Parallel patch-based texture synthesis

Steven Diviney 08462267, Trinity College Dublin

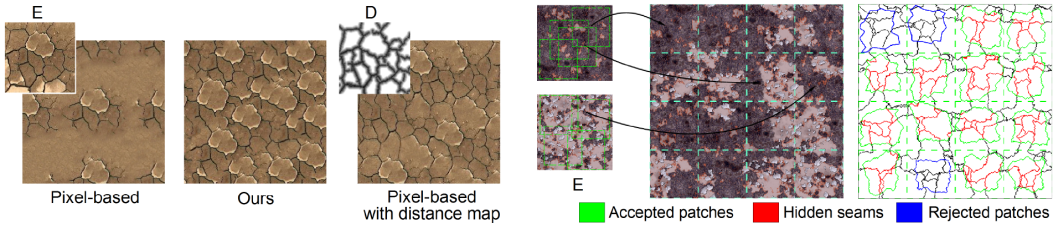

Figure 1:(Left) Comparison with pixel-based synthesis. The algorithm presented in this paper does not require a distance map to achieve high quality output. (Right) The patch-based synthesizer samples in parallel multiple patches from the exemplar E and stitches them to an existing texture. For each patch the synthesizer aims to hide existing visible seams (red) and to avoid creating new ones (green). Patches that produce visible seams are rejected (blue).

This paper presents an overview of the paper Parallel Patch-Based Texture Synthesis. Some areas are expanded on to clarify their meaning. The paper being examined presents an algorithm for fast parallel patch-based texture synthesis. Pixel-based approaches are generally much faster than patch-based techniques but their results break down when they try to preserve any significant structure. Patch-based techniques are slower but produce better results. The paper under consideration attempts to create a fast patch-based approach by improving several prior techniques. This is achieved through two main contributions; an algorithm to quickly find a good cut around a patch and a deformation algorithm to further align features. The algorithm outlined can be easily modified to achieve different synthesis result.

1. INTRODUCTION

Texture synthesis algorithms have been the focus of a relativly large body of research over the past ten years. There are two main synthesis approaches; pixel based and patch based[Wei et al. 2009]. A third approach [Saisan et al. 2001] not discussed in this paper uses a parametric model to describe a variety of textures. While they can be used for texture generation they perform better for analysis of textures. Although there is a variety of pixel and patch based algorithms they all share similar characteristics. Pixel-based approaches exhibit a high degree of parallelism and map well to GPUs but the results can seem quite lacking compared to patch-based approaches. This becomes more pronounced as the amount of structure in a texture increases.

Patch-based approaches require relatively slow algorithms to layout patches and stitch them together. However, they are better at preserving structure. Pixel based approaches can be improved by the addition of a feature distance map, as shown in figure 1, but obtaining such a map is another complex problem. [Lefebvre and Hoppe 2006]

The reason for the differences in output between the two approaches come down assumptions made by the algorithms. Pixel based texture synthesis typically fill in the result image by finding and copying pixels with the most similar local neighborhood as the synthetic texture. [Paget and Longstaff 1998] The neighborhoods are quite small so strong features become deformed or get completely destroyed. This approach works well when applied to the more traditional notion of a texture, that is, an infinite pattern that can be modeled by a stationary stochastic process.

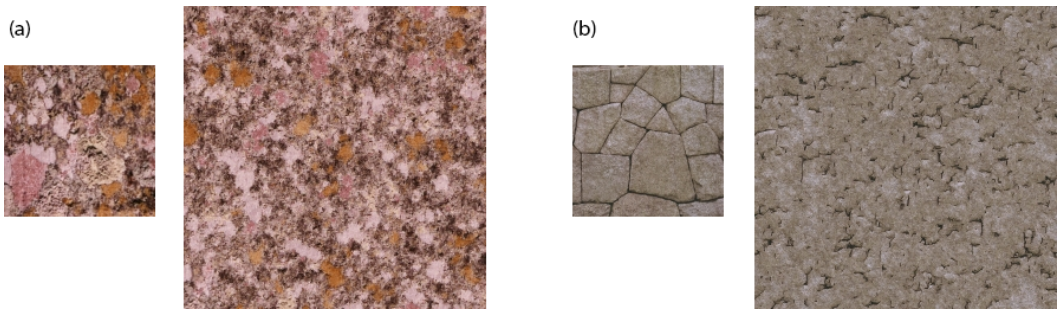

Figure 2: (a) shows texture synthesis of tree bark using a pixel based approach. The same technique is applied to (b) and the results are of much lower quality. The structure of the stones has been completely destroyed. Taken from [Paget and Longstaff 1998]

Patch based approaches attempt to create a new texture by copying and stitching together textures at various offsets. The patches are aligned so as to preserve features and reduce the number of visual artifacts such as seams.

The paper discussed presents a fast patch-based texture synthesizer. While still not as fast as pixel-based approaches it is significantly faster than previous patch-based algorithms. This is accomplished through four main contributions;

— A fast, approximate, algorithm to optimize the patch boundaries. This greatly improves performance with little impact on quality.
— An algorithm to deform the patches after boundary optimization and improve feature alignment.

— A scheme for using new patches to hide existing errors. Patches with strong visible seams are rejected. Patches can be processed independently.
— A full GPU implementation.

2. RELATED WORK

Patch-based texture synthesis is typically broken down into four problems; patch synthesis, patch placement, patch stitching and feature alignment. The paper being discussed does not stray too far from this model but gives more attention to the contributions presented. These four areas are used to categorize related work.

2.1. Patch-based texture synthesis

Early schemes select patches at random and feather the edges to form a new texture[Guo et al. 2000]. Feathering is a relatively simplistic approach to merging patches. Every pixel along a seam is assigned a weighted, typically a Gaussian kernel, sum of neighboring pixel values. This destroys any sharp features and amounts to blurring the seams.

Image quilting adds patches in scan-line order to a grid. The first block is copied at random and the subsequent blocks are placed such that they partly overlap with previously placed blocks. The boundary between two images is optimized by means of a minimum cost path. This ensures a seam introduces minimal colour difference[Efros and Freeman 2001].

Graph-cut improves this process by creating patches of arbitrary shape. It also introduces the ability to hide existing error using new patches[Kwatra et al. 2003]. The implementation of the cycle cuts presented in this paper differs in its approach from the algorithm presented by V. Kwatra, et al. but the idea is quite similar. The approach presented in this paper also attempts to find the cut of minimum cost, although it is approximate, not optimal. It is worth examining Graph-cut in some detail to aid the comprehension of the ideas presented in this paper.

2.1.1. Graph-cut. At its core, Graph-cut is a classical graph problem called minimum cut. The wish is to find a low-cost path through the overlap region between patches. The measure used is the difference of colour between the pairs of pixels in this region. This is further improved by incorporating old seam costs into new cuts, potentially hiding visible seams under new patches. Lastly the frequency of the region is taken into account. Seam boundaries are prominent in low frequency regions so the cost function is adjusted to account for the gradient of the image.

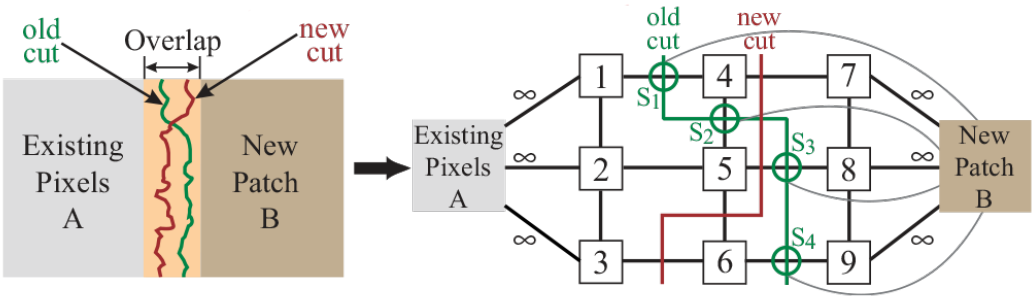

Figure 3: (Left) Finding the best cut (red) with an old seam (green) already present. (Right) Graph formulation with old seams present. Nodes s_1 to s_4 and their arcs to **B** encode the cost of the old seam.

2.2. Patch placement

The method outlined in this paper attempts to align patches based on seam cost optimization and feature alignment. Each new patch added is constrained to a small part of the output image. The patches are large compared to other approaches. The patch sampling strategy is a simple uniform random search. The large patch sizes allow for the synthesis of structured patterns. The iterative nature of the solution, the repeated overlay of patches, hides most of the error. As such, patch placement does not receive a great level of attention directly.

Patchmatch uses a more sophisticated sampling strategy for smaller patches. The search is alternated between a random and coherent method[Barnes et al. 2009].

2.3. Patch stitching

Graph-cut achieves good stitching results through use of optimal seam cuts [Boykov et al. 2001]. This is further improved by its patch placement and matching strategy [Kwatra et al. 2003].

Cut calculation in drag-and-drop pasting [Jia et al. 2006] uses a somewhat similar boundary optimization to the method described in this paper. The boundary optimization uses several passes of dynamic programming to find the shortest cycle around the patch. In this paper an approximate cycle is computed in a single pass. This approximate path loses very little accuracy when compared to the optimal path.

2.4. Feature Alignment

Feature alignment is one of the most difficult areas of texture synthesis. The solution presented aligns features by deforming patches during parallel optimization. The colour alignment step is inspired by stero-pair matching [Baker 1982]. Dynamic programming is used to align features along the seams. After aligning the colurs along the seam the deformations are propagated inside the patch.

3. PARALLEL PATCH BASED SYNTHESIZER

Each of the four main contributions of this paper will be examined in detail. A high-level outline of the algorithm follows.

Given a source image, or exemplar E, the algorithm creates a visually similar texture by repeatedly stitching multiple patches from E. The result is stored in a map S containing coordinates in E. $E[S]$ is the final coloured texture.

The synthesis is done iteratively. With each iteration multiple patches are randomly selected and placed onto S. In order to process the patches in parallel the must be placed in S so they do not overlap. This is done by overlay S with a grid where each cell contains one patch. The alignment of the grid with respect to S changes randomly between iterations. This is done to avoid any bias. Each cell in the grid is then processed independently in parallel.

Each patch placed in a cell is optimized to minimize visual seams. The optimization attempts to minimize the discontinuities along the cut of the patch and to hide existing cuts in $E[S]$, produced in previous iterations.

The patches are then deformed to align features. The deformation modifies the colour of pixels along the seam, not their placement. It is constrained to limit obvious visual deformation in the result image.

Finally, a patch can be rejected depending on its quality. The patch is rejected if the seam along its cut has more error than all existing seams inside the cut. Rejection is done twice, once before deformation and once after.

4. FAST APPROXIMATE CYCLE CUTS

Every patch boundary is optimized. A patch, P, is a disk of radius R centered at a position o_e in E and placed at a center position o_s on S. The goal is to find a closed cut C in P that contains at least the point o_s in S. C should produce as little colour difference as possible between P and $E[S]$.

Instead of using the procedural graph-cut as outlined by [Kwatra et al. 2003] C is computed using dynamic programming, hence forth referred to as DP. DP relies on simple arrays suitable to a GPU implementation. However, to make the optimization compatible with DP P must be processed in polar space.

P_{polar} is the parameterized version of P with polar coordinates. P_{polar} is a rectangle of size $W \times H$ such as: $W = N_p \times R$ and $H = N_\theta \times 2\pi R$. N_p and N_θ are two constant factors used to add some accuracy to the discrete sampling when transforming P into P_{polar}. The function T transforms the image coordinates from Cartesian to polar space. T^{-1} is the inverse of this transform.

The idea of polar space is not an intuitive one and it's definition resists a simple summary. For the purpose of this paper it may be useful to regard T as a function mapping P to a linear structure suitable for fast processing on a GPU.

To account for the distortions that appear in P_{polar} a normalization function is used.

Using P_{polar} the cut C is now a path that starts at the first row of P_{polar} and ends at its last row. Since C is closed in P it has to start and end at the same abscissa, or x coordinate, in P_{polar}.

The Y-monotony constraint used in image quilting is relaxed in this implementation. This constraint, labeled J_{max} is a positive integer that limits the maximum offset between $C[y]$ and $C[y+1]$. In image quilting $J_{max} = 1$. In practice this means the cut is able to follow existing horizontal seams.

T^{-1} is used to transform C to P. The existing seams in S now lie on the left side of C in P_{polar} and will be hidden by the newly place patch P.

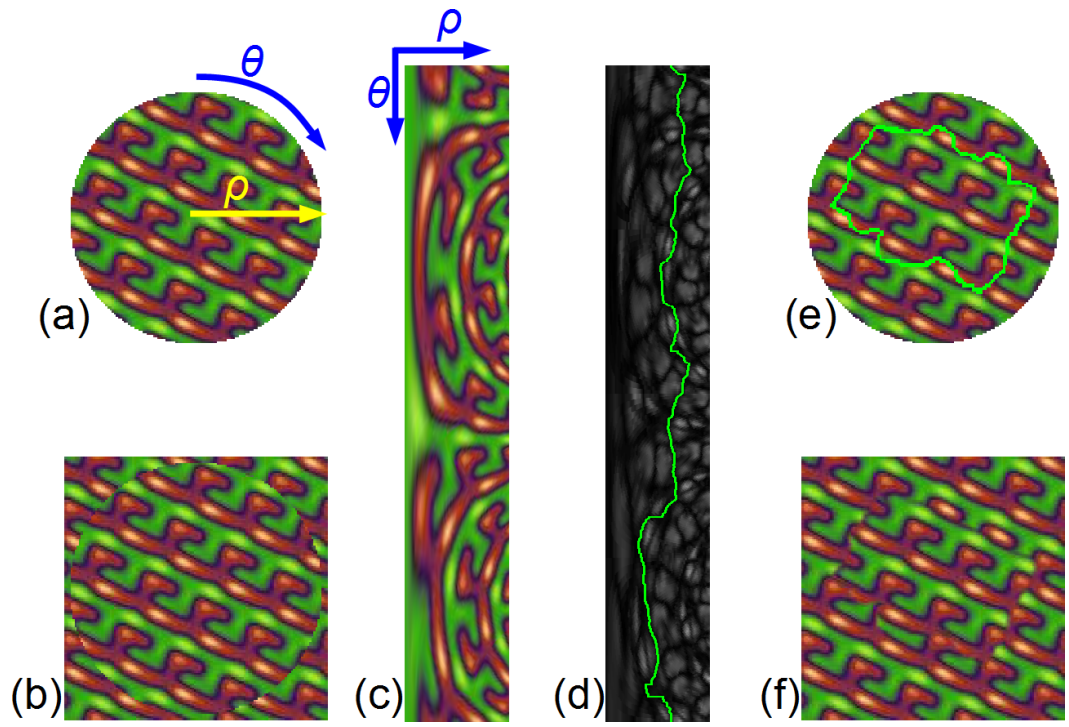

Figure 4: (a) P (b) P placed on $E[S]$ (c) P_{polar} (d) Error map with the cut C in green. (e) The optimal boundary $T^{-1}(C)$ on P (f) Stitching result.

4.1. Seam cost

Seam cost is used to quantify the visible discontinuities along C. M is similar to the cost function used in graph-cut outlined on p. 5. It is defined as

$$M(t_e, t_s, \delta) = \left(\frac{||E[t_e] - E[S[t_s - \delta]]||^2 + ||E[S[t_s]] - E[t_e + \delta]||^2}{\eta + ||E[t_e] - E[t_e + \delta]||^2 + ||E[S[t_s]] - E[S[t_s - \delta]]||^2} \right)^d$$

t_e are coordinates in E, t_s are coordinates in S, δ is a displacement vector. $\delta = (1, 0)$ when cutting vertically and $\delta = (0, 1)$ when cutting horizontally. η is a strictly positive regularization factor used to limit the effect of the denominator. The denominator allows for free transactions at high frequency locations in E and $E[S]$. Graph-cut takes the gradient into account at each point in the denominator, a more computationally expensive approach. The exponent d penalizes strong seams when it's value is high. It is typically set to 2.

The existing errors in S can be easily computed as follows:

$$M^s(t_e, t_s, \delta) = M(S[t_e], t_s, \delta$$

Because the optimization is done in polar space a polar version of M is also defined.

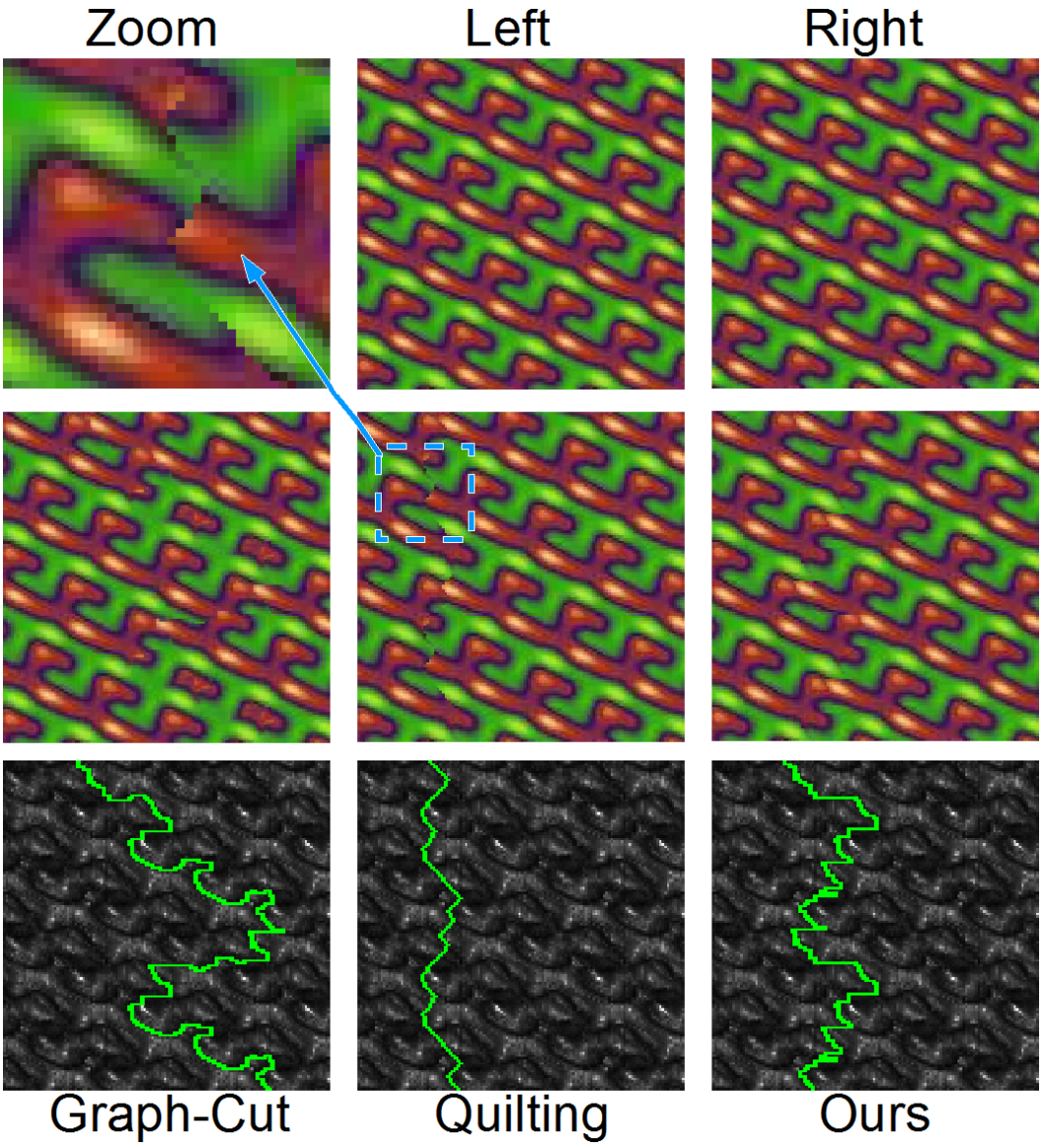

Figure 5: Top: A left/right texture regions to be overlapped. Middle: From left to right: separating the two regions with graph-cut, image quilting and the cut presented in this paper using $J_{max} = 16$. Bottom: Error maps produced by the overlap. Each map sums up the vertical and the horizontal transition errors. Light areas indicate high errors. Cuts are shown in green.

4.2. Patch cost

A quality cost is associated with every patch. The existing cost on the left side of the cut C is subtracted from the cost of C, giving the cost inside the patch, or $T^{-1}(C)$. This gives an energy function involving three terms: H, V and ϵ

H is the horizontal transition cost along C. V is the vertical transition cost along C. It is slightly more complicated because a horizontal gap as long as J_{max} pixels may appear between $C[y]$ to $C[y+1]$. V has to sum up all the vertical costs along this gap. Both of these functions use the polar version of the seam cost function with the appropriate displacement vector, i.e $\delta = (1, 0)$ for H and $\delta = (0, 1)$ for V.

ϵ represents the existing errors in S. These errors lie on the left side of C and will be hidden by P. This means that for any row u_y in P_{polar} all existing errors that precede $C[u_y]$ will be subtracted. The energy of a cut can now be defined:

$$\epsilon(C) = \sum_{y=1}^{H}(H(C[y], y) + V(C[y], C[y+1], y) - \epsilon(C[y], y))$$

Or the energy of a cut is equal to its horizontal cost plus its vertical cost minus the existing errors in S.

All sub-solutions for $\epsilon(C)$ are pre-computed and stored in a table.

4.3. Approximate cuts

The cut C must start and end at the same abscissa. Drag-and-drop pasting [Jia et al. 2006] optimizes this boundary using an iterative solution. The boundary is recalculated until its energy no longer decreases between iterations. The solution presented here approximates an optimal boundary in a single iteration. The approximation is based on the premise that there exists at least one path that starts and ends at the same abscissa. This path is found by backtracking all paths from bottom to top.

4.4. Approximate cut quality

The approximate cut calculated using this method is very close to the optimal cut in the vast majority of cases. The two algorithms were compared over 3000 samples. The Optimal cut achieved an average error of 50.855 while the approximate cut achieved 53.633. The approximate cut was ranked 8.59 / 256. There are rare cases where the approximation preforms poorly, producing strong seams but these are likely to be rejected in later stages of the process.

4.5. Dynamic programming

Dynamic programming attempts to break complex problems down into sub-problems. These sub-problems are nested recursively to create the larger problem. The sub-problems must also contain some overlap and are typically only smaller than the overall problem by a constant additive factor. If the subproblems are a multiplicative factor smaller they are no longer classified as DP, such problems are usually classed as divide-and-conquer. Recursive calculation of the Fibonacci sequence is the classic example of a dynamic programming problem.

Once a subproblem as been solved its result is stored and used instead of re-evaluating the sub-problem in later calculations. As DP is quite simplistic, relying on simple arrays, it is well suited to a GPU implementation.

5. FEATURE ALIGNMENT

Minimizing $\epsilon(C)$ does not always guarantee a seamless result. The effectiveness of minimizing cut energy depends largely on the nature of the exemplar. This is especially true of

textures that contain aligned structural patterns, such as bricks. It is possible to offset two overlapped textures to produce a case where any cut optimization produces strong seams.

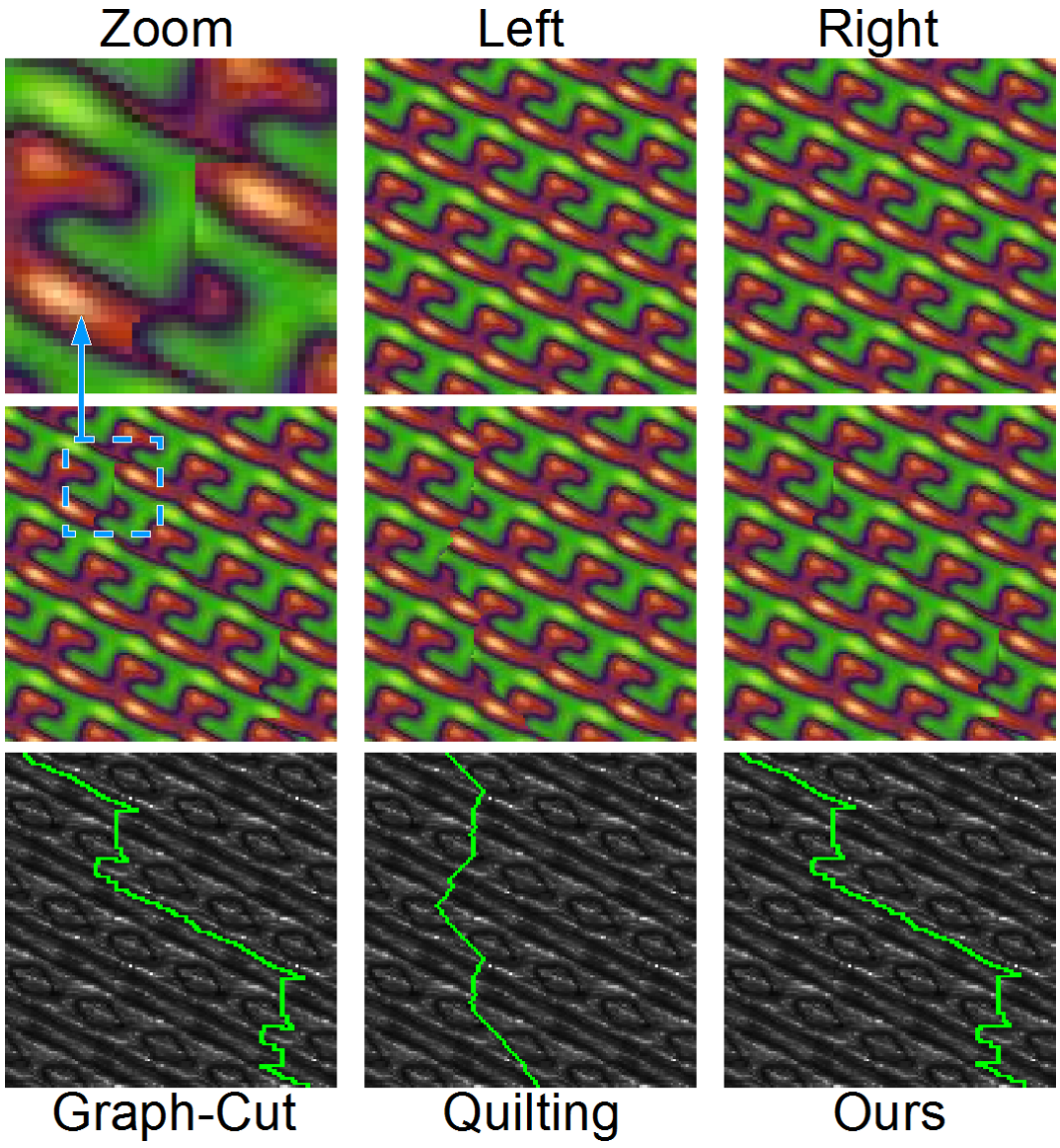

Figure 6: Top: A left/right texture regions to be overlapped. Middle: From left to right: separating the two regions with graph-cut, image quilting and the cut presented here using $J_{max} = 16$. Bottom: Error maps produced by the overlap. Each map sums up the vertical and the horizontal transition errors. Light areas indicate high errors. Cuts are shown in green.

If the cut optimization fails and produces seams small deformations are applied to align features on each side of C. The deformation consists of two steps. First, the colours along C in P_{polar} are offset to match the colours in $E[S]$ lying on the other side of C. After this is done the deformation is propagated to the inside of P.

5.1. Offsetting colours along the cut

The goal is to offset the indices of C to align features. The cut C is represented by an array where $C[y]$ is the x coordinate of the curve at row y in P_{polar}. D is the array that contains the new indices in C after offsetting. i.e. The colour at $P_{polar}[C[y], y]$ is replaces by the colour at $P_{polar}[C[D[y]], D[y]]$. $C[D]$ is the cut with the offset colours but the same shape as C.

The cost of vertical transitions is ignored. As C only encodes one x coordinate per row in P_{polar} offsetting the colours within the gap between $C[y]$ and $C[y+1]$ is difficult, as the gap can be up to J_{max} long.

Again, an energy function is used to calculate D. This is the cost of replacing the colour at $(C[y], y)$ with the colour at $(C[D[y]], D[y])$.

The offset is limited between any two rows. This ensures the deformations are small. The constant 2 has been chosen empirically. It represents the minimum possible offset. There is also a global maximum limit set by the user.

This process is optimized in a similar way to the cut calculation. Using dynamic programming all of the costs are pre-computed and stored in a table T.

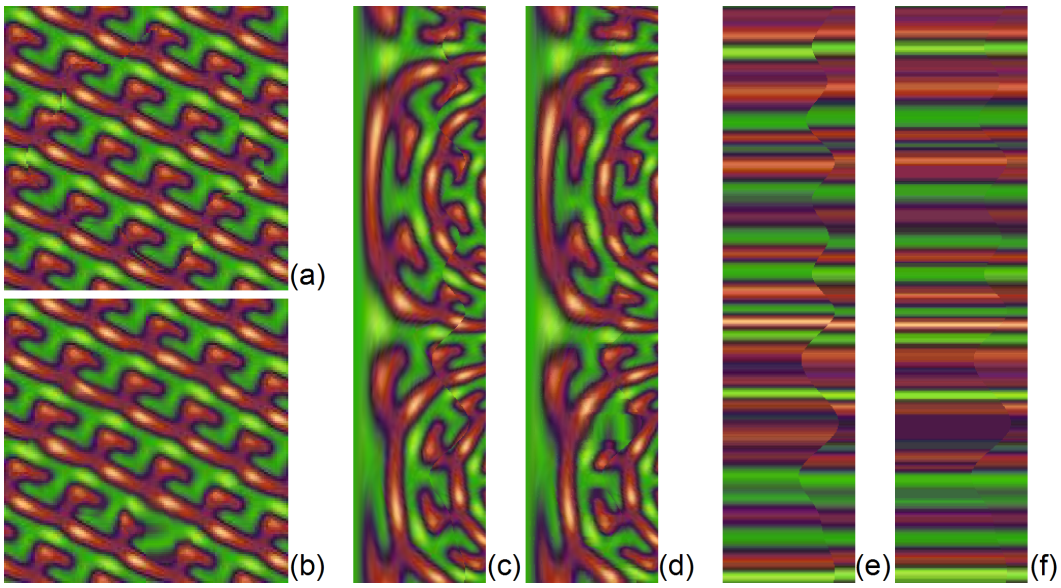

Figure 7: (a) The patch P produces sames when placed on $E[S]$. (b) Result after feature alignment. (c) Polar space view before alignment. (d) Polar space view after feature alignment. (e) Colurs along C before feature alignment. (f) Colours along C after feature alignment and propitiated inwards.

5.2. Initial State

The solution D must start and end at the same abscissa in T or parts of the texture will be lost after propagation inwards. When optimizing the energy function $E[C]$, instead of iterating to a solution, dynamic programming is used to backtrack all the paths. A similar approach is used here. Rather than repeating the optimization for all initial states the same approximation solution of $E[C]$ is used.

5.3. Deformation propagation

The deformation must be propagated inwards for the feature alignment to work. If it is not the cut will only appear to have been moved "in" by a single pixel. This is done in polar space using a linear interpolation function. It considers the fact that P_{polar} is vertically cyclic, e.g, $C[0]$ is closer to $C[H]$ than $C[H/2]$ and always interpolates along the shortest path.

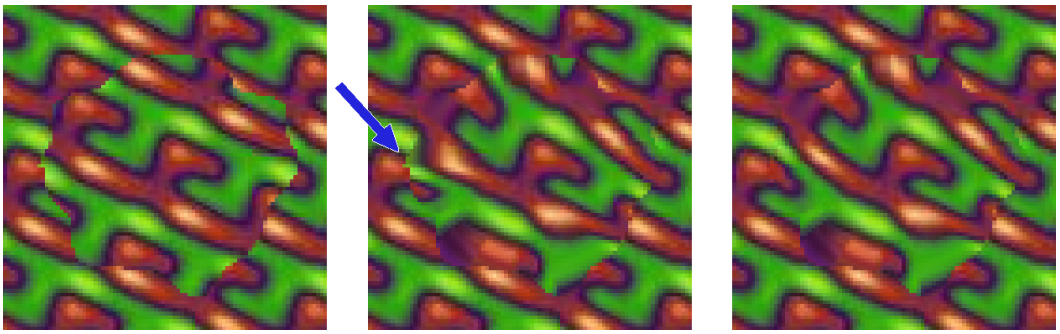

Figure 7: Left: Random cut with no deformation. Middle: Deformation using fixed initial state. The starting position is pointed by the blue arrow. Right: Deformation using out approximate solution for D.

6. PATCH REJECTION

Even after the optimizations the patch not contribute to the overall quality of the image. To decided if the patch is to be merged with the result it goes through two separate rejection policies.

The first one is applied before feature alignment. The patch is accepted if the subtracted errors in S, i.e. ϵ, are greater than the errors produced by C and rejected otherwise. In other words if the total energy of the cut is less than or equal to zero it has a positive effect on quality. Rejecting patches before feature alignment is a good heuristic. Accepted patches would have few seams and feature alignment performs well in these cases. As this energy calculation has already been performed this step is almost free.

The second rejection is applied after feature alignment and is as follows:

$$isImproving_{after}(C) = \epsilon(C[D]) + \epsilon(P) - \epsilon \leq 0$$

$\epsilon(C[D])$ is the energy along C after colour offsetting, $\epsilon(P)$ is the total energy in P after deformation and ϵ is the total existing energy on the left of C. This is very similar to the method using in the first round of rejection but does require some new extra calculations as the deformed image is being used.

7. IMPLEMENTATION DETAILS

The algorithm is implemented using Nvidia CUDA. Multiple patches are processed concurrently by placing their optimization tables one next to the other from left to right. With n patches the global optimization table has a size of $w \times h$ where $w = n \times W$ and $h = H$. In other words it's a standard array with its size calculated the standard way. This allows a thread to work on any single patch as long as it knows it's boundaries. It also allows the algorithm to be ran in a linear fashion if required.

The algorithm executes two main optimizations; the optimization of $\epsilon(C)$ followed by the optimization of $E_D(D)$. The buffers allocated for the optimization of $\epsilon(C)$ are reused during the optimization of $E_D(D)$ as neither optimization is ever ran at the same time.

Three texture buffers with a size of $w \times h$ are allocated. A buffer H first stores existent horizontal costs then stores new horizontal costs, a buffer V that stores new vertical costs and a buffer ϵ_v that stores existent vertical costs.

Prior to optimization all transition costs are pre-computed by performing the following steps:

— H is filled with horizontal existing costs and ϵ_v with vertical existing costs. First pass with one thread per entry in the table
— For every row in each patch in H and ϵ_v costs are accumulated from left to right. Second pass with a thread per row of a patch
— V is filled with zeros
— The new horizontal costs in H and the new vertical costs in V are computed. The existing content is subtracted from them and they are then stored.

H, V and ϵ_v share the same texture unit to assist efficient processing. Texture memory is mainly used to avoid cache conflicts when accumulating existing errors. After pre-computing the costs, H is used as the main optimization table while V and ϵ_v are set as read only. V and ϵ provide the additional terms of $\epsilon(C)$.

The dynamic programming consists of a top-down sub-solution pre-computation in H. This is followed by backtrack, or a bottom up calculation, of all cuts. The result is then placed in H. Cuts that do not start and end at the same abscissa are assigned infinite cost. The cut with minimum cost then selected for each patch.

A top-down calculation is typically the approach of memoization. However memoization is very similar to dynamic programming, the main difference being memoization is top-down while dynamic programming is bottom up. The majority if this papers references to dynamic programming are strictly correct but here there is memoization followed by dynamic programming.

The DP is such that each row y only depends on the preceding row $y-1$. This allows for massive parallelization. However, the accumulation task for any single row has 3 limitations:

— The number of threads is too limited to fully exploit modern GPUs when a single row at a time is parallelized.
— A global synchronization is required after processing each row.
— The synchronization suspends the computation within a column for a long time. During this time other computations will fill up the cache which cause it to loose its coherence.

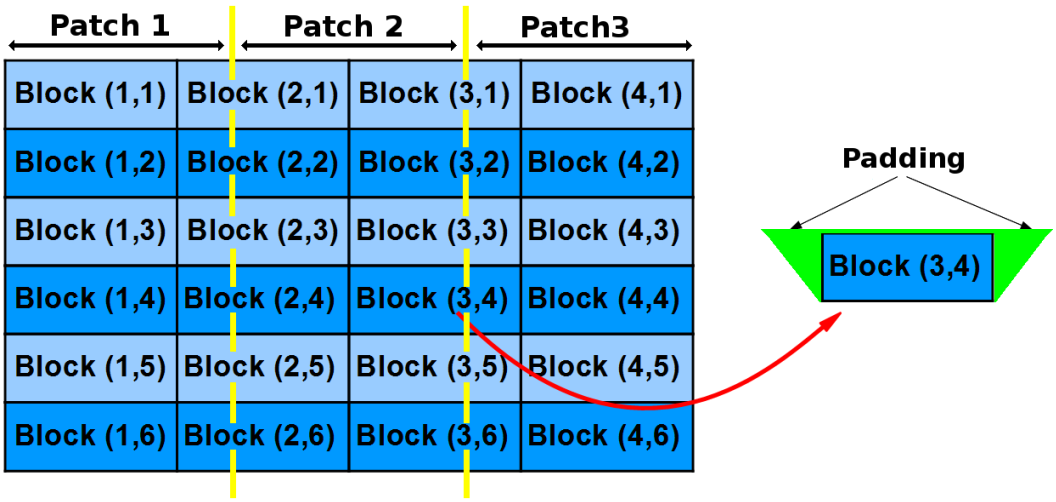

Figure 8: The DP table is subdivided into blocks. The threads responsible for processing a block need access to some data to the left and right of the block and the first row above the block. These areas are shown in green.

The simplest way around these problems is to process many rows before a global synchronization. This is done by sub-dividing H into blocks. A global synchronization is done after processing a line of blocks.

Dividing the data into blocks does present some additional problems. Namely the blocks need to be padded with additional data for the calculations to be valid. The padded data belongs to blocks above, left and right of the current block. The additional calculations with these padded regions are wasted but the expense is worth it for the increased coherence. As the padded data can be processed by multiple threads at the same time and is never stored they are stored in read-only buffer. Thus, the threads responsible for processing a block must load the data in shared memory. When the block has been processed the data is copied from shared memory back to H, without the padding. The size of the blocks has been determined empirically, as is the number of threads per block.

The same process is used for the bottom-up backtracking. No padding is required in this case. The temporary memory stores the complete solution which is then copied back to the table H before a global synchronization.

The optimization of $E_D(D)$ is quite similar to the optimization of $\epsilon(C)$ so the same DP can be used. The one change is vertical transitions are ignored when optimizing $E_D(D)$, so buffers V and ϵ_v are set to zero while J_{max} is set to 1.

8. RESULTS

Six different constants need to be set, as well as the image size. In the results presented in paper under consideration S is of size 512×512 and is initialized with white-noise. $R = 64$, $D_{max} = 32$, $N_p = 2$, $N_\theta = 1.5$, $J_{max} = 4$ and $\gamma = 1$. The algorithm is iterated 3 to 5 times to ensure the whole map is covered.

8.1. Varying the maximum radius R

The parameter with the most effect on quality is the maximum patch radius R. This is illustrated in figure 9. When R is small the synthesizer fails to capture the flowers. It

either breaks them or rejects them. As the algorithm is iterated the flowers may completely disappear. When R is large the synthesizer copies large patches from E. This produces a seamless result with little variation. Setting R to be just large enough to capture individual flowers in a patch results in a nice distribution.

The algorithm has been implemented such that R automatically decreases. It is still up to the user to select the best resulting image. The same idea has been applied to D_{max}. A large value for D_{max} makes the synthesis converge faster. Again, it is up to the user to select the best image.

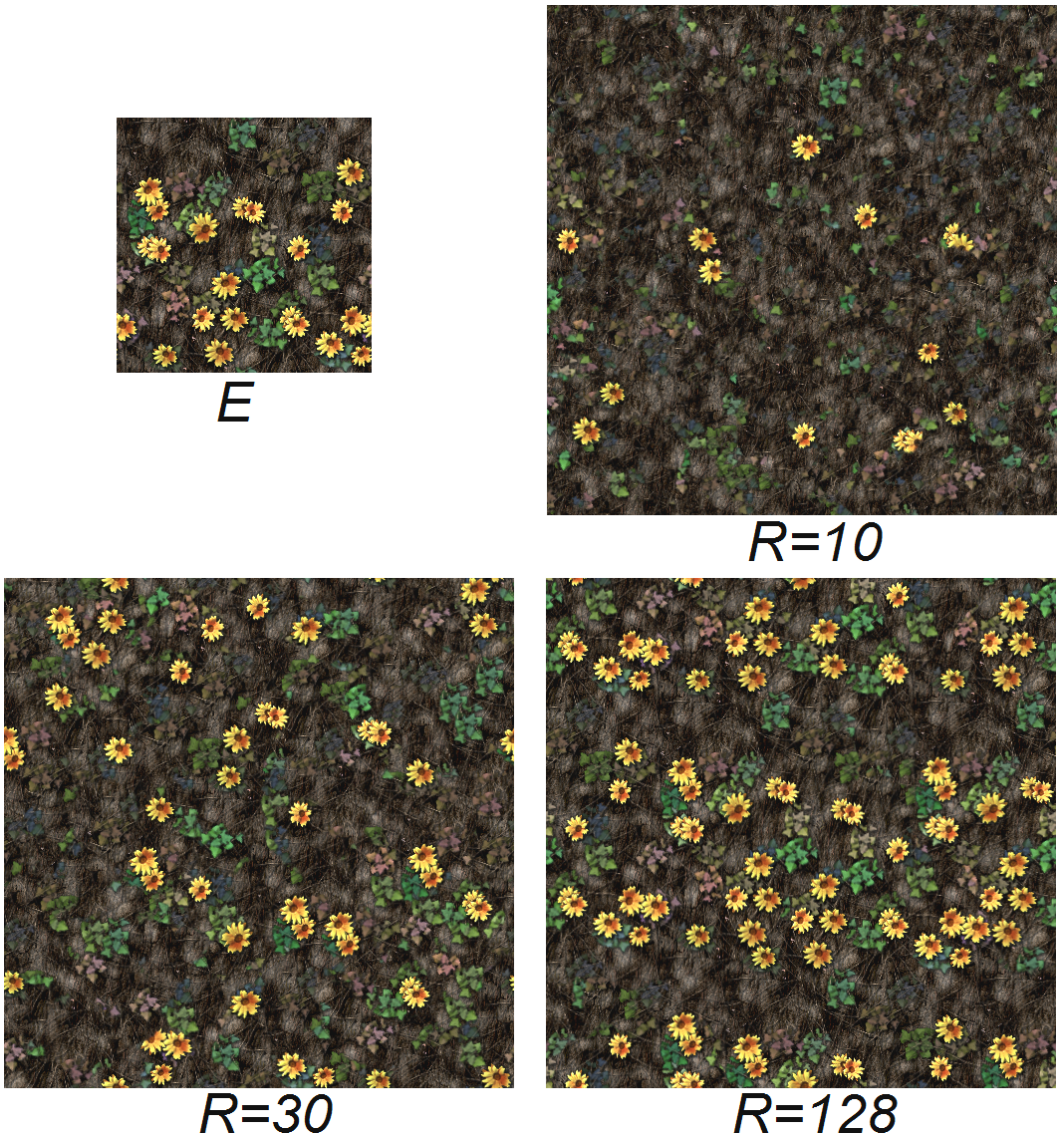

Figure 9: Varying the maximum radius R

8.2. Applications

The method presented here has several applications.

8.2.1. Texture completion. If a texture contains holes they can easily be filled by setting an infinite existing cost inside them.

8.2.2. Texture painting. Another advantage of this approach is that multiple exemplars can be used. This allows one texture to be used as a brush to paint on another texture. The painted zones are considered holes which the synthesizer fills.

8.3. Performance

The table below lists the execution time and memory usage for one iteration on an Nvidia GeForce GTX580. $R = 64$.

size of S	256	512	1024	2048
used memory in MB	$\leq$4	14	56	226
iteration time in ms	256	512	1024	2048
C DP initialization	9%	19%	29%	51%
C DP optimization	27%	22%	11%	12%
D DP initialization	3%	5%	8%	14%
D DP optimization	28%	22%	10%	8%

DP initialization cost grows quickly with S. This is due to the intensive evaluation of M_{polar} which requires texture fetches in E and S in addition to multiple calls to T. The DP optimization scales well because the number of patches increases with S. Increasing the number of patches increases the degree of parallelism but requires more memory.

8.4. Comparison with pixel-based synthesis

The algorithm presented here is roughly 10 times slower than per-pixel synthesizers. This is due to the number of iterations needed before convergence. This number depends on the nature of the texture and the synthesis parameters. Typically convergence occurs at 10 iterations but a satisfactory result can be obtained in less than half of that, as illustrated above.

The results achieved here benefit from patch-based approaches. The quality achieved by fast per-pixel algorithms largely depend on the amount of jitter added during synthesis. This has to be manually selected and is different for each texture. Additionally, patterns with a large amount of structure require the addition of a feature distance map with pixel-based approaches, as shown in figure 1. As far as the authors know, there is no fast per-pixel algorithm that automatically reaches the quality they demonstrate on highly regular patterns.

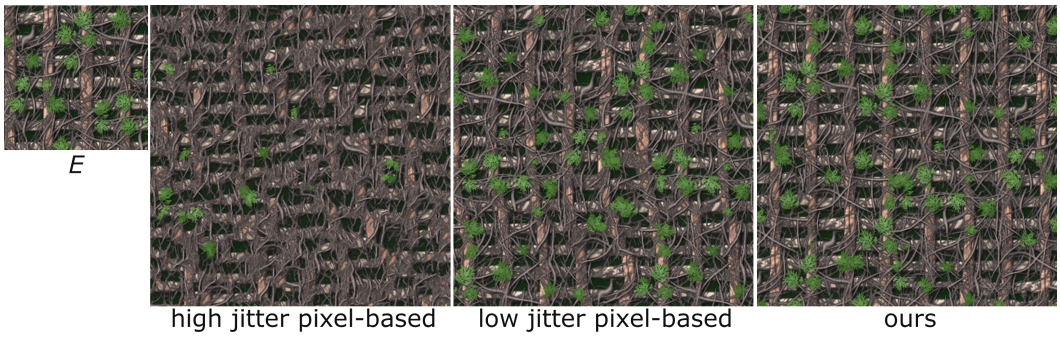

Figure 10: Comparison with pixel-based synthesis. From left to right: E having a size of 256^2. Pixel-based results computed in 17ms using a high jitter. Pixel-based result computed in 15ms using a low jitter. Result computed using presented algorithm in 112 ms using $R = 128$. S has a size of 512^2.

9. DISCUSSION

This paper presented a parallel patch-based texture synthesis algorithm that quickly achieves high quality results. It relies on a parallel implementation of an approximate boundary optimization and patch deformation to align features. A patch rejection scheme ensures a progressive increase in quality. The algorithm uses several iterations and improves upon several different methods cited throughout the paper.

The main limitation is the patch selection and placement. Global structures in textures cannot be preserved. This problem is inherited from patch-based texture synthesis and texture synthesis as a whole. Such a problem falls more in the domain of image synthesis so this criticism is minor.

Performance is hindered by the high rate of patch rejection, despite the fast iterations. A possible solution would be to intelligently propagate good patches throughout the grid or testing several patches per cell and selecting the one of lowest cost rather than just testing one.

REFERENCES

Baker, H. 1982. Depth from edge and intensity based stereo. Tech. rep., DTIC Document.

Barnes, C., Shechtman, E., Finkelstein, A., and Goldman, D. B. 2009. PatchMatch: A randomized correspondence algorithm for structural image editing. ACM Transactions on Graphics (Proc. SIGGRAPH) 28.

Boykov, Y., Veksler, O., and Zabih, R. 2001. Fast approximate energy minimization via graph cuts. Pattern Analysis and Machine Intelligence, IEEE Transactions on 23, 11, 1222–1239.

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

Guo, B., Shum, H., and Xu, Y. 2000. Chaos mosaic: Fast and memory efficient texture synthesis. Microsoft research paper MSR-TR-2000-32.

Jia, J., Sun, J., Tang, C., and Shum, H. 2006. Drag-and-drop pasting. In ACM Transactions on Graphics (TOG). Vol. 25. ACM, 631–637.

Kwatra, V., Schödl, A., Essa, I., Turk, G., and Bobick, A. 2003. Graphcut textures: image and video synthesis using graph cuts. In ACM Transactions on Graphics (TOG). Vol. 22. ACM, 277–286.

Lefebvre, S. and Hoppe, H. 2006. Appearance-space texture synthesis. In ACM Transactions on Graphics (TOG). Vol. 25. ACM, 541–548.

Paget, R. and Longstaff, I. 1998. Texture synthesis via a noncausal nonparametric multiscale markov random field. Image Processing, IEEE Transactions on 7, 6, 925–931.

Saisan, P., Doretto, G., Wu, Y., and Soatto, S. 2001. Dynamic texture recognition. In Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on. Vol. 2. IEEE, II–58.

Wei, L., Lefebvre, S., Kwatra, V., Turk, G., et al. 2009. State of the art in example-based texture synthesis. In Eurographics 2009, State of the Art Report, EG-STAR. 93–117.