State of the Art in Example-based Texture Synthesis

Li-Yi Wei[1] Sylvain Lefebvre[2] Vivek Kwatra[3] Greg Turk[4]

[1]Microsoft Incubation [2]REVES / INRIA Sophia-Antipolis [3]Google Research [4]Georgia Institute of Technology

Abstract

Recent years have witnessed significant progress in example-based texture synthesis algorithms. Given an example texture, these methods produce a larger texture that is tailored to the user's needs. In this state-of-the-art report, we aim to achieve three goals: (1) provide a tutorial that is easy to follow for readers who are not already familiar with the subject, (2) make a comprehensive survey and comparisons of different methods, and (3) sketch a vision for future work that can help motivate and guide readers that are interested in texture synthesis research. We cover fundamental algorithms as well as extensions and applications of texture synthesis.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture: Graphics Processors; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism: Texture;

Keywords: texture synthesis, texture mapping, pixel, patch, optimization, surface, video, flow, fluids, parallel computing, real-time rendering, solid, globally varying, inverse synthesis, super-resolution, geometry

1. Introduction

Texturing is a core process for computer graphics applications. The texturing process can be divided into three components: (1) texture acquisition, (2) texture mapping, and (3) texture rendering, which includes a variety of issues such as access, sampling, and filtering. Although a source texture can be acquired by a variety of methods such as manual drawing or photography, example-based texture synthesis [KWLT07] remains one of the most powerful methods as it works on a large variety of textures, is easy to use – the user only needs to supply an exemplar – and provides high output quality. The result is an arbitrarily large output texture that is visually similar to the exemplar and does not contain any unnatural artifacts or repetition. In addition to making texture creation easier, example-based texture synthesis also provides benefits to other parts of the rendering pipeline. Distortion free textures are automatically generated over complex geometries, and on-the-fly generation of texture content strongly reduces storage requirements.

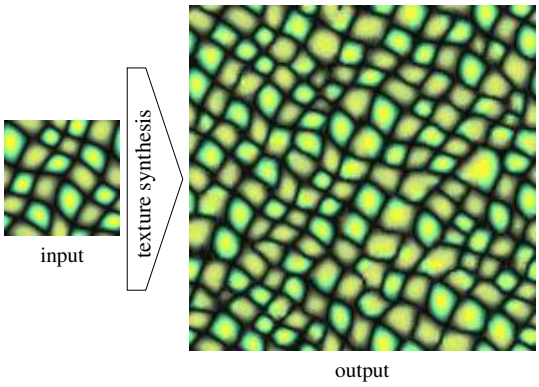


Figure 1: Texture synthesis. Given a sample texture, the goal is to synthesize a new texture that looks like the input. The synthesized texture is tileable and can be of arbitrary size specified by the user.

In this paper, we provide a state-of-the-art report for example-based texture synthesis. In addition to giving a comprehensive coverage of current methods, this report can also act as a tutorial that is easy for a novice to follow. This report also provides a vision for experienced readers who are interested in pursuing further research. To achieve these goals, we organize the paper as follows. We start with the fundamental concepts (Section 2) and algorithms (Section 3), followed by various extensions and applications (Sections 4 through 13). Although our main focus is on texture synthesis techniques that provide high enough quality for graphics applications, we will also briefly mention

alternative methods that may provide additional theoretical and practical insights (Section 14). We conclude by summarizing the main differences between the two major categories of texture synthesis methods: procedural [EMP*02] and example-based, and from this we present a roadmap for potential future work in example-based texture synthesis.

2. Preliminary

Before describing methods for texture synthesis, let us begin by answering some preliminary questions, such as: what are textures, and what is texture synthesis.

2.1. What are textures?

Texture mapping is a standard technique to represent surface details without explicit modeling for geometry or material properties. The mapped image, usually rectangular, is called a *texture map* or *texture*. A texture can be used to modulate various surface properties, including color, reflection, transparency, or displacements. Due to this generality, the term *texture* in computer graphics can refer to an image containing arbitrary patterns.

Unfortunately, the meaning of texture in graphics is somewhat abused from its usual meaning. In other contexts, such as ordinary English as well as specific research fields including computer vision and image processing, textures are usually referred to as visual or tactile surfaces composed of repeating patterns, such as a fabric. This definition of texture is more restricted than the notion of texture in graphics. However, since a majority of natural surfaces consist of repeating elements, this narrower definition of texture is still powerful enough to describe many surface properties.

In this paper, we concentrate on the narrower definition of textures, i.e. images containing repeating patterns. Since natural textures may contain interesting variations or imperfections, we also allow a certain amount of randomness over the repeating patterns. The amount of randomness can vary for different textures, from stochastic (sand beach) to deterministic (tiled floor). See [LLH04] for a classification.

2.2. What is texture synthesis?

Textures can be obtained from a variety of sources such as hand-drawn pictures or scanned photographs. Hand-drawn pictures can be aesthetically pleasing, but it is hard to make them photo-realistic. Moreover not everybody is an artist and it could be difficult for ordinary people to come up with good texture images – there is a specific profession in game and film studios named *texture artist*, whose job is to produce high quality textures. Scanned images, however, may be of inadequate size or quality, e.g. containing non-uniform lighting, shadows, or geometry and could lead to visible seams or repetition if directly used for texture mapping.

Texture synthesis is an alternative way to create textures. It is general and easy to use, as the user only needs to supply an input exemplar and a small set of synthesis parameters. The output can be made to be of any size without unnatural repetition. Texture synthesis can also produce tileable images by properly handling the boundary conditions.

The goal of texture synthesis can be stated as follows: Given a texture sample, synthesize a new texture that, when perceived by a human observer, appears to be generated by the same underlying process (Figure 1). The process of texture synthesis could be decomposed into two main components, analysis and synthesis:

Analysis How to estimate the underlying generation process from a given finite texture sample. The estimated process should be able to model both the structural and stochastic parts of the input texture. The success of the model is determined by the visual fidelity of the synthesized textures with respect to the given samples.

Synthesis How to develop an efficient generation procedure to produce new textures from a given analysis model. The efficiency of the sampling procedure will directly determine the computational cost of texture generation.

Essentially, texture synthesis algorithms differ in their specifics regarding the models used in the analysis part as well as the computations used in the synthesis part.

2.3. Markov Random Field

Although a variety of texture models have been proposed throughout the history, so far the most successful model for graphics applications is based on Markov Random Field (MRF). Thus, in the rest of this paper we will focus mainly on MRF-based algorithms, and briefly discuss alternative methods in Section 14.

Markov Random Field methods model a texture as a realization of a *local* and *stationary* random process. That is, each pixel of a texture image is characterized by a small set of spatially neighboring pixels, and this characterization is the same for all pixels. The intuition behind this model can be demonstrated by the following thought experiment. Imagine that a viewer is given an image, but only allowed to observe it through a small movable window. As the window is moved the viewer can observe different parts of the image. The image is stationary if, under a proper window size, the observable portion always appears similar. The image is local if each pixel is predictable from a small set of neighboring pixels and is independent of the rest of the image.

Based on this Markov-Random-Field model, the goal of texture synthesis can be formulated as follows: given an input texture, synthesize an output texture so that for each output pixel, its spatial neighborhood is similar to at least one neighborhood at the input. The size of the neighborhood is a user-specifiable parameter and should be proportional to

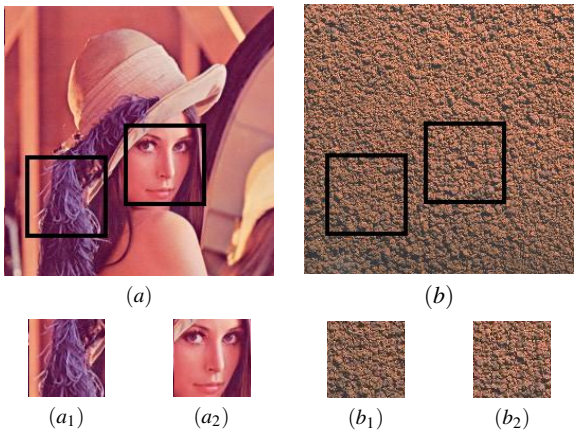


Figure 2: How textures differ from images. (a) is a general image while (b) is a texture. A movable window with two different positions are drawn as black squares in (a) and (b), with the corresponding contents shown below. Different regions of a texture are always perceived to be similar (b_1, b_2), which is not the case for a general image (a_1, a_2). In addition, each pixel in (b) is only related to a small set of neighboring pixels. These two characteristics are called stationarity and locality, respectively.

the feature size of the texture patterns. Due to the MRF assumption, similarity of local neighborhoods between input and output will guarantee their perceptual similarity as well. In addition to this quality concern, the texture synthesis algorithm should also be efficient and controllable. In the rest of this paper we will present several recent texture synthesis algorithms that are based on this MRF model.

3. Basic Algorithms

As mentioned in Section 2.3, a significant portion of recent texture synthesis algorithms for graphics applications are based on Markov Random Fields (MRF). MRF is a rigorous mathematical tool, and has indeed be utilized as such in some earlier texture analysis and synthesis algorithms (e.g. see [Pop97, Pag04]). However, even though a rigorous approach has many merits, it could exhibit certain disadvantages such as heavy computation or complex algorithms, making such methods difficult to use and understand.

3.1. Pixel-based synthesis

One of the first methods that break through this barrier and thus blossomed the study of texture synthesis algorithms is the work by [EL99]. The basic idea of [EL99] is very simple. As illustrated in Figure 3, given an input exemplar, the output is first initialized by copying a small seed region from the input. The synthesized region is then gradually grown from the initial seed by assigning the output pixels one by one in

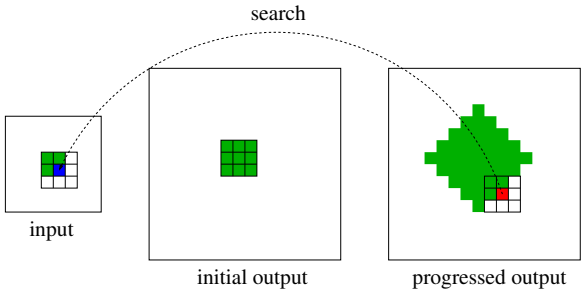


Figure 3: The algorithm by [EL99]. Given an input exemplar (left), the output is initialized by copying a small initial seed from the input (middle). The synthesis then progresses by assigning output pixels one by one via neighborhood search (right).

an inside-out, onion layer fashion. Each output pixel is determined by a neighborhood search process. For example, to determine the value of the red pixel in Figure 3, [EL99] places a neighborhood with user-determined size (3×3 for this toy example) around it and collects the set of already synthesized pixels (shown as green in Figure 3). The method then finds the candidate set of good matches from the input with respect to this partial neighborhood composed of these already synthesized pixels, and assigns the output pixel (red) as the center of a randomly selected neighborhood from the candidate set (blue). This process is repeated for every output pixel by growing from the initial region until all the output pixels are assigned.

The algorithm by [EL99] is very easy to understand and implement and works well for a variety of textures. It is also user friendly, as the only user specifiable parameter is the neighborhood size. Intuitively, the neighborhood size should be roughly equal to the texture element sizes. If the neighborhood is too small, the output may be too random. On the other hand if the neighborhood is too big, the output may reduce to a regular pattern or contain garbage regions as the synthesis process would be over-constrained. Note that even though the basic idea in [EL99] is inspired by MRF, the synthesis process does not really perform a rigorous MRF sampling. Thus, it is much easier to understand and implement, and potentially runs faster.

Despite its elegance and simplicity, [EL99] could be slow and subject to non-uniform pattern distribution due to the use of variable-occupancy neighborhoods and the inside-out synthesis. To address these issues, [WL00] proposed a simple algorithm based on fixed neighborhood search.

The basic idea is illustrated in Figure 4. Similar to [Pop97, EL99], the output is constructed by synthesizing the pixels one by one via a neighborhood match process. However, unlike [EL99] where the neighborhood may contain a varying number of valid pixels, [WL00] always uses a fixed neigh-

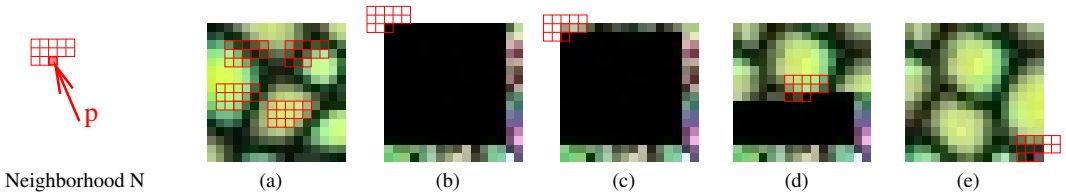


Figure 4: The algorithm by [WL00]. (a) is the input texture and (b)-(e) show different synthesis stages of the output image. Pixels in the output image are assigned in a raster scan ordering. The value of each output pixel p is determined by comparing its spatial neighborhood $N(p)$ with all neighborhoods in the input texture. The input pixel with the most similar neighborhood will be assigned to the corresponding output pixel. Neighborhoods crossing the output image boundaries (shown in (b), (c) and (e)) are handled toroidally. Although the output image starts as a random noise, only the last few rows and columns of the noise are actually used. For clarity, we present the unused noise pixels as black. (b) synthesizing the first pixel, (c) synthesizing the first pixel of the second row, (d) synthesizing the middle pixel, (e) synthesizing the last pixel.

borhood. Also, the output is synthesized in a pre-determined sequence such as a scanline order instead of the inside-out fashion as in [EL99]. The algorithm begins by initializing the output as a noise (i.e. randomly copying pixels from the input to the output). To determine the value for the first pixel at the upper-left corner of the output, [WL00] simply finds the best match for its neighborhood. However, since this is the first pixel, its neighborhood will contain only noise pixels and thus it is essentially randomly copied from the input. (For neighborhood pixels outside the output image, a toroidal boundary condition is used to wrap them around.) However, as the synthesis progresses, eventually the output neighborhood will contain only valid pixels and the noise values only show up in neighborhoods for pixels in the first few rows or columns.

One primary advantage of using a fixed neighborhood is that the search process can be easily accelerated by various methods, such as tree-structured vector quantization (TSVQ), kd-tree, or k-coherence; we will talk more about acceleration in Section 3.2. Another advantage is that a fixed neighborhood could be extended for a variety of synthesis orders; in addition to the scanline order as illustrated in Figure 4, other synthesis orders are also possible such as multi-resolution synthesis, which facilitates the use of smaller neighborhoods to capture larger texture elements/patterns, or order-independent synthesis, which allows parallel computation and random access as discussed in Section 7.

3.2. Acceleration

The basic neighborhood search algorithms [EL99, WL00] have issues both in quality and speed. Quality-wise, local neighborhood search would often result in noisy results [WL00] or garbage regions [EL99]. Speed-wise, exhaustive search can be computationally slow.

Throughout the years many solutions have been proposed to solve these quality and speed issues (e.g. tree search [WL00,KEBK05]), but so far the most effective methods are

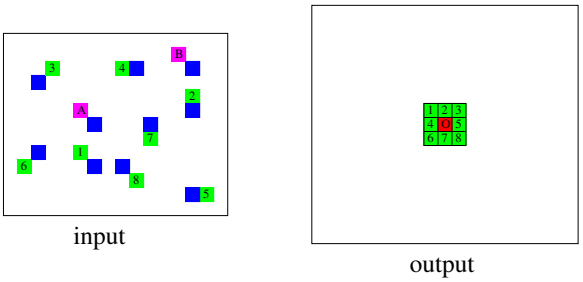


Figure 5: Illustration of K-coherence algorithm. The final value of output pixel O is chosen from its candidate set, shown as blue pixels on the input. See main text for details.

based on the notion of *coherence*. One of the first papers that explored coherence was [Ash01] and the basic idea is pretty simple. When pixels are copied from input to output during the synthesis process, it is very unlikely that they will land on random output locations. Instead, pixels that are together in the input ought to have a tendency to be also together in the output. Similar ideas have also appeared in other methods such as jump maps [ZG04] and k-coherence [TZL*02]. In our experience, k-coherence is one of the best algorithms in terms of quality and speed, so we focus on it here.

The k-coherence algorithm is divided into two phases: analysis and synthesis. During analysis, the algorithm builds a similarity-set for each input texel, where the similarity-set contains a list of other texels with similar neighborhoods to the specific input texel. During synthesis, the algorithm copies pixel from the input to the output, but in addition to colors, we also copy the source pixel location. To synthesize a particular output pixel, the algorithm builds a candidate-set by taking the union of all similarity-sets of the neighborhood texels for each output texel, and then searches through this candidate-set to find out the best match. The size of the similarity-set, K, is a user-controllable parameter (usually in the range $[2\ 11]$) that determines the overall speed/quality.

Perhaps the best way to explain this K-coherence algorithm is by an example, as illustrated in Figure 5. During analysis, we compute a similarity set for each input pixel. For example, since pixels A and B are the two most similar neighborhoods to pixel 1, they constitute pixel 1's similarity set in addition to pixel 1 itself. During synthesis, the pixels are copied from input to output, including both color and location information. To synthesize output pixel O, we look at its 8 spatial neighbors that are already synthesized, shown in green pixels with numerical marks 1 to 8. We build pixel O's candidate set from the similarity sets of the 8 spatial neighbors. For example, since pixel 1 is one pixel up and one pixel left with respect to pixel O, it contributes the 3 blue pixels with complementary shifting from pixel 1's similarity set, including pixel 1 itself and pixels A and B. Similar process can be conducted for pixels 2 to 8. In the end, the candidate set of pixel O will include all blue pixels shown in the input. (For clarity, we only show pixel 1's similarity set; pixels 2 to 8 will have similar pixels in their similarity sets just like pixels A and B with respect to pixel 1.) From this candidate set, we then search the input pixel P that has most similar neighborhood to the output pixel O, and copy over P's color and location information to pixel O. This process is repeated for every output pixel in every pyramid level (if a multi-resolution algorithm is used) until the entire output is synthesized.

3.3. Patch-based synthesis

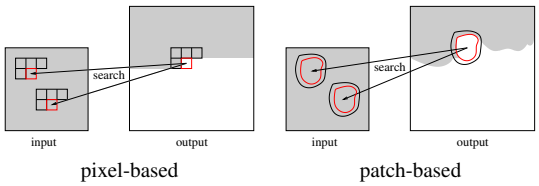


Figure 6: Comparisons of pixel-based and patch-based texture synthesis algorithms. The gray region in the output indicates already synthesized portion.

The quality and speed of pixel-based approaches can be improved by synthesizing patches rather than pixels. Intuitively, when the output is synthesized by assembling patches rather than pixels from the input, the quality ought to improve as pixels within the same copied patch ought to look good with respect to each other. Figure 6 illustrates the basic idea of patch-based texture synthesis. In some sense, patch-based synthesis is an extension from pixel-based synthesis, in that the units for copying are patches instead of pixels. Specifically, in pixel-based synthesis, the output is synthesized by copying pixels one by one from the input. The value of each output pixel is determined by neighborhood search to ensure that it is consistent with already synthesized pixels. Patch-based synthesis is very similar to pixel-based synthesis, except that instead of copying pixels, we copy patches. As illustrated in Figure 6, to ensure output quality, patches are selected according to its neighborhood, which, just like in pixel-based synthesis, is a thin band of pixels around the unit being copied (being pixel in pixel-based synthesis or patch in patch-based synthesis).

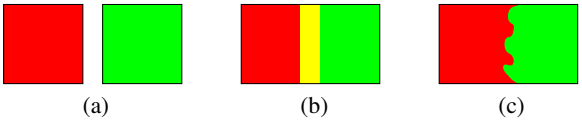


Figure 7: Methods for handling adjancent patches during synthesis. (a) two patches shown in different colors. (b) the overlapped region is simply blended from the two patches. (c) an optimal path is computed from the overlapped region.

The major difference between pixel-based and patch-based algorithm lies in how the synthesis unit is copied onto the output. In pixel-based algorithm, the copy is just a copy. However, in patch-based algorithms, the issue is more complicated as a patch, being larger than a pixel, usually overlaps with the already synthesized portions, so some decision has to be made about how to handle the conflicting regions. In [PFH00], new patches simply overwrite over existing regions. By using patches with irregular shapes, this approach took advantage of the texture masking effects of human visual system and works surprisingly well for stochastic textures. [LLX*01] took a different approach by blending the overlapped regions (Figure 7 b). As expected, this can cause blurry artifacts in some situations. Instead of blending, [EF01] uses dynamic programming to find an optimal path to cut through the overlapped regions, and this idea is further improved by [KSE*03] via graph cut (Figure 7 c). Finally, another possibility is to warp the patches to ensure pattern continuity across patch boundaries [SCA02, WY04].

Another approach inspired by patch-based synthesis is to prepare sets of square patches with compatible edge constraints. By tiling them in the plane, different textures can be obtained. This approach is further detailed in Section 7.2.

3.4. Texture optimization

[KEBK05] proposed texture optimization as an alternative method beyond pixel-based and patch-based algorithms. The algorithm is interesting in that it combines the properties of both pixel and patch based algorithms. Similar to pixel-based algorithms, texture optimization synthesizes an output texture in the units of pixels (instead of patches). But unlike previous pixel-based methods which synthesize pixels one by one in a greedy fashion, this technique considers them all together, and determine their values by optimizing a quadratic energy function. The energy function is determined by mismatches of input/output neighborhoods, so minimizing this function leads to better output quality.

Specifically, the energy function can be expressed as follows:

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

where E measures local neighborhood similarity across the current active subset $X^\dagger$ of the output, and $\mathbf{z}_p$ indicates the most similar input neighborhood to each output neighborhood $\mathbf{x}_p$.

[KEBK05] solves the energy function via an EM-like algorithm; in the E (expectation) step, the set of matching input neighborhoods $\{\mathbf{z}_p\}$ remains fixed and the set of output pixels $\mathbf{x}$ is solved via a least-squares method; in the M (maximization) step, the set of output pixels $\mathbf{x}$ remains fixed and the set of matching input neighborhoods $\{\mathbf{z}_p\}$ is found by tree search. These two steps are iterated multiple times until convergence, or a maximum number of iterations is reached. Please refer to Table 1 for math details of these two steps.

This energy minimization framework blends the flavor of both pixel and patch based algorithms; while the neighborhood metric is pixel-centric, the global optimization considers multiple pixels together, bearing resemblance to patch-based algorithms.

Least Squares Solver [KEBK05]

$\mathbf{z}_p^0 \leftarrow$ random neighborhood in $Z \; \forall$ p $\in X^\dagger$
for iteration n = 0:N **do**
  $\mathbf{x}^{n+1} \leftarrow \text{argmin}_{\mathbf{x}} \; E(\mathbf{x};\{\mathbf{z}_p^n\})$ *// E-step via least squares*
  $\mathbf{z}_p^{n+1} \leftarrow \text{argmin}_{\mathbf{z}} \; |\mathbf{x}_p - \mathbf{z}|^2$ *// M-step via tree search*
  if $\mathbf{z}_p^{n+1} == \mathbf{z}_p^n \; \forall$ p $\in X^\dagger$
    x = $\mathbf{x}^{n+1}$
    break
  end if
end for

Discrete Solver [HZW*06]

$\mathbf{z}_p^0 \leftarrow$ random neighborhood in $Z \; \forall$ p $\in X^\dagger$
for iteration n = 0:N **do**
  // E-step via k-coherence
  $\mathbf{x}^{n+1} \leftarrow \text{argmin}_{\mathbf{x},\mathbf{x}(p)\in\mathbf{k}(p)\forall p} \; E(\mathbf{x};\{\mathbf{z}_p^n\})$
  // M-step via k-coherence search
  $\mathbf{z}_p^{n+1} \leftarrow \text{argmin}_{\mathbf{z}} \; |\mathbf{x}_p - \mathbf{z}|^2$
  if $\mathbf{z}_p^{n+1} == \mathbf{z}_p^n \; \forall$ p $\in X^\dagger$
    x = $\mathbf{x}^{n+1}$
    break
  end if
end for

Table 1: Pseudocode for optimization-based texture synthesis. The top portion is the least squares solver from [KEBK05], while the bottom portion is the discrete solver from [HZW*06]. The major differences include (1) the restriction of each output pixel color **x**(p) to its k-coherence candidate set **k**(p) in the E-step, and (2) the use of k-coherence as the search algorithm in the M-step.

Discrete solver The solver presented in [KEBK05] has two issues. First, it utilized hierarchical tree search for the M-step. Since tree search has an average time complexity of $O(log(N))$ where N is the total number of input neighborhoods, this step can become the bottleneck of the solver (as reported in [KEBK05]). Second, the least squares solver in the E-step may cause blur. Intuitively, the least squares solver is equivalent to performing an average of overlapping input neighborhoods on the output. This blur issue can be ameliorated by a properly spaced subset $X^\dagger$ (a heuristic provided by [KEBK05] is to allow adjacent neighborhoods in $X^\dagger$ to have $\frac{1}{4}$ overlap of the neighborhood size), but cannot be completely eliminated.

[HZW*06] addressed these issues by incorporating k-coherence into both the E- and M-steps of the original EM solver in [KEBK05]. In the E-step, instead of least squares, the candidate values of each output pixel is limited to its k-coherence candidate set and the final value is chosen as the one that best minimizes the energy function. Similar to k-coherence, both pixel color and location information are copied from the input to the output. In the M-step, instead of hierarchical tree search, we again use the k-coherence candidate set for each output pixel to find its best matched input neighborhood. Please refer to Table 1 for details. Injecting k-coherence into both the E- and M-steps addresses both the blur (E-step) and speed (M-step) issues in the original EM solver. In particular, the blur is eliminated because now pixels are copied directly instead of averaged. The speed is improved because k-coherence provides a constant instead of logarithmic time complexity. Due to the discrete nature of k-coherence search (i.e. only a small discrete number of options are available in both the E- and M-steps instead of many more possibilities in the original least squares solver), this algorithm is dubbed discrete solver. A comparison of image quality is provided in [HZW*06].

4. Surface Texture Synthesis

Producing a new texture from an example is not sufficient for many applications. Often the goal is to place a texture onto a particular curved surface. There are two logical ways to place example-based texture synthesis onto a given surface. One way is to create a flat texture and then attempt to wrap that texture onto the surface. Unfortunately, this approach has the usual texture mapping problems of distortion and texture seams. The other approach is to synthesize the texture in a manner that is tailored for the particular surface in question, and this is the approach that we will explore in this section.

The task of performing texture synthesis on a surface can be divided into two sub-task: 1) create an orientation field over the surface, and 2) perform texture synthesis according to the orientation field. We will examine these two sub-tasks one at a time.

The purpose of the orientation field is to specify the direction

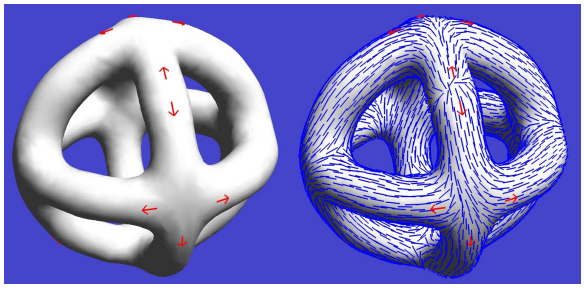


Figure 8: Vector constraints from user (left) and resulting orientation field (right).

of the texture as it flows across the surface. In this way, the orientation field plays the same role as the rows and columns in a set of pixels. We will discuss using a *vector field* to represent the orientation, but other representations are possible. A vector field associates with each point on the surface a vector that is tangent to the surface.

The first step in creating a vector field is to allow the user to specify a few sparse constraints that will guide the creation of the rest of the vector field. A typical system presents the user with the surface and allow the user to specify vector constraints at particular points by dragging the mouse over the surface. Figure 8 (left) shows a small number of red vectors that have been set by the user as vector field constraints. With these constraints as guides, the system then creates a vector field everywhere on the surface that interpolates these constraints. The right portion of Figure 8 shows the final vector field based on the given constraints.

Various methods can be used to extend these constraints to a complete vector field. One method is to use *parallel transport*, which is a way of moving a vector along a path from one position on the surface to another position in a way that maintains the angle between the original vector and the path. Using parallel transport, Praun et al. specify an orientation field for their lapped textures [PFH00]. Another method of satisfying the user's vector constraints is to treat them as boundary conditions for vector-valued diffusion, as was done by Turk [Tur01]. Zhang et al. sums a set of basis functions, one per vector constraint, in order to create vector fields [ZMT06]. Fisher et al. solve a linear system that they formulate using discrete exterior calculus in order to smoothly interpolate the user's direction constraints [FSDH07]. Other methods of creating orientation fields are possible, including the N-symmetry field design method of Ray et al. [RVLL08].

The end result of all of these vector field creation methods is a dense collections of points on the polygon mesh, each of which has an associated vector that is tangent to the surface. By rotating this vector field by 90 degrees, a second vector field is created that is orthogonal to the first field. The two vectors thus specified at each point form a local coordinate frame for helping perform texture synthesis. In effect, this coordinate frame allows us to step over the surface in either of two perpendicular directions just as if we are stepping from one pixel to another. Figure 9 shows the effect of using different orientation fields to guide the synthesis process. If the vector field magnitude is allowed to vary, then this can be used to specify variation in the texture scale across the surface.

With an orientation field in hand, we can turn our attention to performing the actual process of texture synthesis on the given mesh. Just as there are different ways of synthesizing texture in a regular grid of pixels, there are also various ways in which synthesis can be performed on the surface. There are at least three distinct approaches to texture synthesis on surfaces, and we will review each of these approaches.

One approach is the point-at-a-time style of synthesis, and such methods are quite similar to pixel-based methods. This method assumes that a dense, evenly-spaced set of points have been placed on the mesh. There are a number of ways in which to achieve this. Perhaps the most common method is to randomly place points on the surface and then have the points repel one another as though they are oppositely charged particles [Tur91]. It is also possible to create a hierarchy of such evenly-spaced points, so that the synthesis process can be performed at multiple scales.

With a dense set of points on the mesh, the point-at-a-time synthesis process can proceed much like the pixel-based fixed neighborhood synthesis method of Wei and

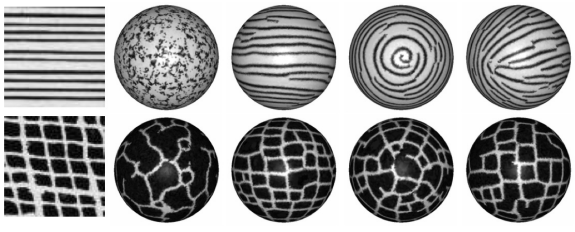


Figure 9: Result of using different orientation field for synthesis, from [WL01].

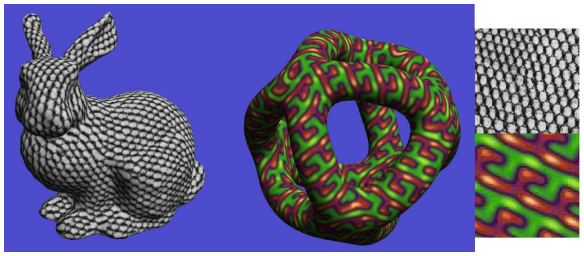


Figure 10: Surface texture synthesis, from [Tur01].

Levoy [WL00]. Each point on the surface is visited in turn, and a color neighborhood is built up for this point, much like the L-shaped neighborhoods shown in Figure 4. In building up the color samples in this neighborhood, the algorithm steps over the surface according to the two orthogonal vector fields on the surface. Moving over the surface based on these vector fields is just like moving left/right or up/down in a row or column of pixels. Once the color samples in the neighborhood of a point have been collected together, the best matching neighborhood in the texture exemplar is found and the color of the corresponding pixel from the exemplar is copied to the point in question on the surface. Once this process has been carried out for all of the points on the surface, the synthesis process is complete. This basic approach was used in [WL01] and [Tur01], and results from this approach are in Figure 10. This method can be extended to produce textures that vary in form over the surface, as was demonstrated by [ZZV*03].

A second method of performing texture synthesis on a surface is to create a mapping between regions of the plane and the surface. Synthesis is then carried out in the plane on a regular pixel grid. The work of Ying et al. exemplifies this approach [YHBZ01]. They decompose the surface into a collection of overlapping pieces, and they create one chart in the plane that corresponds to each such surface piece. Each chart has a mapping onto the surface, and the collection of such charts is an *atlas*. Each chart in the plane has a corresponding pixel grid, and the pixels in this grid are visited in scan-line order to perform texture synthesis.

A similar style of texture atlas creation was used by Lefebvre and Hoppe [LH06]. They build on their work of synthesis with small pixel neighborhoods and GPU-accelerated neighborhood matching. They use a set of overlapping charts to texture a surface, and they make use of the Jacobian map for each chart to construct their fixed-size neighborhoods. They use an indirection map in order to share samples between overlapping charts. Results of their method and the corresponding texture atlas can be seen in Figure 11.

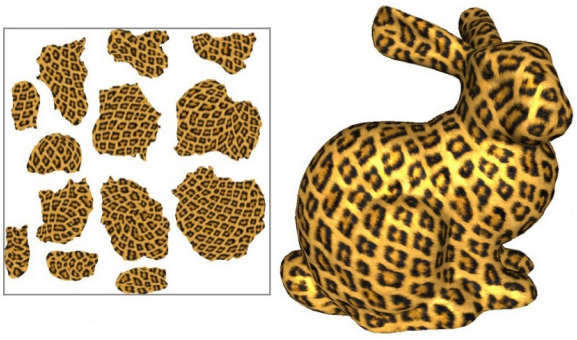


Figure 11: Texture atlas (left) and resulting textured surface (right), from the method of [LH06].

A third style of texture synthesis on surface is to treat triangles of a mesh as texture patches, similar to planar patch-based methods (see Section 3.3). One example of this approach is the work of Soler et al. [SCA02], in which they create a collection of triangle clusters. The goal of the algorithm assigns texture coordinates (that is, locations in the exemplar) to each triangle of each cluster. After the triangles of one patch have been assigned their coordinates, then an adjacent cluster is examined and the exemplar is searched to find a region that matches at the point where the two clusters meet. This is similar to finding high quality overlaps between regions in patch-based 2D synthesis approaches such as [EF01].

Zelnika and Garland have extended their jump-maps to performing texture synthesis on surfaces [ZG03]. Just like their 2D texture synthesis approach [ZG04], they first pre-process the texture to locate pairs of pixels in the exemplar that have similar neighborhoods. Using this information, they assign (u,v) texture coordiinates to the triangles of a mesh. They visit each triangle of the mesh according to a user-created vector field. When a new triangle is visited, its already-visited neighbors are examined and the jump map is used to find a location in the exemplar that is a good match.

5. Dynamic Texture Synthesis

So far we have described how to synthesize 2D textures, either on an image plane or over a mesh surface. The same principles can also be applied to synthesizing dynamic textures, *i.e.* textures whose appearance evolves over time. A typical example is video of a dynamic phenomena; however, other modalities are also possible, *e.g.* time-variant appearance of materials [GTR*06] (see Section 9). Besides using a video exemplar as input, another approach to synthesizing dynamic textures is to start with static image exemplars, but achieve dynamism by changing synthesis parameters as a function of time. An example is the texturing of an animated fluid surface, guided by the flow of the fluid, using a bubbles or waves texture as the exemplar. We focus on video texture synthesis in this section and describe flow-guided texture synthesis in the next section.

5.1. Temporal Video Texture Synthesis

A commonly studied case for texture synthesis from video is the treatment of video as a temporal texture. This treatment of video relies on the applicability of the Markov Random Field (MRF) property to videos, *i.e.* the video needs to exhibit *locality* and *stationarity* as described in Section 2. In the case of image textures, these properties manifest themselves as repeating patterns in the spatial domain. On the other hand, a video may consist of purely temporal patterns that repeat over the course of the video, *e.g.* a person running on a treadmill, or a swinging pendulum. Such videos may be treated as *pseudo one-dimensional* textures along time.

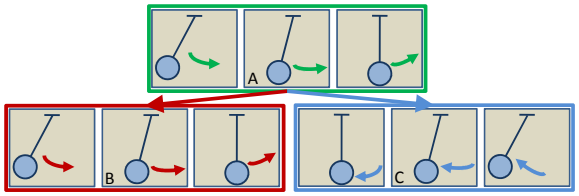


Figure 12: Pendulum (arrows indicate velocity): transitioning from frame A to B preserves dynamics; transitioning from A to C does not.

Schödl et al. [SSSE00] present a general technique for finding smooth transitions and loops in a given input video clip. They synthesize *infinitely playing* video clips, which may be used as dynamic backdrops in personal web pages, or as replacement for manually generated video loops in computer games and movies. The idea behind their technique is to find sets of matching frames in the input video and then transition between these frames during playback. In the first phase of the algorithm, the video structure is analyzed for existence of matching frames within the sequence. This analysis yields a *transition matrix D* that encodes the cost of jumping between two distant frames during playback. The similarity between two frames i and j is expressed as

$$D_{ij} = \| \mathcal{I}_i - \mathcal{I}_j \|_2,$$

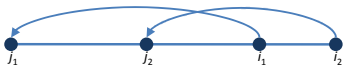
which denotes the L_2 distance between each pair of images $\mathcal{I}_i$ and $\mathcal{I}_j$. During synthesis, transitions from frame i to frame j are created if the successor of i is similar to j, *i.e.* whenever $D_{i+1,j}$ is small. A simple way of doing this is to map these distances to probabilities through an exponential function,

$$P_{ij} \propto e^{-D_{i+1,j}/\sigma}.$$

where σ controls the sensitivity of this distribution to the L_2 distance. Then, at run time, the next frame j, to be displayed after frame i, may be selected according to P_{ij}.

Preserving Dynamics: Video textures need to preserve the dynamics of motion observed in the video in addition to similarity across frames. Consider, for example, a swinging pendulum (illustrated in Figure 12). While the appearance of the pendulum in frame A matches both frames B and C, its dynamics are only preserved when transitioning from A to B. On the other hand, jumping from A to C would lead to an abrupt and unacceptable change in pendulum's motion. One way to overcome this problem is to consider *temporal windows* of adjacent frames when computing similarity, *i.e.* matching sub-sequences instead of individual frames.

Video playback: To play a video texture, one option is to randomly transition between frames based on the pre-computed probability distribution matrix. However, *video loops* are preferable when playing the video texture in a conventional digital video player. Given the transition matrix, one can pre-compute an optimal set of transitions that generate a video loop. A loop is created by making backward transitions, *i.e.* by going from frame i to j, such that $j < i$. The cost of this loop is the cost D_{ij} of this transition. Several such loops may be combined to form a *compound loop* if there is overlap between loops. If two loops have transitions $i_1 \rightarrow j_1$ and $i_2 \rightarrow j_2$ respectively, where $j_1 < i_1$ and $j_2 < i_2$, then they overlap if $j_1 < i_2$ and $j_2 < i_1$ (see inset figure for an example). One can then combine the two loops as $i_1 \rightarrow j_1 \ldots i_2 \rightarrow j_2$. An optimal compound loop minimizes the total cost of its primitive loops' transitions, while constraining the union of lengths of these primitive loops to be equal to the desired length of the synthesized sequence.



Controllable synthesis: It is possible to add finer control to the synthesis process with user-guided motion paths and velocities. For example, if synthesizing a video of a runner on a treadmill, one may control the speed of the runner by selectively choosing frames from the input video with the appropriate motion; assuming different segments of the input video capture the runner running at various speeds. Another example is to use *video sprites*, such as the fish shown in Figure 13, as input – a video sprite may be constructed by subtracting the background to reveal just the foreground object and then compensating for motion by aligning its centroid to the origin. In this example, the motion of the fish is controlled by the user with a mouse: not only is the fish moved smoothly towards the mouse, but also its sprite frames are chosen to match the original velocity in those frames with the motion requested by the user. A more sophisticated technique for controllable synthesis using video sprites was presented in [SE02].

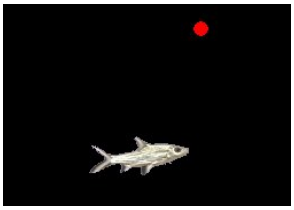


Figure 13: Mouse-controlled fish [SSSE00].

5.2. Spatio-Temporal Synthesis

Videos are not always limited to being temporal textures. Several phenomena, such as river flow, waterfalls, smoke, clouds, fire, windy grass fields, etc. exhibit repeating patterns in both space and time. In such cases, it is beneficial to treat them as 3D spatio-temporal textures as opposed to purely temporal textures, which has two advantages. Firstly, it allows for more flexible synthesis algorithms that can exploit the spatial structure of the video in addition to its temporal structure. Secondly, several 2D algorithms can be directly lifted to the 3D spatio-temporal domain. The trade-off

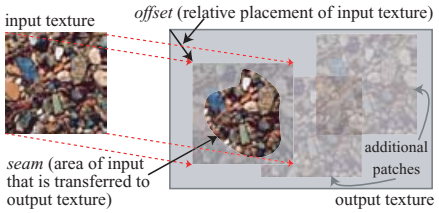
is the higher dimensionality of the problem, which can lead to increased computational cost and even impact synthesis quality if the algorithms are not designed carefully.

Pixel-based techniques: Bar-Joseph et al. [BJEYLW01] extend the 2D approach of De Bonet [De 97] to work for time-varying textures. De Bonet's technique works by building a multi-resolution tree of wavelet filter coefficients of the input image texture. The synthesis phase creates a new *output* tree by statistical sampling of these filter coefficients. The output tree is constructed in a coarse-to-fine fashion: at each resolution level, the filter coefficients are sampled conditioned upon the coefficients synthesized in the previous coarser levels. This sampling is performed such that the local statistics of coefficients are preserved from the input tree. In [BJEYLW01], this approach is extended to video by performing a 3D wavelet transform in space-time. However, a specially designed 3D transform is used, which is separable in the time dimension. This leads to a considerable speed up in the otherwise expensive 3D filtering operation, and also allows asymmetric treatment of space and time.

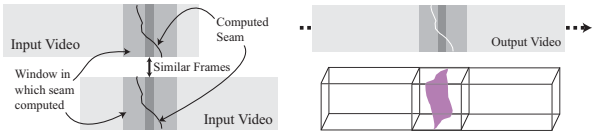
Wei and Levoy [WL00] (see Section 3.1) apply their texture synthesis technique to spatio-temporal textures such as fire, smoke and ocean waves. One of their contributions is the use of multi-resolution input neighborhoods and a tree structure to store them, which significantly accelerates the search phase, and makes their algorithm feasible for video synthesis, where the neighborhoods are larger in both size and number. For video, the algorithm proceeds as in 2D, but uses 3D spatio-temporal neighborhoods instead of 2D spatial ones, and synthesizes the entire video volume at once.

Patch-based techniques: While the above described pixel-based techniques are able to capture the local statistics of a spatio-temporal texture quite well, they sometimes fail to reproduce its global structure. This usually happens if the input consists of a larger global appearance in addition to its local textural properties. For example in Figure 15, the fire has a global structure to its flame. The same is true for the smoke, the sparkling ball, and the waterfall. Patch-based techniques generally perform better in these difficult cases because they operate upon a larger window of pixels at a time, and therefore are able to preserve the global structure of the texture in addition to its local properties.

In the case of video, a patch is a 3D block in space-time (see Figure 14b). One can interpret the temporal synthesis technique [SSSE00] described in Section 5.1 as a peculiar case of patch-based synthesis, where the patch size is the entire frame in space and the temporal window length in time; although during synthesis, the patch only has freedom to displace in time. It is therefore able to preserve the spatial structure perfectly and only introduces artifacts in time. While this is sufficient for purely temporal synthesis, for spatio-temporal textures, it makes sense to allow these patches to displace in both space and time.



(a) Patch placement and seam computation in 2D



(b) Seams in video: $y-t$ slices (left column and top-right) and 3D seam *surface* (bottom-right).

Figure 14: Patches and seams in the graph cuts algorithm [KSE*03].

Kwatra et al. [KSE*03] present a technique for both image and video synthesis that treats the entire input texture as one big patch. It works by stitching together appropriately placed copies of this 2D or 3D (space-time) patch in the output, using a graph cuts algorithm (see Figure 14a). In 2D, the graph cut algorithm bears a lot of similarity to the dynamic programming based image quilting technique [EF01] (see Section 3.3). However, its ability to work in 3D or even higher dimensions is a distinct advantage that the graph cut approach enjoys over image quilting. The graph cut algorithm computes a seam between overlapping patches, which in 3D, becomes a *surface* and separates two or more spatio-temporal blocks (as shown in Figure 14b, bottom-right).

An important aspect of this technique is the determination of where to place the new patch in the output domain before applying graph cut. There are several different options: (1) For highly stochastic textures, it is sufficient to just randomly pick the output location. (2) For videos that are primarily temporal textures with too much structure in the spatial domain, it is appropriate to only limit the patch displacements to time. In this case, one may first find matching frames or sub-sequences within the video (using methods described in Section 5.1), and then displace the video so that these frames are aligned with each other, before invoking graph cuts to find a temporal transition surface within a window around these frames (Figure 14b illustrates this case). (3) For general spatio-temporal textures, the best results are obtained by searching for an output location where the input patch is visually most consistent with its surrounding patches. This requires a sliding window search for the best match, which can be computationally expensive: $O(n^2)$ in the number of pixels in the output video. However, *Fast Fourier Transforms* (FFT) can be used to bring down the complexity to $O(n \log n)$, which is several orders of magnitude faster, espe-

Figure 15: Video synthesis using graph cuts [KSE*03].

cially for video. Figure 15 shows frames from sample video synthesized using the graph cuts based approach.

Agarwala et al. [AZP*05] extend the graph cut approach to synthesize panoramic video textures. They take the output of a panning video camera and stitch it in space and time to create a single, wide field of view video that appears to play continuously and indefinitely.

5.3. Parametric Dynamic Texture Modeling

An alternate formulation for video texture synthesis is to interpret the video sequence as the outcome of a *dynamical system*. This interpretation still treats the video as a Markov process, where either a single frame is conditioned upon a small window of previous frames [DCWS03], or a single pixel is conditioned upon a small spatio-temporal neighborhood of pixels [SP96]. However, these conditional relationships are expressed in parametric form through *linear auto-regressive* models, which is in contrast to the non-parametric nature of techniques described in previous sections.

In particular, Doretto et al. [DCWS03] first express the individual frames in a reduced dimensional space, called the state-space, which is typically obtained through Principal Component Analysis (PCA) of the original vector space spanned by the frames. Then, a linear dynamical system (LDS), expressed as a transformation matrix, that converts the current state to the next state in the sequence is learned from the input states – the state may represent a dimensionality-reduced frame or sub-sequence of frames. A closed form solution for learning this LDS transformation matrix may be obtained by minimizing the total least-square error incurred when applying the transformation to all pairs of current and subsequent states of the input sequence. If the input video sequence consists of frames $\mathcal{I}_1, \mathcal{I}_2 \ldots \mathcal{I}_n$, and has the corresponding dimensionality-reduced states $x_1, x_2 \ldots x_n$, then their relationship may be expressed as:

$$\begin{aligned} \mathcal{I}_i &= Cx_i + w_i, \\ x_{i+1} &= Ax_i + v_i, \end{aligned}$$

where the matrix C projects states to frames and is obtained through PCA. w_i and v_i are white zero-mean Gaussian noise terms. A is the transformation matrix between subsequent states, and may be estimated by solving the following least squares minimization problem:

$$A = \arg\min_{A'} \sum_{i=1}^{n-1} \| x_{i+1} - A' x_i \|^2.$$

During synthesis, this transformation matrix is repeatedly applied upon the current state to reveal the subsequent state. The synthesized states are then projected back to the original space of frames to obtain the synthesized video texture.

While this formulation works for short input sequences, it gets increasingly difficult to learn a single satisfactory transformation matrix A for long sequences with several conflicting state changes. A solution that alleviates this problem is to take a mixed parametric and non-parametric approach, where several different LDS matrices are learned within clusters of similar states. Additionally, a transition probability table is learned for jumping between these different clusters. During synthesis, one may jump to a cluster and then synthesize from its LDS before again moving to a different cluster. This approach was taken by Li et al. [LWS02], albeit for *character motion* synthesis.

6. Flow-guided Texture Synthesis

In the previous section, we talked about dynamic texture synthesis using videos as exemplars. A different way of synthesizing dynamic textures is to *animate* static image exemplars, guided by a flow field. The topic of animating textures using flow fields has been studied for the purpose of flow visualization as well as rendering fluids (in 2D and 3D). There are two aspects to be considered when animating textures using flow fields. Firstly, the animated texture sequence should be consistent with the flow field, *i.e.* it should convey the motion represented by the flow. Secondly, the synthesized texture should be visually similar to the texture exemplar in every frame of the animation.

A simple way to enforce flow consistency is to use the flow to *advect* the texture. In other words, the texture is warped using the flow field to obtain the next frame in the sequence. Of course, if this process is continued indefinitely, the texture would likely get unrecognizably distorted or could even disappear from the viewport, which violates the latter criteria of maintaining visual consistency. Several techniques have explored different ways of measuring and preserving visual consistency, but they can be broadly classified into two categories: (1) techniques that preserve statistical properties of the texture, and (2) techniques that enforce appearance-based similarity.

The two sets of approaches have different trade-offs. Statistical approaches work by blending together multiple textures over time, which works quite well for procedural and white noise textures, and can be easily adapted to run in

real-time [vW02, Ney03]. The drawback is the introduction of ghosting artifacts in more structured textures, due to blending of mismatched features. Appearance-based approaches, on the other hand, use example-based texture synthesis to synthesize each frame, but force the synthesis to stay close to the texture predicted by flow-guided advection [KEBK05, LH06, HZW*06]. The advantage is more generality in terms of the variety of textures they allow. However, if the texture consists of large features, flow consistency may not always be preserved because these features may induce an artificial flow of their own. Another drawback is the running time, which can be several seconds per frame; however, real-time implementations are possible [LH06].

Statistical approaches: Image based flow visualization [vW02] is a real-time technique for texture animation, that uses simple texture mapping operations for advection and blending of successive frames with specially designed *background* images for preserving visual consistency. These background images are typically white noise textures, filtered spatially and temporally based on careful frequency-space analysis of their behavior. This technique is only loosely *example*-based, since the images used for blending are pre-designed. Although arbitrary foreground images may be used, they are only used for advection.

Neyret [Ney03] also uses a combination of advection and blending to synthesize animated textures. However, the textures used for blending are obtained from the same initial texture by advecting it for different time durations (or *latencies*). The latencies used for blending are adapted based on the local velocity and deformation of the domain. This technique accepts both image and procedural textures as input. However, it works best with procedural noise textures, as a special frequency-space method can be used for blending those textures, which reduces the ghosting artifacts. Figure 16a shows an example where a flow field (left) is textured using this technique (right) in order to amplify its resolution.

Appearance-based approaches: Kwatra et al. [KEBK05] augment their optimization-based texture synthesis technique to preserve flow consistency in addition to appearance similarity. They use the advected texture from the previous frame as a soft constraint by modifying (1) to obtain

$$E_{\mathbf{f}}(\mathbf{x};\{\mathbf{z}_p\}) = \sum_{p\in X^\dagger} \|\mathbf{x}_p - \mathbf{z}_p\|^2 + \lambda\|\mathbf{x}_p - \mathbf{a}_p\|^2.$$

where $\mathbf{a}_p$ refers to the neighborhoods in the advected texture $\mathbf{a}$. This favors the solution for the current frame, $\mathbf{x}$, to stay close to $\mathbf{a}$. Additionally $\mathbf{a}$ is also used to initialize $\mathbf{x}$, which encourages the search step to prefer neighborhoods $\{\mathbf{z}_p\}$ that match the advected texture. One problem with this approach is that the solution for the current frame is obtained by blending the advected texture with the searched neighborhoods, which can sometimes lead to excessive blurring. This problem is effectively handled by using a discrete solver for optimization [HZW*06] which replaces the blending operation with a copy operation that picks the discrete pixel that best minimizes $E_{\mathbf{f}}$ (see Section 3.4).

Lefebvre and Hoppe [LH06] present an example-based texture advection algorithm that works in real-time. They perform advection in coordinate-space as opposed to color-space, *i.e.* they remember the distortions introduced by the advection over the initial texture grid. This allows them to determine and selectively correct just the areas of excessive distortion (see Section 7 for more details on how they achieve real-time performance).

6.1. Video Exemplars

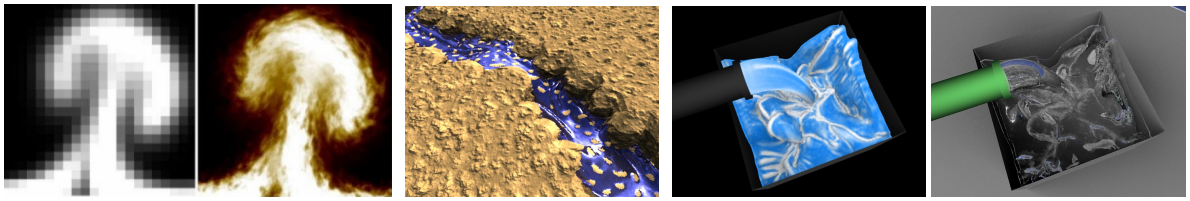
Bhat et al. [BSHK04] present a flow-based synthesis technique that uses *video* as input. They use sparse flow lines instead of a dense field to guide the synthesis. A user marks flow lines in the input video and maps them to new flow lines in the output. The algorithm then transfers the video texture from the input to the output while maintaining its temporal evolution along the specified flow lines. One can think of this approach as restricting the video texture synthesis problem discussed in Section 5.1 to these user-specified flow lines, as opposed to the entire frame. However, the difficulty arises from the fact that the flow lines are free to change shape. This problem is addressed by representing the flow line as a string of particles with texture patches attached to them. During synthesis, these particle patches are treated as small moving video textures that periodically transition between input frames to create seamless, infinite sequences. The different particles are transitioned at different times in the output sequence, in a staggered fashion, to remove any visual discontinuities that may be apparent otherwise.

Narain et al. [NKL*07] present a technique that allows using *dense* flow fields in conjunction with video exemplars. They first decouple the video into motion and texture evolution components. During synthesis, the original motion in the video is replaced by a new flow field, which may be manually specified or obtained from a different video's motion field, but the texture evolution is retained.

6.2. Texturing Fluids

A compelling application of flow-guided texture animation is the texturing of 3D fluids. While fluid simulation is the *de facto* standard in animating fluid phenomena, texture synthesis can be used to augment simulations for generating complex, small scale detail that may be computationally too expensive to simulate. It can also be used as a rendering engine, or to generate effects that are artistically or aesthetically interesting but not necessarily physically correct, and therefore not amenable to simulation.

Kwatra et al. [KAK*07] and Bargteil et al. [BSM*06] extend the 2D texture optimization technique of [KEBK05] to



(a) courtesy [Ney03] (b) courtesy [KAK*07] (c) courtesy [NKL*07]

Figure 16: Flow-guided texture synthesis examples.

the 3D domain for texturing liquid surfaces. Besides texturing the surface with arbitrary textures to synthesize interesting animations, these methods can also be used to add detail such as waves, bubbles, etc to the fluid. Figure 16b shows an example where a river is textured with floating leaves.

An important difference between texturing 2D flows and 3D fluids is the need to perform texture synthesis on a non-planar liquid surface that is also dynamically changing over time. While [HZW*06] and [LH06] present methods for animating textures on arbitrary non-planar surfaces, they assume the surface to be static. With a dynamic surface, the texture needs to be advected volumetrically in 3D and re-projected back on to the liquid surface in every time step. For anisotropic textures, this includes advecting a vector field that guides the orientation of the texture over the surface (Section 4 explains the need for this vector field). Advecting the orientation vector field is more complicated than advecting scalar quantities such as texture coordinates or colors, because while scalars only get translated under the advection field, orientation vectors additionally also undergo rotation. This combined distortion is formulated in [KAK*07] as the solution of a *vector advection* equation.

In [NKL*07], further connections are explored between orientation fields and fluid flow features such as curvature, curl, divergence, etc. They also perform globally variant synthesis (see Section 9) in which the appropriate texture exemplar is chosen based on these features. Figure 16c shows an example where curvature-based features (visualized on left) are selectively rendered using a turbulent texture (right).

Yu et al. [YNBH09] present a real-time technique for animating and texturing river flows. Besides generating the river flows procedurally, they also present a methodology for rendering these flows using procedural textures. They attach texture patches to particles, which are advected by the flow field and periodically deleted and regenerated to maintain a Poisson disk distribution. Rendering is done by blending these patches together. This technique is similar in spirit to the statistical flow texturing techniques described in the previous section. However, it uses a Lagrangian particle-based approach for advection in contrast to the Eulerian grid-based approaches taken by prior methods.

7. Runtime Texture Synthesis

The texture synthesis algorithms we have described so far generate an entire image at once: The algorithm runs once and the output is stored for later display. While this saves authoring time, it still requires as much storage as a hand–painted image. This is unfortunate since the synthesized images essentially contain no more information than the exemplar. Besides storage, this also wastes synthesis time: The final rendering generally only needs a subset of the entire texture, either in spatial extent or resolution.

The ability to compute the appearance of a texture on-the-fly at any given point during rendering was introduced by *procedural texturing* [EMP*02]. A procedural texture is defined by a small function computing the color at a point using only coordinates and a few global parameters. This requires very little storage, and computation only occurs on visible surfaces. While many approaches have been proposed to create visually interesting procedural textures, it is in general very difficult to reproduce a given appearance. A drawback that texture synthesis from example does not have. Hence, we would like the best of both approaches: The fast point evaluation and low memory cost of procedural textures, and the by–example capabilities of approaches described earlier.

The two key properties missing from previously described algorithms are (see Figure 17): 1) Spatial determinism - the computed color at any given point should always remain the same, even if only a subset is computed - and 2) local evaluation - determining the color at a point should only involve knowing the color at a small number of other points. In addition, a run–time on–the–fly texture synthesis scheme should be fast enough to answer synthesis queries during rendering.

As often in texture synthesis, two categories of approaches have been proposed. A first is inspired by per–pixel synthesis schemes (Section 7.1), while the second is most similar to patch–based approaches (Section 7.2).

7.1. Per–pixel runtime synthesis

The work of Wei and Levoy [WL02] is the first algorithm to target on-the-fly synthesis. The key idea is to break the sequential processing of previous algorithms, where pixels are

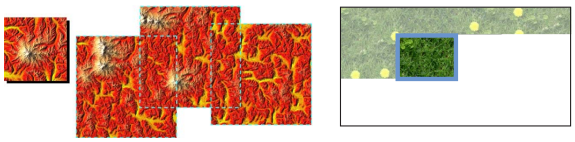


Figure 17: *Left:* Overlapping regions synthesized independently must correspond. *Right:* Spatial determinism requires proper boundary conditions. Here, the subregion in the blue rectangle is requested. With sequential synthesis all the content located before (shown ghosted) must be synthesized in order to guarantee spatial determinism.

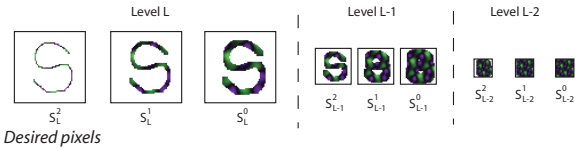


Figure 18: The pixels to be synthesized throughout the pyramid is determined from the set of desired pixels at the last iteration S_L^2, using the neighborhood shape to dilate the set.

generated in sequence (typically scan–line ordering): The color of new pixels depends on the color of all previously synthesized pixels. Instead, Wei and Levoy cast texture synthesis as an iterative process. A first image, made by randomly choosing pixels from the exemplar is iteratively improved using neighborhood matching. The important point is that neighborhoods are always read from the *previous* step image, while new colors are written in the *next* step image. Hence, the computations performed at each pixel are independent: Pixels can be processed in any order without changing the result. In addition, the dependency chain to compute the color of a single pixel remains local and is easily determined: to compute the color of a pixel p, its neighborhood $\mathcal{N}(p)$ must exist at the previous iteration. By using this rule throughout the entire multi–resolution synthesis pyramid, the set of pixels that must be computed at each iteration is easily determined from the set of desired pixels at the last iteration (see Figure 18). Using a caching scheme for efficiency, this approach can answer random queries in a synthesized texture, generating the content on-the-fly as the renderer requires it (see Figure 19).

Unfortunately order-independent neighborhood matching is less efficient and many iterations are required to achieve good results. Lefebvre and Hoppe [LH05] proposed a new parallel texture synthesis algorithm to overcome this issue. A guiding principle of the algorithm is that it manipulates pixel *coordinates* rather than colors directly: Whenever a best matching neighborhood is found, the coordinates of its center are recorded, rather than its color. Colors are trivially recovered with a simple lookup in the example image (see Figure 20). Manipulating coordinates has several advantages

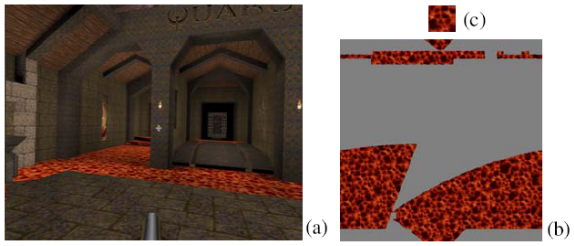


Figure 19: A frame from the game Quake enhanced with on–the–fly texture synthesis. *(a)* Texture synthesis is used on the lava texture preventing unnatural repetitions. *(b)* Content of the texture cache after rendering the frame. Gray regions have not been used by the renderer. *(c)* 48^2 exemplar.

exploited in the three main steps of the algorithm: Upsampling, Jitter and Correction (see Figure 21). These steps are performed at each level of the synthesis pyramid.

- The Upsampling step increases the resolution of the previous level result. Since the algorithm manipulates coordinates, this is done through simple coordinate inheritance rather than neighborhood matching. A key advantage is that coherent patches are formed through this process.
- The Jitter step introduces variety *explicitly* in the result, adding an offset to pixel coordinates during the multi-resolution synthesis process. This visually displaces blocks of texture in the result. Neighborhood matching (next step) will recover from any perturbation incompatible with the exemplar appearance. An important property is that in absence of Jitter a simple tiling of the exemplar is produced. Explicit Jitter enables powerful controls over the result, such as drag–and–drop.
- The last step is the Correction step. It performs a few iter-

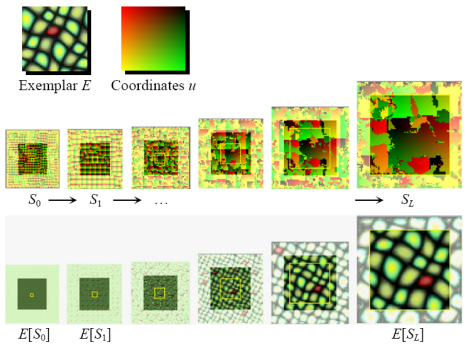


Figure 20: Given an exemplar, *coordinates* are synthesized into a coarse-to-fine pyramid; the bottom row shows the corresponding exemplar colors. Padding of the synthesis pyramid ensures that all necessary pixels are synthesized for a spatially deterministic result.

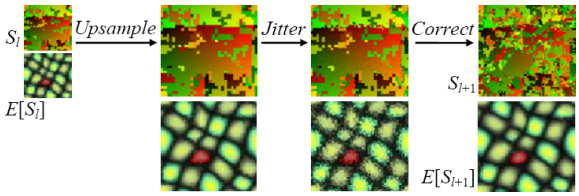


Figure 21: The three steps of the parallel synthesis algorithm of Lefebvre and Hoppe [LH05]. These are performed at each level of the synthesis pyramid.

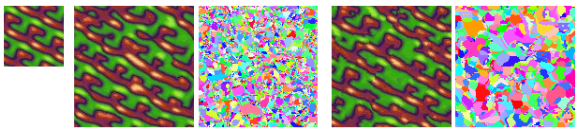


Figure 22: Synthesis results and corresponding patches (color coded). *Left:* The parallel algorithm tends to form patches (i.e. small pieces of the exemplar). *Right:* Favoring coherent candidates produces larger patches (whether this is desirable depends on the texture).

ations of order–independent neighborhood matching. An interleaved processing pattern is used to further improve synthesis efficiency. The search space is kept small by using k-coherent candidates [TZL*02] (see Section 3.2).

For efficiency, the algorithm synthesizes rectangular windows of texture rather than answering point queries (see Figure 20). The GPU implementation achieves real–time on–the–fly synthesis, enabling interactive exploration of infinite texture domains while manipulating parameters.

This approach has been later extended [LH06] to synthesize textures under distortions, as well as textures containing higher-dimensional data.

7.2. Tile–based runtime synthesis

Tile–based texture synthesis is based on the notion of *tilings*: Coverage of the plane obtained by putting together a finite set of corresponding *tiles*. In general, tiles are polygons of arbitrary shape with the only requirement that there must remain no hole once the plane is paved.

The simplicity of tilings is only apparent: Tilings are an extensively researched and fascinating mathematical object. Challenges arise when considering properties such as the periodicity of a tiling – whether it repeats itself after some translation – or the set of positions and rotations that tiles can take. Of particular interest is the problem of finding the smallest set of tiles that will *never* produce a periodic tiling. Examples of such *aperiodic* tile sets are the Penrose tiles, the 16 Ammann [GS86] tiles and the 13 Wang tiles [II96].

For a complete overview of tilings, we invite the interested reader to refer to the book "Tilings and patterns" [GS86].

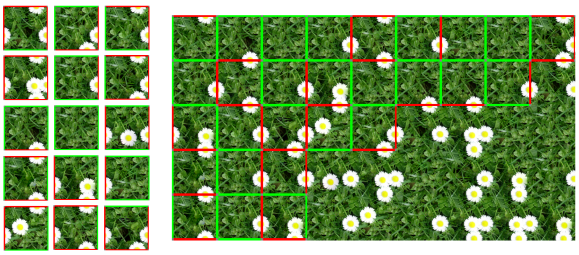


Figure 23: *Left:* Square tiles with color coded edge constraints. The small images have corresponding content along vertical and horizontal edges of same color. *Right:* Random tilings produce different images. Notice how edge constraints are preserved in order to obtain a continuous image.

The use of tilings in Computer Graphics goes back to the early side–scrolling platform games. To save time and memory, many game levels were created by putting together small square tiles, each one supporting a small image. By arranging the tiles in different ways through a *tile–map*, various level layouts were obtained. Because the images have corresponding boundaries, the tile outlines are invisible (see Figure 23). This is similar to a jigsaw puzzle where pieces could be put in different orders to create different images.

Several tiling schemes have been proposed for texture synthesis. Stam [Sta97] creates animated textures using an aperiodic tiling of animated square tiles. The tiles have colored edges representing edge compatibility (see Figure 23). Neyret and Cani [NC99] use triangular tiles to texture complex surfaces with little distortion. Cohen et al. [CSHD03] use a set of tiles similar to Stam [Sta97] but with improved construction rules. One important motivation is that an aperiodic tiling in the strict mathematical sense may not be visually pleasant: Typically many repetitive structures appear. The authors rely on patch–based texture synthesis to generate the interior of the tiles while enforcing edge constraints (see Figure 24). Fu and Leung [FL05] show how to apply this planar scheme to surfaces through a polycubemap parameterization [THCM04]. Lagae and Dutré [LD06] investigate the use of corner constraints for the tiles, which has the advantage of introducing diagonal neighbor constraints.

Similarly to per-pixel synthesis, many of these schemes are sequential and do not support run–time synthesis. Lefebvre and Neyret [LN03] proposed to generate aperiodic tilings directly at rendering time, from the pixel coordinates. First, the algorithm computes in which cell of the tiling the pixel lies. Then, it uses a pseudo–random generator to randomly choose a tile. The spatial distribution of tiles is controlled by the user. The approach also enables random distribution of small texture elements. Wei [Wei04] combined this approach with the work of Cohen et al. [CSHD03], choosing

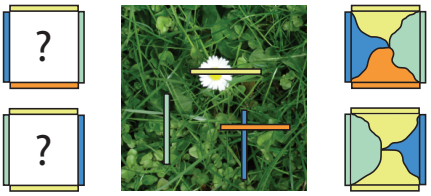


Figure 24: *Left:* Two tiles with four different border constraints must be created. *Middle:* A position for each colored edge is chosen in the source image. This defines the content that will appear along each edge. *Right:* Each edge provides a positioning of the source image in the tile. Graph cut optimization is used to find optimal transitions between the edges. (Several variants are possible, such as adding a patch in the center of the tile to increase visual variety).

edge color constraints with a pseudo-random generator and ensuring a tile exists for all possible combinations of edge colors. Kopf et al. [KCODL06] proposed a recursive version of the same scheme: Tiles are subdivided into smaller tiles, while preserving edge compatibility. The authors demonstrate run–time texture synthesis by randomly positioning many small texture elements (e.g. leaves) in the plane. This scheme is also used to quickly generate stippling patterns, another popular application of tilings in graphics [LKF*08].

An additional challenge of procedural tilings is to provide texture filtering (bi–linear interpolation, MIP–mapping). Without special treatment discontinuities appear at tile boundaries: The graphics hardware filters each tile separately, unaware of the procedural layout of tiles. While it is possible to bypass and re–program filtering in the pixel shader, this is extremely inefficient. Wei [Wei04] packs the tiles of his scheme in a manner which lessens filtering issues. While it works very well on tiles of homogeneous content, visual artifacts appear if the tiles are too different. A more generic solution exploiting latest hardware capabilities is proposed in [Lef08].

8. Solid Texture Synthesis

Solid textures define color content in 3D. They are the Computer Graphics equivalent of a block of matter. Applying a solid texture onto an object is easy, since colors are directly retrieved from the 3D coordinates of the surface points. Solid textures give the impression that the object has been carved out a block of material such as wood or marble.

Solid texturing first appeared in the context of procedural texturing [EMP*02]. Procedural textures are very efficient at this task since they require little storage. Nonetheless, defining visually interesting materials in a volume is challenging, even more so than in 2D. Hence, several texture synthesis from example approaches have been investigated to synthesize volumes of color content from example images. The goal is to generate a volume where each 2D slice looks visually similar to the 2D example. Some algorithms support different 2D examples along different planes, in which case they must be consistent in order to produce convincing results. Figure 25 shows results of solid texture synthesis.

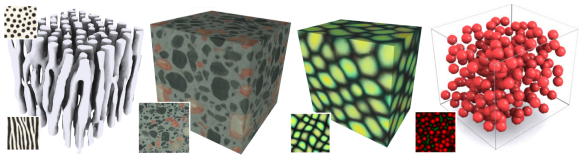


Figure 25: Results of solid texture synthesis using the scheme of Dong et al. [DLTD08]. Transparency is used in the first and last example to reveal internal structures.

The pyramid histogram matching of [HB95] and the spectral analysis methods of [GD95, GD96] pioneered the work on solid texture synthesis from example. The former reproduces global statistics of the 2D example images in the volume, while the latter two create a procedural solid texture from the spectral analysis of multiple images. Dischler et al. [DGF98] proposed to use both approaches (spectral and histogram–based) in a same hybrid scheme. While very efficient on stochastic textures, these approaches do not perform well on structured patterns. For more details, please refer to the survey of Dischler and Ghazanfarpour [DG01]. Jagnow et al. [JDR04] proposed a solid synthesis method targeted at aggregates of particles, whose distribution and shape is analyzed from an image. It is specialized for a class of materials and does not support arbitrary examples.

Several approaches based on neighborhood matching techniques have been proposed to support more generic input. Wei [Wei02] adapted 2D neighborhood matching synthesis schemes to 3D volumes. The key idea is to consider three 2D exemplars, one in each direction. In each pixel of the output volume (*voxel*), three interleaved 2D neighborhoods are extracted. The best matches are found independently in each of the three exemplars. The color of the voxel is updated as the average of the three neighborhood center colors. This process is repeated in each voxel, and several iterations are performed on the entire volume. Figure 26 illustrates this idea. Kopf et al. [KFCO*07] rely on a similar formulation, but use a global optimization approach [KEBK05]. A histogram matching step is introduced, further improving the result by making sure all areas of the exemplar receive equal attention during synthesis. Qin and Yang [QY07] synthesize a volume by capturing the co-occurrences of grayscale levels in the neighborhoods of 2D images.

These approaches truly generate a 3D grid of color content, requiring a significant amount of memory (e.g. 48MB for a 256^3 RGB volume, 3GB for a 1024^3 volume). Instead, other methods avoid the explicit construction of the full solid texture. Pietroni et al. [POB*07] define the object interior

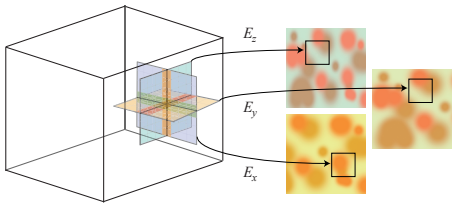


Figure 26: The scheme of Wei [Wei02]. At each voxel, three axis aligned 2D neighborhoods are extracted. The three exemplars are search independently for a best match. The new color at the voxel is computed as the average of the three neighborhood centers.

by morphing a set of 2D images positioned within the object by the user. Owada et al. [ONOI04] adapted guided 2D synthesis [HJO*01] to illustrate object interiors. The interesting aspect of this approach is that it only hallucinates a volume: When the user cuts an object, a new surface appears giving the illusion that the object interior is revealed. In fact, the color content is not defined in a volume and discontinuities can appear at transitions between different cuts. Takayama et al. [TOII08] define a texture inside an object by applying a small solid texture to a tetrahedrization of the interior. Each tetrahedron can be seen as the 3D equivalent of a 'patch', the approach defining the 3D analog of a Lapped Texture [PFH00]. When cutting the object the new surface intersects the textured tetrahedrons, giving the illusion of a continuous solid texture. This approach has two key advantages: First, the tetrahedrons require much less storage than a high–resolution 3D grid covering the entire object. Second, by using a solid texture of varying aspect (e.g. from core to crust) it is possible to arrange the tetrahedrons so as to simulate a solid texture which appearance varies inside the object. The key disadvantage of the technique is to require a fine-enough tetrahedrization of the interior, which can become problematic with many different objects or objects of large volume. The approach of Dong et al. [DLTD08] is specifically designed to address this issue. It draws on the run–time 2D texture synthesis approaches (see Section 7.1) to synthesize the volume content lazily. The key idea is to only synthesize parts of the volume which are used to texture a surface. When the user cuts an object, more voxels are synthesized on–the–fly for the newly appearing surfaces. The result is indistinguishable from using a high-resolution solid texture. Still, the cost of synthesis for a surface is comparable to the cost of 2D synthesis, both in time and space. A key challenge is to reduce the number of neighborhood matching iterations to keep computations local. This is achieved by pre–computing 3D neighborhoods during the analysis step. They are obtained by interleaving well–chosen 2D neighborhoods from the exemplars. During synthesis these 3D neighborhoods are used: Instead of dealing with three intersecting 2D problems, the synthesizer truly works in 3D making it both simpler and more efficient.

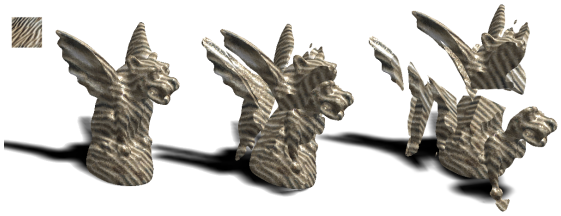
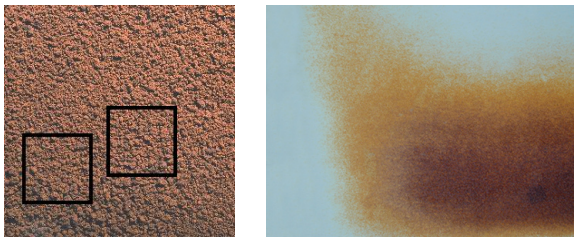


Figure 27: Three frames of an object exploding in real–time [DLTD08]. The texture of newly appearing surfaces is synthesized on–the–fly. Notice the continuity of features between the hull and the internal surfaces: Even though colors are only computed around the surface, the result is indistinguishable from using a complete solid texture.

9. Globally-varying Textures

In Section 2.3, we define textures as satisfying both the local and stationary properties of Markov Random Fields. Even though this definition works for stationary or homogeneous textures, it excludes many interesting natural and man-made textures that exhibit globally-varying pattern changes, such as weathering or biological growth over object surfaces. In this section, we extend the definition of textures to handle such globally varying textures.

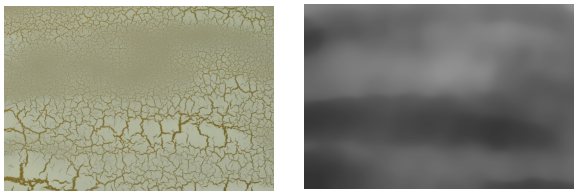


stationary globally varying

Figure 28: Stationary versus globally-varying texture. A stationary texture satisfies both of the MRF properties: local and stationary. A globally varying texture, on the other hand, is only local but not necessarily stationary.

We define globally varying textures as those that satisfy the local but not necessarily the stationary property of Markov Random Fields (Figure 28). The globally varying distribution of texture patterns is often conditioned by certain environment factors, such as rusting over an iron statue conditioned by moisture levels or paint cracks conditioned by paint thickness. When this environment factor is explicitly present, either by measurement or by deduction, we term it the *control map* for the corresponding globally varying texture; see Figure 29 for an example.

Notice that globally varying textures are still local; this is very important as it allows us to characterize globally varying textures by spatial neighborhoods. On the other hand, the



color texture control map

Figure 29: Control map for globally varying textures. In this example, the color texture is a paint crack pattern, and the control map is the original paint thickness.

potential non-stationary patterns of globally varying textures make them more complex to analyze and synthesize. In this section, we describe how to extend the basic algorithms in Section 3 to synthesize globally varying textures.

The single most basic idea that distinguishes globally varying texture synthesis from the basic stationary version is very simple: instead of allowing each input texel to go to any place in the output, use a certain mechanism to control and/or influence who can go to where. The control idea relates to the control maps defined above but the exact mechanism depends on specific algorithms. One of the simplest and earliest control mechanisms is by colors. For example, given an input image containing purple flowers over green grass, [Ash01] synthesizes an output conditioned on a user-painted color map with purple and green colors. This can be achieved by adapting the neighborhood search process in [WL00] so that the best matched input neighborhood is chosen to match not only the output texture colors but also the user supplied map. Similar ideas could also be used for patch-based synthesis [EF01]. [HJO*01] further extended the ideas in [Ash01] by proposing the notion of *image analogies*. Basically, the input to the system consists of three images: the input image, the *filtered* input, and the *filtered* output. The output is then synthesized by analogies where filtered-input-to-original-input is analogous to filtered-output-to-output. The major difference between [HJO*01] and [Ash01] is that image analogies has an additional filtered input, which acts like the control map of the input. The filtered output in [HJO*01] is similar to the target map in [Ash01] even though they are used differently.

[ZZV*03] introduced the idea of *texton mask* to synthesize globally varying output textures from stationary inputs. [MZD05] used a related idea termed *warp field* to seamlessly morph one texture into another. In addition to synthesize globally varying outputs, texton mask has also been found to be an effective mechanism to maintain integrity of synthesized texture elements.

The methods mentioned above [Ash01, EF01, HJO*01] pioneered practice of user controllable texture synthesis from examples, but the control maps embedded in these algorithms are mostly heuristic. Several recent methods continue this line of work but either measure the control maps from real data [GTR*06, LGG*07] or compute the control map via a more principled process [WTL*06]. In addition, all these methods present time-varying texture synthesis.

10. Inverse Texture Synthesis

So far, we have been presenting texture synthesis as a method to produce an arbitrarily large output from a small input exemplar. This is certainly a useful application scenario, as many natural or man-made textures have inherent size limitation, e.g. to obtain an orange skin texture, one is usually constrained to crop a small piece of photograph due to the curvature and associated lighting/geometry variations.

However, with the recent advances in scanning and data acquisition technologies, large textures are becoming more common [GTR*06, LGG*07]. This is particularly true for globally varying textures (Section 9), since to capture their global pattern variance one usually needs sufficiently large coverage. This large data size could cause problems for storage, transmission, and computation.

Inverse texture synthesis [WHZ*08] is a potential solution to this problem. The technique is termed *inverse* texture synthesis because it operates in the inverse direction respect to traditional *forward* texture synthesis (Figure 31). Specifically, the goal of traditional forward texture synthesis is data amplification, where an arbitrarily large output is synthesized from a small input with similar perceptual quality. On the other hand, the goal of inverse texture synthesis is data reduction, where a small output is derived from a large input so that as much information from the input is preserved as possible. Note that even though this information-preservation goal is similar to image compression, for texture synthesis, we evaluate the success of this "compression" process via perceptual quality achieved by forward texture synthesis, not pixel-wise identity.

The goal of inverse texture synthesis can be stated as follows. Given a large input texture, computes a small output compaction so that as much information from the input is preserved as possible. This latter property is evaluated by (forward) synthesizing a new output texture from the computed compaction and see if the output looks similar to the original input. When the input is a globally varying texture, inverse texture synthesis will compute a compaction that contains both the original color texture and control map (Figure 31). The compaction, together with a novel control map, could then be used to synthesize a new texture with user desired intention, such as controlling bronze patination via surface accessibility in Figure 31. Note that, as exemplified in Figure 31, a properly computed compaction would not only preserve quality but also saves computation and storage due to the reduced data size.

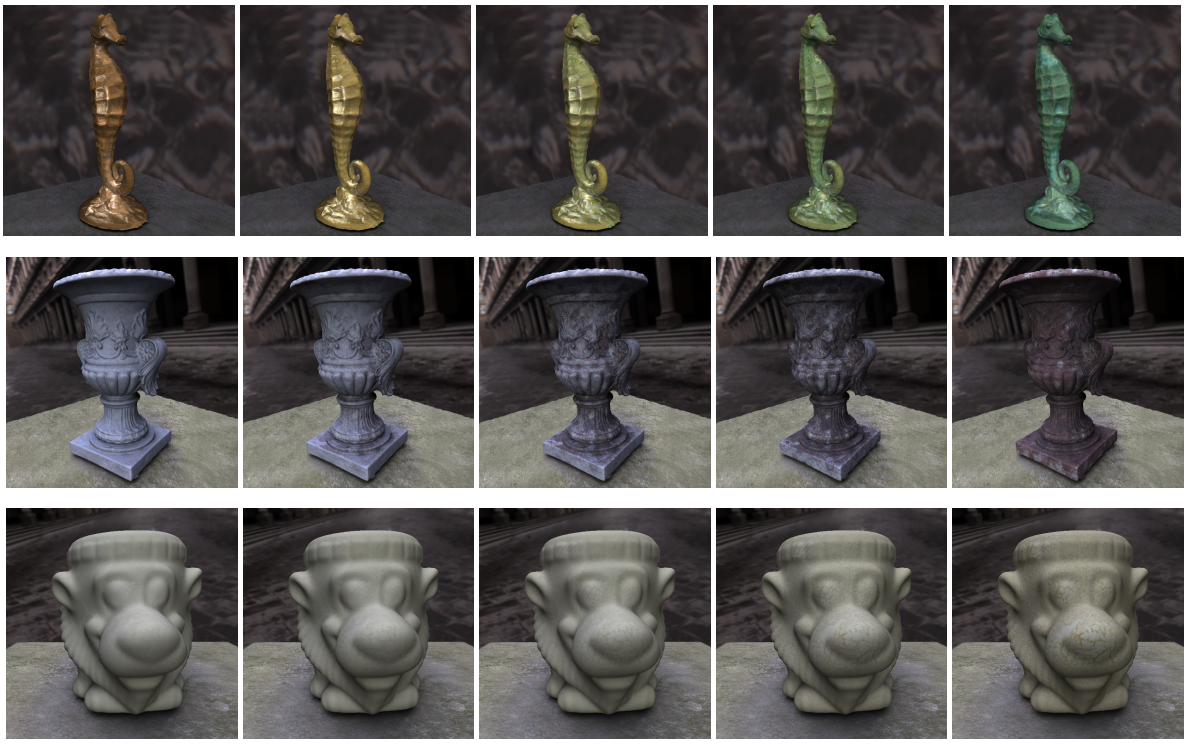


Figure 30: Context aware texture synthesis [LGG*07]. From top to bottom: bronze patination, iron rusting, and paint cracking. Time progresses from left to right.

10.1. Related methods

In addition to be considered as operating in the inverse way with respect to traditional forward texture synthesis, inverse texture synthesis also belongs to a larger category of image summarization algorithms such as [JFK03, KWR07, WWOH08]. In general, one major difference between these methods and inverse texture synthesis is that similar to image compression, they are mainly designed for general images, not just textures. It remains an interesting future work to see if it is possible to apply or extend the neighborhood based method in [WHZ*08] to general images as well.

11. Image Completion by Texture Synthesis

Texture synthesis from example provides a powerful tool to repair holes or replace objects in images. For this reason, it is one of the core components of many image completion techniques (e.g. [BVSO03, DCOY03, HE07]).

The challenges for successful image completion often lie in making the best use of existing texture synthesis algorithms. For instance, preserving structural elements such as windows or fences requires to guide synthesis by reconstructing the outlines of these shapes prior to synthesis [BSCB00, SYJS05]. Some approaches undistort the surfaces seen at an angle before feeding them into a standard patch–based synthesizer [PSK06]. Note that a few texture synthesis algorithms have been specifically tailored to operate along the distorted textures found in images [FH04, ELS08].

Since we focus here on algorithms operating on textures rather than arbitrary images, we do not describe these approaches in detail. We refer the interested reader to the survey on image completion and inpainting by Fidaner [Fid08].

12. Resolution Enhancement by Texture Synthesis

Texture synthesis by example is also used to produce images of arbitrary resolution. A first scenario is to add detail in a low resolution image. This is often referred to as *super–resolution*. Hertzmann et al. [HJO*01] use as example a small part of the image given at high–resolution. The low–resolution image is guiding (see Section 9) a texture synthesis process operating at the higher resolution. Freeman et al. [FJP02] follow a similar approach, copying small patches of high-resolution details from a database of examples.

A second scenario is to generate textures with arbitrary amount of detail, adding sharper and sharper features as the viewpoint is getting closer to the surface. Such an algorithm was recently proposed by Han et al. [HRRG08]. The idea is to use several exemplars describing the texture at various

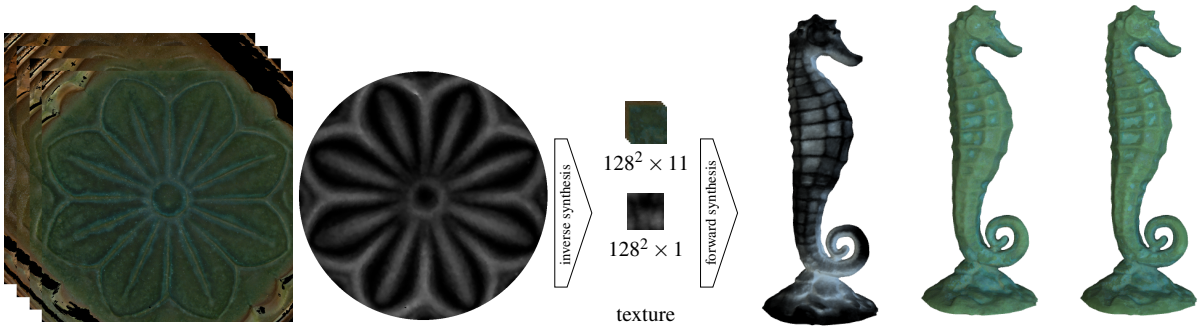


Figure 31: Inverse texture synthesis [WHZ*08]. Given a large globally-varying texture with an auxiliary control map (patina sequence from [LGG*07] in this case), the algorithm automatically computes a small texture compaction that best summarizes the original, including both texture and control. This small texture compaction can be used to reconstruct the original texture from its original control map, or to re-synthesize a new texture under a user-supplied control map. Due to the reduced data size, re-synthesis from our compaction is much faster than from the original without compromising image quality (right two images). In this example we use [KEBK05] for forward synthesis, but other algorithms can also be used since our compactions are just ordinary images.

scales. These exemplars are automatically linked to create an exemplar graph. For instance, a green area in a terrain image will link to an image containing detailed grass. Typically exemplars will have similar resolution (e.g. 64^2 pixels), but will represent very different spatial extents (e.g. from square inches to square miles). During synthesis the synthesizer will automatically 'jump' from one exemplar to the next, effectively behaving as if a unique exemplar of much larger resolution was available. In addition, since the graph may contain loops the multi–scale synthesis process can continue endlessly, producing auto–similar multi–scale content.

Finally, Lefebvre and Hoppe [LH05] achieve high resolution synthesis by first synthesizing using a low resolution version of an exemplar, and then re-interpreting the result for the high–resolution version of the same exemplar. The underlying idea is to only perform synthesis at the scale where some structure is present, while small discontinuities will not be visible on unstructured fine scale detail.

13. Geometry Texture Synthesis

Some researchers have extended the idea of texture synthesis to the creation of geometric details from example geometry. There are a number of different approaches that have been invented for performing geometry synthesis from example, and often the style of algorithm is guided by the choice of geometric representation that is being used, such as polygon mesh, volumetric model, or height field.

Using volumetric models of geometry, Bhat et al. performed geometry synthesis [BIT04] using an approach that was inspired by Image Analogies [HJO*01]. They begin the geometry synthesis process by creating a coordinate frame field throughout the volume of space that is to be modified. Using these coordinate frames, they build 3D voxel neighborhoods, similar to those used for pixel-at-a-time texture synthesis. Using an Image Analogy style of neighborhood matching, they determine the voxel value that is to be copied from the volumetric geometry exemplar. Figure 32 shows an example of this approach.

Several approaches to geometry synthesis have used polygon meshes to represent geometry. The Mesh Quilting work of Zhou et al. places geometric detail (taken from a polygonal *swatch*) over the surface of a given input base mesh [ZHW*06]. This approach is similar to the hole filling texture synthesis approach of Efros and Leung [EL99] in that new geometry is placed on the base mesh in accor-

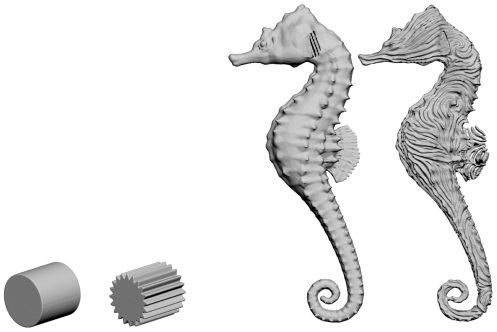


Figure 32: The resulting volumetric geometry (far right) is synthesized over the seahorse mesh (middle right) by analogy with the flat cylinder and the grooved cylinder (at left). From [BIT04].

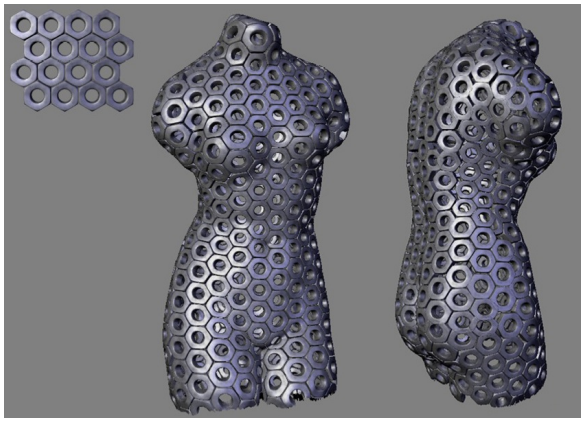


Figure 33: Mesh quilting, from [ZHW*06]. The geometric exemplar (the polygonal swatch) is shown in the upper left.

dance to how well it matches already-placed geometry. Figure 33 demonstrates its results. Merrell used quite a different approach to geometry synthesis from polygonal models [Mer07]. The input to his approach is a polygonal model that has been designed for geometry synthesis, and that is partitioned by the model builder into meaningfull cells, each with a particular label. For example, a stone column might be partitioned into a base, a middle section, and a top cell. Then, the synthesis algorithm creates a 3D collection of labelled cells in such a way that label placement is guided by the adjacency information of labelled cells in the input geometry. This method of geometry synthesis is particularly useful for synthesis of architectural detail.

Zhou et al. create terrain geometry that is assembled from real terrain data (given by a heigh field), but with land forms that are guided by a user's sketch [ZSTR07]. Their approach segments the user's sketch into small curvalinear features

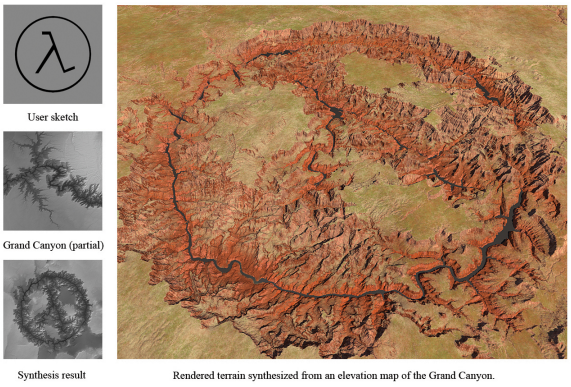


Figure 34: Terrain synthesis (right) based on a user's input sketch (upper left), based on the method of [ZSTR07].

(straight segments, curves, branch points) and then finds matching features in a collection of patches from the input heigh field data. These height field patches are blended together using a combination of 2D warping, Poisson blending and graph cut merging. The patches are large, typically 80×80 height samples, and the speed of this method depends on reducing the terrain features (rivers, mountains, valleys) into simplified curvalinear features. Figure 34 shows a result from this approach.

14. Alternative Methods

So far, we have been focused mainly on neighborhood-based texture synthesis algorithms due to their success in producing high quality results for graphics applications. However, neighborhood-match is not the only possible method for texture synthesis, and there are many notable alternatives out there.

One of the earlier methods for texture synthesis is sampling via Markov Random Fields [Pop97, Pag04]. These methods could be considered as precursors for later neighborhood-search-based methods such as [EL99, WL00], but since they are based on rigorous MRF math framework, they are often much more computationally demanding that neighborhood-based methods. In essence, both [Pop97, Pag04] and [EL99, WL00] attempt to synthesize textures by insuring local neighborhood quality, but [EL99, WL00] provide simpler algorithms and thus run faster. They are also more intuitive and easier to understand, and thus have helped inspire recent advances in texture synthesis research.

Another notable methodology is based on parametric models for human perceptions [HB95, PS00]. Specifically, [HB95] is based on color histograms and [PS00] a more refined model based on matching coefficients for multi-scale oriented filter responses. Being *parametric*, these methods have several main advantages compared to neighborhood-based methods which are often considered to be *non-parametric*. (But an alternative interpretation is that the parameters are the neighborhoods themselves, especially if the input is reduced into a compaction via inverse texture synthesis [WHZ*08].) Being parametric, these methods describe any texture via a few parameters, and is thus very compact, a similar advantage to procedural texturing. The parameter sets also allow potential texture editing operations, e.g. hybrid from multiple textures, by manipulating the proper parameters from the multiple inputs. This is possible because each parameter should (in theory) correspond to certain human perception mechanisms for textures. Most importantly, these methods shed light on the underlying human perception mechanisms for textures, which, if properly understood, might lead to the ultimate texture synthesis algorithms.

Finally, even though most texture synthesis algorithms are designed to handle general situations, there are certain domain-specific algorithms that could produce better results

for the target domain. One recent example is near-regular texture synthesis [LLH04], where the authors treat near-regular textures as statistical deviations from an underlying regular structure. Armed by a more robust model, this method is able to handle tasks considered difficult for general texture models such as changing lighting and geometry structures. The authors also provided a classification of textures based on geometric and appearance regularity.

15. Future Work

Example-based texture synthesis is a fast moving field, and there are a variety of possible directions for future work.

We are particularly motivated to close the gap between procedural and example-based texture synthesis. As shown in Table 2, example-based methods usually consume more memory, are not randomly accessible, are more difficult to edit, and offer only limited resolution. Recently, however, there have been several exciting new techniques that are addressing these issues. We believe that further research in these directions can make example-based texture synthesis even more useful, and our hunch is that doing so will be easier than trying to improve the generality of procedural texturing techniques.

Another promising direction of research is to explore further the link between geometry and texture synthesis. Most current methods apply synthesis to geometry by directly adapting concepts developed for images (patch stitching [ZHW*06] or height-field synthesis [ZSTR07]). However, synthesizing varied 3D shapes from a small set of examples remains a challenging open problem, even though some preliminary work has been done [Mer07, MM08]. We believe this is a great opportunity for further research.

16. Conclusion

Synthesis by example is one of the most promising ideas for providing end-users with powerful content creation tools. Almost anyone can learn to create new results by combining examples. This is an intuitive approach, requiring minimal experience and technical skills. We believe that texture synthesis by example provides key tools and insights towards this goal. Generalizing and combining texture synthesis tools with tools from other fields (geometric modeling, animation) is an exciting and promising challenge. We hope this survey will help to spread these ideas and that it will encourage research efforts directed towards these goals.

	procedural	example-based
generality	×	√
data size	√ compact	[WHZ*08]
random access	√ yes	[WL02, LH05]
editability	√ easy	[BD02, MZD05]
resolution	√ infinite	[HRRG08]

Table 2: Comparison of procedural and example-based texture synthesis. Even though example-based texture synthesis has several shortcomings compared to procedural texture synthesis, such as data size, random accessibility, editability/controllability, and resolution, the gap has been closing by recent advances as cited in the table. The main drawback of procedural texturing, of limited generality, remains a largely unsolved open problem.

References

[Ash01] ASHIKHMIN M.: Synthesizing natural textures. In *I3D '01* (2001), pp. 217–226.

[AZP*05] AGARWALA A., ZHENG K. C., PAL C., AGRAWALA M., COHEN M., CURLESS B., SALESIN D., SZELISKI R.: Panoramic video textures. In *SIGGRAPH '05* (2005), pp. 821–827.

[BD02] BROOKS S., DODGSON N.: Self-similarity based texture editing. In *SIGGRAPH '02* (2002), pp. 653–656.

[BIT04] BHAT P., INGRAM S., TURK G.: Geometric texture synthesis by example. In *SGP '04* (2004).

[BJEYLW01] BAR-JOSEPH Z., EL-YANIV R., LISCHINSKI D., WERMAN M.: Texture mixing and texture movie synthesis using statistical learning. *IEEE Transactions on Visualization and Computer Graphics 7* (2001), 120–135.

[BSCB00] BERTALMIO M., SAPIRO G., CASELLES V., BALLESTER C.: Image inpainting. In *SIGGRAPH '00* (2000), pp. 417–424.

[BSHK04] BHAT K. S., SEITZ S. M., HODGINS J. K., KHOSLA P. K.: Flow-based video synthesis and editing. In *SIGGRAPH '04* (2004), pp. 360–363.

[BSM*06] BARGTEIL A. W., SIN F., MICHAELS J. E., GOKTEKIN T. G., O'BRIEN J. F.: A texture synthesis method for liquid animations. In *SCA '06* (2006), pp. 345–351.

[BVSO03] BERTALMIO M., VESE L., SAPIRO G., OSHER S.: Simultaneous structure and texture image inpainting. *IEEE Transactions on Image Processing 12* (2003), 882–889.

[CSHD03] COHEN M. F., SHADE J., HILLER S., DEUSSEN O.: Wang tiles for image and texture generation. In *SIGGRAPH '03* (2003), pp. 287–294.

[DCOY03] DRORI I., COHEN-OR D., YESHURUN H.: Fragment-based image completion. In *SIGGRAPH '03* (2003), pp. 303–312.

[DCWS03] DORETTO G., CHIUSO A., WU Y., SOATTO S.: Dynamic textures. *International Journal of Computer Vision 51*, 2 (2003), 91–109.

[De 97] DE BONET J. S.: Multiresolution sampling pro-

cedure for analysis and synthesis of texture images. In *SIGGRAPH '97* (1997), pp. 361–368.

[DG01] DISCHLER J.-M., GHAZANFARPOUR D.: A survey of 3d texturing. *Computers & Graphics 25*, 10 (2001).

[DGF98] DISCHLER J. M., GHAZANFARPOUR D., FREYDIER R.: Anisotropic solid texture synthesis using orthogonal 2d views. *Computer Graphics Forum 17*, 3 (1998), 87–96.

[DLTD08] DONG Y., LEFEBVRE S., TONG X., DRETTAKIS G.: Lazy solid texture synthesis. In *EGSR '08* (2008).

[EF01] EFROS A. A., FREEMAN W. T.: Image quilting for texture synthesis and transfer. In *SIGGRAPH '01* (2001), pp. 341–346.

[EL99] EFROS A. A., LEUNG T. K.: Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision* (1999), pp. 1033–1038.

[ELS08] EISENACHER C., LEFEBVRE S., STAMMINGER M.: Texture synthesis from photographs. In *Proceedings of the Eurographics conference* (2008).

[EMP*02] EBERT D. S., MUSGRAVE F. K., PEACHEY D., PERLIN K., WORLEY S.: *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., 2002.

[FH04] FANG H., HART J. C.: Textureshop: texture synthesis as a photograph editing tool. In *SIGGRAPH '04* (2004), pp. 354–359.

[Fid08] FIDANER I. B.: A survey on variational image inpainting, texture synthesis and image completion, 2008. http://www.scribd.com/doc/3012627/.

[FJP02] FREEMAN W. T., JONES T. R., PASZTOR E. C.: Example-based super-resolution. *IEEE Comput. Graph. Appl. 22*, 2 (2002), 56–65.

[FL05] FU C.-W., LEUNG M.-K.: Texture tiling on arbitrary topological surfaces. In *EGSR '05* (2005), pp. 99–104.

[FSDH07] FISHER M., SCHRODER P., DESBRUN M., HOPPE H.: Design of Tangent Vector Fields. *ACM TRANSACTIONS ON GRAPHICS 26*, 3 (2007), 56.

[GD95] GHAZANFARPOUR D., DISCHLER J.-M.: Spectral analysis for automatic 3d texture generation. *Computers & Graphics 19*, 3 (1995).

[GD96] GHAZANFARPOUR D., DISCHLER J.-M.: Generation of 3d texture using multiple 2d models analysis. *Computers & Graphics 15*, 3 (1996).

[GS86] GRÜNBAUM B., SHEPARD G.: *Tilings and Patterns*. 1986.

[GTR*06] GU J., TU C.-I., RAMAMOORTHI R., BELHUMEUR P., MATUSIK W., NAYAR S.: Time-varying surface appearance: acquisition, modeling and rendering. In *SIGGRAPH '06* (2006), pp. 762–771.

[HB95] HEEGER D. J., BERGEN J. R.: Pyramid-based texture analysis/synthesis. In *SIGGRAPH '95* (1995), pp. 229–238.

[HE07] HAYS J., EFROS A. A.: Scene completion using millions of photographs. In *SIGGRAPH '07* (2007), p. 4.

[HJO*01] HERTZMANN A., JACOBS C. E., OLIVER N., CURLESS B., SALESIN D. H.: Image analogies. In *SIGGRAPH '01* (2001), pp. 327–340.

[HRRG08] HAN C., RISSER E., RAMAMOORTHI R., GRINSPUN E.: Multiscale texture synthesis. In *SIGGRAPH '08* (2008), pp. 1–8.

[HZW*06] HAN J., ZHOU K., WEI L.-Y., GONG M., BAO H., ZHANG X., GUO B.: Fast example-based surface texture synthesis via discrete optimization. *Vis. Comput. 22*, 9 (2006), 918–925.

[II96] II K. C.: An aperiodic set of 13 wang tiles, 1996. Discrete Mathematics 160, pp 245-251.

[JDR04] JAGNOW R., DORSEY J., RUSHMEIER H.: Stereological techniques for solid textures. In *SIGGRAPH '04* (2004), pp. 329–335.

[JFK03] JOJIC N., FREY B. J., KANNAN A.: Epitomic analysis of appearance and shape. In *ICCV '03* (2003), p. 34.

[KAK*07] KWATRA V., ADALSTEINSSON D., KIM T., KWATRA N., CARLSON M., LIN M.: Texturing fluids. *IEEE Trans. Visualization and Computer Graphics 13*, 5 (2007), 939–952.

[KCODL06] KOPF J., COHEN-OR D., DEUSSEN O., LISCHINSKI D.: Recursive wang tiles for real-time blue noise. In *SIGGRAPH '06* (2006), pp. 509–518.

[KEBK05] KWATRA V., ESSA I., BOBICK A., KWATRA N.: Texture optimization for example-based synthesis. In *SIGGRAPH '05* (2005), pp. 795–802.

[KFCO*07] KOPF J., FU C.-W., COHEN-OR D., DEUSSEN O., LISCHINSKI D., WONG T.-T.: Solid texture synthesis from 2d exemplars. In *SIGGRAPH '07* (2007), p. 2.

[KSE*03] KWATRA V., SCHÖDL A., ESSA I., TURK G., BOBICK A.: Graphcut textures: image and video synthesis using graph cuts. In *SIGGRAPH '03* (2003), pp. 277–286.

[KWLT07] KWATRA V., WEI L.-Y., LEFEBVRE S., TURK G.: Course 15: Example-based texture synthesis. In *SIGGRAPH '07 courses* (2007).

[KWR07] KANNAN A., WINN J., ROTHER C.: Clustering appearance and shape by learning jigsaws. In *Advances in Neural Information Processing Systems 19*. 2007.

[LD06] LAGAE A., DUTRÉ P.: An alternative for Wang tiles: Colored edges versus colored corners. *ACM Transactions on Graphics 25*, 4 (2006), 1442–1459.

[Lef08] LEFEBVRE S.: *Filtered Tilemaps (in Shader X6)*. Shader X6. 2008, ch. 2, pp. 63–72.

[LGG*07] LU J., GEORGHIADES A., GLASER A., WU H., WEI L.-Y., GUO B.: Context aware texture. *ACM Trans. Graph. 26*, 1 (2007).

[LH05] LEFEBVRE S., HOPPE H.: Parallel controllable texture synthesis. In *SIGGRAPH '05* (2005), pp. 777–786.

[LH06] LEFEBVRE S., HOPPE H.: Appearance-space texture synthesis. In *SIGGRAPH '06* (2006), pp. 541–548.

[LKF*08] LAGAE A., KAPLAN C. S., FU C.-W., OSTROMOUKHOV V., KOPF J., DEUSSEN O.: Tile-based methods for interactive applications. SIGGRAPH '08 Class, August 2008.

[LLH04] LIU Y., LIN W.-C., HAYS J.: Near-regular texture analysis and manipulation. In *SIGGRAPH '04* (2004), pp. 368–376.

[LLX*01] LIANG L., LIU C., XU Y., GUO B., SHUM H.-Y.: Real-time texture synthesis using patch-based sampling. In *ACM Transactions on Graphics* (2001).

[LN03] LEFEBVRE S., NEYRET F.: Pattern based procedural textures. In *I3D '03* (2003), pp. 203–212.

[LWS02] LI Y., WANG T., SHUM H.-Y.: Motion texture: a two-level statistical model for character motion synthesis. In *SIGGRAPH '02* (2002), pp. 465–472.

[Mer07] MERRELL P.: Example-based model synthesis. In *I3D '07* (2007), pp. 105–112.

[MM08] MERRELL P., MANOCHA D.: Continuous model synthesis. In *SIGGRAPH Asia '08* (2008), pp. 1–7.

[MZD05] MATUSIK W., ZWICKER M., DURAND F.: Texture design using a simplicial complex of morphable textures. In *SIGGRAPH '05* (2005), pp. 787–794.

[NC99] NEYRET F., CANI M.-P.: Pattern-based texturing revisited. In *SIGGRAPH '99* (1999), pp. 235–242.

[Ney03] NEYRET F.: Advected textures. In *SCA '03* (july 2003).

[NKL*07] NARAIN R., KWATRA V., LEE H.-P., KIM T., CARLSON M., LIN M.: Feature-guided dynamic texture synthesis on continuous flows. In *EGSR '07* (2007).

[ONOI04] OWADA S., NIELSEN F., OKABE M., IGARASHI T.: Volumetric illustration: Designing 3d models with internal textures. In *SIGGRAPH '04* (2004), pp. 322–328.

[Pag04] PAGET R.: Strong markov random field model. *IEEE Trans. Pattern Anal. Mach. Intell. 26*, 3 (2004), 408–413.

[PFH00] PRAUN E., FINKELSTEIN A., HOPPE H.: Lapped textures. In *SIGGRAPH '00* (2000), pp. 465–470.

[POB*07] PIETRONI N., OTADUY M. A., BICKEL B., GANOVELLI F., GROSS M.: Texturing internal surfaces from a few cross sections. *Computer Graphics Forum 26*, 3 (2007).

[Pop97] POPAT A. C.: *Conjoint Probabilistic Subband Modeling*. PhD thesis, Massachusetts Institute of Technology, 1997.

[PS00] PORTILLA J., SIMONCELLI E. P.: A parametric texture model based on joint statistics of complex wavelet coefficients. *Int. J. Comput. Vision 40*, 1 (2000), 49–70.

[PSK06] PAVIĆ D., SCHÖNEFELD V., KOBBELT L.: Interactive image completion with perspective correction. *Vis. Comput. 22*, 9 (2006), 671–681.

[QY07] QIN X., YANG Y.-H.: Aura 3d textures. *IEEE Transactions on Visualization and Computer Graphics 13*, 2 (2007), 379–389.

[RVLL08] RAY N., VALLET B., LI W., LEVY B.: Article N-Symmetry Direction Field Design. *ACM Transactions on Graphics-TOG 27*, 2 (2008).

[SCA02] SOLER C., CANI M.-P., ANGELIDIS A.: Hierarchical pattern mapping. In *SIGGRAPH '02* (2002), pp. 673–680.

[SE02] SCHÖDL A., ESSA I. A.: Controlled animation of video sprites. In *SCA '02* (2002), pp. 121–127.

[SP96] SZUMMER M., PICARD R. W.: Temporal texture modeling. In *In IEEE International Conference on Image Processing* (1996), pp. 823–826.

[SSSE00] SCHÖDL A., SZELISKI R., SALESIN D. H., ESSA I.: Video textures. In *SIGGRAPH '00* (2000), pp. 489–498.

[Sta97] STAM J.: *Aperiodic Texture Mapping*. Tech. Rep. R046, European Research Consortium for Informatics and Mathematics (ERCIM), Jan. 1997.

[SYJS05] SUN J., YUAN L., JIA J., SHUM H.-Y.: Image completion with structure propagation. In *SIGGRAPH '05* (2005), pp. 861–868.

[THCM04] TARINI M., HORMANN K., CIGNONI P., MONTANI C.: Polycube-maps. In *SIGGRAPH '04* (2004), pp. 853–860.

[TOII08] TAKAYAMA K., OKABE M., IJIRI T., IGARASHI T.: Lapped solid textures: filling a model with anisotropic textures. In *SIGGRAPH '08* (2008), pp. 1–9.

[Tur91] TURK G.: Generating textures on arbitrary surfaces using reaction-diffusion. In *SIGGRAPH '91* (1991), pp. 289–298.

[Tur01] TURK G.: Texture synthesis on surfaces. In *SIGGRAPH '01* (2001), pp. 347–354.

[TZL*02] TONG X., ZHANG J., LIU L., WANG X., GUO B., SHUM H.-Y.: Synthesis of bidirectional texture functions on arbitrary surfaces. In *SIGGRAPH '02* (2002), pp. 665–672.

[vW02] VAN WIJK J. J.: Image based flow visualization. In *SIGGRAPH '02* (2002), pp. 745–754.

[Wei02] WEI L.-Y.: *Texture synthesis by fixed neighborhood searching*. PhD thesis, Stanford University, 2002. Adviser-Marc Levoy.

[Wei04] WEI L.-Y.: Tile-based texture mapping on graphics hardware. In *HWWS '04* (2004), pp. 55–63.

[WHZ*08] WEI L.-Y., HAN J., ZHOU K., BAO H., GUO B., SHUM H.-Y.: Inverse texture synthesis. In *SIGGRAPH '08* (2008), pp. 1–9.

[WL00] WEI L.-Y., LEVOY M.: Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH '00* (2000), pp. 479–488.

[WL01] WEI L.-Y., LEVOY M.: Texture synthesis over arbitrary manifold surfaces. In *SIGGRAPH '01* (2001), pp. 355–360.

[WL02] WEI L.-Y., LEVOY M.: *Order-Independent Texture Synthesis*. Tech. Rep. TR-2002-01, Computer Science Department, Stanford University, 2002.

[WTL*06] WANG J., TONG X., LIN S., PAN M., WANG C., BAO H., GUO B., SHUM H.-Y.: Appearance manifolds for modeling time-variant appearance of materials. In *SIGGRAPH '06* (2006), pp. 754–761.

[WWOH08] WANG H., WEXLER Y., OFEK E., HOPPE H.: Factoring repeated content within and among images. In *SIGGRAPH '08* (2008), pp. 1–10.

[WY04] WU Q., YU Y.: Feature matching and deformation for texture synthesis. In *SIGGRAPH '04* (2004), pp. 364–367.

[YHBZ01] YING L., HERTZMANN A., BIERMANN H., ZORIN D.: Texture and Shape Synthesis on Surfaces. In *EGSR '02* (2001).

[YNBH09] YU Q., NEYRET F., BRUNETON E., HOLZSCHUCH N.: Scalable real-time animation of rivers. *Computer Graphics Forum (Proceedings of Eurographics 2009) 28*, 2 (mar 2009).

[ZG03] ZELINKA S., GARLAND M.: Interactive texture synthesis on surfaces using jump maps. In *EGSR '03* (2003), pp. 90–96.

[ZG04] ZELINKA S., GARLAND M.: Jump map-based interactive texture synthesis. *ACM Trans. Graph. 23*, 4 (2004), 930–962.

[ZHW*06] ZHOU K., HUANG X., WANG X., TONG Y., DESBRUN M., GUO B., SHUM H.-Y.: Mesh quilting for geometric texture synthesis. In *SIGGRAPH '06* (2006), pp. 690–697.

[ZMT06] ZHANG E., MISCHAIKOW K., TURK G.: Vector field design on surfaces. *ACM Transactions on Graphics 25*, 4 (2006), 1294–1326.

[ZSTR07] ZHOU H., SUN J., TURK G., REHG J. M.: Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics 13*, 4 (2007), 834–848.

[ZZV*03] ZHANG J., ZHOU K., VELHO L., GUO B., SHUM H.-Y.: Synthesis of progressively-variant textures on arbitrary surfaces. In *SIGGRAPH '03* (2003), pp. 295–302.