Hierarchical Pattern Mapping

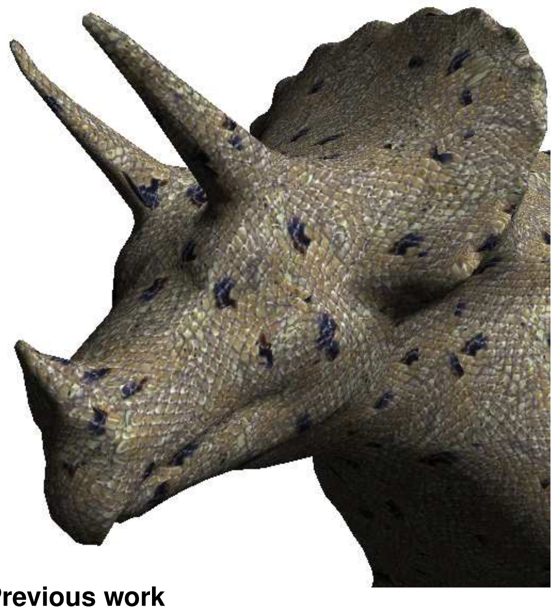
Cyril Soler Marie-Paule Cani Alexis Angelidis

iMAGIS* / GRAVIR-IMAG / INRIA

Abstract

We present a multi-scale algorithm for mapping a texture defined by an input image onto an arbitrary surface. It avoids the generation and storage of a new, specific texture. The idea is to progressively cover the surface by texture patches of various sizes and shapes, selected from a single input image. The process starts with large patches. A mapping that minimizes the texture fitting error with already textured neighbouring patches is selected. When this error is above a threshold, the patch is split into smaller ones, and the algorithm recursively looks for good fits at a smaller scale. The process ends when the surface is entirely covered. Our results show that the method correctly handles a wide set of texture patterns, which can be used at different mapping scales. Hierarchical texture mapping only outputs texture coordinates in the original texture for each triangle of the initial mesh. Rendering is therefore easy and memory cost minimal. Moreover the initial geometry is preserved.

Keywords: Level of Detail Algorithms, Texture Mapping, Texture Synthesis



1 Introduction

Being able to add small non-geometric details to CG objects is essential for enhancing the visual complexity of virtual worlds. Texturing arbitrary surfaces has thus attracted much interest within the past few years. Among these methods, those which map or generate repetitive patterns onto surfaces (*i.e*, pattern mapping) are particularly promising, since they allow easy modeling of natural-style materials such as stone, wood, marble or scales.

Pattern mapping is more effective, in terms of memory cost, than painting or generating a new global texture map on each object. However, regularly mapping rectangular texture samples is not applicable in the general case, since most surfaces have no global parameterization and cannot be unfolded onto a plane. Our new hierarchical texture mapping method addresses this problem. It shows that seamless texturing can be obtained by directly selecting and mapping, from coarse to fine scales, some possibly not contiguous texture patches from the input image. This method, which outputs texture coordinates at the original mesh vertices, works for arbitrary manifold meshes and texture patterns.

*iMAGIS is a joint project of CNRS, INRIA, Institut National Polytechnique de Grenoble and Université Joseph Fourier.
Address: iMAGIS-GRAVIR, INRIA Rhne-Alpes, 655 avenue de l'Europe, Montbonnot, 38334 Saint Ismier Cedex, France.
Email: [Cyril.Soler|Marie-Paule.Cani|Alexis.Angelidis]@imag.fr.

1.1 Previous work

A complete review of all previous texture mapping and texture synthesis methods is beyond the scope of this paper. Instead, we focus on the recent progress made towards texturing an arbitrary surface with patterns from an input image.

Pattern mapping methods Traditional approaches for texturing a surface with repetitive patterns consist of mapping a 2D, toroidal texture pattern on it. However, most surfaces do not have a toroidal parameterization and cannot be unfolded onto a plane. Optimization algorithms can be used to limit texture distortion [Bennis et al. 1991; Maillot et al. 1993; Lévy and Mallet 1998; Lévy 2001], but cracks and singularities can hardly be avoided. Moreover, texture periodicity resulting from the mapping of a rectangular pattern is obvious and often spoils the visual quality of results. Recently, two new approaches were introduced to overcome these problems.

Neyret and Cani [Neyret and Cani 1999] precompute a set of triangular texture samples that match together along borders. The surface is tiled at the desired scale into curved triangular areas in which these texture samples are mapped. Arbitrary surfaces are thus textured with low distortion, no cracks, and no singularities. Moreover, the method uses little memory, since no global texture map needs to be stored. It is, however, restricted to isotropic texture patterns. In addition, creating the samples either demands delicate manual edition, or the implementation of specific procedural texture synthesis methods. Lastly, the resulting textured mesh is more complex than the initial one, since all the triangles crossing the limit of a texture tile are split during the process.

A year later, Praun, Finkelstein and Hoppe [Praun et al. 2000] proposed a suitable method for anisotropic textures. The idea is to iteratively paste irregular texture patches onto a surface. The patches locally overlap and align their main features with a predefined vector field. The method generates very nice results, but is again only applicable to a specific class of textures: the latter should not be sensitive to discontinuities (in practice, the textures used have internal discontinuities), and should have no low frequency variations. As with the previous method, this one demands some specific user input (the user must define texture tiles that fit with the intrinsic texture discontinuities). Lastly, rendering requires either composition operations or the storage of a global texture map.

Our method can be seen as an extension of these two techniques. However, we believe that performing texture mapping at a given, preselected scale is not sufficient: as stressed by Ying [Ying et al. 2001], the above methods cannot use texture tiles that are large with respect to the underlying geometry, thus considerably restricting the range of scales at which the pattern can be mapped. Moreover, they cannot capture low-frequency patterns while preserving high frequency randomness in the texture. Our method does not suffer from these drawbacks, since it uses a hierarchical approach inspired from the texture synthesis techniques described next.

Texture synthesis on surfaces The most recent contributions [Ying et al. 2001; Wei and Levoy 2001; Turk 2001] to texturing an arbitrary surface from example follow a different strategy. Rather than mapping texture samples, they generate a new texture directly on the surface, an idea that had only been tried in the past for specific classes of procedural textures [Turk 1991; Ebert et al. 1998; Walter et al. 2001].

These new methods were inspired by the recent progress of 2D texture synthesis from example [Efros and Leung 1999; Wei and Levoy 2000; Ashikhmin 2001; Hertzmann et al. 2001]. Image generation is performed pixel by pixel by fitting a region adjacent to the current pixel to similar regions of the input sample. The pixel value is then chosen randomly among the possible, resulting values. Wei and Levoy [Wei and Levoy 2000] make the method more robust for smooth textures by synthesizing an image pyramid from coarse to fine scales, and accelerate neighborhood search using tree-structured vector quantization. Ashikhmin [Ashikhmin 2001] instead restricts the search to locations predicted by already processed neighbouring pixels, thus reducing computations and increasing image coherency. This makes the method usable for highly structured patterns (*e.g.*, small objects of familiar shapes). Hertzmann [Hertzmann et al. 2001] combines Wei's and Ashikhmin's approaches, thus getting the benefits of both.

Three slightly different generalizations of Wei and Levoy's multi-scale synthesis algorithm [Wei and Levoy 2000] to surface texturing were proposed in 2001, respectively by themselves [Wei and Levoy 2001], by Turk [Turk 2001], and by Ying et al. [Ying et al. 2001]. The first step for making the method usable in 3D is to compute or define a vector field on the surface (as noted by Wei, a random field can be used if the texture pattern is isotropic). A mesh hierarchy is then built to serve as the image pyramid in the 2D synthesis approach. Since a level of the hierarchy will correspond to a given texture scale, uniform surface sampling is required for all meshes. Both Wei and Turk rely on Turk's mesh re-tiling method [Turk 1992] for building such a hierarchy from an arbitrary mesh. Ying's implementation only handles multi-resolution subdivision surfaces. Another step is to define a correspondence between the texture grid and the region of the mesh that surrounds a vertex. The multi-scale synthesis process can then be run, by treating vertices either in sweeping or in random order. The quality of the results is almost similar to those obtained for 2D images. Ying also extends Ashikhmin's coherent synthesis algorithm [Ashikhmin 2001], thus making texture synthesis on surfaces usable for highly structured patterns. Although they rely on a hierarchy of regular meshes for texture generation, the methods above may output texture coordinates on the initial mesh, using either a global texture map [Turk 2001] or a set of local maps that cover the surface [Ying et al. 2001]. In all cases, they require the storage of a new texture that fully covers each object.

In a way, our method synthesizes the texture on the surface, and also uses a multi-scale process for ensuring global pattern coherency while minimizing texture distortion. However, we do so by directly *selecting texture patches of various size from the input image*. This accelerates the process since surface regions that could be textured at a large scale require no further processing. Moreover, it suppresses texture storage memory requirements, since *no new texture* is created.

Stitching image patches The idea of creating a new texture by stitching together texture patches from a sample image was already used in 2D. Efros and Freeman [Efros and Freeman 2001] create a larger image by selecting rectangular texture tiles of a given size from the input image. Discontinuities across tiles boundaries are reduced by connecting them along a non-straight path, which minimizes an error. This method gives impressive results for a wide range of textures, although small scale artifacts are generally noticeable.

Extending Efros's approach to 3D is not straightforward: in 2D, Efros makes the method work by choosing the size of the texture tiles according to the size of the patterns in the input image. This cannot be done in 3D since the sharpness of the local surface geometry must also be taken into account. Moreover, texture patches cannot be rectangles anymore if we want to make the method effective on arbitrary surfaces. Lastly, the orientation of the patches in the texture may vary, which makes the search for good fits much more difficult. Our hierarchical texture mapping algorithm solves these problems. It improves the quality of connectivity across texture patches edges thanks to a local relaxation process.

1.2 Overview

The aim of this paper is to provide a general texturing method, where the user provides a sample image and an arbitrary 3D mesh, and gets texture coordinates on the mesh vertices as an output. The new method belongs to pattern mapping approaches, our basic claim being that everything we need is already there in the sample image. As shown by the work cited above, such pattern mapping cannot be performed at a single scale: Mapping large parts of the image sample will work in almost flat regions, but may cause unbearable texture distortions over the surface's sharp features. Moreover, due to the surface's specific topology and geometry, texture fitting with neighbouring patches cannot necessarily be done at a large scale. We solve these problems by introducing a hierarchical texturing algorithm. The idea is to recursively pick up texture patches from the sample image and map them onto the surface so that they fit with their neighbors. We do so in a coarse-to-fine manner, a surface region that could not be textured correctly being subdivided into smaller regions to improve the mapping. In the worst case, the process ends up at the scale of individual mesh faces, where remaining gaps in the texture are filled as well as possible.

Since the texture we get is made of patches of different shapes, sizes, and orientations from the input image, no periodicity can be observed. Moreover, although there will always be remaining artifacts if we zoom in close enough, visible texture discontinuities along texture patch edges can be avoided. This is done by keeping the fitting error under a given threshold.

2 Hierarchical mapping

2.1 Preprocessing

Before running the algorithm, the mesh, which is supposed to be a single manifold surface, is first converted into a hierarchy of regions. regions at each level of the hierarchy form a partition of the surface. Sibling hierarchies of regions have already been used in computer graphics [Garland et al. 2001]; We will use the already existing term of *face-clusters*. Each cluster is defined by the list of the mesh faces it includes and must have the topology of a disc.

For our algorithm, there are no further constraints: a face-cluster may possess any number of sub-clusters at the next hierarchy level; the cluster's shapes are arbitrary. Their sizes need not be homogeneous, even at a given hierarchy level. Although many methods could be used for building such a hierarchy, we give a very simple algorithm for constructing it in Section 3.

In addition to the mesh, the user inputs an image of the desired texture pattern. This image needs not be toroidal, although this property will increase the number of possible fits for texture patches. The user also specifies at which scale he wants the pattern to be used.

2.2 Algorithm

The cluster hierarchy is processed in a coarse-to-fine manner until all the mesh has been textured. At each level of the hierarchy, texture mapping is grown from one face-cluster to neighbouring ones. Only clusters where no good texture mapping could be found at a given hierarchy level are marked for further processing at the next level.

The traversal of a level is implemented using a priority queue: clusters that have the largest number of already textured neighbouring regions are processed first, so as to increase the chances for finding compatible solutions for all patches.

The process starts at the coarsest level of the hierarchy which has a region verifying the flatness criteria. This face-cluster is flattened in texture space, and a random choice is taken for its texture map, according to the user-specified pattern scale. The neighbouring clusters of the same hierarchy level are added to the priority queue. The process then drains all queues from coarse to fine hierarchy levels:

1. Take a face-cluster from the current queue;

2. Flatten the current cluster; If it is not sufficiently flat, computing the mapping at this scale would produce texture distortions. In this case, put its children into the queue corresponding to the next hierarchy level, and go back to step 1;

3. Look in the texture for a possible position and orientation of the flattened face-cluster that matches already textured neighbors. Estimate the error due to texture discontinuities;

4. *If this error is below a threshold*, map texture coordinates onto all polygons in the face-cluster;

   Else subdivide the region (i.e., put its children into the queue corresponding to the next level);

5. Add the non-textured neighbors of the face-cluster to the current queue;

Although a vector field could be precomputed or specified to guide pattern orientation as was done in [Praun et al. 2000; Wei and Levoy 2001; Turk 2001], we instead let the orientation propagate from the first texture patch to the others. As shown in Section 5, this simple approach gives good results for the set of anisotropic textures we used.

2.3 Problems to be solved

In addition to computing a hierarchy of regions from an arbitrary mesh, we need algorithms for flattening a given face-cluster and for efficiently searching for texture patches that fit with already textured neighboring regions. This needs to be done efficiently, since evaluating the fit, pixel by pixel, with each possible position and orientation could easily become a bottleneck in the algorithm.

3 Hierarchy of face-clusters

3.1 Setting up the hierarchy

Our algorithm needs a hierarchy of regions subdivided into levels, such that *(a)* the regions of a given level constitute a partition of the input mesh; *(b)* region borders are defined using the edges of the input mesh; and *(c)* the regions of the deepest level are the faces of the input mesh.

We define the *control vertices* of a face-cluster at level l as the points of its contour shared by at least three regions at this level. For each region, we store the list of control vertices and border vertices, as well as pointers to its sub-regions and to the neighboring regions of the same hierarchy level (see Figure 1).

We present two ways of obtaining such hierarchies: subdivision surfaces and general mesh hierarchies.

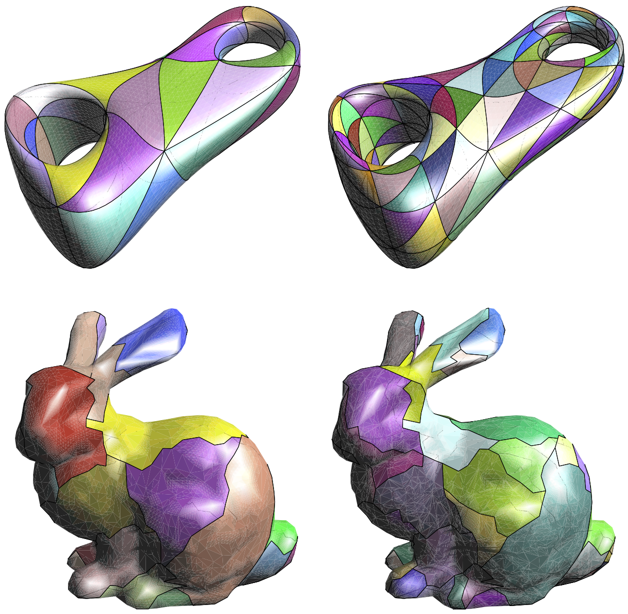


Figure 1: Face cluster hierarchies. *top row :* level 1 and 2 of a hierarchy built from subdivision surfaces (loop scheme). *bottom row:* levels 1 and 2 of a general hierarchy constructed using our method.

Subdivision surfaces Getting the required hierarchy of regions is obvious if the input mesh is a subdivision surface. We simply use the parent-child relationship between subdivision patches for setting up the hierarchy. Control vertices are defined as the vertices of the subdivision patches (see Figure 1, *top row*). Some of the results we show in Section 5 were computed with such surfaces. The drawback of this approach is that its restricts the class of usable surfaces.

General mesh hierarchies For a given input mesh, we proceed as follows: The first, coarsest face-cluster is made of the entire surface. Then, we recursively subdivide a face-cluster into a variable number n of sub-regions in the following way: we first select n faces of the mesh in this region to be used as seeds for growing the sub-regions; for each untreated face of the mesh sharing an edge with at least one of the growing sub-regions we merge it to the sub-region for which the distance to the seed face is minimal. Sub-regions grow inside the current face-cluster until paving is completed.

The number n of sub-regions is adapted to each face-cluster so as to keep a nearly uniform region size at each hierarchy level and to avoid sub-regions with fewer than three control vertices. The seeds are randomly chosen while checking that the control vertices of the sub-regions satisfy the non alignment property enounced in Section 3.2, for the stable computation of texture coordinates. At each level we compute the borders of the sub-regions and possibly exchange triangles between neighboring sub-regions in order to improve the border's regularity. We also deduce from these borders the control vertices of the sub-regions.

This method results in a hierarchy of regions of arbitrary shapes and sizes. This is not a problem since the fitting method described in Section 4 works for any projected patch shape in texture space. We may even say that for some textures, complex region borders tend to give better visual results, since tinny fitting errors are less visible when they occur along a non-straight path [Efros and Freeman 2001].

3.2 Flattening a surface region

Mapping a portion of the input texture onto the mesh requires the definition of a local mapping from surface space to texture space. Each time a face-cluster is processed (see Section 2), it first needs to be flattened. In practice, regions at the coarsest levels of the hierarchy are hardly ever used, since there is little chance to unfold them properly, at least for closed meshes. They may even have a non-zero topological genus. Similarly, face-clusters whose border is made of several disconnected components (such as a cylinder for instance) are discarded.

Most local flattening methods used in previous work [Bennis et al. 1991; Praun et al. 2000; Wei and Levoy 2001; Turk 2001] were aimed at flattening the neighborhood of a given mesh vertex. A spiral traversal has been, for instance, used for progressively flattening the region around a vertex. Our problem is different, since we have to flatten regions of totally arbitrary shapes in such a way that their projected border fits, in 2D, with the projected border computed for neighboring regions. This can actually be achieved using harmonic maps [Eck et al. 1995] through the resolution of a linear system. We have chosen to develop our own approach to avoid the cost of inverting a linear system for each mapped face-cluster, based on barycentric coordinates. This approach is described next.

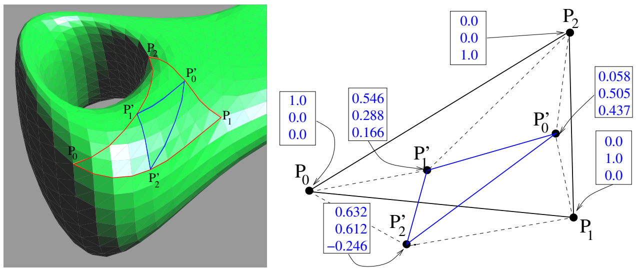
Barycentric coordinates for mesh vertices We do not compute any analytical expression for the mapping function of a region of the mesh to texture space. It is rather defined by a recursive procedure which relies on the face-cluster hierarchy:

(1) Suppose that we know the coordinates for control points $P_0 \dots P_n$ of a face-cluster in texture space. We define the relative positions of the mapped control points $P'_0 \dots P'_N$ of its sub-faces using barycentric coordinates (see Fig. 2). Let us call $T^{(l,j)}$ the vector of coordinates of the control points of a patch j at level l in texture space, and $C^{(l,j)}_{bary}$ the matrix of its barycentric coordinates with respect to the parents' control points. We set:

$$T^{(l,j)} = C^{l,j}_{bary} \times T^{(l-1,Parent(l,j))} \tag{1}$$

(2) When projecting a face-cluster in texture-space however, we need to fix the texture coordinates of its control vertices. These texture coordinates are precomputed by constructing a control polygon in texture space. As any face-cluster may be a starting point for recursive texture mapping, such a control polygon is computed for each face-cluster (see right parts of Figures 2 and 3).

The control polygon of a face-cluster is obtained by measuring the relative distance and angles of the polygon formed by its control vertices in 3D space. Then we build, in texture space, a closed polygon with the same edge lengths, and angles as close as possible to the original ones. Flat polygons in 3D space are shape-invariant through this transformation. Others are all the more deformed that the face-cluster contour is large and non flat (see examples in Figures 2 and 3).



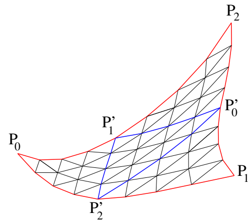


Figure 2: *Top left:* a face-cluster defined by 3 control points P_0, P_1 and P_2, one of its sub-faces out-lined in blue. *Top right:* the corresponding control polygon in texture space, and the barycentric coordinates of the control vertices of sub-faces, in blue. *Right:* the resulting flattened patch in texture space.

The barycentric coordinates of the sub-vertices in texture space are computed in three different ways depending on their nature: (1) Vertices that are already vertices of the parent face-cluster get a zero coordinate everywhere except on that vertex. (2) For vertices that belong to an edge of the parent face-cluster, we compute the average value of the mesh normal along this edge and project the 3D polygon formed by the parent face-cluster's control points onto the plane defined by this normal and the current vertex. Barycentric coordinates of the vertex are computed with respect to this projected polygon. (3) Finally, for vertices lying inside the parent face-cluster, we use the orientation of the mean square plane defined by the parent face-cluster border: we project the polygon formed by the parent face-cluster's control points and the current vertex onto this plane. We then compute the barycentric coordinates using this projected polygon.

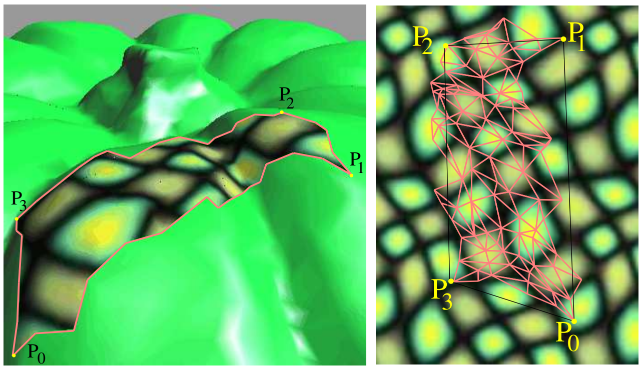


Figure 3: A face-cluster in surface-space (*left*) with its 4 control points, and its projection in texture space using our procedural mapping algorithm (*right*). Only the control points of this cluster and the barycentric coordinates of clusters below are needed to produce the mapping.

This procedural definition has some advantages against explicit mapping functions : First it lets us continuously deform the patch in texture space by moving one of its control vertices. Secondly, the mapping of a face-cluster is very fast and straightforward whatever the geometry of its control polygon in texture space (this is an advantage over harmonic maps), which lets us modify the mapping at an optimal cost. Finally the computation of texture coordinates is very stable, provided that the control points the barycentric coordinates are refering to do not form a very flat polygon (this produces very high or even infinite values that accumulate errors down the recursive computation of texture coordinates). In any case, we may choose among the possible vertices of the parent control polygon the base that gives the smallest barycentric coordinates, which is all the more easy that the face-clusters are not too much elongated. This condition is intrinsically verified by subdivision surfaces, whereas it is enforced for general surface hierarchies during their construction.

Flattening error criterion For a given face-cluster, let r be the ratio between its area in surface space and its area in texture space. We then define the *flattening error* of a face-cluster as E_{flat} by: $E_{flat} = |r-1|$

By construction E_{flat} is zero for flat face-clusters and grows with surface deformation. It is used as an heuristic measure of flattening error in step 2 of the algorithm of Section 2.

4 Texture patches fitting

This section explains how to texture a face-cluster which has some already textured neighbors while ensuring the match of the texture along common edges. The next paragraphs describe the four steps of the algorithm: we first extract a mask that represents the (already fixed) texture surrounding the current patch. We then look for the location in the texture image that best fits this mask. In a third step, we obtain texture coordinates through a recursive computation. Finally, we slightly tune texture coordinates along the edges using a local relaxation. This gives us the final mapping and an output fitting error for the patch.

4.1 Extraction of the mask

For most texture patterns, matching color values of pixels along common edges would not be sufficient for obtaining a good fit. We rather match the texture over the union of narrow bands extracted from textured neighbouring regions, along common edges (see Figure 5, *left*). This is a simple way for taking into account derivatives and local characteristics of the texture across regions' borders.

The extraction of the mask from neighboring texture patches requires being able to cut from these patches a band of approximately constant width. Without this ability, the algorithm would assign a nonuniform importance to the quality of texture fitting at different locations along the edge. In order to achieve this, we equip the control vertices of all face-clusters in the hierarchy with so called *topological barycentric coordinates* computed in surface space (see Figure 4, *left*). Note that these coordinates *are different* from the texture space barycentric coordinates defined in Section 3.2 (which will serve for recursive texture mapping) and are related to the topology of the surface mesh, hence their name.

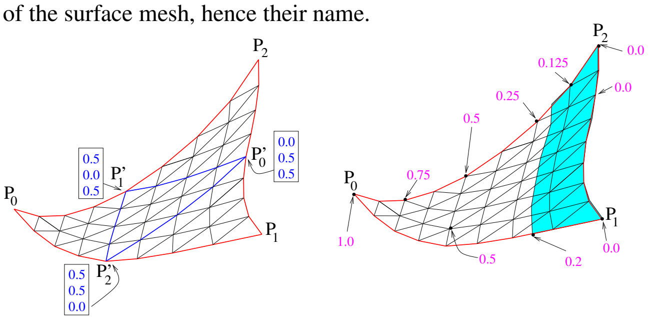


Figure 4: *Left*: *topological coordinates* assigned to control points of a sub-cluster (see text). *Right:* points in the patch for which the *position value* (in magenta) toward the edge is less than 0.2 are kept to obtain the mask (in cyan).

Let us call $C_{topo}^{(l,j)}$ the matrix of topological coordinates for the control vertices of a face-cluster defined with respect to the parent's control vertices. To extract a band along the edge of a given face-cluster, we need such coordinates for all mesh faces (i.e., for control points at the bottom of the hierarchy), but *in the base of the current face-cluster* j_0, of level l_0. Let us call these coordinates $M^{(l,j)}$ for the face-cluster j at level l. We have:

$$M^{(l,j)} = C_{topo}^{(l,j)} \times M^{(l-1,Parent(l,j))} \quad \text{and} \quad M^{(l_0,j_0)} = Identity$$

This gives us a recursive way of computing $M_{maxlevel,j}$.

To extract the band, we then define the *position value* of a vertex with respect to an edge of the face-cluster j_0 as being the sum of its topological barycentric coordinates with respect to j_0, except for the coordinates associated with the two vertices that define the edge. For instance, in Figure 4, *right*, each face has three topological barycentric coordinates respectively associated with P_0, P_1, and P_2; the position value of a vertex with respect to the edge (P_1, P_2) is given by its first coordinate. The band is obtained by selecting the portion of the base mesh triangles for which the position value is less than a given threshold (typically 0.2).

For subdivision surfaces, obtaining topological barycentric coordinates $C_{topo}^{(l,j)}$ with respect to the parent face-cluster is straightforward, since control vertices of the sub-faces of a given face-cluster are always located at the middle of an edge. Values of the *topological coordinates* $C_{l,j}$ are thus set to $\frac{1}{2}$ with respect to the two extremities of the edge and 0 with respect to the other control vertices of the parent face-cluster. For a general face-cluster hierarchy, we keep the same policy for control points located on edges. For other ones, we use a set of coordinates proportional to the respective distances to the control points of the parent face-cluster. This apparently complex solution actually allows us to extract a band from an edge of any face-cluster, regardless of its shape and deformation.

The mask I of the current region is defined as the union of the bands from neighbouring already textured face-clusters. In practice we obtain it using off-screen rendering (see Figure 5,*right*).

4.2 Search for the best match

The second step of texture patch fitting is to look for the portion of the texture image that best matches the mask. We operate entirely in texture space. To do this, we try to find the position and orientation of the mask such that it fits best in the underlying texture. Once this position found, the adequate part of the texture adjacent to the mask is used to texture the current face-cluster.

Best mask position for a given orientation Let $T(x)$, $x \in [0,1]^2$, be the texture sample, first supposed to be toroidal. Let J be the support function of the current mask I, that is, $J(x) = 1$ or 0 depending whether x lies in the union of bands from neighbouring textured patches or not. Note that $J(x)T(x) = I(x)$.

For a given translation x_0 of I over T, we define the mean square error between the mask I and the sample T by:

$$E(x_0) = \sum_x J(x)\left(I(x) - T(x+x_0)\right)^2$$

We are thus looking for points x_0 for which $E(x_0)$ is minimal. We have:

$$E(x_0) = \sum_{J(x)\neq 0} I(x)^2 - 2\sum_{J(x)\neq 0} I(x)T(x+x_0) + \sum_{J(x)\neq 0} T(x+x_0)^2$$

But J cancels in the first two terms since I is supposed to be zero out of the mask support. Moreover, if we denote by $f \diamond g$ the correlation operation[2] between two images f and g, we can write:

$$\begin{aligned} E(x_0) &= \sum_x I(x)^2 - 2(I \diamond T)(x_0) + (J \diamond (T^2))(x_0) \\ &= \sum_x I(x)^2 + \left(-2(I \diamond T) + J \diamond (T^2)\right)(x_0) \qquad (2) \end{aligned}$$

[2] The correlation between two images f and g is defined by: $(f \diamond g)(x_0) = \sum_x f(x)g(x+x_0)$.

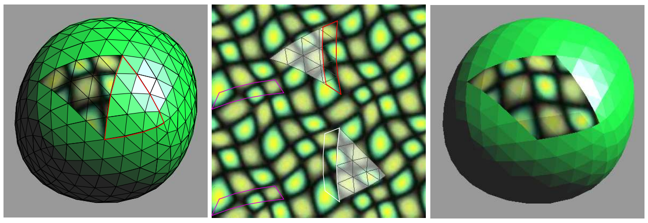


Figure 5: To texture a region that has already textured neighbors (*upper-left, in red*), we build a mask image (*at right*) from the texture of these neighbors (*upper-middle, light red*). We then look over the texture sample for candidate locations that match this mask (*upper-middle, dark red and white*). The best location (*in white*) is used to define the texture of the face-cluster (*upper-right*).

Computing this term directly would lead to a huge amount of computations, since the complexity would be the square of the number N of pixels in the texture. Fortunately, the correlation of two functions f and g can be computed $O(N \log N)$ by moving to Fourier space. We use the fast Fourier transform (FFT) (see [Press et al. 1992]) for computing the Fourier transform $F(f)$ and the conjugate Fourier transform $\overline{F(g)}$. We then have:

$$f \diamond g = F^{-1}(F(f) * \overline{F(g)})$$

Looking at Equation (2), we thus need to compute the Fourier transforms of T, T^2, I and J, build the image of E and look into it for values near 0. Fortunately, T and T^2 being constants, their Fourier transforms need to be computed only once for texturing an entire object.

If the texture sample is not toroidal, the technique above still works. However, the position found must not make the new texture patch cross the border of the image. This reduces the number of possible matches, especially when the mapping scale is large.

Best mask position and orientation The computation above finds the best mask position for a given image orientation. In practice, we are also looking for the best fit through various orientations. For this, we precompute and store the Fourier transforms of rotated versions of T and T^2 for a fixed number of angles (19 in our implementation). During the texturing process, we compute the best position for the mask for each orientation, and select the best result. (A random orientation is used when rendering the mask, in order to avoid getting trivial solutions such as the direct neighbor of a single neighbouring patch). The overall cost of each fitting is still the cost of a single Fourier transform for I and J, so the computation remains efficient.

In practice, all orientations of T and T^2 are precomputed using the graphics hardware. Then we precompute and store their Fourier transforms. During texturing, I and J corresponding to the current region to texture are computed using off-screen rendering of the mask. I is obtained by reading the color values, whereas J comes from the depth values. From these two images we compute $\sum I(x)^2$ and the Fourier transforms $F(I)$ and $F(J)$, and thus get the error function E using equation 2 for all orientations.

The computations have been presented up to now without paying attention to the value domain for functions T and I. We work in *hsv* color space. We find that this gives better results than working in *rgb* space or gray levels, especially for textures like the one in Figure 5. Note that only T, T^2 and I (but not J) need to be turned into *hsv* in equation 2.

4.3 Recursively mapping texture coordinates

From the position of the mask in the texture sample produced by the previous calculation, we deduce the position in texture space of the control polygon of the current face-cluster. We still need to assign texture coordinates to the vertices of the faces of the base mesh that belong to it. This is done by recursively applying equation (1) down to the mesh triangles.

Careful attention must however be paid to the following issues: (1) some vertices of the base-mesh are reached more than once during the recursive traversal of the hierarchy (typically vertices on the boundary of two sub-clusters). Although the coordinates of these vertices are computed using the same original set of coordinates for the cluster at the top of the hierarchy, their final values may differ slightly: due to the nonlinearity of the procedural mapping function defined in section 3.2, their position may depend on the sequence of parents through which they have been computed. This may cause what we call *in-cluster cracks*. (2) this difference of relative position also appears also on vertices of the edges shared with already computed neighbors of the face-cluster. We call these errors *border-cracks*. Both problems are efficiently solved as follows: an array of texture coordinates for the vertex of the base mesh is initialized with zero values; then, the contribution of all vertices is added during the recursive traversal of the hierarchy and averaged to obtain a unique value of texture coordinate for each vertex. This solves the *in-cluster cracks* problem. The second problem is solved by computing, for each shared vertex, the texture coordinates it would take if computed from the neighboring patch positioned exactly next to the current one. These values are used during the recursive traversal of the current face-cluster in replacement of the locally computed values, which exactly adapts the texture deformation to fit the mapped edge of already textured neighbors. The resulting small deformation is automatically distributed inside the face-cluster thanks to the use of the barycentric coordinates.

4.4 Local relaxation along edges and output error

After mapping the texture onto a face-cluster, some discontinuities with the texture on the neighboring face-clusters usually persist. This happens because a perfect solution for matching the mask of the neighbors does not always exist, especially when more than one neighbor is textured. Even a small discontinuity may become noticeable, due to the particular sensitivity of the human eye.

Consequently, we deform, *in texture space*, the edges that are shared with neighboring patches in order to minimize discontinuities along them. This is done by recursively visiting the vertices of sub-clusters located along the edge, and moving them in order to minimize a local error criterion. This local error is defined as the sum of differences per pixel, along the common edge, between the textures of the current and neighbouring patches, weighted by a function representing the influence of the current vertex, depicted in Figure 6, *right*. The search for a new position for each vertex is done using a recursive greedy algorithm which randomly moves the vertex within a smaller and smaller area depending on the hierarchy level. An similar algorithm is also applied to the vertices of the control polygon in order to minimize error at these particular points. In any case, we only move vertices of the current face-cluster, not those of its already textured neighbors.

Note that the side effect of moving the control vertices of a face-cluster in texture space is to deform the texture mapped on the surface. This deformation is seamlessly distributed *inside* the face-cluster thanks to our recursive mapping method.

5 Results

Multi-scale pattern mapping works surprisingly well for a wide variety of textures. Most of the texture samples used in this section

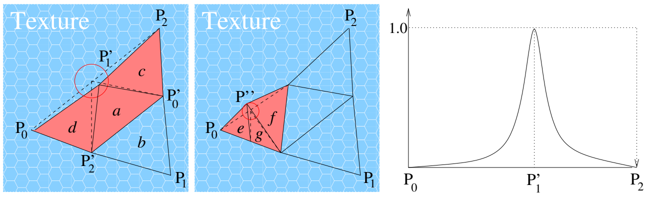


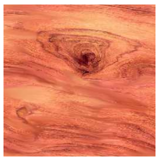
Figure 6: Local relaxation along the edge (P_0, P_2) of a face-cluster: Two successive steps of discontinuity minimization are shown. *Left* : point P'_1 is first allowed to move in the red circle. Once its new position is found, we look for vertices of the next level (*Middle*); P'' is concerned and its new position is searched within a smaller region, and so on. In pink, we show the region of the patch onto which the deformation of the texture will occur due to the vertex displacement. *Right* : weighting function used when measuring the discontinuity at a point with respect to an edge.

were taken from previous work [Efros and Leung 1999; Neyret and Cani 1999; Ashikhmin 2001; Ying et al. 2001; Wei and Levoy 2001; Turk 2001; Efros and Freeman 2001], which simplifies the comparison of results. The table below shows typical computation times.

Object (type)	Polygons	Related figure, and texture size	Time (*min*)
Sphere (S)	3072	Fig. 10, 256×256	1
Bunny (M)	2962	Fig. 9, 128×128	2
Pumpkin (M)	10 000	Fig. 8, 128×128	21
Octopus (S)	34 176	Fig. 9, 256×256	29
Triceratops (M)	5 660	Fig. 10, 128×128	30

Figure 7: Computation times: Types (S) and (M) respectively indicate a subdivision surface or a general mesh.

Using our method is easy: the user just provides a mesh, a texture pattern, and specifies the scale at which the pattern should be mapped, as illustrated in Figure 8. He may also choose the first texture patch for controlling the global orientation of the patterns over the surface. The hierarchical texture mapping computation then takes from a few minutes to a few tens of minutes on standard graphics workstations. Since the texturing method will be able to save time by mapping flat regions at a very coarse scale (if the texture is not too constrained yet), better results are obtained by starting texture mapping in such regions.



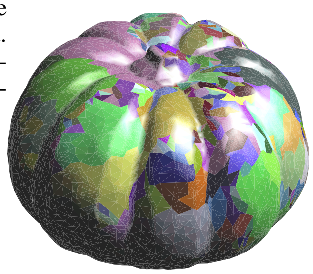


Figure 8: The user controls the scale at which the texture pattern is mapped. *Right* : face-clusters used by the algorithm during the mapping of the top-right pumpkin. Note the appearent continuity of the result, and the various sizes (and hierarchy levels) of the clusters used.

5.1 Isotropic versus anisotropic patterns

The method works with no problem for isotropic patterns such as those used in [Neyret and Cani 1999; Ying et al. 2001]. Results are depicted in figure 9.

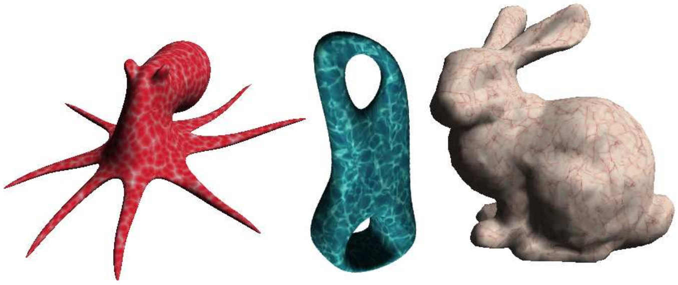


Figure 9: Mapping isotropic texture patterns.

In the case of anisotropic patterns, increasing the width of the band used for texture fitting along patch boundaries may be necessary to better capture and propagate pattern orientation. Once this parameter was tuned, we had no problem generating textures such as bark, wood, or text with a coherent orientation. See Figure 10.

Figure 10: Our method succeeds in capturing and propagating the main orientation of anisotropic texture patterns.

5.2 Highly structured patterns

Our method is particularly adapted to highly structured patterns. Some results are depicted in Figure 11.

As with previous methods [Neyret and Cani 1999; Efros and Freeman 2001], the minimal size of the texture patches should not

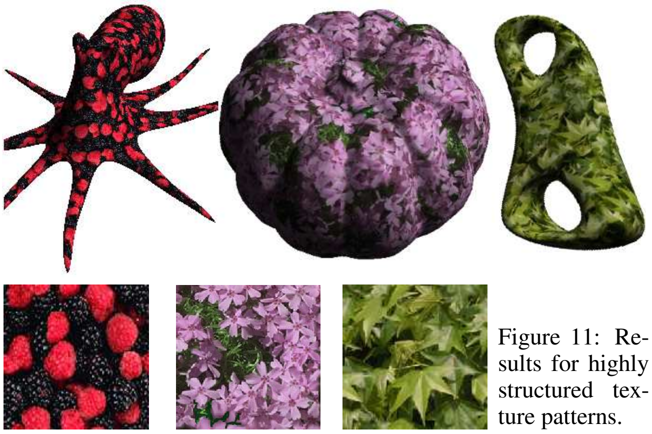


Figure 11: Results for highly structured texture patterns.

be smaller than the pattern size, or we may loose the pattern structure. A solution to this problem, which would occur if we mapped the pattern at a very large scale relatively to the surface geometry, is suggested in the next section.

6 Conclusion

The hierarchical texture mapping algorithm we have introduced demonstrates that an arbitrary surface can be textured by directly mapping patches of various sizes, shapes and orientations from an input image. This new method combines many advantages: The texture patches mapped at a large scale propagate the global structure of the pattern, while texturing at a smaller scale adequately fills the gaps and reduces texture distortion over sharp geometric features. Combining these different scales increases efficiency, since texture mapping at the finest scale will only be done in specific sparse regions. Moreover, the algorithm directly outputs texture coordinates for the triangles of the initial mesh, which minimizes memory requirement (no new texture needs to be stored) and eases rendering (which makes it preferable to methods like lapped textures at equivalent image quality). Lastly, the approach works for a wide range of textures, regardless of their isotropy, frequency of variation and inner structure.

The method will generally leave small texture fitting errors across patch boundaries. For many applications, this is not a problem: just as with surface approximations using triangles, the textures we generate should be viewed from at least a given minimal distance. Texture discontinuities will indeed appear in very close views, together with other artifacts such as tangent discontinuities due to the discretization into triangles. If we need to avoid these discontinuities, an idea would be to refine the mesh using a subdivision scheme, in order to map texture on sub-triangles. However, using smaller texture patches is not sufficient for improving the visual quality of the mapping: while doing so, the low frequency structure of the texture patterns may be lost. A solution would then be to combine our multi-scale mapping algorithm with the texture transfer methods of Efros [Efros and Freeman 2001] and Hertzmann [Hertzmann et al. 2001]. The texture found to fit badly at a large scale would serve as a reference image to guide texture mapping at a finer scale, thus maintaining the global coherency of the patterns. Implementing this extension would only demand a change to the error measure, in order to take the reference image into account.

Another idea would be to rely on texture transfer techniques for offering more user control, as first suggested by Ashikhmin [Ashikhmin 2001]. The user would paint the desired average colors directly on the initial mesh, and texture mapping would follow these choices by picking the appropriate texture patches in the sample image.

Another extension of our method would consist of locally controlling the scale of the texture along the surface, as many natural objects tend to produce a similar patterns at multiple scales. Also, mapping more than one texture while maintaining smooth transitions would further enrich the method.

7 Acknowledgements

Special thanks to John F. Hugues for re-reading the last version of the paper and for his suggestions about possible improvements. Thanks also to Rob Jagnow for the pumpkin model and to the reviewers for their comments.

References

ASHIKHMIN, M. 2001. Synthesizing natural textures. *2001 ACM Symposium on Interactive 3D Graphics* (March), 217–226. ISBN 1-58113-292-1.

BENNIS, C., VÉZIEN, J.-M., IGLÉSIAS, G., AND GAGALOWICZ, A. 1991. Piecewise surface flattening for non-distorted texture mapping. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, T. W. Sederberg, Ed., vol. 25, 237–246.

EBERT, D., MUSGRAVE, F., PEACHEY, D., PERLIN, K., AND WORLEY, S., Eds. 1998. *Texturing and Modelling: A procedural approach*. Morgan Kaufmann Publishers.

ECK, M., DEROSE, T., DUCHAMP, T., HOPPE, H., LOUNSBERY, M., AND STUETZLE, W. 1995. Multiresolution analysis of arbitrary meshes. In *SIGGRAPH 95 Conference Proceedings*, Addison Wesley, R. Cook, Ed., ACM SIGGRAPH, 173–182.

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

EFROS, A., AND LEUNG, T. 1999. Texture synthesis by non-parametric sampling. In *International Conference of Computer Vision*, vol. 2, 1033–1038.

GARLAND, M., WILLMOTT, ., AND HECKBERT, P. 2001. Hierarchical face clustering on polygonal surfaces. In *ACM Symposium on Interactive 3D Graphics*.

HERTZMANN, A., JACOBS, C. E., OLIVER, N., CURLESS, B., AND SALESIN, D. H. 2001. Image analogies. *Proceedings of SIGGRAPH 2001* (August), 327–340. ISBN 1-58113-292-1.

LÉVY, B., AND MALLET, J.-L. 1998. Non-distorted texture mapping for sheared triangulated meshes. *Proceedings of SIGGRAPH 98* (July), 343–352. ISBN 0-89791-999-8. Held in Orlando, Florida.

LÉVY, B. 2001. Constrained texture mapping for polygonal meshes. *Proceedings of SIGGRAPH 2001* (August), 417–424. ISBN 1-58113-292-1.

MAILLOT, J., YAHIA, H., AND VERROUST, A. 1993. Interactive texture mapping. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, J. T. Kajiya, Ed., vol. 27, 27–34.

NEYRET, F., AND CANI, M.-P. 1999. Pattern-based texturing revisited. *Proceedings of SIGGRAPH 99* (August), 235–242.

PRAUN, E., FINKELSTEIN, A., AND HOPPE, H. 2000. Lapped textures. *Proceedings of SIGGRAPH 2000* (July), 465–470. ISBN 1-58113-208-5.

PRESS, TEUKOLSKI, VETTERLING, AND FLANNERY. 1992. *Numerical Recipes in C*. Cambridge University Press.

TURK, G. 1991. Generating textures for arbitrary surfaces using reaction-diffusion. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, T. W. Sederberg, Ed., vol. 25, 289–298.

TURK, G. 1992. Re-tiling polygonal surfaces. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, E. E. Catmull, Ed., vol. 26, 55–64.

TURK, G. 2001. Texture synthesis on surfaces. *Proceedings of SIGGRAPH 2001* (August), 347–354. ISBN 1-58113-292-1.

WALTER, M., FOURNIER, A., AND MENEVAUX, D. 2001. Integrating shape and pattern in mammalian models. *Proceedings of SIGGRAPH 2001* (August), 317–326. ISBN 1-58113-292-1.

WEI, L.-Y., AND LEVOY, M. 2000. Fast texture synthesis using tree-structured vector quantization. *Proceedings of SIGGRAPH 2000* (July), 479–488. ISBN 1-58113-208-5.

WEI, L.-Y., AND LEVOY, M. 2001. Texture synthesis over arbitrary manifold surfaces. *Proceedings of SIGGRAPH 2001* (August), 355–360. ISBN 1-58113-292-1.

YING, L., HERTZMANN, A., BIERMANN, H., AND ZORIN, D. 2001. Texture and shape synthesis on surfaces. In *Eurographics Rendering Workshop 2001*, Springer Wein, S. Gortler and K. Myszkowski, Eds., Eurographics, 301–312.