Parallel Controllable Texture Synthesis

Sylvain Lefebvre Hugues Hoppe
Microsoft Research

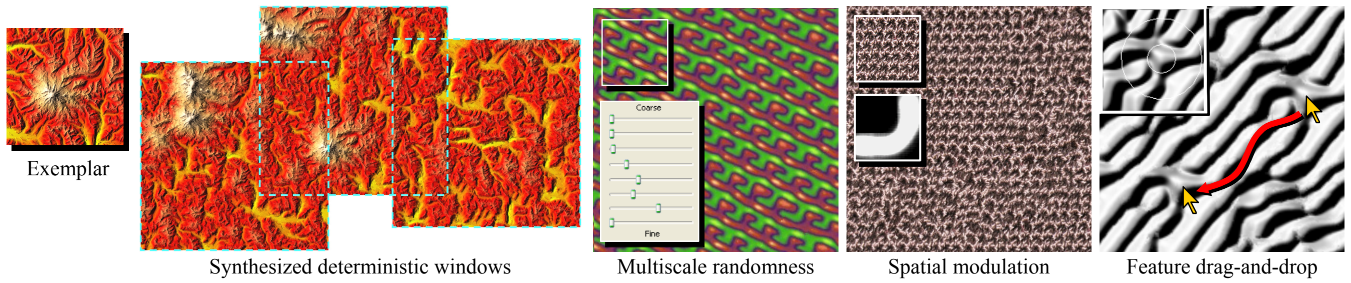


Figure 1: Given a small exemplar image, our parallel synthesis algorithm computes windows of spatially deterministic texture from an infinite landscape in real-time. Synthesis variation is obtained using a novel jittering technique that enables several intuitive controls.

Abstract

We present a texture synthesis scheme based on neighborhood matching, with contributions in two areas: parallelism and control. Our scheme defines an infinite, deterministic, aperiodic texture, from which windows can be computed in real-time on a GPU. We attain high-quality synthesis using a new analysis structure called the Gaussian stack, together with a coordinate upsampling step and a subpass correction approach. Texture variation is achieved by multiresolution jittering of exemplar coordinates. Combined with the local support of parallel synthesis, the jitter enables intuitive user controls including multiscale randomness, spatial modulation over both exemplar and output, feature drag-and-drop, and periodicity constraints. We also introduce synthesis magnification, a fast method for amplifying coarse synthesis results to higher resolution.

Keywords: runtime content synthesis, data amplification, Gaussian stack, neighborhood matching, coordinate jitter, synthesis magnification.

1. Introduction

Sample-based texture synthesis analyzes a given exemplar to create visually similar images. In graphics, these images often contain surface attributes like colors and normals, as well as displacement maps that define geometry itself. Our interest is in applying synthesis to define infinite, aperiodic, deterministic content from a compact representation. Such data amplification is particularly beneficial in memory-constrained systems. Thanks to growing processor parallelism, we can now envision sophisticated techniques for on-demand content synthesis at runtime.

There are several approaches to sampled-based texture synthesis, as reviewed in Section 2. While tiling methods are the fastest, and patch optimization methods produce some of the best results, neighborhood-matching algorithms allow greater fine-scale adaptability during synthesis.

In this paper, we present a new neighborhood-matching method with contributions in two areas: efficient parallel synthesis and intuitive user control.

Parallel synthesis. Most neighborhood-matching synthesis algorithms cannot support parallel evaluation because their sequential assignment of output pixels involves long chains of causal dependencies. For the purpose of creating large environments, such sequential algorithms have two shortcomings:

(1) It is impractical to define huge deterministic landscapes because the entire image must be synthesized at one time, i.e. one cannot incrementally compute just a window of it.

(2) The computation cannot be mapped efficiently onto a parallel architecture like a GPU or multicore CPU.

Our method achieves parallelism by building on the order-independent texture synthesis scheme of Wei and Levoy [2003]. They perform synthesis using a multiscale pyramid, applying multiple passes of pixel correction at each pyramid level to match neighborhoods of the exemplar [Popat and Picard 1993; Wei and Levoy 2000]. Their crucial innovation is to perform correction on all pixels *independently*, to allow deterministic synthesis of pixels in arbitrary order. They investigate a texture synthesis cache for on-demand per-pixel synthesis in rasterization and ray tracing.

We extend their approach in several directions. We improve synthesis quality using three novel ideas:

- *Gaussian image stack*: During texture analysis, we (conceptually) capture Gaussian pyramids shifted at all locations of the exemplar image, to boost synthesis variety.
- *Coordinate upsampling*: We initialize each pyramid level using coordinate inheritance, to maintain patch coherence.
- *Correction subpasses*: We split each neighborhood-matching pass into several subpasses to improve correction. Surprisingly, the results surpass even a traditional sequential traversal.

Moreover, by evaluating texture windows rather than pixel queries, we are able to cast synthesis as a parallel SIMD computation. We adapt our scheme for efficient GPU evaluation using several optimizations. Our system generates arbitrary windows of texture from an infinite deterministic canvas in real-time – 256^2 pixels in 26 msec. For continuous window motions, incremental computation provides further speedup.

User control. While many synthesis schemes offer some forms of user guidance (as discussed in Section 2), they provide little control over the amount of texture variability. Typically, output variation is obtained by random seeding of boundary conditions. As one modifies the random seeds or adjusts algorithmic parameters, the synthesized result changes rather unpredictably.

We introduce an approach for more explicit, intuitive control. The key principle is *coordinate jitter* – achieving variation *solely* by perturbing exemplar coordinates at each level of the synthesized pyramid. We initialize each level by simple coordinate inheritance, so by design our scheme produces a tiling in the absence of jitter. And, the tiles equal the exemplar if it is toroidal. Starting with this simple but crucial result, randomness can be gradually added at any resolution, for instance to displace the macro-features in the texture, or to instead alter their fine detail.

We expose a set of continuous sliders that control the magnitude of random jitter at each scale of synthesis (Figure 1). Because parallel synthesis has local support, the output is quite coherent with respect to continuous changes in jitter parameters, particularly in conjunction with our new Gaussian image stack.

Multiresolution coordinate jitter also enables several forms of local control. It lets randomness be adjusted spatially over the source exemplar or over the output image. The jitter can also be overridden to explicitly position features, through a convenient *drag-and-drop* user interface. Thanks to the multiscale coherent synthesis, the positioned features blend seamlessly with the surrounding texture. Finally, the jittered coordinates can be constrained to more faithfully reconstruct near-regular textures. For all these control paradigms, real-time GPU evaluation provides invaluable feedback to the user.

Synthesis magnification. A common theme in our contributions is that the primary operand of synthesis is exemplar coordinates rather than color. As another contribution along these lines, we introduce a fast technique for generating high-resolution textures. The idea is to interpret the synthesized coordinates as a 2D patch parametrization, and to use this map to efficiently sample a higher-resolution exemplar. This magnification is performed in the final surface shader and thus provides additional data amplification with little memory cost.

2. Related work

There are a number of approaches for sampled-based synthesis.

Image Statistics. Texture can be synthesized by reproducing joint statistics of the exemplar [e.g. Zalesny and Van Gool 2001].

Precomputed tiles. Cohen et al [2003] precompute a set of Wang Tiles designed to abut seamlessly along their boundaries. With a complete tile set, runtime evaluation is simple and parallel, and is therefore achievable in the GPU pixel shader [Wei 2004]. Some coarse control is possible by transitioning between tiles of different textures [Cohen et al 2003; Lefebvre and Neyret 2003]. The main drawback of tile-based textures is their limited variety due to the fixed tile set. Also, the regular tiling structure may become apparent when the texture is viewed from afar, especially for non-homogeneous textures.

Patch optimization. Texture is created by iteratively overlapping irregular patches of the exemplar [Praun et al 2000] to minimize overlap error [Liang et al 2001]. Inter-patch boundaries are improved using dynamic programming [Efros and Freeman 2001] or graph cut [Kwatra et al 2003]. Patch layout is a nontrivial optimization, and is therefore precomputed. The layout process seems to be inherently sequential. Control is possible by letting the user override the delineation and positioning of patches.

Neighborhood matching. The texture is typically generated one pixel at a time in scanline or spiral order. For each pixel, the partial neighborhood already synthesized is compared with exemplar neighborhoods to identify the most likely pixels, and one is chosen at random [Garber 1981; Efros and Leung 1999]. Improvements include hierarchical synthesis [Popat and Picard 1993], fast VQ matching [Wei and Levoy 2000], coherent synthesis to favor patch formation [Ashikhmin 2001], and precomputed similarity sets [Tong et al 2002; Zelinka and Garland 2002].

Very few neighborhood-matching schemes offer the potential for synthesis parallelism. One is the early work of De Bonet [1997] in which pyramid matching is based solely on ancestor coordinates. The other is the order-independent approach of Wei and Levoy [2003] which considers same-level neighbors in a multi-pass neighborhood correction process.

One advantage of neighborhood-matching schemes is their flexibility in fine-scale control. For instance, Ashikhmin [2001] lets the user guide the process by initializing the output pixels with desired colors. The image analogies framework of Hertzmann et al [2001] uses a pair of auxiliary images to obtain many effects including super-resolution, texture transfer, artistic filters, and texture-by-numbers. Tonietto and Walter [2002] smoothly transition between scaled patches of a texture to locally control the pattern scale. Zhang et al [2003] synthesize binary texton masks to maintain integrity of texture elements, locally deform their shapes, and transition between two homogeneous textures. Our contribution is control over the magnitude of texture variability (both spectrally and spatially), and our approach can be used in conjunction with these other techniques.

For an incrementally moving window, one might consider filling the exposed window region by applying existing constrained synthesis schemes [e.g. Efros and Leung 1999; Liang et al 2001]. Note however that the resulting data then lacks spatial determinism because it depends on the motion path of the window, unlike in parallel synthesis.

3. Parallel synthesis method

3.1 Basic scheme

From an $m \times m$ exemplar image E, we synthesize an image S in which each pixel $S[p]$ stores the coordinates u of an exemplar pixel (where both $p,u \in \mathbb{Z}^2$). Thus, the color at pixel p is given by $E[u] = E[S[p]]$.

Overview. As illustrated in Figure 2, we apply traditional hierarchical synthesis, creating an image pyramid S_0, S_1, …, (S_L=S) in coarse-to-fine order, where L=log$_2m$.

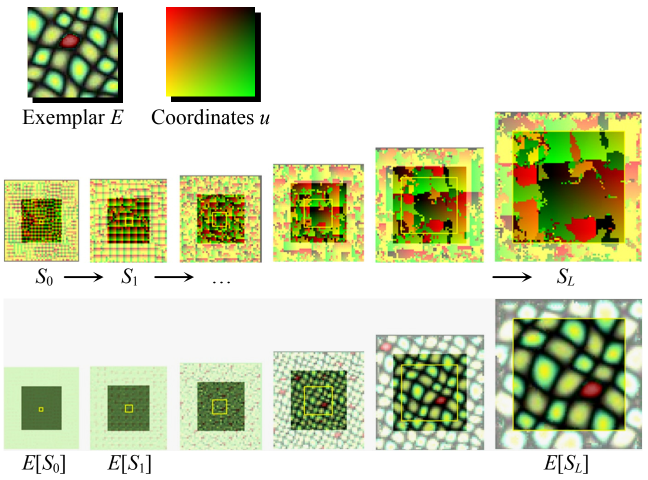

Exemplar E
Coordinates u
$S_0 \longrightarrow S_1 \longrightarrow \ldots \longrightarrow S_L$
$E[S_0]$
$E[S_1]$
$E[S_L]$

Figure 2: Given an exemplar, we synthesize its coordinates into a coarse-to-fine pyramid; the bottom row shows the corresponding exemplar colors. As explained later in Section 3.4, deterministically evaluating the requested (yellow) pyramid requires a broader pyramid padded with some indeterministic (hazy) pixels.

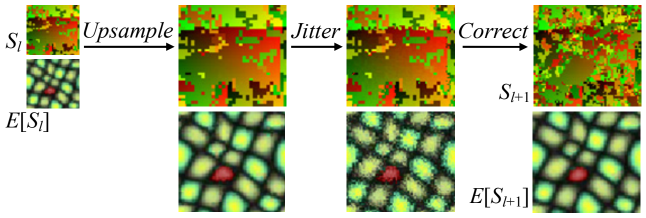


Figure 3: The three steps of synthesis at each pyramid level.

For each pyramid level, we perform three steps (Figure 3): *upsampling*, *jitter*, and *correction*. As in [Wei and Levoy 2003], the correction step consists of a sequence of passes, where within a pass, each pixel is replaced *independently* by the exemplar pixel whose neighborhood best matches the current synthesized neighborhood. Here is pseudocode of the overall process:

```
Synthesize()
  $S_{-1}$ := $(0\ 0)^T$                // Start with zero 2D coordinates.
  for $l \in \{0...L\}$                // Traverse levels coarse-to-fine.
    $S_l$ := Upsample($S_{l-1}$)       // Upsample the coordinates.
    $S_l$ := Jitter($S_l$)             // Perturb the coordinates.
    if ($l$>2)                         // For all but three coarsest levels,
      for {1..$c$}                     //  apply several correction passes,
        $S_l$ := Correct($S_l$)        //  matching exemplar neighborhoods.
  return $S_L$
```

Previous hierarchical techniques represent the exemplar using a Gaussian pyramid E_0, E_1, …, (E_L=E). We first describe our technique using this conventional approach, and in Section 3.2 modify it to use the new *Gaussian stack* structure. To simplify the exposition, we let h_l denote the regular output spacing of exemplar coordinates in level l: $h_l = 1$ for a pyramid and $h_l = 2^{L-l}$ for a stack.

Upsampling. Rather than using a separate synthesis pass to create the finer image from the next-coarser level, we simply upsample the *coordinates* of the parent pixels. Specifically, we assign to each of the four children the scaled parent coordinates plus a child-dependent offset:

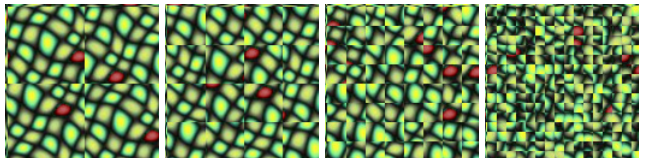
$$S_l[2p+\Delta] := \left(2\,S_{l-1}[p] + h_l\,\Delta\right) \bmod m\ , \quad \Delta \in \left\{ \begin{pmatrix}0\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}, \begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}1\\1\end{pmatrix} \right\}.^{1}$$

If the exemplar is toroidal and jitter is disabled, successive upsamplings create the image $S_L[p] = p \bmod m$, which corresponds to tiled copies of the exemplar E; the correction step then has no effect because all neighborhoods of S_L are present in E.

Jitter. To introduce spatially deterministic randomness, we perturb the upsampled coordinates at each level by a jitter function, which is the product of a hash function $\mathcal{H} : \mathbb{Z}^2 \to [-1,+1]^2$ and a user-specified per-level randomness parameter $0 \le r_l \le 1$:

$$S_l[p] := \left(S_l[p] + J_l(p)\right) \bmod m\ , \text{ where } J_l(p) = \left\lfloor h_l\,\mathcal{H}(p)\,r_l + \begin{pmatrix}.5\\.5\end{pmatrix} \right\rfloor .$$

Note that the output-spacing factor h_l reduces the jitter amplitude at finer levels, as this is generally desirable. If the correction step is turned off, the effect of jitter at each level looks like a quadtree of translated windows in the final image:



Jitter at coarsest level… …Jitter at fine levels

[1] We use "$u \bmod m$" and $\lfloor u \rfloor$ to denote per-coordinate operations.

Correction. The correction step takes the jittered coordinates and alters them to recreate neighborhoods similar to those in the exemplar. Because the output pixels cannot consider their simultaneously corrected neighbors, several passes of neighborhood matching are necessary at each level to obtain good results. We generally perform two correction passes.

For each pixel p, we gather the pixel colors of its 5×5 neighborhood at the current level, represented as a vector $N_{S_l}(p)$. This neighborhood is compared with exemplar neighborhoods $N_{E_l}(u)$ to find the L_2 best matching one.

To accelerate neighborhood matching, we use *coherent synthesis* [Ashikhmin 2001], only considering those locations u in the exemplar given by the 3×3 immediate neighbors of p. To provide additional variety, we use *k-coherence* search [Tong et al 2002], precomputing for each exemplar pixel u a candidate set $C^l_{1\ldots k}(u)$ of k exemplar pixels with similar 7×7 neighborhoods. We set k=2. For good spatial distribution [Zelinka and Garland 2002], we require the k neighborhoods to be separated by at least 5% of the image size. The first entry is usually the identity $C^l_1(u) = u$. We favor patch formation by penalizing jumps ($C^l_{2\ldots k}$) using a parameter κ as in [Hertzmann et al 2001].

These elements are captured more precisely by the expression

$$S_l[p] \;:=\; C^l_{i_{\min}}\left(S_l[p+\Delta_{\min}] - h_l\,\Delta_{\min}\right), \text{ where}$$

$$i_{\min}, \Delta_{\min} \;=\; \underset{\substack{i \in \{1..k\} \\ \Delta \in \left\{ \binom{-1}{-1}, \binom{-1}{0}, \ldots, \binom{+1}{+1} \right\}}}{\operatorname{argmin}} \left\| N_{S_l}(p) - N_{E_l}\left(C^l_i\left(S_l[p+\Delta] - h_l\,\Delta\right)\right) \right\| \varphi(i)$$

$$\text{in which } \varphi(i) = \begin{cases} 1, & i = 1 \\ 1+\kappa, & i > 1 \end{cases}.$$

3.2 Gaussian image stack

For exemplar matching, the traditional approach is to construct a Gaussian image pyramid E_0, E_1, …, (E_L=E) with the same structure as the synthesis pyramid. However, we find that this often results in synthesized features that align with a coarser "grid" (Figure 6a) because ancestor coordinates in the synthesis pyramid are snapped to the quantized positions of the exemplar pyramid. One workaround would be to construct a few Gaussian pyramids at different locations within the exemplar, similar to the multiple sample pyramids used in [Bar-Joseph et al 2001].

Our continuous jitter framework has led us to a more general solution, which is to allow synthesized coordinates u to have *fine* resolution at all levels of synthesis. Of course, we should then analyze additional exemplar neighborhoods. Ideally, we desire a full family of m^2 Gaussian pyramids, one for each translation of the exemplar image at its finest level. Fortunately, the samples of *all* m^2 exemplar pyramids actually correspond to a single stack of $\log_2 m$ images of size m×m, which we call the *Gaussian stack* (see illustration in Figure 4 and example in Figure 5).

To create the Gaussian stack, we augment the exemplar image on all sides to have size 2m×2m. These additional image samples come from one of three sources: an actual larger texture, a tiling if the exemplar is toroidal, or else reflected copies of the exemplar. We then apply traditional Gaussian filtering to form each coarser level, but *without* subsampling.

To use the Gaussian stack in our synthesis scheme, the algorithm from Section 3.1 is unchanged, except that we reassign $h_l = 2^{L-l}$ and replace the upsampling step to account for the parent-child relations of the stack:

$$S_l[2p+\Delta] := \left(S_{l-1}[p] + \left\lfloor h_l \left(\Delta - \begin{pmatrix}.5\\.5\end{pmatrix} \right) \right\rfloor \right) \bmod m\ .$$

If the exemplar is non-toroidal, synthesis artifacts occur when the upsampled coordinates of four sibling pixels span a "mod m" boundary. Our solution is to constrain the similarity sets such that $C_i(\cdot) \neq u$ for any pixel u near the exemplar border (colored red in Figure 4). The effect is that the synthesized texture is forced to jump to another similar neighborhood away from the boundary.

At the coarsest level (l=0), all Gaussian stack samples equal the mean color of the exemplar image, so correction has no effect and we therefore disable it. In practice, we find that correction at the next two levels (l=1,2) is overly restrictive as it tends to lock the alignment of coarse features, so to allow greater control we disable correction on those levels as well.

Figure 6b shows the improvement in synthesis quality due to the stack structure. The effect is more pronounced during interactive control as seen on the video.

The quadtree pyramid [Liang et al 2001] is a related structure that stores the same samples but in a different order – an array of smaller images at each level rather than a single image. Because the quadtree samples are not spatially continuous with respect to fine-scale offsets, jitter would require more complex addressing.

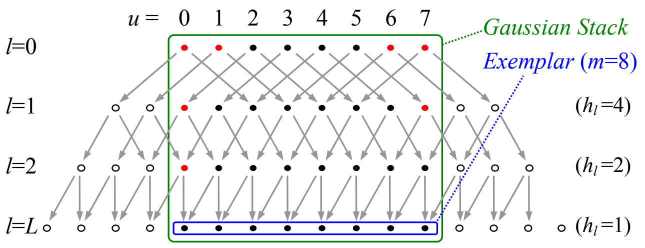


Figure 4: The Gaussian stack (shown here in 1D) is formed by a union of pyramids shifted at all image locations. It is constructed from an augmented exemplar image in the bottom row.

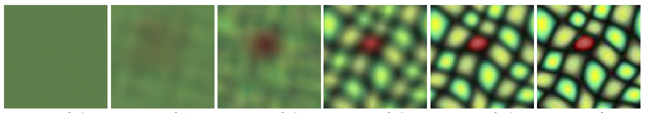
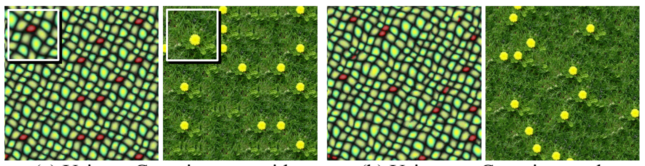


Figure 5: The Gaussian stack captures successively larger filter kernels *without the subsampling* of Gaussian pyramids.



(a) Using a Gaussian pyramid (b) Using our Gaussian stack

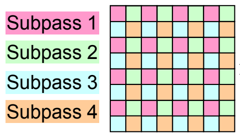
Figure 6: Compared to a traditional Gaussian pyramid, the Gaussian stack analysis structure leads to a more spatially uniform distribution of synthesized texture features.

3.3 Correction subpasses

When correcting all pixels simultaneously, one problem is that the pixels are corrected according to neighborhoods that are also changing. This may lead to slow convergence of pixel colors, or even to cyclic behavior.

We improve results by partitioning a correction pass into a sequence of subpasses on subsets of nonadjacent pixels. Specifically, we apply s^2 subpasses, each one processing the pixels p such that $p \bmod s = (i \;\; j)^T$, $i,j \in \{0 \ldots s{-}1\}$.

The evaluation order of the subpasses can be represented graphically as an $s{\times}s$ matrix, shown here for s^2=4:



represented as $\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}$.

There are a number of factors to consider in selecting the number and order of the subpasses (see Figure 7):

- Synthesis quality improves with more subpasses, although not much beyond s^2=9.
- A traditional sequential algorithm is similar to a large number of subpasses applied in scanline order. Interestingly, this yields worse results. The intuitive explanation is that a scanline order gives fewer opportunities to "go back and fix earlier mistakes", and consequently it is important to shuffle the subpass order.
- On the GPU, each subpass requires a SetRenderTarget() call, which incurs a small cost.
- For spatially deterministic results (Section 3.4), each subpass requires additional padding of the synthesis pyramid. For example, this cost becomes evident at {c=8, s^2=4} in Figure 7.
- Even as neighborhood error decreases with many correction passes, the texture may begin to look less like the exemplar, because correction may have a bias to disproportionately use subregions of the exemplar that join well together. For example, in Figure 7 the "ripple" features become straighter. Removing this bias is an interesting area for future work.

These factors present a tradeoff between performance and quality. We find that using two correction passes and s^2=4 subpasses provides a good compromise. We select the following subpass order to reduce the necessary pyramid padding:

$$\begin{pmatrix}1 & 4\\ 3 & 2\end{pmatrix}, \begin{pmatrix}6 & 8\\ 7 & 5\end{pmatrix}.$$

In practice, multiple subpasses result in a significant improvement as demonstrated in Figure 7.

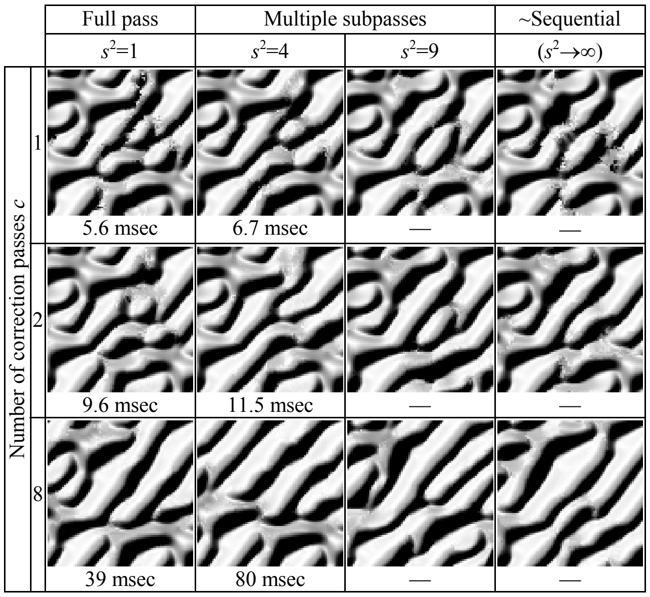


Figure 7: Effect of modifying the number of passes and subpasses. Note how {c=1, s^2=4} is nearly as good as {c=2, s^2=1}. Exemplar is from Figure 1. Execution times are for full pyramid synthesis of an 80^2 window. (s^2>4 is emulated on the CPU.)

3.4 Spatially deterministic computation

As shown in [Wei and Levoy 2003], deterministic synthesis requires that each correction pass start with a dilated domain that includes the neighborhoods of all the desired corrected pixels. For our case of synthesizing a deterministic texture window W_l at a level l, we must start with a larger window $W_l' \supset W_l$ such that the correction of pixels in W_l is influenced only by those in W_l'.

Recall that we apply c=2 correction passes, each consisting of s^2=4 subpasses. Because the correction of each pixel depends on the previous state of pixels in its 5×5 neighborhood, i.e. at most 2 pixels away, the transitive closure of all dependencies in the complete sequence of subpasses extends at most $2cs^2$=16 pixels. Therefore, it is sufficient that window W_l' have a border padding of 16 pixels on all sides of window W_l. In fact, for our specific ordering of subpasses, this border only requires 10 and 11 pixels on the left/right and top/bottom respectively.

The window W_l' is created by upsampling and jittering a smaller (half-size) window W_{l-1} at the next coarser level. The padding process is thus repeated iteratively until reaching W_0'. Figure 2 shows the resulting padded pyramid, in which W_l' and W_l are identified by the outer boundaries and the non-hazy pixels respectively. For comparison, the overlaid yellow squares show a pyramid without padding. Table 1 lists the padded window sizes required at each level to synthesize various windows.

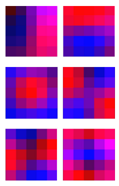
When panning through a texture image, we maintain a prefetch border of 16 pixels around the windows of all levels to reduce the number of updates, and we incrementally compute just the two strips of texture exposed at the boundaries. Timing results are presented in Section 6.

Dim. of desired window W_L=W_8	Dimension of padded window at each level								
	W_0'	W_1'	W_2'	W_3'	W_4'	W_5'	W_6'	W_7'	W_8'
256	45	46	48	52	59	74	103	161	278
1	44	44	44	44	43	42	39	34	24

Table 1: Padded window sizes required for spatial determinism.

3.5 PCA projection of pixel neighborhoods

We reduce both memory and time by projecting pixel neighborhoods into a lower-dimensional space. During a preprocess on each exemplar and at each level, we run principal component analysis (PCA) on the neighborhoods $N_{E_l}(u)$, and project them as $\tilde{N}_{E_l}(u) = P_6 N_{E_l}(u)$ where matrix P_6 contains the 6 largest principal components (see inset example). The synthesis correction step then computes $\tilde{N}_{S_l}(p) = P_6 N_{S_l}(p)$ and evaluates neighborhood distance as the 6-dimensional distance $\| \tilde{N}_{S_l}(p) - \tilde{N}_{E_l}(\cdot) \|$.



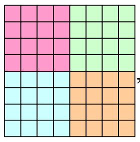
3.6 GPU implementation

We perform all three steps of texture synthesis using the GPU rasterization pipeline. The upsampling and jitter steps are simple enough that they are combined into a single rasterization pass. The most challenging step is of course the correction step.

When implementing the correction algorithm on current GPUs, we must forgo some opportunities for optimization. For instance, the array of 18 candidate neighborhoods $C_i^l\left(S_l[p+\Delta]-\Delta\right)$ often has many duplicates over the range of i and Δ, and these duplicates are usually culled in a CPU computation. While this culling is currently infeasible on a GPU, the broad parallelism and efficient texture caching still makes the brute-force approach practical, particularly if we tune the algorithm as described next.

Quadrant packing. Each correction subpass must write to a set of nonadjacent pixels, but current GPU pixel shaders do not support efficient branching on such fine granularity. Thus, in a straightforward implementation, each subpass would cost almost as much as a full pass. Instead, we reorganize pixels according to their "mod s" location. For s^2=4, we store the image as

$$\left(\begin{array}{ccc:ccc} [0,0] & [0,2] & \cdots & [0,1] & [0,3] & \cdots \\ [2,0] & [2,2] & \cdots & [2,1] & [2,3] & \cdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\ \hdashline [1,0] & [1,2] & \cdots & [1,1] & [1,3] & \cdots \\ [3,0] & [3,2] & \cdots & [3,1] & [3,3] & \cdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \end{array}\right), \text{ i.e.}$$



so that each subpass corrects a small contiguous block of pixels – a quadrant – while copying the remaining three quadrants. This quadrant reorganization complicates the texture reads somewhat, since the neighborhoods $N_{S_l}(p)$ are no longer strictly adjacent in texture space. However, the read sequence still provides excellent texture cache locality. The end result is that we can perform all four correction subpasses in nearly the same time as one full pass (Figure 7). The upsampling and jitter steps still execute efficiently as a single pass on these quadrant-packed images.

Color caching. Gathering the neighborhood $N_{S_l}(p)$ requires fetching the colors $E_l[S_l[p+\ldots]]$ of 25 pixels. We reduce each color fetch from two texture lookups to one by caching color along with the exemplar coordinate u at each pixel. Thus, each pixel $S[p]$ stores a tuple (u, $E[u]$).

PCA projection of colors. Because the coordinates u are 2D and the colors $E[u]$ are 3D, tuple (u, $E[u]$) would require 5 channels. To make it fit into 4 channels, we project colors onto their top two principal components (computed per exemplar). For most textures, neighborhood information is adequately captured in this 2D subspace. In fact, Hertzmann et al [2001] observe that just 1D luminance is often adequate.

Channel quantization. GPU parallelism demands a high ratio of computation to bandwidth. We reduce bandwidth by packing all precomputed and synthesized data into 8-bit/channel textures. We quantize the 6 coefficients of $\tilde{N}_E(u)$ using different scaling factors to fully exploit the 8-bit range. We store the k=2 candidate sets $\left(C_1^l(u), C_2^l(u)\right)$ and projected neighborhoods $\tilde{N}_{E_l}(u)$ as an RGBA and two RGB textures. Alternatively, for a 25% synthesis speedup we store them as $\left(C_1^l(u), \tilde{N}_{E_l}(C_1^l(u)),\ C_2^l(u), \tilde{N}_{E_l}(C_2^l(u))\right)$ into four RGBA textures. Note that byte-sized u_x, u_y coordinates limit the exemplar size to 256×256, but that proves sufficient.

2D Hash function. One common approach to define a hash function is to fill a 1D texture table with random numbers, and apply successive permutations through this table using the input coordinates [e.g. Perlin 2002]. We adopt a similar approach, but with a 2D 16×16 texture, and with an affine transform between permutations to further shuffle the coordinates. Note that the interaction of jitter across levels helps to hide imperfections of the hash function. Because the hash function is only evaluated once per pixel during the jitter step, it is not a bottleneck – the correction pass is by far the bottleneck.

GPU shader. The correction shader executes 384 instructions, including 62 texture reads. Because the 50×6 PCA matrix P_6 takes up 75 constant vector4 registers (50 is 5×5 pixels with 2D colors), we require shader model ps_3_0. It may be possible to approximate the PCA bases using a smaller number of unique constants, to allow compilation under ps_2_a and ps_2_b.

4. Synthesis control

Our parallel multiresolution jitter enables a variety of controls.

4.1 Multiscale randomness control

The randomness parameters r_l set the jitter amplitude at each level, and thus provide a form of "spectral variation control". We link these parameters to a set of sliders (Figure 1). As demonstrated on the accompanying video, the feedback of real-time evaluation lets the parameters be chosen easily and quickly for each exemplar. Figure 8 shows some example results.

Note that the introduction of coarse-scale jitter removes visible repetitive patterns at a large scale. In contrast, texture tiling schemes behave poorly on non-homogeneous textures (e.g. with features like mountains) since the features then have "quantized" locations that are obvious when viewed from afar.

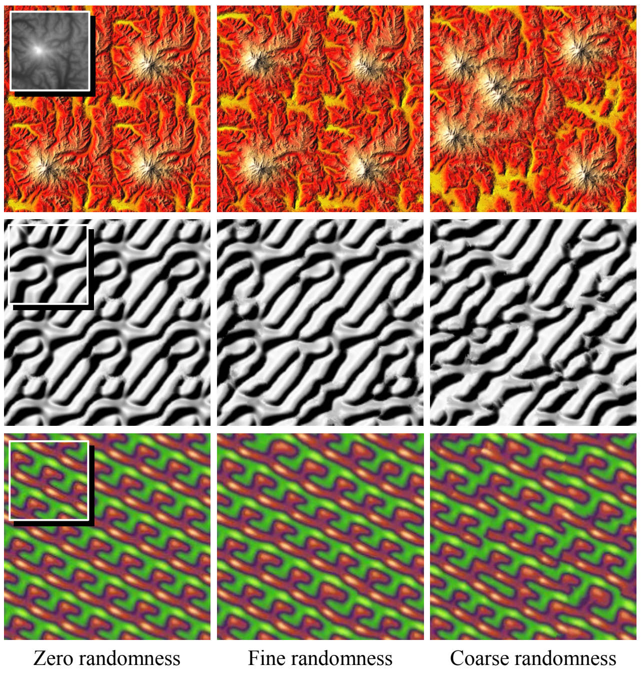


Figure 8: Examples of multiscale randomness control. For the elevation map in the top row, note how fine-scale randomness alters the mountains in place, whereas coarse-scale randomness displaces identical copies. (We shade the elevation map after synthesis for easier visualization.)

4.2 Spatial modulation over source exemplar

We can control the amount of randomness introduced in different parts of the exemplar by painting a randomness field R_E above it. From R_E we create a mipmap pyramid $R_{E_l}[u]$ where the mipmap rule is to assign each node the minimum randomness of its children. Then, jitter is modulated by the value of the randomness field R_E at the current exemplar location:

$$J_l(p) = \left\lfloor \mathcal{H}(p)\, h_l\, r_l\, R_{E_l}[S_l[p]] + \begin{pmatrix} .5 \\ .5 \end{pmatrix} \right\rfloor .$$

We find that this spatial modulation is most useful for preserving the integrity of selected texture elements in nonstationary textures, as shown in Figure 9.

The randomness field R_E serves a purpose similar to that of the binary texton mask introduced by Zhang et al [2003]. One difference is that Zhang et al first synthesize the texton mask over a surface, and use it as a prior for subsequent color synthesis. Unfortunately, the larger neighborhoods necessary for good texton mask synthesis are currently an obstacle for real-time implementation; their results require tens of minutes of CPU time.

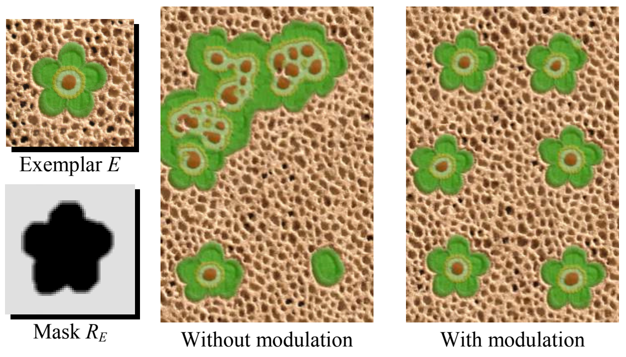


Figure 9: Synthesis using spatial modulation over exemplar.

4.3 Spatial modulation over output

The user can also paint a randomness field R_S above the output image S to spatially adjust texture variation over the synthesized range. This may be used for instance to roughen a surface pattern in areas of wear or damage. Given the painted image R_S, we let the hardware automatically create and access a mipmap pyramid $R_{S_l}[p]$. We modulate the jitter using:

$$J_l(p) = \left\lfloor \mathcal{H}(p)\, h_l\, r_l\, R_{S_l}[p] + \begin{pmatrix} .5 \\ .5 \end{pmatrix} \right\rfloor .$$

Figure 1 and Figure 10 show examples of the resulting local irregularities.

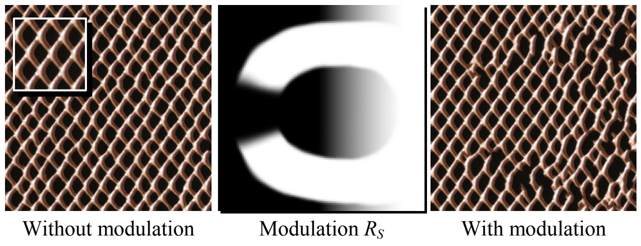


Figure 10: Synthesis using spatial modulation over output space. Left shows small uniform randomness whereas right shows effect of user-specified spatial modulation R_s.

4.4 Feature drag-and-drop

Our most exciting control is the drag-and-drop interface, which locally overrides jitter to explicitly position texture features. For example, the user can fine-tune the locations of buildings in a cityscape or relocate mountains in a terrain. Also, decals can be instanced at particular locations, such as bullet impacts on a wall.

The approach is to constrain the synthesized coordinates in a circular region of the output. Let the circle have center p_F and radius r_F, and let u_F be the desired exemplar coordinate at p_F. We then override $S_l[p] := (u_F + (p - p_F)) \bmod m$ if $\|p - p_F\| < r_F$. It is important to apply this constraint across many synthesis levels, so that the surrounding texture can best correct itself to merge seamlessly. For added control and broader adaptation at coarser levels, we actually store two radii, an inner radius r_i and an outer radius r_o, and interpolate the radius per-level as $r_F = r_i\, l / L + r_o (L - l) / L$.

The user selects the feature u_F by dragging from either the exemplar domain or the current synthesized image, and the dragged pointer then interactively updates p_F (Figure 11).

The parameters u_F, p_F, r_i, r_o are stored in the square cells associated with a coarse image I_F (at resolution l=1 in our system), similar to the tiled sprites in [Lefebvre and Neyret 2003]. Unlike texture sprites, our synthesized features merge seamlessly with the surrounding texture. In addition, we can introduce feature variations by disabling the synthesis constraint at the finest levels.

Our drag-and-drop framework is a restricted case of constrained synthesis, because the dragged feature must exist in the exemplar. On the other hand, drag-and-drop offers a number of benefits: the constraints are satisfied using multiscale coherence (resulting in nearly seamless blends with the surrounding texture); parallel synthesis allows arbitrary placement of multiple features; and, the feature positions are efficiently encoded in a sparse image.

In our current implementation, dragged features cannot lie too close together since each cell of image I_F can intersect at most one feature. However, one could allow multiple features per cell, or use a denser, sparse image representation [Kraus and Ertl 2002].

Another limitation is that the default synthesized image generally contains random distributions of all features in the exemplar. Therefore, in creating the "2005" example of Figure 11, we needed to perform a few extra drag-and-drop operations to remove mountains already present. A better approach would be to reserve areas of the exemplar for special "decal" features that should only appear when explicitly positioned.

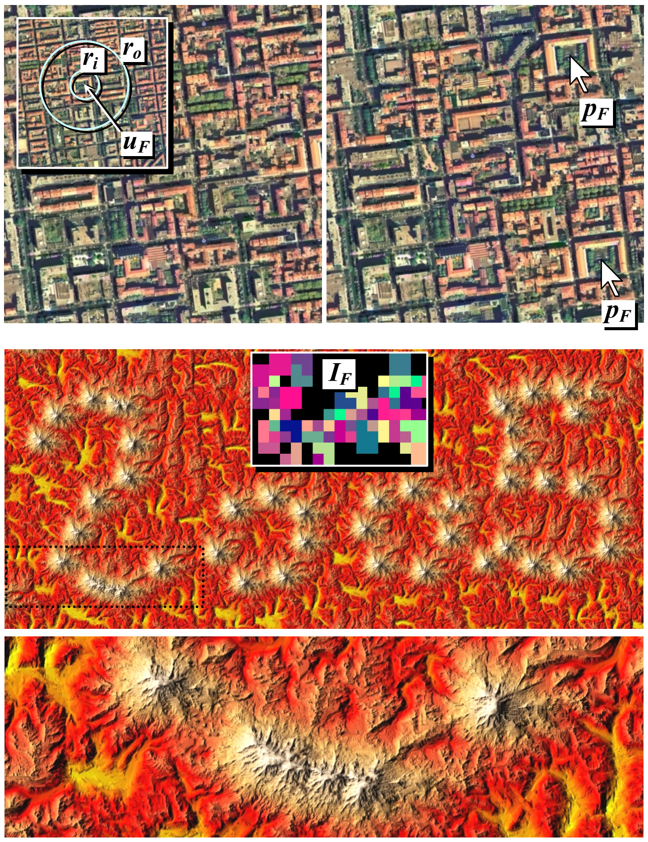


Figure 11: Examples of drag-and-drop interface and its results. The coarse 20×10 image I_F encodes the mountain positions for the 1280×560 synthesized terrain, which is then magnified (as described in Section 5) to define a 19,000×8,300 landscape.

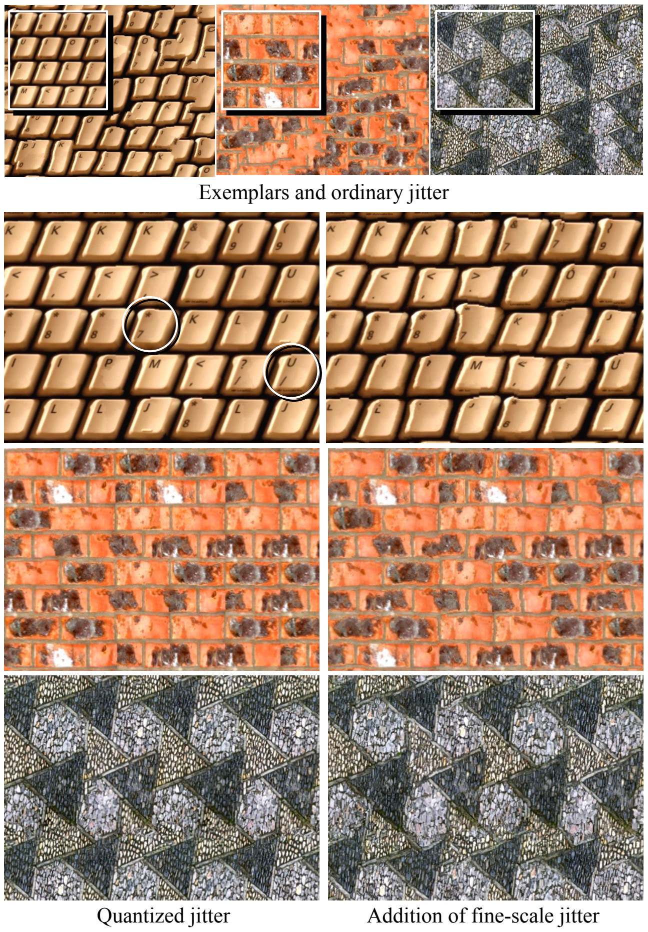


Figure 12: Results of near-regular texture synthesis. White circles indicate examples of newly formed tiles.

4.5 Near-regular textures

Some textures are near-regular [Liu et al 2004], in the sense that they deviate from periodic tilings in two ways: (1) their geometric structure may be only approximately periodic, and (2) their tiles may have irregular color. Our synthesis scheme can be adapted to create such textures as follows.

Given a near-regular texture image E′, we resample it onto an exemplar E such that its lattice structure becomes (1) regular and (2) a subdivision of the unit square domain. The first goal is achieved using the technique of [Liu et al 2004], which determines the two translation vectors representing the underlying translational lattice and warps E' into a "straightened" lattice. To satisfy the second goal, we select an $n_x \times n_y$ grid of lattice tiles bounded by a parallelogram and map it affinely to the unit square. As an example, the first exemplar in Figure 12 is created from the inset image. We can then treat the exemplar as being toroidal in our synthesis scheme.

At runtime, we maintain tiling periodicity by quantizing each jitter coordinate as

$$J'_{l,x}(p) = (m/n_x)\lfloor J_{l,x}(p)/(m/n_x) + .5 \rfloor \quad \text{if} \quad h_l \geq (m/n_x)$$

and similarly for $J_{l,y}(p)$. The quantization is disabled on the fine levels (where spacing h_l is small) to allow local geometric distortion if desired. During the analysis preprocess, we also constrain each similarity set $C^l(u)$ to the same quantized lattice (e.g. $u_x + i(m/n_x)$ for integer i) on levels for which $h_l \geq (m/n_x)$.

Some results are presented in Figure 12. The lattice parameters (n_x,n_y) are respectively (4,2), (2,2), and (2,1). (The keyboard example is not (4,4) because the rows of keys are slightly offset.) Of course, the synthesized images can easily be transformed back to their original space using inverses of the affine maps. Transforming the synthesized coordinates avoids resampling.

Our approach differs from that of Liu et al [2004; 2005] in several ways. By directly synthesizing a deformation field, they accurately model geometric irregularities found in real textures, while our fine-scale jitter process is completely random. They obtain color irregularity by iteratively selecting and stitching overlapping lattice tiles in scanline order, whereas our jitter process allows parallel synthesis. Using mid-scale jitter, we can optionally combine portions of different tiles, as evident in the creation of new keys in Figure 12. Finally, their tile selection depends on the choice of a lattice anchor point, whereas our synthesis process treats the exemplar domain in a truly toroidal fashion.

5. Synthesis magnification

Image super-resolution schemes [e.g. Freeman et al 2000; Hertzmann et al 2001] hallucinate image detail by learning from a set of low- and high-resolution color exemplars. One key difference in texture synthesis schemes like ours is that each pixel of the output S_L contains not just a color but also coordinates referring back to the source exemplar. These coordinates effectively define a 2D parametric patch structure. We next present *synthesis magnification*, a scheme for exploiting this 2D map to create higher-resolution images.

Let the exemplar E=E_L be obtained as the downsampled version of some higher-resolution exemplar E_H. The idea is to use the synthesized coordinates in S_L to create a higher-resolution image by copying the same patch structure from E_H (see Figure 13).

Using this magnification, we can synthesize the patch structure at low resolution, and later amplify it with detail. This lets us exceed the 256^2 exemplar size limit in our GPU implementation. More importantly, synthesis magnification is such a simple algorithm that it can be embedded into the final surface pixel shader, and therefore does not involve any additional memory.

The process is as follows. Given texture coordinates p, we access the 4 nearest texels in the (low-resolution) synthesized image S_L. For each of the 4 texels, we compute the exemplar coordinates that point p would have if it was contained in the same parametric patch, and we sample the high-resolution exemplar E_H at those coordinates to obtain a color. Finally, we bilinearly blend the 4 colors according to the position of p within the 4 texels.

The procedure is best summarized with HLSL code:

```
sampler SL = sampler_state { ... MagFilter=Point; };
float sizeSL, sizeEL;
float ratio = sizeSL / sizeEL;

float4 MagnifyTexture(float2 p : TEXCOORD0) : COLOR {
  float2 pfrac = frac(p*sizeSL);
  float4 colors[2][2];
  for (int i=0; i<2; i++) for (int j=0; j<2; j++) {
    // Get patch coordinates at one of the 4 nearest samples.
    float2 u = tex2D(SL, p + float2(i,j) / sizeSL);
    // Extrapolate patch coordinates to current point p.
    float2 uh = u + (pfrac - float2(i,j)) / sizeSL;
    // Fetch color from the high-resolution exemplar.
    colors[i][j] = tex2D(EH, uh, ddx(p*ratio), ddy(p*ratio));
  }
  // Bilinearly blend the 4 colors.
  return lerp(lerp(colors[0][0], colors[0][1], pfrac.y),
              lerp(colors[1][0], colors[1][1], pfrac.y),
              pfrac.x);
}
```

In the common case that p lies in the interior of a patch, the 4 computed colors are identical, and the reconstructed texture simply duplicates a cell of the high-resolution exemplar E_H. If instead p lies at the boundary between 2-4 patches, the bilinear blending nicely feathers the inter-patch seams.

Even though the neighborhood matching used to create the low-resolution image S_L cannot anticipate how well higher-resolution features will match up at patch boundaries, the magnification approach is remarkably effective.

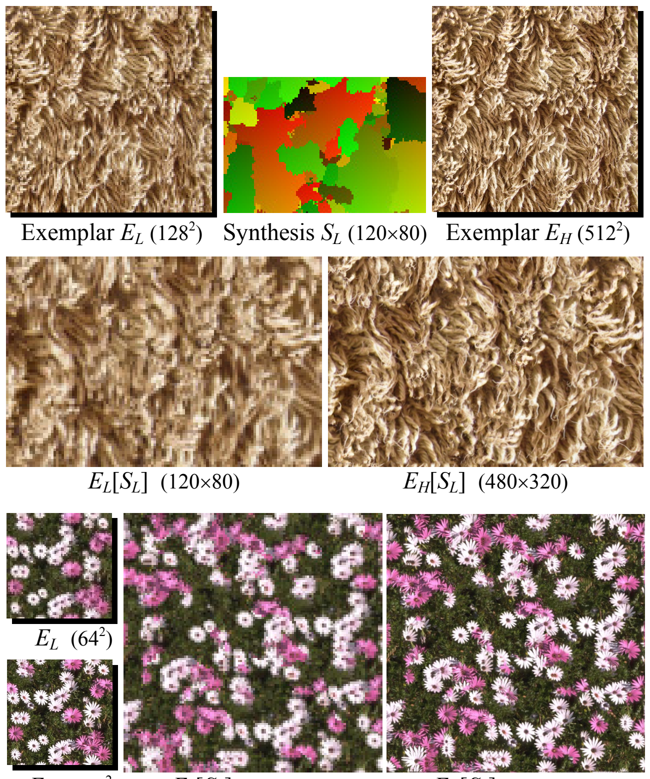


Figure 13: Two examples of synthesis magnification.

6. Additional results and discussion

All our results are obtained on an NVIDIA GeForce 6800 Ultra using Microsoft DirectX 9. CPU utilization is near zero. We use similarity sets of size k=2, and c=2 correction passes, each with s^2=4 subpasses. Exemplar sizes are 64×64 or 128×128.

Synthesis quality. As seen in Figure 14, our technique improves on the result of Wei and Levoy [2003] in terms of both synthesis quality and execution speed. Also compared is the fast but order-dependent method of Zelinka and Garland [2002].

Synthesis results on some 90 exemplars can be found on the Web at http://research.microsoft.com/projects/ParaTexSyn/. Multiscale randomness was chosen manually per exemplar, requiring about an hour of interaction in total. The examples show a default window position even though other windows may look better.

The quality of our synthesis results is generally comparable to or better than previous neighborhood-matching schemes. This is significant since previous schemes use broader neighborhoods and larger similarity sets while we use only a one-level 5×5 neighborhood and k=2. We believe that several factors make this possible: (1) the increased coherence of the coordinate upsampling step, (2) the added neighborhood-matching opportunities of the Gaussian stack, and (3) the enhancement of subpass correction.

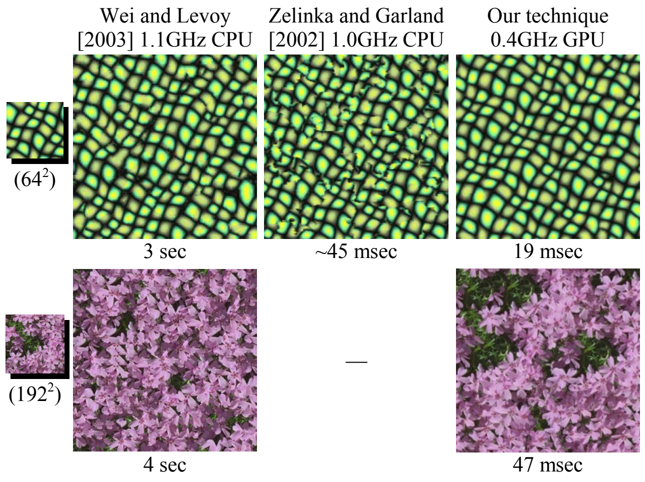

Wei and Levoy [2003] 1.1GHz CPU
Zelinka and Garland [2002] 1.0GHz CPU
Our technique 0.4GHz GPU
(64²)
3 sec
~45 msec
19 msec
(192²)
—
4 sec
47 msec

Figure 14: Comparison of synthesis quality and evaluation speed. Images are 192^2 in top row and 288^2 in bottom row.

Synthesis speed. The following table lists GPU execution times for a 64×64 exemplar, both for full recomputation of all pyramid levels and for incremental update when panning the fine-level window at a speed of 1 pixel/frame in both X and Y. The times are largely invariant to the particular texture content. An added benefit of GPU evaluation is that the output is written directly into video memory, and we thus avoid the overhead of texture upload. Synthesis magnification processes 100-200 Mpixels/sec (depending on patch coherence), so we can synthesize a 320×240 window from scratch and magnify it to 1600×1200, all at 22 frames/sec.

Window size	Average synthesis times (msec)	
	Full padded pyramid	Incremental panning
128^2	13.8	1.3
256^2	25.6	1.4
512^2	72.4	2.1

Preprocess. The per-exemplar precomputation can be summarized as follows. We iteratively filter the exemplar image E to form the Gaussian stack, and compute the PCA bases for both colors and neighborhoods. We find the similarity sets $C^l(u)$ using an approximate nearest-neighbor search on 7×7 neighborhoods [Arya et al 1998]. For each exemplar pixel, we store the PCA-projected neighborhoods of its two candidate set entries. The complete process takes about a minute on a 64×64 exemplar, and 4-12 minutes on a 128×128 exemplar.

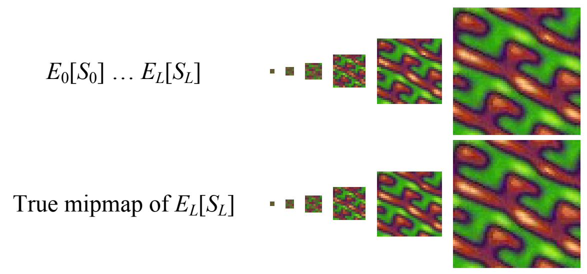
Representation compactness. Let us examine memory requirements. We omit the synthesis pyramid $S_l[p]$ since it is a temporary buffer. The per-exemplar structures require:

Structure	Bytes
Neighborhood PCA projection matrix P_6	1200
Gaussian image stack $E_l[u]$	$3m^2(\log_2 m-2)$
Projected neighborhood models $\tilde{N}_{E_l}(u)$	$6m^2(\log_2 m-2)$
Similarity sets $C^l_{1\ldots k=2}(u)$	$4m^2(\log_2 m-2)$
Total	$1200+13m^2(\log_2 m-2)$

For a 64×64 exemplar, the total is 214KB. Thus, the minimum texture size for which the synthesis-based representation is more compact than the final image is a 270×270 image. For a 128×128 exemplar, it is an 600×600 image. We believe that there are several opportunities to lower this crossover point by compressing the representation. In particular, all the structures have much less information at the coarser levels, as can be seen visually for E in Figure 5. This is an area for future investigation.

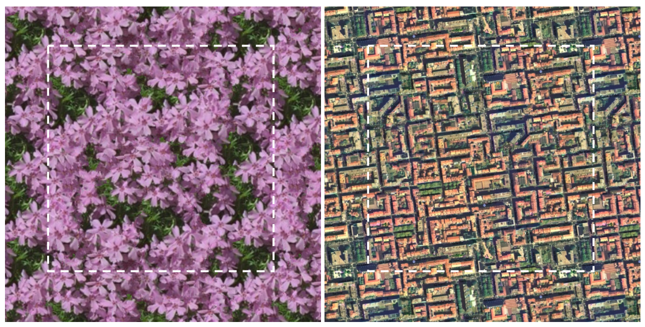
Note that for storage, one need only keep P_6, E_L, and $C^l(u)$ since the remaining structures can be rebuilt efficiently.

Mipmapping. To properly filter the synthesized texture $E_L[S_L]$ during rendering, one usually computes a mipmap pyramid through successive downsampling. Because we are generating the texture at runtime, an alternative becomes relevant – that of using the intermediate-resolution synthesized images. Of course, these are only approximations since they ignore subsequent jitter and correction, but we find that they are often adequate.



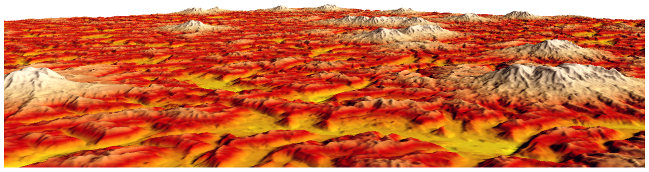
When using synthesis magnification, we access a mipmap of the high-resolution exemplar E_H. As shown in the code of Section 5, texture derivatives must be specified explicitly (`ddx`,`ddy`) to avoid erroneous (overly blurred) mipmap levels at patch boundaries.

Toroidal synthesis. If we want to create a toroidal image, we disable both the pyramid padding and the quadrant packing, and let the synthesized neighborhoods N_{S_l} be evaluated toroidally. Alternatively, to retain the efficiency of quadrant packing, we perform synthesis using a padded pyramid as before, but redefine the jitter function to be periodic using $J'(p) = J(p \bmod n)$ where n is the size of the synthesized image. Unlike in sequential per-pixel algorithms, there are no special cases at the image boundaries. Here are two example results:



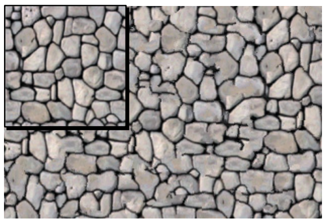
On-demand synthesis. Wei and Levoy [2003] synthesize texture on-the-fly as needed for each screen pixel. In some sense this is an ideal solution, but the resulting irregular processing does not map efficiently onto current GPUs. Our approach of synthesizing windows in texture space permits on-demand synthesis at coarser granularities such as per-tile, per-chart, or per-model. For example, several papers [e.g. Goss and Yuasa 1998; Tanner et al 1998] describe tile-based schemes that adaptively load texture as a function of changing view parameters, and these schemes could be adapted for texture synthesis. More broadly, on-demand texture synthesis has parallels with the treatment of geometric level-of-detail, which is also adjusted in model space as opposed to per-pixel. Our real-time framework will hopefully spur new research into the scheduling of on-demand data synthesis.

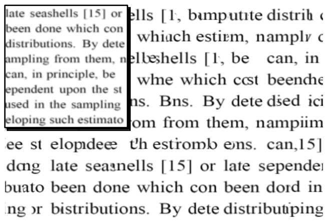
Terrain synthesis. Here is an example of terrain geometry created from the elevation map exemplar of Figure 8:



Limitations. We have referred to an "infinite" synthesized canvas. There are of course numerical limits to its extent. Fortunately, GPUs now support 32-bit floats, so the 2D integer lattice of samples is well defined up to 2^{24}=16M samples on each axis (and synthesis magnification extends this limit further).

The main weakness of our approach is the well known drawback of neighborhood-based per-pixel synthesis: it performs poorly on textures with semantic structures not captured by small neighborhoods. Here are two examples; for the text we use quantized jitter to maintain line structure, but still get muddled characters.





7. Summary and future work

We have presented a parallel synthesis algorithm for creating infinite texture without the limited variety of tiles. It is implemented as a sequence of pixel shading passes on a GPU, and can synthesize a 256^2 window of deterministic texture in 26 msec, or pan the window at over 700 frames/sec. Using a new synthesis magnification technique, we can amplify this content to fill a 1600×1200 screen in real time.

Based on multiresolution jitter, our synthesis scheme is designed to reproduce tilings by default, so that one can control the scale and amplitude of statistical distortions applied to the texture. We have explored a variety of controls that adapt the jitter both spectrally and spatially. In this setting, parallel synthesis offers the nice property of local causality – local modifications have finite spatial effect.

There are a number of avenues for future work:

- Compress the Gaussian stack representation for improved data amplification.
- Support composite textures (i.e. texture-by-numbers) [Hertzmann et al 2001; Zalesny et al 2005].
- Combine with geometry clipmaps for efficient terrain synthesis and rendering [Losasso and Hoppe 2004].
- Incorporate synthesis of vector shapes such as polygons, fonts, roads, and coastlines.
- Automatically determine the best exemplar image and randomness parameters to visually approximate a given texture.

References

ARYA, S., MOUNT, D., NETANYAHU, N., SILVERMAN, R., AND WU, A. 1998. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM* 45(6), 891-923.

ASHIKHMIN, M. 2001. Synthesizing natural textures. *Symposium on Interactive 3D Graphics*, 217-226.

BAR-JOSEPH, Z., EL-YANIV, R., LISCHINSKI, D., AND WERMAN, M. 2001. Texture mixing and texture movie synthesis using statistical learning. *IEEE TVCG* 7(2), 120-135.

COHEN, M., SHADE, J., HILLER, S., AND DEUSSEN, O. 2003. Wang tiles for image and texture generation. *ACM SIGGRAPH*, 287-294.

DE BONET, J. 1997. Multiresolution sampling procedure for analysis and synthesis of texture images. *ACM SIGGRAPH*, 361-368.

EFROS, A., AND FREEMAN, W. 2001. Image quilting for texture synthesis and transfer. *ACM SIGGRAPH*, 341-346.

EFROS, A., AND LEUNG, T. 1999. Texture synthesis by non-parametric sampling. *ICCV*, 1033-1038.

FREEMAN, W., PASZTOR, E., AND CARMICHAEL, O. 2000. Learning low-level vision. *IJCV* 40(1), 25-47.

GARBER, D. 1981. Computational models for texture analysis and texture synthesis. PhD Dissertation, University of Southern California.

GOSS, M., AND YUASA, K. 1998. Texture tile visibility determination for dynamic texture loading. *Graphics Hardware*, 55-60.

HERTZMANN, A., JACOBS, C., OLIVER, N., CURLESS, B., AND SALESIN, D. 2001. Image analogies. *ACM SIGGRAPH*, 327-340.

KRAUS, M., AND ERTL, T. 2002. Adaptive texture maps. *Graphics Hardware*, 7-15.

KWATRA, V., SCHÖDL, A., ESSA, I., TURK, G., AND BOBICK, A. 2003. Graphcut textures: image and video synthesis using graph cuts. *ACM SIGGRAPH*, 277-286.

LEFEBVRE, S., AND NEYRET, F. 2003. Pattern based procedural textures. *Symposium on Interactive 3D Graphics*, 203-212.

LIANG, L., LIU, C., XU, Y., GUO, B., AND SHUM, H.-Y. 2001. Real-time texture synthesis by patch-based sampling. *ACM TOG* 20(3), 127-150.

LIU, Y., LIN, W.-C., AND HAYS, J. 2004. Near-regular texture analysis and manipulation. *ACM SIGGRAPH*, 368-376.

LIU, Y., TSIN, Y., AND LIN, W.-C. 2005. The promise and perils of near-regular texture. *IJCV* 62(1-2), 149-159.

LOSASSO, F., AND HOPPE, H. 2004. Geometry clipmaps: terrain rendering using nested regular grids. *ACM SIGGRAPH*, 769-776.

PERLIN, K. 2002. Improving noise. *ACM SIGGRAPH*, 681-682.

POPAT, K., AND PICARD, R. 1993. Novel cluster-based probability model for texture synthesis, classification, and compression. *Visual Communications and Image Processing*, 756-768.

PRAUN, E., FINKELSTEIN, A., AND HOPPE, H. 2000. Lapped textures. *ACM SIGGRAPH*, 465-470.

TANNER, C., MIGDAL, C., AND JONES, M. 1998. The clipmap: A virtual mipmap. *ACM SIGGRAPH*, 151-158.

TONG, X., ZHANG, J., LIU, L., WANG, X., GUO, B., AND SHUM, H.-Y.. 2002. Synthesis of bidirectional texture functions on arbitrary surfaces. *ACM SIGGRAPH*, 665-672.

TONIETTO, L., AND WALTER, M. 2002. Towards local control for image-based texture synthesis. In *Proceedings of SIBGRAPI 2002 – XV Brazilian Symposium on Computer Graphics and Image Processing*.

WEI, L.-Y., AND LEVOY, M. 2000. Fast texture synthesis using tree-structured vector quantization. *ACM SIGGRAPH*, 479-488.

WEI, L.-Y., AND LEVOY, M. 2003. Order-independent texture synthesis. http://graphics.stanford.edu/papers/texture-synthesis-sig03/. (Earlier version is Stanford University Computer Science TR-2002-01.)

WEI, L.-Y. 2004. Tile-based texture mapping on graphics hardware. *Graphics Hardware*, 55-64.

ZALESNY, A., AND VAN GOOL, L. 2001. A compact model for viewpoint dependent texture synthesis. In *SMILE 2000: Workshop on 3D Structure from Images*, 124-143.

ZALESNY, A., FERRARI, V., CAENEN, G., AND VAN GOOL, L. 2005. Composite texture synthesis. *IJCV* 62(1-2), 161-176.

ZELINKA, S., AND GARLAND, M. 2002. Towards real-time texture synthesis with the jump map. *Eurographics Workshop on Rendering*.

ZHANG, J., ZHOU, K., VELHO, L., GUO, B., AND SHUM, H.-Y. 2003. Synthesis of progressively-variant textures on arbitrary surfaces. *ACM SIGGRAPH*, 295-302.