Volume xx (200y), Number z, pp. 1–8

Patchmatch for Texture Synthesis

Daniel Prieler, Stefan Jeschke (Advisor)

Vienna University of Technology, Austria

1. Introduction and Related Literature

This paper is a report about a project practical course ("Projektpraktikum") done at the Institute for Computer Graphics and Algorithms of the Vienna University of Technology. Advisor: Stefan Jeschke. The goal of this course was to implement a texture synthesis algorithm and analyze its performance and visual quality. Two papers form the theoretical foundation of this project: "Texture Optimization for Example-based Synthesis" by Kwatra et al [KEBK05] and "Patchmatch: A randomized Correspondence Algorithm for Structural Image Editing" by Barnes et al [BSFG09].

1.1. Texture Optimization for Example-based Synthesis

The technique proposed in this paper [KEBK05] works on the whole image at once. It defines a metric for similarity between two images, which is called texture energy. The algorithm then tries to minimize it iteratively. In short, this energy compares local neighborhoods of the input texture to patches in the synthesized image (the terms patches and neighborhoods are used interchangeably). In contrast to region-growing methods this energy is a global metric for the whole texture.

The texture energy is defined formally as follows:

$$E_t(\mathbf{x}; \{\mathbf{z}_p\}) = \sum_{p \in X^\dagger} \|\mathbf{x}_p - \mathbf{z}_p\|^2 \qquad (1)$$

In this notion X is the set of all pixels of the synthesized texture and Z would be the example texture. The lower case x and z are the vectors of the pixels of X and Z respectively. x_p and z_p are vectors of length w, which contain some pixels of x and z. These correspond to little regions in an image. The index p indicates the pixel p which is the center of the current patch (or region). The similarity metric works as follows: Take two patches - one of the synthesized and one of the input texture - and take the Euclidean norm of the two. (Basically the distance between the two vectors consisting of w pixels) Do this for all patches of the synthesized texture and sum up all the distances. It is not necessary to perform these steps for every pixel, (not every pixel is the center of a patch). Kwatra et al claimed that using only every $\frac{w}{4}$ pixel increases both visual quality and performance. ($X^\dagger$ is the set of pixels of the output texture which are centers of those neighborhoods which are taken into account with the energy function) Image synthesis can be carried out by minimizing this texture energy. At first an initial guess of the output texture has to be made; each neighborhood in the output texture is assigned randomly to a neighborhood of the sample texture. After this initialization step the algorithm progresses iteratively:

- The new output texture ($\mathbf{x}$) is determined by taking all neighborhoods of the sample texture ($\mathbf{z}_p$) (which are currently chosen) and combine them so that the texture energy is minimized.
- The new neighborhoods of the sample texture are computed by finding the nearest neighbor of every patch in the output texture ($\mathbf{x}_p$) in the sample texture.

These steps are repeated until a fixed number of times or if the neighborhood of the sample texture does not change anymore. The idea of not taking every pixel of the output texture into account has two positive effects: On the one hand it reduces the number of computations, since the nearest neighbor search (which is quite expensive) is not used that often. On the other hand the quality of the synthesized texture is greater as well: By using fewer patches, each pixel is influenced by a smaller number of neighborhoods which produces better results when areas overlap that do not fit to each other. As a result these regions are less blurry.
In order to improve the convergence of the optimization process one can adapt the energy metric 1 of the algorithm: The Euclidean norm performs similar to a least square approximation of the synthesized texture. Fitting a line in an over-determined system is quite prone to outlier. Similarly a single non-fitting pixel can deteriorate the speed of convergence significantly. To address this problem a different distance metric can be used: Coleman et al [CHK*80] suggested a method called "Iteratively reweighted least squares" (IRLS) which is used by Kwatra et al. This adds a weight-

term wp to every item in the sum of the texture energy. These weight terms are computed for every neighborhood with formula 2:

$$\omega_p = \|\mathbf{x}_p - \mathbf{z}_p\|^{r-2} \tag{2}$$

$$E_t(\mathbf{x}; \{\mathbf{z}_p\}) = \sum_{p \in X^\dagger} \omega_p \|\mathbf{x}_p - \mathbf{z}_p\|^2 \tag{3}$$

A possible value for r would be 0.8 according to Kwatra. The w_p terms are obtained in iteration n and used in iteration $n+1$ in equation 3. The energy function can be further tweaked by weighting each pixel of a patch with respect to its distance to the neighborhood center with a Gaussian bell curve. Kwatra et al propose a multi-resolution synthesis pipeline for further optimizations of the texture generation process: The first step would be to synthesize a coarse texture (a texture which is bigger than the sample texture but smaller than the desired output texture). This intermediate result gets upsampled to a higher resolution and is then used as the input for the synthesis algorithm. Of course this represents a much better estimation as a random assignment and leads to a better convergence. These steps can be repeated until the final resolution is achieved. At each resolution the synthesis can be run with different neighborhood sizes w (first rather big patches - e.g 32x32 - and successively smaller ones - down to 8x8). The big advantage of this approach is that large patterns and structures of the sample texture are captured by the algorithm.
The benefits of having a general minimization algorithm as a synthesis algorithm become clear in an application of this technique: In order to implement an additional feature labeled "Controllable Synthesis" the texture energy metric is the only element (in essence) that has to be extended. The goal thereof is that the output texture will be similar to the appearance of the sample texture and satisfy some other criterion as well. In particular they experimented with flow fields: Instead of generating only one image, a whole sequence of textures is created which represents an animated texture whose flow corresponds to the input flow field.
In general this texture optimization algorithm produces high quality images. Both fine grained patterns and large structures are synthesized successfully. The additional method of controllable synthesis underlines the flexibility of this optimization technique, which can be applied to a wide range of applications. The computational complexity is mainly dependent on the nearest neighbor search, which is done $\frac{M}{w^2}$ times per iteration. (M denotes the number of pixels of the synthesized texture and w the size of the neighborhood) The complexity of one NN-search call depends on w^2. So even relatively small output textures (256^2) take minutes on the CPU.

1.2. Patchmatch: A randomized Correspondence Algorithm

The central idea behind Patchmatch [BSFG09] is an approximate nearest neighbor algorithm. This outperforms the brute force NN-search and can be applied to many image editing problems. Besides explaining this technique Barnes et al analyze certain properties of images and try to exploit those for image editing tools. Similar to texture optimization, (and as the name suggests) Patchmatch is a patch based algorithm that works on neighborhoods in the output texture which refer to a neighborhood in the sample. This is advantageous since the number of NN-searches is minimized. A huge advantage of this algorithm is that it is aware of the natural structure of images. Usually images have large contiguous areas, so instead of processing each pixel independently Patchmatch tries to propagate good matches throughout the neighborhood.
The Patchmatch NN-algorithm works as follows: The general idea is to create a nearest neighbor field (NNF), which stores offsets (like pointers) from one image to the other: Given a patch-center coordinate in the output texture a and its corresponding patch-center coordinate in the sample texture b, the offset would be $b-a$. These offset are stored in the aforementioned NNF, which is basically a vector-field with the same size as the output image A. The NN-algorithm itself works on the NNF and tries to optimize it according to a certain distance metric. Patchmatch consists of 3 parts: Initialization, Propagation and Random Search. (See figure 1; the latter two are repeated until some halting criterion)

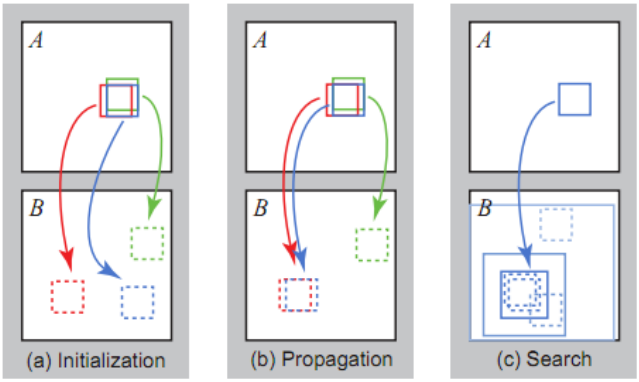

Figure 1: *The three stages of the Patchmatch NN-search. [BSFG09]*

In general the **Initialization** works like the bootstrap in the Kwatra paper: Each patch in the output texture is assigned to a neighborhood in the sample image. These evenly distributed random offsets ensure that at least some patches have a good offset from the beginning which can be distributed over the rest of the image in the next steps. The original Patchmatch works sequentially for every pixel: on odd iterations it starts on the top left corner $(0,0)$ and

proceeds pixel by pixel to the bottom right corner. On even iterations it goes the other way round from the bottom right to the top. For every pixel it executes the **Propagation** and the **Random Search** steps:

Propagation tries to distribute good offsets over the image: therefore the offsets of 2 neighbors are compared to the old offset of the pixel. So at the coordinates (x,y) the neighbors at $(x-1,y)$ and $(x,y-1)$ are checked: The offsets of the neighbors are added to the current position (x,y) to determine a candidate neighborhood in the sample texture. Then the distance between this neighborhood and the patch in the output-texture centered about (x,y) is computed. Now, the offset which has produced the smallest distance is taken and stored in the NNF for the position (x,y). Thus, if some patch center to the left (on odd iterations) has a perfect matching its offset is propagated to its neighbors, creating a coherent region. On subsequent iterations the Propagation phase is once (on odd iterations) executed from the left upper corner down to the right lower corner of the image. On even iterations it reverses its direction and starts at the bottom right corner. Therefore it checks the neighbors at position $(x+1,y)$ and $(x,y+1)$. A possible distance function would be the Euclidean distance between the pixels of one patch of the sample image to the pixels of a patch of the synthesized texture.

In order to not get stuck into minima the algorithm has a **Random Search** phase: After analyzing the neighborhood of a pixel, the found offset is compared to random selected offsets which are computed with equation 4.

$$\mathbf{u}_i = \mathbf{v}_0 + w\alpha^i \mathbf{R}_i \tag{4}$$

The best offset of the Propagation phase (v_0) is used as a point of origin for computing some random pointers into the sample texture: $\mathbf{u}_i$ is the new candidate vector, w denotes the maximal image dimension and $\mathbf{R}$ is a random vector between $[-1,-1]$ and $[1,1]$ and α is a miniaturization factor (typically 0.5). In other words it chooses a candidate offset by looking for a random vector in starting at offset v_0 in a search window which has the size of the whole sample-image (at the beginning) and halves every iteration. This is done until the search window is below one pixel. Like in the propagation phase, the candidate offsets are tested with the distance function against the original offset.
These steps lead to fast convergence of the NNF; according to Barnes et al 4-5 iterations suffice in most cases. The reason for this is that good offsets are passed on to other pixels across the image. The only prerequisite is that there are at least some good offsets after the initialization step. Even though it is very unlikely that a specific pixel gets a good offset, there is a good chance that at least one patch receives a perfect match: Given two images A and B (size of $A =$ size of $B = M$), the probability that one specific patch has a good offset would be $\frac{1}{M}$. Therefore the chance that no single pixel out of M has a good offset would be $(1-\frac{1}{M})^M$. So for large M the probability that at least one pixel is assigned to a good offset is $1-(1-\frac{1}{M})^M$, which converges to $1-\frac{1}{e}$, which is approximately 63.2%. Furthermore the probability of at least one correct assignment during the Random Search phase is $1-(1-\frac{C}{M})^M$ (where C denotes the number of pixels around a patch center) which is relatively high. The complexity is substantially smaller than with a naïve brute force NNF: Propagation is $O(M*p^2)$ (where p is the width of a patch), random search is $O(M*\lg(M)*p^2)$. This is a massive speedup compared to $O(p*M^2)$, especially with bigger images (big values for M).
This approximation NNF-algorithm performs reasonably well for real-world images. Problems which do not need a perfect nearest neighbor match such as image retargeting, reshuffling and image completion are ideal areas of application for this technique. Barnes et al mention a GPU implementation which executes the algorithm in parallel. Of course certain adjustments have to be done since the algorithm, especially the Propagation phase is a sequential problem. According to them the implementation running on a graphics card is about 7 times faster than on the CPU, which on the other hand should be about 20 times faster than previous approaches.

2. Implementation

The overall aim of the project was to implement the Patchmatch nearest neighbor search (optionally like the CPU-implementation proposed in the paper) as a parallel algorithm on the GPU. It then should be used for texture synthesis, similar to the texture optimization method proposed by Kwatra. Due to the fact that the randomized NN-search is much more efficient than the brute force implementation used by Kwatra high performance gains are expected. But in contrast to the global texture optimization technique the Patchmatch algorithm does not optimize the whole image at once but only on patch level. This will produce certain artifacts and in general a lower visual quality.
The major problem implementing the Patchmatch algorithm on the GPU was to parallelize the Propagation phase: In the Patchmatch NN-search each pixel tries to improve its offset by looking at its two neighbors. After finding the optimal solution in this tiny neighborhood the next pixel can rely on the fact that its predecessor has the best offset it can get (at least in this iteration in the Propagation phase). If run as a pixel shader, every pixel gets processed in parallel. Therefore, a pixel can only evaluate its neighbors offset of the last iteration. But the problem is that after the iteration the neighbors may already have a different offset. So the offsets are not necessarily coherent over a certain region. The Random Search algorithm is not as bad, since it does not rely on the closest neighborhood. This phase may determine an offset which is relatively far away from the current pixel, so it is

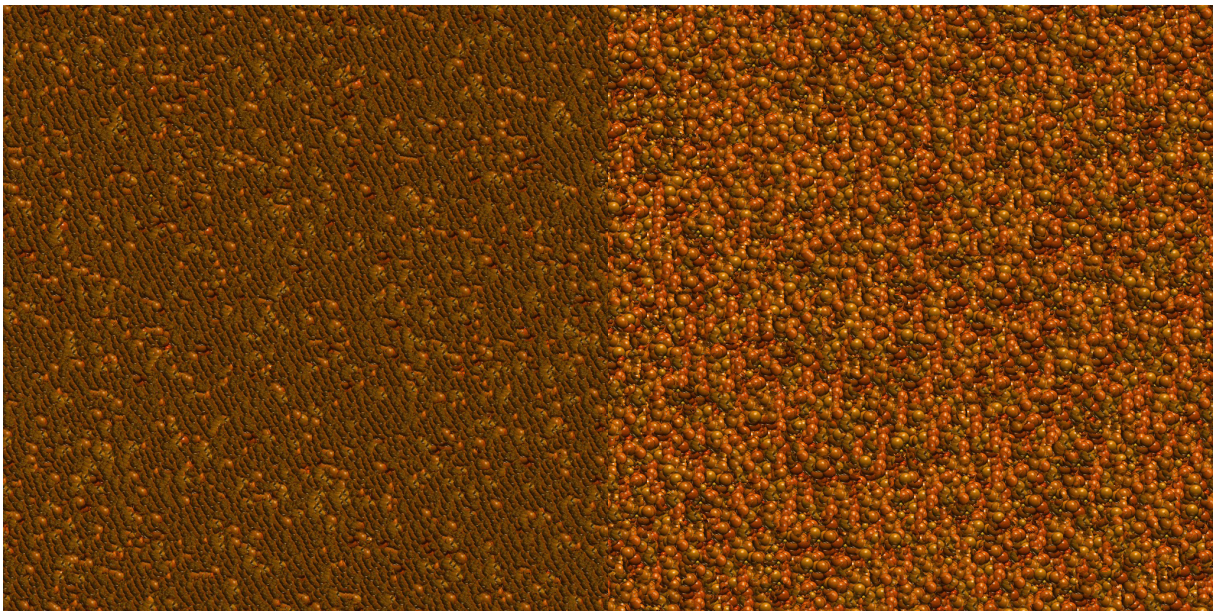

Figure 2: *Left side whithout seed-initialization, right side with seeds*

no problem if they do not share the same offset after a specific iteration.
A possible approach is to extend the search space during the propagation phase for every pixel. Instead of just checking offsets in a certain direction (e.g. only left and upper neighbor pixel) a whole neighborhood around every pixel is analyzed. Barnes et al suggested using the jump flood scheme [RT06]. Instead of testing every pixel in a - for example 8x8 - neighborhood only certain pixels depending on their logarithmically distance from the center are checked. This ensures that the closest neighborhood is inspected thoroughly; the outer regions are sampled in a more sparse way. Of course this method does not eliminate the aforementioned problem of the parallel Propagation phase but it ensures that a quite large area is checked in each iteration. This is necessary for the actual propagation of good offsets: In the sequential case a perfect offset can be carried over the whole image in a single iteration. In this parallel approach a pixel can adopt a possible perfect offset only if it is within its reach - the search space of the propagation phase. (Of course it could randomly hit a perfect match during the random search phase but this is not guaranteed.) So the worst case propagation speed of a perfect offset would be the width of the search space per iteration.
During evaluating the implementation a certain behavior could be observed: Even though the output texture was initialized with random offsets covering the whole sample image, the algorithm got stuck at some parts of the input texture. This effect increases if the sample image has relatively large contiguous areas: (See image 2) In this case many different patches of the output texture tend to point to those areas in order to minimize their own distance. Since the global distance of the whole image is not taken into account this leads to very bad synthesis results.
Instead of assigning only random offsets to the nearest neighbor field some seeds were planted into the synthesized texture: In this case seeds are pixel that are fixed and should not move during the synthesis process. These seeds are also distributed randomly across the image (typically with a probability of 0.001) During the NN-search the seeds are guaranteed to contribute to the patch distance with a negative number. Therefore patches that contain seeds have a very low distance and pass on their offsets. The overall effect of this method is that there are fixed points in the output texture and the rest of the image has to adjust itself them. The density of seeds affects the quality of the synthesized texture and can be tweaked according to the size of the patterns in the input image. This procedure ensures (probabilistically) that a wide range of different areas of the input sample are in the output texture.

3. Results

The overall visual quality of the synthesized textures is dependent on the size of the patterns contained in the input textures. This method tries to form patches (contiguous regions) and propagate their offsets to other pixels. In contrast to Texture Optimization this technique does not work on the global image: The neighborhoods are not aware of their neighbors and do not take them into account during the iterations.
Figure 2 demonstrates the difference between synthesis with (right) and without (left) seeds: In the left image the algo-

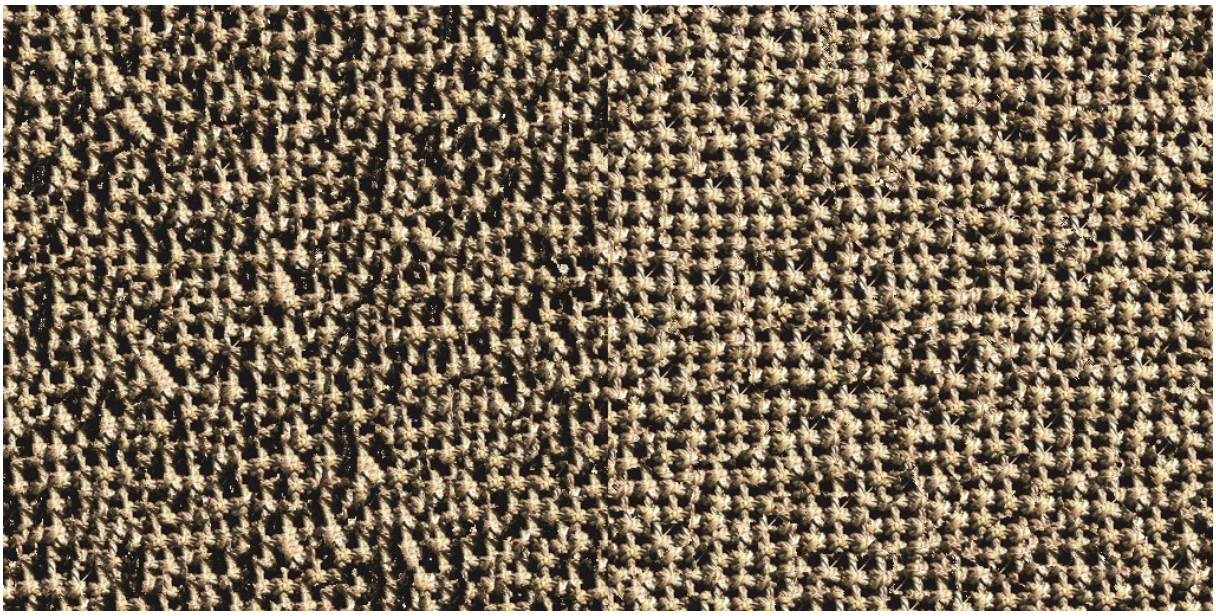

Figure 3: *meshwork; left without seeds, right with seed-initialization*

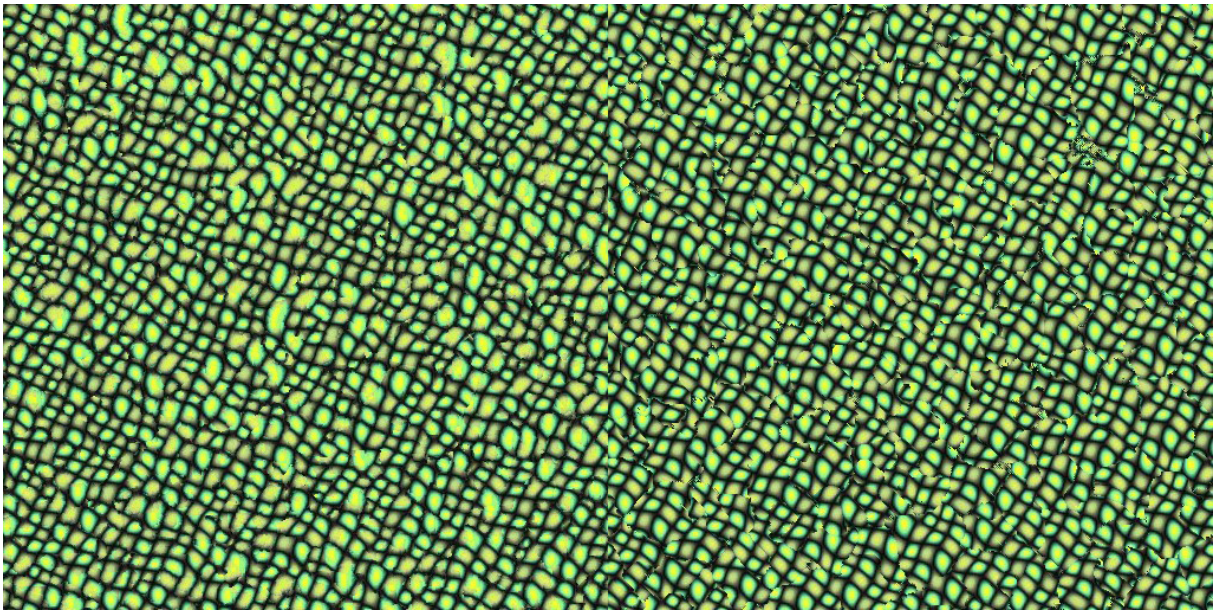

Figure 4: *levoy-pattern; hard seams between patches defined by random seeds on the right side*

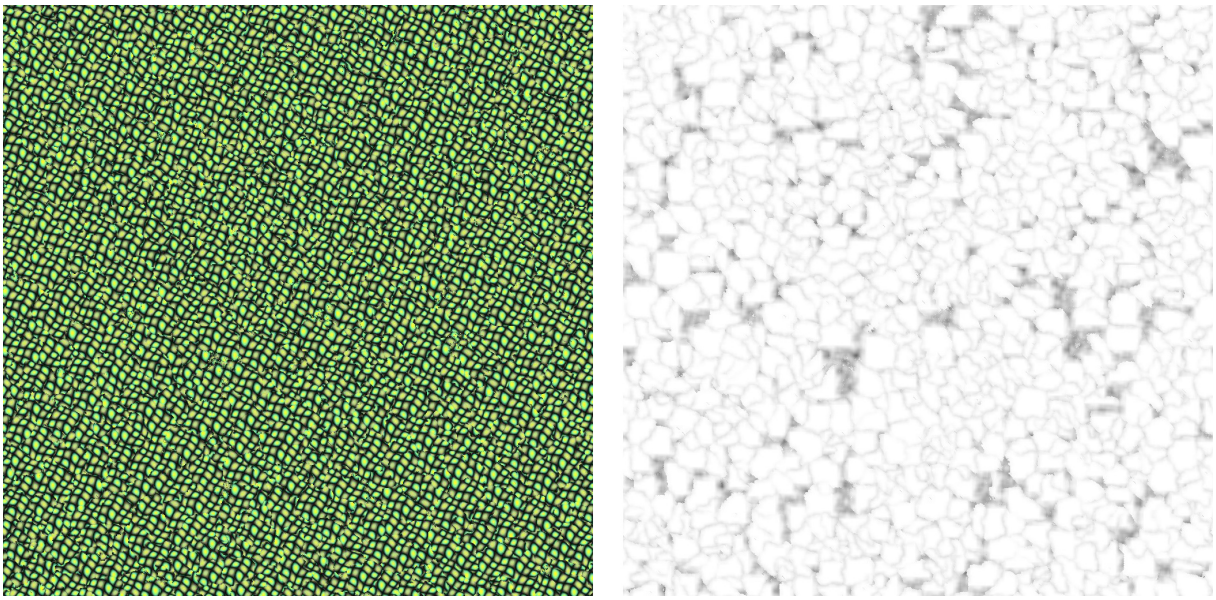

Figure 5: *The right picture is an error map of the left synthesized texture.*

rithm found a certain patch in the sample image which fitted very well to the majority of the output-patches. Once stuck in this state, it won't try other patches because its distance is already vanishingly small. On the right side the synthesized texture was initialized with about 1100 seeds (at a resolution of 1024^2) which produced a drastically better result. In this case the algorithm was run twice as often on the left side as on the right. This effect is even greater with input images with large contiguous areas. Textures with a regular pattern benefit from the fixed seeds as well. This is demonstrated in figure 3 which consists of a regular meshwork.

The drawback of this technique is that the patches around a seed are fixed before the actual synthesis procedure and will not change their position. This leads to regions which do not fit to each other and seams between adjacent patches. Figure 5 illustrates this problem: White areas in the error map are pixel regions which fit well to each other (= the distance of this patch is quite small). The grey and darker pixel indicate a greater patch distance, mismatches between neighborhoods. Thus, the algorithm sometimes produces slightly better results without seed initialization. Figure 4 is an example for this effect: The borders between neighborhoods are clearly visible and produce unpleasent seams in the output texture (right side). Without seeds, the image has an overall blurry impression but does not have sharp borders between patches.
In figure 6 the Patchmatch synthesis is compared to an implementation of Kwatra's Texture Optimization. Both images (512^2) are created using a 64^2 input sample. This particular sample texture is quite hard for patch-based synthesis algorithms due to the distinct lines which should be consistent over the whole image. Kwatra's method produces blurred areas where those lines do not fit together. The texture generated with the Patchmatch algorithm has clearly visible border areas between patches with incorrect pixels. The visual quality of Kwatra's method is in general better: The synthesized textures have a more consistent appearance and the neighborhoods blend into each other much better.

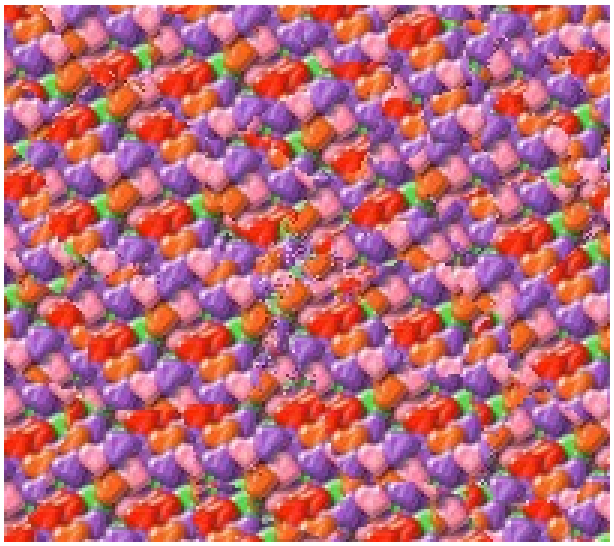

Figure 7: *incorrect pixels between patches*

When using the Patchmatch algorithm, the borders between patches often have certain pixels that converge very

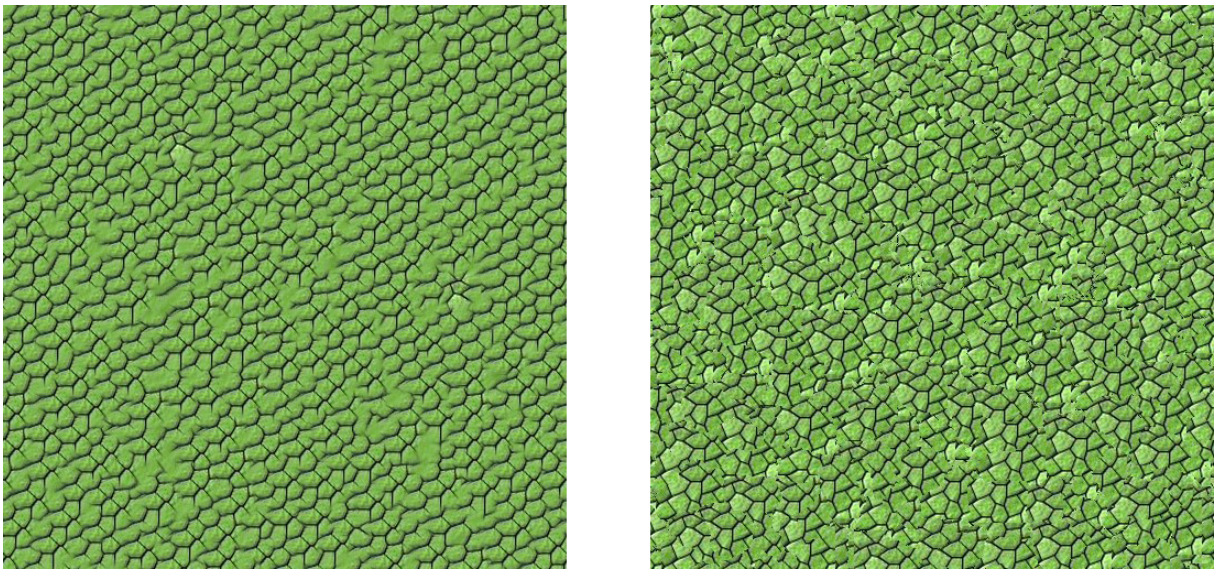

Figure 6: *Comparison of an implementation of Kwatra's method (left) with Patchmatch-synthesis (right)*

slowly. Due to the parallel implementation, a pixel near the border of two patches changes its offset in each iteration: In one iteration it may adopt the offset of a neighbor x and in the next iteration (in the meantime the other pixels have changed as well) from a different neighbor. Some of the neighbors have a similar behavior, which causes the general low convergence in those areas. Figure 7 shows a detail from a hearts-pattern. The wrong pixels in the center are clearly visible and are corrected very slowly (this is the result after 30 iterations).

The major advantage of this texture generation method is the execution speed. Especially with bigger textures it is much faster than the Kwatra implementation. Synthesizing a 512^2 texture takes about 15.8 seconds with the Kwatra-method and about 10.3 seconds with the Patchmatch (about 25 iterations). The difference gets bigger with the size of the synthesized textures: One Patchmatch iteration for a 1024^2 texture takes about 0.8 seconds. Without the seed-initialization the shader execution time is slightly better (about 0.7 seconds). In most cases the nearest neighbor field converges after 20-30 iterations. A 1024^2 texture takes 52.5 seconds with Texture Optimization, so at this resolution the Patchmatch is approximately twice as fast. The algorithms were tested on a Nvidia GTX 570.

4. Texture Tool

The texture tool which was used for synthesizing the shown images is an application which starts as a console application (see figure 9). There the name of the input file has to be specified. (The file itself should be in the /Media/ folder of the application) Afterwards it can be specified if the texture should be synthesized with the Patchmatch algorithm or with an implementation of Kwatra's Texture Optimization. Then the application sets up a DirectX context and opens an additional window (see image 10). The main frame of this window is used to display a texture (output texture by default); on the right there are 5 GUI-elements: The slider is used to change the tiling-rate. If it has a different value than 1, the current texture is tiled according to the value of the slider. If the checkbox below is activated the image which is used for the current synthesis procedure is shown. The upper button switches between the output texture and the error map. Right below, the button labeled "Next pass" invokes the next Patchmatch-iteration. Note that if not using the Patchmatch algorithm all three buttons have no effect at all. The last button simply batches 9 iterations. The last generated texture is saved in the /Media/ folder with the name "testOutput.jpg".

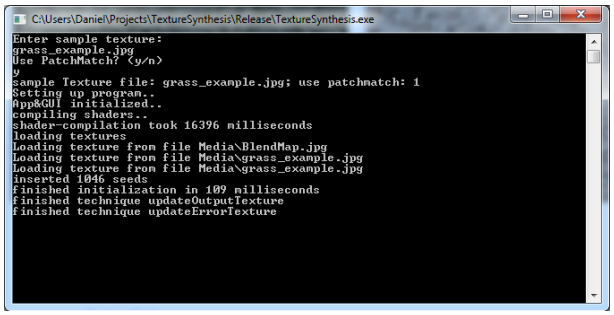

Figure 9: *Command line interface*

Figure 8: *Two different wall textures generated from a* 128^2 *sample to* 1024^2

Figure 10: *Main window of the Texture Tool*

References

[BSFG09] BARNES C., SHECHTMAN E., FINKELSTEIN A., GOLDMAN D. B.: PatchMatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics (Proc. SIGGRAPH) 28*, 3 (Aug. 2009). 1, 2

[CHK*80] COLEMAN D., HOLLAND P., KADEN N., KLEMA V., PETERS S. C.: A system of subroutines for iteratively reweighted least squares computations. *ACM Trans. Math. Softw. 6* (September 1980), 327–336. 1

[KEBK05] KWATRA V., ESSA I., BOBICK A., KWATRA N.: Texture optimization for example-based synthesis. *ACM Transactions on Graphics, SIGGRAPH 2005* (August 2005). 1

[RT06] RONG G., TAN T.-S.: Jump flooding in gpu with applications to voronoi diagram and distance transform. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2006), I3D '06, ACM, pp. 109–116. 4