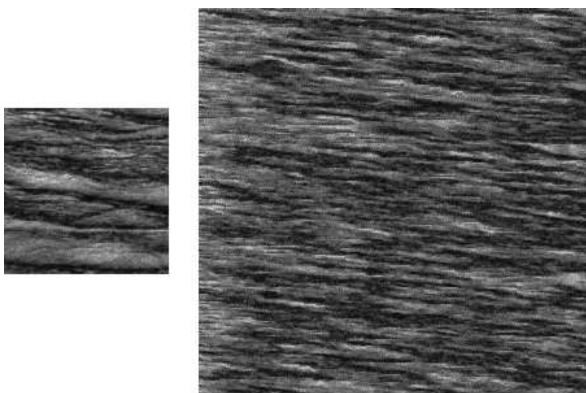

Figure 12: *The texture sample on the left is 177x177 pixels and the new texture is 400x400 pixels. The sample has a structure which suits the algorithm, although a little noise is visible.*

Figure 13: *The texture sample on the left is 88x88 pixels and the new texture is 400x400 pixels. The results is rather good, although little noise is visible.*

Figure 14: *The texture sample on the left is 264x264 pixels and the new texture is 400x400 pixels. This is an example for a natural texture, where the algorithms tends to blur edges and corners.*

Figure 15: *The texture sample on the left is 277x190 pixels and the new texture is 400x400 pixels. This sample is taken from a photograph, but although the algorithm shows promise, some noise and blurring is visible.*