By-example Synthesis of Compactly Stored Textures

Kun Zou, Xiang Xu, Rong Li, Yueqiao Li and Zan Li

Department of Computer Engineering
University of Electronic Science and Technology of China, Zhongshan Institute
Zhongshan, P.R. China, cszoukun@foxmail.com

Abstract

This paper proposes a novel by-example texture synthesis algorithm. Unlike most present algorithms whose synthesized textures of different sizes must be stored in memory until rendering, our results are compactly stored as paths of the generative graphs. This is achieved by adapting the graph-based synthesis framework to be suitable for arbitrary types of textures. Two synthesis processes are carried out successively to grow the texture strip by strip to the desired horizontal and vertical dimensions, each of which is cast as a constrained shortest path problem that can be solved efficiently by our proposed algorithm. The strips are formed by consecutive cuts which are precomputed in the preprocessing step, and are selected with a redesigned mechanism accounting for both global and local matching errors. The variances make the graph-based synthesis framework also applicable to a wide variety of textures besides architectural textures, and the synthesis quality for them is greatly improved.

Keywords: texture synthesis; compactly stored; graph-based; constrained shortest path

1. Introduction

Texturing is an important process in Computer Graphics to add reality to computer generated models. By-example texture synthesis can produce arbitrarily sized textures based on the given samples. Its generative model is more widely applicable than procedural methods, and the synthesis process needs no complicated parameter tuning. Thus recent years it has gained much attention from researchers, and plenty of example-based synthesis algorithms have been proposed.

The great majority of synthesis algorithms [1-5] are off-line, which synthesize textures in demand beforehand, and outputs are stored for later rendering. These algorithms save authoring time but bring a waste of storing spaces. On the contrary, runtime synthesis algorithms [6][7] can synthesize textures on the fly, which require little storage. Their critical points lie in the spatial determinism and the acceleration of the synthesis process to achieve real-time. For interactive applications that involve large scene rendering, a great variety of textures of different sizes and types may be requested. Therefore, the storage demand for the target textures makes the off-line algorithms prohibitive. For runtime synthesis algorithms, the situation can not be much better as a result of the great challenge posed by synthesis of so many textures in real time.
Lefebvre at el. [8] solve this problem by proposing a graph-based synthesis framework, which casts synthesis as a shortest path problem in a graph describing the space of images that can be synthesized. After the synthesis, only the paths describing the result need to be stored in memory, and synthesized textures can be reconstructed at rendering time. However, they only focus on architectural textures, and the measurement of the jumping cost only considers local matching errors, thus may produce unrealistic features.

The main objective of this paper is to adapt the graph-based synthesis framework in [8] to be suitable for arbitrary types of textures. Our contributions include:

- Redesign the mechanism for parallel cuts searching in the preprocessing step, which takes into account both global and local matching errors.
- Adapt the restrictive conditions of the shortest path problem, and solve it efficiently by an improved algorithm.

The variances make the graph-based synthesis framework also applicable to a wide variety of textures besides architectural textures, and the synthesis quality for them is greatly improved.

The organization of the remaining parts is as follows. Section II reviews the graph-based synthesis framework in [8]. Section III analyzes the limitations disqualifying it from being applicable to general textures and discusses our proposed adaptations in detail. Experimental results are given in section IV, and conclusions are made at last in section V.

2. Graph-based synthesis framework

The synthesis framework in [8] can be divided into three phases: preprocessing, synthesis and rendering. During synthesis, two successive processes are carried out to grow the texture strip by strip to the desired horizontal and vertical dimensions, each of which is cast as a shortest path problem. The strips are formed by consecutive cuts which are precomputed in preprocessing. Next we only briefly review the unidirectional (horizontal for simplicity) synthesis process, for the extension to bidirectional synthesis in our method is the same as in [8].

Preprocessing: Consume that the source image I has width W and height H. In preprocessing, for each \( \sigma \) in \([1, W-1]\), parallel cuts located \( \sigma \) pixels apart are computed and added to the set of cuts \( C \). First, an error map \( E^\sigma_I \) of size \([W-\sigma] \times H\) is derived using

\[
E^\sigma_I(x, y) = \| I(x, y) - I(x + \sigma, y) \|^p,
\]

in which \( p \) is a predefined constant. Then the minimal-cost Y-monotone path from the top to the bottom row of \( E^\sigma_I \) is computed using dynamic programming as in [3]. After that, the value of pixels in \( E^\sigma_I \) less than \( \sigma \) pixels away from the path just obtained is set to infinity, and return to the path computation step. If no path with finite cost can be found, the cycle then terminates.

Synthesis: The synthesis result can be assembled by strips from the source image as a sequence of cuts \( A = c^*, c_0^0, c_0^1, ..., c^n_0, c^n_1, c^+ \) where \( c^* \) and \( c^+ \) are starting and ending cuts chosen by the user, and parallel cuts \( c^0_1, c^n_1 \) are the ending and starting cuts of two successive strips. Consider synthesizing by adding cuts to the image under construction as in Fig. 1. At a time point when the current ending cut is \( c^a \), there are two choices to continue: either growing the current strip to one succeeding cut \( c^b \) (shown green) with no cost, or starting a new strip by jumping to the parallel cut \( c^a_1 \) (shown red) with cost
\[
\delta(c, c_i) = \sum_{y \in [0,H-1]} |I(c(y)+1, y) - I(c_i(y)+1, y)|^p.
\] (2)

In this manner a graph \( G = (C \times Z, E_\prec \cup E_{||}) \) can be constructed where \( E_\prec \) and \( E_{||} \) denote growth edges and start edges respectively that correspond to the aforementioned two choices. Each node \((c, z) \in C \times Z\) represents the cut \( c \) translated to abscissa \( z \) in the result image. A path in \( G \) from node \((c^\ast, c_{\min}^\ast - c_{\max}^\ast)\) to node \((c^\ast, W_T)\) corresponds to an assembly \( A \) with cost

\[
\delta(A) = \sum_{n=0}^n \delta(c^n, c^n_\ast),
\] (3)

where \( W_T \) is the target width. Thus finding the assembly with minimum cost can be cast as a shortest path problem. In order to limit repetitions, a restrictive condition is added that prevents a source column from appearing more than \( R \) times within a \( \xi \)-pixel wide window. The constrained shortest path problem is solved by using Dijkstra’s algorithm except that an edge is not followed if it would result in a path not enforcing the constraint.

Figure 1. Illustration of synthesis by adding cuts [8].

**Rendering:** After the synthesis step, only the source image and the path need to be stored in memory, and the synthesized image can be reconstructed easily at rendering time. The details are given in [8] and we will not dwell on it.

3. Adaptations

Adaptations are made primarily on preprocessing and unidirectional synthesis to enable the graph-based framework applicable to general textures. The extension to bidirectional synthesis and rendering are the same as in [8], so they are omitted in the following.

3.1. Preprocessing

The most important task of preprocessing is searching for and adding parallel cuts to cut set \( C \). In essence, the variety of the synthesis results is provided by jumping between parallel cuts which corresponds to start edges in the graph. If only grow edges exist, the synthesized texture has to be a cropped version of the source.

Consider the simple example in Fig.2 (a), a single pair of parallel cuts with \( w_c \) pixels apart is shown in the source image. If only one jump from \( c^a \) to \( c^b_\ast \) is taken and the ending cut is the last column of the source, the result will be \( W-w_c \) pixels with the yellow region deleted as in Fig.2 (b). And if the jump direction is reversed, the result will be \( w_c \)-pixel wider than the source with the yellow region doubled, as in Fig.2 (c).
Ref. [8] searches for parallel cuts with $\sigma$ pixels apart, $\sigma$ varying from 1 to $W$-1, so that a variety of paths exist regardless of the value of $W$. However, because the neighboring pixels usually bare close values, the start edge with minimum cost probably corresponds to a shortest jump ($\sigma$=1), and very likely that the shortest path contains a few such edges, which means some entities in the source may be elongated or shortened (such as narrower doors, longer cars). When applying the framework to general texture synthesis, especial when the texture is regular or semi-regular, things get even worse. So we limit $\sigma$ to be no less than the width of the texel.

![Illustration of jumps between parallel cuts](image)

Figure 2. Illustrion of jumps between parallel cuts: (a) is the source image with a single pair of parallel cuts shown; (b) is the result by taking only one jump from $c^a$ to $c^b$; (c) is the result when the jump direction is reversed.

Another insight is that the jump cost is calculated as (2) by summing the per-pair errors between pixels right to the two cuts. This is a local measure and can not guarantee the consistence of large features. Regular or semi-regular textures usually have periodicity, and jumps between parallel cuts with $\sigma$ pixels apart may induce artifacts when $\sigma$ collides with the period.

In fact, the error map calculation with (1) is the same as computing the per-pair pixel inconsistence when two source images are aligned with $\omega$-pixel-wide overlap as in Fig. 3. The matching errors of the overlap region have much to do with the periodicity. So we search for parallel cuts with $\sigma$-pixel apart only when the average matching error of the overlap region is small.

This adaption may lead to a result that a path doesn’t exist when the ending cut is specified, but fortunately, general texture synthesis usually need not specify the ending cut, and even if we need, the path’s inexistence may mean that the specification does not coincide with the texture’s periodicity.

The adapted preprocessing step can be described with the pseudo-code hereafter.

```plaintext
for width \( \omega \) from texelWidth to \( W \)-1 do
    Compute the average matching error of \( \omega \)-pixel-wide overlap.
    Add the first \( N W - \omega \) values with minimum matching errors to \( T \).
for each \( \sigma \in T \) do
    Compute an error map \( E_{\sigma}^a \) of size \([W-\sigma]\times H\).
    while the shortest-path \( \pi \) from top to bottom of \( E_{\sigma}^a \) has finite cost do
        Augment \( C \) with the two parallel cuts corresponding to \( \pi \).
        In \( E_{\sigma}^a \), set the value of pixels less than \( \sigma \) pixels away from \( \pi \) to \( \infty \).
```

(a) (b) (c)
3.2. Synthesis

In our adapted graph-based synthesis framework, besides the synthesis with ending cut specified, unconstrained synthesis is also supported, which aims at synthesizing a texture with the desired size that has the minimum matching error. This corresponds to searching the shortest path among paths from the start node to a node $(c, z)$ with $z \geq W_f$. We solve it efficiently with the following algorithm.

The shortest path searching process involves two kinds of operations, named “Grow” and “Jump” respectively, corresponding to the two choices mentioned in Section II.

![Two source images with $\omega$-pixel-wide overlap.](image)

**Figure 3.** Two source images with $\omega$-pixel-wide overlap.

Grow$(c, z)$ traverses the graph from node $(c, z)$ to any unvisited node along growth edges (0 cost), while Jump$(c_1, c_2, c, z)$ follows a start edge from an visited node $(c_1, z')$ to an unvisited node $(c_2, z')$, and updates the current node to $(c_2, z')$ by setting $c = c_2$ and $z = z'$, where $c_2$ is parallel to $c_1$. Only the Jump operation adds cost to the current path. Rather than traverse all paths in the graph, we add an edge with minimum cost to the visiting tree until termination condition is satisfied. The frame is like this:

```
Initialize $c$ to be the start cut and $z = c_{\min} - c_{\max}$
while (TRUE) do
    if (Grow$(c, z)$) break;
    for each pair $(c_1, c_2)$ of parallel cuts (sorted by cost) do
        if $c_1$ is active
            if (Jump$(c_1, c_2, c, z)$) break;
        if $c_2$ is active
            if (Jump$(c_2, c_1, c, z)$) break;

All nodes in the graph are organized as a two dimensional array indexed by cut id and $z$ value, so graph nodes corresponding to one cut is easily accessed. In the Grow stage, before following a growth edge, a check for limiting repetition is conducted, which is the same as in [8]. In synthesis with ending cut specified, the termination condition is that the newly visited node is just the target node; while in unconstrained synthesis, it is that the new $z$ value overtakes $W_T$. If termination condition is satisfied, Grow$(c, z)$ will return TRUE. To increase the efficiency of the Jump stage, we sort all parallel cuts by cost, and do one jump with minimum cost each time, then return to the beginning of the cycle to grow the new node. When a new node is visited in Grow, the corresponding cut is set to be active. So the valid jump with minimum cost can be easily found with the aid of the sorted list of parallel cuts and the activity check. The Jump function returns TRUE if a valid jump is done, otherwise FALSE.
4. Experimental results

All results are obtained on an Intel Core2 Duo CPU T5750 2.00GHz using a single core for computations.

Our objective is to synthesize general textures, but the work in [8] is targeted at synthesizing architectural textures and the ending cut is specified, which may easily disagree with the periodicity of regular and near-regular textures. In order to be fair, we compare our results on general textures with that from the unconstrained version of [8], which differs from the original work in that the synthesis phase follows our method. The comparison is given in Fig. 4. The synthesized textures are all sized at $256 \times 256$. Obviously, our results are superior to the ones from the unconstrained version of [8], which show artifacts in the form of elongated or shortened features and repetition. This superiority is gained by the adapted preprocessing which confines $\sigma$ to be large enough and coincide with periodicity of textures.

Timing data for Fig. 4 is shown in Table. I. Although our adapted preprocessing adds one step to compute total matching errors for different $\sigma$, it consumes less time because only a few parallel cuts are computed later. The synthesis phases of both algorithms are the same, but our synthesis time is shorter by reason of fewer nodes in graph. By only adapting the synthesis phase of the algorithm in [8] with specified ending cut, the synthesis from a $512 \times 512$ source to a $500 \times 750$ target consumes 5.224 seconds.

![Figure 4. Synthesis results by the unconstrained version of [8] and the proposed algorithm.](image)
Table I Timing Data for The Synthesis in Fig. 4

<table>
<thead>
<tr>
<th>Sample</th>
<th>Algorithm</th>
<th>Time (in s)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Preprocessing</td>
<td>Synthesis</td>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>Rope</td>
<td>Unconstrained [8]</td>
<td>4.208</td>
<td>0.243</td>
<td></td>
<td>4.451</td>
</tr>
<tr>
<td>(192×192)</td>
<td>Ours</td>
<td>2.563</td>
<td>0.011</td>
<td></td>
<td>2.574</td>
</tr>
<tr>
<td>Cans</td>
<td>Unconstrained [8]</td>
<td>4.225</td>
<td>0.228</td>
<td></td>
<td>4.453</td>
</tr>
<tr>
<td>(192×192)</td>
<td>Ours</td>
<td>2.578</td>
<td>0.012</td>
<td></td>
<td>2.59</td>
</tr>
<tr>
<td>Chains</td>
<td>Unconstrained [8]</td>
<td>3.053</td>
<td>0.257</td>
<td></td>
<td>3.31</td>
</tr>
<tr>
<td>(200×150)</td>
<td>Ours</td>
<td>1.918</td>
<td>0.012</td>
<td></td>
<td>1.93</td>
</tr>
<tr>
<td>Horses</td>
<td>Unconstrained [8]</td>
<td>5.120</td>
<td>0.286</td>
<td></td>
<td>5.406</td>
</tr>
<tr>
<td>(221×191)</td>
<td>Ours</td>
<td>2.499</td>
<td>0.010</td>
<td></td>
<td>2.509</td>
</tr>
</tbody>
</table>

(preprocessing not included), which is shorter than that of the original, indicating that our adapted synthesis is more efficient.

5. Conclusions

In this paper, an algorithm for by-example synthesis of compactly stored textures is proposed by extending the graph-based synthesis framework in [8] to general texture synthesis. By confining distances between parallel cuts to be not too small, and considering global matching errors, the set of parallel cuts only contains cuts between which the jumps will preserve large features and the periodicity of texture. The synthesis step is also adapted to support unconstrained synthesis in the case of general texture synthesis, and an efficient scheme is proposed to obtain the constrained shortest path. Unlike most present off-line synthesis algorithms, only the path of the generative graph and the source need to be stored, and synthesized textures can be reconstructed easily at rendering time.

Acknowledgment

This paper is supported by the Scientific Research Foundation of University of Electronic Science and Technology of China, Zhongshan Institute (408YKQ08), and Natural Science Foundation of Guangdong Province, China (10452840301004638).
References


