Real-Time Procedural Solid Texturing

Nathan A. Carr John C. Hart

Department of Computer Science
University of Illinois, Urbana-Champaign

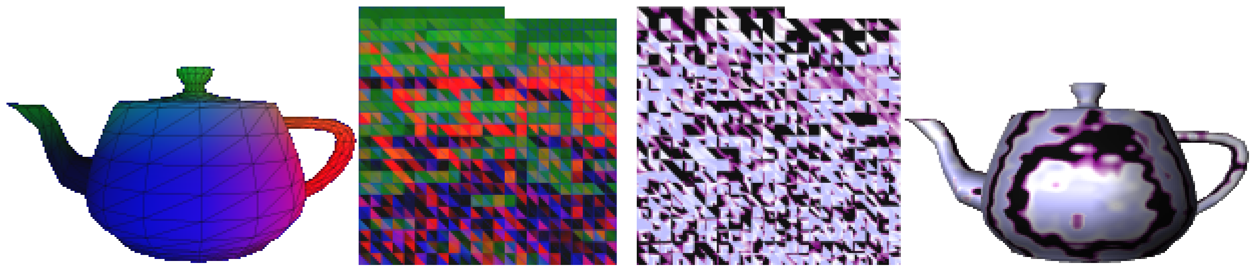

Figure 1. Solid texture coordinates stored as vertex colors of a model (a) are rasterized into a texture atlas (b). A procedural shader replaces the interpolated solid texture coordinates with colors (c), which are applied to the object using texture mapping.

Abstract

Shortly after its introduction in 1985, procedural solid texturing became a must-have tool in the production-quality graphics of the motion-picture industry. Now, over fifteen years later, we are finally able to provide this feature for the real-time consumer graphics used in videogames and virtual environments. A texture atlas is used to create a 2-D texture map of the 3-D solid texture coordinates for a given surface. Applying the procedural texture to this atlas results in a view-independent procedural solid texturing of the object.

Texture atlases are known to suffer from sampling problems and seam artifacts. We discovered that the quality of this texturing method is independent of the continuity and distortion of the atlas, which have been focal points of previous atlas techniques. We instead develop new meshed atlases that ignore continuity and distortion in favor of a balanced distribution of as many texture samples as possible. These atlases are seam-free due to careful attention to their rasterization in the texture map, and can be MIP-mapped using a balanced mesh-clustering algorithm.

Techniques for fast procedural synthesis are also investigated, using either the host processor or with multipass graphics processor operations on the texture map. We used these atlas and synthesis techniques to create a real-time procedural solid texture design system.

CR Categories: I.3.7 [Computer Graphics] Three-Dimensional Graphics and Realism (color, shading and texture).

Keywords: Atlas, mesh partitioning, MIP-map, multipass rendering, procedural texturing, solid texturing, texture mapping.

1. Introduction

The concept of procedural solid texturing is well known [32][37], and has found widespread use in graphics [6]. Solid texturing simulates a sculpted appearance and directly generates texture coordinates regardless of surface topology. Procedural texturing makes solid texturing practical by computing the texture on demand (instead of accessing a stored volumetric array), and 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 [11].

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. [12][38]. Real-time procedural shaders would make videogame graphics richer, virtual environments more realistic and modeling software more faithful to its final result. Section 2 describes previous implementations of real-time procedural texturing and shading systems, all requiring special-purpose graphics supercomputers or processors.

Peercy *et al.* [35] 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.

Unfortunately, ISL did not introduce 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. Applying procedural texturing operations to an entire texture volume also wastes processing time.

Apodaca [1] described how the texture map can be used to store the shading of a model. His technique shaded 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.

We can use this technique to support view-independent procedural solid texturing. Consider a single triangle with 3-D solid texture

Authors' address: Urbana, IL 61801. {nacarr, jch}@uiuc.edu.

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. We apply 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.

This atlas technique was implemented as a tool to preview procedural solid textures in recent modeling packages [2], [45] though it suffered from sampling problems. Lapped textures [40] 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 3 describes the texture atlas in detail, and analyzes the artifacts it can cause. Poor coverage of the texture map by the atlas causes aliasing, whereas discontinuities in the atlas cause seams in the textured surface. Section 4 describes new atlases that overcome these artifacts, with atlases that cover more of the texture map and distributing the resulting samples more evenly to reduce texture magnification aliases. Section 4.3 describes how an atlas that can be MIP mapped to eliminate texture minification aliases.

The use of an atlas enables procedural texturing operations to be applied to the texture map, and Section 5 describes how this step can be implemented efficiently on both the host and the graphics controller. Section 6 concludes with an interactive procedural solid texture editor, other applications of these methods and ideas for further investigation.

2. Previous Work

There have been several implementations of real-time procedural solid texturing over the past fifteen years, though they have either required high-performance graphics computers or special-purpose graphics hardware.

Procedural solid texture has been available on parallel graphics supercomputers, such as the AT&T Pixel Machine [39] and UNC's Pixel Planes 5 and PixelFlow [26]. The Pixel Machine in fact was used as a platform for exploring volumetric procedural solid texture spaces [36].

Rhoades *et al.* [42] 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 *et al.* [30] implemented a real-time subset of the Renderman shading language on Pixel Flow, including the ability to synthesize procedural solid textures. Standard Renderman shader tools including automatic differentiation and clamping [28] were used to antialias the procedural textures.

Hart *et al.* [14] 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 [42].

Current graphics libraries such as OpenGL [44] and Direct3D [24] 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.

Peercy et al. [35] developed a compiler that translated the Renderman shading language into OpenGL source code. The technique used multi-pass rendering and requires an OpenGL 1.2 implementation with its imaging subset, as well as the floating-point-framebuffer and pixel-feedback extensions. As mentioned in the introduction this method depends on texture volumes for solid texturing.

3. The Texture Atlas

A (*surface*) *texture mapping* $\mathbf{u} = f(\mathbf{x})$ 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 $f_i()$ 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 $f() = \cup f_i()$ is called an *atlas* [27]. If the surface texture mapping is one-to-one, then its inverse $f^{-1}()$ 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 [19] 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. Texture atlases do not require the continuity and smooth differentiability that good parameterization strive for.

Texture atlases have strived instead to minimize the distortion of its charts, and to minimize areas of discontinuity between chart images. Section 3.1 shows that distortion does not affect the quality of our method. Section 3.2 describes how discontinuities can cause seam artifacts, but we eliminate these artifacts later in Section 4.1. We instead offer two new measures of atlas quality: coverage (Sec. 3.3) and relative scale (Sec. 3.4), that are used to indicate the sampling fidelity offered by the atlas. Section 4 proposed new atlas techniques that perform well with respect to these two new measures.

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.

[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.

[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.

Previous techniques for creating atlases have focused on reducing the distortion of the charts [43], either by projection [1], deformation energy minimization [20][21][22], or interactive placement [33][34].

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 [25], but have not yet surpassed the abilities of human experts in this area.

Our use of a texture atlas for solid texturing is not directly affected by chart distortion. 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 will be shown later in Figure 6.

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 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. For example, Maillot *et al.* [22] merged portions of the surface of similar curvature. 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.

Seams appear because the rasterization rules differ from texture magnification rules. 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 nearest neighbor or a higher order interpolation of the surrounding pixels.

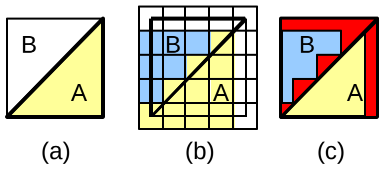

Figure 2. Seams occur due to differences between texture magnification (a) and rasterization (b), shown in red (c).

Figure 2a shows two triangles with integer coordinates in the texture map. Figure 2b shows these two triangles rasterized using the standard rules [7], with unrasterized white pixels in the background. In this figure, the integer pixel coordinates occur at the center of the grid cells. Hence the grid cell indicates the set of points whose nearest neighbor is the pixel located at the cell's center. Figure 2b illustrates that some points in both triangles A and B have background pixels as nearest neighbors, and some points in triangle B have pixels rasterized as triangle A because triangle A's pixels are their nearest neighbors. Figure 2c indicates these points in red.

Higher order texture magnification, such as bilinear or bicubic can reduce but not eliminate the effect of background pixels, and actually exaggerate the problem along the shared edge between triangles A and B. 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.

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 mesh faces

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

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

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

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 of existing atlas techniques is typically less than one half. Inefficient packing yields low coverage, such that triangles must be scaled even smaller in order to make the complex chart images fit into available texture space.

4. Atlases for Solid Texturing

This section describes methods for constructing texture atlases specifically for procedural solid texturing that overcome sampling problems and seams.

[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.

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. Since distortion does not affect the quality of the atlas for our application, we choose to deform the model triangles into a form that can be easily packed. The *uniform mesh atlas* arbitrarily maps all of the triangles into a single shape, an isosceles right triangle. These right triangles are packed into horizontal strips and stacked vertically in the texture map.

Figure 3 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.

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

$$a = \frac{\left\lfloor \sqrt{M/2} \right\rfloor}{H} \qquad (3)$$

where H is the horizontal resolution of a square texture map. The floor ensures that we can plot a full row of triangle pairs. Note that a is not an integer, but non-integer edge lengths can create problems with seams.

Seam Elimination. Seams can be avoided by the careful rasterization of mesh triangles. Triangles A and B have been rasterized into the texture map as shown before. The triangles in Figure 4b are rasterized with half pixel offsets such that no background pixe ls will be accessed by the texture's magnification filter. Nonetheless, samples in triangle B near its hypotenuse will still return A's color. Overscanning the hypotenuse of triangle B and shifting triangle A right one pixel, as shown in Figure 4c, eliminates the seam artifact between A and B. This overscanning solution reduces the coverage slightly, but only costs one column of pixels for each triangle pair in a horizontal strip.

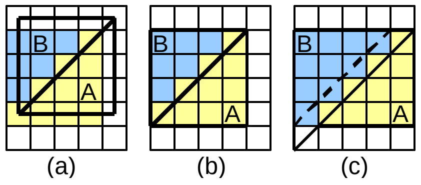

Figure 4. Standard rasterization rules disagree with texture magnification rules (a) and (b). Overscanned polygons are sampled correctly (c).

Since seams are eliminated, triangles can be placed in any order in the uniform mesh atlas. If the 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.

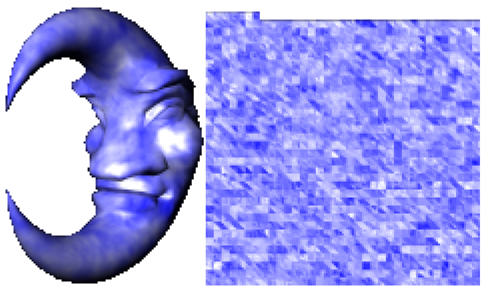

Figure 3. Uniform mesh atlas for a cloud textured moon.

4.2 Non-Uniform 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. The *non-uniform mesh atlas* attempts to more evenly distribute texture samples by varying the size of triangle chart images in the texture map.

Area-Weighted Mesh Atlas. An obvious criterion is that larger model triangles should receive more texture samples, and so their image under the atlas should be larger. We implement this *area-weighted* NUMA by first sorting 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 5 demonstrates the area-weighted atlas on a rhino model.

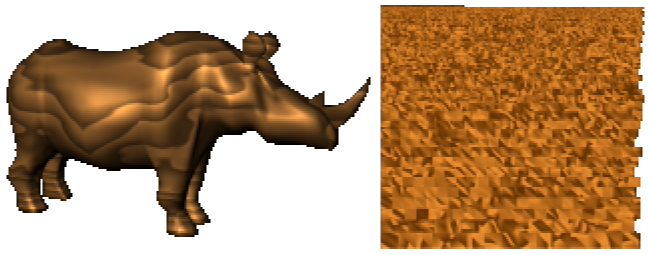

Figure 5: Rhino sculpted from wood and its area-weighted non-uniform mesh atlas.

Length-Weighted Mesh Atlas. Skinny triangles occupy smaller areas, but require extra sampling in their principal axis direction to avoid aliases. The *length-weighted* NUMA uses the triangle's longest edge to prioritize its space utilization in the texture map.

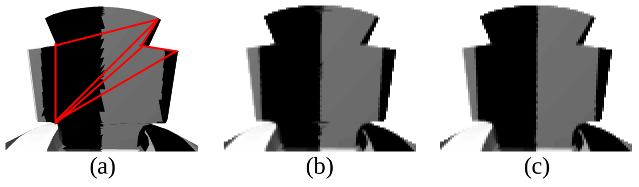

Figure 6. Effects of mesh atlas sample distribution techniques on a poorly tesselated object containing slivers: uniform (a), area weighted (b) and length weighted (c).

Figure 6 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 NUMA reduces these aliasing artifacts, stealing ext ra 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 NUMA 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.

Comparison. We plotted the relative scale of each triangle in the meshed rhino model. The ideal relative scale is equal to the square root of the surface area, and is plotted in green. Since all of the uniform mesh atlas's chart image triangles are the same size, the plot of its relative scale simply indicates the size of the triangle in the model. Hence larger triangles are sample starved, but as Table 1 shows, a larger number of smaller triangles are receiving too many samples.

Mesh Atlas	**Coverage**	**Relative Scale**
Uniform	91%	1.75
Area-Weighted	93%	0.66
Length-Weighted	93%	0.86

Table 1. Measurement of mesh atlas performance on the rhino model.

The area-weighted mesh atlas does a much better job of distributing the samples, and nearly complements the sampling of the uniform mesh atlas. The area-weighted NUMA undersamples smaller triangles because they are assigned to the remaining scraps of the texture map, which also results in its relative scale of less than (but closer to) one.

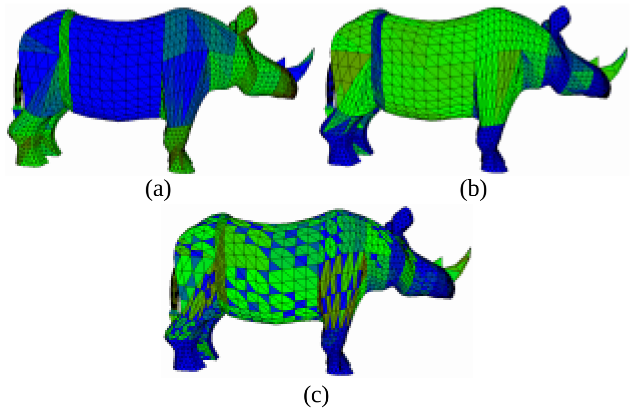

Figure 7. The rhino model color coded by the relative scale of each triangle under the uniform (a), area-weighted (b) and length-weighted (c) atlases. Green indicates optimal sampling, blue indicates too few samples, and red indicates too many.

Figure 7 illustrates the difference with this weighting, increasing the samples in the belt of skinny triangles around the rhino's waist, and the stretched triangles around its shoulder, by sacrificing some of the samples in the rest of the model. The length-weighting heuristic also improves the performance statistics, resulting in a relative scale much closer to the goal of one.

4.3 Multiresolution Mesh Atlases

Section 4.1 described how seam artifacts were removed by making rasterization agree with texture magnification. Texture minification also produces artifacts, aliasing when projected texture resolution exceeds screen resolution.

The MIP-map is a popular method for inhibiting texture minification aliases [46]. The MIP-map creates 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 4^l pixels of the full resolution texture map (at level 0).

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 will be 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.

We achieve this grouping by partitioning the surface mesh hierarchically into a balanced quadtree. Each level of the quadtree partitions the mesh into disjoint contiguous sections with (approximately) the same number of faces.

We implement our face partitioning using a multiconstraint-partitioning algorithm [18]. Such algorithms have found a wide variety of applications in computer graphics, e.g. [9][17][19].

The face hierarchy is constructed using the dual of the mesh. The partitioning algorithm uses edge collapses to repeatedly simplify this dual graph, yielding a hierarchy. The "balanced first choice" [18] heuristic is used to balance the hierarchy during simplification. We then optimize this graph from the top down, exchanging subtrees to minimize the edge length of the boundaries of the partitions. The result is demonstrated in Figure 8.

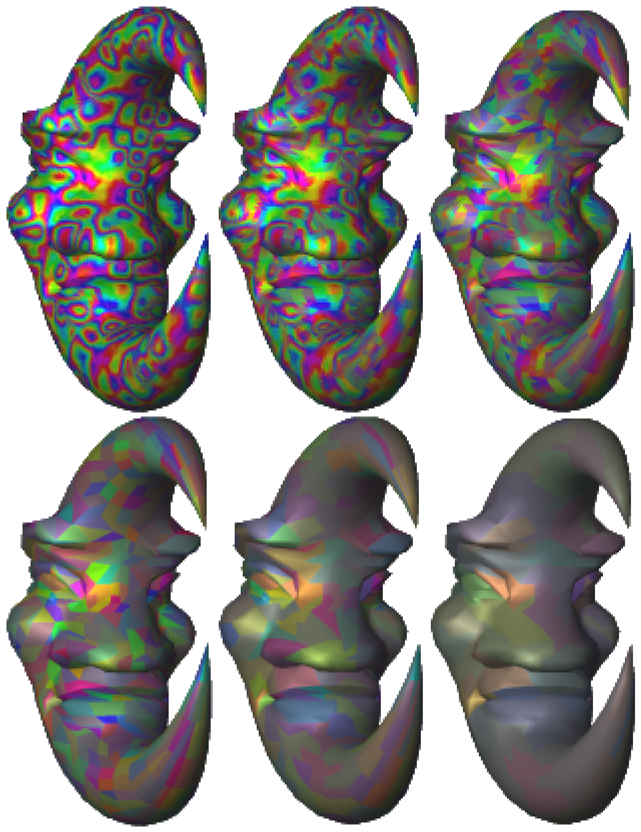

Figure 8. Levels of texture detail in the multiresolution uniform mesh atlas.

5. Procedural Texturing onto the Atlas

The solid texture coordinates resulting from the mesh atlases provides an efficient and direct method for applying procedural textures to an arbitrary object. We apply procedures directly to the texture map using the texture map containing solid texture coordinates interpolated across the polygon faces as input, replacing these coordinates with colors producing a texture map that when applied yields a procedural solid texturing of the object.

Procedural textures can be generated a number of ways. We explore two basic techniques. The first technique runs a procedure sequentially on the host. The second technique compiles the procedure into a multipass program executed in SIMD fashion by the graphics controller. We will focus on the Perlin noise function [37] as this single function is a widely used element of a large portion of procedural textures.

5.1 Host Rasterization

The texture atlas technique allows the procedural texture to be generated from the host. Host procedures provide 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 textures. Goehring *et al.* [10] 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 *et al.* [14] 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 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 is an interactive-time operation, whereas examination of the object is a real-time operation supported completely by the graphics card's texture mapping hardware. Its results are shown later in Table 3.

5.2 A Multipass Noise Algorithm

Following [15][23][35][41], we can harness the power of graphics accelerators to generate procedural textures directly on the graphics board.

The noise function could be implemented using a 3-D texture of random values with a linear magnification filter. A texture atlas of solid texture coordinates can be replaces with noise samples using the OpenGL pixel texture extension [31].

The vertex shader programming model found in Direct3D 8.0 [24] and the recent NVIDIA OpenGL vertex shader extension [31] can support procedural solid texturing. In fact a Perlin noise function has been implemented as a vertex program [29]. But a per-vertex procedural texture will produce vertex colors that are Gouraud interpolated across faces.

```
Input: solid_map with $R,G,B$ containing $s,t,r$ coordinates.
Initialize noise = black
solid_int = solid_map >> $b_f$
solid_intpp = solid_int + $1/(2^b-1)$
weight = (solid_map – (solid_int << $b_f$)) << $b_i$
for ($k = 0$; $k < 8$; $k$++) {
  corner = solid_int
  corner = solid_intpp with glColorMask(k&1,k&2,k&4)
  randomize corner
  corner *= if (k&1) then $R$(weight) else 1 – $R$(weight)[4]
  corner *= if (k&2) then $G$(weight) else 1 – $G$(weight)
  corner *= if (k&4) then $B$(weight) else 1 – $B$(weight)
  noise += corner
}
Output: solid noise texture map
```

Figure 9. Multipass noise algorithm.

We instead implemented a per-pixel noise function using multipass rendering onto the texture atlas. Assume the three channels (R,G,B) of our buffers have a depth of b bits[5]. We will assume a fixed-point representation with b_i integer bits and b_f fractional bits, $b = b_i + b_f$. The algorithm in Figure 9 computes a random value in [0,1] at the integer lattice points, and linearly interpolates these random values across the cells of the lattice.

SGI Implementation. We implemented the noise function in multipass OpenGL on imaging workstations using the glPixelTransfer and glPixelMap functions. The glPixelTransfer function performs a per-component scale and bias, whereas glPixelMap performs a per-component lookup. The results appear in Table 2.

NVidia Implementation. We also implemented a noise function for consumer-level accelerators using the NVidia chipset. Since the NVidia driver did not accelerate glPixelTransfer and glPixelMap, we used register combiners to shift, randomize and isolate/combine components.

Randomization on the NVidia controller was particularly difficult, as its driver did not accelerate logical operations like exclusive-or on the frame buffer. Instead, we used the register combiners to display one of two colors depending on an input color's high bit, then used the register combiners to shift the input color left one bit (without overflowing and causing a clamp to one). This ended up generating 375 passes (!). The source code for these operations can be found on the accompanying CD-ROM.

Implementation	Execution Time
SGI Solid Impact	1.3 Hz
SGI Octane	2.5 Hz
NVidia GeForce 256	0.9 Hz

Table 2. Execution times for the multipass noise algorithm.

Table 2 shows the NVidia implementation did not perform as well as the SGI implementation. Profiling the code revealed that the main bottleneck was the time it took to save the framebuffer in a texture, adding an average of 3 ms per pass for 354 of the passes. OpenGL currently does not support rendering directly to texture, and the register combiner did not directly support the blending of its output with the destination pixel currently in the frame buffer.

[4] The functions R(), G() and B() return a luminance image of the channel.

[5] Framebuffers currently hold only 8 or 12 bits per channel though there is an extension that supports 32-bit floating point, and indications that floating point buffers may soon be supported by a larger variety of graphics hardware and drivers.

The randomization step in the SGI implementation produced white noise using a glPixelMap lookup table of random values, whereas the NVidia implementation blended random colors, yielding Gaussian noise. If desired, one could redistribute the Gaussian noise into white noise with a fixed histogram equalization step.

6. Conclusion

We have shown how the texture atlas can facilitate the real-time application of solid procedural texturing. We showed that for this application, the texture atlas need not be concerned with distortion nor discontinuity, but should instead focus on sampling fidelity. We introduced new mesh-based atlas generation schemes that more efficiently used available texture samples, and non-uniform variations of these meshes distributed these samples more evenly across the object. We also used a mesh partitioning method to construct a MIP-mappable atlas.

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 [1]. 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.

6.1 Interactive Procedural Solid Texture Design

We used the methods described in this paper to create a procedural solid texture design system that would allow the user to load an object and apply a procedural solid texture. This system can be found on the accompanying CD-ROM. Since the procedural solid texturing is applied as a standard 2-D surface texture mapping, the design system supported full real-time observation of a procedurally solid textured object. Using the techniques of Section 4, the object did not suffer from any seam artifacts, and aliasing was reduced by making good use of the available texture samples.

We also allowed the user to interactively change the procedural solid texturing parameters. Using the techniques described in Section 5.1, we were able to support interactive-rate feedback to the user, such that the user could observe the result of a parameter on the procedural solid texture while dragging a slider.

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 render 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 3.

Noise Octaves	Atlas Res.	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)

Table 3. Execution times for procedural texture synthesis into the texture atlas. Parenthetic times measure lower resolution synthesis during interaction.

6.2 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 [13]. The atlas 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.

Normal Maps. The normal map [3][8] 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 [35][41] 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.* [41] 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, ala Section 5.2. 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.

6.3 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.

Higher-Order Texture Magnification. Section 4.1 described the special overscanning measures taken during rasterization of the texture atlas to eliminate seam artifacts. This overscanning works when a nearest neighbor texture magnification filter is used. A linear texture magnification filter would make the textures appear less blocky, but will require overscanning by one pixel along all edges reduces the number of available samples on polygon faces creating additional seldom used samples on polygon edges.

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.

6.4 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 the UMA biases the MIP map in favor of smaller triangles.

References

[1] Apodaca, A.A. Advanced Renderman: Creating CGI for Motion Pictures. Morgan Jaufmannm 1999. See also: Renderman Ticks Everyone Should Know, in SIGGRAPH 98 or SIGGRAPH 99 Advanced Renderman Course Notes.

[1] Bennis, C. J Vezien, and G. Iglesias. Piecewise surface flattening for non-distorted texture mapping. *Proc. SIGGRAPH 91*, July 1991, pp. 237-246.

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

[3] Cohen, J., M. Olano and D. Manocha. Appearance-Preserving Simplification. *Proc. SIGGRAPH 98*, July 1998, pp. 115-122.

[4] Crow, F.C. Summed area tables for texture mapping. Computer Graphics 18(3), (*Proc. SIGGRAPH 84*), July 1984, pp. 137-145.

[5] DoCarmo, M. *Differential Geometry of Curves and Surfaces*. Prentice-Hall, 1976.

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

[7] Foley, J.D., A. van Dam, S.K. Feiner and J.F. Hughes. Computer Graphics, Principles and Practice, Second Edition, Addison-Wesley, 1990.

[8] Fournier, A. Normal distribution functions and multiple surfaces. Graphics Interface '92 Workshop on Local Illumination, May 1992, pp. 45-52.

[9] Garland, M., A. Willmott and P.S. Heckbert. Hierarchical face clustering on polygonal surfaces. Proc. Interactive 3D Graphics, March 2001, To appear.

[10] Goehring, D. and O. Gerlitz. Advanced procedural texturing using MMX technology. Intel MMX Technology Application Note, Oct. 1997. http://developer.intel.com/software/idap/ resources/technical_collateral/mmx/proctex2.htm

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

[12] Hanrahan, P. Procedural shading (keynote). Eurographics / SIGGRAPH Workshop on Graphics Hardware, Aug. 1999. http://graphics.standford.edu/hanrahan/talks/rts1/slides.

[13] Hanrahan, P. and P.E. Haeberli. Direct WYSIWYG Painting and Texturing on 3D Shapes, *Computer Graphics 24* (4), (*Proc. SIGGRAPH 90*), Aug. 1990, pp. 215-223.

[14] Hart, J. C., N. Carr, M. Kameya, S. A. Tibbits, and T.J Colemen. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. *1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Aug. 1999, pp. 45-53.

[15] Heidrich, W. and H.-P. Seidel. Realistic hardware-accelerated shading and lighting. *Proc. SIGGRAPH 99*, Aug. 1999, pp. 171-178.

[16] Kameya, M. and J.C. Hart. Bresenham noise. *SIGGRAPH 2000 Conference Abstracts and Applications*, July 2000.

[17] Karni, Z. and C. Gotsman. Spectral compression of mesh geometry. *Proc. SIGGRAPH 2000*, July 2000, pp. 279-286.

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

[19] Lee, A.W.F., W. Sweldens, P. Schröder, L. Cowsar, D. Dobkin. MAPS: Multiresolution Adaptive Parameterization of Surfaces. *Proc. SIGGRAPH 98*, July 1998, pp. 95-104.

[20] Levy, B. and J.L. Mallet. Non-distorted texture mapping for sheared triangulated meshes. *Proc. SIGGRAPH 98*, July 1998, pp. 343-352.

[21] Ma, S. and H. Lin. Optimal texture mapping. *Proc. Eurographics '88*, Sept. 1988, pp. 421-428.

[22] Maillot, J., H. Yahia and A. Verroust. Interactive texture mapping. *Proc. SIGGRAPH 93*, Aug. 1993, pp. 27-34.

[23] McCool, M.C. and W. Heidrich. Texture Shaders. *1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Aug. 1999, pp. 117-126.

[24] Microsoft Corp. Direct3D 8.0 specification. Available at: http://www.msdn.microsoft.com/directx.

[25] Milenkovic, V.J. Rotational polygon overlap minimization and compaction. *Computational Geometry: Theory and Applications* 10, 1998, pp. 305-318.

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

[27] Munkres, J.R. Topology; A First Course. Prentice Hall, 1974.

[28] Norton, A., A.P. Rockwood, and P.T. Skolmoski. Clamping: A method of antialiasing textured surfaces by bandwidth limiting in object space. *Computer Graphics* 16(3), (*Proc. SIGGRAPH 82*), July 1982, pp. 1-8.

[29] NVidia Corp. Noise, component of the NVEffectsBrowser. Available at: http://www.nvidia.com/developer.

[30] Olano, M. and A. Lastra. A shading language on graphics hardware: The PixelFlow shading system. Proc. SIGGRAPH 98, July 1998, pp. 159-168.

[31] OpenGL Architecture Review Board. OpenGL Extension Registry. Available at: http://oss.sgi.com/projects/ogl-sample/registry/

[32] Peachey, D.R. Solid texturing of complex surfaces. *Computer Graphics* 19(3), July 1985, pp. 279-286.

[33] Pedersen, H.K. Decorating implicit surfaces. *Proc. SIGGRAPH 95*, Aug. 1995, pp. 291-300.

[34] Pedersen, H.K. A framework for interactive texturing operations on curved surfaces. *Proc. SIGGRAPH 96*, Aug. 1996, pp. 295-302.

[35] Peercy, M.S., M. Olano, J. Airey and P.J. Ungar. Interactive multi-pass programmable shading, Proc. SIGGRAPH 2000, July 2000, pp. 425-432.

[36] Perlin, K and E.M. Hoffert. Hypertexture. *Computer Graphics* 23(3), July 1989, pp. 253-262.

[37] Perlin, K. An image synthesizer. *Computer Graphics* 19(3). July 1985, pp. 287-296.

[38] Pixar Animation Studios. Future requirements for graphics hardware. Memo, 12 April 1999.

[39] Potmesil, M., and E.M. Hoffert. The Pixel Machine: A parallel image computer. *Computer Graphics 23*(3), (Proceedings of SIGGRAPH 89), July 1989, pp. 69-78.

[40] Praun, E., A. Finkelstein and H. Hoppe. Lapped Textures, *Proc. SIGGRAPH 2000*, July 2000, pp. 465-470.

[41] Proudfoot, K., W.R. Mark and Pat Hanrahan. A framework for real-time programmable shading with flexible vertex and fragment processing. Manuscript, Jan. 2000. See also: http://graphics.stanford.edu/projects/shading.

[42] Rhoades, J., G. Turk, A. Bell, U. Neumann, and A. Varshney. Real-time procedural textures. 1992 *Symposium on Interactive 3D Graphics* 25(2), March 1992, pp 95-100.

[43] Samek, M. Texture mapping and distortion in digital graphics. *The Visual Computer* 2(5), 1986, pp. 313-320.

[44] Segal, M. and K. Akeley. The OpenGL Graphics System: A Specification, Version 1.2.1. Available at: http://www.opengl.org/.

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

[46] Williams, L. Pyramidal parametrics. *Computer Graphics* 17(3), July 1983, pp. 1-11, Proc. SIGGRAPH 83.

[47] Wyvill G., B. Wyvill, and C. McPheeters. Solid texturing of soft objects. IEEE Computer Graphics and Applications 7(4), Dec. 1987, pp. 20-26.