Thirteenth Eurographics Workshop on Rendering (2002)
P. Debevec and S. Gibson (Editors)

Towards Real-Time Texture Synthesis with the Jump Map

Steve Zelinka and Michael Garland

Department of Computer Science, University of Illinois at Urbana–Champaign, Champaign, Illinois

Abstract

While texture synthesis has been well-studied in recent years, real-time techniques remain elusive. To help facilitate real-time texture synthesis, we divide the task of texture synthesis into two phases: a relatively slow analysis phase, and a real-time synthesis phase. Any particular texture need only be analyzed once, and then an unlimited amount of texture may be synthesized in real-time. Our analysis phase generates a jump map, which stores for each input pixel a set of matching input pixels (jumps). Texture synthesis proceeds in real-time as a random walk through the jump map. Each new pixel is synthesized by extending the patch of input texture from which one of its neighbours was copied. Occasionally, a jump is taken through the jump map to begin a new patch. Despite the method's extreme simplicity, its speed and output quality compares favourably with recent patch-based algorithms.

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

1. Introduction

Increasing realism continues to be a primary goal of computer graphics research. Highly accurate 3D scanners allow the acquisition of extremely detailed geometry, which can help immensely in the visual fidelity of computer graphics. Texture maps augment geometry by providing high frequency details which may be expensive to represent directly using geometry. Using texture maps can vastly reduce the amount of geometry necessary to realistically render a scene. However, creating or acquiring texture maps remains a very labor-intensive affair and the subject of much research. Procedural methods, which involve complex modelling and approximation of natural phenomena, are widely used and often generate good results. However, many real-world textures are difficult or expensive to model procedurally. Image-based rendering, where real-world samples are captured and re-used to create novel imagery, provides a solution for many of these textures. *Texture synthesis* methods allow one to synthesize the requisite amount of texture using only a small sample of the desired texture. Much recent work has focussed on the texture synthesis problem[4, 11, 1, 3], and very high quality results have been produced.

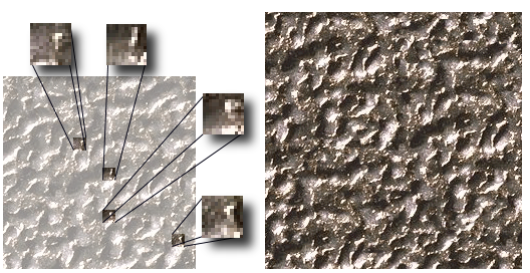

Figure 1: *Jump map links. Left: Input texture (faded for clarity). The jump map records links between closely-matched neighbourhoods, such as those indicated. Right:* 256×256 *image synthesized in scanline order in 0.03 seconds by a random walk through the jump map.*

Despite this, texture synthesis algorithms remain relatively slow, with running times measured in seconds for even the fastest high-quality results. Interactive systems require faster solutions; ideally, new texture could be generated on demand. This could also provide a form of texture compression: a large texture need never be stored if it can be regenerated from a small sample on the fly. Such compression is especially important for bandwidth-limited applications, such as those operating over the Internet.

In this paper, we divide the problem of texture synthesis into two phases, an *analysis phase*, detailed in Section 2, and a *synthesis phase*, covered in Section 3. For any particular texture, the analysis phase need only be performed once; its

output may be re-used for any number of synthesis tasks. Thus, it is possible to build up a library of analyzed textures and perform real-time synthesis from any of them as desired.

The analysis phase examines the input texture at length, building a data structure we call a *jump map*. The jump map builds upon ideas from Video Textures[10], which are endless streams of video generated from a finite-length sample video, that do not perceptibly repeat or cycle. A video texture is created by linking together pairs of sufficiently similar frames within the sample video. Endless video is synthesized by playing through the sample video while randomly following links. The method works very well for repetitive, unpredictable motion, such as flames or flag-waving.

The jump map generalizes video textures from one temporal dimension to two spatial dimensions. Each pixel in the jump map corresponds to a pixel of the input sample, and has an associated set of jumps to other similar pixels of the input sample. At synthesis time, successive pixels are copied from the input sample, until the decision to take a jump is made. The particular jump is selected randomly from the jump map, with the probability of choosing a particular jump weighted according to the similarity between the input image neighbourhoods on either end of the jump. After a jump, synthesis continues by copying successive pixels from the new region of the input texture. Since no neighbourhood comparisons need be done at run-time, the method is extremely fast, making it suitable for interactive applications.

2. Texture Analysis

The purpose of the analysis phase is to build a data structure suitable for performing synthesis in real-time. Previous texture synthesis methods spend the majority of their time comparing pixel neighbourhoods, and thus several attempts have been made to reduce the number of comparisons required[11, 1]. In contrast, our approach is to attempt to precompute the neighbourhood comparisons that will be required for synthesis during the analysis phase, and avoid making any neighbourhood comparisons at synthesis time.

Recent work[3] indicates relatively large patches of texture can be copied into the output, so long as the boundaries of the patches aren't too regular and match sufficiently, as the human visual system has a hard time picking out such boundaries. Thus, an underlying assumption of our method is that the current output neighbourhood is likely to closely resemble the input neighbourhood(s) from which surrounding pixels were copied. This implies that the set of neighbourhood comparisons performed by typical non-parametric sampling methods[4, 11, 1] may be approximated by comparisons *between input neighbourhoods*.

Thus, given an input sample of size $m \times n$ with c colour channels, the analysis phase discovers for each pixel of the input image, a set of k similar pixels within the input. The k similarity values d_i and pixel locations j_i are stored for each pixel to form the *jump map*. Figure 1 shows the links stored in the jump map for a particular sample pixel. Similarity values are computed using standard image comparison metrics. For simplicity, we use the L_2 norm over a small rectangular neighbourhood around each pixel.

For each pixel, we assign a probability to each of its jumps j_i based on the similarity:

$$p_i = \frac{t - d_i}{kt} \tag{1}$$

where t is a similarity threshold (typically $t = \alpha D$, where D is the maximum d_i over the jump map). Note that no jump may be selected with probability:

$$p_{none} = 1 - \sum_{i=0}^{k} p_i \tag{2}$$

This reflects the fact that only relatively good jumps ought to be taken during synthesis; if all of a pixel's jumps are bad, we should not jump from that pixel. As α is increased, greater lenience is allowed. Our results use $\alpha = 1.1$.

To increase efficiency in jump selection, we typically transform the jump map into a summed distribution table:

$$p'_i = \sum_{j=0}^{i} p_j \tag{3}$$

A random number $0 \leq r \leq 1$ selects the jump j_i, such that $p'_i >= r$ and i is minimal.

2.1. Analysis Optimization

Texture analysis must discover, for each pixel of the input, a set of pixels whose neighbourhoods closely match the given pixel's neighbourhood. Brute-force analysis can take hours to analyze samples on the order of 256×256 pixels. While this may be acceptable in some situations, since it need only be performed once per sample, faster analysis is necessary.

We find similar neighbourhoods for a given pixel by solving a high-dimensional approximate nearest-neighbours (ANN) problem[11, 7]. Each $s \times t$ input neighbourhood is transformed into a stc-dimensional vector, each c-tuple of which corresponds to a pixel in the input neighbourhood. ANN search with a kd-tree[8] is applied to these vectors to find matching neighbourhoods. Typically, an ANN epsilon of 10% or more generates adequate matches for the jump map. The speed of the ANN search critically depends on the input vector dimension, so we apply principal component analysis[6] to the neighbourhood set to reduce its dimension. We have typically observed a dimensional reduction of 50-95%, while retaining over 97% of the original variance.

2.2. Ensuring Diversity

A key failing with the L_2 norm is that the matched neighbourhoods for a particular pixel P may be clustered spatially

```
Analyze(Input: sampleImage, Output: jumpMap)
    inPyramid = GenerateGaussianPyramid(sampleImage);
    nbhds = GenerateNeighbourhoodVectors(inPyramid);
    annNbhds = PCAReduce(nbhds);
    annTree = GenerateANNTree(annNbhds);
    foreach pixel in sampleImage
        vec = GenerateNeighbourhoodVector(pixel);
        matches = FindBestMatches(annTree, vec);
        accepted = PoissonDiscFilter(matches, pixel);
        SetJumpMapLinks(jumpMap, pixel, accepted);
    NormalizeJumpMap(jumpMap);
```

Figure 2: *Analysis phase pseudo-code.*

within the image, often close to P itself. Jumps from a particular portion of the image will then usually lead to another particular portion of the image. This lack of jump map diversity can lead to overly repetitive synthesis results. However, this is easily overcome by Poisson disc sampling. We find βk matching neighbourhoods, and from these iteratively accept up to k matches which are a minimum distance in image-space from each already-accepted match and P. We have found $\beta = 5$ sufficient to ensure a good level of diversity within the jump map.

2.3. Multi-resolution Analysis

Similar to previous efforts[2, 11, 7], we have adopted a multi-resolution approach to analyzing textures. The primary benefit to using multi-resolution neighbourhoods for comparison purposes is that a smaller overall neighbourhood is required to produce good results. For example, a texture requiring a 9×9 single resolution neighbourhood may by adequately served by a $5\times5, 3\times3, 1\times1$ multi-resolution neighbourhood. This reduces the dimension of the vectors (i.e., $(25+9+1)c$ versus $81c$), resulting in faster ANN searching. Multi-resolution neighbourhood vectors also tend to allow greater dimensional reduction from PCA, futher compounding their efficiency. We typically use Gaussian image pyramids, where coarser levels are averaged from finer levels, having observed little difference using Laplacian pyramids (where finer levels are delta-encoded from coarser levels).

The complete analysis phase is summarized as pseudo-code in Figure 2.

3. Texture Synthesis

Similar to Video Textures, synthesizing texture amounts to simply walking randomly through the jump map. The extra dimension requires some extra care, however. For illustration, we assume a scanline synthesis order.

Each pixel is synthesized iteratively as follows. First, a random choice decides from which neighbour to continue copying: in scanline order, the only choices are from the above neighbour, a_o, or the left neighbour, l_o. Suppose we choose a_o, and a_i is the input pixel from which a_o was synthesized. The jump map entry for a_i is consulted, and a random choice determines which jump to take, if any; suppose b_i is the destination of the jump (note that $b_i = a_i$ when no jump is made). The next output pixel is then copied from *below* b_i, since we earlier chose to continue the patch from the *above* neighbour. If a different synthesis order is used, there may be a different set of available neighbour choices (only already-synthesized neighbours may be chosen), but the algorithm otherwise remains the same. In any case, it is essential that the choice between neighbours be random, in order to improve spatial coherence. For example, if one always continued from the left, there is no guarantee that the positions of the jumps on one scanline would be anywhere near the jumps on the next scanline. Choosing from neighbours in different directions allows patches to form in those directions.

Small patch sizes allow greater deviation from the sample image, and thus less obvious repetition, while larger patches allow greater local coherence, and thus less noisy output. To control the size of the output patches formed, we scale the random numbers used to select jumps. Since the jump probabilities for each pixel are normalized to sum to at most 1, a range of $[0,20]$, for example, has the effect of jumping an average of every 20 (or more) pixels.

3.1. Patch Boundaries

The synthesis procedure described so far may produce texture with noticeable discontinuities at patch boundaries. We use three methods to alleviate patch boundary artifacts. First, horizontal or vertical patch boundaries are easily noticed by the human visual system, and form naturally when the sample patch from which pixels are being copied hits an input image boundary. Following Ashikhmin[1], we increase the jump probability as the source patch nears an input image boundary. Secondly, jumps leading near order-dependent edges have their probabilities reduced. For example, for scanline order, jumps leading near the bottom and right edges should be avoided, since large patches cannot form after such jumps, resulting in more artifacts. Finally, blending may be used to further obscure patch boundaries. Typically, we copy and blend a small gaussian patch of pixels (3×3 or 5×5) every time we synthesize a pixel, rather than copying a single pixel.

3.2. Synthesis Order

The order in which pixels are synthesized greatly affects the shape of the texture patches, since it determines the set of neighbours from which to choose (only already synthesized neighbours may be chosen). Since scan-line synthesis order only allows neighbours to the left and above, patches extend down and right, and thus tend to be diagonal in shape. It is possible for patches to extend up and right, but unlikely,

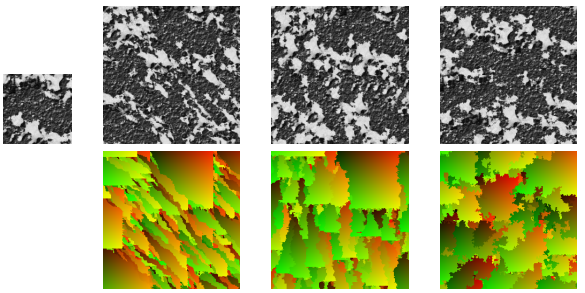

Figure 3: *Synthesis order comparison. Texture patch shape is highly dependent on the order in which pixels are synthesized. Left to right: input, scanline, serpentine, Hilbert. Bottom images show patch structure (colour corresponds to position in input from which pixel was copied).*

```
Synthesize(Input: jumpMap, Output: outImage)
    ordering = Scanline, Hilbert, etc
    while outPixel = Next(ordering)
        neighbours = SynthesizedNeighbours(outPixel);
        outSource = SelectRandom(neighbours);
        inSource = SynthesisSource(outSource);
        if ShouldJump(inSource, offset)
            jumps = JumpMapPixel(jumpMap, inSource);
            range = JumpProbabilityRange(inSource);
            inSource = SelectJump(jumps, range);
        offset = Location(outPixel) - Location(outSource);
        inSource += offset;
        BlendPatch(outImage, outPixel, inSource);
```

Figure 4: *Synthesis phase pseudo-code.*

since early pixels on one scanline would need to jump to the same area as later pixels on earlier scanlines.

A simple change, reversing horizontal direction on every other scanline, allows right neighbours to be chosen occasionally, and forms more regularly-shaped patches. This "serpentine" order results in patches very similar to those produced by Ashikhmin[1], with no loss in efficiency.

If lower efficiency is tolerable, we have found the highest quality results can be produced by synthesizing pixels in order along a space-filling curve[9]. The Hilbert curve, in particular, changes direction frequently, allowing choice of each of the four immediate neighbours with roughly equal frequency. This results in particularly well-shaped patches, as shown in Figure 3. The overhead involved in following such curves slows synthesis by a factor of two to three.

The complete synthesis phase is summarized as pseudo-code in Figure 4.

4. Results

We show texture synthesis results for a variety of samples in Figures 5–8. Sample images are 64×64, 128×128 or 192×192, and results are 256×256. For all images, we use

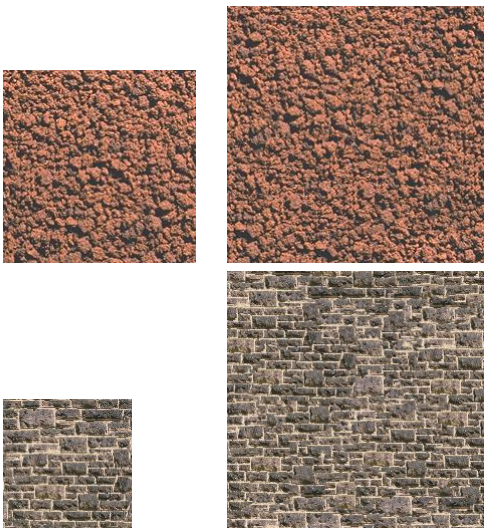

Figure 5: *Texture synthesis results.*

a multi-resolution neighbourhood of $9\times9, 5\times5, 3\times3$, and store at most 3 jumps per pixel. For this neighbourhood size, analysis typically takes from a few seconds to a minute, depending on sample size, on our 1 Ghz Pentium III test machine. Given the $n\times n$ input texture, we typically use a jump frequency equal to $1n$ to $1.5n$, adjusted near the input image edges (within $0.2n$ pixels) by linearly decreasing it to 1 at the actual boundary. Synthesis time is independent of the input image and jump map, and our unoptimized implementation can synthesize about 2.2 million pixels per second (a 256×256 output image takes about 0.03 seconds) in scanline synthesis order, or about 800 thousand pixels per second following a Hilbert curve. All images in this paper, unless otherwise noted, were produced following a Hilbert curve.

Our results are best on stochastic textures (Figure 5), while structured textures (Figure 6) present the most trouble (a greater variety of results may be seen in Figure 8). This is somewhat to be expected, since our method can be seen as an approximation to Ashikhmin's method[1] which is designed for natural textures. However, unlike several recent algorithms, our method will never "fall off" the end of the texture and synthesize garbage; at worst, patch boundaries become noticeable. Theoretically, it is possible for cycling to occur within a small portion of the jump map (an issue requiring non-trivial effort to overcome with Video Textures). However, we have never observed this in practice, likely because our jump maps are much more "dense", having several links per pixel, which allows greater flexibility.

A comparison with Image Quilting[3] and Ashikhmin's method is shown in Figure 7. As can be expected, our method is comparable in quality on stochastic textures, but not quite as good as Image Quilting on structured textures.

Figure 6: *Low frequency content presents more difficulty for the jump map, but the results can often be acceptable.*

5. Related Work

A great body of research in texture synthesis has been published in recent years. For brevity, we shall only review recent methods closely related to ours. The interested reader is referred to the excellent summary in the paper of Hertzmann *et al.* [5] for a broader survey of the field.

As discussed above, our method is a generalization of Video Textures[10]. Our work also derives inspiration from Ashikhmin's method[1]. His method is an optimization of neighbourhood-based texture synthesis[11, 4] which does not use a search data structure. Instead, Ashikhmin recognized that a full neighbourhood search is likely to produce pixels close (within the input image) to previous pixels. Thus, the full search is approximated by checking only those pixel neighbourhoods which continue the source patch from which neighbouring pixels were generated, resulting in a constant number of neighbourhood comparisons per pixel. We take this reasoning one step further and compare input neighbourhoods *only* during the analysis phase. This allows us to avoid neighbourhood comparisons altogether at synthesis time, resulting in orders of magnitude speed-up with comparable image quality.

Image Quilting[3], like our method, is simple to understand and implement, overlapping regularly-sized patches into the output texture while trying to find the best set of border pixels through each overlap region. We effectively work in reverse, discovering which sets of pixels would form good borders with each other, and then implicitly creating patches through our fast randomized synthesis procedure. While our synthesis speed is orders of magnitude faster, Image Quilting produces higher quality results on structured textures.

The Chaos Mosaic[12] performs texture synthesis by pasting and blending random blocks of texture on a tiling of the input. Since the blocks are randomized according to a deterministic chaos transformation, it is possible to sample from the synthesized texture without actually generating it completely. While the results are of relatively low quality, the speed of synthesis is comparable to our method, and the determinism is useful in a variety of contexts.

A recent regular patch-based generalization of Wei and Levoy's algorithm compares patch boundaries for the best match[7]. Using several acceleration data structures, their method synthesizes texture in real-time, with performance comparable to our method (they synthesize 200×200 textures in 0.02 seconds, with roughly similar quality). However, these acceleration data structures add significant complexity and memory overhead to the algorithm.

6. Conclusions and Future Work

In summary, we have presented a viable real-time texture synthesis algorithm, by dividing the problem into that of analyzing the input texture, and that of synthesizing new texture. We solve the analysis problem offline, generating the jump map, and use the jump map to solve synthesis problems in real-time. Our synthesis procedure performs comparably to the fastest known texture synthesis algorithms, while remaining exceedingly simple to understand and implement.

A number of avenues for future work remain under the jump map architecture. One question we have not yet explored is the degree to which the jump map may be compressed. It should also be relatively straightforward to extend the jump map to temporal textures, potentially using some combination of jump maps and the original Video Textures.

The analysis procedure we have selected, while sufficient, is neither fast nor extremely accurate. We believe some perceptual metrics may be helpful in rapidly identifying similar neighbourhoods within the input image with better accuracy than the L_2 norm, which is well-known to be a bad estimate of perceptual similarity. The analysis phase may be subject to further acceleration using clustering techniques; we currently do not exploit the fact that the set of nearest-neighbour queries we wish to perform is identical to the input point set for the search data structure. It would also be interesting to investigate the performance of using a multi-pass procedure on today's graphics hardware to assist the analysis phase.

The synthesis procedure we use may also be significantly improved. First, copying patches rather than single pixels may yield performance improvements, as less logic need be performed per pixel. A multi-resolution synthesis procedure would likely enhance the image quality on more structured textures. A mechanism for "seeding" the output needs development, to allow a relatively simple hardware implementation and the possibility of arbitrarily sampling the output without completely generating it, similar to the Chaos

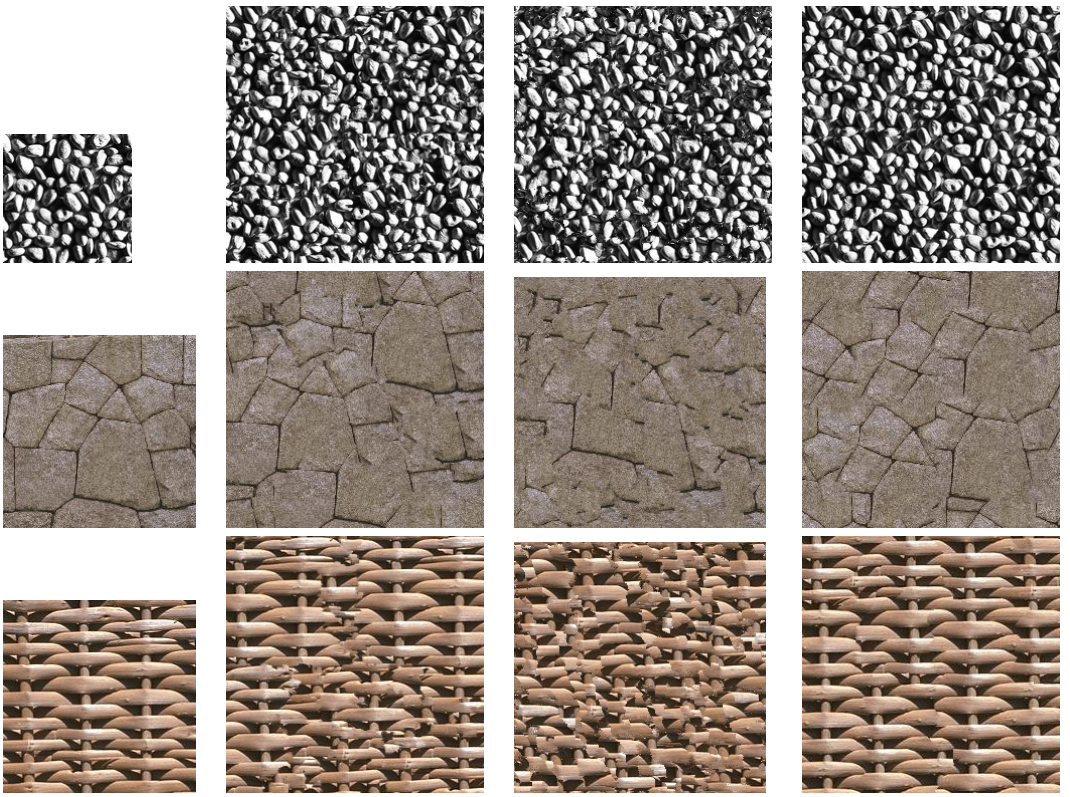

Figure 7: *Quality comparison between our method (left), Ashikhmin's method (middle), and Image Quilting (right).*

Mosaic[12]. Similarly, mechanisms for controlling the output, similar to Ashikhmin's extensions[1], or the texture-by-numbers approach of Image Analogies[5], would be useful.

Acknowledgements

We would like to thank the anonymous reviewers for all of their helpful comments. Textures were acquired from Li-Yi Wei's webpage, and Image Quilting comparison images were taken from Alexei A. Efros's webpage. Code from Michael Ashikhmin's webpage was used to generate comparison images with his algorithm. This research was supported under a grant from the NSF (CCR-0086084).

References

1. ASHIKHMIN, M. Synthesizing natural textures. In *Proceedings of 2001 ACM Symposium on Interactive 3D Graphics* (March 2001), pp. 217–226. 1, 2, 3, 4, 5, 6

2. DE BONET, J. S. Multiresolution sampling procedure for analysis and synthesis of texture images. In *SIGGRAPH 97 Proc.* (Aug. 1997), ACM SIGGRAPH, pp. 361–368. 3

3. EFROS, A. A., AND FREEMAN, W. T. Image quilting for texture synthesis and transfer. In *Proceedings of SIGGRAPH 2001* (August 2001), pp. 341–346. 1, 2, 4, 5

4. EFROS, A. A., AND LEUNG, T. K. Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision* (Corfu, Greece, September 1999), pp. 1033–1038. 1, 2, 5

5. HERTZMANN, A., JACOBS, C. E., OLIVER, N., CURLESS, B., AND SALESIN, D. H. Image analogies. In *Proceedings of SIGGRAPH 2001* (August 2001), pp. 327–340. 5, 6

6. JOLLIFE, I. T. *Principal Component Analysis*. Springer-Verlag, New York, 1986. 2

7. LIANG, L., LIU, C., XU, Y.-Q., GUO, B., AND SHUM, H.-Y. Real-time texture synthesis by patch-based sampling. *ACM Transactions on Graphics (TOG) 20*, 3 (2001), 127–150. 2, 3, 5

8. MOUNT, D. M. Ann programming manual. Tech. rep., Department of Computer Science, University of Maryland, College Park, Maryland, 1998. 2

9. SAGAN, H. *Space-Filling Curves*. Springer-Verlag, New York, 1994. 4

10. SCHÖDL, A., SZELISKI, R., SALESIN, D., AND ESSA, I. Video textures. In *Proceedings of SIGGRAPH 2000* (July 2000), pp. 489–498. 2, 5

11. WEI, L.-Y., AND LEVOY, M. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of SIGGRAPH 2000* (July 2000), pp. 479–488. 1, 2, 3, 5

12. XU, Y.-Q., GUO, B., AND SHUM, H. Chaos mosaic: Fast and memory efficient texture synthesis. Tech. Rep. MSR-TR-2000-32, Microsoft Research, April 2000. 5, 6

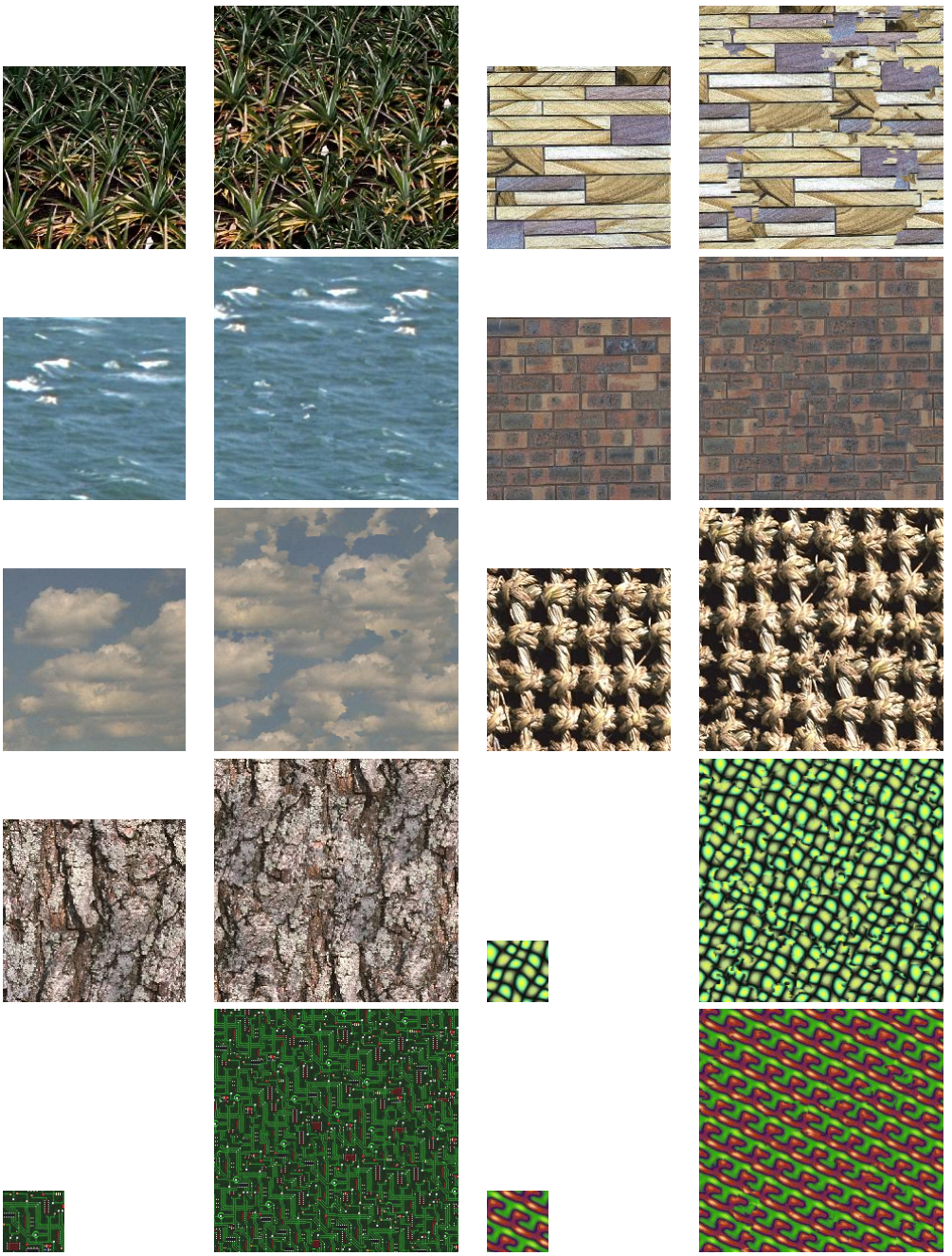

Figure 8: *Jump map texture synthesis results.*