# The Solid Map: Methods for Generating a 2-D Texture Map for Solid Texturing

Nate Carr
Washington State University
ncarr@eecs.wsu.edu

John Hart
Washington State University
hart@eecs.wsu.edu

Jerome Maillot
Alias | Wavefront
jmaillot@aw.sgi.com

## ABSTRACT

The solid map provides a view-independent method for solid texturing using an ordinary 2-D surface texture map. The solid map transforms a model's polygons into 2-D texture space without overlap. It then rasterizes the polygons in this space, interpolating the solid texture coordinates across the pixels of the polygon. These stored solid texture coordinates are then read by a texture synthesis procedure, which generates a color that is stored at the corresponding location in the texture map. This texture map when applied to the model's polygons yields a procedural solid texturing of the model.

Artifacts of this method include coverage, sampling, distortion and seams. Four algorithms for mapping polygons into texture space without overlap are presented and compared across these artifacts. Applications of the solid map are also presented, including solid texturing deformed objects, real time procedural solid texturing, procedural solid texturing in OpenGL and storing solid textures in model description files.

## 1. INTRODUCTION

Solid texturing [17; 19] is now a well known and often used tool in computer graphics. Solid texturing uses three space coordinates instead of two surface coordinates, removing the need to navigate a surface in order to texture it. It also accurately simulates the sculpting of a shape out of a solid substance, such that the features of the solid appear on the shape's surface. Solid texturing also provides a simple and direct method for aligning texture features across the seams at edges, mesh boundaries and intersection curves.

Solid texturing was necessarily invented in conjunction with procedural texturing, for at the time the concept of storing a 3-D array of texture colors was prohibitively expensive. The use of programs to generate the texture colors given three space coordinates as a parameter overcame the limitations of memory storage at the expense of additional computation time. Compared to stored image textures, procedural textures provide a seemingly infinite amount of non-repeating texture across space, and allow higher resolution textures with more intricate detail.

Conversely, many procedural textures, such as those based on the Perlin noise function, work well with any set of texture coordinates so long as they vary consistently across the surface. Using solid texture coordinates based on the model coordinate system saves the user from the burden of surface parameterization to obtain two-dimensional surface texturing.

Procedural solid texturing is a central component of the Renderman shading language [7]. An entire high-level little language was developed to facilitate the simplified coding of *shaders*, procedures invoked during the rendering of a model. The flexibility offered by these programmable shaders has permeated nearly every application of high quality rendering, and elevated Renderman to an industry standard for procedural shader description.

Because procedural textures trade computational power for storage space, they have been used extensively in parallel graphics systems whose numerous processors have little internal memory storage. [20] used a Pixel Machine to render models generated by marching rays through solid procedural textures. [22] implemented procedural solid textures on the PixelPlanes parallel image computer, and [16] developed this into a fully programmable shading system on PixelFlow. However, even fifteen years after its publication, procedural solid texturing has yet to find a real-time implementation in consumer graphics systems, though there have been several recent calls for it [21; 6].

Current graphics libraries support solid texturing with the management of three (or more) texture coordinates, and provide the storage of 3-D texture volumes. While modern computers have enough memory to store such textures, storage of a 3-D array of texture colors remains a highly inefficient use of resources since only 2-D slices of the texture will appear on the surface polygons. Hence, the main roadblock to procedural shading hardware is specification, namely extending existing standards to include procedures, and determining how to specify a procedure efficiently to a graphics API.

[8] based a hardware design around a single function capable of generating several of the most popular procedural solid textures, such that an extended version of OpenGL could specify a procedural texture as a set of parameters to this function. [18] described a compiler that translates Renderman shading procedures into OpenGL source code. The technique is based on costly multipass rendering, and required the extension of the precision of OpenGL's framebuffers. [12] proposed extending OpenGL with programmable blending operations to allow it to generate the high quality multipass images using a single blended multitexturing rendering pass.

Rather than depending on an API extension for solid texturing, we instead rely on a new technique for implementing real-time procedural solid texturing using the surface texture mapping capabilities existing in current graphics API's. The technique uses a data structure we will call the *solid map*. Whereas the texture map contains colors indexed by two dimensional surface texture coordinates, the solid map contains solid texture coordinates indexed by these same two-dimensional surface texture coordinates. Unlike the texture map, the solid map uses the 2-d texture coordinates to rasterize each of the object's polygons into a unique area of the solid map. This rasterization interpolates the solid texture coordinates across the polygon in the solid map. The solid texture coordinates in the solid map may then be used as the input to a procedural texture that

generates a corresponding texture map. This texture map, when applied back onto the model's polygons using standard texture mapping, yields a procedural solid texture as shown in Figure 1.

Figure 1: A wood-sculpted teapot (front) rendered using a 2-D surface texture map (back) generated from a solid map.

In addition to real-time procedural solid texturing, the solid map also supports the deformations of sculpted objects. When a real object deforms, even an object sculpted from a solid, the features of the texture on the surface deforms with the surface. The texture certainly does not remain static with respect to its own coordinate system as if the deformed object were resculpted from the same material. [26] attempted a solution when they investigated methods for deforming solid textured implicit surfaces using an auxilliary coordinate system that deformed with the object. The solid map allows the features of the solid texture to adhere to the surface, instead of having the surface "swim" through the material it was sculpted from, as shown in Figure 2.

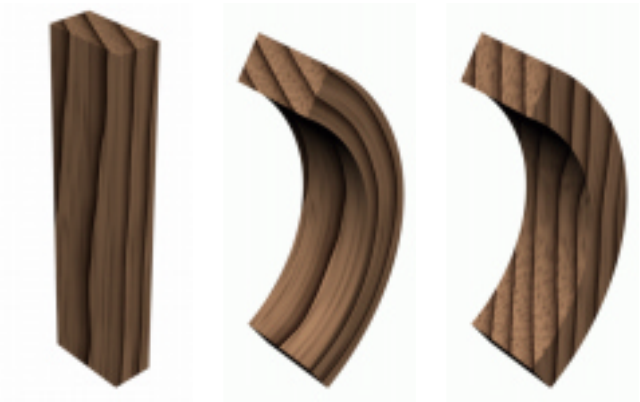

Figure 2: A piece of wood (left) warped by having the solid texture adhere to the surface (center) versus a warped piece swimming through the solid texture (right).

The solid map is already finding use in mainstream computer graphics. [1] described briefly an implementation in Renderman of a technique similar to the solid map to store solid textures (and even shading information) using a reference mesh on a view-perpendicular plane filling the screen to generate a texture map that could be applied to a deformed version of the object or even a different object altogether. A version of the solid map was also implemented as a tool in Alias|Wavefront's PowerAnimator [3] and Maya [24] to generate texture maps of procedural solid textures based on the

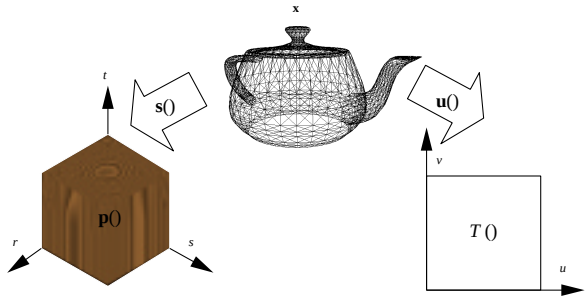


Figure 3: Model (center), solid texture (left) and surface texture (right) coordinate spaces.

model's existing surface texture map coordinates. Prompted by its recent popularity, this paper documents and describes the solid map method, and proposes and compares several algorithms to automatically generate surface texture coordinates for use with solid mapping.

Section 2 of this paper describes the solid map in detail, using a notation to more rigorously trace the cause of the aliasing artifacts it can generate. Section 3 describes four attributes of the solid mapping method and the kind of display artifacts they can cause. Section 4 presents four different algorithms for laying out an object's polygons into texture memory and compares them based on the artifacts they generate. Section 5 concludes with recommendations for selecting the appropriate solid mapping technique, describes the implementation of solid mapping in OpenGL, and proposes ideas for future work on antialiasing.

## 2. THE SOLID MAP

We will use two simultaneous texture coordinates. The two-dimensional *surface* texture coordinates $\mathbf{u} = (u, v)$ and the three-dimensional *solid* texture coordinates $\mathbf{s} = (s, t, r)$. Let $\mathbf{x}$ be the coordinates of a point on a given triangle in *model* coordinates. We will denote the 2-D surface texture coordinates of this point as $\mathbf{u}(\mathbf{x})$ and the 3-D solid texture coordinates of this point as $\mathbf{s}(\mathbf{x})$. We will also denote the color $\mathbf{c} = (R, G, B)$ and the color of a point $\mathbf{x}$ on the surface as $\mathbf{c}(\mathbf{x})$. (Following a convention commonly used in parametric curves and surfaces research papers, we are using the same letter for a coordinate and a map that yields that coordinate. We will attempt to avoid ambiguous cases by appending parentheses $()$ to the letter when it denotes a map.)

Let $\mathbf{p} : \mathbf{s} \mapsto \mathbf{c}$ synthesize a procedural solid texture, mapping solid texture coordinates $\mathbf{s}$ to a color $\mathbf{c}$. Let $T : \mathbf{u} \mapsto \mathbf{c}$ be a 2-D texture map that returns a color $\mathbf{c}$ given surface texture coordinates $\mathbf{u}$. Figure 3 pictorially differentiates the two different texture coordinates and how they are evaluated.

We use capital letters to denote maps that are implemented with a lookup table, such as the texture map $T$. We will use the $\leftarrow$ operator to denote assignment to this table. For example, the framebuffer $C : (x_s, y_s) \rightarrow \mathbf{c}$ is a mapping from screen coordinates $(x_s, y_s)$ to a color $\mathbf{c}$. The frame buffer is implemented as a table, and assignment of an element $\mathbf{c}$ into this table at index $x_s, y_s$ is denoted as $C(x_s, y_s) \leftarrow \mathbf{c}$.

Since the driving application for this research is real-time procedural solid texturing, we assume a forward "polygon-to-pixels" graphics pipeline. The bulk of the graphics pipeline can be considered a general projection, which we will denote $\pi$, that maps vertices from 3-D model coordinates to 2-D screen coordinates. Rasterization takes these screen coordinate vertices and fills in the polygon

they describe using linear interpolation. Rasterization also linearly interpolates other attributes in addition to screen coordinates. It will be useful for the analysis of the aliasing artifacts to know exactly when attributes are interpolated across a polygon, as this signals when continuous functions are discretely sampled. We will indicate that such an interpolation has occured by rasterization with the operator *lerp*(). Hence, $\mathbf{x}$ is a point on the polygon in model-coordinates, $\pi(\mathbf{x})$ is the screen coordinate corresponding to that point and $\mathrm{lerp}(\pi(\mathbf{x}))$ reminds us that the coordinates of that pixel were interpolated from the screen coordinates of the polygon's vertices. Our goal is to determine an expression for the color of these pixels and store it in the frame buffer $C(\mathrm{lerp}(\pi(\mathbf{x})))$.

Standard texture mapping is expressed in this notation as

$$C(\mathrm{lerp}(\pi(\mathbf{x}))) \leftarrow T(\mathrm{lerp}(\mathbf{u}(\mathbf{x}))). \tag{1}$$

In other words, the color of each pixel in the polygon's projection $\pi(\mathbf{x})$ is given by the pixel in the texture map $T$ at the coordinate $\mathbf{u}$ corresponding to the surface point $\mathbf{x}$.

The notation now gives us an indication of potential aliasing. The aliasing artifacts introduced by standard texture mapping occur when the sampling rate of the lerp on the LHS of (1) (the resolution of the polygon's screen projection) disagrees with the sampling rate of the lerp on the RHS (the texture's resolution). Methods for resampling the texture map based on the MIP map [25] or the summed-area table [5] fix this problem by adjusting the lerp sampling density on the RHS of (1).

Standard procedural solid texturing is implemented as

$$C(\mathrm{lerp}(\pi(\mathbf{x}))) \leftarrow \mathbf{p}(\mathrm{lerp}(\mathbf{s}(\mathbf{x}))). \tag{2}$$

In this case, the solid texture coordinates stored at each vertex are interpolated across the pixels of the polygon's projection, and a procedure is called for each pixel to synthesize the color of that pixel. This was the basic texturing technique for a variety of procedural texturing renderers [7; 8].

Aliasing from procedural solid texturing occurs when the sampling rate of the lerp in the LHS of (2) (again the resolution of the polygon's projection) disagrees with the sampling rate of the RHS (the resolution of the polygon sampling the solid texture). Existing methods for antialiasing procedural solid textures include bandlimiting [15] and a gradient magnitude technique [22] which modify the texture procedure $\mathbf{p}$ to only generate signals properly sampled by the coordinates discretized by the lerp.

Deferred shading [14] implements procedural solid texturing in two phases. In the first phase

$$\hat{S}(\mathrm{lerp}(\pi(\mathbf{x}))) \leftarrow \mathrm{lerp}(\mathbf{s}(\mathbf{x})) \tag{3}$$

such that the solid texture coordinates are stored in a map $\hat{S}$ which is the same resolution as the display. Once all of the polygons have been scan converted, the second phase makes a single shading pass through every pixel in the frame buffer

$$C(x_s, y_s) = \mathbf{p}(\hat{S}(x_s, y_s)) \tag{4}$$

replacing the color with the results of the procedure applied to the stored solid texture coordinates.

This representation reveals a shortcoming of deferred shading. Antialiasing is diffficult for deferred shading systems since the procedural texture is generated in a separate step of the algorithm than the step where the samples have been recorded from the lerp. Unless a significant amount of auxilliary information is also recorded, previous procedural texturing antialiasing algorithms do not apply to deferred shading.

Like deferred shading, the solid map implements procedural solid texturing in multiple phases. In the first phase, the solid map is similarly

$$S(\mathrm{lerp}(\mathbf{u}(\mathbf{x}))) \leftarrow \mathrm{lerp}(\mathbf{s}(\mathbf{x})). \tag{5}$$

Each polygon is rasterized into the solid map $S$ using its surface texture coordinates $\mathbf{u}(\mathbf{x})$. The data it places in the solid map (the data that gets interpolated across the face of the rasterized polygon) is the solid texture coordinates $\mathbf{s}(\mathbf{x})$. Note that the solid map $S : \mathbf{u} \mapsto \mathbf{s}$ is the same resolution as the texture map whereas the deferred shading map $\hat{S} : (x_s, y_s) \mapsto \mathbf{s}$ was the same resolution as the display.

The solid texture coordinates are converted into texture colors in the second phase by the assignment

$$T(\mathbf{u}) \leftarrow \mathbf{p}(S(\mathbf{u})). \tag{6}$$

The color of each pixel in the texture map $T$ at surface texture coordinates $\mathbf{u}$ is synthesized by the procedural texture $\mathbf{p}$ on the solid texture coordinates in the solid map $S$ located at the same texure coordinates $\mathbf{u}$.

The texture map $T$ now contains a surface texture that when mapped by the third phase onto the polygons using (1) corresponds to the procedural solid texture generated by (2).

Because the solid map equations (5) and (6) resemble the deffered shading equations (3) and (4), this solid map could be considered deferred shading in the texture map instead of the display. However, the benefit of deferred shading is that it reduces the shading depth complexity to one; only the visible parts of polygons are shaded. The solid map contains all of the model's polygons without overlap, so every polygon is is "visible" in the solid map and needs to be textured, regardless of whether it is visible in the display.

Unlike deferred shading, the solid map is view independent. The triangles are rasterized and the procedural texture is rendered onto them in the texture buffer only once. The surface texture mapping of the solid map can occur any number of times from arbitrary viewpoints.

The real benefit of the solid map instead comes from the fact that the procedure is executed as a second pass, independent of the display rasterization of the model. This allows a graphics process to rasterize polygons and a host processor to synthesize the texture for them. By separating rasterization from texture synethisis, this procedural solid texturing technique can be implemented in modern pipelined graphics API's, such as OpenGL. We also expect the solid map will support antialiasing of the solid texture better than deferred shading could.

The aliasing artifacts introduced by this method occur when the sampling rate of the lerp in the LHS of (5) (the surface texture coordinates) disagrees with the sampling rate of the RHS (the solid texture coordinates). We find that this aliasing is entirely governed by the map $\mathbf{u}()$, called the u-map, in the LHS of (5).

## 3. THE U-MAP

Surface texture mapping uses texture coordinates assigned to polygon vertices to define the *u-map* $\mathbf{u} : \mathbf{x} \mapsto \mathbf{u}$. If the u-map is one-to-one, then its inverse $\mathbf{x} : \mathbf{u} \mapsto \mathbf{x}$ parameterizes the surface. The u-map takes points from the surface into a texture map, which then assigns a color to the point based on its location in the texture map.

Typical u-maps are constructed for texture mapping by discretely assigning texture coordinates to vertices on the model, and making this map continuous by interpolating the texture coordinates across the faces of the polygons. This interpolation provides an association of texture coordinates $\mathbf{u}$ with model coordinates $\mathbf{x}$.

Texture mapping does not require the u-map to be 1-1. However, this is required for the u-map used for solid mapping.

In addition to being 1-1, there are several other features of the u-map that can reduce the aliasing artifacts sometimes produced by the solid mapping technique. These features of the u-map on a model's polygons are the efficiency of the covering of texture space, scaling, distortion and the number of new boundary edges it introduces, which we call "seams."

## 3.1 Coverage

The solid map method depends on the resolution of the solid map $S$ and the texture map $T$ used to sample the solid texture. The u-map is 1-1, placing all of the model's polygons into a rectangle of this resolution without overlap. The *coverage* $c$ of the u-map is how effectively the u-map covers the solid map/texure map, utilizing as many pixels in the solid map/texture map as possible, and therefore sampling the solid texture as much as possible.

We measure the u-map coverage as the percentage of the solid map/texture map that are covered by the images of the $M$ polygons under the u-map

$$c = \sum_{i=1}^{M} A(\mathbf{u}_{i_1}, \mathbf{u}_{i_2}, \mathbf{u}_{i_3}) \tag{7}$$

where $A()$ returns the area of a triangle. We assume the solid map/texture map is a unit square in surface texture coordinates.

## 3.2 Relative Scale

Whereas the coverage measures how well the entire u-map utilizes texture samples, the relative scale indicates how well the u-map utilizes samples locally, per polygon. The solid map can vary widely from polygon to polygon so the relative scale need not be necessarily correlated with the coverage of the u-map.

We measure the relative scale as the relative change in size between the model coordinate polygon and its image under the u-map in surface texture coordinates. There are several methods for measuring the change in size due to a map. The Lispchitz constant of a map finds the closest the map brings any two points relative to their original distance apart. However, measuring the most severe compression of a u-map does not seem to be a fair indication of the average number of samples it supports within a given triangle, especially when the u-map compresses the triangle more in one direction than another.

We choose instead to measure the relative scale for a single triangle as the square root of the ratio of the triangle's area before and after the u-map is applied, as

$$\text{scale}(\mathbf{x}_{i_1}, \mathbf{x}_{i_2}, \mathbf{x}_{i_3}) = \sqrt{\frac{A(\mathbf{u}_{i_1}, \mathbf{u}_{i_2}, \mathbf{u}_{i_3})}{A(\mathbf{x}_{i_1}, \mathbf{x}_{i_2}, \mathbf{x}_{i_3})}}. \tag{8}$$

We average the relative scale for all the triangles as

$$\text{scale} = \frac{1}{M} \sum_{\mathbf{t} \in \mathcal{T}} \text{scale}(\mathbf{t}) \tag{9}$$

where $\mathbf{t}$ denotes a triangle and $\mathcal{T}$ denotes an $M$ element triangulation of the model.

Since the scale of model coordinates is not related to the scale of texture coordinates, and polygons are not rasterized in model coordinates, the relative scale is not an absolute measure of the u-map, and only useful for the comparison of different u-maps from the same model into the same texture space.

## 3.3 Distortion

The standard texture mapping practice of assigning two-dimensional coordinates to vertices of a three-dimensional model can easily introduce distortions in the induced u-map. A distorted surface texturing will not directly distort the solid texturing generated by the solid map. The point $\mathbf{x}$ on the surface will still have the same solid texture coordinates generated by $\mathbf{s}()$ regardless of the u-map. However, differences between neighboring polygons in the resolution of the solid texture, and its "grain" due to the axes of the texture map, will still be visible. Hence, the u-map used for implementing the solid map should likewise avoid distortion.

Research in non-distorted texture mapping fixes this problem by assigning texture coordinates such that the resulting u-map is as close to similarity as possible, consisting, at least locally, of little else than rotations, translations and uniform scales. [10], [2], textcitemaillot93 and more recently [9] devised global optimization methods that assigned texture coordinates that minimized a distortion metric whereas others such as [23] instead reduced distortion by flattening the polygons onto a cube surrounding the object.

We assume the u-map is locally affine in that it affinely maps model coordinate triangles into surface texture coordinate triangles, but that this affine transformation can be different for each triangle. For the sake of simplicity, assume also that we have already rotated and translated the model coordinate triangle from model space into the plane of the surface texture space triangle. Then let $A$ be the $2 \times 2$ transformation matrix that represents the linear part of the rest of the u-map. Note that the linear transformation $A$ contains any distortion components of the u-map for the current triangle.

The first fundamental form [4] of a linear transformation $A$ is given by $\mathcal{I} = AA^T - Id$ where $Id$ is the $2 \times 2$ identity matrix. The first fundamental form has been used as a distortion metric for optimization in non-distorted texture mapping [11]. We will instead use it to measure distortion in the u-map.

Since the relative scale has already been isolated, we need to factor uniform scales out of the distortion measure. We first label the elements of $AA^T$ as

$$AA^T = \begin{bmatrix} a & b \\ c & d \end{bmatrix}. \tag{10}$$

and assume without loss of generality that $a < d$ and recall that $b = c$. We then define

$$\text{stretch} = \sqrt{\frac{d}{a}} - 1 \tag{11}$$

as the percentage the u-map stretches one coordinate with respect to the other, and

$$\text{shear} = \sqrt{|b|} \tag{12}$$

as the percentage that the u-map shears one coordinate into the other.

## 3.4 Seams

The u-map used for the solid map is in general different than the u-map for standard texture mapping. The u-map for standard texture mapping is usually continuous, such that a connected texture maps to a connected mesh of polygons. The u-map for solid mapping on the other hand does not necessarily need to be continuous. Allowing the u-map to be discontinuous changes the optimization problem from a distortion metric minimization problem into a packing problem. Packing can even permit a distortion free u-map. However, such packing-style u-maps introduce a new artifact called "seams."

*Seams* are pixels in the texture buffer covered by the image of more than one triangle. If the texture for these two triangles is different, then the texture of the pixels on the edge of one triangle can overflow into the other during texture mapping.

The rules of polygon scan conversion are designed with the goal of plotting each pixel in a local polygonal mesh neighborhood only once. 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.

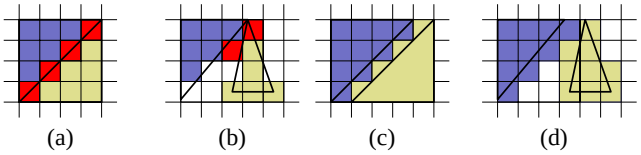


Figure 4: Seam artifacts in red (a) and (b) corrected by overscanning (c) and (d).

The rules of rasterization do not serve the solid map well, because the polygons rasterized in the solid map do not necessarily share the same mesh topology as they do in model coordinates. Neighboring polygons in the solid map may not be neighboring polygons in the model, and pixels along their boundary will have to decide which polygon they belong to (e.g. the red pixels in Figure 4 (a) and (b)). When the solid map texture is mapped onto the polygon, some of these boundary pixels may be colored from a completely different location in solid texture space.

Furthermore, some polygons rasterized in the solid map might not share an edge with any other polygon. Pixels on this edge of this polygon might not be covered by the rasterization, and instead of the procedural texture, the texture map's initial (background) color may be mapped onto the corresponding edge when the model is displayed (e.g. the white pixels in Figure 4).

Seams can be eliminated by overscanning each polygon, ensuring that every pixel that covers even a portion of the polygon contains a color specific to that polygon. This can be understandably wasteful and decreases the coverage of the u-map. In Figure 4 (c) and (d), we surrounded the second triangle by a one pixel buffer zone, and rasterized it such that every pixel covered by any portion of the triangle will still have that triangle's attributes.

Seams can also be reduced if the u-map at least partially preserves the mesh topology. This can be accomplish by cutting the mesh at a small number of polygon edges, and spreading out sections of polygons in patches. Since the patches are usually not flat, this increases the distortion of the u-map.

We measure seams as the length of boundary edges (edges not shared by two polygons) in surface texture space

$$\text{seam} = \sum \{ ||\mathbf{u}_i - \mathbf{u}_j||, < i,j > \in \mathcal{T}, < j,i > \notin \mathcal{T} \} \qquad (13)$$

where $< i,j >$ indicates the edge from vertex $i$ to vertex $j$, and assuming triangle vertices in $\mathcal{T}$ are consistently ordered clockwise or counterclockwise.

# 4. SOLID MAPPING ALGORITHMS

Given the possible artifacts of coverage, relative scale, distortion and seams, several u-map algorithms can be devised to spread out the model's surface onto the solid map $S$. These algorithms assume the model is constructed from triangles.

## 4.1 Simple Mesh

The *simple mesh* u-map rasterizes the model's triangles into texture memory as an axis-aligned mesh of uniformly-shaped right triangles. The simple mesh packing is illustrated in Figure 5. Ideally the horizontal triangle strips of the packing will match the topology of triangle strips in the model.

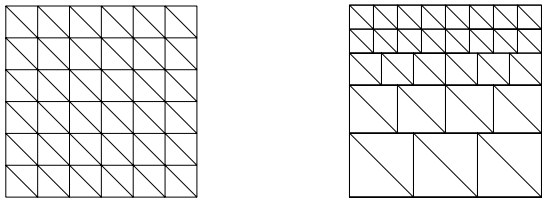

Figure 5: Simple (left) and area-approximating mesh packing of triangles.

This technique uses the integer $a$ to set the adjacent length of the right triangles, and should be set to

$$a = \lfloor \sqrt{2HV/n} \rfloor \qquad (14)$$

to maximize the size of triangles in order to efficiently use of the texture buffer space available.

A shortcoming of the simple mesh is that it distributes samples unevenly, such that large model polygons will get the same number of texture samples as small model polygons.

## 4.2 Area Approximating

The *area approximating* u-map is similar to the mesh u-map, but with a goal to distribute the texture samples more equitably. The model's triangles are first sorted by non-increasing area. Then the mesh is constructed in horizontal strips, but the scale factor $a$ used for each strip is set such that the area of the first triangle in each strip more closely corresponds to the area of its model-coordinate counterpart. This area approximating sampling is illustrated in Figure 5.

The problem with packing arbitrarily-shaped triangles into right-triangle meshes is that it distorts the shape of the polygon.

## 4.3 Polyhedral Projection

The *polyhedral projection* u-map is designed to reduce distortion in the polygon shape while maintaining a reasonable amount of the original mesh topology. The technique projects the model polygons onto a plane. To reduce the distortion of projection, the projection can occur on any one of a set of planes of different orientations.

The 3D object is first segmented into large areas based on the normal at each triangle. Every triangle is projected onto the plane closest to parallel with it. The sphere in Figure 6 shows this first step. If a single piece self overlaps, additional cuts are added to split the area into smaller pieces. This is the case in the spiral object showed in figure 7.

Every connected piece is then moved in the texture plane to avoid overlaps. Currently a simple rectangle packing algorithm, using the texture space bounding boxes, is used.

This method allows a simple control of the overall distortion due to the mapping function. Every triangle is mapped through a planar projection.

The first fundamental form [4] written in the appropriate basis only depends on the angle $\alpha$ between the triangle and the projection plane

$$\mathcal{I} = \begin{pmatrix} 1 & 0 \\ 0 & s^2 \end{pmatrix}, \qquad (15)$$

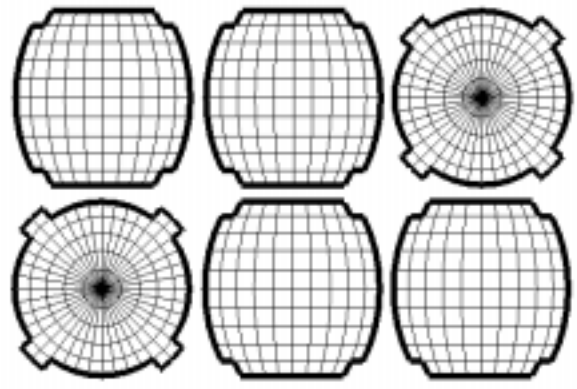

Figure 6: Projection of mesh polygons onto similarly oriented planes.

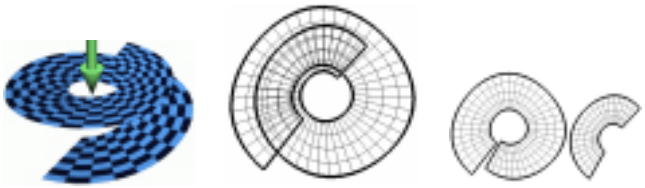

Figure 7: Cutting a self-overlapping object.

where $s = cos(\alpha)$. The deformation value is then $||\mathcal{I} - I|| = 1 - s^2 = sin(\alpha)^2$.
Even though any projection set could be used, we found that in practice projecting onto the faces of a simple polyhedron gives a good result. The Platonic solids (tetrahedron, cube, octahedron and dodecahedron, icosahedron) provide uniformly sampled orientations. We found that there is a large performance gap between the tetrahedron and the cube, and so investigated five-sided non-regular polyhedra as well. We found the best orientation covering with 5 planes is achieved using an equilateral triangle based cylinder, which we will call a *prism*, as shown in Figure 8.

Figure 8: The five sided prism used as an intermediate between the tetrahedron and the cube.

The deformation values in Figure 9 were computed using a sphere tessellated into very small triangles, which is a good sample model representative for most 3D objects because its polygons uniformly span a discrete subset of all orientations.
Figure 9 shows that there is no need to have more than 12 projection planes. The 20 plane projection based on the icosahedron introduces a lot more seams without a real noticeable improvement of the distortion.

## 4.4 Scalene Triangle Packing

In order to avoid any distortion, a mapping technique that maintains the shape of the triangles is needed. This is equivalent to triangle packing.

| Polyhedron | Min. $s$ | Ave. $s$ | Worst | Ave. |
|---|---|---|---|---|
| Tetra. | 0.333 | 0.665 | 89% | 56% |
| Prism | 0.449 | 0.731 | 80% | 47% |
| Cube | 0.577 | 0.793 | 67% | 37% |
| Octa. | 0.577 | 0.893 | 67% | 20% |
| Dodeca. | 0.795 | 0.914 | 37% | 16% |
| Icosa. | 0.801 | 0.945 | 36% | 10% |

Figure 9: Polyhedral projection distortion for a tesselated sphere model.

While there have been some studies of equilateral triangle packing, very little work has been performed on the packing of general *scalene* triangles. The packing of triangles (and many other shapes) is an NP-complete problem, and judging by the small amount of literature, approximate solutions do not appear to be very interesting theoretically. The driving application for packing of general shapes is the textile industry, which seeks to optimize the use of fabric for manufacturing clothing. For example, [13] uses a global optimization technique to find a locally optimal oriented packing of complex and possibly concave polygons.
Hence, we propose the following *scalene triangle packing* algorithm as a u-map that maintains triangle shape. The algorithm is a sub-optimal first-fit strip pack, similar to the area preserving packing in Section 4.2. Such algorithms are relatively fast, requiring an $O(n \log n)$ sort of the items by decreasing size, but otherwise operate in seveal linear passes through the data. This complexity makes the proposed algorithm well suited for the large polygon datasets used in computer graphics models. Figure 10 overviews the steps of the algorithm. Except for the sort in step 3, each step performs a linear pass through the triangle data.

1. Rotate triangles into a single shared plane.
2. Orient longest side of triangles onto x-axis.
3. Sort triangles by non-increasing altitude.
4. Flip every other triangle vertically about midpoint.
5. Pack triangles horizontally along x-axis.
6. Group triangles into equal length sections.
7. Invert every other group of triangles.
8. Stack triangle groups vertically.

Figure 10: Scalene triangle packing algorithm.

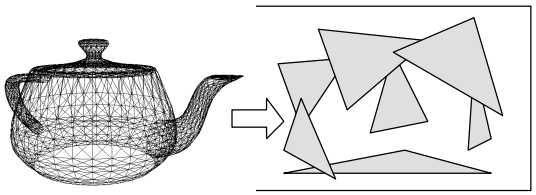

Figure 11: Step 1. Triangles are rotated into the same plane.

Figure 11 shows the first step of the algorithm, which applies individual rigid body transformations to the triangles to make them all coplanar. This transformation first rotates the polygon's normal to be perpendicular to the reference plane, and translates along this rotated normal to place the polygon in this plane.
The second step rotates each triangle in the reference plane such that its longest edge is horizontal, and the third vertex is above the the x-axis. In this position, it is convenient to label the vertices

*left*, *right* and top. It is also convenient to translate the triangle such that the left vertex is at the origin. Note that any triangle in this canonical position can be described by three real values: the $x$ coordinate of the right vertex and the $x$ and $y$ coordinates of the top vertex. Note also that the $x$ coordinate of the top vertex is necessarily between the $x$ coordinates of the left and right vertices, for if not, then the base of the triangle would not be the longest edge.

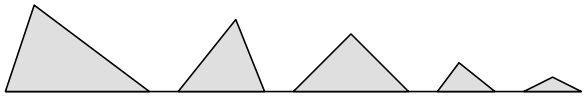

Figure 12: Step 2 rotates and translates triangles in the plane to place their longest edge along the x-axis, and Step 3 sorts these triangles by decreasing altitude.

The third step sorts the polygons by non-increasing altitude. Note that the altitude is given by the $y$ coordinate of the top vertex. The triangles are now as shown in Figure 12,
The fourth step flips every other triangle vertically, so the fifth step can pack these triangles horizontally. The alternating orientiations of the triangles form a toothed pattern of decreasing altitude, as shown in Figure 13. Note that the triangles lined up this way suggest the shape of a bounding quadrilateral, a truncated right triangle.

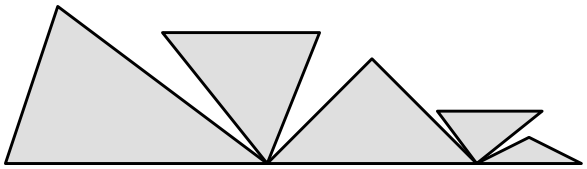

Figure 13: Step 4 alternates triangles into a toothed pattern so that Step 5 can translate them left along the x-axis into a tightly packed configuration.

This truncated triangle shape can be packed into a square of the same area as shown in Figure 14, by slicing the quadrilateral into segments equal to the width of the square, rotating every other segment $180^\circ$ and stacking the segments vertically. Hence, the fifth step of the algorithm transforms the triangles bound by each segment of this quadrilateral into the square. Note that triangles will inevitably intersect the slicing lines of the quadrilateral, and this discretization is such that the packing of triangles is not as efficient as packing the quadrilateral segments.

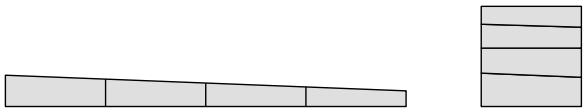

Figure 14: Steps 5, 6 and 7. Triangles are bounded by a truncated triangle, which is sliced and packed into a square.

The trickiest part of the implementation is predicting the size of slices of the quadrilateral. The area of the triangles is well less than the area of the square they are packed into. We have found empirically that increasing the square root of the total area of the triangles by 20% yields a packing whose width approximates the eventual height of the stack of toothed strips.
The scalene triangle packing is shown in Figure 15, as resulting from the "head" dataset described in the next section. The triangles toward the top of the packing are very small, or at least have very low altitudes. In any case, they will likely not rasterize. The packing algorithm preserves the relative size and shape of the polygons so these are the smallest polygons and may be inconsequential. However, if they are grouped together, say at an area of high curvature or roughness in the surface, they collectively can form a sizable section of the surface.
Furthermore, the solid map is view independent, so the shapes and sizes of the polygons match the model coordinates and not the screen coordinate. A smaller polygon in model coordinates could appear larger than the rest of the polygons depending on perspective and the viewpoint.
We depend on the alteration of rasterization rules described in Section 3.4 to ensure that large collections of small polygons get rasterized appropriately in the packed solid map.

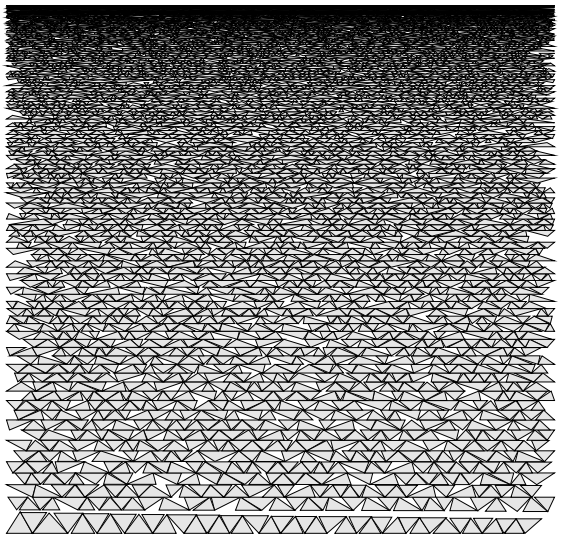

Figure 15: 7,232 triangles packed into a square solid map.

## 4.5 Results and Discussion

We performed experiments and gathered statistics on the results for two datasets: a cartoon head and a kangaroo. The head dataset contains 3,633 vertices and 7,232 faces. The kangaroo dataset contains 23,556 vertices and 45,784 faces. The Performance of the solid mapping algorithms are compared in Figure 16.

| Method | Cover | Scale | Stretch | Shear | Comps. | Seam (# edges) |
|---|---|---|---|---|---|---|
| Head Dataset | | | | | | |
| Simple | 100% | 183% | 51% | 3% | 7,232 | 396.82 (21,696) |
| Area | 100% | 121% | 51% | 2% | 7,232 | 333.46 (21,696) |
| Tetra. | 33% | 69% | 24% | 33% | 50 | 14.47 (1,644) |
| Prism | 26% | 61% | 19% | 30% | 59 | 15.06 (1,852) |
| Cube | 29% | 65% | 12% | 28% | 79 | 17.45 (1,954) |
| Oct. | 24% | 60% | 11% | 26% | 121 | 19.12 (2,336) |
| Dodeca. | 24% | 60% | 6% | 21% | 154 | 23.86 (2,828) |
| Scalene | 72% | 100% | 0% | 0% | 7,232 | 372.05 (21,696) |
| Kangaroo Dataset | | | | | | |
| Simple | 100% | 289% | 164% | 41% | 45,784 | 956.17 (137,352) |
| Area | 100% | 201% | 165% | 28% | 45,784 | 779.51 (137,352) |
| Tetra. | 18% | 54% | 25% | 1% | 285 | 28.98 (13,610) |
| Prism | 20% | 56% | 16% | 0% | 307 | 32.57 (14,624) |
| Cube | 24% | 61% | 11% | 0% | 369 | 38.97 (15,956) |
| Octa. | 27% | 68% | 11% | 1% | 555 | 55.14 (20,576) |
| Dodeca. | 18% | 54% | 6% | 0% | 628 | 54.60 (22,636) |
| Scalene | 65% | 100% | 0% | 0% | 45,784 | 983.26 (137,352) |

Figure 16: Solid mapping algorithm performance statistics.

The *cover* was measured according to the percentage of the texture space the output of the u-map covered. The scale was measured relative to the distortionless scale of the scalene packing result. Re-

call that larger scales correspond to better sampling. The stretch and shear factors are distortion metrics that affect only the "grain" of the aliasing of the solid texture, and are both ideally 0%. The components is the number of connected mesh components placed in texture space. The seam column measures the length of seams in **u** coordinate lengths, and also counts the number of edges in the seam as well.

The simple mesh provides the most efficient use of texture memory, ensuring that every pixel in the texture map is used. According to the relative scaling, the simple mesh also provides more samples on average per triangle. The subtle mismatched sampling due to high distortion rates can be detected in Figure 17 (left).

The area approximating mesh should do a better job of distributing more samples to larger polygons, but according to the disappointing relative scale and distortion statistics this does not appear to be happening. This is likely do to the fact that all triangles in each row are still equally sized. Nonetheless, the larger triangles in Figure 17 (center) are better sampled than for the simple-mesh case.

Figure 18: Head database rendered with wood solid texture (top) with (left) and without (right) overscanning, using the cube projection u-map (bottom).

The mesh preserving connectivity makes the polyhedral projection method the best choice for models with many small polygons, as demonstrated in Figure 18. As the number of polygons increases, their u-map images into the solid map and texture map become smaller, because the size (and resolution) of the domains of these maps remains constant (usually the unit square) regardless of the number of polygons. The most disappointing result from the projection methods is the sparse texture space coverage, which also causes the low scaling results which causes the blocky sampling in Figure 17 (right). These artifacts are caused by the bounding-rectangle packing algorithm used to place the connected mesh components into texture space. The packing results could be improved with the implementation of a more advanced general polygon packing algorithm, such as the one described by [13].

While the scalene triangle packing method is the best choice to avoid distortion, and packs texture memory more efficiently than the projection methods, its disregard for mesh topology results in numerous seams. These seams are more obvious than the seams resulting from the simple and area approximating meshes because the scalene packing seams bound the packed polygons against the texture's initial (background) color which creates a more noticable artifact than the color of other similarly textured polygons.

# 5. CONCLUSION

We have shown how procedural solid texturing can be constructed using standard texture mapping functionality. We have also described and measured several algorithms for implementing the u-map to transform model polygons into the texture map without overlap.

We note that unlike surface texture coordinates, solid texture coordinates are not uniformly implemented by graphic file formats. Using surface texturing of solid textures 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. However, if the u-map does not preserve the topology of the original mesh, the resulting model description file will necessarily contain multiple copies of the same model vertex with different **u** values, which can increase the size of some model description files.

## 5.1 Implementation

For testing a proof-of-concept prototype, we have implemented the solid map using an existing procedural texturing rasterizer, and combined Phases 1 and 2 in the solid map construction. Instead of storing the solid texture coordinates in a solid map, our current software rasterizer executes the procedural texturing procedure on the solid texture coordinates as they are interpolated, and stores the result in the texture map.

We are currently porting the solid map to an OpenGL implementation. Since the method does not require extension to the OpenGL API, it can run at the same level as the GLU library, as a utility library on top of the OpenGL standard functionality.

Since graphics API's typically do not store texture coordinates in the frame buffer, we use the color channels to interpolate the solid texture coordinates. Figure 19 shows a sample implementation of this technique in OpenGL.

```
float x[M][3][3]; /* model coords */
float s[M][3][3]; /* solid tex co-
ords */
float u[M][3][2]; /* surface tex co-
ords */

for (j = 0; j < m; j++) {

  umap(x[j],s[j],u[j]);

  glBegin(GL_POLYGON)
    glColor3fv(s[j][0]);
    glVertex2fv(u[j][0]);
    glColor3fv(s[j][1]);
    glVertex2fv(u[j][1]);
    glColor3fv(s[j][2]);
    glVertex2fv(u[j][2]);
  glEnd();
}
```

```
for (j = 0; j < m; j++) {
  glTexCoord2fv(u[j][0]);
  glVertex3fv(x[j][0]);
  glTexCoord2fv(u[j][1]);
  glVertex3fv(x[j][1]);
  glTexCoord2fv(u[j][2]);
  glVertex3fv(x[j][2]);
}
```

Figure 19: OpenGL code for Phase 1 (left) and Phase 3 (right) of the solid mapping technique.

The problem with using the color channels to interpolate texture coordinates is the depth of the color channel. Color channels are commonly 8 bits deep, which allow 256 levels of quantization of the solid texture coordinates.

Figure 17: Kangaroo u-map artifacts from simple mesh (left), area approximation (center) and cube projection (right).

Phase 2 of the solid mapping technique reads each solid texture coordinate at each pixel $\mathbf{u}$ in the solid map, executes the texturing procedure on the solid coordinates, and places the resulting color in the texture map at the same coordinates $\mathbf{u}$ in the texture map. As shown in Figure 19 Phase 3 uses standard texture mapping to display the results.

## 5.2 Future Work

The solid map provides a method for procedural solid texturing to use an intermediate stored image texture. This stored image texture allows standard texture antialiasing techniques to be applied to a procedural solid texture. MIP mapping is an obvious technique to use as it is supported by several graphics libraries. However, MIP mapping does not apply to the solid mapped triangles in the texture map because the triangles in the texture map are not necessarily continuously arranged, nor are they necessarily in mesh proximity. Hence the lower-resolution levels of the MIP map will include area samples combined from unrelated polygons. One solution would generalize the quadtree MIP map boundary structure with a $k$-d tree. Another solution is to pack the triangles along a space-filling Hilbert curve to better ensure that neighboring polygons in the model are proximate in the texture map.

Another u-map we plan to investigate is the topological cut. Some new methods based on the Morse theory of meshes are being developed to find the smallest collection of edges to cut to be able to spread a given model flat, like a bearskin rug. The number of cuts necessary is equal to the genus of the model, but the number of edges needed for each cut may be more than one. Nonetheless, this topological cut method should greatly decrease the resulting seams and yield a mesh in the texture map with little topological difference to the model's polygon mesh. We expect however that the distortions necessary to spread the model flat will be large.

## 5.3 Acknowledgments

Chris Thorne, Claudio Gatti and Andrew Woo worked on the prior implementation of the solid mapping technique for Alias | Wavefront's Maya, and their private communication on this research has been insightful.

Ulrike Axen was very helpful in finding polygon packing results in computational geometry. We expect her research based on the Morse theory of polygonal meshes to yield the proposed topological cut method very soon.

Conversations with Michael McCool have been helpful in understanding the breadth of real-time procedural shading techniques, especially during a visit with the second author at Waterloo. Thanks also to Kurt Akelely and Pat Hanrahan for an interesting conversation with the second author on antialiasing and deferred shading at the Hardware Workshop.

The research of the first two authors are supported in part by a grant from Evans and Sutherland Computer Company, thanks especially for the efforts of Pete Doenges and Steve Tibbitts.

# 6. REFERENCES

[1] T. Apodaca. *Advanced Renderman*, chapter Renderman Tricks Everyone Should Know. SIGGRAPH 98 Course Notes, July 1998. Also available in SIGGRAPH 99 Course Notes.

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

[3] D. Brinsmead. Convert solid texture. *Alias | Wavefront Power Animator*, 5, 1993.

[4] M. D. Carmo. *Differential Geometry of Curves and Surfaces*. Prentice-Hall, 1976.

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

[6] P. Hanrahan. Procedural shading (keynote). *Eurographics / SIGGRAPH Workshop on Graphics Hardware*, Aug. 1999. http://graphics.stanford.EDU/ hanrahan/talks/rtsl/slides/.

[7] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. *Computer Graphics*, 24(4):289–298, Aug. 1990.

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

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

[10] S. Ma and H. Lin. Optimal texture mapping. *Proc. Eurographics '88*, pages 421–428, Sept. 1988.

[11] J. Maillot, H. Yahia, and A. Verroust. Interactive texture mapping. *Proc. SIGGRAPH 93*, pages 27–34, Aug. 1993.

[12] M. D. McCool and W. Heidrich. Texture shaders. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 117–126, Aug. 1999.

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

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

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

[16] M. Olano and A. Lastra. A shading language on graphics hardware: The PixelFlow shading system. *Proceedings of SIGGRAPH 98*, pages 159–168, July 1998.

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

[18] M. Peercy. Re: Hardware accelerated renderman. *comp.graphics.rendering.renderman*, 25 Aug. 1999. Refers to http://reality.sgi.com/blythe for more details.

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

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

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

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

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

[24] C. Thorne. Convert solid texture. *Alias | Wavefront Maya*, 1, 1997.

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

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