Lapped Textures

Emil Praun Adam Finkelstein Hugues Hoppe
Princeton University Microsoft Research
http://www.cs.princeton.edu/∼{emilp,af} http://research.microsoft.com/∼hoppe

Figure 1: Four different textures pasted on the bunny model. The last picture illustrates changing local orientation and scale on the body.

Abstract

We present a method for creating texture over an arbitrary surface mesh using an example 2D texture. The approach is to identify interesting regions (*texture patches*) in the 2D example, and to repeatedly paste them onto the surface until it is completely covered. We call such a collection of overlapping patches a *lapped texture*. It is rendered using compositing operations, either into a traditional global texture map during a preprocess, or directly with the surface at runtime. The runtime compositing approach avoids resampling artifacts and drastically reduces texture memory requirements.

Through a simple interface, the user specifies a tangential vector field over the surface, providing local control over the texture scale, and for anisotropic textures, the orientation. To paste a texture patch onto the surface, a *surface patch* is grown and parametrized over texture space. Specifically, we optimize the parametrization of each surface patch such that the tangential vector field aligns everywhere with the standard frame of the texture patch. We show that this optimization is solved efficiently as a sparse linear system.

Keywords: Texture synthesis, texture mapping, parametrizations.
URL: http://www.cs.princeton.edu/gfx/proj/lapped_tex

1 Introduction

This paper describes a method for creating a texture over an arbitrary surface mesh, using a given example 2D texture (Figure 1). Computer graphics applications often use surface textures to give the illusion of fine detail without explicit geometric modeling. There exist several schemes for synthesizing texture on the 2D plane based on example texture. However, these methods cannot be readily extended to cover surfaces of arbitrary topology because such surfaces lack continuous parametrizations over the plane.

Our approach to this problem relies on the observation that even though a global parametrization may not exist, any manifold surface may be *locally* mapped onto the 2D plane. We repeatedly paste small regions of the example texture (*texture patches*) onto parts of the mesh that can be easily mapped (*surface patches*), until the entire mesh is covered with a series of overlapping texture patches, called collectively a *lapped texture*. The perceptibility of seams is reduced by applying alpha-blending at the edges of the pasted texture patches. The user provides local control over the orientation and scale of the synthesized texture by specifying a tangential vector field over the mesh surface.

For each paste operation, we form a surface patch on the mesh by growing a region homeomorphic to a disc. The surface patch is parametrized into texture space so as to locally align the axes of the texture patch with the surface tangential vector field. We cast this as an optimization of a least squares functional. Unlike other approaches that rely on an explicit fairness functional for minimizing texture distortion, the fairness comes from the underlying vector field. Our optimization is therefore extremely fast, as it only involves solving a sparse linear system.

We have tested our method by applying more than sixty textures over seven models, and found that it works surprisingly well. Figure 1 shows a bunny model covered using four different example textures. The fifth bunny has the same texture as the fourth, but the tangential vector field has been modified on the body. As shown in Figure 7, our scheme applies a variety of isotropic and anisotropic textures over complex, organic surfaces in a natural fashion.

Lapped textures embody the simple idea of texturing a surface with overlapping patches. Efficient implementation of this idea is straightforward. Precomputation of patch placements takes only minutes, and the resulting texture displays in real time. Little human effort is necessary to delineate a texture patch and specify texture direction and scale. Because a single texture patch is instantiated many times over the mesh, a large surface may be covered using a compact texture footprint. Finally, the method extends trivially to bump maps, displacement maps, and other surface appearance fields.

The contributions of this paper are: (1) the idea of covering an arbitrary surface using overlapping copies of a texture patch, (2) a fast parametric optimization that allows control over the local texture orientation and scale, (3) a simple scheme for specifying the vector field necessary for this control, and (4) a method of rendering the texture in real time through the composition of precomputed, overlapping surface patches.

2 Previous work

Previous methods for generating texture by example work mainly on images, and are difficult to extend to surfaces. Conversely, the methods proposed for texturing surfaces are mainly procedural in nature, and therefore difficult to control. None of these methods provide convenient control over local orientation and scale. We address the problem of generating texture by example on complex surfaces, while providing control over local orientation and scale.

Texture synthesis The problem of synthesizing textures in 2D has been studied extensively. Heeger and Bergen [8] perturb a noisy image in order to match the histograms of the original image and its steerable pyramid representation with the corresponding histograms of the generated image. They report good results with stochastic textures, but cannot produce realistic replicas of more structured textures such as bricks. DeBonet [2] synthesizes texture from a wide variety of input images by shuffling elements in the Laplacian pyramid representation. Recently, Efros and Leung [5] proposed a scheme based on non-parametric sampling. They grow the texture one pixel at a time, creating for each target pixel a probability distribution based on windows of the original image. The scheme is relatively slow, but produces impressive results for a large class of input images. Xu *et al.* [20] developed a 2D texture synthesis scheme based on the random motion of image blocks; their prototype inspired us to consider texturing surface meshes using overlapping patches.

Many 2D texture synthesis schemes can be extended to 3D using *solid textures:* the color function is defined over a volume, and then sampled on the surface. Heeger and Bergen [8] and Dischler *et al.* [4, 7] propose schemes that apply spectral and histogram analysis to produce a volume-filling function.

A different class of methods synthesizes texture based on a few parameters, instead of by example. Pioneering work in this area by Perlin [15] and subsequent extensions by Worley [19] generate a color defined over the volume using a noise function. In a different approach, reaction-diffusion and biologic evolution can procedurally texture surfaces in 3D directly [6, 17, 18]. The main drawback of these methods is the difficulty of controlling the input parameters in order to get the desired visual result. This parameter-to-visual-appearance feedback loop is further hindered by the long simulation times necessary to produce an image. Also, these methods explore a limited space of possible textures. Of these techniques, perhaps the most suitable for the problem we address is that of Fleischer *et al.* [6]; while their method is designed to produce geometric detail, it might be adapted so that each of their cell models carries a texture patch, and gets aligned locally to a direction field.

Neyret and Cani [12] introduce a scheme for texturing a mesh with a given set of triangular texture tiles. They partition the mesh into a coarse tiling, where each triangular tile is as close to equilateral as possible. Each surface tile is assigned one of the given texture tiles, subject to continuity constraints across tile boundaries. The user chooses *a priori* the number of distinct tile boundaries, and creates a set of tiles that match all possible boundary conditions.

Surface parametrization for texture mapping In seeking a surface parametrization for texture mapping, the primary objective is to minimize distortion. The usual strategy is to define an energy functional for the mapping, and to try to minimize it. In early work in this area, Bennis *et al.* [1] "flatten" a series of user-defined patches via optimization. Maillot *et al.* [10] propose as a deformation functional the Green-Lagrange tensor from elasticity theory. They discretize the problem by meshing the surface, placing on each mesh edge a spring of nonzero rest length. To prevent surface buckling, they also measure squared differences of signed face areas. They minimize the energy functional using a nonlinear optimization procedure, which makes the method relatively slow.

Lévy and Mallet [9] propose a functional that combines orthogonality and homogeneous spacing of isoparametric curves. Although the resulting functional is nonlinear, it can be minimized iteratively as a sequence of linear problems by solving alternately for the *s* and the *t* parametric coordinates.

Pedersen [13, 14] extends texture mapping and cut-and-paste operations to a broader class of surfaces (including implicit surfaces). He positions a meshed version of a square domain onto the surface and allows the vertices of this regular parametrization to slide over the surface, while minimizing the energy of an associated mesh of springs. Arbitrarily-shaped regions of the sliding patch are cut and pasted, using curve-drawing on the surface to define alpha masks. The sliding patch is translated, rotated, scaled and even warped through the manipulation of the control points defining the parametrization. This is probably the closest work to our own. The main difference is that Pedersen's system is designed to be an interactive paint system while ours is aimed at the automatic texturing of objects. Another difference is our use of a tangential vector field to guide the texture orientation and scale during the parametrization process.

3 Our Approach

Our approach consists of identifying a set of broad features from the example texture ("texture patches"), and then repeatedly pasting them onto "surface patches" grown on the mesh, until the mesh is completely covered. Here is an overview of our procedure:

Cut texture patches from input texture	(Figure 2a, § 3.1)
Specify direction and scale fields over mesh	(Figure 2a, § 3.2)
Repeat	
Select random texture patch $\mathcal{T}$	
Select random uncovered location $\mathcal{L}$ for paste	
Grow surface patch $\mathcal{S}$ around $\mathcal{L}$ to size of $\mathcal{T}$	(§ 3.3)
Flatten $\mathcal{S}$ over $\mathcal{T}$	(§ 3.4)
Record paste operation	(§ 3.5)
Update face coverages	(§ 3.6)
Until the mesh is covered	(Figure 2b, § 3.6)

For recording the paste operations and rendering the final model, we propose two completely different approaches which are presented and compared in Section 4.

3.1 Creating the texture patches

For highly structured textures, the texture patch boundaries should avoid cutting across important features, so as to minimize obtrusive seams in the resulting lapped texture. For example, in a brick texture, the patch boundary should not intersect the bricks but instead follow the grout. The user manually outlines image regions using a commercial drawing package. This outlining process is facilitated by gradient-seeking tools such as the "edge finder" in Microsoft PhotoDraw. The first six examples in Figure 7 were created this way.

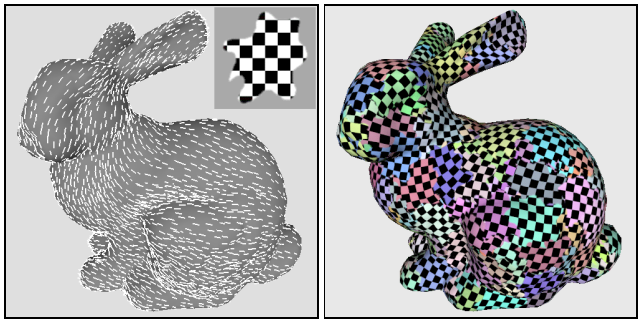
For homogeneous or stochastic textures, the outline of the texture patch is less important, so we use a set of predefined shapes, such as a circle or an irregular "splotch".

In either case, the texture patch is assigned an alpha mask that falls off near the patch boundary (e.g. over a distance of 3 pixels).

3.2 Establishing local orientation and scale

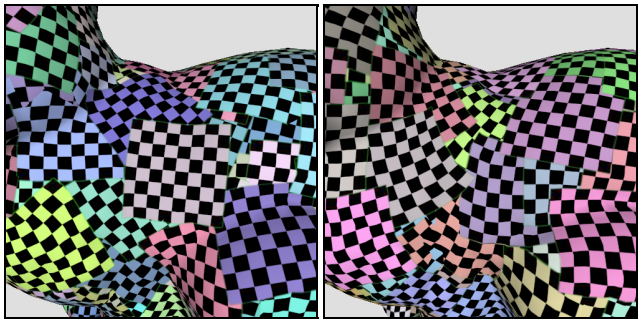
The desired orientation and scale for the texture are specified over the mesh as a tangential vector field. Specifically, each mesh face is assigned a vector **T** within its plane (Figure 2). The direction of **T** is the desired the texture "up" direction, and the magnitude of **T** is the desired local uniform scaling.

A simple choice for the tangential vector field is to project the global up direction onto the surface, and then normalize the resulting tangent vector. More often, the user needs more control



(a) Input: mesh, vector field, texture; (b) Output: covered mesh

Figure 2: Process overview. The inputs are a triangle mesh, a tangential vector field defined on this mesh, and a texture cut into patches. The patches are pasted onto the mesh until it is covered.



(a) Align only patch center (b) Align locally to field

Figure 3: Continuity of the texture direction is improved when the optimization aligns the entire surface patch with the vector field rather than just its center.

over the vector field. With our interface, the user specifies vectors at a few faces. We interpolate vectors at the remaining faces using Gaussian radial basis functions, where radius is defined as distance over the mesh, as computed using Dijkstra's algorithm. The user has control over the spatial extent and weight of each basis function.

We convert the tangential vector $\mathbf{T}$ at each face into a tangential basis $(\mathbf{S}, \mathbf{T})$ by using the "right" direction $\mathbf{S} = \mathbf{T} \times \hat{\mathbf{N}}$, where $\hat{\mathbf{N}}$ is the unit face normal (see Figure 5). The user could alternatively specify this tangential basis directly for local control of a full linear transform for the texture, including shearing and non-uniform stretching. However, we have found this additional flexibility to be unnecessary in our experiments.

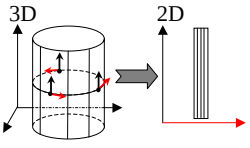
For isotropic textures, the orientations of the paste operations are unimportant. For such cases, we only specify a scaling field over the mesh (and in most cases it is a global constant). During the growth of surface patches (described in the next section), a local orientation field is instead defined through propagation. That is, the center face of the patch is assigned an arbitrary tangent direction, and the direction of each subsequently added face is computed by projecting onto its face plane the average direction of neighboring faces already in the patch.

3.3 Growing the surface patch

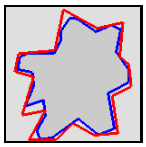
For each paste operation, we grow a surface patch on the mesh by successively adding faces, and form an initial parametrization ϕ of the surface patch into texture space. The parametrization $\phi : R^3 \rightarrow R^2$ is a piecewise linear map specified by texture coordinates assigned to the surface patch vertices. The growth of the surface patch is guided so that its image through ϕ fully covers the texture patch. We next present the details.

First, a random point is chosen on a triangle face that is not yet fully textured, using the coverage test presented in Section 3.6. (Early in the process we give higher priority both to areas of high curvature and to discontinuities in the direction field – where the parametrization is difficult – with the hope that any distorted regions will be covered over later.) The triangle is mapped to texture space such that the chosen point maps to the texture patch center, and the face tangential basis $(\mathbf{S}, \mathbf{T})$ maps to the texture space standard axes $(\hat{\mathbf{s}}, \hat{\mathbf{t}})$.

Next, the surface patch is grown around this seed face, one triangle at a time. Faces are added in order of increasing distance from the center face, but subject to three constraints. First, the surface patch is required to be homeomorphic to a disc. The adjacent figure illustrates a problem that may arise when this constraint is not enforced: the optimization of Section 3.4 maps a tubular patch to a thin vertical band due to the circular dependency in the specified texture $\mathbf{S}$ direction (in red). Second, the patch is only grown over an edge if the edge is still partially inside the texture patch, since there is no point in growing the surface patch beyond the image region that will be pasted. This test is made efficient using the polygonal hull representation discussed later in this section. Third, patch growth is stopped when distortion becomes excessive, which can occur in surface areas with high curvature. In these cases where we are unable to extend the surface patch to fully cover the texture patch, the pasted texture lacks an alpha falloff across one or more edges. We find that the few noticeable artifacts from these "hard edges" are less objectionable than distorted texture.

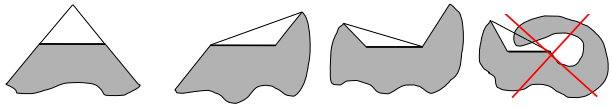


When a face is added to the patch, the newly added vertex is assigned an initial parametrization using the heuristic in [10]. Specifically, for each face that contains the new vertex and an already mapped edge, we predict the parametrization of the new vertex by extending the edge with a triangle similar to the face in 3D. The new vertex is assigned the centroid of these predictions. Note that for texture pasting, we do not prevent the patch from folding or wrapping over itself in texture space.



To determine if we need to grow the patch over a given edge, we test to see if the edge intersects the interior of a **polygonal hull** of the texture patch. We first construct a polygon with vertices at all the boundary pixels (in blue), and then we conservatively simplify it, allowing it only to grow (red outline). Sander *et al.* [16] construct conservative approximations of polyhedral surface meshes using *progressive hull* simplification. We adapt their construction to the 2D setting. Simplification is done using a sequence of edges collapses, but with the constraint that the resulting vertex lie within the correct half-spaces of the previous model. In 3D this involves linear programming, but in 2D it reduces to just the 3 cases illustrated in Figure 4. The simplification operations are prioritized according to the area they add to the polygon interior. Operations that would give rise to self-intersections are disallowed.

Once the surface patch stops growing, we optimize the map ϕ as discussed in the next section. The optimization may sometimes uncover parts of the texture patch. When this occurs, we further grow the patch and optimize again.



(i) Both angles convex (ii) Convex - concave (iii) Both concave No self intersections

Figure 4: Simplification of the outer hull polygon. The thick edge is replaced with the thin lines, thereby removing one vertex.

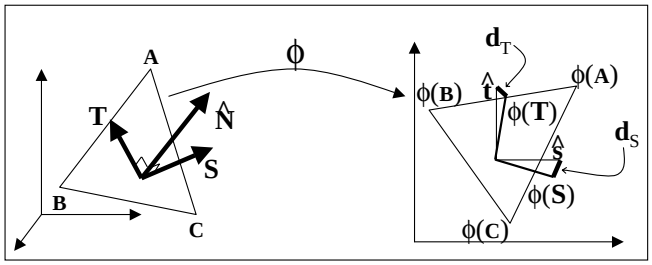


Figure 5: The optimization process minimizes the differences $(\mathbf{d}_S, \mathbf{d}_T)$ between the texture coordinate axes $(\hat{\mathbf{s}}, \hat{\mathbf{t}})$ and the images $(\phi(\mathbf{S}), \phi(\mathbf{T}))$ in texture space of the user-specified vectors $(\mathbf{S}, \mathbf{T})$.

3.4 Optimizing the surface patch parametrization

Having formed a surface patch together with its initial parametrization ϕ, we optimize ϕ so as to locally match both the orientation and scale of the texture with the vector field defined on the surface. More precisely, we attempt to match the images of the surface tangent vectors $(\mathbf{S}, \mathbf{T})$ with the texture coordinate axes $(\hat{\mathbf{s}}, \hat{\mathbf{t}})$. Figure 3 shows the importance of aligning the vector field over the whole patch, as opposed to just at its center.

For each mesh face $f = \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$, the "up" vector $\mathbf{T}$ lies within the face plane. We can therefore express it using its barycentric coordinates with respect to the vertex positions[1]:

$$\mathbf{T} = \alpha\,\mathbf{A} + \beta\,\mathbf{B} + \gamma\,\mathbf{C}\,, \quad \text{where} \quad \alpha + \beta + \gamma = 0\,.$$

Since the map ϕ is linear over the face, the image $\phi(\mathbf{T})$ is therefore a linear function of the vertex parametrizations $\phi(\mathbf{A})$, $\phi(\mathbf{B})$, and $\phi(\mathbf{C})$. As shown in Figure 5, we define the difference vector

$$\mathbf{d}_T \;=\; \alpha\,\phi(\mathbf{A}) + \beta\,\phi(\mathbf{B}) + \gamma\,\phi(\mathbf{C}) - \hat{\mathbf{t}}\,,$$

and we do likewise for the difference vector $\mathbf{d}_S$. Our optimization problem is to find the vertex parametrizations that minimize the least squares functional

$$\sum_f \|\mathbf{d}_S\|^2 + \|\mathbf{d}_T\|^2\,.$$

The minimum of this function is unique up to a translation. We therefore add a positional constraint to fix the location of the patch center. The exact solution to the minimization problem only requires solving a sparse linear system. Since we begin with a reasonable approximation of the solution, we use a conjugate-gradient iterative solver, which is faster than an explicit solver like Gaussian elimination.

Note that the functional does not include any explicit "fairness" term to penalize distortion in the parametrization. Instead, continuity of the parametrization across mesh edges relies on the continuity of the user-provided tangential vector field. Unlike many local edge-spring functionals, our functional does not have local minima when the face orientations flip, and thus avoids "buckling" artifacts. Finally, the parametrization is well-behaved even though the patch boundary is left unconstrained.

3.5 Recording the paste operations

The paste operation sends image samples from the texture patch onto the surface patch using ϕ^{-1}. Section 4 presents two schemes for recording these paste operations.

[1] Given a set of points P in general position, any vector in the affine subspace spanning P is uniquely expressed as a linear combination of P, and these barycentric coordinates sum to zero.

3.6 Computing face coverages

To decide where to apply paste operations (Section 3.3), we need to know if a face is already fully covered by texture. We answer this query using a rasterization algorithm. After each paste operation, we render all the patch faces in an offscreen buffer, with the parametrization from Section 3.4. (In the rare case that the patch overlaps itself in texture space, we compute the coverage in several passes, for subsets of non-overlapping faces.) Each face in the patch is rendered using all paste operations that overlap it. We use the R and G color channels to store the face ID, and the B channel to accumulate the opaque regions of the paste operations. To determine the coverage of a face, we divide the number of covered pixels (in the B channel) by the number of pixels in the triangle. For each faces that is not fully covered, we remember an uncovered point inside the face, in order to start a future paste operation centered there. When all faces are fully covered, we are done pasting.

4 Texture storage and rendering

We propose two approaches for representing the textured object. The first approach constructs a traditional surface parametrization using a texture atlas, and pre-renders the lapped texture into this atlas. The second and more interesting approach uses the hardware graphics pipeline to composite the texture patches at runtime.

Rendering with a texture atlas Previous approaches for storing texture on meshes use a texture atlas (e.g. [3, 10, 16]). An atlas is a collection of charts that map regions of the surface to subsets of a texture unit square, such that no two distinct surface points map to the same texture point (see Figure 6b). Ideally, the charts should have low parametric distortion, and should have uniform resolution across the mesh.

To build an atlas, we use a method similar to Maillot *et al.* [10]. We segment the mesh into regions by bounding each region's space of face normals, flatten each region using relaxation, and let the user arrange the flattened pieces. To grow and flatten the surface regions, we use the algorithm described in Section 3.3 (for the case of isotropic textures), but with two additional constraints. We require the normals of the added faces to be within a certain angle from the one of the center face. And, we prevent overlaps by checking for intersections using a spatial hash table. The next step is to arrange the chart images inside the unit square. Packing a set of non-convex polygons into a given 2D domain is a well-studied problem in computational geometry known as "pants packing" [11] due to its application in the clothing industry. Since the problem is NP-hard, an exact solution cannot generally be computed. Heuristic algorithms for arranging on the order of a hundred polygons with no initial layout produce significantly worse results than a trained human. Therefore, we let the user manually arrange the charts.

The atlas is represented using sets of texture coordinates at the mesh vertices. During a preprocess, the texture paste operations are composited into the atlas charts. At runtime, the mesh is rendered using ordinary texture mapping.

(a) Runtime pasting (b) Rendering with a texture atlas

Figure 6: Comparison of our two texture representations. The atlas representation is more portable but may have sampling problems.

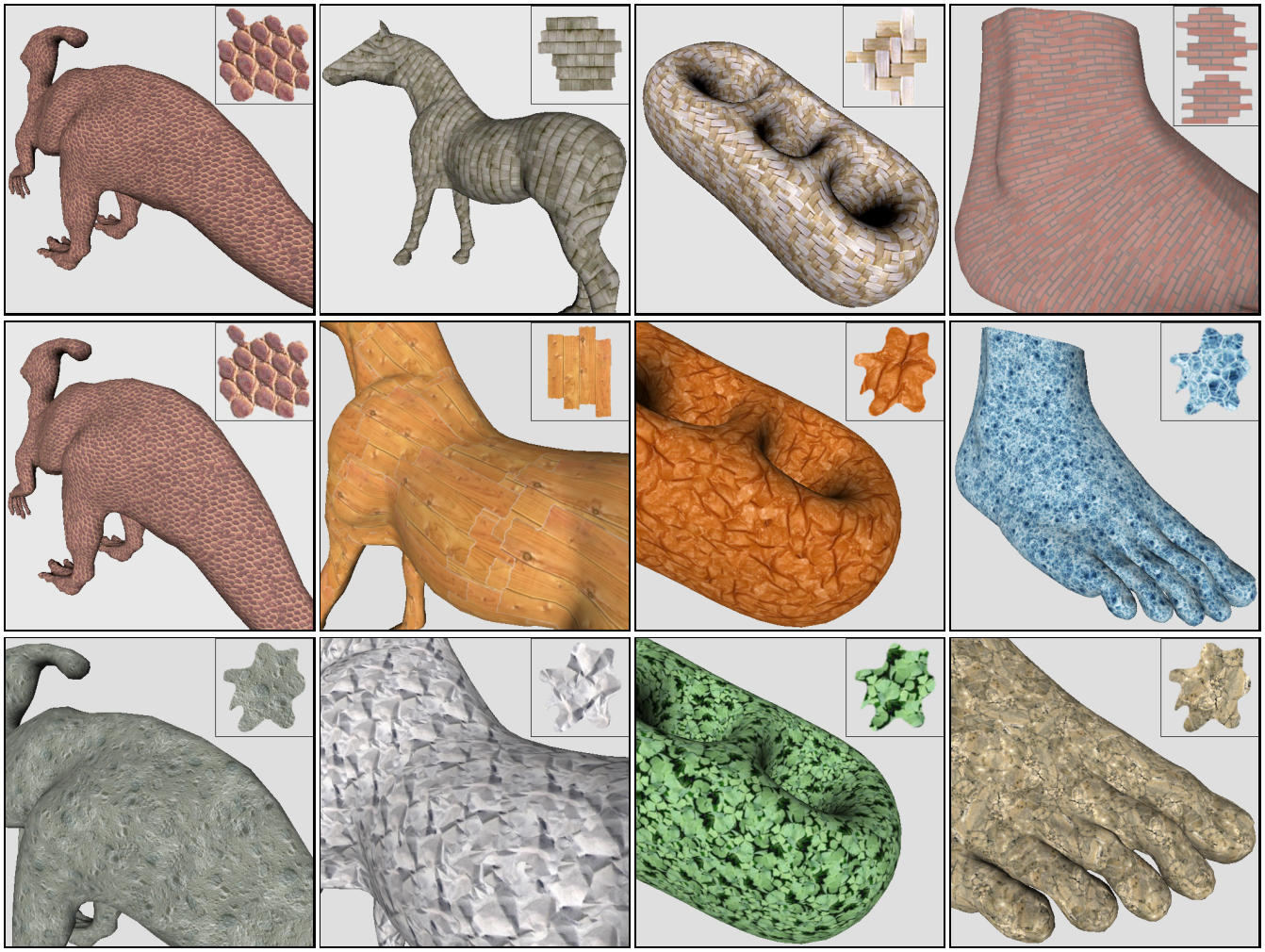


Figure 7: For the first 6 pictures the user specified the texture patch boundary and a vector field over the mesh. The remaining pictures are examples of isotropic textures, and are generated automatically. The two upper dinosaurs are frames from an animation of the tail (see video).

Runtime pasting of lapped textures Our preferred approach for rendering the lapped texture is to record the parameters for each paste operation (texture patch index, list of surface patch faces, and texture coordinates for vertices), and to render these surface patches at runtime with alpha blending enabled.

With runtime pasting, each face of the model is rendered several times, once for each surface patch to which the face belongs. This increases the load on the graphics system, in terms of both geometry processing and rasterization. To reduce this overhead, during a preprocess we remove for a given paste operation any faces that are completely occluded by subsequent paste operations; such faces are detected through a rasterization algorithm as in Section 3.6. With this optimization, the average number of times that each face gets rendered ranges between 1.5 and 3.2, depending on the texture scale and the model. Graphics systems have begun to support *multitexturing*, whereby the rasterizer can directly evaluate a complicated shading expression involving several texture lookups (recently, as many as four). Although not used in our current prototype, this multitexturing capability could reduce the number of times each face is rendered.

Since our flattening process may produce texture coordinates outside the unit square, we use texture coordinate clamping. For proper interaction of blending and mipmapping, the texture patch must have a border of transparent pixels at all mipmap levels finer than 2x2. We therefore build the mipmap levels explicitly.

Tradeoffs of rendering approaches Compared to the atlas, runtime pasting requires little texture memory, since it only involves storing the initial texture patches (and often there is only one such patch). As in ordinary 2D texture tiling, large amounts of apparent texture can be created with little actual texture memory usage. The randomness of our surface patch construction makes repetitiveness of the texture less obvious than with ordinary 2D tiling.

Runtime pasting offers better visual quality because it does not suffer from several problems inherent to a texture atlas (Figure 6):

- *Sampling.* The use of an atlas adds one more resampling step, thus inherently degrading the texture image quality. The quality is improved by increasing the texture resolution, but this further reduces the ratio of apparent texture to texture memory usage.
- *Discontinuities.* The tri-linear interpolation filter used to sample the texture does not match exactly at chart boundaries.
- *Mipmapping problems.* At coarser mipmap levels, distinct charts of the atlas are wrongly averaged together.

The disadvantages of the runtime pasting approach are the following. As discussed above, rendering is likely slower since faces are drawn multiple times (though multitexturing may alleviate this). Also, the storage format for the model is somewhat less portable since it involves textures with alpha. Finally, each face must be rendered with different textures in a specific order, and some rendering systems may not guarantee the order in which overlapping polygons are drawn.

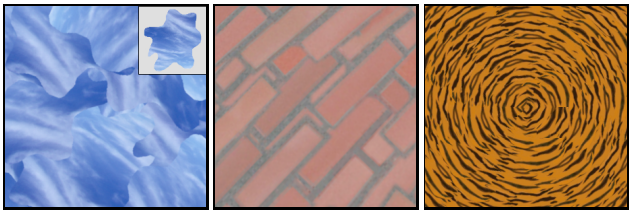


Figure 8: Limitations of our method: (a) Strong low-frequency components, (b) Boundary mismatch, (c) Singularity point.

5 Results

Several textured meshes are shown in Figure 7. In all the examples, the synthesized texture is generated from a single texture patch extracted from the example texture (except the brick foot which uses two). It takes the user about 15 minutes to create a non-trivial direction field for the meshes shown. The growth and parametrization of the patches takes between 20 seconds and 6 minutes to compute on a 733 MHz Pentium III with a GeForce graphics card. Except where noted, all of the examples shown in the paper and the accompanying video tape use runtime pasting rather than the atlas approach. The meshes used in this paper average 5000 faces. Since each face is rendered on average two or three times, all of the lapped textures shown here display in real time. For homogeneous textures we can use a generic texture patch boundary (e.g., the splotch), and switch between different texture examples instantaneously at runtime.

Figure 8 shows some of the limitations of lapped textures. Patch seams become noticeable when the texture patch has strong low frequency components. Seams are also apparent when viewing highly structured textures up close. For anisotropic textures, the user-specified vector field generally has singularity points, since of course one cannot smoothly comb a hairy ball. Sometimes, visual artifacts are caused by poor vector field sampling near these points due to the presence of large faces; we reduce such artifacts by locally subdividing the mesh.

6 Summary and future work

We have introduced lapped textures, a new approach for covering arbitrary triangle meshes with an example 2D texture. The approach is to apply copy/paste operations to cover the surface with overlapping texture patches, using alpha blending to hide seams. The paste operation relies on a new, fast, robust flattening scheme that simultaneously minimizes distortion of the texture and matches local orientation and scale specified by the user.

Our scheme proves to be highly practical, allowing the creation of complex textures on meshes at a fraction of the user effort required by 3D painting. Lapped textures can be used as a starting point for further manual painting (e.g. for unique details such as the mouth and eyes of a bunny).

This work suggests a number of areas for future investigation:

Fine-tuning patch placement. It may be beneficial to fine-tune the placement of the surface patches so that sharp texture features align across patch boundaries (Figure 8b). Initially, we anticipated that this process would be absolutely necessary to make patch boundaries unobtrusive; we were pleasantly surprised to find that the method works quite well without this embellishment. Nonetheless, we still believe it could enhance the results.

Greater automation. We believe that methods to reduce user interaction would make this system even more practical. For example, we have considered automatic texture patch creation, automatic equalization of low-frequency information in these patches (Figure 8a), and automatic direction field construction using surface curvatures.

Other texture types. Within the lapped texture framework, we are now exploring several other types of textures, including animated, volumetric and view-dependent textures.

Acknowledgements

We thank Harry Shum for demonstrating a prototype 2D texture synthesis scheme [20] that largely inspired this work. Thanks to Viewpoint DataLabs and Stanford University for the surface meshes, and Michael Cohen and Rico Malvar for proposing the name “lapped textures”.

Emil Praun was supported in part by a Microsoft Research internship. The research of Adam Finkelstein is supported by an NSF CAREER Award and an Alfred P. Sloan Fellowship.

References

[1] Bennis, C., Vézien, J.-M., Iglésias, G., and Gagalowicz, A. Piecewise surface flattening for non-distorted texture mapping. *Computer Graphics (Proceedings of SIGGRAPH 91) 25*, 4, 237–246.

[2] Bonet, J. S. D. Multiresolution sampling procedure for analysis and synthesis of texture images. *Computer Graphics (Proceedings of SIGGRAPH 97)*, 361–368.

[3] Cignoni, P., Montani, C., Rocchini, C., and Scopigno, R. A general method for preserving attribute values on simplified meshes. In *IEEE Visualization* (1998), pp. 59–66.

[4] Dischler, J. M., Ghazanfarpour, D., and Freydier, R. Anisotropic solid texture synthesis using orthogonal 2D views. *Computer Graphics Forum 17*, 3 (1998), 87–96.

[5] Efros, A. A., and Leung, T. K. Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision* (Sept. 1999).

[6] Fleischer, K., Laidlaw, D., Currin, B., and Barr, A. Cellular texture generation. *Computer Graphics (Proceedings of SIGGRAPH 95)*, 239–248.

[7] Ghazanfarpour, D., and Dischler, J.-M. Generation of 3D texture using multiple 2D models analysis. *Computer Graphics Forum 15*, 3 (1996), 311–324.

[8] Heeger, D. J., and Bergen, J. R. Pyramid-based texture analysis/synthesis. *Computer Graphics (Proceedings of SIGGRAPH 95)*, 229–238.

[9] Lévy, B., and Mallet, J.-L. Non-distorted texture mapping for sheared triangulated meshes. *Computer Graphics (Proceedings of SIGGRAPH 98)*, 343–352.

[10] Maillot, J., Yahia, H., and Verroust, A. Interactive texture mapping. *Computer Graphics (Proceedings of SIGGRAPH 93)*, 27–34.

[11] Milenkovic, V. J. Rotational polygon containment and minimum enclosure. *Proc. of the 14th Annual Symp. on Computational Geometry, ACM* (June 1998).

[12] Neyret, F., and Cani, M.-P. Pattern-based texturing revisited. *Computer Graphics (Proceedings of SIGGRAPH 99)*, 235–242.

[13] Pedersen, H. K. Decorating implicit surfaces. *Computer Graphics (Proceedings of SIGGRAPH 95)*, 291–300.

[14] Pedersen, H. K. A framework for interactive texturing operations on curved surfaces. *Computer Graphics (Proceedings of SIGGRAPH 96)*, 295–302.

[15] Perlin, K. An image synthesizer. *Computer Graphics (Proceedings of SIGGRAPH 85) 19*, 3, 287–296.

[16] Sander, P., Gu, X., Gortler, S., Hoppe, H., and Snyder, J. Silhouette clipping. *Computer Graphics (Proceedings of SIGGRAPH 2000)*.

[17] Turk, G. Generating textures for arbitrary surfaces using reaction-diffusion. *Computer Graphics (Proceedings of SIGGRAPH 91) 25*, 4, 289–298.

[18] Witkin, A., and Kass, M. Reaction-diffusion textures. *Computer Graphics (Proceedings of SIGGRAPH 91) 25*, 4, 299–308.

[19] Worley, S. P. A cellular texture basis function. *Computer Graphics (Proceedings of SIGGRAPH 96)*, 291–294.

[20] Xu, Y., Guo, B., and Shum, H.-Y. Chaos mosaic: Fast and memory efficient texture synthesis. Tech. Rep. MSR-TR-2000-32, Microsoft Research, 2000.