High Performance Graphics (2012)
C. Dachsbacher, J. Munkberg, and J. Pantaleoni (Editors)

Parallel patch–based texture synthesis

Anass Lasram Sylvain Lefebvre
ALICE/INRIA

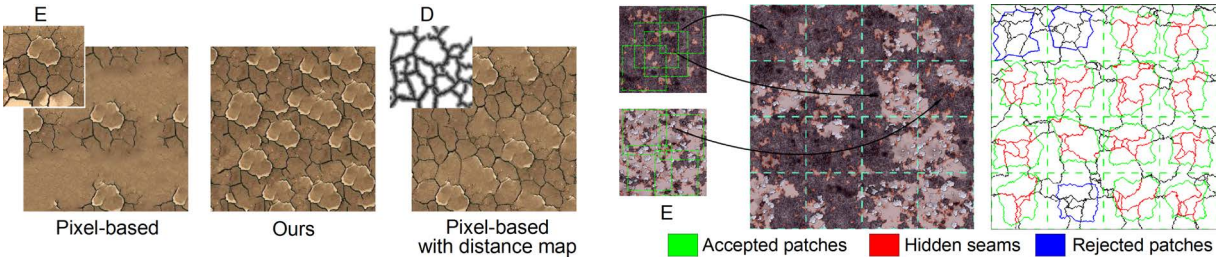

Figure 1: Left: *Comparison with pixel–based synthesis of [LH06]. Our result does not require a distance map to achieve high quality synthesis.* Right: *Our patch–based synthesizer samples in parallel multiple patches from E and stitches them to an existing texture. For each patch the synthesizer aims to hide existing visible seams (red) and to avoid new ones (green). Patches that still produce visible seams are rejected (blue).*

Abstract

Fast parallel algorithms exist for pixel–based texture synthesizers. Unfortunately, these synthesizers often fail to preserve structures from the exemplar without the user specifying additional feature information. On the contrary, patch–based synthesizers are better at capturing and preserving structural patterns. However, they require relatively slow algorithms to layout the patches and stitch them together.
We present a parallel patch–based texture synthesis technique that achieves high degree of parallelism. Our synthesizer starts from a low–quality result and adds several patches in parallel to improve it. It selects patches that blend in a seamless way with the existing result, and that hide existing visual artifacts. This is made possible through two main algorithmic contributions: An algorithm to quickly find a good cut around a patch, and a deformation algorithm to further align features crossing the patch boundary. We show that even with a uniform parallel random sampling of the patches, our improved patch stitching achieves high quality synthesis results.
We discuss several synthesis strategies, such as using patches of decreasing size or using various amounts of deformation during the optimization. We propose a complete implementation tuned to take advantage of massive GPU parallelism.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

1. Introduction

The past decade has seen the development of many by example texture synthesis algorithms. They can be roughly categorized into pixel–based approaches and patch–based approaches [WLKT09].

Pixel–based approaches exhibit a high degree of parallelism and map well to GPUs [LH05]. In addition, many search strategies have been proposed to accelerate neighborhood–matching during the synthesis [Ash01, BSFG09, PELS10].

In contrast to pixel–based synthesis, patch–based approaches require relatively slow algorithms to layout the patches and stitch them together, for instance optimizing for the patch frontier (or *cut*) using graph–cut [KSE*03]. However, patch–based techniques are better at preserving structures, while per–pixel synthesis must be guided by the addition of a feature distance map [LH06], which is sometimes difficult to define (Figure 1).

Contributions We present in this paper a fast patch–based texture synthesizer having the following main contributions:

DOI: 10.2312/EGGH/HPG12/115-124

- We introduce a fast, albeit approximate algorithm to optimize for the patch boundaries. This greatly improves performance with little impact on quality.
- We propose a novel algorithm to deform the patches after boundary optimization and improve feature alignment.
- We synthesize textures by optimizing for multiple patches in parallel. Our global scheme uses newly added patches to hide existing errors. It rejects patches producing visible seams or requiring strong deformations.
- We design a full GPU implementation enabling fast synthesis and interactive user controls with the same level of quality than state–of–the–art patch based synthesizers.

2. Related work

Patch–based texture synthesis Early schemes [GSX, PFH00] select patches at random and feather the edges to form a new texture. In image quilting [EF01] patches are added in scanline order to a grid. The frontier between new and previous patches is optimized, finding a path of minimal color difference with dynamic programming. Graph–cut textures [KSE*03] later improved this process by relying on patches of arbitrary shapes, also adding the ability to hide existing error with newly added patches.

Patch placement Kwatra et al. [KSE*03] proposed to select patches using an FFT-based block matching algorithm to match regions of overlap before stitching. Patchmatch [BSFG09] introduces an efficient sampling strategy for matching small patches by alternating between coherent and random searches.

In this paper we focus on improving the stitching of large patches and rely on a simple, uniform random patch sampling strategy. Despite the random search, our optimization strategy enables the synthesis of structured patterns: Patches are constantly added to the result, hiding previous errors.

Patch stitching Graph–cut [BVZ99] and gradient domain Poisson image editing [PGB03] are two successful tools applied to patch stitching in the context of textures and images [KSE*03, ADA*04, LZPW04].

In drag–and–drop pasting [JSTS06], a cyclic boundary around the patch is computed so as to reduce gradient mismatch prior to the Poisson optimization. The boundary optimization involves several passes of dynamic programming to find a shortest cycle around the patch. In our method, rather than computing an optimal cycle we compute an approximate cycle using a *single* dynamic programming pass.

Feature alignment Most texture synthesizers have difficulties capturing contours and edges. Wu et al. [WY04] extract a sparse set of curvilinear features capturing contours. A contour map is then synthesized and used to guide the synthesis of the colors. Matusik et al. [MZD05] align the features of multiple textures prior to interpolating between them. In appearance space texture synthesis [LH06] neighborhood–matching comparisons incorporate a feature distance to better preserve contours. In each of these papers the synthesis results are greatly improved by the explicit feature alignment step.

We explicitly align features by deforming patches during our parallel optimization. Such deformation typically requires extracting structural information from the example to produce a feature map. i.e. a sparse set of matchable features [JT05]. Instead, our color alignment step is inspired by work in stereo–pair matching [BB81] where dynamic programming is used to densely align features along epipolar lines. After aligning the colors along the boundary of the patch we propagate the deformation inside.

3. Our patch–based synthesizer

Given a source exemplar E our method synthesizes a visually similar toroidal texture by repeatedly stitching multiple patches from E. The result is stored in a map S containing coordinates in E. We note $E[S]$ the final colored texture.

The synthesis is done in an iterative manner where in each iteration multiple patches are randomly selected in E and placed on S. To process patches in parallel within an iteration, the placement is made in a way such as patches placed on S do not overlap. To maximize the number of non–overlapping patches, S is overlaid with a grid where each cell contains one patch. To avoid any bias, the alignment of the grid with respect to the synthesized image randomly changes between iterations. Cells in the grid are then independently processed in parallel (Figure 1 Left).

For each patch placed in each cell we optimize for the boundary of the patch so as to minimize visual seams. The optimization aims to minimize the discontinuities along the cut of the patch and to hide existing cuts in $E[S]$ that are produced during previous iterations (Section 4).

We further reduce seams by deforming the patch so as to align features (Section 5). In order to avoid altering structural patterns in $E[S]$ we use constraints that limit the deformations in the result.

We can decide to either accept or reject a patch depending on its benefit to the overall quality. The patch is rejected if the seam along its cut has more error than all existing seams inside the cut (Section 6).

4. Fast approximate cyclic cuts

In this section we describe the boundary optimization for a single patch that we note $\mathcal{P}$. Section 7 gives details on how to optimize simultaneously for multiple patches.

We interpret $\mathcal{P}$ as 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

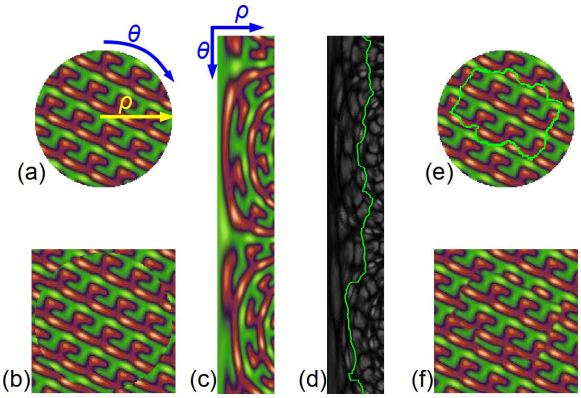

Figure 2: (a): $\mathcal{P}$ (b): *Placing* $\mathcal{P}$ *on* $E[S]$. (c): $\mathcal{P}_{polar}$ (d): *Error map with the cut* $\mathcal{C}$ *in green.* (e): *The optimized boundary* $\mathcal{T}^{-1}(\mathcal{C})$ *on* $\mathcal{P}$. (f): *Stitching result.*

to find a closed cut $\mathcal{C}$ in $\mathcal{P}$ that contains at least the point o_s in S. $\mathcal{C}$ should produce as little color differences as possible between $\mathcal{P}$ and $E[S]$.

Instead of using graph–cut to compute $\mathcal{C}$ we want to use dynamic programming (DP). DP is simpler, faster and relies on simple arrays suitable to a GPU implementation. However, to make the optimization compatible with DP we process $\mathcal{P}$ in polar space.

We note $\mathcal{P}_{polar}$ the parameterized version of $\mathcal{P}$ with polar coordinates (Figure 2). $\mathcal{P}_{polar}$ is a rectangle of size $\mathcal{W} \times \mathcal{H}$ such as: $\mathcal{W} = N_\rho \times R$ and $\mathcal{H} = N_\theta \times 2\pi R$. N_ρ and N_θ are two constant factors used to add some accuracy to the discrete sampling when transforming $\mathcal{P}$ into $\mathcal{P}_{polar}$. We note $\mathcal{T}$ the transformation from Cartesian to polar space and $\mathcal{T}^{-1}$ the inverse transformation.

One can notice that in Figure 2 distortions appear in $\mathcal{P}_{polar}$ and this especially in the left area of $\mathcal{P}_{polar}$ which corresponds to the interior of $\mathcal{P}$. To account for this distortion a normalization $\mathcal{N}$ is necessary. For a position located at u in $\mathcal{P}_{polar}$, $\mathcal{N}$ is such as: $\mathcal{N}(u) = \frac{2\pi u_x}{\mathcal{H} \times N_\rho^2}$. The computation of $\mathcal{N}$ is given in Appendix A.

Using $\mathcal{P}_{polar}$, the cut $\mathcal{C}$ is now a path that starts at the first row of $\mathcal{P}_{polar}$ and ends at the last row of $\mathcal{P}_{polar}$. Since $\mathcal{C}$ is closed in $\mathcal{P}$, it has to start and end at the same abscissa in $\mathcal{P}_{polar}$ (cyclic cut).

We now define $\mathcal{C}$ as being in polar space and let $\mathcal{T}^{-1}(\mathcal{C})$ to be the actual boundary of $\mathcal{P}$. We note $\mathcal{C}[y]$ the x coordinate of the curve $\mathcal{C}$ at row y in $\mathcal{P}_{polar}$. Since $\mathcal{T}^{-1}(\mathcal{C})$ is cyclic we access $\mathcal{C}$ in a wrap mode. e.g. $\mathcal{C}[\mathcal{H}+1] = \mathcal{C}[0]$. Using this definition, the existing seams in S now lie on the left side of $\mathcal{C}$ in $\mathcal{P}_{polar}$ and will be hidden by the newly placed patch $\mathcal{P}$.

As in graph–cut textures, we define $\mathcal{C}$ as lying *between* the pixels of $\mathcal{P}_{polar}$. Also, both horizontal and vertical transition errors along $\mathcal{C}$ will be taken into account.

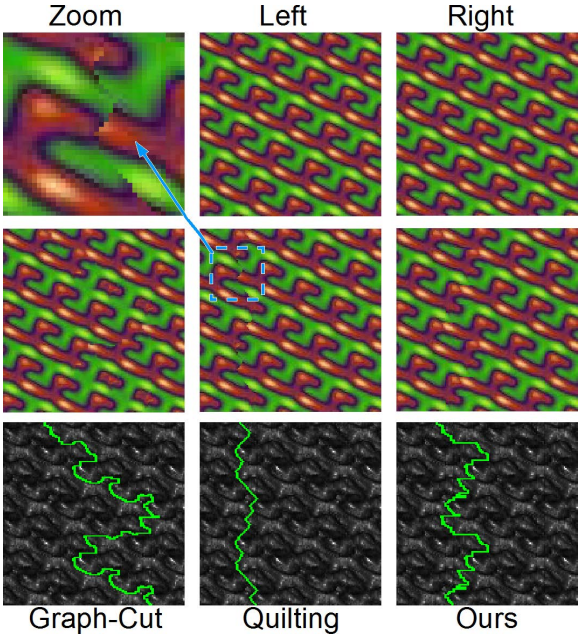

Figure 3: Top: *A left/right texture regions to be overlapped.* Middle: *From left to right: separating the two regions with graph–cut, image quilting and our cut using* $J_{max} = 16$. Bottom: *Error maps produced by the overlap. Each map sums up the vertical and the horizontal transition errors. Light areas indicate high errors. Cuts are shown with a green color.*

We relax the Y–monotony constraint of image quilting to be: $\forall y \in \{1..\mathcal{H}-1\}\ \ |\mathcal{C}[y] - \mathcal{C}[y+1]| \leq J_{max}$. J_{max} is a positive integer that limits the maximum offset between $\mathcal{C}[y]$ and $\mathcal{C}[y+1]$. In image quilting $J_{max} = 1$. The improvement of relaxing this constraint is shown in Figure 3.

Seams cost To quantify the visible discontinuities along $\mathcal{C}$ we use a cost function $\mathcal{M}$ similar to the one used in graph–cut textures and defined as:

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

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 transitions at high frequency locations in E and $E[S]$. The exponent d penalizes strong seams when its value is high. It is typically set to 2.

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

$$\mathcal{M}^S(t_e, t_s, \delta) = \mathcal{M}(S[t_e], t_s, \delta)$$

Because our optimization is done in polar space we define a polar version of $\mathcal{M}$ as follows:

$$\begin{cases} \mathcal{M}_{polar}(u, \delta) = \mathcal{N}(u) \times \mathcal{M}(o_e + p, o_s + q, \overrightarrow{pq}) \\ p = \mathcal{T}^{-1}(u) \\ q = \mathcal{T}^{-1}(u + \delta) \end{cases}$$

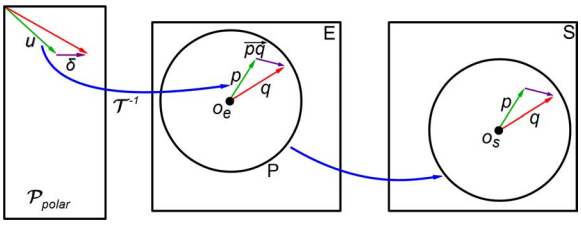

Figure 4: *Parameters used in $\mathcal{M}_{polar}$. p is the image of u by $\mathcal{T}^{-1}$, q is the image of $u+\delta$ by $\mathcal{T}^{-1}$, p and q have different directions for clarity reasons, the actual p and q have the same direction but different lengths. Note that $\vec{pq}$ is not the image of δ and unlike δ it is not a constant due to the non linearity of $\mathcal{T}$.*

u are coordinates in $\mathcal{P}_{polar}$. As before, $\delta=(1,0)$ when cutting vertically and $\delta=(0,1)$ when cutting horizontally. Figure 4 shows the parameters used in $\mathcal{M}_{polar}$.

Similarly, we define a polar version of $\mathcal{M}^S$ as follows:

$$\begin{cases} \mathcal{M}^S_{polar}(u,\delta)=\mathcal{N}(u)\times\mathcal{M}(S[o_s+p],o_s+q,\overrightarrow{pq}) \\ p=\mathcal{T}^{-1}(u) \\ q=\mathcal{T}^{-1}(u+\delta) \end{cases}$$

Patch cost We associate a quality cost to the patch $\mathcal{P}$. It consists in taking the cost of the seam $\mathcal{C}$ from which we subtract the existing costs on the left side $\mathcal{C}$ (costs inside $\mathcal{T}^{-1}(\mathcal{C})$). This leads to an energy function involving three terms: H, V and $\mathcal{E}$.

H represents the cost of horizontal transitions along $\mathcal{C}$ and is defined as:
$H(u)=N_\rho^2\times\mathcal{M}_{polar}(u,(1,0))$.
The constant N_ρ^2 cancels the term N_ρ in $\mathcal{N}$. It is used because $\mathcal{C}$ has a unit thickness and is not affected by N_ρ.

V represents the cost of vertical transitions along $\mathcal{C}$. Because a horizontal gap, that can be as long as J_{max} pixels, may appear between $\mathcal{C}[y]$ to $\mathcal{C}[y+1]$ (Figure 3 bottom–right), V has to sum up all the vertical costs along this gap. The gap starts at $x_1=\min(\mathcal{C}[y],\mathcal{C}[y+1])$ and ends at $x_2=\max(\mathcal{C}[y],\mathcal{C}[y+1])$. V is thus defined as:

$$\begin{cases} V(x_1,x_2,y)=\sum_{x=min(x_1,x_2)}^{max(x_1,x_2)}\mathcal{M}_{polar}((x,y),(0,1)) \;\; if\; y<\mathcal{H} \\ V(x_1,x_2,y)=0 \;\; otherwise \end{cases}$$

$\mathcal{E}$ represents existing errors in S. These existing errors lie on the left side of $\mathcal{C}$ and will be hidden by $\mathcal{P}$. This means that for the row u_y in $\mathcal{P}_{polar}$ all existing errors that precede $\mathcal{C}[u_y]$ will be subtracted. $\mathcal{E}$ is as follows:

$$\begin{cases} \mathcal{E}(u)=\sum_{x_i=1}^{u_x}(h+v) \\ h=\mathcal{M}^S_{polar}((x_i,u_y),(1,0)) \\ v=\mathcal{M}^S_{polar}((x_i,u_y),(0,1)) \end{cases}$$

We now define the energy $\mathscr{E}(\mathcal{C})$ as:

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

Our goal is to find a cut $\mathcal{C}$ such as $\mathscr{E}(\mathcal{C})$ is minimized.

Optimization with DP To optimize for $\mathscr{E}(\mathcal{C})$ with DP we pre–compute all sub–solutions in a table T using the following recurrence:

$$T[y,i]=\underset{j=i-J_{max}..i+J_{max}}{\arg\min}\;(T[y-1,j]+V(j,i,y))+H(i,y)-\mathcal{E}(i,y)$$

$T[y,i]$ is the cost of having $\mathcal{C}[y]=i$. $y\in\{1..\mathcal{H}\}$ and $i\in\{1..\mathcal{W}\}$.

Approximate cuts Our cut $\mathcal{C}$ is constrained to start and end at the same abscissa. This usually requires repeating the optimization of $\mathscr{E}(\mathcal{C})$ for all starting and ending abscissas as in drag–and–drop pasting [JSTS06]. Repeating the optimization requires an $O(\mathcal{W}^2\times\mathcal{H}\times J_{max})$ complexity. Instead, we propose an approximation that optimizes once for $\mathscr{E}(\mathcal{C})$ and reduces the complexity to $O(\mathcal{W}\times\mathcal{H}\times J_{max})$. The approximation is based on the following property of our DP:
by backtracking all the paths from bottom to top, there exists at least one path that starts and ends at the same abscissa.
This property is proved in Appendix B.

Approximate cut quality We have experimentally compared our approximate cut to the optimal one by running multiple tests on a large number of textures. The table below shows the average results of the experiment.

Number of textures	3000
Optimal cut average error	50.855
Approximate cut average error	53.633
Approximate cut average ranking	8.59/256
All cuts average error	81.962
Approximate cut ranked first	17.76%

Compared to the average, our approximate cut gives a low energy. It is indeed a shortest path and it quickly joins some of the few shortest paths along low–cost channels. However, there are rare cases where our approximation produces seams stronger than the ones produced by the optimal cut. Such seams are likely to be rejected (Section 6).

5. Feature alignment

Minimizing $\mathscr{E}(\mathcal{C})$ does not always guarantee a seamless result. This is especially the case if the texture contains aligned structural patterns like bricks or straws. For instance it is possible to offset the two overlapped textures of Figure 3 to produce a case where *any* cut optimization produces seams. In Figure 5 the overlapping is made so that the error map contains high–cost strips (the bright slanted strips in the figure) and our cut as well as graph–cut are constrained to cross these strips, therefore producing visible seams.

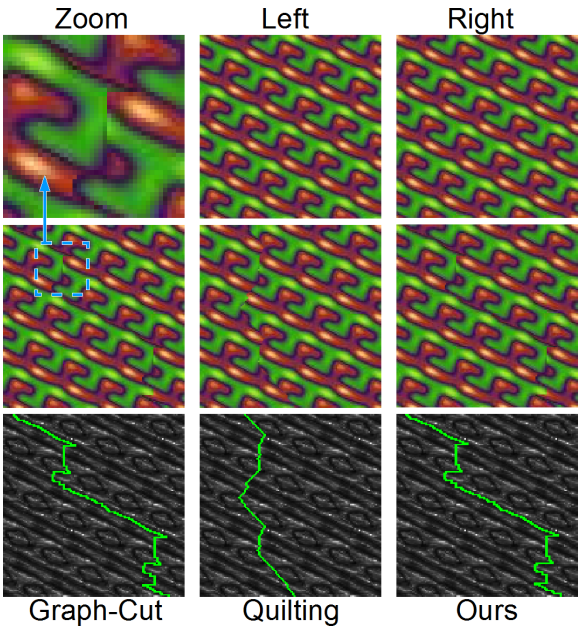

Figure 5: Top: *A left/right texture regions to be overlapped.* Middle: *From left to right: separating the two regions with graph–cut, image quilting and our cut using* $J_{max} = 16$. Bottom: *Error maps produced by the overlap. Each map sums up the vertical and the horizontal transition errors. Light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

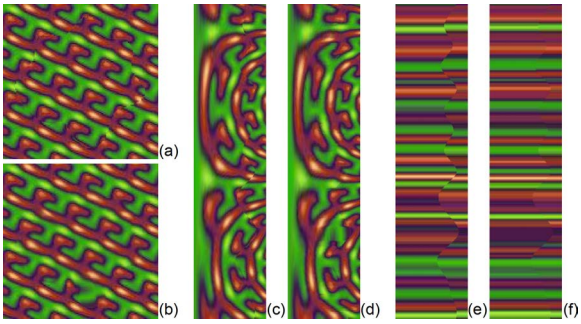

Figure 6: (a) *The patch* $\mathcal{P}$ *produces seams when placed on* $E[S]$. (b) *Result after feature alignment.* (c) *Polar space view before feature alignment.* (d) *Polar space view after feature alignment.* (e) *Colors along* $\mathcal{C}$ *before feature alignment.* (f) *Colors along* $\mathcal{C}$ *after feature alignment.*

If the cut optimization fails and produces seams, we propose to apply small deformations to align features on each side of $\mathcal{C}$. The deformation consists of two steps: In a first step, the colors in $\mathcal{P}_{polar}$ along $\mathcal{C}$ are offset to match the colors in $E[S]$ lying on the other side of $\mathcal{C}$ (Figure 6 **e** and **f**). In a second step a deformation is smoothly propagated to the inside of $\mathcal{P}$ (the left of $\mathcal{P}_{polar}$) based on color displacements made in the first step (Figure 6 **b**).

Offsetting colors along the cut Recall that the curve $\mathcal{C}$ is represented as an array where $\mathcal{C}[y]$ is the x coordinate of the curve at row y in $\mathcal{P}_{polar}$. Our goal is to offset the indices of $\mathcal{C}$ in order to to align features. We note $\mathcal{D}$ the array that contains the new indices in $\mathcal{C}$ after offsetting. i.e. the color at $\mathcal{P}_{polar}[\mathcal{C}[y],y]$ is replaced by the color at $\mathcal{P}_{polar}[\mathcal{C}[\mathcal{D}[y]],\mathcal{D}[y]]$.

As $\mathcal{C}$, $\mathcal{D}$ is accessed using wrap mode. We note $\mathcal{C}[\mathcal{D}]$ the cut with the offset colors but having the same shape as $\mathcal{C}$.

Since we propagate the deformation at the next stage, we take care that no fold–over occurs when optimizing for $\mathcal{D}$. For this we ensure that: $y \leq z \Rightarrow \mathcal{D}[y] \leq \mathcal{D}[z]$.

Recall that the shape of $\mathcal{C}$ can have a gap as long as J_{max} between $\mathcal{C}[y]$ and $\mathcal{C}[y+1]$. Since $\mathcal{C}$ only encodes one x coordinate per row in $\mathcal{P}_{polar}$, offsetting colors within the gap is difficult. We therefore ignore the cost of vertical transitions.

$\mathcal{D}$ is obtained by minimizing the following energy:

$$\mathcal{E}_{\mathcal{D}}(\mathcal{D}) = \sum_{y=1}^{\mathcal{H}} \mathcal{M}_{\mathcal{D}}(\mathcal{D}[y],y)$$

$\mathcal{M}_{\mathcal{D}}$ is the cost of replacing the color at coordinates $(\mathcal{C}[y],y)$ with the color at coordinates $(\mathcal{C}[\mathcal{D}[y]],\mathcal{D}[y])$. It is defined as:

$$\begin{cases} \mathcal{M}_{\mathcal{D}}(i,y) = \mathcal{M}(o_e+d, o_s+q, \overrightarrow{pq}) \\ p = \mathcal{T}^{-1}(\mathcal{C}[y],y) \\ q = \mathcal{T}^{-1}(\mathcal{C}[y]+1,y) \\ d = \mathcal{T}^{-1}(\mathcal{C}[i],i) \end{cases}$$

Since we are only interested in applying small deformations, we limit the maximum offset between two rows to be:
$\forall y \in \{1..\mathcal{H}-1\} \;\; \mathcal{D}[y+1]-\mathcal{D}[y] \leq 2$
The constant 2 has been chosen empirically. It represents the smallest possible offset. We also limit the maximum amount of deformation by setting the global constraint:
$\forall y \in \{1..\mathcal{H}\} \;\; |\mathcal{D}[y]-y| \leq D_{max}$
D_{max} is a constant set by the user to limit the maximum possible offset. By using a small value, we prevent strong deformations and reduce the required memory (The DP optimization table T has a size of $(2D_{max}+1)\times\mathcal{H}$).

Optimization with DP The optimization computes all sub–solutions using the recurrence:

$$T[y,i] = \underset{j=i-2..i}{\arg\min}(T[y-1,j]) + \mathcal{M}_{\mathcal{D}}(i,y)$$

$T[y,i]$ is the cost of having $\mathcal{D}[y]=i$. Notice how the DP minimizes $\mathcal{E}_{\mathcal{D}}(\mathcal{D})$ by performing 3 actions:

- $\mathcal{D}[y] = \mathcal{D}[y-1]$: repeating the same pixel.
- $\mathcal{D}[y] = \mathcal{D}[y-1]+1$: advance to the next pixel.
- $\mathcal{D}[y] = \mathcal{D}[y-1]+2$: advance twice (jumping one pixel).

Initial state The solution $\mathcal{D}$ must start and end at the same abscissa in T otherwise parts of the texture will be lost after propagation. We face the same issue we encountered during

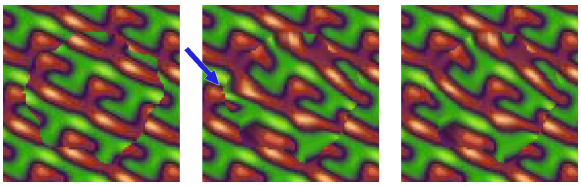

Figure 7: Left: *Random cut with no deformation.* Middle: *Deformation using a fixed initial state. The starting position is pointed by the blue arrow.* Right: *Deformation using our approximate solution for $\mathcal{D}$.*

the optimization of $\mathscr{E}(\mathcal{C})$. Repeating the optimization for all initial states is impractical. Starting from a random initial state could lead to artifacts (Figure 7). We therefore re–use the same approximate solution of $\mathscr{E}(\mathcal{C})$ optimization.

Deformation propagation The propagation is done by replacing the color at pixel (x,y) in $\mathcal{P}_{polar}$ with the color at pixel (x',y') using the following interpolation:

$$y' = lerp(y, \mathcal{D}[y], \left(\frac{x}{\mathcal{C}[y]}\right)^{\gamma})$$

$$x' = x \times \frac{\mathcal{C}[y']}{\mathcal{C}[y]}$$

γ controls the amount of deformation inside $\mathcal{P}_{polar}$. A small value will make the propagation spread further. $lerp$ function represents linear interpolation. It considers the fact that $\mathcal{P}_{polar}$ is vertically cyclic and always interpolates along the shortest path. e.g. $\mathcal{C}[0]$ is closer to $\mathcal{C}[\mathcal{H}]$ than $\mathcal{C}[\mathcal{H}/2]$.

6. Patch rejection

After the optimization, we decide to merge the patch with the result or reject it if it has no benefit to the overall quality.

We use two rejection policies: The first one is applied before feature alignment and is based on a simple predicate noted $isImproving_{before}(\mathcal{C})$ returning $true$ iff $\mathscr{E}(\mathcal{C}) \leq 0$. When $isImproving_{before}(\mathcal{C})$ is $true$ the subtracted existing errors in S, i.e. $\mathcal{E}$, are greater than the errors produced by $\mathcal{C}$ and in this case the patch corresponding to $\mathcal{C}$ provides a benefit and will be accepted. The patch will be rejected otherwise. Rejecting patches before feature alignment is a good heuristic. Accepted patches would have few seams and feature alignment performs well in these cases.

The second rejection is applied after feature alignment and requires changing the predicate as follows:

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

$\mathscr{E}(\mathcal{C}[\mathcal{D}])$ is the energy along $\mathcal{C}$ after color offsetting, $\mathscr{E}(\mathcal{P})$ is the total energy in $\mathcal{P}$ after deformation and $\mathcal{E}$ is the total existing energy on the left of $\mathcal{C}$. $isImproving_{after}$ requires some extra computations while $isImproving_{before}(\mathcal{C})$ comes practically for free. However $isImproving_{after}$ ensures a monotonically decreasing global energy.

7. Implementation details

We implement the algorithm using NVIDIA CUDA.
Multiple patches are processed simultaneously by placing their optimization tables one next to the other from left to right. For n patches, the global optimization table will have a size of $w \times h$ where $w = n \times \mathcal{W}$ and $h = \mathcal{H}$. Using this layout, the optimization is quite similar when processing a single or multiple patches. A thread just needs to know which patch is being processed to set the corresponding boundaries (The yellow dashed lines in Figure 8). Data beyond the boundaries are not accessed by the thread (clamp mode is used).

The algorithm executes two main optimizations: the optimization of $\mathscr{E}(\mathcal{C})$ followed by the optimization of $\mathscr{E}_{\mathcal{D}}(\mathcal{D})$. The buffers allotted for the optimization of $\mathscr{E}(\mathcal{C})$ will be re–used during the optimization of $\mathscr{E}_{\mathcal{D}}(\mathcal{D})$. We therefore make sure that there is enough memory for both optimizations.

We allocate the following *texture* buffers having a size of $w \times h$ each:
A buffer H that first stores existent horizontal costs then stores new horizontal costs, a buffer V that stores new vertical costs and a buffer $\mathcal{E}_v$ that stores existent vertical costs.

Prior to optimization, we start pre–computing all transition costs by performing the following steps:

- Fill H with horizontal existing costs and $\mathcal{E}_v$ with vertical existing costs (Pass1: one thread per entry in the table).
- For each row within each patch in H and $\mathcal{E}_v$, accumulate costs from left to right (Pass2: a thread per row of a patch).
- Fill V with zeros.
- Compute the new horizontal costs in H and the new vertical costs in V and subtract the existing content before storage (Pass3: one thread per entry in the table).

To efficiently process each pass within a same kernel H, V and $\mathcal{E}_v$ share the same texture unit. Texture memory is mainly used to avoid cache conflicts when accumulating existing errors (Pass2). After pre–computing the costs, H will be used as the main optimization table. V and $\mathcal{E}_v$ will be read–only and will provide the additional terms of $\mathscr{E}(\mathcal{C})$.

The DP optimization consists of a top–down sub–solution pre–computation in H followed by backtracking all cuts and storing the result in–place in H. Cuts that do not start and end at the same abscissa will be assigned an infinite cost. A reduction algorithm then selects for each patch the cut with the minimum cost.

The DP we solve has the property that each row y can be processed in parallel and only depends on the preceding row $y-1$. For the DP accumulation task a single row parallelism suffers from 3 limitations:

- By parallelizing a single row at a time, the number of threads will be too limited to fully exploit the GPU.
- After processing one row a global synchronization (stopping and re–running the kernel) is required before processing the next row.

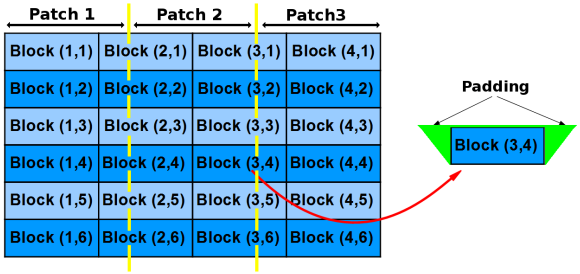

Figure 8: *The DP table is subdivided into blocks and each block is processed by a block of threads. A block of threads needs to access some data in the left and right blocks and the first row of above blocks (the green padded regions).*

- The synchronization suspends the computation within a column for a long time. Meanwhile other computations will fill–up the cache losing its coherence.

We address these limitations by processing many rows before making a global synchronization. This is done by subdividing H into blocks as shown Figure 8. A local DP accumulation is done in each block and the global synchronization happens after processing a line of blocks.

To correctly compute the result in each block, the blocks are padded with additional data (The green parts in Figure 8). These data belong to the left, right and above neighboring blocks. The padded data are only used to ensure correctness within each block. The additional computations within these padded regions are wasted but the overhead is small compared to the benefit of the increased coherence. We use one block of threads to process one block of data (including the padded regions). Because the padded data can be processed simultaneously by multiple blocks, they are first stored in a read–only temporary buffer. A block of threads is thus responsible of loading the data in shared memory and a subset of the threads applies the DP accumulation in this shared memory. The threads in the block finish by copying the processed data (without the padding) from shared memory back to H. The size of the blocks is determined empirically to be $B_w = 32 + J_{max}$ and $B_h = \frac{32\sqrt{J_{max}}-16}{J_{max}}$. The number of threads per block is also determined empirically by rounding–up the constant $32 + B_h \times J_{max}$ to the next multiple of 32.

The same process is used for the bottom–up backtracking with the difference that there is no padding. The temporary memory will now store the solution which is then copied back to the table H before a global synchronization. For the backtracking operation, if memory is not an issue one can eliminate the synchronization by allocating a table having the same size as H to store all the results.

The optimization of $\mathscr{E}_{\mathcal{D}}(\mathcal{D})$ is quite similar to the optimization of $\mathscr{E}(\mathcal{C})$. We can easily re–use the same DP. However since we are ignoring vertical transition costs when optimizing $\mathscr{E}_{\mathcal{D}}(\mathcal{D})$ and since only 3 actions are performed at each step, the buffers V and $\mathcal{E}_v$ are set to zero while $J_{max} = 1$.

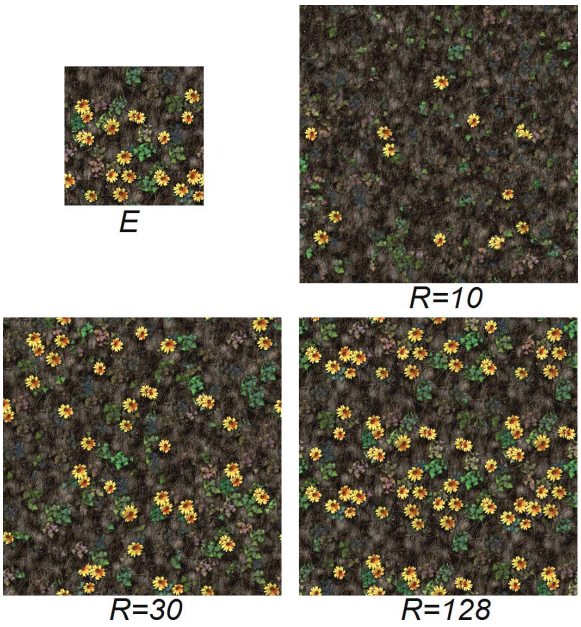

Figure 9: *Varying the maximum radius R.*

8. Results

In this section we use the following synthesis settings: S is initialized with white noise texture coordinates and it has a size of 512×512. $R = 64$, $D_{max} = 32$, $N_\rho = 2$, $N_\theta = 1.5$, $J_{max} = 4$ and $\gamma = 1$. We use the predicate $isImproving_{after}$ for rejection. We notify the reader if settings are changed. In general, the user can interactively tweak any parameter to improve the quality for a specific texture. User interactions are shown in the accompanying video.

Varying the maximum radius R The parameter with most effect on quality is the maximum patch radius R. Figure 9 shows synthesis results for the same texture but using different values for R. The synthesis is only iterated 8 times (it is necessary to iterate the synthesis a few times to cover the whole map S; 3 to 5 iterations are often enough to cover the whole map S with patches). When R is small the synthesizer fails to capture the flower patterns. It either breaks the flowers or rejects them. Because of rejection, the flowers will potentially disappear if the synthesis is further iterated. When R is large the synthesizer copies large patches from E producing a seamless result but having little variety. Setting R to be just large enough to capture the flowers results in a nice distribution of flowers. Please keep in mind that R is the maximum possible radius. A patch can be as small as one pixel regardless of R.

Automatic radius R We allow synthesis with an automatically decreasing radius R. Starting with a large value for R the synthesizer can quickly capture coarse structures like the canes in Figure 10. By decreasing R small patches are ac-

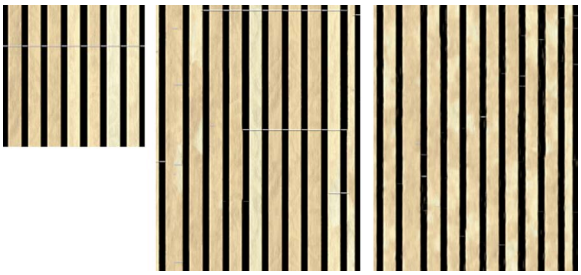

Figure 10: *Starting from a patch radius $R = 132$, R is decremented by 4 every iteration.* From left to right: *The exemplar E. Result when $R = 72$. Result when $R = 4$. Notice the variations within the canes when $R = 4$.*

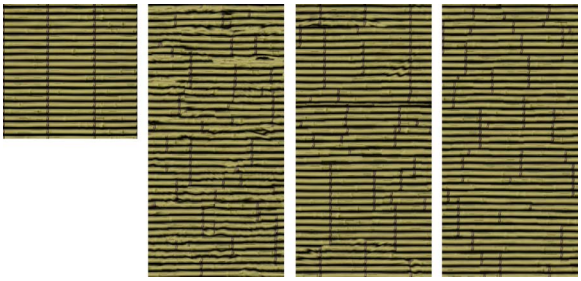

Figure 11: *Starting from a max deformation of $D_{max} = 160$, D_{max} is decremented by 10 every iteration.* From left to right: *The exemplar E, result when $D_{max} = 100$, $D_{max} = 50$ and $D_{max} = 0$. S has a size of* 256×512.

cepted making local variations in the result. Patch rejection ensures an unchanged structure during the late iterations.

Automatic D_{max} Similar to R, automatically reducing D_{max} starting from a large value can make the synthesis converge faster. Although the result is highly deformed during earlier iterations, the late iterations will focus on removing deformed features rather than avoiding seams (Figure 11).

Texture completion Our synthesizer allows for a straightforward texture completion application. If a texture contains holes it is easy to fill in these holes by setting an infinite existing cost in the holes. In practice we obtain the high cost by filling the holes with random coordinates. By running the synthesizer, patches are accepted within high cost areas corresponding to holes. It is also possible to prevent copying patches from the holes by providing a mask to the synthesizer. Figure 12 shows a completion result.

Multiple exemplars and texture painting One advantage of patch–based synthesis over pixel–based synthesis is the ease of using multiple exemplars. In pixel–based approaches the transition between different texture regions is not well defined and requires special handling. To use multiple exemplars the only requirement is to add a third coordinate in S to store the exemplar index. Figure 13 shows a synthesis

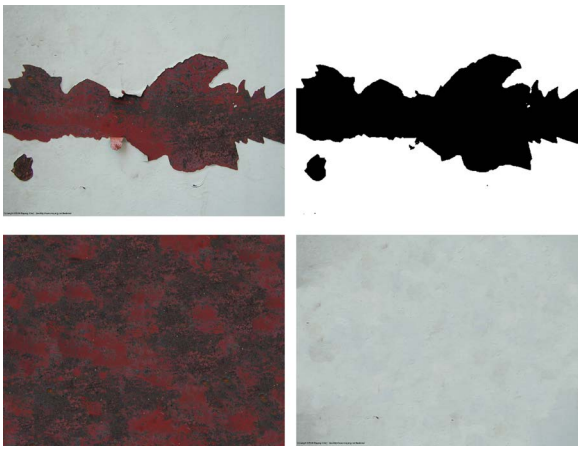

Figure 12: *Texture completion.* Top–left: *E.* Top–right: *mask.* Bottom–left: *Using the invert mask.* Bottom–right: *Using the mask. S has size* 1600×1200.

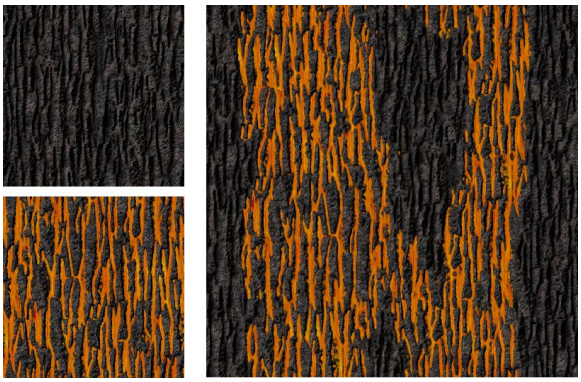

Figure 13: left: *Multiple exemplars.* right: *Synthesis result.*

result that uses patches from different textures.
Using multiple exemplars one can use a texture as a brush to paint on another texture. The painted zones are considered as holes and the synthesizer instantly fills these holes.

Patch drag–and–drop In a drag–and–drop task our fast synthesizer allows seeing the stitching result while dragging a patch. This gives more intuition to the user on where to place the patch. We disable rejection for this task. In addition the user can set a minimum patch radius in which the existing costs are forced to be zero. The accompanying video shows an example.

Synthesis convergence The plot in Figure 14 shows the global cost evolution and the patch acceptance rate during the iterative synthesis process. At the tenth iteration the synthesis went near a local minimum dropping the patch acceptance rate and making the result almost unchanged during the next iterations. Manually changing synthesis parameters or disabling rejection is a simple way to get away from local minima. Please note that convergence could vary from one

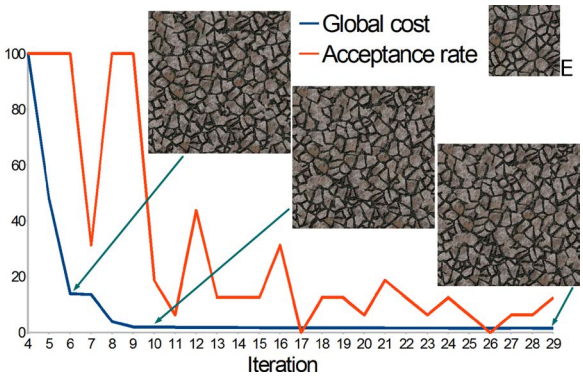

Figure 14: *Global cost and patch acceptance rate evolution. The global error is scaled to be in range [0..100]. The first 4 iterations produce very high errors and are ignored.*

texture to another. In addition, note that the algorithm starts from random coordinates in S (worst initialization) and that other initializations may improve convergence.

Performance The following table lists the execution time and the used memory for one iteration running on an Nvidia GeForce GTX580. Recall that $R = 64$.

size of S	256	512	1024	2048
used memory in MB	<4	14	56	226
iteration time in ms	6	6	25	59
$\mathcal{C}$ DP initialization	9%	19%	29%	51%
$\mathcal{C}$ DP optimization	27%	22%	11%	12%
$\mathcal{D}$ DP initialization	3%	5%	8%	14%
$\mathcal{D}$ DP optimization	28%	22%	10%	8%

In the table above one can notice that the cost of DP initialization grows quickly with S. This is due to the intensive evaluation of $\mathcal{M}_{polar}$ which requires many texture fetches in E and S in addition to multiple calls to $\mathcal{T}$. The DP optimization scales well because the number of patches increases when the size of S increases. Increasing the number of patches enlarges the optimization table x axis and therefore increases the degree of parallelism. However for a same S using more patches also requires more memory. The number of patches can be reduced by increasing R. However this increases the optimization table y axis and reduces the degree of parallelism. Figure 15 shows the DP performance while R increases.

Comparison with pixel–based synthesis Our algorithm is roughly ten times slower than fast per–pixel synthesizers due to the number of iterations required before convergence. For instance, our GPU implementation of [LH05] synthesizes the texture of Figure 16 in 15 ms using only 4 iterations while our method needs 16 iterations to correctly align features and this requires 112 ms of computation. Nevertheless, our implementation achieves interactivity and inherits the benefits of patch–based approaches. In particular, the quality achieved by fast per–pixel algorithms largely depends on

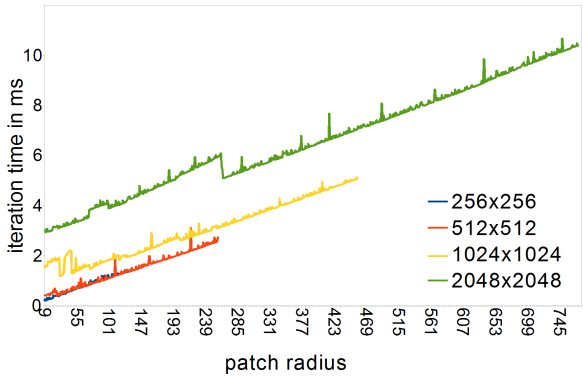

Figure 15: *Effect of increasing the maximum patch radius R. Each curve corresponds to one resolution of S. The optimization table uses a much larger resolution because of $\mathcal{T}$ and the factors N_ρ and N_θ.*

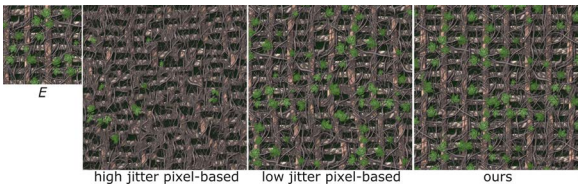

Figure 16: *Comparison with pixel–based synthesis.* From left to right: *E having a size of 256^2. Pixel–based result computed in 17 ms using a high jitter. Pixel–based result computed in 15 ms using a low jitter. Our result computed in 112 ms using $R = 128$. S has a size of 512^2.*

the amount of jitter added during multi–resolution synthesis. This has to be carefully selected by the user and differs for each texture. Similarly, structured patterns require the addition of a feature distance (see Figure 1). Our algorithm offers superior quality without this requirement. To the best of our knowledge no fast per–pixel algorithm *automatically* reaches the quality we demonstrate on highly regular patterns such as the ones shown in Figures 10,11.

9. Discussion

We introduced a parallel patch–based texture synthesis algorithm that quickly achieves high quality results and enables interactive controls. Our algorithm relies on a parallel implementation of an approximate boundary optimization, as well as a patch deformation to align features. A patch rejection scheme ensures a progressively improving synthesis quality.

The main limitations of our synthesizer stem from the random patch selection and placement. Quality–wise, global structures in textures cannot be preserved as shown in Figure 17. Performance–wise, despite fast iterations global convergence is slowed–down by the high patch rejection rate as shown in Figure 14. As future work we would like to improve the sampling: First, we could build upon ideas devel-

Figure 17: *Our synthesizer cannot capture global structures like doors.* Left: E. Right: $E[S]$.

opped in [BSFG09] to propagate good choices of patches throughout the grid. Second, rather than testing a single patch per cell we could easily test several patches, keeping only the best (if any). This would enable adaptive sampling, testing more patches in areas of high error and thereby improving the acceptance probability.

Acknowledgments

We thank Allegorithmic for providing the textures used in Figures 1, 9, 10, 11, 13, 14, 16, 17 and mayang.com/textures/ for providing the texture used in Figure 12; Samuel Hornus, Ismael Garcia and Nicolas Bonneel for discussions; Cyrille Damez for discussions and proofreading the paper; This work was supported by the ANR SIMILAR-CITIES 2008-COORD-021-01.

References

[ADA*04] AGARWALA A., DONTCHEVA M., AGRAWALA M., DRUCKER S., COLBURN A., CURLESS B., SALESIN D., COHEN M.: Interactive digital photomontage. 2

[Ash01] ASHIKHMIN M.: Synthesizing natural textures. In *I3D* (2001). 1

[BB81] BAKER H. H., BINFORD T. O.: Depth from edge and intensity based stereo. In *IJCAI* (1981). 2

[Bel54] BELLMAN R.: The theory of dynamic programming. *Bull. Amer. Math. Soc* (1954). 10

[BSFG09] BARNES C., SHECHTMAN E., FINKELSTEIN A., GOLDMAN D. B.: PatchMatch: A randomized correspondence algorithm for structural image editing. *Transactions on Graphics* (2009). 1, 2, 10

[BVZ99] BOYKOV Y., VEKSLER O., ZABIH R.: Fast approximate energy minimization via graph cuts. In *ICCV* (1999). 2

[EF01] EFROS A. A., FREEMAN W. T.: Image quilting for texture synthesis and transfer. In *SIGGRAPH* (2001). 2

[GSX] GUOA B., SHUM H., XU Y.-Q.: Chaos mosaic: Fast and memory efficient texture synthesis. Microsoft Research technical report MSR-TR-2000-32. 2

[JSTS06] JIA J., SUN J., TANG C.-K., SHUM H.-Y.: Drag-and-drop pasting. *Transactions on Graphics* (2006). 2, 4

[JT05] JIA J., TANG C.: Eliminating structure and intensity misalignment in image stitching. In *ICCV* (2005). 2

[KSE*03] KWATRA V., SCHÖDL A., ESSA I., TURK G., BOBICK A.: Graphcut textures: Image and video synthesis using graph cuts. *Transactions on Graphics* (2003). 1, 2

[LH05] LEFEBVRE S., HOPPE H.: Parallel controllable texture synthesis. *SIGGRAPH* (2005). 1, 9

[LH06] LEFEBVRE S., HOPPE H.: Appearance-space texture synthesis. *Transactions on Graphics* (2006). 1, 2

[LZPW04] LEVIN A., ZOMET A., PELEG S., WEISS Y.: Seamless image stitching in the gradient domain. *ECCV* (2004). 2

[MZD05] MATUSIK W., ZWICKER M., DURAND F.: Texture design using a simplicial complex of morphable textures. *Transactions on Graphics* (2005). 2

[PELS10] PANAREDA BUSTO P., EISENACHER C., LEFEBVRE S., STAMMINGER M.: Instant Texture Synthesis by Numbers. *Vision, Modeling & Visualization* (2010). 1

[PFH00] PRAUN E., FINKELSTEIN A., HOPPE H.: Lapped textures. In *SIGGRAPH* (2000). 2

[PGB03] PÉREZ P., GANGNET M., BLAKE A.: Poisson image editing. 2

[WLKT09] WEI L.-Y., LEFEBVRE S., KWATRA V., TURK G.: State of the art in example-based texture synthesis. In *Eurographics, EG-STAR* (2009). 1

[WY04] WU Q., YU Y.: Feature matching and deformation for texture synthesis. *Transactions on Graphics* (2004). 2

Appendix A: Normalization Factor

The normalization factor $\mathcal{N}$ is used to account for distortions and for the scale factors N_ρ and N_θ. $\mathcal{N}$ makes sure that the total cost inside $\mathcal{T}^{-1}(\mathcal{C})$ in $\mathcal{P}$ matches the total cost in the left area of $\mathcal{C}$ in $\mathcal{P}_{polar}$ (energy preservation). It is therefore the solution of the equality:
$\sum_{u\in\mathcal{P}_{polar}} \mathcal{N}(u)\times(\mathcal{M}_{polar}(u,(1,0))+\mathcal{M}_{polar}(u,(0,1))) =$
$\sum_{u\in\mathcal{P}} \mathcal{M}(u,u+(1,0),(1,0))+\mathcal{M}(u,u+(0,1),(0,1))$

Appendix B: Approximate cut existence

To prove the existence of our approximate cut, we note $\mathcal{C}_{ij}$ the path starting at abscissa i and ending at abscissa j in $\mathcal{P}_{polar}$. $\mathcal{C}_{ii}$ is a closed cut starting and ending at i. In our DP two shortest paths cannot *cross* (principal of optimality [Bel54]). If two sub-paths meet, they will continue along the same sub–optimal path.

Let us assume that there is no closed cut $\mathcal{C}_{ii}$. We first prove by induction that in this case for any path $\mathcal{C}_{nk_n}$ with $n \geq 1$ we have $k_n > n$.

The very first path $\mathcal{C}_{1k_1}$ is not a closed cut, and we necessarily have $k_1 > 1$. Now, consider a path $\mathcal{C}_{nk_n}$ with $n > 1$ and assume that $k_n > n$. The next path $\mathcal{C}_{n+1k_{n+1}}$ has to be such that $k_{n+1} \geq k_n$ otherwise $\mathcal{C}_{nk_n}$ and $\mathcal{C}_{n+1k_{n+1}}$ would cross each other. If $k_n > n+1$ then $k_{n+1} > n+1$ since $k_{n+1} \geq k_n$. If $k_n = n+1$ then we have $k_{n+1} \geq n+1$. However, since there is no closed cut, it follows that $k_{n+1} > n+1$. The first proof is complete.

Let us now consider the very last path: $\mathcal{C}_{\mathcal{W}k_\mathcal{W}}$. We have $k_\mathcal{W} > \mathcal{W}$ which is impossible since the path would exit $\mathcal{P}_{polar}$. Therefore, a path $\mathcal{C}_{ii}$ has to exist. This proves the existence of at least one cut $\mathcal{C}_{ii}$.