Order-Independent Texture Synthesis

Category: Research

Abstract

Search-based texture synthesis algorithms are sensitive to the order in which texture samples are generated; different synthesis orders yield different textures. Unfortunately, most polygon rasterizers and ray tracers do not guarantee the order with which surfaces are sampled. To circumvent this problem, textures are synthesized beforehand at some maximum resolution and rendered using texture mapping.

We describe a search-based texture synthesis algorithm in which samples can be generated in arbitrary order, yet the resulting texture remains identical. The key to our algorithm is a pyramidal representation in which each texture sample depends only on a fixed number of neighboring samples at each level of the pyramid. The bottom (coarsest) level of the pyramid consists of a noise image, which is small and predetermined. When a sample is requested by the renderer, all samples on which it depends are generated at once. Using this approach, samples can be generated in any order. To make the algorithm efficient, we propose storing texture samples and their dependents in a pyramidal cache. Although the first few samples are expensive to generate, there is substantial reuse, so subsequent samples cost less. Fortunately, most rendering algorithms exhibit good coherence, so cache reuse is high.

Keywords: Texture Synthesis, Texture Mapping, Graphics Hardware

1 Introduction

Textures are employed to represent surface details in synthetic scenes without adding geometric complexity. A common way to acquire textures is by texture synthesis. Existing texture synthesis techniques can be classified as either *implicit* or *explicit* [Ebert et al. 1998, Chapter 2]; an implicit method computes only required texels on demand while an explicit method always computes the entire texture. For example, Perlin noise is an implicit method while histogram matching [Heeger and Bergen 1995] is an explicit method. Implicit methods are usually more flexible and storage efficient than explicit ones since they compute only needed texels on the fly without storing the entire texture. On the other hand, explicit methods are often more general than implicit ones since they can model many different textures based on image statistics. In addition, explicit methods are easier to use; the user only needs to provide an example texture rather than writing and tuning a procedural shader.

It is desirable to have an algorithm that combines the advantages of both implicit and explicit methods. We present a new algorithm, termed *order-independent* texture synthesis, that achieves this combination. The algorithm is as general as existing explicit methods in that it can synthesize new textures simply from given examples. It is as flexible as implicit methods in that it allows textures to be evaluated on demand in any traversal order. Specifically, the algorithm allows *view-dependent* evaluation where only needed texels are synthesized on the fly, while at the same time remains *order-independent*, where the same texel always receives the same synthesized values regardless of which texels are computed and the relative traversal orders. This hybrid algorithm has numerous potential applications. For example, it can be invoked like a procedural texture shader in a ray tracing software without writing different shaders for different textures; it can be used for interactive image texture editing; it can also be implemented in graphics hardware and function like a texture mipmap for polygonal rasterization without storing the entire texture.

2 Previous Work

Explicit Synthesis. Many explicit synthesis algorithms model textures as a set of statistical features, and generate new textures by matching the statistics between the new texture and a given input sample [Heeger and Bergen 1995; Bonet 1997; Efros and Leung 1999; Wei and Levoy 2000]. These algorithms are general and can generate many different textures based on input samples. However, because these methods impose statistical dependencies between texture samples, it is impossible to evaluate only a subset of the samples while guaranteeing that these evaluated samples remain identical with respect to different synthesis orders. For example, histograms [Heeger and Bergen 1995] need to be computed from all texture samples, and pixel-neighborhood-searching [Efros and Leung 1999; Wei and Levoy 2000; Ashikhmin 2001; Tong et al. 2002] will produce different results if adjacent samples are synthesized in different orders.

Another methodology is to generate textures patch by patch [Efros and Freeman 2001; Soler et al. 2002] rather than pixel by pixel. These algorithms preserve large scale structures better than pixel-neighborhood-searching techniques. However, they usually require pre-computation of the entire patch locations or texture coordinates, regardless of the number of actual texels requested.

Implicit Synthesis. Existing implicit synthesis algorithms simulate the texture formation process using specialized procedures [Ebert et al. 1998]. These techniques can be highly efficient and they allow texels to be evaluated independently from each other. However, since different textures may require different procedures, these algorithms are less general than explicit methods. In addition it can be difficult to tune the parameters for those procedures to achieve the desired visual appearance of the result texture.

3 Algorithm

Our algorithm extends previous multi-resolution neighborhood-search texture synthesis algorithms [Wei and Levoy 2000; Tong et al. 2002]. In this section, we first describe the basic idea of our algorithm: why previous work is not order-independent, and how our approach addresses this (Section 3.1). We then describe implementation details on how to make our algorithm both fast and memory efficient (Section 3.3 and Section 3.2).

3.1 Basic Idea

We can summarize previous neighborhood-search algorithms [Efros and Leung 1999; Wei and Levoy 2000; Ashikhmin 2001; Tong et al. 2002] as follows.

Goal: given an input texture, generate an output texture that is similar to the input.

1. Build pyramids for both the input and output, and initialize the output.

2. Synthesize the output texels one by one from lower to higher resolutions in a certain order, such as scanline [Wei and Levoy 2000] or spiral [Efros and Leung 1999].

3. For each output texel, build a neighborhood around it. Use this neighborhood to find the best-matched neighborhood from the input, and assign the corresponding input texel to the output. Different search algorithms can be used, such as TSVQ [Wei and Levoy 2000] or coherence search [Ashikhmin 2001].

4. Repeat the above step for each output texel.

An algorithm is order-independent if we can generate texels in arbitrary order (Step 2) while guaranteeing the resulting texture to be identical, given that we start with the same input and initial conditions (Step 1). However, existing methods [Efros and Leung 1999; Wei and Levoy 2000; Ashikhmin 2001; Tong et al. 2002] are not order-independent; since their neighborhoods always include the most recently synthesized results, different traversal orders will cause different combinations of old and new values at each neighborhood, altering the result of neighorhood search. For example, consider two nearby texels, A and B, where A is within B's neighborhood. If A is synthesized before B, then the neighborhood of B will include the new value of A; on the other hand, if A is synthesized after B, then the neighborhood of B will include the old value of A. As a result, the synthesized value B will depend on whether or not it is synthesized before A.

We address this limitation by a simple idea: instead of overwriting the old values with new results, we keep the old and new values *in separate output pyramids* and use only the old values in the neighborhood search process. This idea is inspired by image convolution. When convolving image X by a filter kernel to produce image Y, each pixel in Y is computed by convolving the kernel with the "old" pixels in X rather than the new pixels in Y. As a result, there are no dependencies among pixels in Y, and we can compute their values in any order.

To apply this convolution idea to texture synthesis, we keep multiple output pyramids, each storing a specific *generation* of the output. For example, we use generation 0 to store the oldest values, generation 1 to store values computed from generation 0, and so on. During synthesis, we traverse the output pyramids from lower to higher levels as usual (Step 2), but within each level, we compute the texels from lower to higher generations. To avoid dependencies between texels at the same level and generation, we modify the definition of neighborhoods (Step 3) so that, for a given output texel at a certain level and generation, its neighborhood can contain only "old" texels that are already computed, where the old texels are located at lower levels, or at the same level but lower generations. To be more precise, a texel located at $\langle \text{level } L_i, \text{ generation } G_i \rangle$ can belong to the neighborhood of a texel located at $\langle \text{level } L_j, \text{ generation } G_j \rangle$ only if $\langle L_i, G_i \rangle \prec \langle L_j, G_j \rangle$, where

$$\langle L_i, G_i \rangle \prec \langle L_j, G_j \rangle \text{ iff } [L_i < L_j] \text{ or } [L_i = L_j \text{ and } G_i < G_j] \quad (1)$$

This lexically smaller operator $\prec$ defines an acyclic ordering for all texels located at different levels or generations. Therefore it removes dependency between texels at the same $\langle$level , generation$\rangle$ and allows order-independent synthesis.

Example. Figure 1 shows an example of walking through our algorithm. The output pyramid contains 4 levels and 3 generations, and we use a 2-level neighborhood template with size 5×5. Given a user request pattern, our goal is to compute as few texels as possible to satisfy this request. From the initial pattern at $\langle L_3, G_2 \rangle$ and the neighborhood templates, we can figure out the set of necessary texels at every $\langle$level, generation$\rangle$ recursively, from higher to lower levels, and higher to lower generations. For example, the footprint at $\langle L_3, G_1 \rangle$ is computed by taking the union of all 5×5 neighborhoods of each texel within the "S" pattern in $\langle L_3, G_2 \rangle$, and the footprint at $\langle L_3, G_0 \rangle$ is determined from $\langle L_3, G_1 \rangle$ in a similar fashion. After we figure out this minimum set of texels at each $\langle$level , generation$\rangle$, we can then generate them from lower to higher levels, and from lower to higher generations. That is, we start at $\langle L_0, G_0 \rangle$, and then go to $\langle L_1, G_0 \rangle$, $\langle L_1, G_1 \rangle$, $\langle L_1, G_2 \rangle$, $\langle L_2, G_0 \rangle$, etc, until we end at $\langle L_3, G_2 \rangle$.

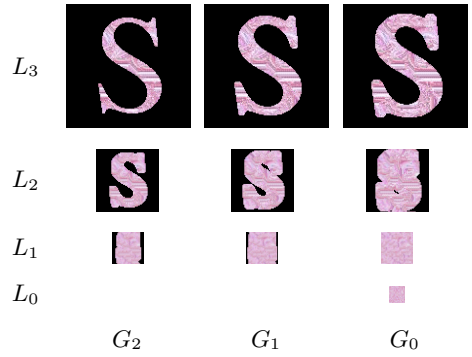

Figure 1: *Basic idea of our algorithm. In this example, we use an output pyramid with 4 levels and 3 generations, and a neighborhood template of size 5×5. $\langle L_3, G_2 \rangle$ contains an user-requested pattern "S". $\langle L_0, G_0 \rangle$ is initialized by randomly copying from the input. In the squares representing the remaining levels and generations, only a portion of the texels (denoted in pink above) must be computed.*

According to Equation 1, texels at $\langle L_0, G_0 \rangle$ cannot use any other texels in the neighborhood; therefore they are initialized by randomly copying from the input pyramid. Similarly, texels at G_0 (the right-most column of images in Figure 1) can only depend on texels at lower levels since there is no texel located at even lower generations. We can consider these texels as "extrapolation" or "super-resolution" from lower pyramid levels. For the rest of $\langle$level, generation$\rangle$, the texels can depend on lower levels as well as the same level with lower generations. However, for lower level neighborhood texels, we usually use only those located at the highest generation since these are the most up-to-date values at the specific level. Note that for a fixed initial condition at $\langle L_0, G_0 \rangle$, our algorithm will always generate identical results regardless of the traversal order within each $\langle$level, generation$\rangle$. This is similar to Perlin noise where the computed texel values remain invariant given a fixed random permutation table.

3.2 Making It Memory Efficient: Texture Cache

Our basic algorithm presented above is not memory efficient since it keeps multiple generations of the output. Fortunately, for many applications where only a subset of the texture is accessed (Figure 1), we only need to store this subset. In addition, since most practical rendering algorithms exhibit good texture coherence [Hakura and Gupta 1997], we can further reduce the storage requirement by caching only the recently computed/accessed values.

Our revised algorithm works as follows. Instead of storing the entire output, we use a small texture cache to store already-generated texels. This is different from prior methods [Hakura and Gupta 1997] where the cache is only used to reduce texture memory reads. In the beginning of the algorithm, the cache is empty and every requested texel needs to be computed. However, as the cache gradually fills up, previously computed texels may be requested again, and they can be found in the cache without any computation. If a cached texel is kicked out of the cache due to capacity limit, we may need to re-compute it if it is requested again. The

actual extent of re-computation depends on the rendering algorithm that drives these requests. Fortunately, since most applications exhibit good coherence, the ratio of re-computation is usually low, as we shall see in the result section (Section 4).

3.3 Making It Fast: K-Coherence Search

Our algorithm is orthogonal with respect to the specific neighborhood-search algorithm used in Step 3. However, since the search is in the inner loop, the choice of the search procedure will determine the overall speed of our algorithm. In addition, the search procedure will determine the quality of texture synthesis results.

We have experimented with several different approaches and found that the K-coherence algorithm [Tong et al. 2002] provides the best quality/speed trade off. The algorithm has constant time complexity per output texel and provides state-of-art synthesis quality. The 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 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 that determines the overall speed/quality. The authors in [Tong et al. 2002] have found that a small K (in the range of 1 to 11) works well for a variety of textures.

3.4 Summary

We can summarize our algorithm as follows. Given an input texture, an output texture cache size, a neighborhood template, and a texture access pattern:

1. Build a Gaussian pyramid for the input. Pre-process the input for the specific search algorithm, if necessary. The output pyramid is implemented as a cache, capable of holding texels at multiple generations. The lowest $\langle$level, generation$\rangle$ is initialized by randomly copying from the input.

2. For each requsted texel, determine the minimum set of neighborhood texels located at lower levels and generations. This can be achieved by recursively expanding the request patterns from higher to lower levels, and from higher to lower generations, as shown in Figure 1.

3. For texels in this minimum set that are not in the cache (including the requested texel), synthesize them one by one from lower to higher levels, and from lower to higher generations. The synthesis can be done via any neighborhood-search algorithms; we use K-coherence search in this paper.

4. Repeat the above two steps for each requsted texel that is not in the cache.

4 Results

In this section, we demonstrate several aspects of our algorithm's performance. We begin by demonstrating the effects of various synthesis parameters on image quality. We then characterize the cache performance under different access patterns.

Synthesis Quality

We have applied our approach with many different textures, and we found that it works as well as the original K-coherence search algorithm [Tong et al. 2002]. The only additional parameter we need to set is the number of generations. We have found that 2 or 3 generations work well for all textures we tried. If only 1 generation is used, the algorithm essentially uses only lower-resolution information for synthesis. This is very similar to [Bonet 1997] and, as shown in Figure 2, the result textures have similar artifacts.

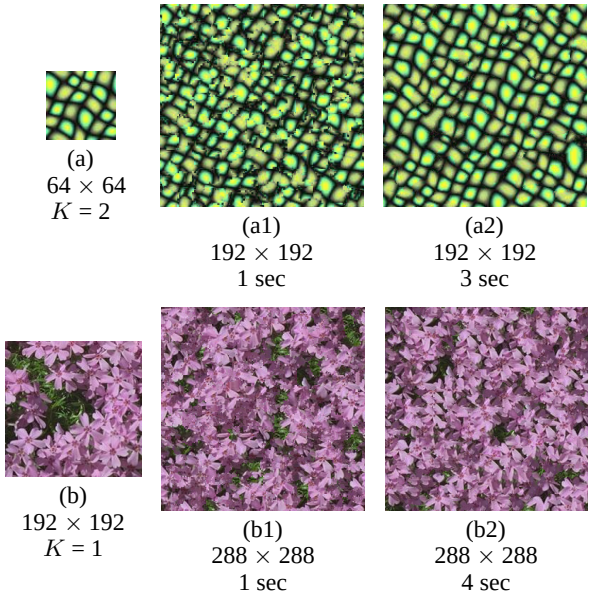

Figure 2: *Synthesis Quality. For each group of images, the input sample is on the left, the result with 1 generation is shown in the middle, and the result with 3 generations is shown on the right. Below each figure, we label the image size and computation time in seconds, measured on a 1.1GHz AMD Athlon PC. These synthesis results compare favorably to the best existing algorithms.*

Cache Performance

As in most cache-based algorithms, the size of the cache plays a critical role in its performance. When the cache in our algorithm is large, it can hold all the computed texels. However when the cache is small, it cannot hold all texels, and some texels may be computed multiple times. As a result, the performance of small caches is affected by the coherence of the request patterns.

We analyze the performance of our algorithm under different cache sizes using ray tracing and polygonal rasterizer scenes. We collect texel requests from the renderer of these scenes, and feed them into our algorithm for simulation. In our experiment, we use 3 benchmarks with different texture characteristics and triangle sizes, as shown in Figure 3:

- **Teapot.** This is a ray-casting scene containing a large textured teapot occluded by 3 non-textured teapots. The texture has size 256×256; due to ray-casting, the occluded portion of the texture is not computed and has the shape of the projected occluders. The scene is rendered by tracing rays in 32-pixel-wide vertical swathes, from top to bottom, left to right.

- **Single Polygon.** This scene contains a large textured polygon with size 512×512, viewed in perspective. The polygon is tessellated into 64×64 tiles and covered by a large 512×512 texture pyramid with 4 levels. Since the polygon is clipped by the viewing frustum, only 19 percent of the texels are requested by the rasterizer, and this portion is wedge-shaped due to the perspective view. Our algorithm synthesizes 23 percent of all cache pixels. Although this is slightly larger than 19 percent, it is still 4 times faster than synthesizing the whole texture using K-coherence search.

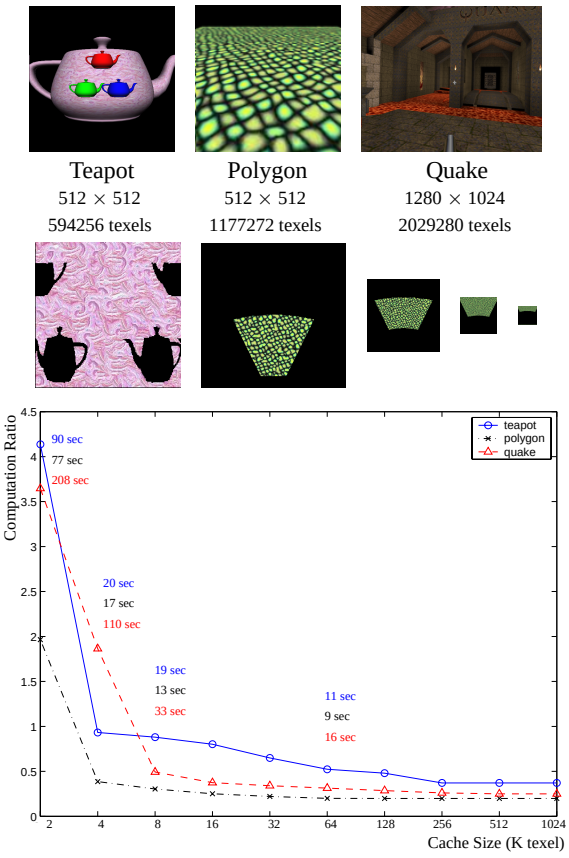

Figure 3: *Texture cache performance for several scenes: teapot, polygon, and quake. (Top) The screen shots of the 3 scenes, along with the frame size and total number of requested texels. (Middle) The requested texels for the teapot (only the highest resolution) and polygon (all 4 resolutions) scenes. The black regions indicate texels not requested (and therefore not synthesized). (Bottom) The cache size v.s. performance. We measure the performance in terms of computation ratio, which is the average number of computed texels per requested texel. The cache replacement policy is LRU. We also mark wall-clock simulation time (synthesis + cache emulation) at several key cache sizes.*

- **Quake.** This scene shows a frame from the OpenGL port of the video game Quake, containing mainly architectural walkthroughs with large polygons. We choose this benchmark to see how rasterizing large polygons may degrade cache coherence. The original Quake game uses many small textures to reduce texture load; although this works well for regular patterns such as brick walls, small textures may introduce unnatural visual repetitions for stochastic textures such as the lava. Our algorithm can address is problem by synthesizing large textures on the fly from a small input. We demonstrate this by synthesizing the lava texture with size 512×512 from a small 64×64 crop, and eliminate the majority of the unnatural repetitions in the rendering.

We measure the performance of our algorithm by *computation ratio*, which we define to be the average number of synthesized texels per requested texel. The performance is better when the computation ratio is lower since less computation is involved. In Figure 3, we plot the computation ratio versus different cache sizes for all three benchmarks. The algorithm performs reasonably well at small cache sizes (between 2K to 8K), and the computation ratio drops as the cache size increases. The performance remains roughly constant after the cache size reaches 8K, indicating that a small cache size is sufficient to hold the working set for these scenes.

Computation Time

As shown in Figure 2, our current implementation takes a few seconds to generate typical textures on a commodity PC without any hardware acceleration. This is sufficiently fast for interactive applications with small windows (such as the ray tracer shown in the video). However, for full-screen applications such as the Quake game in Figure 3, our current implementation is not fast enough. We plan to address this issue by implementing our algorithm on a programmable graphics hardware. This is likely to provide at least 2 orders of magnitude speed-up over our software-only implementation.

5 Conclusions and Future Work

In this paper, we have presented and analyzed a new algorithm for order-independent texture synthesis. The algorithm allows texture samples to be generated in arbitrary order on demand, yet the resulting texture always looks the same. The algorithm has comparable image quality with the state of art algorithms. It is computationally efficient with K-coherence search. It is storage efficient due to its use of a texture cache. We demonstrate that small caches are sufficient by analyzing our algorithm through different texture mapping scenes.

There are several possible directions for future work. Although our algorithm is fast enough for software applications such as ray tracing and image editing, it needs furthur speed improvements to be practical for hardware polygonal rasterizers. This can be achieved by finding a search algorithm more efficient than K-coherence search. Another possibility is to implement our algorithm as a fragment program in new generation GPUs. The fragment program would synthesize the texture in multiple passes, where each pass renders a specific generation/resolution of the texture. Our algorithm can also be used as a texture decompresser for a software viewer such as VRML or QuickTime VR. This would substantially reduce the storage space and transmission time for viewing scenes containing large textured regions. Finally, our algorithm can be implemented in a shading language and integrated with a ray tracing package. This could make statistical texture synthesis more useful to animators, who are more accustomed to procedural shaders.

References

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

BONET, J. S. D. 1997. Multiresolution sampling procedure for analysis and synthesis of texture images. In *Proceedings of SIGGRAPH 97*, 361–368.

EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. 1998. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers.

EFROS, A. A., AND FREEMAN, W. T. 2001. Image quilting for texture synthesis and transfer. In *Proceedings of ACM SIGGRAPH 2001*, 341–346.

EFROS, A., AND LEUNG, T. 1999. Texture synthesis by non-parametric sampling. In *International Conference on Computer Vision*, vol. 2, 1033–8.

HAKURA, Z. S., AND GUPTA, A. 1997. The design and analysis of a cache architecture for texture mapping. *24th International Symposium on Computer Architecture*.

HEEGER, D. J., AND BERGEN, J. R. 1995. Pyramid-based texture analysis/synthesis. In *Proceedings of SIGGRAPH 95*, 229–238.

SOLER, C., CANI, M.-P., AND ANGELIDIS, A. 2002. Hierarchical pattern mapping. *Proceedings of ACM SIGGRAPH 2002* (July), 673–680.

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

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