Mesh Quilting For Geometric Texture Synthesis

Kun Zhou∗ Xin Huang∗ Xi Wang∗ Yiyung Tong† Mathieu Desbrun† Baining Guo∗ Heung-Yeung Shum∗

∗Microsoft Research Asia †Caltech

Abstract

We introduce mesh quilting, a geometric texture synthesis algorithm in which a 3D texture sample given in the form of a triangle mesh is seamlessly applied inside a thin shell around an arbitrary surface through local stitching and deformation. We show that such geometric textures allow interactive and versatile editing and animation, producing compelling visual effects that are difficult to achieve with traditional texturing methods. Unlike pixel-based image quilting, mesh quilting is based on stitching together 3D geometry elements. Our quilting algorithm finds corresponding geometry elements in adjacent texture patches, aligns elements through local deformation, and merges elements to seamlessly connect texture patches. For mesh quilting on curved surfaces, a critical issue is to reduce distortion of geometry elements inside the 3D space of the thin shell. To address this problem we introduce a low-distortion parameterization of the shell space so that geometry elements can be synthesized even on very curved objects without the visual distortion present in previous approaches. We demonstrate how mesh quilting can be used to generate convincing decorations for a wide range of geometric textures.

Keywords: texture synthesis, geometric detail, shell mapping

1 Introduction

Caught between the need for ever richer computer-generated scenes and the hardware limit of polygon throughput, early computer graphics researchers developed texture mapping as an efficient means to create visual complexity while maintaining the geometric complexity to a reasonable level. More general forms of textures, such as bump mapping and volumetric textures, were introduced to palliate the artifacts of image texturing, while still eliminating the tedium of modeling and rendering every 3D detail of a surface. However, the graphics processor on today’s commodity video cards has, over the past five years, significantly increased the ease of defining complex image-based details on arbitrary meshes. However, manipulating and animating the content of these volumetric textures can be particularly delicate. Recently, [Elber 2005; Potrumbescu et al. 2005] advocated a more versatile representation for geometric detail modeling using mesh-based details. Both method simply tiles textures over the plane and then maps the textures to 3D surfaces. However, this process is limited to periodic textures and most importantly it will produce texture discontinuity across chart boundaries since arbitrary surfaces do not have a global parameterization over the plane. Although a few other papers have proposed such a mesh-based creation of geometric textures on arbitrary meshes (see [Fleischer et al. 1995]), they are mostly restricted to the dissemination of simple texture elements over the surface, like scales or thorns, and do not allow the design of woven materials for instance.

Example-based Texture Synthesis Texture synthesis on surfaces has, over the past five years, significantly increased the ease of designing complex image-based details on arbitrary meshes. A first category of algorithms [Turk 2001; Wei and Levoy 2001; Ying et al. 2001; Tong et al. 2002; Zelinka and Garland 2003] achieved such a synthesis based on per-pixel non-parametric sampling [Efros and
Leung 1999; Wei and Levoy 2000]. Based on the $L^2$-norm, a relatively poor measure of perceptual similarity, such algorithms are not applicable to a large spectrum of textures. The second category of algorithms synthesizes textures by directly copying small parts of an input texture sample (i.e., a swatch), an approach more amenable to all types of texture input. Earlier algorithms randomly paste patches and use alpha-blending to hide patch seams [Praun et al. 2000]. Recently, quilting [Efros and Freeman 2001; Liang et al. 2001; Soler et al. 2002; Magda and Kriegman 2003; Kwatra et al. 2003; Wu and Yu 2004; Zhou et al. 2005] generate significantly better results by carefully placing patches to minimize the discontinuity across patch seams. After placing those patches, [Liang et al. 2001; Magda and Kriegman 2003] simply use alpha-blending to hide patch seams, while [Efros and Freeman 2001; Kwatra et al. 2003; Zhou et al. 2005] further enhance the smoothness across the seams by searching for the “min-cut” seams. Since our human visual system is so sensitive to edges, corners and other high-level features in textures, [Wu and Yu 2004] proposes instead to extract a feature map from the sample texture, and perform feature matching and deformation to best keep the integrity of features. Other relevant texture synthesis techniques include an hybrid approach combining pixel-based and patch-based schemes [Nealen and Alexa 2003], texturing objects in photographs [Fang and Hart 2004], parallel controllable texture synthesis on GPU [Lefebvre and Hoppe 2005] and texture synthesis using Expectation Maximization optimization [Kwatra et al. 2005].

Shell Space Parameterization If a geometric texture given as a mesh over a planar surface needs to be mapped to a 3D curved surface, a one-to-one mapping between the geometric texture and the shell space (i.e., the thin layer around the curved surface) can be defined. Unlike previous methods which implicitly construct the shell space using normal displacement, [Peng et al. 2004] define a point-to-surface function to generate displacements for vertices. The shell map approach [Porumbescu et al. 2005] proposes to tetrahedralize both the shell space and texture space to build a bijective mapping between the two spaces using barycentric coordinates of the corresponding tetrahedra. However, as admitted by the authors as a limitation, shell mapping is very sensitive to the height of the shell as it can introduce significant distortion resulting in noticeable artifacts.

1.2 Challenges

Despite fast progress in example-based image texture synthesis, there has been very little work to provide a similar tool for 3D geometric texture. To the best of the authors’ knowledge, only two papers [Bhat et al. 2004; Lagae et al. 2005] explored this issue recently. However, both methods dealt with a more restrictive representation than meshes. [Bhat et al. 2004] used a voxel-based approach and [Lagae et al. 2005] used distance fields. Providing a mesh quilting technique that seamlessly applies a 3D texture sample (given as a triangulated mesh) inside a thin shell around an arbitrary surface presents multiple hurdles. First, the input texture sample is not a regular array of pixel values but an irregular mesh given by vertex positions and connectivity; making adaptation of image quilting a difficult task. Second, our texture sample consists of geometry elements, each being truly a small 3D object identified as a connected component in 3D, and we wish to maintain the integrity of these geometry elements in the synthesized geometric texture to facilitate subsequent applications such as interactive editing and animation. Finally, when mesh quilting is performed on curved surfaces, geometry elements often exhibit severe distortion in the 3D space within the shell and a shell mapping procedure to minimize distortion is essential for generating visually-pleasing geometry textures.

1.3 Contributions

We built upon the traditional texture synthesis framework to design a mesh-based geometric texture synthesis technique to decorate arbitrary meshes: from a base mesh and a given 3D texture swatch, a geometric texture, locally similar to the swatch everywhere, is synthesized over the base mesh. In contrast to previous volumetric texture methods, a unique feature of mesh quilting is that both the input geometry and output geometry are represented by triangle meshes. Moreover, our technique maintains the integrity of geometry elements in the synthesized texture so that subsequent texture editing and texture animation can be easily performed. For stitching together geometry elements our algorithm finds corresponding elements in adjacent texture patches, aligns elements through local deformation, and merges elements to connect texture patches. This explicit manipulation of geometry elements distinguishes our technique from image texture synthesis, which strives to maintain image features implicitly through manipulation of image pixel values. For mesh quilting on curved surfaces, we introduce a low-distortion parameterization of the shell space so that geometry elements can be synthesized without the visual distortion present in previous approaches.

2 Mesh Quilting Synthesis

As already proposed in the 2D case for texture images, an interesting design tool to generate geometric details on a surface is to use a swatch of geometry texture (a mesh representing a portion of wicker or rattan fabric, or a section of a wall-climbing vine with its foliage, see Figure 10) and create an entire “shell volume” by repeating and stitching this swatch in a visually seamless manner. Alas, such a geometry texture synthesis cannot proceed like its 2D counterpart: the domain to synthesize upon is non-flat, and the swatch is an irregular, potentially high-genus mesh—not a regular array of pixels. In this section, we present the details of our algorithm to synthesize such a swatch-based texture. We will only consider a planar extension of the swatch first; its extension to curved surfaces will be presented in a subsequent section.

2.1 Setup & Nomenclature

Let $M_{\text{in}} = \{V_{\text{in}}, F_{\text{in}}\}$ be the input sample mesh of geometry texture, where $V_{\text{in}}$ is the set of vertex positions and $F_{\text{in}}$ is the set of triangles. Its bounding box has dimensions $l_{\text{in}} \times w_{\text{in}} \times h_{\text{in}}$, where $h_{\text{in}}$ is the smallest dimension (that is to say, the thickness of the shell). We wish to synthesize a larger mesh $M_{\text{out}}$ with bounding box $l_{\text{out}} \times w_{\text{out}} \times h_{\text{in}}$ (that we will denote as the output texture space) by growing outward the original mesh sample in order to create a seamless geometric pattern.
Each connected component of \( M_{in} \) or \( M_{out} \) will be referred to as a geometric element and we will explain how mesh quilting synthesis can preserve the integrity of these texture elements. Notice the similarity to previous work [Dischler et al. 2002; Zhang et al. 2003] that show that image texture synthesis can benefit greatly if salient 2D texture elements can be extracted from the input sample and kept intact during texture synthesis. Unfortunately, these methods rely on the user’s manual labor for finding image features, while our method is fully automated.

### 2.2 Algorithm Overview

Our mesh quilting synthesis proceeds in steps similar in spirit to the patch matching and patch fitting stages of [Kwatra et al. 2003]. More precisely, the output texture space is filled progressively by iterating the following actions:

1. **Seed Finding**: Find a seed region \( R \) from which to grow the output mesh texture further out.
2. **Geometry Matching**: Find the best patch placement around region \( R \) using geometry matching to minimize mismatch between the new and the old patch.
3. **Element Correspondences**: Find correspondences between elements in the new patch and those in the old patch.
4. **Element Deformation**: Align the corresponding elements through local deformation.
5. **Element Merging**: Expand the output texture by merging the new patch into the output texture space.

### 2.3 Seed Finding

Traditional texture synthesis algorithms often proceed by growing a new texture patch out with a reference texture swatch, until all pixels of the patch have been processed. Alas, mesh textures have an irregular connectivity, so finding where the output mesh \( M_{out} \) needs to be extended and when the process is completed requires special care. In order to both closely mimic previous algorithms and reduce computational cost, we also use a grid-based approach. The bounding boxes of both \( M_{out} \) and \( M_{in} \) are subdivided in finer regular grids, of the same grid cell size, and each triangle of these two meshes is assigned to the grid cells containing it. Note that these grids are only two-dimensional: there is no need for subdividing the height of the space. Initially, the cells of \( M_{out} \) are tagged unprocessed. Then, each time we wish to grow out the current mesh \( M_{out} \), we look for an unprocessed cell with the largest number of adjacent cells that are already processed: this will be the seed cell that we will try to process next. Notice that this cell is selected so that it already contains some nearby patches of the input texture: this will ensure that our mesh texture will be extended in a manner that is consistent with the already processed portions of the output mesh texture \( M_{out} \).

### 2.4 Geometry Matching

We now need to find how to complete the mesh texture in the seed cell, and possibly add to its surroundings too. Using the nearby existing mesh texture available near the seed cell, we want to find a portion of the original swatch \( M_{in} \) best matching this surrounding to extend \( M_{out} \). To find the best placement of the swatch over the seed cell and its surroundings, we employ an approach similar to the sub-patch matching technique presented in [Kwatra et al. 2003].

We first pick a small output-sub-patch \( P_{out} \) (which consists of a set of grid cells around the seed cell, but is smaller than the input texture). We now look for translations of the input mesh texture such that the portion of the input overlapping the output-sub-patch matches it best. Of course, only translations that allow complete overlap of the input with the output-sub-patch are considered. Let \( M_{in}(t) \) be the input geometry texture translated by \( t \). We compute the matching cost as the sum of distances between the output geometry \( M_{out} \) and the input \( M_{in}(t) \) within the overlapping region. Suppose \( \mathbf{f}_{in}^{j} \) is a face of \( M_{in}(t) \). For each vertex \( \mathbf{v}_{out} \) of the output mesh \( M_{out} \) in the output-sub-patch we define the “distance” between \( \mathbf{v}_{out} \) and \( \mathbf{f}_{in}^{j} \) as a combination of geometric distance and normal difference:

\[
D(\mathbf{v}_{out}, \mathbf{f}_{in}^{j}) = (1 + \lambda \text{Dist}(\mathbf{v}_{out}, \mathbf{f}_{in}^{j}))(1 + ||\mathbf{n} \cdot (\mathbf{v}_{out} - \mathbf{f}_{in}^{j})||),
\]

where \( \text{Dist}(\mathbf{v}_{out}, \mathbf{f}_{in}^{j}) \) is the shortest distance between \( \mathbf{v}_{out} \) and triangle \( \mathbf{f}_{in}^{j} \), \( \mathbf{n} \cdot (\cdot) \) is the mesh normal and \( \lambda \) is the weighting parameter (we set it to 1 for all examples in this paper). The matching cost of \( \mathbf{v}_{out} \) with respect to \( M_{in}(t) \) is then defined as the smallest \( D(\mathbf{v}_{out}, \mathbf{f}_{in}^{j}) \):

\[
E(\mathbf{v}_{out}, M_{in}(t)) = \min_{f_{in}^{j}} D(\mathbf{v}_{out}, f_{in}^{j}).
\]

The face with the smallest value is denoted as \( f_{in}^{j} \). Now we are ready to compute the global matching cost for translation \( t \):

\[
E(t) = \sum_{v_{out} \in P_{out}} E(\mathbf{v}_{out}, M_{in}(t)).
\]

Ideally, this cost should be minimized over all allowed translations, leading to impractical computational time. Therefore, we restrict the translation \( t \) to be in grid unit. Such discrete translations are enough for finding a good patch placement. One could use finer discretization to get a better placement; but the element deformation described in Section 2.6 will also compensate for an imperfect element alignment, and indeed, our tests show that this algorithmic optimization is well justified. Additionally, we obtained a significant speed-up factor for the matching cost computations by building an octree data structure for the input texture, as distances between vertices and faces can be more efficiently computed.

### 2.5 Element Correspondences

Once the best patch placement is found, we are ready to build the correspondences between the output elements and the input elements within the overlapping region. Note that the overlapping region is usually larger than the small sub-patch \( P_{out} \) since the input mesh texture covers \( P_{out} \) completely.
We first compute the “nearest” face $f^i_{in}$ (for the distance function defined in Equation 1) for each vertex $v^i_{out}$ in the overlapping region and collect them together as a set of vertex-face pairs ($S = \{(v^i_{out}, f^i_{in})\}$). We then prune the pairs that obviously indicate poor matching: vertex-face pairs are removed if the normals of the face and its corresponding vertex normals are opposite; vertex-face pairs with a distance much larger than the local edge length average of the input mesh are also dismissed.

For the remaining pair set $S$, we tag an output element $C_{out}$ as “related” to an input element $C_{in}$ if there exists a vertex-face pair $(v^i_{out}, f^i_{in}) \in S$ such that $v^i_{out} \in C_{out}$ and $f^i_{in} \in C_{in}$. This simple test can, however, create false correspondence: an element $C_{out}$ may be tagged as related to an element $C_{in}$ even if they are far away. To remove such irrelevant correspondences, we project the triangles of both elements onto the plane and check whether the two projections overlap: if they do not overlap, we can safely remove the relationship between the two elements and the vertex-face pairs belonging to these two elements is subsequently removed from $S$.

Finally, for each output element within the overlapping region, we find a set of input elements to which it corresponds.

### 2.6 Element Deformation

For each output element $C_{out}$ corresponding to an input element $C_{in}$, we now try to deform them both slightly in order to better align them: in addition to improving the geometric alignment between the input swatch and the current output mesh, this will also go a long way in providing a smooth and visually seamless extension of the output element.

Our deformation method is built upon recent Laplacian-based mesh editing techniques [Yu et al. 2004; Sorkine et al. 2004] to best satisfy positional constraints while preserving local geometric details.

For every pair $(v^i_{out}, f^i_{in}) \in S$, we will call $(v^1_{in}, v^2_{in}, v^3_{in})$ the three vertices of face $f^i_{in}$, while:

$$ h^i_{in} = \alpha^1 v^1_{in} + \beta^1 v^2_{in} + \gamma^1 v^3_{in} $$

will denote the closest point to $v^i_{out}$ in $f^i_{in}$. We also compute the Laplacian coordinates for all vertices:

$$ \mathcal{L}(v^i) = v^i - \frac{1}{\#N(v^i)} \sum_{v^j \in N(v^i)} v^j, \quad (4) $$

where $N(v^i)$ is the 1-ring vertex neighbors of vertex $v^i$, and $\#$ indicates its cardinality.

We wish to compute new positions $\{p^j_{out}\}$ ($\{p^j_{in}\}$) for vertices in $C_{out}$ ($C_{in}$). First we get the position constraints by computing the average points of the vertex-face pairs: $c^i = (v^1_{out} + v^2_{out} + v^3_{out})/3$. The deformation should satisfy the position constraints while preserving the local geometry details (i.e. Laplacian coordinates). For the output element $C_{out}$, this goal can be achieved by solving the following quadratic minimization problem:

$$ E_{out}(\{w^i\}) = \sum_{i=1}^{N_{out}} \|\mathcal{L}(w^i) - \mathcal{L}(v^i_{out})\|^2 + \mu \sum_{i=1}^{m} \|w^i - c^i\|^2, \quad (5) $$

where the parameter $\mu$ balances the two objectives and is set to 1 by default.

Similarly, the vertices of $C_{in}$ can be deformed by finding the positions $\{w^i\}$ minimizing the following energy:

$$ E_{in}(\{w^i\}) = \sum_{i=1}^{N_{in}} \|\mathcal{L}(w^i) - \mathcal{L}(v^i_{in})\|^2 + \mu \sum_{i=1}^{m} \|w^i - c^i\|^2 + \alpha \langle w^i - c^i, 1 \rangle^2 + \beta \langle w^i - c^i, 2 \rangle^2 + \gamma \langle w^i - c^i, 3 \rangle^2. \quad (6) $$

The above deformation energies can be adapted to elements with multiple corresponding elements by collecting all position constraints together. We found that maintaining the original Laplacian coordinates instead of using a transformed Laplacian coordinates as in [Yu et al. 2004; Sorkine et al. 2004] works well enough in our context since we only have to deal with small deformations to get a better element alignment.

### 2.7 Element Merging

We are now ready to piece elements together to extend the current output mesh. First, every element (either from $C_{out}$ or $C_{in}$) without correspondence is directly added to $M_{out}$. For every established correspondence ($C_{out}.C_{in}$), the merging proceeds as follows. If $C_{out}$ is entirely within the overlapping region, $C_{out}$ is ignored and $C_{in}$ is instead added to the final results. Similarly, if $C_{in}$ is entirely within the overlapping region, $C_{in}$ is ignored and $C_{out}$ is added to $M_{out}$.

In all other cases, we need to stitch parts of $C_{in}$ and $C_{out}$ to get a singly-connected, combined element, and we add it to $M_{out}$. To smoothly stitch two partially overlapping elements $C_{out}$ and $C_{in}$ together, we first seek a cut path in each element such that the two cut paths are close to each other: these paths can be found using the graph cut algorithm presented in [Boykov et al. 2001] using the following approach. We first build an undirected flow network graph for $C_{out}$ representing the dual graph adjacency between triangles. The weights of this graph are set as follows: for two adjacent triangles sharing an edge $(v^i_{out}, v^j_{out})$, a weight of

$$ (1 + \|v^i_{out} - v^j_{out}\|)(1 + Dist(v^i_{out}, C_{in}) + Dist(v^j_{out}, C_{in})), \quad (7) $$

is assigned to the graph edge, where $Dist(v^i_{out}, C_{in})$ is the shortest distance from $v^i_{out}$ to $C_{in}$. Two additional nodes are added, representing the two possible choices for triangles, deleted (SINK) or undeleted (SOURCE). Triangles lying outside the overlapping region are linked to SOURCE by an edge with infinite weight, to guarantee that those triangles will not be deleted. Suppose that a vertex $v^i_{out}$ in $C_{out}$ has a closest face $f^i_{in}$ in $C_{in}$. If $f^i_{in}$ lies outside of the overlapping region or there exists a face which is adjacent to $f^i_{in}$ and does not have any corresponding vertices in $C_{out}$, then all triangles sharing vertex $v^i_{out}$ are linked to SINK with infinite weight.
to guarantee this time that these triangles will be deleted. Applying
the graph cut optimization algorithm [Boykov et al. 2001] to the
constructed graph will provide a min-cost cut which separates $C_{\text{out}}$
into disconnected parts: triangles linked to SOURCE are kept while
those linked to SINK are simply deleted. A cut path for $C_{\text{in}}$ is found
using the exact same approach.

Stitching together the two cut elements is finally performed through
mesh merging [Yu et al. 2004]. We simply set the average boundary
points as position constraints and deform the two cut elements using
the deformation energy defined in Equation 5. The mesh connec-
tivity of the elements is then updated to create a single connected
component.

2.8 Results

Even though the technique described above is only valid for planar
design, several interesting effects can already be obtained. Figure 4
demonstrates how a small swatch representing a few apples can
generate a whole, seemingly non-repetitive layer of apples; it also
shows the more challenging case of a non-tileable weave swatch,
creating long thread-like elements interwoven in the output mesh.

3 Mesh Quilting Over Curved Surfaces

Extending mesh quilting synthesis to be applicable to curved sur-
faces in 3D requires further work. In this section, we describe how
a seamless quilting can be obtained using local surface parameter-
izations and, optionally, a guidance vector field, before embedding
the resulting mesh into shell-space.

3.1 Geometric Texture Synthesis on Surfaces

Setup Let $M_{\text{base}}$ be the base mesh that we wish to enhance with
added geometric details. We still denote as $M_{\text{in}}$ the geometric tex-
ture mesh used as a swatch that we wish to seamlessly tile the base
mesh with. A parameter $s$ is also provided to allow the user to
specify the relative size of the input texture with respect to the base
surface, i.e., to choose the scale of the geometric details.

From Planar to Curved The algorithm presented in Section 2 re-
quires several modifications to accommodate curved domains.

First, the 2D grid we used in the planar case is easily replaced
by the base mesh itself: the quilting process will stop only when
there are no more unprocessed triangles. Similar to the 2D case,
we pick the most constrained un-synthesized triangle, i.e., the one
with most triangles synthesized in the neighborhood. We define a
local surface patch by starting from the chosen triangle and grow-
ing the region using breadth-first traversal until we reach a certain
depth or when the total area of the patch exceeds a user-defined
threshold.

Additionally, the position of vertices should no longer be put in a
global coordinate system: they should, instead, be located with re-
spect to the base mesh itself. Consequently, the coordinates of the
vertices of the texture output mesh are stored as follows: the loca-
tion of a vertex $v$ over a triangle $T_{\text{base}}$ is defined by the barycentric
coordinates of its orthogonal projection on $T_{\text{base}}$ along with the
orthogonal distance (i.e., height) from the triangle to $v$.

The surface patch is flattened over the 2D plane using a discrete
conformal mapping (DCM [Desbrun et al. 2002] or equivalently,
LSCM [Levy et al. 2002]). Based on this parameterization, we
can convert the local mesh-based representation of the part of
$M_{\text{out}}$ inside this patch into an absolute representation as in the 2D
case. Finally, the local operations described for planar mesh quilting
can be performed over this parameterization plane, then the position
of the newly synthesized vertices will be reprojected onto
the local mesh-based coordinate system described above. Note
that the geometry matching step will still restrict its search to dis-
crete translations in this parametric domain to keep the matching
cost computations to a minimum.

Finally, we also need to accommodate for the distortion caused
by DCM in very curved regions. In our implementation, if the
area distortion induced by the local parameterization is too large
(above a factor of 4), we reduce the area of the surface patch: this
will, in turn, decrease the size of the output-sub-patch $P_{\text{out}}$.

Using Guidance Vector Fields One of the major differences be-
tween synthesis on a planar region and synthesis on a curved sur-
face is that it may often be necessary to control the orientation of
the geometric texture over the surface when the swatch contains
obvious privileged directions. We thus allow the user to specify
a vector field in order to control the direction of synthesis [Praun
et al. 2000]. We can use this field to align the direction of the grid
(see Section 2.3) in the shell space.

3.2 Final Mesh Embedding

Using the above method, we can automatically generate highly de-
tailed geometric textures on meshes. However, harnessing the po-
tential of such a representation requires one final step to convert the
generated details into a proper mesh: we must convert the vertex
positions, stored in local coordinates for now, into a stand-alone,
common embedding. A simple conversion to $\mathbb{R}^3$ is, alas, not suffi-
cient: self-intersections can be created in regions of high concavity
since the local coordinate frames of two adjacent triangles forming
a concave angle may overlap (Figure 8, left and middle). Instead,
we build a texture atlas for $M_{\text{base}}$, and convert the above local rep-
resentation of vertex positions to locations in a geometry texture
space (see Figure 6). Then we construct a shell space around $M_{\text{base}}$,
i.e., a thin volume between $M_{\text{base}}$ and one offset of it. Finally, map-
ing the vertices from the geometry texture space to the shell space
will fix the location of the vertices in 3D space, thus turning $M_{\text{out}}$
into a properly embedded mesh (see Figure 8, right). To minimize

Figure 5: Mesh quilting results on curved surface: (a) from a weave struc-
ture, our approach synthesizes the structure over a cup base mesh; (b) a
Venus model is densely covered with nut elements.
the distortion introduced in this step, we propose next to design a stretch-minimized shell mapping.

**Shell Mapping** Defining a thin volume around an arbitrary mesh has been proposed as an easy way to model geometric texture [Peng et al. 2004; Elber 2005]. Porumbescu et al. [2005] even proposed a simple mapping between shell space and texture space for an arbitrary surface patch by meshing the shell space with tetrahedra and using barycentric coordinates. Unfortunately, the authors also pointed out that such a mapping design systematically creates large distortion in curved regions (see Figure 8(left) where the flower patterns are significantly enlarged). In this section, we alleviate this issue by optimizing a *stretch metric* on this tetrahedral mesh—a natural extension of low-distortion parameterization of triangle meshes [Sander et al. 2001].

**Stretch Metric on Tetrahedra** A shell map defines a piecewise-linear, bijective mapping between shell space and texture space based on barycentric coordinates (see Figure 3 in [Porumbescu et al. 2005]). However, as often happens for piecewise-linear mapping between a triangle mesh and its parameterization, this bijection can have significant stretching in certain regions if no special care is taken. To limit this occurrence, one can tweak the texture space coordinates in order to minimize a distortion measure.

Let \( g \) be the shell mapping defined between a point in shell space \((x, y, z)\) (inside a tetrahedron \( T_0 = (v_1, v_2, v_3, v_4) \)) to a point in texture space \((u, v, w)\) (inside a tetrahedron \( T'_0 = (q_1, q_2, q_3, q_4) \)). Due its piecewise linear nature, the Jacobian of \( g \) (i.e., the deformation gradient) \( J = [\frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}, \frac{\partial v}{\partial z}] \) is constant over \( T'_0 \). Let \( \pi_1, \pi_2 \) and \( \pi_3 \) denote the three eigenvalues of the Cauchy deformation tensor \( J^T J \), representing the principal length dilatation factors (called *stretch*). The root-mean-square stretch over all directions can now be computed as:

\[
L^2(g, T'_0) = \sqrt{(\pi_1 + \pi_2 + \pi_3)/3} = \sqrt{(a_x + b_y + c_z)/3}, \quad (8)
\]

with \( a_x = \frac{\partial x}{\partial v}, b_y = \frac{\partial y}{\partial v}, c_z = \frac{\partial z}{\partial v} \).

Assuming that the shell space consists of tetrahedra \( \{T'_j\} \), the total \( L^2 \) stretch is then:

\[
L^2(g, M) = \sqrt{\sum_j (L^2(g, T'_j))^2 |T'_j|/\sum_i |T'_i|} \quad (9)
\]

where \( |T'_i| \) is the volume of tetrahedron \( T'_i \) in the shell space corresponding to the tetrahedron \( T'_j \) in texture space. As indicated in [Sander et al. 2001], the \( L^2 \)-stretch value can be further normalized by multiplying it by \( \sqrt{\sum k |T'_k|/\sum_m |T'_m|} \) such that 1.0 is a lower bound for the stretch value.

**Minimization Algorithm** To minimize \( L^2(g, M) \), we start with the initial shell map and perform several optimization iterations to min-
Figure 8: Shell embedding on a curved surface. Left: using normal displacement; Middle: using original shell mapping; Right: our low distortion shell mapping. Notice the even scale of the flower patterns in our result, as well as the intersection-free behavior in concave regions.

Mesh quilting has similar limitations to traditional 2D texture synthesis algorithms. First, since mesh quilting on surface depends on local parameterization of surface patches, regions with very high curvature (as in high-genus, complex models) can be badly handled since the parametric distortion of small surface patches may be high. Another issue is that our algorithm cannot always achieve perfect matching if the swatch is untileable, even with major element deformation; in that case, the integrity of the geometry elements can sometimes not be established. When this happens, a postprocessing step is performed to remove those visually-displeasing elements. This was done in the example with nuts synthesized on the Venus model shown in Figure 5(b).

5 Conclusions

We presented the first mesh-based 3D texture synthesis algorithm on arbitrary manifolds. As such, there is no doubt that several refinements can be provided to further increase the possibilities offered by such a technique. We believe however that our mesh quilting approach offers a good foundation for detail synthesis. Similarly, the use of low-distortion shell mapping is a necessary tool that we contributed, and it should be found useful in a large number of applications. Additionally, we can foresee that a notion of mesh montage (similar to photo- or texture-montage) would also provide a nice extension to our work.

Acknowledgements

The authors would like to thank Stanford University and for providing the 3D models used in this paper. The 3D texture samples were modeled by Mingdong Xie. Special thanks to Becky Sundling and Rui Zhang for their help with video production. The authors are grateful to the anonymous reviewers for their helpful comments. Mathieu Desbrun and Yiyi Tong were partially supported by NSF (CAREER CCR-0133983, and ITR DMS-0453145), DOE (DE-FG02-04ER25657), and Pixar.

Table 1: Statistics on all base meshes, texture samples and synthesis results presented in this paper.

<table>
<thead>
<tr>
<th>Base Mesh</th>
<th>Swatch Mesh</th>
<th>Output Mesh</th>
<th>Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Face</td>
<td>#Vert</td>
<td>#Face</td>
<td>#Vert</td>
</tr>
<tr>
<td>Plane + Wire</td>
<td>NA</td>
<td>NA</td>
<td>15168</td>
</tr>
<tr>
<td>Plane + Apple</td>
<td>NA</td>
<td>NA</td>
<td>2784</td>
</tr>
<tr>
<td>Bunny + Wave</td>
<td>5243</td>
<td>2602</td>
<td>15168</td>
</tr>
<tr>
<td>Cup</td>
<td>5243</td>
<td>2602</td>
<td>2916</td>
</tr>
<tr>
<td>Venus</td>
<td>5560</td>
<td>2793</td>
<td>20608</td>
</tr>
<tr>
<td>Armor</td>
<td>4224</td>
<td>2383</td>
<td>36540</td>
</tr>
<tr>
<td>Chair</td>
<td>12242</td>
<td>6489</td>
<td>8394</td>
</tr>
<tr>
<td>David</td>
<td>49988</td>
<td>24988</td>
<td>700</td>
</tr>
<tr>
<td>Ivy Wall</td>
<td>2448</td>
<td>1253</td>
<td>700</td>
</tr>
</tbody>
</table>

References


Figure 9: A chair modeled by our approach. From a triangle mesh of a chair and a geometric texture swatch, we can easily (left) create a wicker chair; (middle) change the texture to create varying visual effects; (right) and remove a few geometric elements to simulate aging.

Figure 10: Growing Ivy: a 3D sample of an ivy’s stem and leaves is applied on the David mesh. With a vector field, the artist can design the final, complex mesh by guiding our texture synthesis algorithm. Here, the ivy is made to wrap around the leg and climb to the torso.


