Meshed Atlases for Real-Time Procedural Solid Texturing

NATHAN A. CARR and JOHN C. HART
University of Illinois, Urbana-Champaign

We describe an implementation of procedural solid texturing that uses the texture atlas, a one-to-one mapping from an object's surface into its texture space. The method uses the graphics hardware to rasterize the solid texture coordinates as colors directly into the atlas. A texturing procedure is applied per-pixel to the texture map, replacing each solid texture coordinate with its corresponding procedural solid texture result. The procedural solid texture is then mapped back onto the object surface using standard texture mapping. The implementation renders procedural solid textures in real time, and the user can design them interactively.

The quality of this technique depends greatly on the layout of the texture atlas. A broad survey of texture atlas schemes is used to develop a set of general purpose mesh atlases and tools for measuring their effectiveness at distributing as many available texture samples as evenly across the surface as possible. The main contribution of this paper is a new multiresolution texture atlas. It distributes all available texture samples in a nearly uniform distribution. This multiresolution texture atlas is the first of its kind to fully support MIP-mapped minification antialiasing and linear magnification filtering.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*color, shading and texture*

Additional Key Words and Phrases: Mesh partitioning, MIP-map, procedural texturing, solid texturing, texture mapping, texture atlas

1. INTRODUCTION

The concept of procedural solid texturing is well known [Peachey 1985; Perlin 1985], and has found widespread use in graphics [Ebert et al. 1994]. Solid texturing simulates a sculpted appearance and its texture coordinates are easy to generate regardless of surface topology. Procedural texturing makes solid texturing practical by computing the texture on demand (instead of accessing a stored volumetric array), at a level detail limited only by numerical precision. These features were quickly adopted for production-quality rendering by the entertainment industry, and became a core component of the Renderman Shading Language [Hanrahan and Lawson 1990].

With the acceleration of graphics processors outpacing the exponential growth of general processors, there have been several recent calls for real-time implementations of procedural shaders, e.g. [Hanrahan 1999; Pixar Corp. 1999]. Real-time procedural shaders make videogame graphics richer, virtual environments more realistic and modeling software more faithful to its final result. Until recently, real time procedural shading systems required special-purpose graphics supercomputers or processors, but with the recent acceleration of PC graphics processors, real-time shading is now available at the consumer level.

Peercy et al. [2000] recently took a large step toward this goal by developing a compiler that translated Renderman shaders into multipass OpenGL code. While complex Renderman shaders could not yet be rendered in real-time, this compiler showed that their implementation on graphics accelerators was at least feasible. They created new interactive shading language, ISL, to produce more efficient OpenGL shaders. Proudfoot et al. [2001] has further developed such techniques with a procedural shading language, RTSL, designed specifically for compilation to modern PC-based graphics accelerators.

Unfortunately, neither ISL nor RTSL introduced any new techniques for solid texturing, supporting it instead with texture volumes. While modern graphics accelerator boards now have enough texture memory to store a moderate resolution volume, and some even support texture compression, storing a 3-D dataset to produce a 2-D surface texture is inefficient and an unnecessarily wasteful use of texture memory. Synthesizing

a procedural texture over an entire texture volume also wastes processing time. We can instead support procedural solid texturing using standard 2-D surface texture mapping.

A (surface) texture mapping $\mathbf{u} = \mathbf{u}(\mathbf{x})$ is a function from a surface into a compact subset of the plane called the *texture map.* The texture mapping need not be continuous, but usually consists of piecewise continuous parts $\mathbf{u}_i()$ called *charts.* The area on the surface in model coordinates is called the *chart domain* whereas the area the domain maps to in the texture map is called the *chart image.* The collection of charts that forms a texture mapping $\mathbf{u}(\cdot) = \cup\mathbf{u}_i(\cdot)$ is called an *atlas* [Munkres 1975]. In computer graphics, an atlas creates a one-to-one correspondence between an object surface and its texture map. In this case, neither chart domains nor chart images overlap.

Apodaca and Gritz [1999] described how a texture atlas can store the shading of a model. This technique shades a mesh in world coordinates, but stored the resulting colors in a second "reference" copy of the mesh embedded in a 2-D texture map. The mesh could then be later shaded by applying the texture map instead of computing its original shading.

The texture atlas can support real-time view-independent procedural solid texturing, derived from an example by Cignoni et al. [1998]. Consider a single triangle with 3-D solid texture coordinates[1] $\mathbf{s}_i$ and 2-D surface coordinates $\mathbf{u}_i$ assigned to its vertices $\mathbf{x}_i$ for $i = 1, 2, 3$. Figure 1a shows such triangles, plotted in model coordinates with color indicating their solid coordinates. This technique applies a procedural solid texture to the triangle $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ in three steps. The first step rasterizes the triangle into a texture map using its surface texture coordinates $(\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3)$. This rasterization interpolates its vertices' solid texture coordinates $\mathbf{s}_i$ across its face. Figure 1b shows each pixel (u, v) in the rasterization now contains the interpolated solid texture coordinates $\mathbf{s}(u, v)$. The second step executes a texturing procedure $\mathbf{p}()$ on these solid texture coordinates, resulting in the color $\mathbf{c}(u, v) = \mathbf{p}(\mathbf{s}(u, v))$ shown in Figure 1c. This color table $\mathbf{c}(u, v)$ is a texture map that we apply to the original triangle $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ via its surface coordinates $\mathbf{u}_i$, resulting in the view-independent procedural solid texturing shown in Figure 1d.

One serious problem with the texture atlas has been the proper utilization and distribution of samples. An atlas-based procedural solid texture preview mode has been implemented in recent modeling packages [Brinsmead 1993; Thorne 1997] though they have been known to suffer from sampling problems. Lapped textures [Praun et al. 2000] also used a texture atlas to allow the lapped texture swatches to be applied in a simple texture mapping operation, noting "the atlas representation is more portable, but may have sampling problems." Section 2 reviews previous techniques for automatically generating a texture atlas for a given meshed object. The most relevant of these methods attacked the problem of preserving surface detail during simplification, which also emphasized sampling fidelity.

Section 3 identifies key properties of the texture atlas important for proper distribution of texture samples, and describes the artifacts that result when these properties are not satisfied. Previous texture atlas quality criteria such as distortion and seam length are shown to be irrelevant for this task. More appropriate texture atlas quality criteria include coverage, which indicates how completely available texture samples are utilized, and relative scale, which shows how evenly these samples are distributed across the object surface. Triangles need not be contiguous in the texture atlas, and careful rasterization of these triangles has been used to avoid seam artifacts. This section also describes a new subtle variation of this rasterization scheme that is later used to support linear magnification filtering.

Section 4 provides some general purpose texture atlases, the uniform, area weighted and length weighted mesh atlas, which are similar to those found in previous work. It concludes with a comparison of their effectiveness at using and distributing texture samples using the criteria of Section 3.

Section 5 describes the main contribution of this paper: a multiresolution texture atlas. The multiresolu-

[1]To keep these two textures straight, we will use $\mathbf{s} = (s, t, r)$ to indicate the solid texture coordinates and $\mathbf{u} = (u, v)$ to indicate the texture map coordinates. We will need to assign both kinds of coordinates to the vertices of a mesh.

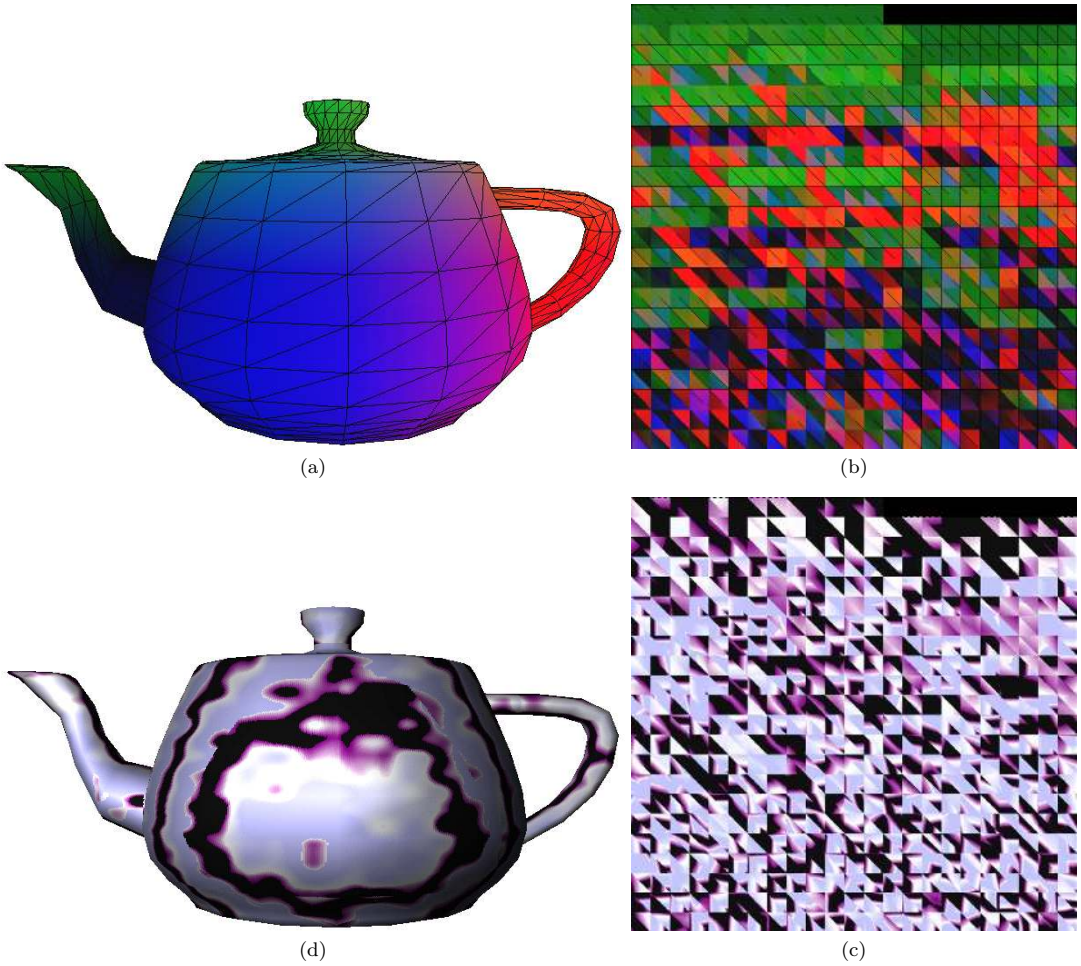


Fig. 1. (Clockwise.) Solid texture coordinates stored as vertex colors of a model (a) are rasterized into a texture atlas (b). A procedural shader replaces the interpolated solid texture coordinates with colors (c), which are applied to the object using texture mapping (d).

tion atlas is created through the application of the Metis algorithm [Karypis and Kumar 1998] and results in a quadtree texture atlas whose quadrants correspond to partitions of proximate triangles. This subset property allows the resulting texture to be MIP-mapped, and the pairing of triangles that share edges coupled with the rasterization variation described in the previous section allows linear magnification filtering. Partitions of two, three and four triangles are collected into motifs and packed into square or rectangular blocks, which results in complete utilization of all available texture samples. The partitioning also preserves triangle area which distributes samples more evenly across the object surface.

Section 6 applied the atlases from Sections 4 and 5 on several examples. It then compares their results both visually and numerically using the measures of coverage and relative scale developed in Section 3.

Section 7 describes a system that uses the texture atlas to design and apply procedural solid textures for

objects in real time. The use of an atlas enables procedural texturing operations to be applied directly to the texture map, either on the host or with a multipass program on the graphics accelerator. This allows the user to manipulate the variables of the texture procedure and observe the results interactively.

Section 8 concludes with mention of other applications of this method (e.g. surface painting, appearance-preserving simplification and procedural shading) and some new ideas it has inspired.

2. PREVIOUS WORK

Section 2.1 reviews past implementations of real-time procedural solid texturing, which have either required high-performance graphics computers or special-purpose graphics hardware. Procedural solid texting in this paper is based on the texture atlas. Section 2.2 reviews past texture atlases designed to reduce distortion, whereas Section 2.3 surveys atlases used for appearance preserving simplification.

2.1 Procedural Solid Texturing Systems

Parallel graphics computers have been used to overcome the computational burden of procedural solid texturing. The Pixel Machine [Potmesil and Hoffert 1989], for example, was used to exploring volumetric rendering of procedural solid texture spaces known as "hypertexture" [Perlin and Hoffert 1989].

Parallel implementations were also able to synthesize procedural solid textures in real time. Rhoades et al. [1992] developed a specialized assembly language, called T-code, for procedural shading on Pixel Planes 5. The T-code interpreter included automatic differentiation to estimate the variation of the procedure across the domain of a pixel. This estimate of the variation was used as a filter width to antialias the procedural texture, by averaging the range of colors the procedure could generate within the pixel.

Olano and Lastra [1998] (see also Lastra et al. [1995]) implemented a real-time subset of the Renderman shading language on Pixel Flow [Molnar et al. 1992], including the ability to synthesize procedural solid textures. Standard Renderman shader tools including automatic differentiation and clamping [Norton et al. 1982] were used to antialias the procedural textures.

Dedicated hardware can also be used to support real-time procedural solid texturing. Hart et al. [1999] designed a VLSI processor based around a single function capable of generating several of the most popular procedural solid textures. Procedural solid textures were transmitted to this hardware as a set of parameters to the texturing function. The derivative of the function was also implemented to automatically antialias the output, à la Rhoades et al. [1992].

Current graphics libraries such as OpenGL [Segal and Akeley 1999] and Direct3D [Microsoft Corp. 2000] support solid texturing with the management of homogeneous 3-D texture coordinates, and recent versions of these libraries support three-dimensional texture volumes that can be MIP-mapped to support antialiasing. The addition of a third array dimension can quickly consume available texture memory resources. Graphics libraries can also support procedural solid texturing as vertex programs [NVidia Corp. 2000] but the detail of such procedural textures is limited by the sampling frequency of the surface tessellation.

Peercy et al. [2000] and Proudfoot et al. [2001] have developed multipass shading compilers. As mentioned in the introduction, these methods do not provide new techniques for solid texturing. They instead combine texture volumes for solid texturing, and could use any of the existing techniques including the texture atlas method described by this paper.

2.2 Non-Distorted Texture Mapping and Parameterizations

Some of the previous research on texture atlas construction has focused on the problem of reducing distortion. These techniques assign texture coordinates such that the resulting texture mapping is locally close to similarity. Ma and Lin [1988; Bennis et al. [1991; Maillot et al. [1993] and more recently [Lévy and Mallet 1998] have devised global optimization methods that assigned texture coordinates that minimized a distortion

metric. Others such as Samek [1986] reduce distortion by flattening the polygons onto a cube surrounding the object.

If the surface texture mapping is one-to-one, then its inverse $\mathbf{u}^{-1}()$ is a parameterization of the surface. Atlases often (but not always) parameterize the surface, such that each pixel in the texture map represents a unique location on the object surface[2]. Hence parameterization methods could be used to generate atlases. For example, MAPS [Lee et al. 1998] parameterizes a mesh of arbitrary topological type, using a simplified version of the mesh embedded in three-space to serve as the base domain of smoothed piecewise barycentric parameterizations. This base mesh and the parameterization it supports could be flattened into a 2-D texture map, but the same flattening could also create an atlas by directly flattening the original mesh.

These techniques focused on preserving connectivity in the texture atlas, such that neighboring triangles in the surface mesh are also neighbors in the texture atlas. Such connectivity-preserving atlases use a single 2-D image texture overlayed on the texture atlas as the texture map. As Section 3 will show, the quality of our procedural solid texturing approach is not dependent on the connectivity or distortion of the texture atlas. Hence these connectivity-preserving and distortion-minimizing atlas generation techniques are not the most appropriate choices for real-time procedural solid texturing.

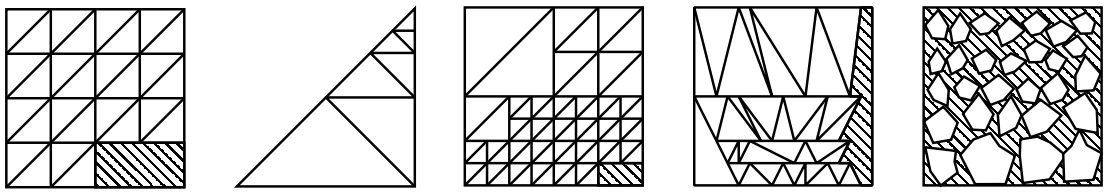
2.3 Appearance-Preserving Simplification

The simplification community has used the texture atlas to preserve surface attributes such as color, texture or normals during the simplification process. Triangles in the simplified surface are packed into a texture atlas. The texture imposed on these triangles is sampled from the regions these simplified triangles correspond to in the original detailed mesh. Section 3 will show such techniques can safely ignore the distortion and continuity of the texture atlas in favor of sampling fidelity. These techniques hence are directly relevant to procedural solid texturing, and in fact one previous method [Cignoni et al. 1998] demonstrated its atlas with a procedural solid texture example.

2.3.1 *Triangular Block Atlas.* Maruya [1995] was apparently the first to devise a packed triangle texture atlas, placing the triangles in an arbitrary order in the texture map. He devised a block packing mechanism consisting of rows of squares, with each square containing two triangle blocks as shown in Figure 2(a). Each triangle block could then contain one large triangle, two medium triangles, four small triangles, and so on for any combination of triangles of "size" $1/2i$ so long as their sizes summed to no more than one, as shown in Figure 2(b). Triangles were quantized into a given size depending on their "scale," which was defined as the number of texture samples the triangle's texture atlas image would cover divided by the number of color samples the surface triangle contained. A minimum scale α_0 would be set a priori (based on the number of triangles and the texture resolution), and surface triangles would be mapped to atlas triangles of a size such that the resulting scale would be no less than α_0. Hence this technique would result in a texture atlas with α_0 texture samples per surface color sample.

2.3.2 *Quantized Area Mesh Atlas.* Rioux et al. [1996] described a packing of triangles assumed to be from a simplified mesh, such that the color data from the vertices of the original unsimplified mesh will become the texture of the simplified mesh. Let D_i be the average distance between vertices in the region of the original high resolution mesh that simplifies to triangle i. Let $E_{\max}$ be the length of the longest edge of triangle i. Then their technique would require triangle i to map to a triangle of edge-length $E_{\max}/D_i$ in the atlas, with a goal of achieving at least one texture sample per vertex color sample in the high resolution mesh. The atlas triangles were quantized into right triangles whose adjacent and opposite edges were equal and integer powers of two, as shown in Figure 2(c). Arbitrary pairs of equal-sized atlas triangles were joined together at

[2]In topology, the atlas is used to define manifolds. In this context the atlas need not be one-to-one and the range of its charts may overlap.



(a) (b) (c) (d) (e)

Fig. 2. Some previous atlases, with wasted texture space hatched. Triangle block packing (a) of 32 triangular blocks. Blowup of a sample triangle block (b) containing triangles of size 1/2, 1/4, 1/8, 1/16, 1/32, and two of size 1/64. Quantized area mesh atlas (c) of 68 triangles quantized to three different power-of-two widths. Sheared mesh atlas (d) of 34 triangles quantized into three power-of-two altitudes. Progressive mesh atlas (e) of 28 clusters.

the hypotenuse to form a rectangular cell. The triangles were offset by one pixel at the hypotenuse to avoid a seam problem described in Section 3.2, resulting in rectangular cells one pixel longer in one dimension than the other. The resulting rectangular cells were strip packed into the texture map.

2.3.3 *Undistorted Mesh Atlas.* Battke et al. [1996] constructed a texture atlas by performing a strip packing of triangles into a square region. The triangles were rotated and translated such that their longest edge lies along the horizontal axis, then sorted by altitude. The triangles are then alternately flipped and compacted to form the efficient though sub-optimal greedy packing shown in Figure 3. This packing was sensitive to texture sampling such that the packing did not cause neighboring triangles to share the same border pixel in the texture atlas. Their algorithm could optionally surround triangles with a boundary to support bilinear filtering and to remove seam artifacts.

2.3.4 *Sheared Mesh Atlas.* Cignoni et al. [1998] developed an anisotropic mesh atlas to better accommodate skinny triangles. This mesh atlas worked similarly to the undistorted texture atlas that rigidly packed triangles into the texture atlas, except that it quantized the altitude of the triangles to the next higher power of two, then sheared each triangle to fit exactly with the previous triangle edge in the current strip, as shown in Figure 2(d). The method also packed large triangles first, and packed smaller triangle strips in the remaining gaps.

2.3.5 *Progressive Mesh Atlas.* Sander et al. [2001] devised an atlas of packed clusters resulting from a modified progressive-mesh simplification process. Neighboring triangles were repeatedly merged into sufficiently co-planar clusters with minimal boundary length, and these boundaries were "straightened" afterward into rough polygons. These clusters were then rescaled relative to each other to more evenly distribute the samples, and strip packed (using bounding rectangles) into the texture map as shown in Figure 2(e).

3. ANALYSIS

Texture atlases classically used for texture mapping have strived to minimize chart distortion, and to minimize seams between chart domains. The texture atlases used for real-time procedural solid texturing (and appearance preserving simplification) focus instead on the even distribution of as many texture samples as possible across the surface. Section 3.1 reviews previous work on minimizing distortion, and shows that for our particular application, its artifacts are minor. Section 3.2 describes how seam artifacts can be eliminated with careful rasterization. We instead use two new measures of atlas quality: coverage (Sec. 3.3) and relative scale (Sec. 3.4), which better indicate how well an atlas utilizes and distributes available texture samples for procedural solid texturing.

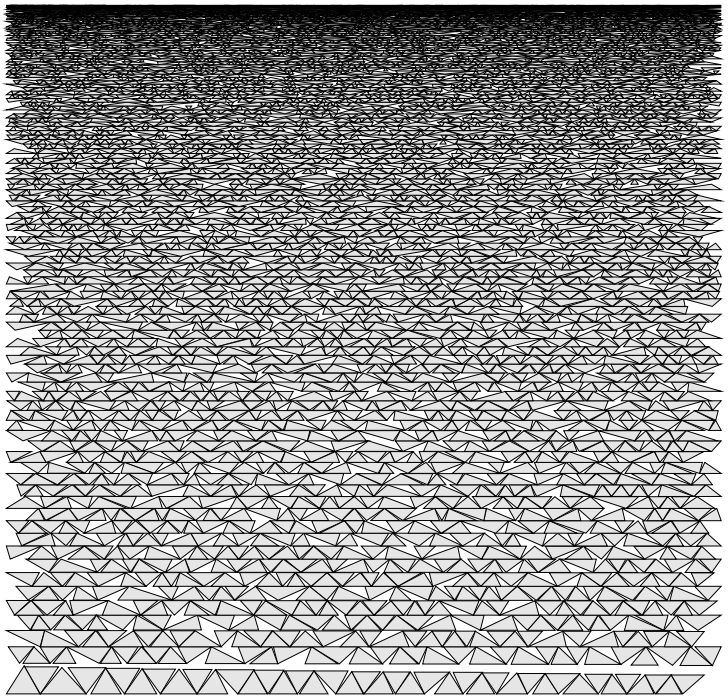


Fig. 3. Strip packing of 7,232 undistorted scalene triangles into a texture atlas.

3.1 Distortion

The distortion of a texture mapping is responsible for the deformation of a fixed image as it is mapped onto a surface. Previous techniques for creating atlases have focused on reducing the distortion of the charts [Samek 1986], either by projection [Bennis et al. 1991], deformation energy minimization [Ma and Lin 1988; Maillot et al. 1993; Lévy and Mallet 1998], or interactive placement [Pedersen 1995; 1996].

Chart images are often complex polygons, and must then be packed (without further distortion) efficiently into the texture map to construct the atlas. Automatic packing methods for complex polygons are improving [Milenkovic 1997], but have not yet surpassed the abilities of human experts in this area.

When a texture has already been defined for a meshed surface, distortion in the texture atlas is accompanied by an equivalent distortion of the texture stored in the texture atlas. Hence, the mesh texture atlas is not directly affected by chart distortion, as observed before [Battke et al. 1996]. Solid texture coordinates are properly interpolated across the chart image in the texture map regardless of the difference in shape between the model-coordinate and the surface-texture-coordinate triangles. Chart distortion affects only the direction, or "grain" of the artifacts, but not their existence, as shown later in Figure 9.

3.2 Discontinuity

Texture atlases are discontinuous along the boundaries of their charts. Texture mapping can reveal these discontinuities as a rendering artifact known as a seam. Seams are pixels in the texture map along the

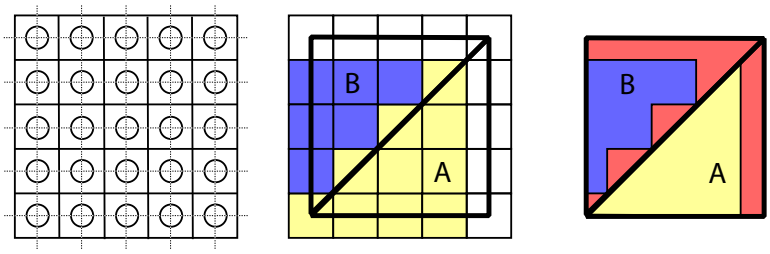


Fig. 4. Texture pixel samples are located at the center of grid cells (a). The rasterization of triangles A and B (shown in blue and yellow) does not cover their entire domain (shown by their boundaries). Because of this difference, nearest-neighbor sampling misclassifies the red pixels (c) resulting in seam artifacts.

edges of charts. They appear along the mesh edges as specks of the wrong color, either the texture map's background color or a color from a different part of the texture.

Previous techniques have reduced seams by maximizing the size and connectivity of the chart images in the texture atlas, e.g. Sander et al. [2001]. These partitions improved the atlas continuity, resulting in fewer charts, though with complex boundaries. While this method reduced seams to the complex boundaries of fewer charts, it did not eliminate them.

Our real-time procedural solid texturing system rasterizes triangles into the texture map using its surface texture coordinates. Seams can appear because the rasterization rules differ from the sampling rules of the texture magnification filter. The rules of polygon scan conversion are designed with the goal of plotting each pixel in a local polygonal mesh neighborhood only once[3]. The rules for texture magnification are designed to appropriately sample a texture when the sample location is not the center of a pixel, usually either the nearest-neighbor sample or bilinear interpolation of the nearest four samples.

Figure 4 demonstrates the disparity between the rasterization and sampling of two triangles A and B. Assume for the moment that integer coordinates in the texture map correspond to the centers of texture samples (the pixels of the texture map) in the configuration shown in (a). Since these integer pixel coordinates occur at the center of the grid cells, the grid cell indicates the set of points whose nearest neighbor is the sample located at the cell's center. Two triangles with integer coordinates are rasterized into the texture map, as shown in (b) using the standard rules [Foley et al. 1990]. Unrasterized pixels remain white. The chart images of triangles A and B are also shown in (b). The texture sampling process needs to reconstruct an approximate sample for any point in these chart image regions. The nearest sample for some points in both regions A and B are white background pixels. The nearest neighbor for some points near the shared hypotenuse in region A are pixels rasterized from plotting B. These regions are indicated in red in (c).

A common solution is to overscan the polygons in the texture map, but surrounding all three edges of each triangle with a one-pixel safety zone wastes valuable texture samples.

Figure 5 illustrates a better solution, adapted from [Rioux et al. 1996]. Triangles A and B have been rasterized into the texture map in (a), as shown before. The surface texture coordinates of the triangles in (b) have been offset by one half pixel but the rasterization of this triangle in (b) generates the same pixels as in (a). The rasterization rules generate the same pixels for the triangles in both (a) and (b), but the texture coordinates interpolated across the background pixels (e.g. white pixels) would be accessed by a

[3]Missing pixels can result in holes or even cracks in the mesh, whereas plotting the same pixel twice (once for each of two different polygons) can cause pixel flashing as neighboring polygons battle for ownership of the pixel on their border.

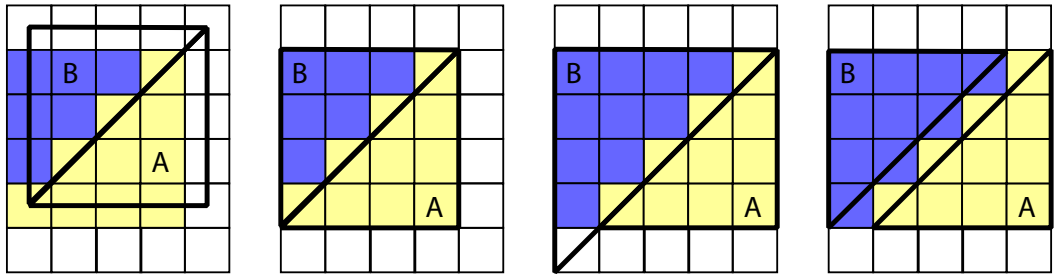


Fig. 5. Standard rasterization rules disagree with texture sampling rules (a). Moving the texture coordinate vertices one-half pixel diagonally aligns the rasterization with the sampling (b), except along the hypotenuse. Rasterizing a shifted A and an enlarged B (c) results in a rasterization that can be sampled correctly using the texture coordinates shown in (d).

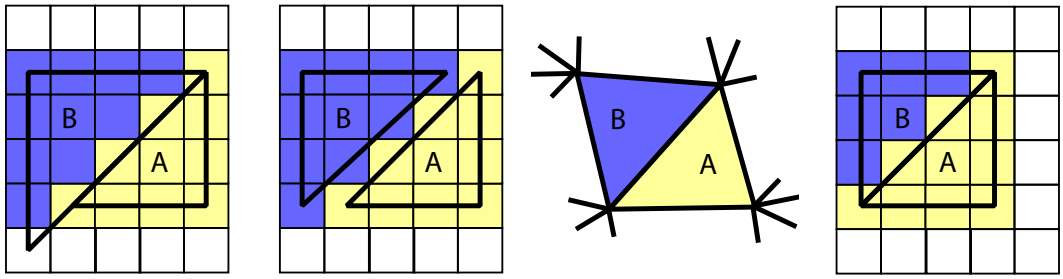


Fig. 6. Reducing the rasterization of the triangles by half a pixel yields the same rasterization (a). Similarly reducing the texture regions (b) leads to partial support for bilinear reconstruction. If A and B are neighboring triangles on the object surface and sharing their hypotenuse edge (c) then support for bilinear reconstruction is complete (d).

nearest-neighbor texture magnification filter in (b). However, a nearest neighbor filter of samples in triangle B near its hypotenuse can still return A's color. The nearest-neighbor misclassification along the hypotenuse is fixed by shifting the texture coordinates of triangle A right by one pixel, and enlarging B by one pixel horizontally and vertically, as shown in (c). The nearest-neighbor sampling of the shifted A and the original B, as shown in (d), now find samples from the appropriate rasterization. This overscanning solution reduces the coverage slightly, but only costs one column of pixels for each triangle pair in a horizontal strip.

If a model contains triangle strips, then these strips can be inserted directly into the uniform mesh atlas without overscanning, as the edge they share has appropriate pixels on either side of it.

Figure 6 illustrates how contracting the triangles by one half pixel yields a result that supports bilinear interpolation of samples. The rasterization remains the same, shown in (a). For points away from the hypotenuse in the reduced triangle regions A and B shown in (b), the four nearest samples (recall the samples are located at the centers of cells) are from the appropriate rasterization. If triangles A and B are neighbors on the object surface and share the edge that forms their hypotenuse, then bilinear interpolation would also be correct near the hypotenuse (and the extra column of pixels would not be necessary). The

multiresolution atlas introduced in Section 5 can support this shared hypotenuse property, such that the resulting atlas can support bilinear magnification filtering of texture samples.

3.3 Coverage

The coverage C of an atlas measures how effectively the parameterization uses the available pixels in the texture map. The coverage ranges between zero and one and indicates the percentage of the texture map covered by the image of the M mesh faces

$$C = \sum_{j=1}^{M} A(\mathbf{u}_{j1}, \mathbf{u}_{j2}, \mathbf{u}_{j3}) \tag{1}$$

where $A()$ returns the area of a triangle. We assume the texture map is a unit square.

The coverage of atlases of packed complex polygons was quite low, covering less than half of the available texture samples in our tests. We also implemented a simple polygon packing method that used a single chart for each triangle. This triangle packing performed much better than the complex polygon packing, but still covered only 70% of the available texture samples. Since distortion does not affect the quality of our procedural solid texturing technique, the next section shows that the chart images of triangles can be distorted to cover most if not all of the available texture samples.

3.4 Relative Scale

Whereas the coverage measures how well the parameterization utilizes texture samples, the relative scale S indicates how evenly samples are distributed across the surface. We measure the relative scale as the RMS of the ratio of the square root of the areas before and after each chart of the atlas is applied

$$S^2 = \left(\sum_{j=1}^{M} A(\mathbf{x}_{j1}, \mathbf{x}_{j2}, \mathbf{x}_{j3}) \right) \frac{1}{M} \sum_{j=1}^{M} \frac{A(\mathbf{u}_{j1}, \mathbf{u}_{j2}, \mathbf{u}_{j3})}{A(\mathbf{x}_{j1}, \mathbf{x}_{j2}, \mathbf{x}_{j3})}. \tag{2}$$

The additional summation factor computes the surface area of the object in model space, and normalizes the relative scale so it can be used as a measure to compare the quality of atlases across different models. A relative scale less than one indicates that the atlas is contracting a significant number of large triangles too severely, whereas a relative scale greater than one indicates that small triangles are taking up too large a portion of the texture map.

The relative scale is similar to the L^2 "stretch" metric [Sander et al. 2001]. Given the triangle $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ and its surface texture coordinates $(\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3)$, let ϕ be the implied affine map such that $\phi(\mathbf{u}_i) = \mathbf{x}_i$, and let J be its 2×2 Jacobian. (The inverse of ϕ is the texture map $\mathbf{u}(\mathbf{x})$ described in the Introduction.) The $L^2(\phi)$ norm was set to the RMS of the two singular values of J, and was extended across the entire surface mesh as

$$L^2 = \sqrt{\frac{\sum_{j=1}^{M} L^2(\phi_j) A(\mathbf{x}_{j1}, \mathbf{x}_{j2}, \mathbf{x}_{j3})}{\sum_{j=1}^{M} A(\mathbf{x}_{j1}, \mathbf{x}_{j2}, \mathbf{x}_{j3})}}. \tag{3}$$

(This was not the first use of the singular values of ϕ. Their squares were present in the trace of the first fundamental form $I_\phi = JJ^T$ used by Maillot et al. [1993] in their minimization of texture distortion.)

Both the relative scale S and stretch L^2 represent the RMS of the area sampling, but the stretch metric can detect anisotropy in the scaling, such as when a sliver surface triangle maps to an equilateral texture triangle of equal area. We use the relative scale to measure the distribution of samples in meshed atlases, and let a length-weighting (adapted later in Section 4.3 from Rioux et al. [1996]) that indicates a form of "stretch" useful for properly distributing anisotropic samples. It is interesting that both the progressive

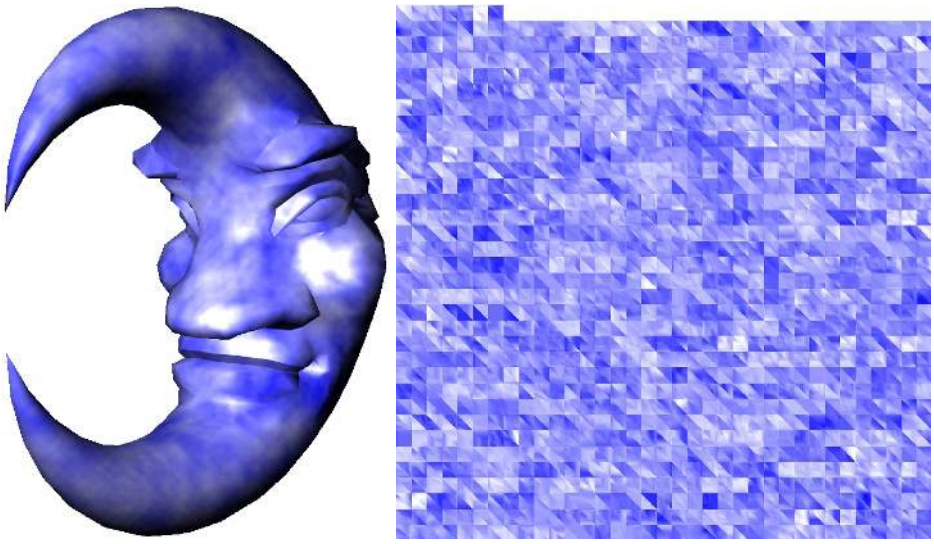


Fig. 7. Uniform mesh atlas for a cloud textured moon.

mesh atlas [Sander et al. 2001] and the multiresolution atlas presented in Section 5 both try to minimize the perimeter of chart boundaries, yielding disk-shaped clusters that avoid anisotropic scaling.

The relative scale is also affected by the coverage. An inefficient packing can scale chart images smaller than necessary in order to make them fit into available texture space.

4. MESH ATLASES FOR SOLID TEXTURING

Our technique for supporting real-time procedural solid texturing focuses on the texture atlas as its basic data structure. This texture atlas allows the entire surface to be represented in a compact subset of the plane, which can be shaded efficiently with a single procedural texturing pass before being re-applied to the object during a standard texture-mapping pass.

Any of the atlases in Sections 2.2 and 2.3 would work for solid texturing, though we have found the atlases designed for appearance-preserving simplification are more effective than those designed for non-distorted texture mapping. In an effort to analyze the performance of these atlases, we have constructed three basic atlases to compare and analyze.

4.1 Uniform Mesh Atlases

One way to take as many samples as possible is to maximize the coverage of texture map by the atlas. This can be done simply by deforming the model triangles into a form that can be easily packed. The uniform mesh atlas arbitrarily maps all of the triangles into the same shape, an isosceles right triangle. These right triangles are packed into horizontal strips and stacked vertically in the texture map.

Figure 7 demonstrates the uniform mesh atlas. Continuity is ignored and the texture map can be thought of as a collection of rubber jigsaw puzzle pieces that must be stretched into an appropriate place on the model surface.

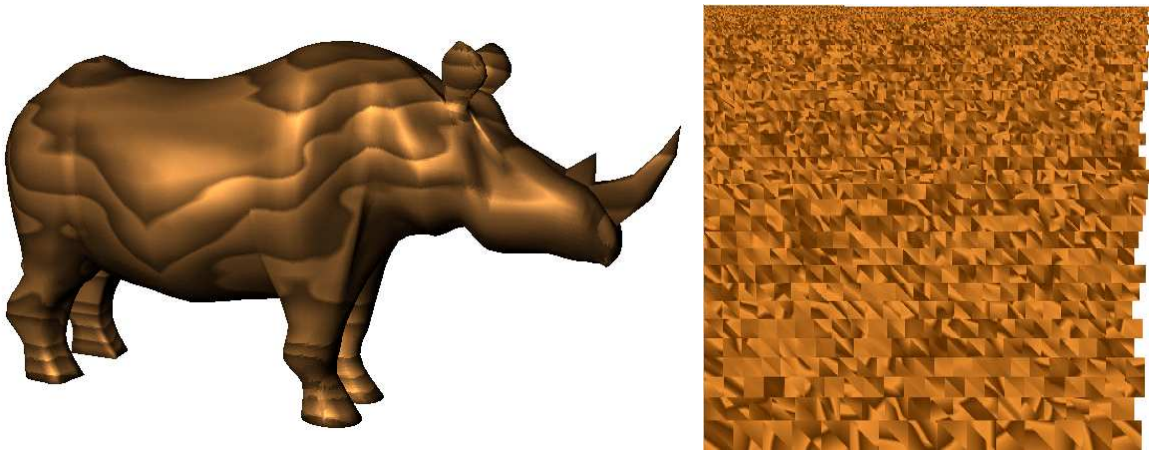


Fig. 8. Rhino sculpted from wood and its area-weighted mesh atlas.

The length of each adjacent edge of the mesh triangles is given by

$$a = \frac{\lfloor \sqrt{M/2} \rfloor}{H} \tag{4}$$

where H is the horizontal resolution of a square texture map. The floor ensures that we can plot a full row of triangle pairs. Unlike previous methods, we do not require a to be a power of two. This provides greater coverage given an arbitrary number of triangles. In the worst case, we waste an $(a-1) \times H$ pixel region of texture memory on the right column and an $H \times (2a-1) - 1.5a$ pixel region on the top row (if it consists of one triangle, taking care not to count the pixels in the wasted column.) We could pack tighter if a was not an integer, but non-integer edge lengths can create problems with seams. We could also pack tighter if a was allowed to vary by one occasionally but the additional coverage improvement hardly seems worth the added implementation complexity.

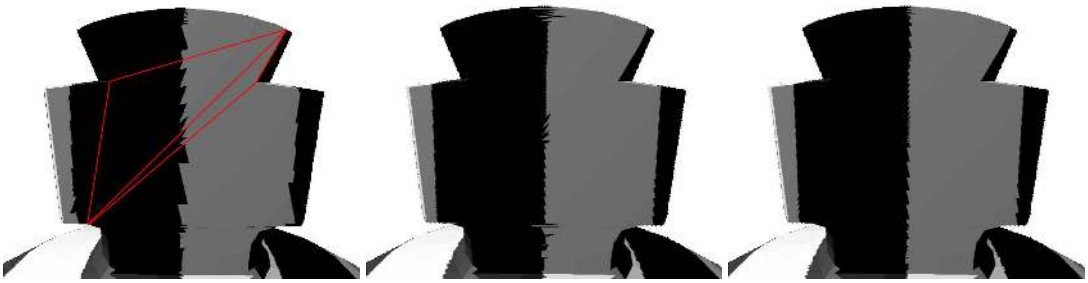
4.2 Area-Weighted Mesh Atlases

While the uniform mesh atlas does a good job of using available texture samples, it distributes those samples unevenly. Object polygons both large and small get the same number of texture samples. The uniform mesh atlas biases the sampling of texture space in favor of areas with small triangles. While smaller polygons may appear in more interesting areas of the model, geometric detail might not correlate with texture detail.

Our goal is to not only use as many samples of the texture as possible, but to distribute those samples evenly across the model. Non-uniform mesh atlases attempt to more evenly distribute texture samples by varying the size of triangle chart images in the texture map.

An obvious criterion is that larger model triangles should receive more texture samples, and so their image under the atlas should be larger. An area-weighted mesh atlas first sorts the mesh triangles by non-increasing area. The mesh atlas is again constructed in horizontal strips, but the size of the triangles in the strip is weighted by the inverse of the relative scale of the triangles in the strip. This allows larger triangles to get more texture samples. Figure 8 demonstrates the area-weighted atlas on a rhino model.

Maruya [1995] also organized triangles in the texture map according to area. The Maruya packing quantized triangles to the nearest power-of-two area, and packed these triangles into triangle blocks as shown in Figure 2(a). Our area-based strip packing also quantizes triangles within a strip to the same area, but this



(a) (b) (c)

Fig. 9. Effects of mesh atlas sample distribution techniques on a poorly tesselated object containing slivers: uniform (a), area weighted (b) and length weighted (c). Some sample polygons are outlined in red in (a) to reveal the correlation between the tessellation and the artifacts.

area need not be a power of two. Removing this power-of-two property results in the jagged edge on the right.

4.3 Length-Weighted Mesh Atlases

Skinny triangles occupy smaller areas, but require extra sampling in their principal axis direction to avoid aliases [Rioux et al. 1996]. A length-weighted mesh atlas uses the triangle's longest edge to prioritize its space utilization in the texture map.

Figure 9 demonstrates the appearance of artifacts from the mesh atlases on the cross of a chess king piece. The procedural texture in this example is a simple striped pattern. Every triangle in the uniform mesh atlas (a) gets the same number of texture samples, regardless of size, resulting in the jagged sampling of the textured stripe on the left. The area-weighted mesh atlas reduces these aliasing artifacts, stealing extra samples from the rest of the model's smaller triangles. But the sliver polygon needs more samples than its area indicates, and the length-weighted mesh atlas gives the sliver triangles the same weight as their neighbors, reducing the aliasing completely, leaving only the artifacts of the nearest-neighbor texture magnification filter.

5. A MULTIRESOLUTION MESH ATLAS

Section 3.2 described how seam artifacts were removed by making rasterization agree with texture magnification. Texture minification also produces aliasing artifacts when projected texture resolution exceeds screen resolution.

The MIP-map [Williams 1983] is the primary method for inhibiting texture minification aliases in modern computer graphics hardware. In order to inhibit texture minification aliases in our real-time procedural solid texturing system, our meshed texture atlases need to be modified to support MIP mapping.

The progressive mesh atlas can also support MIP-mapping by interpolating colors in the crevices between packed chart images [Sander et al. 2001]. But because the chart images were strip packed in size order, areas of separate sections of the surface could be blended together at lower resolution levels of minification. This section describes an alternative clustering method that yields rectangular chart images that pack efficiently and according to a property that preserves spatial coherence across all levels of the MIP map.

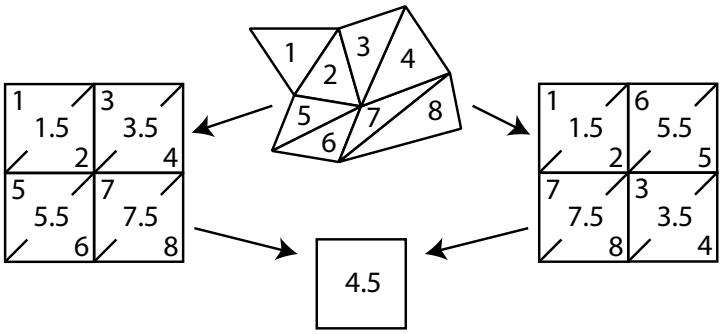


Fig. 10. The triangles on the left are mapped into texture map in the same order they appear in the surface mesh. The triangles on the right are mapped into the texture map in a different order, though the triangles in each quadrant are connected neighborhoods. Averaging quadrants for the next level of the MIP map does not depend on the order of the elements.

5.1 MIP Mapping a Texture Atlas

A MIP-map is a multiresolution pyramid of textures, filtering the texture from full resolution in half-resolution steps down to a single pixel. Each pixel at level l of a MIP-map represents 4l pixels of the full resolution texture map (at level zero).

Assume we have a uniform mesh atlas where the adjacent edge a of each of the triangles is a power of two. Then at levels up to $l_a = \lg a$, some pixels from both sides of a triangle pair will combine into a single pixel. This averaging is correct only if the triangle pair also shares an edge in the surface mesh.

At level l_a+1, four neighboring triangle-pairs in the texture map are averaged together. The uniform mesh atlas cannot be MIP-mapped at level l_a+1 or above as there is no spatial relationship between triangles in the atlas. We can however impose a spatial relationship on the uniform mesh atlas that permits MIP-mapping above level l_a.

At level l_a, triangle pairs are each represented by a single pixel. At level $l_a + 1$, the result of averaging neighboring triangles pairs is a single pixel. Hence, the mesh needs to have neighborhoods of triangle pairs grouped together, but the grouping need not be in any particular order.

This subset property states that each quadrant at levels above l_a of the MIP-map describes a connected neighborhood of mesh elements, but these elements need not adhere to any particular order. Figure 10 illustrates this subset property.

We implement this subset property on the atlas by partitioning the surface mesh hierarchically into an area-balanced quadtree. Each level of the quadtree partitions the mesh into disjoint contiguous sections, as shown later in Figure 15. We implement this quadtree using a face clustering method described in the next section.

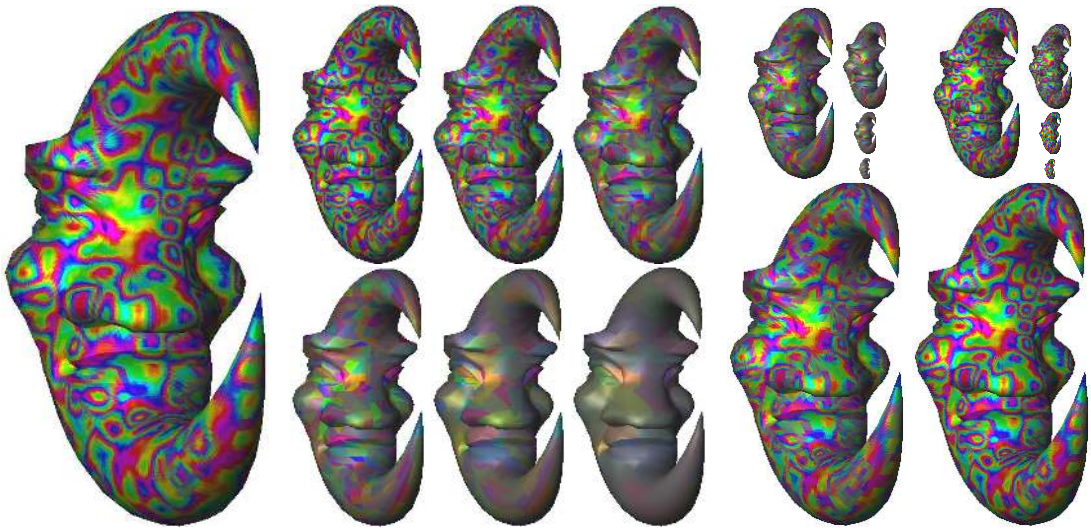
As Figure 11 shows, the square subquandrants of the MIP-map correspond to partitions on the model surface. These partitions are not square and in fact contain acute corners that result in some aliasing artifacts. However, the partitions do contain proximate groups of triangle faces.

We have two other goals in the construction of our quadtree partition of the mesh.

(1) The quadtree should be balanced, yielding a hierarchy of submeshes of equal surface area.

(2) The seam length separating partitions at each level of the quadtree should be minimized.

We used the Metis partitioning algorithm to achieve these goals, yielding balanced partitions such as the one shown in Figure 12.



(a) (b) (c) (d)

Fig. 11. Moon textured with a multiresolution atlas: at full texture resolution (a) and at decreasing MIP map levels (c). Moon rendered at different spatial resolutions with (c) and without (d) the MIP map.

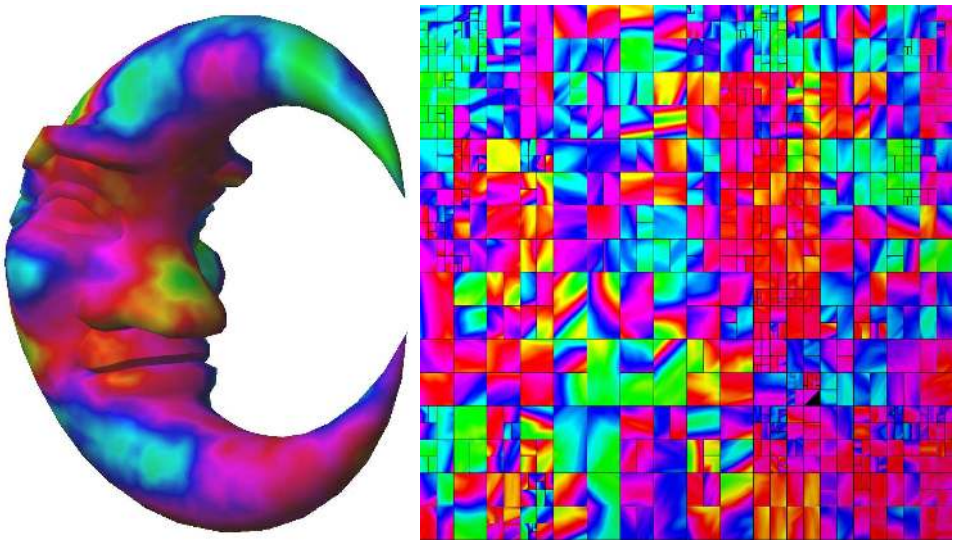


Fig. 12. A tie-dyed moon at full texture resolution (left) and the MIP-map used to generate it (right). (The missing triangle is explained at the end of Section 5.3.)

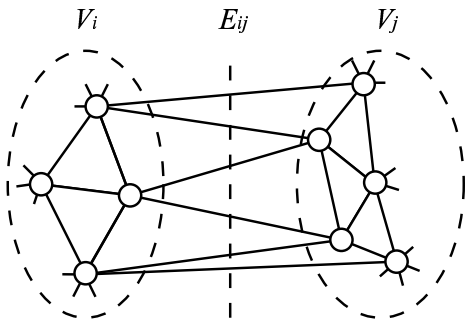


Fig. 13. Balanced disjoint partition of vertices minimizing interconnecting edges.

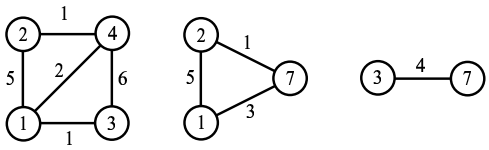


Fig. 14. Weighted edge collapses.

5.2 The Metis Partitioning Algorithm

We cluster the faces of our mesh using the Metis multi-constraint mesh partitioning algorithm [Karypis and Kumar 1998; 1999; Karypis 1999]. Mesh clustering has already found a wide variety of applications in computer graphics, e.g. [Karni and Gotsman 2000; Garland et al. 2001]. The Metis algorithm begins by finding an acceptable partitioning of weighted vertices of a graph simplified by repeated edge collapses. It then tries to preserve this solution as the graph is unsimplified via vertex splits back to its original resolution. Metis can generate face clusters when applied to the dual of the mesh.

Each vertex in a simplification of the dual mesh is weighted by the surface area of the face cluster it represents in the original mesh. The merging of vertices in a simplified dual represents the clustering of faces in the original mesh. Each edge in the dual mesh is weighted by the number of edges it crosses in the original graph.

Every dual vertex v is assigned a weight, denoted $|v|$, corresponding to the surface area of the triangle face it represents in the 3D model. Given a dual $M = (V, E)$ of a given subset of the original mesh, we need to find a disjoint partition $V = V_i \cup V_j$ of its vertices whose summed weights are balanced such that

$$\text{abs}\left(\sum_{v_i \in V_i} |v_i| - \sum_{v_j \in V_j} |v_j|\right) \leq 1 \tag{5}$$

and the cardinality of the set $E_{ij} = (V_i \times Vj) \cap E$ of edges connecting the partition V_i to V_j is at least close to minimal. Such a partition is illustrated in Figure 13.

Figure 14 demonstrates a pair of edge collapses. The weight of a vertex created by an edge collapse is the sum of the collapsed edge's vertex weights. The weight of the collapsed edge is removed ("hidden" inside the vertex), but if an edge collapse merges two edges, then their weights are summed. Hence the weight of each vertex is the area of the cluster of faces it represents, and the weight of each edge is the number of edges it represents.

We perform a heavy-edge matching to determine the order of edge collapse. By collapsing the heaviest

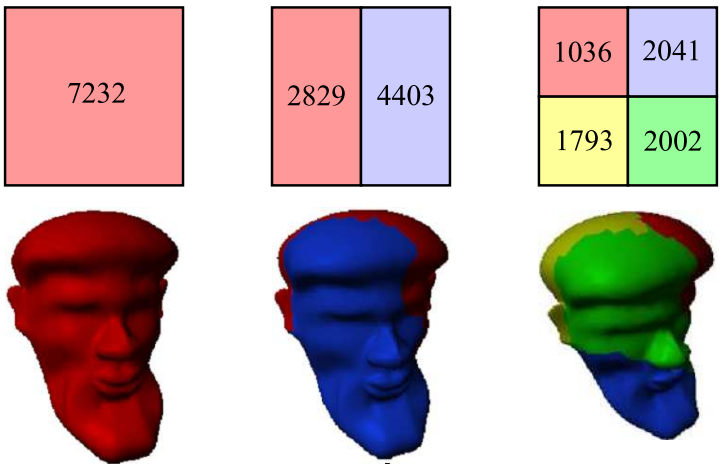


Fig. 15. Recursive partitioning of 2-D texture space (top) and head mesh data set (bottom).

edges first, we effectively hide as many edges in the original base as possible. This leaves edges with lower weights exposed as candidates for E_{ij}, whose collective weights

$$|E_{ij}| = \sum_{e \in E_{ij}} |e| \tag{6}$$

we are trying to minimize in the partitioning phase.

Once the dual graph has been simplified, typically down to about ten percent of its original vertex count, a greedy region-growing algorithm finds a good initial partitioning.

As the simplified dual graph is refined back to its original resolution, the FM refinement method shuffles the partitions to maintain optimality [Fiduccia and Mattheyses 1982]. It prioritizes each vertex in both partitions based on its "gain," the change in $|E_{ij}|$ if the vertex were moved from one partition to the other. Then it repeatedly selects the vertex with the most negative gain in either partition, moves it to the other partition, and recomputes the gain for all affected vertices. If the new partitioning is nearly balanced, then the partition data is saved.

While a direct four-way partitioning algorithm could achieve better results than repeated two-way partitioning [Karypis and Kumar 1999], the simpler two-way partitioning scheme provided satisfactory results. Figure 15 shows the face clusters on the original mesh and their corresponding allocations of texture space (along with the number of faces in the corresponding cluster). These surface area of these partitions is balanced within 1.5%. This process is recursively applied to each partition containing more than four triangles.

5.3 Mapping Triangles into Texture Space

When a partition contains four or fewer triangles, it is mapped directly into 2-D texture space based on its configuration.

Figure 16 shows layouts of six of the eight possible configurations of four or fewer triangles. These layouts are either square or rectangular in texture space, depending on the parity of the current level within the partition tree. Note that all shared edges between triangles internal to the rectangular region are also shared edges between the triangles in object space. This eliminates the necessity to follow the rasterization rules

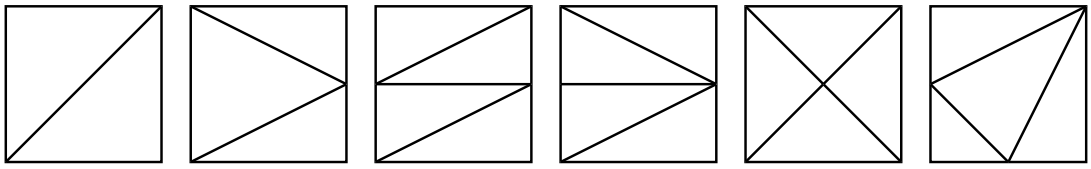


Fig. 16. Base case square atlas partitions for mesh components of two, three or four triangles.

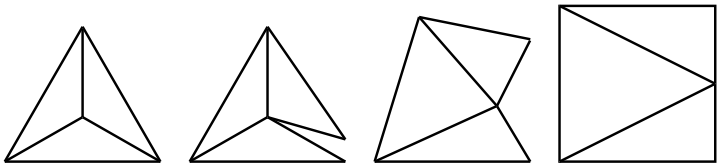


Fig. 17. Cutting a 3-ring into a 3-fan.

mentioned in Section 3.2 within each square/rectangular region of texture space, which resulted in blocks one pixel wider than tall. Since blocks are now square, they easily cover the entire texture map. Furthermore in all but the 2-strip, the location of non-corner texture coordinates may be chosen to reduce sampling bias in the map. Our present implementation chooses these free coordinates as bisections.

Figure 17 shows the 3-ring, one of the two remaining partitions that cannot be directly deformed into a square. The one edge of the 3-ring case must be cut to turn the 3-ring into a 3-fan for it to be laid out in a rectangular region.

The final case is when a partition contains a single triangle. We modified the partitioning algorithm to borrow a triangle from a neighboring partition to ensure all partitions have at least two connected triangles. This slightly reduces the optimality of the partitioning but avoids wasted samples, which was the ultimate goal of the partitioning. Our present implementation will not borrow a triangle when the imbalance is greater than 7:1, which is why a black triangle appears in the atlas in Figure 12. We developed special filters for our MIP-map construction that would ignore samples from these triangle holes during the downsampling process.

5.4 Discussion

Observation and analysis of the multiresolution atlas algorithm has yielded a number of benefits and drawbacks.

— **Total Coverage.** Out of the existing known mesh-atlas generation techniques, none guarantee complete coverage of texture space. The multi-resolution approach represents a complete use of texture information, which results in more texture detail being placed on the model during rendering time.

— **Magnification Filtering.** Texture magnification filtering, particularly for high frequency textures is imperative for reducing temporal and spatial aliasing. The spatial coherence of the partitioning allows for linear texture magnification filters to be applied to the mesh within the rectangular partition boundary. The half-pixel boundary introduced by contracting our partition corner vertices one-half pixel (as described in Section 3.2) creates exactly enough room to support linear magnification filtering at the boundary of the partition. Our experiments verified that bilinear magnification filtering did not yield any seam artifacts.

Table I. Measurement of mesh atlas performance on the head model (7,232 triangles).

Mesh Atlas	Coverage	Relative Scale
Uniform	91%	2.24
Area-Weighted	83.5%	0.94
Length-Weighted	93%	0.88
Multiresolution	100%	1.37

We did however notice that bilinear magnification filtering combined with trilinear minification filtering mysteriously created seam artifacts.

— **Large Models.** Models containing a large number of triangles will necessarily contain triangles whose image in texture space is smaller than a single pixel. In the simple uniform, area and length weighted mesh atlas schemes, triangles are mapped individually and each one must receive at least one texture pixel. In these cases, large numbers of very small triangles incrementally steal samples needed by larger triangles. The multiresolution mesh atlas algorithm creates proximate triangle neighborhoods. Small neighborhoods containing many small triangles can be mapped into the area of a single texture pixel. This frees up more samples that can be used to better sample neighborhoods with larger surface area.

— **Atlas Compression.** The 2-, 3- and 4-triangle partitions created by the multiresolution mesh atlas can be easily compressed by storing an index of the partition type. Also, since neighboring triangles in the partitions share vertices, they require the storage of fewer vertices per face than for example the uniform or non-uniform mesh atlases which require the storage of all three vertices per face.

— **Efficient Rendering.** All but one of these partitions can also be sent down the graphics pipeline as strips or fans, which can potentially increase rendering throughput of the texture atlas (and the object itself). The 4-ring becomes a 4-fan when one triangle is chosen to be transmitted first. The 4-star can not be transmitted efficiently, and ends up as a 3-fan and an isolated triangle.

— **Sliver Sensitivity.** While providing more texels per triangle has produced better visual results in general, models containing sliver triangles appear aliased. Retessellation of such models was required to achieve satisfactory results. Our method completely ignored texture distortion and measured atlas quality completely in terms of sampling area ratios. Integration of distortion and/or use of the L^2 stretch metric [Sander et al. 2001] could overcome this sliver sensitivity.

— **Manifold Restriction.** The multiresolution mesh atlas scheme requires the mesh to be a single manifold (or manifold with boundary), but this manifold can be of arbitrary genus. In the case where a model contains numerous disjoint meshes, an atlas for each mesh must be created individually.

6. RESULTS

We have tested the mesh atlas algorithms on several datasets. We measured and compared their performance with respect to coverage and relative scale. Coverage indicates the global utilization of available texture samples whereas relative scale indicates the distribution of texture samples across the model surface.

6.1 Global Statistics

Table I shows the global statistics: coverage and relative scale for the head mesh model. A global relative scale of one is ideal which means all triangles receive texture samples in proportion to their area.

The uniform approach is the easiest to implement but performs poorly in both utilizing available samples and distributing those samples. Its global relative scale of 2.24 means that the average triangle received twice as many samples as it should have. This occurs in typical models that consist of few large triangles and many small triangles, all receiving the same number of texture samples.

The area-weighted mesh atlas distributes samples more fairly than any of the other techniques, though the jagged edges of this atlas's triangle-pair blocks cause its coverage to suffer.

The multiresolution mesh atlas uses all available texture samples, though its global relative scale reveals that its distribution slightly oversamples on average. This is presumably due to small triangles that are stretched slightly larger in partition blocks.

Because the chart images are rectangular, the coverage for the multiresolution atlas is 100% which means all of the available texture samples are used. The progressive mesh atlas [Sander et al. 2001] can also be MIP mapped, but its coverage was at most about 70%.

6.2 Per Triangle Relative Scale

After examining the global metrics we examined the local characteristics for the atlas schemes. Figure 18 and Figure 19 plot the relative scale of each triangle for the head and wineglass models respectively.

The relative scale for each triangle i was computed as

$$S_i = \frac{A(\mathbf{u}_{i1}, \mathbf{u}_{i2}, \mathbf{u}_{i3}) \sum_{j=1}^{M} A(\mathbf{x}_{j1}, \mathbf{x}_{j2}, \mathbf{x}_{j3})}{A(\mathbf{x}_{i1}, \mathbf{x}_{i2}, \mathbf{x}_{i3})} \tag{7}$$

and should ideally equal one. A relative scale of one is indicated in white. A relative scale $S_i < 1$ indicates an undersampled triangle which is tinted blue using the RGB color $(S_i, S_i, 1)$. A relative scale $S_i > 1$ indicates an oversampled triangle which is tinted red using the RGB color $(1, 1/S_i, 1/S_i)$.

The uniform scheme under-sampled the larger triangles on the top of the wineglass, placing more texture detail on the stem of the glass where the smaller triangles reside. Both the area-weighted schemes give ample area weights for the larger triangles, but greatly under-sample the smaller ones. This leads to poor texture resolution on the base of the glass and on the stem.

The area-weighted schemes and length-weighted schemes performed well when their performance is amortized by a global metric but their variance is much greater than that of the multiresolution atlas. They did not work well on models with large numbers of triangles because even the smallest triangles were still mapped to at least one pixel in the texture map, which reduced the area of the atlas available to larger triangles.

Visually, even without applying linear texture magnification filtering to our models, the multi-resolution scheme performed the best, yielding continuous texture detail across the model. Smaller triangles were collected into larger clusters, which allowed smaller triangles to map to regions smaller than one pixel in the atlas. This provided a more balanced allocation of the available texture samples.

7. INTERACTIVE PROCEDURAL SOLID TEXTURE DESIGN

We used the methods described in this paper to create a procedural solid texture design system that allowed the user to interactively modify procedural solid textures applied to a given 3-D object. The interactivity of this system depends on the speed of the components of the procedural solid texturing process.

The synthesis of the texture atlas for a given object is generally not interactive and is performed as a preprocessing step. Its execution depends on the size of the mesh. The simple atlases of Section 4 can be run interactively on simple objects. Our implementations ran in about 0.2 seconds on the 2,324-polygon moon model, but took nearly a full second for the 7,232-polygon head model. The multiresolution atlas is much slower and more sensitive to model size, running in one minute for the moon and in 12 minutes for the head. Changing the connectivity of mesh vertices requires recomputation of the atlas. Changing to the position of mesh vertices does not invalidate the meshed atlas, but can affect the quality of the area-preserving and length-weighted atlases, and can affect the area balance of the multiresolution atlas.

Once generated, the atlas provides an efficient and direct method for applying procedural textures to the object. As described in the introduction, the atlas is rasterized by plotting the triangles using their surface

Fig. 18. Head model (7,232 triangles) per-triangle relative scale (top row, white: perfect sampling, red: over-sampled, blue: under-sampled) and with procedural texture applied (bottom row).

texture coordinates and setting their color to their solid texture coordinates. The texturing procedure is applied to the solid texture coordinate colors interpolated across the polygon faces in the texture map. The procedure replaces these coordinates with colors producing a texture map that when applied yields a procedural solid texturing of the object. Since standard texture mapping applies the results of the texturing procedure, static procedure textures can be rendered in real time.

Changing the procedural solid texture parameters requires the texturing procedure to be re-evaluated on the pixels in the texture atlas rasterization. This is the main feature of an interactive procedural solid texturing design, and requires a fast implementation of the texturing procedure to support interactive manipulation.

Procedural textures can be generated in a number of ways. We explored two basic techniques. One class of techniques ran the texture procedure sequentially on the host whereas a second class of techniques compiled the procedure into a multipass program executed in SIMD fashion by the graphics controller [Heidrich and Seidel 1999; McCool and Heidrich 1999; Peercy et al. 2000; Proudfoot et al. 2001].

Host procedures provided the highest level of flexibility, allowing all of the benefits of a high-level language compiled into a broad instruction set. Several fast host-processor methods exist for synthesizing procedural

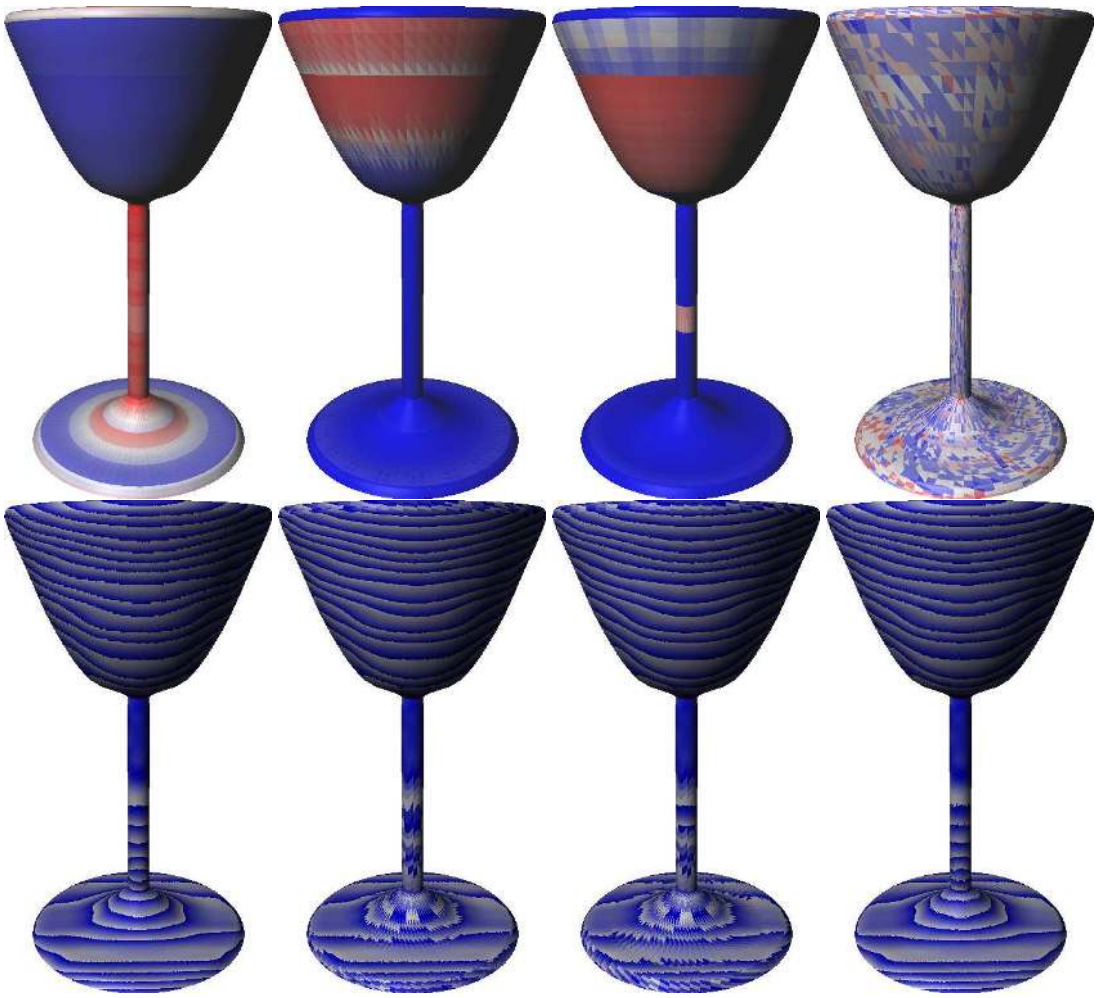


Fig. 19. Wineglass model (42,768 triangles) per-triangle relative scale (top row, white: perfect sampling, red: over-sampled, blue: under-sampled) and procedural texture applied (bottom row).

textures. Goehring and Gerlitz. [1997] implemented a smooth noise function in Intel MMX assembly language, evaluating the function on a sparse grid and using quadratic interpolation for the rest of the values. [Kameya and Hart 2000] used streaming SIMD instructions that forward differenced a linearly interpolated noise function for fast rasterization of procedurally textured triangles.

One could use the graphics processor to rasterize the texture atlas, and then let the host processor replace the interpolated solid coordinates with procedural texture colors. The main drawback to this technique is the asymmetry of the graphics bus, which is designed for high speed transmission from the host to the graphics card. The channel from the graphics card to the host is very slow, taking nearly a second to perform an

Table II. Execution times for procedural texture synthesis into the texture atlas. Parenthetic times measure lower resolution synthesis during interaction.

Noise Octaves	Atlas Resolution	Procedural Synthesis Speed
1	256^2	9.09 Hz (18 Hz)
1	512^2	2.56 Hz (4.55 Hz)
1	1024^2	0.72 Hz (1.30 Hz)
4	256^2	6.25 Hz (10 Hz)
4	512^2	1.82 Hz (3.03 Hz)
4	1024^2	0.40 Hz (0.76 Hz)

OpenGL ReadPixels command on an Intel PC AGP bus.

To overcome this bottleneck, our host-procedure implementation uses the host to rasterize the atlas directly into the texture map. Host rasterization provides full control over the rasterization rules and full precision for the interpolated texture coordinates. While the host processor is not nearly as fast as the graphics processor at rasterization, the generation and rendering of the atlas into texture memory can still be performed at interactive speeds.

The software procedural texture renderer simultaneously rasterized the texture atlas into texture memory and applied the texturing procedure to the texture atlas. We increased the responsiveness of our system by having this renderer generate a lower resolution interpolated version of the atlas during manipulation, and replace it with a higher resolution version at rest. The rendering speed of this system is shown in Table II.

8. CONCLUSION

We have shown how the texture atlas can facilitate the real-time application of solid procedural texturing. We showed that for this application, texture atlas construction should focus more on sampling fidelity and less on distortion and discontinuity. We introduced new mesh-based atlas generation schemes that more efficiently used available texture samples and distributed these samples more evenly across the object. The multiresolution mesh also supports MIP-mapping of the texture atlas, and coupled with the contracted rasterization of triangles, supports linear magnification filtering.

The texture atlas allows solid texturing procedures to be applied to the texture map, allowing efficient multipass programming using the accelerated operations available on the graphics controller as they become feasible.

The system makes effective use of preprocessing. The procedural texture needs to be resynthesized only when its parameters change, and the texture atlas needs to be reconstructed only when the object changes shape. Specifically, if the position of the object's vertices move, but the topology of the mesh remains invariant, then the procedural solid texturing generated by this method will adhere to the surface [Apodaca and Gritz 1999]. This is a useful property that prevents texture "swimming," such that for example the grain of a warped wood plank follows the warp of the plank.

8.1 Applications

We have focused this paper on the application of real-time procedural solid texturing, though the techniques described appear to impact other areas as well.

— **Solid Texture Encapsulation.** Unlike surface texture coordinates, solid texture coordinates are not uniformly implemented by graphics file formats. Using surface texture of a solid texture allows the texture coordinates to be more robustly specified in object files and also allows the solid texture to be included as a more compact texture map image instead of a wasteful 3-D solid texture array.

— **3-D Painting.** The meshed atlas techniques can also be used to support 3-D painting onto surfaces [Hanrahan and Haeberli 1990]. Since the atlas is a one-to-one map, it provides an automatic parameterization. The discontinuities of the parameterization do not impact painting as the texture atlas maintains a per face correspondence between the surface and the texture map. The meshed atlas techniques presented in Section 4 also improve surface painting by using as many texture samples as possible distributed evenly across the surface. Alternatively, a different metric than area could be used to redistribute samples to more finely sample the painted texture in areas of fine detail.

— **Appearance-Preserving Simplification.** As shown in Section 2.3, a number of atlases were originally constructed for appearance preserving simplification. The atlases described in this paper were designed for procedural solid texturing, but would also be directly applicable to simplification. The primary difference is that the information for the texture map comes from the vertices of the unsimplified mesh instead of a texturing procedure. This adds a new sampling criterion to the atlas generation problem that was not yet exploited. (In addition, this paper did not examine the spectral properties of the procedural texture to determine if it too should be sampled non-uniformly.)

— **Normal Maps.** The normal map [Fournier 1992; Cohen et al. 1998] is a texture map whose pixels hold a surface normal instead of a color. Normal maps are used for real-time per-pixel bump mapping using dot-product texture combiners found in Direct3D and extensions of OpenGL. The meshed atlas generation techniques can be used to create well-sampled normal maps since normal maps do not require continuity between faces.

— **Real-Time Shading Languages.** Recent real time shading languages [Peercy et al. 2000; Proudfoot et al. 2001] have been developed to support procedural shaders, including texturing and lighting, by converting shader descriptions into multipass graphics library routines. In particular, Proudfoot et al. [2001] focuses on the difference between per object, per vertex and per fragment processes in real-time shaders. The texture atlas supports additional categories of view-dependent and view-independent processes. View dependent processes utilize multipass operations to the framebuffer, whereas view independent processes utilize multipass operations to the texture map. The results of view independent processes can be stored and accessed directly from the texture map, accelerating the rendering of real time shading language shaders.

8.2 Future Work

While this work achieved our goal of real-time procedural solid texturing, it has also inspired several directions for further improvement.

— **Direct Manipulation of Procedural Textures.** The interactive procedural solid texture design system is a first step. Another step would be to allow the sliders to be bypassed, supporting direct manipulation of procedural textures. The user could drag a texture feature to a desired location and have the software automatically reconfigure the parameters appropriately.

— **Preservation of Mesh Structure.** The mesh atlases do not preserve the object's original mesh structure, and our mesh atlas processing program outputs multiple copies of shared mesh vertices with different surface texture coordinates. This increases the size of the model description files, and may cause the resulting models to render more slowly. Preservation of mesh structure, or at least triangle strips, would be a useful addition to this stage of the process.

— **Atlas Compression.** The texture atlas resembles the codebook used in vector quantization. The number of faces in the atlas could be reduced by allowing the atlas to no longer be one-to-one, and to let triangles with similar procedural texture features to map to the same location in the texture atlas. This kind of atlas compression would increase the number of available texture samples with larger chart images in the texture atlas.

ACKNOWLEDGMENTS

This research was funded in part by the Evans & Sutherland Computer Corp. overseen by Peter K. Doenges. The research was performed using facilities at both Washington State University and the University of Illinois. Jerome Maillot was instrumental in showing us the state of the art in this area, including Alias|Wavefront's work. Pat Hanrahan observed that MIP-mapping a uniform mesh atlas biased the partition colors in favor of smaller triangles, which led us to investigate partitioning with respect to surface area instead of number of triangles.

REFERENCES

Apodaca, A. A. and Gritz, L. 1999. *Advanced Renderman: Creating CGI for Motion Pictures*. Morgan Kaufmann.

Battke, H., Stalling, D., and Hege., H.-C. 1996. Fast line integral convolution for arbitrary surfaces in 3d. Tech. Rep. SC-96-59, Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB). Dec.

Bennis, C., Vézien, J.-M., Iglésias, G., and Gagalowicz, A. 1991. Piecewise surface flattening for non-distorted texture mapping. *Computer Graphics (Proc. SIGGRAPH 91) 25,* 4 (July), 237–246.

Brinsmead, D. 1993. Convert solid texture. Software component of Alias|Wavefront Power Animator 5.

Cignoni, P., Montani, C., Rocchini, C., and Scopigno, R. 1998. A general method for preserving attribute values on simplified meshes. In *Proc. Visualization '98*. IEEE, 59–66.

Cohen, J., Olano, M., and Manocha, D. 1998. Appearance-preserving simplification. In *Proc. SIGGRAPH 98*. 115–122.

Ebert, D., Musgrave, F., Peachey, D., Perlin, K., and Worley, S. 1994. *Texturing and Modeling: A Procedural Approach*. Academic Press.

Fiduccia, C. M. and Mattheyses, R. M. 1982. A linear-time heuristic for improving network partitions. In *Proc. 19th IEEE Design Automation Conference*. 175–181.

Foley, J., van Dam, A., Feiner, S., and Hughes, J. 1990. *Computer Graphics: Principles and Practice*, 2nd ed. Addison-Wesley.

Fournier, A. 1992. Normal distribution functions and multiple surfaces. In *Proc. Graphics Interface '92 Workshop on Local Illumination*. Canadian Information Processing Society, 45–52.

Garland, M., Willmott, A., and Heckbert, P. S. 2001. Hierarchical face clustering on polygonal surfaces. In *Symposium on Interactive 3D Graphics*. ACM, 49–58.

Goehring, D. and Gerlitz., O. 1997. Advanced procedural texturing using MMX technology. Tech. rep., Intel MMX Technology Application Note. Oct.

Hanrahan, P. 1999. Procedural shading (keynote). Eurographics / SIGGRAPH Workshop on Graphics Hardware.

Hanrahan, P. and Haeberli, P. E. 1990. Direct wysiwyg painting and texturing on 3d shapes. *Computer Graphics (Proc. SIGGRAPH 90) 24,* 4 (Aug.), 215–223.

Hanrahan, P. and Lawson, J. 1990. A language for shading and lighting calculations. *Computer Graphics (Proc. SIGGRAPH 90) 24,* 4 (Aug.), 289–298.

Hart, J. C., Carr, N., Kameya, M., Tibbitts, S. A., and Coleman, T. J. 1999. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. In *Proc. Graphics Hardware Workshop*. SIGGRAPH/Eurographics, 45–53.

Heidrich, W. and Seidel, H.-P. 1999. Realistic, hardware-accelerated shading and lighting. In *Proc. SIGGRAPH 99*. 171–178.

Kameya, M. and Hart, J. 2000. Bresenham noise. In *SIGGRAPH 2000 Conference Abstracts and Applications*.

Karni, Z. and Gotsman, C. 2000. Spectral compression of mesh geometry. In *Proc. SIGGRAPH 2000*. 279–286.

Karypis, G. 1999. Multi-level algorithms for muli-constraint hypergraph partitioning. Tech. Rep. 99-034, University of Minnesota. Nov.

Karypis, G. and Kumar, V. 1998. Multilevel algorithms for multi-constraint graph partitioning. In *Proc. Supercomputing 98*.

Karypis, G. and Kumar, V. 1999. Multilevel k-way hypergraph partitioning. In *Proc. Design Automation Conference*. IEEE, 343–348.

Lastra, A., Molnar, S., Olano, M., and Wang, Y. 1995. Real-time programmable shading. In *Proc. Symposium on Interactive 3D Graphics*. ACM SIGGRAPH, 59–66.

Lee, A. W. F., Sweldens, W., Schröder, P., Cowsar, L., and Dobkin, D. 1998. Maps: Multiresolution adaptive parameterization of surfaces. In *Proc. SIGGRAPH 98*. 95–104.

Lévy, B. and Mallet, J.-L. 1998. Non-distorted texture mapping for sheared triangulated meshes. In *Proc. SIGGRAPH 98*. 343–352.

Ma, S. D. and Lin, H. 1988. Optimal texture mapping. In *Proc. Eurographics '88*. North-Holland, 421–428.

Maillot, J., Yahia, H., and Verroust, A. 1993. Interactive texture mapping. In *Proc. SIGGRAPH 93*. 27–34.

Maruya, M. 1995. Generating a texture map from object-surface texture data. *Computer Graphics Forum 14,* 3 (Aug.), 397–406.

McCool, M. D. and Heidrich, W. 1999. Texture shaders. In *Proc. Graphics Hardware Workshop*. SIGGRAPH/Eurographics, 117–126.

Microsoft Corp. 2000. Direct3d 8.0 specification. http://www.msdn.microsoft.com/directx.

Milenkovic, V. 1997. Rotational polygon overlap minimization. *Computational Geometry: Theory and Applications 10*, 305–318.

Molnar, S., Eyles, J., and Poulton, J. 1992. Pixelflow: High-speed rendering using image composition. *Computer Graphics (Proc. SIGGRAPH 92) 26,* 2 (July), 231–240.

Munkres, J. R. 1975. *Topology: a first course*. Prentice-Hall, Englewood Cliffs, New Jersey.

Norton, A., Rockwood, A. P., and Skolmoski, P. T. 1982. Clamping: A method of antialiasing textured surfaces by bandwidth limiting in object space. *Computer Graphics 16,* 3 (July), 1–8.

NVidia Corp. 2000. Noise. Component of the NVEffectsBrowser.

Olano, M. and Lastra, A. 1998. A shading language on graphics hardware: The pixelflow shading system. In *Proc. SIGGRAPH 98*. 159–168.

Peachey, D. R. 1985. Solid texturing of complex surfaces. *Computer Graphics 19,* 3 (July), 279–286.

Pedersen, H. K. 1995. Decorating implicit surfaces. In *Proc. SIGGRAPH 95*. 291–300.

Pedersen, H. K. 1996. A framework for interactive texturing operations on curved surfaces. In *Proc. SIGGRAPH 96*. 295–302.

Peercy, M. S., Olano, M., Airey, J., and Ungar, P. J. 2000. Interactive multi-pass programmable shading. In *Proc. SIGGRAPH 2000*. 425–432.

Perlin, K. 1985. An image synthesizer. *Computer Graphics 19,* 3 (July), 287–296.

Perlin, K. and Hoffert, E. M. 1989. Hypertexture. *Computer Graphics 23,* 3 (July), 253–262.

Pixar Corp. 1999. Future requirements for graphics hardware. Memo.

Potmesil, M. and Hoffert, E. M. 1989. The Pixel Machine: A parallel image computer. *Computer Graphics (Proc. SIGGRAPH 89) 23,* 3 (July), 69–78.

Praun, E., Finkelstein, A., and Hoppe, H. 2000. Lapped textures. In *Proc. SIGGRAPH 2000*. 465–470.

Proudfoot, K., Mark, W. R., Tzvetkov, S., and Hanrahan, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proc. SIGGRAPH 2001*. 159–170.

Rhoades, J., Turk, G., Bell, A., State, A., Neumann, U., and Varshney, A. 1992. Real-time procedural textures. *Computer Graphics (Proc. Symp. Interactive 3D Graphics) 25,* 2 (Mar.), 95–100.

Rioux, M., Soucy, M., and Godin, G. 1996. A texture-mapping approach for the compression of colored 3d triangulations. *The Visual Computer 12,* 10, 503–514.

Samek, M. 1986. Texture mapping and distortion in digital graphics. *The Visual Computer 2,* 5, 313–320.

Sander, P. V., Snyder, J., Gortler, S. J., and Hoppe, H. 2001. Texture mapping progressive meshes. In *Proc. SIGGRAPH 2001*. 409–416.

Segal, M. and Akeley, K. 1999. The opengl graphics system: A specification, version 1.2.1. http://www.opengl.org/.

Thorne, C. 1997. Convert solid texture. Software component of Alias|Wavefront Maya 1.

Williams, L. 1983. Pyramidal parametrics. *Computer Graphics (Proc. SIGGRAPH 83) 17,* 3 (July), 1–11.