Chapter 6

Procedural Solid Texturing

John C. Hart

Proc. Eurographics/SIGGRAPH Graphics Hardware Workshop, Aug. 1999, pp. 45-53.

Antialiased Parameterized Solid Texturing Simplified for Consumer-Level Hardware Implementation

John C. Hart, Nate Carr, Masaki Kameya

Washington State University

Stephen A. Tibbitts, Terrance J. Coleman

Evans and Sutherland Computer Corp.

Abstract

Procedural solid texturing was introduced fourteen years ago, but has yet to find its way into consumer level graphics hardware for real-time operation. To this end, a new model is introduced that yields a parameterized function capable of synthesizing the most common procedural solid textures, specifically wood, marble, clouds and fire. This model is simple enough to be implemented in hardware, and can be realized in VLSI with as little as 100,000 gates.

The new model also yields a new method for antialiasing synthesized textures. An expression for the necessary box filter width is derived as a function of the texturing parameters, the texture coordinates and the rasterization variables. Given this filter width, a technique for efficiently box filtering the synthesized texture by either mip mapping the color table or using a summed area color table are presented. Examples of the antialiased results are shown.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture --- Graphics processors; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism --- Color, shading, shadowing and texture.
Keywords: antialiasing, hardware. procedural texturing, solid texturing.

1. INTRODUCTION

Peachey [1985] and Perlin [1985] introduced procedural solid texturing as a method for simulating the sculpture of objects (of arbitrary detail and genus) out of a solid material such as wood or stone, and also the simulation of the natural elements of fire, water (waves), air (clouds) and earth (terrain and planets). Figure 1 through Figure 6 illustrate the variety of images that can be synthesized using procedural solid textures.

Addresses: WSU, School of EECS, Pullman. WA 99164-2752 {hart,ncarr,mkameya}@eecs.wsu.edu.
E&S (Seattle), 33400 8th Ave. S. #136, Federal Way, WA 98003 {stibbitt,tcoleman}@es.com.

Solid texturing creates the illusion that a shape is carved out of a solid three-dimensional substance. The details of a solid texture align across edges and corners of an object surface. For example the grain features on the teapots in Figure 1 and Figure 2 align with the block of material out of which they were sculpted. Depending on the detail and genus of the object, similar alignment of 2-D image texture maps can be very tricky [Peachey, 1985].

Procedural textures require much less memory than stored image textures, and unlike image textures their resolution depends only on computation precision. The sky and water in Figure 3 extend to infinity with non-repeating procedural detail. The fire in Figure 4 is procedurally textured on a single polygon. Zooming into the coastlines of the planet in Figure 5 reveals an arbitrarily intricate level of detail depending on the number of noise functions used in its generation. Figure 6 simulates the reflection of the moon on water without ray tracing or environment mapping by clever manipulation of the color maps of a procedural texture.

While this popular, powerful and flexible technique is found in nearly all high-quality photorealistic rendering packages, it has not yet found its way into consumer-level hardware for real-time rendering. Procedural solid textures would greatly enrich the quality of some of the 2D-image-textured graphical elements found in 3-D interactive games and virtual worlds, not only with wooden and stone objects, but with expansive terrain, oceans and skies filled with non-repeating detail.

Hardware implementation would also support the real-time animation of procedural textures. Varying the parameters of a procedure yields a dynamic animated texture. Depending on the paths chosen through parameter space, these animations can smoothly loop or be non-repeating. These animated textures would support such effects as ripples forming in marble, fire exploding, waves gently rising and falling, clouds billowing, and continents forming on planets.

1.1. Previous Work

Some have identified memory bandwidth as a major obstacle in increasing the performance of real-time graphics hardware. While memory size grows at a rate of 50% per year (one thousandfold over the past two decades), memory bandwidth only grows 12% per year (only tenfold over the past two decades) [Torborg & Kajiya, 1996]. Texture mapping in particular relies heavily on memory, and the bandwidth of this memory is the primary factor limiting the number and complexity of 2-D image textures

Figure 1: Carved wooden teapot.

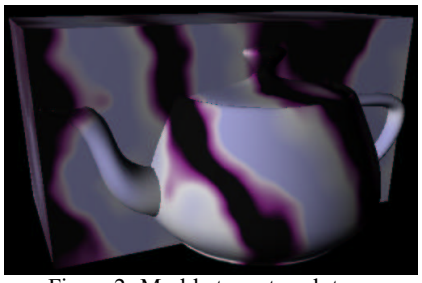


Figure 2: Marble teapot sculpture.

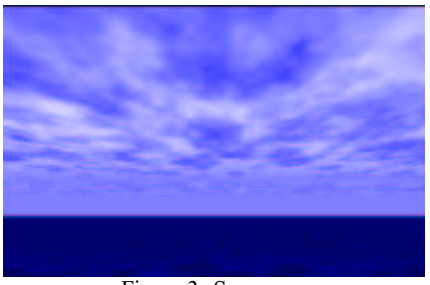


Figure 3: Seascape.

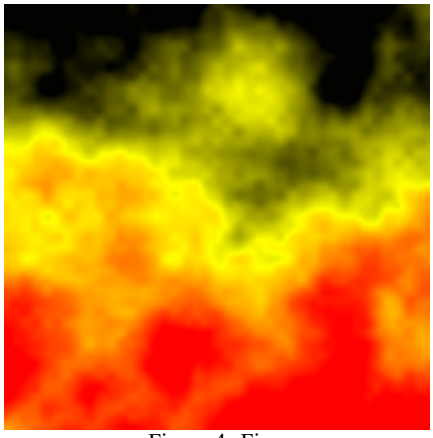


Figure 4: Fire.

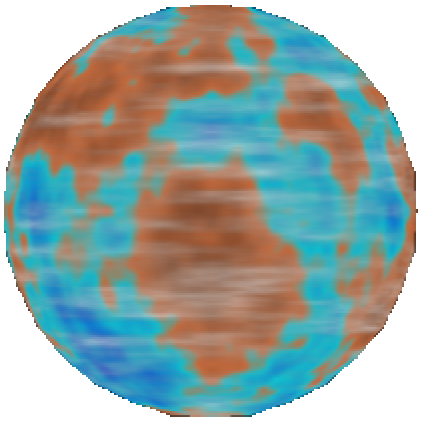


Figure 5: Planet.

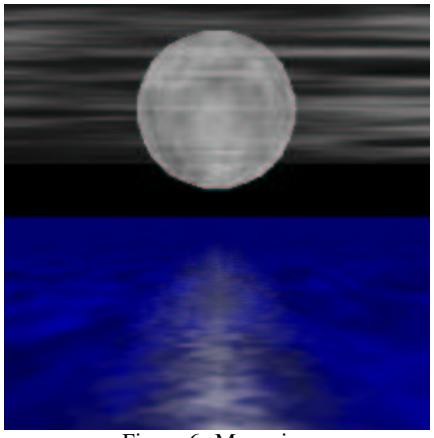


Figure 6: Moonrise.

available in real-time. Some have overcome the memory bandwidth limitation at the expense of increasing memory size to hold multiple redundant copies of the texture [Akeley, 1993], [Montrym, *et al.,* 1997]. Others relaxed the memory bandwidth limitation by reducing the size of the textures via compression [Torborg & Kajiya, 1996],[Beers, *et al.,* 1996]. Procedural texturing hardware is a way of increasing the performance of current graphics hardware by augmenting its existing pre-stored 2-D image textures with a variety of procedural solid textures without impacting the hardware's memory requirements.

Accessing a procedural texture requires more time than an image texture as the texture value must be computed instead of accessed from memory. Hence, real-time procedural texturing has previously only been available in high-end parallel graphics systems. For example, Pixel Planes [Rhoades, *et al.*, 1992], PixelFlow [Molnar, *et al.,* 1992] and the Pixel Machine [Potmesil & Hoffert, 1989] all supported real-time procedural texturing. Indeed, PixelFlow now has a fully-developed procedural shading system, including support for procedural solid texturing [Olano & Lastra, 1998].

Solid texturing is also not new to hardware implementation. The Reality Engine, for example, has the memory bandwidth necessary to support prestored solid texture volumes up to a maximum resolution of 256 x 256 x 64 texture elements [Akeley, 1993]. The InfiniteReality graphics system [Montrym, *et al.,* 1997] has 1GB of physical texture memory that could be organized into a 1024^3 pre-stored solid texture volume.

Antialiasing procedural textures is more complicated than for stored image textures. Whereas MIP maps [Williams, 1983] and summed-area tables [Crow, 1984] can be precomputed and stored for image textures, procedural textures are generated on the fly and such antialiasing techniques can not be readily applied. Supersampling is a common technique for antialiasing procedural textures but directly increases rendering time. For example, supersampling was the method used to inhibit aliasing in PixelFlow's procedural textures [Olano & Lastra, 1998]. Bandlimiting the procedural texture is also an effective technique [Norton, et al., 1982], but works easily and efficiently only on procedures based completely on spectral synthesis.

1.2. Overview

Section 2 introduces a texture model capable of synthesizing the most commonly used procedural textures (in fact all textures in Figure 1 through Figure 6) but concise enough to implement in hardware. The identification of this model allows the textures to be specified by parameters to a fixed procedure which can be simplified enough to be implemented in present-day VLSI technology.

Section 3 introduces a new method for antialiasing procedural textures based on computing a first order approximation of the color index variance over the area of a pixel. This approximation allows the antialiasing method to simulate an area sample of the textured image faster than supersampling. Unlike bandlimiting (which is a *pre*-filter), the new method is a *post*-filter that does not affect the parameters of the generation of the texture.

Section 4 exhibits the results of this model, exploring the various tradeoffs necessary to feasibly implement the model without significantly compromising image quality. An effective but reduced model can be implemented with as few as 100,000 gates, which is about 10% of the real-estate of modern consumer-level graphics processors.

2. A MODEL FOR PROCEDURAL TEXTURING

Various formalisms on procedural solid texture specifications have been proposed. Perhaps the most pervasive has been the Renderman shading language [Hanrahan & Lawson, 1990], but there are also other alternatives (e.g. [Abram & Whitted, 1990]). We propose a concise class of procedures capable of synthesizing a variety of textures and effects, but simple and direct enough to facilitate hardware implementation. The procedures are parameterized by values that completely control the type and character of the texture this model generates, such that these parameters (and the texture's color map) are the only representation of the texture that need be stored.

2.1. Analytical Model

Procedural solid texture mapping uses a mapping of the form $\mathbf{p}$: $\mathbf{R}^3 \rightarrow \mathbf{R}^4$ from solid texture coordinates $\mathbf{s} = (s,t,r)$ into a color space (R,G,B,α). (We follow the convention of using boldface to indicate vector values and functions, and italics to indicate scalar values and functions.) Some texture mapping techniques also include a homogeneous texture coordinate [Segal, *et al.,* 1992] but it remains to be explored how such a coordinate benefits procedural solid texturing. Often procedural solid textures incorporate a color map. In such cases, $\mathbf{p} = \mathbf{c} \circ f$ consisting of an implicit classification of the texture space f: $\mathbf{R}^3 \rightarrow \mathbf{R}$ and a color map $\mathbf{c}$: $\mathbf{R} \rightarrow \mathbf{R}^4$.

For a given polygon, the texture coordinate functions $\mathbf{s}(\mathbf{x}) = (s(\mathbf{x}),t(\mathbf{x}),r(\mathbf{x}))$ indicate the range of the texture coordinates with respect to screen coordinates $\mathbf{x}=(x,y)$. Hence, the procedural texture can be evaluated with respect to screen coordinates as $\mathbf{p}(\mathbf{x}) = \mathbf{c} \circ f \circ \mathbf{s}(\mathbf{x})$.

We restrict the texture map $\mathbf{p}$ to the family of functions

$$\mathbf{p}(\mathbf{s}) = \mathbf{c}\left(q(\mathbf{s}) + \sum_i a_i n(T_i(\mathbf{s})) \right), \tag{1}$$

where q: $\mathbf{R}^3 \rightarrow \mathbf{R}$ is a quadric classification function and n: $\mathbf{R}^3 \rightarrow \mathbf{R}$ is a noise function. The combination of quadrics and noise yields a specification sufficient to generate a wide variety of commonly used procedural solid textures. The affine transformations T_i control the frequency and phase of the noise functions.

2.1.1. Color Map

The color map $\mathbf{c}$ associates a color (R,G,B) with each index returned by the classification function f. The color map $\mathbf{c}$ is typically implemented as a lookup table

$$\mathbf{c}(f) = \mathbf{clut}[\text{round}(n \cdot \text{modclamp}(f))] \tag{2}$$

where **clut**[] is an array of n RGB color vectors. Color map indices returned by f are, depending on a flag parameter, either clamped to [0,1] or taken modulo one to map within the bounds of the lookup table.

2.1.2. Quadric Classification Function

The function q: $\mathbf{R}^3 \rightarrow \mathbf{R}$ in (1) is the quadric

$$q(s,t,r) = As^2 + 2Bst + 2Csr + 2Ds + Et^2 + 2Ftr + 2Gt + Hr^2 + 2Ir + J \tag{3}$$

which can more conveniently be represented homogeneously as

$$q(\mathbf{s}) = \mathbf{s}^T Q \mathbf{s} = [s,t,r,1] \begin{bmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{bmatrix} \begin{bmatrix} s \\ t \\ r \\ 1 \end{bmatrix} \tag{4}$$

treating $\mathbf{s}$ as a homogeneous column vector [Blinn, 1982].

The quadric function supports the spherical, cylindrical, hyperbolic and parabolic classification of space for texturing.

2.1.3. Noise Function

The function n: $\mathbf{R}^3 \rightarrow \mathbf{R}$ in (1) is an implementation of the Perlin noise function [Perlin, 1985]. The values a_i control the amplitude of the noise function, whereas the affine transformation T_i controls the frequency and phase of each noise component. There are a fixed number of noise components available, and this limit is typically between four and eight in typical texturing examples.

2.2. Texture Examples

The space of solid textures spanned by (1) covers the textures most commonly found in procedural solid texturing. The four fundamental procedural solid textures are: wood, clouds, marble and fire.

2.2.1. Wood

The texture model generated the wood texture shown in Figure 1, by using the quadratic function to classify the texture space into a collection of concentric cylinders [Peachey, 1985]. Waviness in the grain is created by modulation of a noise function

$$f(s,t,r) = s^2 + t^2 + n(4s,4t,r). \tag{5}$$

The color map consists of a modulo-one linear interpolation of a light "earlywood" grain and a darker "latewood" grain. The quadric classification makes the early rings wider than the later rings, which is to a first approximation consistent with tree development.

2.2.2. Clouds

Cloudy skies are made with a fractal $1/f$ sum of noise

$$f(\mathbf{s}) = \sum_{i=1}^{4} 2^{-i} n(2^i \mathbf{s}). \tag{6}$$

The texture described by (6) is mapped onto a very large high-altitude polygon parallel to the ground plane in Figure 3, resulting in clouds that become more dense in the distance due to perspective-corrected texturing coordinate interpolation. The color map is a clamped linear interpolation from blue to white. The water is the same procedural texture with a blue-to-black colormap.

2.2.3. Marble

Marble uses the noise function to distort a linear ramp function of one coordinate [Perlin, 1985]

$$f(s,t,r) = r + \sum_{i=1}^{4} 2^{-i} n(2^i s, 2^i t, 2^i r)). \tag{7}$$

The color map consists of a modulo-one table of colors from a cross section of the marble. Figure 2 demonstrates the marble texture on a cube, and the solid texturing again aligns the texture details on the edges of the cube. Continuously increasing the noise amplitude animates the formation of the ripples in the marble, simulating the pressure and heating process involved in the development of marble [Ebert, 1994].

2.2.4. Fire

Like marble, fire is simulated by offsetting a texture coordinate with fractal noise [Musgrave & Mandelbrot, 1989]. The fire example shown in Figure 4 was textured onto a single polygon and modeled as

$$f(s,t,r) = r + \sum_{i=1}^{4} 2^{-i} n(2^i s, 0, 2^i r + \phi)). \quad (8)$$

Continuously varying the noise phase term ϕ animates the fire texture.

2.2.5. Planet

A wide variety of different worlds, such as the one shown in Figure 5, can be generated by applying fractal textures, such as (6), to spheres. The color map for such images resembles a cartographic "legend." The cloudy atmosphere was rendered on the same sphere "over" the planet in a second pass using a color map with varying opacity values.

2.2.6. Moonrise

The moonrise in Figure 6 was rendered completely using synthesized textures, without any other kind of shading. The moon is a sphere with a fractal texture. The clouds were rendered on a single polygon perpendicular to the viewer and imposed over the moon. The water was rendered with a single polygon extending off to infinity. The highlight on the water was faked with two triangles textured using (7) with a partially transparent color map.

3. ANTIALIASING

Image texture aliases occur due to texture magnification and minification. Texture magnification occurs when the texture image itself contains too few samples such that a single texture element projects to several screen pixels. Texture minification results when the projection of the texture image covers too few pixels and several texture elements project to the same screen pixel. Modern texture mapping hardware inhibits aliases due to texture magnification by bilinear or bicubic interpolation of the appropriate texture elements. Such hardware inhibits texture minification aliases through the use of a MIP map that precomputes lower resolution versions of the texture, and samples the MIP map using trilinear or tricubic interpolation of neighboring pixels at the appropriate resolution level.

Aliases of synthesized textures do not fall into such categories since there is no fixed image resolution. Each such texture will exhibit some form of aliasing if sampled below twice the highest frequency in the texture's spectrum, which may be infinite for some textures. Hence, procedural textures do not suffer from magnification aliases, but require filtering to remove frequencies above the Nyquist limit to avoid minification aliases.

Synthetic textures could be antialiased by precomputing them, storing the results in MIP-mapped image textures. However, such

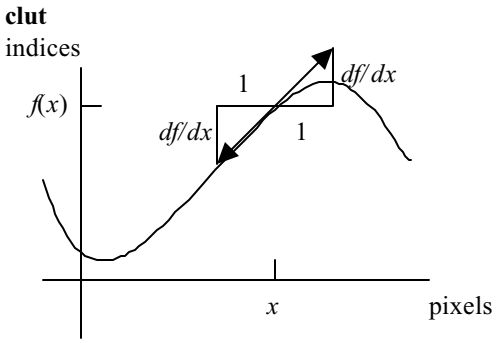


Figure 7: The derivative *df/dx* approximates the extent of the color map indices one pixel in either direction. Half of the derivative estimates the variation in color map indices half of a pixel in either direction.

an antialiasing technique would remove the flexibility such textures provided, and would also consume a tremendous amount of space when used on solid textures. Band limiting the output of the texture map removes aliases by prefiltering the texture before sampling [Norton, et al., 1982], but is difficult to implement in a generalized texturing environment. Supersampling the texture degrades time perfomance and arbitrarily increases the complexity of the hardware implementation.

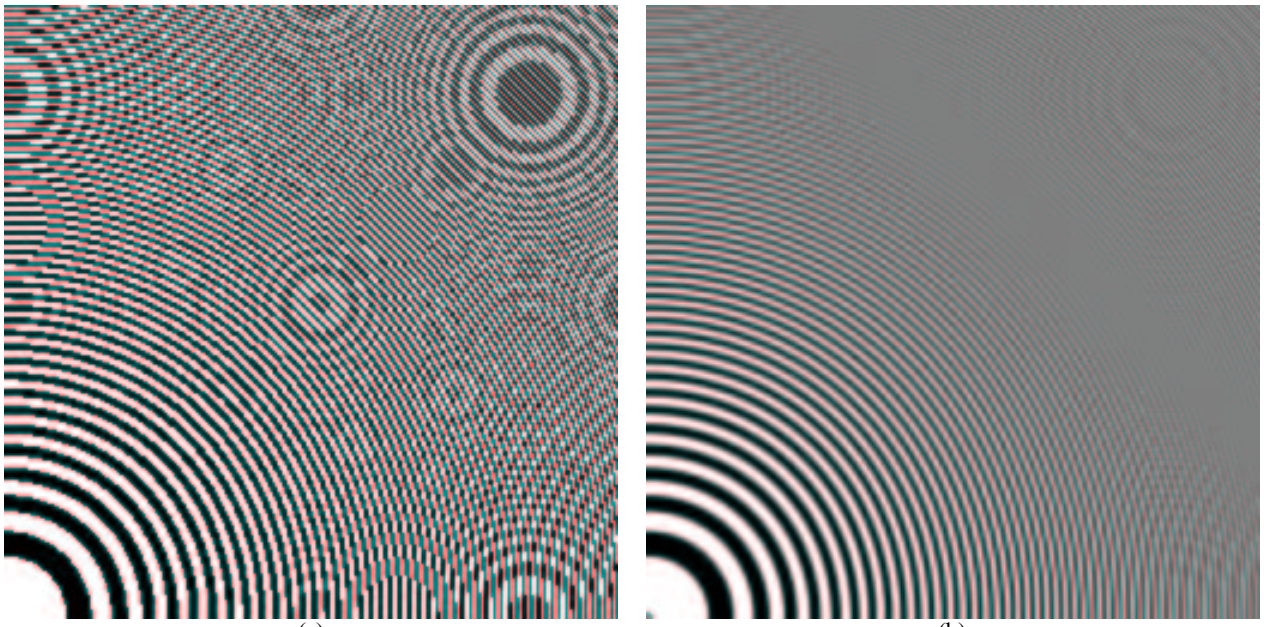
Instead, we analyze the function $\mathbf{p}(\mathbf{x})$ that textures pixels to determine the width of a box filter that would eliminate the aliasing frequencies from the spectrum of the synthesized texture. Several have described techniques for antialiasing procedural textures by antialiasing the textures' colormaps [Rhoades, *et al.*, 1992], [Worley, 1994]. In the next section, we provide a more rigorous mathematical justification and derivation of the technique, resulting in an ideal filter width for the texture which is used to box filter to the procedural texture by averaging the elements of the color table that the texture procedure generates over the support of the filter.

3.1. Texture Filtering via Color Table Filtering

Consider a domain D on the screen consisting of pixels whose color is determined solely by the projection of a single procedurally texture mapped polygon. We assume the color map indices generated by the procedural texture are continuous across the polygon. Let $a = \min_D f(\mathbf{x})$ be the least possible color map index used in the pixels in D, and let $b = \max_D f(\mathbf{x})$ be the greatest such index. Then we assume

$$\frac{\int_D \mathbf{p}(\mathbf{x})d\mathbf{x}}{\int_D d\mathbf{x}} \approx \frac{\int_a^b \mathbf{c}(f)df}{b-a} \quad (9)$$

the average color in D is sufficiently approximated by the average of the color table entries between indices a and b. As shown in Figure 7, we provide a first-order approximation of the bounds a and b used in the RHS of (9) by differentiating the texture function $f(\mathbf{x})$ and setting $a = f(\mathbf{x}) - \|\nabla f(\mathbf{x})\|/2$ and $b = f(\mathbf{x}) + \|\nabla f(\mathbf{x})\|/2$. If either a or b or both fall outside the bounds of the color table, then the boundary of the color table is extended using



(a) (b)

Figure 8: Zone plate aliased (a) and filtered (b).

either the modulo or clamp operators according to the modclamp flag.

The remainder of this section describes this differentiation in detail, applies efficient methods for integrating the color map to determine the numerator of the RHS of (9), and demonstrates the results.

3.2. Differentiating the Texture Procedure

The magnitude of the gradient $\nabla f = (\partial f/\partial x, \partial f/\partial y)$ indicates the width of the appropriate filter on the color map. From (1), we have that the gradient of f is

$$\nabla f = \nabla q + \sum_i a_i \nabla n_i \tag{10}$$

where n_i is the ith noise function: $n(T_i(\mathbf{s}))$. From (3) we have that the gradient of q is

$$\begin{aligned} \nabla q(\mathbf{x}) &= \mathbf{s}^T Q \frac{d\mathbf{s}}{d\mathbf{x}} + \left(\frac{d\mathbf{s}}{d\mathbf{x}}\right)^T Q\mathbf{s} \\ &= 2\mathbf{s}^T Q \frac{d\mathbf{s}}{d\mathbf{x}} \\ &= 2\begin{bmatrix} s & t & r & 0 \end{bmatrix} \begin{bmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{bmatrix} \begin{bmatrix} \frac{\partial s}{\partial x} & \frac{\partial s}{\partial y} \\ \frac{\partial t}{\partial x} & \frac{\partial t}{\partial y} \\ \frac{\partial r}{\partial x} & \frac{\partial r}{\partial x} \\ 0 & 0 \end{bmatrix} \end{aligned} \tag{11}$$

since Q is symmetric.

The derivative of the noise terms are given by

$$a_i \nabla n(T_i\mathbf{s}(\mathbf{x})) = a_i \frac{dn(T_i\mathbf{s}(\mathbf{x}))}{d\mathbf{s}} T_i \frac{d\mathbf{s}}{d\mathbf{x}}. \tag{12}$$

The gradient $dn/d\mathbf{s} = [\partial n/\partial s \ \partial n/\partial t \ \partial n/\partial r \ 0]$ is also known as the function DNoise [Perlin, 1985].

The value $d\mathbf{s}/d\mathbf{x}$ is the Jacobian of the texture coordinates $\mathbf{s}$ with respect to the screen coordinates $\mathbf{x}$. The values of $d\mathbf{s}/dx$ is computed during the scan conversion of the polygon as the perspective-corrected pixel increments. The values of $d\mathbf{s}/dy$ can be computed for each triangle using the plane equation and performing a perspective-correcting division.

3.3. Filtering the Color Table

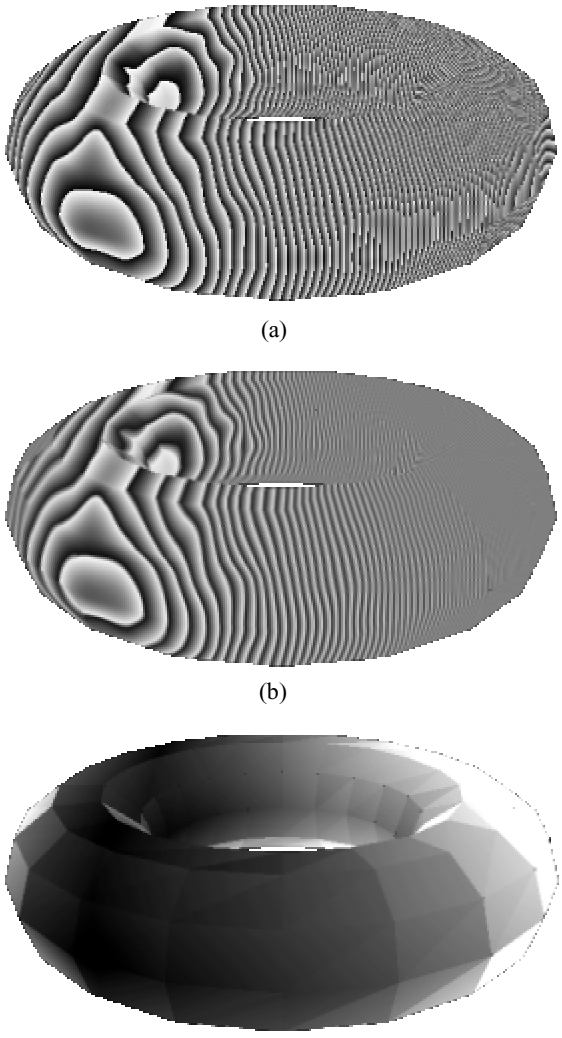
The filtering of color map values can be evaluated efficiently using either a color table MIP map or a summed area color table.

3.3.1. Color table MIP map

MIP maps are commonly used in standard texturing systems to prefilter image textures and sample from the prefiltered texture when the texture is minified (insufficiently sampled by the image pixels) [Williams, 1983].

One may also create a MIP map of a color table. The process begins with the n-element full resolution color table $\mathbf{clut}_1[]$. Then neighboring colors in the table are averaged to create a half-resolution $n/2$-element color table $\mathbf{clut}_2[]$. This process is repeated until a one-element color table $\mathbf{clut}_{\lg n}[]$ results, representing the average color of the entire color table.

Given a filter width w, let i = floor(lg w). Then the proper resolution color table from the mip map is selected and the color indexed is returned as $\text{clut}_i[f/i]$ (or more accurately the linear or cubic interpolation of the values of $\text{clut}_i[f/i]$ and $\text{clut}_{i+1}[f/(i+1)]$).



(a) (b) (c)

Figure 9: Torus rendered with wood texture (a) is antialiased (b) using filterwidths shown in (c) ranging from one (black) to 256 (white).

3.3.2. Summed area color table

Image textures are also antialiased efficiently using the summed area table [Crow, 1984]. A summed area table transforms information into a structure that can quickly perform integration, specifically a box filtering operation.

The summed area color table consists of a table where each entry consists of the sum of all elements in the color table including the current entry's element

$$\mathbf{csat}[i] = \sum_{j=0}^{i} \mathbf{clut}[j] \tag{13}$$

or recurrently as $\mathbf{csat}[i] = \mathbf{csat}[i\text{-}1] + \mathbf{clut}[i]$. The current entry's element can be recovered by subtracting the previous summed area element from the current summed area element as

$$\mathbf{clut}[i] = \mathbf{csat}[i] - \mathbf{csat}[i\text{-}1] \tag{14}$$

for i > 0. Box filtering the color map entries for a given filter width is computed as

$$(\mathbf{csat}[f + w/2] - \mathbf{csat}[f - w/2])/w. \tag{15}$$

Special care must be taken for the cases where the support of the filter crosses the bounds of the color table. For the following cases let N is the number of entries in the color table.

- $w \geq N$: Return the average of the entire color map: $\mathbf{csat}[\text{N-1}]/N$.
- $f + w/2 \geq N$:
  mod: $(\mathbf{csat}[f + w/2 - \text{N}] + \mathbf{csat}[\text{N-1}] - \mathbf{csat}[f - w/2 - 1])/w$.
  clamp: $((f+w/2-(\text{N-1}))\mathbf{clut}[\text{N-1}] + \mathbf{csat}[\text{N-1}] - \mathbf{csat}[f-w/2\text{-}1])/w$.
- $f - w/2 < 0$:
  mod: $(\mathbf{csat}[f + w/2] + \mathbf{csat}[N\text{-}1] - \mathbf{csat}[N + f - w/2 - 1])/w$.
  clamp: $(-(f - w/2)\ \mathbf{clut}[0] + \mathbf{csat}[f + w/2])/w$.

An alternative to performing the above computations at render time is to use the above formulae to precompute a color summed area table three times as long, ranging from $-N$ to $2N - 1$.

3.4. Examples

The derivations in Section 3.2 show that procedural textures produce aliasing artifacts from three possible places.

1. **Quadric Variation:** The quadric classification changes too quickly: $\|dq/d\mathbf{s}\|$ too large.
2. **Noise Variation:** The noise changes too quickly: $a_i\|dn(T_i\mathbf{s})/d\mathbf{s}\|$ too large.
3. **Texture Coordinate:** The texture coordinates change too quickly: $\|d\mathbf{s}/d\mathbf{x}\|$ too large.

Each of these components can create a signal containing frequencies exceeding the Nyqist limit of the pixel sampling rate.

Figure 8 demonstrates quadratic variation aliasing (type #1) with a zone plate constructed from the procedure

$$f(s,t,r) = 50s^2 + 50t^2 \cdot \tag{16}$$

rendered with an extremely harsh "zebra" color map. Analysis of (16) shows that the aliases are governed by $\nabla f = dq/d\mathbf{s}\ d\mathbf{s}/d\mathbf{x}$, with $dq/d\mathbf{s} = (100\ s, 100\ t)$. The zone plate was plotted at a resolution of 256^2 and over the unit square in texture coordinate space, hence $\partial s/\partial x = \partial t/\partial y = 1/256$. Setting the colormap filter width to $(100\ s + 100\ t)/256$ reduces the aliases to the point of being barely noticable.

Noise variation aliases (type #2) happen in concert with texture coordinate aliasing (type #3), since in a single scene the frequency and amplitude of noise is constant, and only varies across the image with distance from the viewer. For example, the clouds on the horizon in Figure 3 do not alias near the horizon because the filter width is scaled in part by the noise function derivative, and increases as the magnitude of $d\mathbf{s}/d\mathbf{x}$ increases. In the distance as the projection of the noise reaches the Nyquist limit, the filter width reaches the size of the entire color table, yielding a homogeneous hazy blue color.

Figure 9 illustrates all three types of texture aliasing on a torus. The centerline of the woodgrain rings passes through the left side of the torus, creating grain of increasing frequency on the right. Hence the filterwidth increases from the left to the right side of

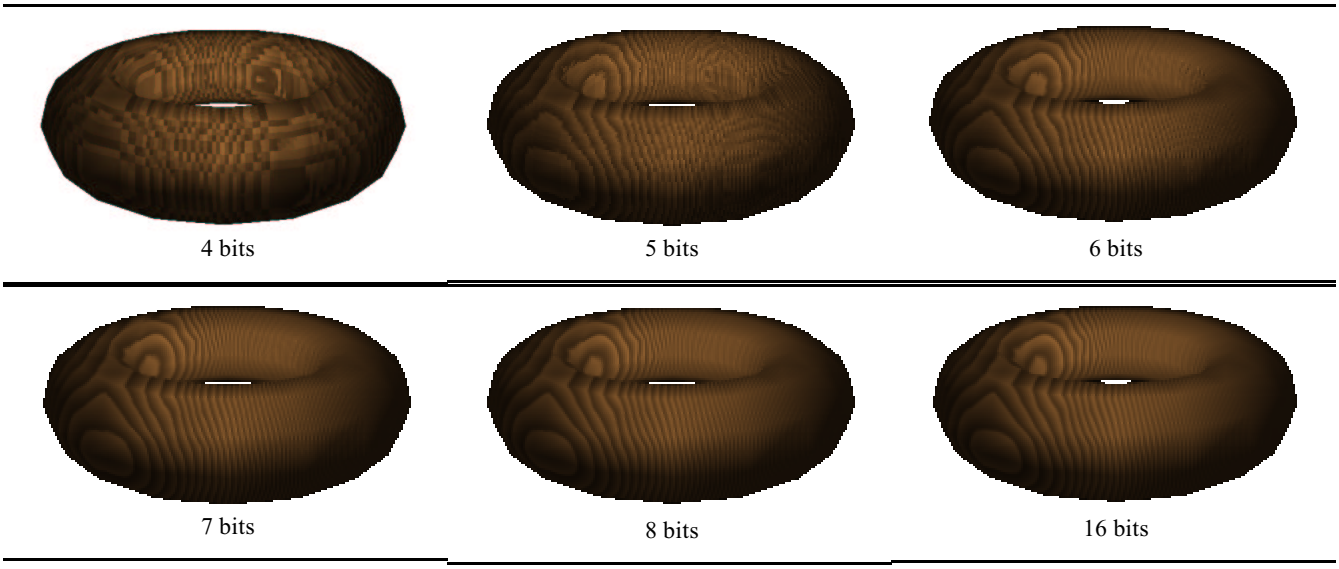


Figure 10: The effect of numerical precision on texture appearance.

the torus demonstrating quadric variation (type #1) aliasing. The amplitude and frequency of the noise term remains constant over the torus object, and so causes a uniform increase of the filterwidth due to noise variation (type #2) aliasing. The polygons on the silhouette of the torus have larger filterwidths than their neighbors, demonstrating texture coordinate (type #3) aliasing.

4. Results

The goal of the previous sections was to simplify the synthesis of antialiased solid textures. In this section, we describe and demonstrate software and simulated hardware implementations, and document some of the tests performed in the process.

4.1. Software Implementation

The basic tool of this research is a simulator that implements in fixed point arithmetic the texture synthesis model along with its associated filtering and color table mechanisms, as well as a prototype rasterizer. This simulator is responsible for all of the textured images in this paper. While the textures themselves were antialiased, the polygon edges were not. In fact, we avoided the temptation to use many small polygons to create smoother surfaces and silhouettes in order to better demonstrate the ability of procedural textures instead of geometry to provide visual detail.

This simulator serves as an antialiasing procedural texturing shader, and could be incorporated as a plug-in to existing software rendering systems. This simulator also serves as the basis of an extension to OpenGL, which already supports solid texture coordinates. The current implementation uses the OpenGL feedback buffer to collect the transformed polygons in screen coordinates for rasterization by the simulator [Carr & Hart, 1999]. The resulting textured raster image generated by the simulator is then combined with the raster image generated by OpenGL's rasterization engine using the associated z-buffers to negotiate visibility. Hence the simulator integrates synthesized solid textures into OpenGL's existing texturing, lighting and modeling system.

4.2. Hardware Implementation

A complete implementation of the model can be realized in VLSI with 1.25 million gates, resulting in the image quality shown in Figure 1 through Figure 6. A reduced and approximated version of the texture synthesis model can be implemented in as few as 100,000 gates. Sample images from such an implementation are exhibited in Figure 11.

Overall, the compromises in image quality necessary to implement the model in 100,000 gates appear minor, and the effects are very subtle. Some texture coordinate aliasing is noticable on the polygons of the teapots closest to the viewer. The character of the water, sky, planet and moonrise are slightly smoother due to a reduction in the number of noise function evaluations. The teapots and fire have noticable artifacts due to a linear approximation to the noise function.

4.3. Precision Tests

Several tests have been conducted to determine the texture coordinate precision necessary to avoid magnification aliases [Kameya & Hart, 1999]. Figure 10 shows the results of tests with a 512^2-pixel scene of a coarsely-triangulated objects computed using a variety of texture coordinate precisions.

4.4. Animation Tests

The seascape was animated to determine the effectiveness of the antialiasing technique. The seascape scene (Figure 3) was the most taxing on the colormap filtering algorithm because it textures infinite planes. Two animations of flights into the horizon were generated, one with and one without filtering. The unfiltered animation resulted in severe aliasing in the form of distracting noise near the horizon. The filtered animation significantly reduced these aliases, although some very slight flicker is still observable. This subtle flicker seems to be an inevitable compromise of the colormap-averaging filter in that removing the flicker results in textured planes that get too blurry too soon before reaching the horizon.

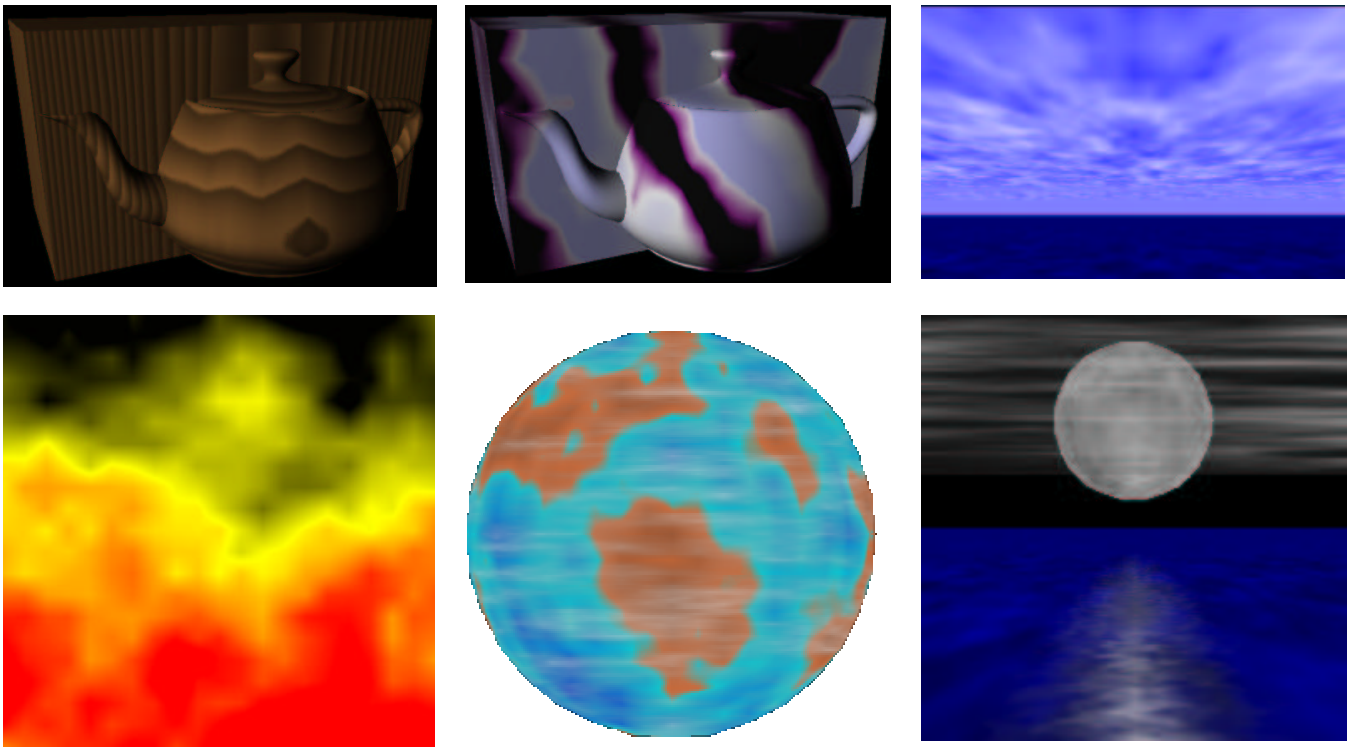


Figure 11: 100,000 gate simulations of Figure 1 through Figure 6.

The flame shown in Figure 11 was also animated to determine how effectively they would appear in the hardware implementation. The rectilinear grid basis of the noise functions is clearly evident due to the reduced number of noise octaves and the tri-linear interpolation. However the animation does clearly resemble burning flames and would sufficently represent such in typical consumer real-time graphics applications.

5. Conclusion

We set out to formalize a model for synthesizing popular procedural solid textures, and analyzed this model to derive an effective antialiasing scheme and an efficient hardware implementation. We showed that the model is capable of simulating the common procedural textures of wood, clouds, marble and fire, but is also simple enough to adequately implement in hardware.

Often textures are animated, to simulate fire, billowing clouds and other dynamic effects. Animation of texture map images requires a significant amount of texture memory and fast CPU access to the texture memory. The procedural texturing hardware will be capable of real-time animation of clouds billowing, fire burning and marble forming.

PixelFlow defered shading until after all of the rasterization was completed [Molnar, *et al.,* 1992]. It stored all of the shading information in the frame buffer, such that each pixel was shaded only once regardless of the number of polygons that overlapped it. The procedural texturing hardware described in this paper could be used to texture such pixels if the texture index, coordinates and Jacobian were stored in the framebuffer.

5.1. Future Work

This work only scratches the surface of procedural texturing hardware. Procedural texturing inexpensively overcomes the fundamental graphics texture rendering problems of memory bandwidth. With the success of this particular model, we expect other more sophisticated texturing models will be developed. The connotation of procedural texturing is that an actual program is run to generate the texture. While our model uses a fixed program with parameters controlling the character of its output, future procedural texturing hardware might be designed to permit uploading of texture programs. While such machines already exist (e.g. the Pixel Machine, Pixel Planes) there is no restriction on the texturing programs. Hence the user is burdened responsibility of antialiasing. Restricting the language used to write a procedural shader can increase the quality of its output, as it allows the hardware to better analyse the program to predict the aliases its output may contain, and automatically take measures to inhibit those aliases.

The antialiasing technique was derived from the model, but there is nothing specific to the model that makes this antialiasing technique work. Hence the color map antialiasing technique could be generalized and applied to any procedural texture so long as the derivatives are available. Computation of these derivatives is straightforward for this simple model, but could be quite complicated for true procedural textures described in a programming language. The error associated with approximation (9) should also be investigated further.

The colormap of the planet in Figure 5 is not continuous, jumping from a sandy color to an aquamarine to mark the coastlines of the world. As the filterwidth increases due to the noise contributions, this sharp coastline diffuses into a muddy color inbetween. A

more sophisticated antialiasing system might mark such jump discontinuities in the colormap and affect the filterwidth in these areas to further inhibit this artifact.

The noise function used was adapted from Rayshade [Skinner & Kolb, 1991], which uses cubic blending functions on a lattice of random numbers. This particular version lends itself to efficient hardware implementation, but the details of such an implementation are left as future work.

Procedural hardware need not be limited to just texture. Procedural hardware bump mapping, displacement mapping and shading in general seem to be logical extensions of this work. Recently, minor extensions to existing graphics pipelines for increased shading language support have been proposed [McCool & Heidrich, 1999]. Further extension might lead to the generation of procedural geometry that would overcome the bandwidth problem of transmitting polygons from the host to the graphics processor.

5.2. Acknowledgments

This research is supported in part by Evans and Sutherland Computer Corp., with a matching grant by the Washington Technology Center. This research was performed in part using the facilities of the Image Research Laboratory in the School of EECS at Washington State University.

Bibliography

[Abram & Whitted, 1990] Abram, Gregory D. and Turner Whitted. Building block shaders. *Computer Graphics 24*(4), (Proc. SIGGRAPH 90), Aug. 1990, pp. 283-288.

[Akeley, 1993] Akeley, Kurt. Reality engine graphics. *Computer Graphics 27*, Annual Conference Series, (Proc. SIGGRAPH 93), July 1993, pp. 109-116.

[Beers, *et al.,* 1996] Beers, Andrew C., Maneesh Agrawala and Navin Chaddha. Rendering from Compressed Textures. *Computer Graphics*, Annual Conference Series, (Proc. SIGGRAPH 96), Aug. 1996, pp. 373-378.

[Blinn, 1982] Blinn, James F., A generalization of algebraic surface drawing *ACM Transactions on Graphics 1*(3), July 1982, pp. 235-256.

[Carr & Hart, 1999] Carr, Nate and John C. Hart. APST Antialiased Procedural Texturing Interface for OpenGL. Proc. Western Computer Graphics Symposium. March 1999, pp. 46-55.

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

[Ebert, 1994] Ebert, David. Animating Solid Spaces: Animating Solid Textures. Chapter in: Texturing and Modeling: A Procedural Approach, Ebert, D., Ed. Academic Press Professional, Boston, 1984, pp. 165-170.

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

[Kameya & Hart, 1999] Kameya, Masaki and John C. Hart. Bit width necessary for solid texturing hardware. Proc. Western Computer Graphics Symposium. March 1999, pp. 121-126.

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

[Montrym, *et al.,* 1997] Montrym, John S., Daniel R. Baum, David L. Dignam and Christopher J. Migdal. InfiniteReality: A real-time graphics system. *Computer Graphics,* Annual Conference Proceedings, (Proc. SIGGRAPH 97), Aug. 1997, pp. 293-302.

[Musgrave & Mandelbrot, 1989] Musgrave, F. Kenton and Benoit B. Mandelbrot. Natura Ex Machina. *IEEE Computer Graphics and Applications 9*(1), Jan. 1989, p. 4-7.

[McCool & Heidrich, 1999] McCool, Michael D. and Wolfgang Heidrich. Texture Shaders. Proc. Eurographics-SIGGRAPH Graphics Hardware Workshop, Aug. 1999.

[Norton, et al., 1982] Norton, Alan, Alyn P. Rockwood and Phillip T. Skolmoski. Clamping: A method for antialiased textured surfaces by bandwidth limiting in object space. *Computer Graphics 16*(3), (Proc. SIGGRAPH 82), July 1982, pp. 1-8.

[Olano & Lastra, 1998] Marc Olano and Anselmo Lastra. A Shading Language on Graphics Hardware: The PixelFlow Shading System. Computer Graphics, Annual Conference Proceedings, (Proc. SIGGRAPH 98), July 1998, pp. 159-168.

[Peachey, 1985] Peachey, Darwyn. R. Solid texturing of complex surfaces. *Computer Graphics 19*(3), (Proc. SIGGRAPH 85), July 1985, pp. 279-286.

[Perlin, 1985] Perlin, Ken. An image synthesizer. *Computer Graphics 19*(3), (Proc. SIGGRAPH 85), July 1985, pp. 287-296.

[Potmesil & Hoffert, 1989] Potmesil, Michael and Eric M. Hoffert. The Pixel Machine: A parallel image computer. *Computer Graphics 23*(3), (Proc. SIGGRAPH 89), July 1989, pp. 69-78.

[Rhoades, *et al.*, 1992] Rhoades, John, Greg Turk, Andrew Bellm Andrei State, Ulrich Neumann and Amitabh Varshney. Real-Time Procedural Textures. Proc. Interactive 3-D Graphics Workshop, 1992. pp. 95-100.

[Segal, *et al.,* 1992] Segal, Mark, Carl Korobkin, Rolf van Widenfelt, Jim Foran and Paul Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics 26*(2), (Proc. SIGGRAPH 92), July 1992, pp. 249-252.

[Skinner & Kolb, 1991] Skinner, Robert and Craig E. Kolb. noise.c (file in the Rayshade raytracing system).

[Torborg & Kajiya, 1996] Torborg, Jay and James T. Kajiya. Talisman: Commodity realtime 3D graphics for the PC. *Computer Graphics* Annual Conference Proceedings, (Proc. SIGGRAPH 96), Aug. 1996, pp.353-363.

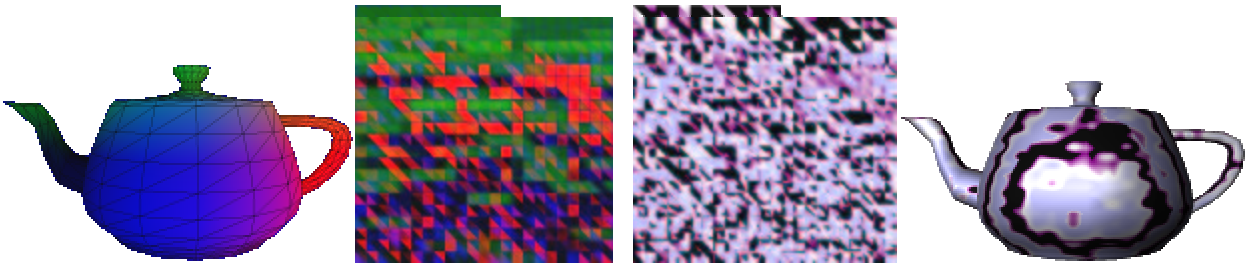
[Williams, 1983] Williams, Lance. Pyramidal parametrics. *Computer Graphics 17*(3), (Proc. SIGGRAPH 83), July 1983, pp. 1-11.

[Worley, 1994] Steven Worley. Practical Methods for Texture Design: Antialiasing. Chapter in: Texturing and Modeling: A Procedural Approach, Ebert, D., Ed. Academic Press Professional, Boston, 1984, pp. 117-124.

Real-Time Procedural Solid Texturing

Nathan A. Carr John C. Hart

Department of Computer Science
University of Illinois, Urbana-Champaign



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

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* $\mathbf{u} = \phi(\mathbf{x})$ 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 $\phi_i()$ 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 $\phi() = \cup \phi_i()$ is called an *atlas* [27]. If the surface texture mapping is one-to-one, then its inverse $\phi^{-1}()$ 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.

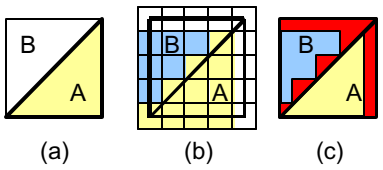


Figure 2. Seams occur due to differences between texture magnification (a) and rasterization (b), shown in red (c).

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

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

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 pixels 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.

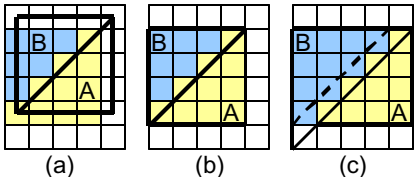


Figure 4. Standard rasterization rules disagree with texture magnification rules (a) and (b). Overscanned polygons are sampled correctly (c).

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.

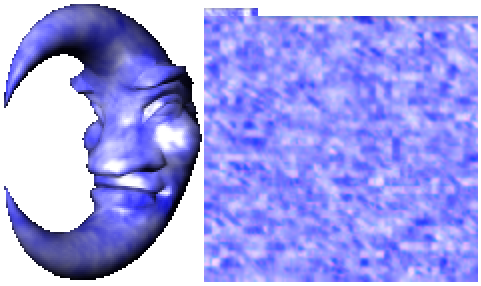


Figure 3. Uniform mesh atlas for a cloud textured moon.

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.

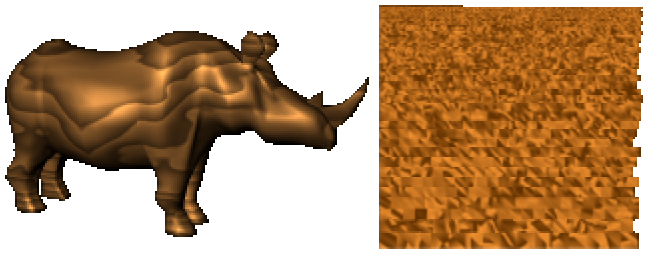


Figure 5: Rhino sculpted from wood and its area-weighted non-uniform mesh atlas.

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.

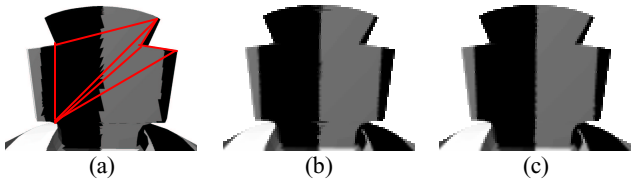


Figure 6. Effects of mesh atlas sample distribution techniques on a poorly tesselated object containing slivers: uniform (a), area weighted (b) and length weighted (c).

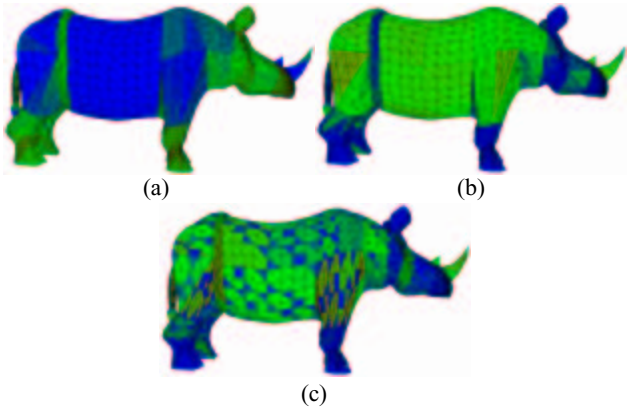
Figure 6 demonstrates the appearance of artifacts from the mesh atlases on the cross of a chess king piece. The procedural texture in this example is a simple striped pattern. Every triangle in the uniform mesh atlas (a) gets the same number of texture samples, regardless of size, resulting in the jagged sampling of the textured stripe on the left. The area-weighted NUMA reduces these aliasing artifacts, stealing extra samples from the rest of the model's smaller triangles. But the sliver polygon needs more samples than its area indicates, and the length-weighted 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.



(a) (b)

(c)

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.

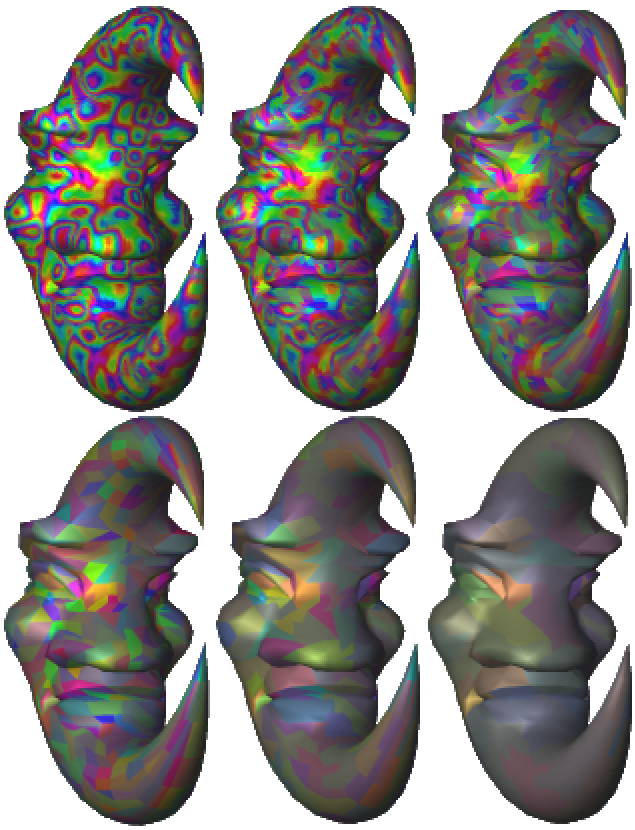


Figure 8. Levels of texture detail in the multiresolution uniform mesh atlas.

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.

Perlin Noise Pixel Shaders

John C. Hart
University of Illinois, Urbana-Champaign

Abstract

While working on a method for supporting real-time procedural solid texturing, we developed a general purpose multipass pixel shader to generate the Perlin noise function. We implemented this algorithm on SGI workstations using accelerated OpenGL PixelMap and PixelTransfer operations, achieving a rate of 2.5 Hz for a 256x256 image. We also implemented the noise algorithm on the NVidia GeForce2 using register combiners. Our register combiner implementation required 375 passes, but ran at 1.3 Hz. This exercise illustrated a variety of abilities and shortcomings of current graphics hardware. The paper concludes with an exploration of directions for expanding pixel shading hardware to further support iterative multipass pixel-shader applications.

Keywords: Pixel shaders, Perlin noise function, hardware shading, register combiners.

1. Introduction

The concept of procedural shading is well known [17][19], and has found widespread use in graphics [3]. Procedural shading computes arbitrary lighting and texture models on demand. Procedural textures efficiently support high resolution, non-repeating features indexed by three-dimensional solid texture coordinates. These features were quickly adopted for production-quality rendering by the entertainment industry, and became a core component of the Renderman Shading Language [5].

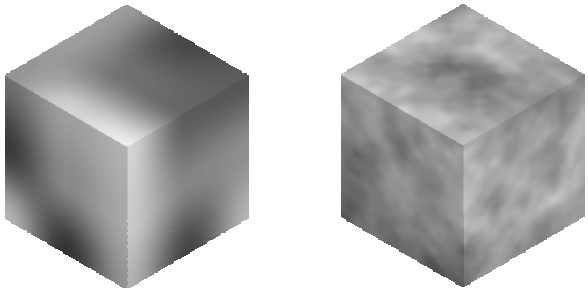
With the acceleration of graphics processors outpacing the exponential growth of general processors, there have been several recent calls for real-time implementations of procedural shaders, e.g. [6][20]. Real-time procedural shading makes videogames richer, virtual environments more realistic and modeling software more faithful to its final result. Real-time procedural texturing, in particular, allows modelers to use solid textures to seamlessly simulate sculptures of wood and stone. It yields complex animated environments with billowing clouds and flickering fires. Designers and users can interactively synthesize and investigate new procedural worlds that seem vaguely familiar to our own but with features unique to themselves.

Contact info: Dept. of Computer Science, 1304 W. Springfield Ave., Urbana, IL 61801, (217) 333-8740, jch@cs.uiuc.edu.

Several have researched techniques for supporting procedural shading with real-time graphics hardware [15][18][21][22]. These shading methods reorganize the architecture of the graphics API to suit the needs of procedural shading, applying API components to tasks for which they were not originally designed [8][11].

One such technique supports real-time procedural solid texturing [2] by using the texture map to store the shading of an object [1]. The technique maintains a texture atlas that maps triangles from a surface mesh into a non-overlapping array in texture memory. The triangles are plotted in texture memory using their solid texture coordinates as vertex colors. Rasterization then interpolates solid texture coordinates across their faces in the texture map. A procedural texturing pass replaces the solid texture coordinates in the texture map with the procedural texture color. Finally, this color is reapplied to the object surface via standard texture mapping. The result is a view-independent procedural solid texturing of the object.

One of the most common components of a procedural shading system is the Perlin noise function [19], a correlated three-dimensional field of uniform random values. This versatile function provides a deterministic random function whose bandwidth can be controlled to inhibit aliasing. Moreover, $1/f^{\beta}$ sums of noise functions can be used to form turbulence and other fractal structures whose statistics can be set to match those of various kinds of natural phenomena.



(a) (b)

Figure 1. Perlin noise function (a) and a $1/f$ sum (b).

We integrated the Perlin noise function into our real-time procedural solid texturing system in a variety of different ways, both as a CPU process and as a GPU process. This paper describes an algorithm for implementing the Perlin noise function as a multipass pixel shader. It also analyzes this noise implementation on a variety of systems. We used the available accelerated implementations of the OpenGL API and its device-dependent extensions on two SGI systems and an NVidia GeForce2. The paper concludes with suggestions for further

hardware accelerator development that would facilitate faster implementations of the Perlin noise function as well as a broader variety of texturing procedures.

2. Previous work

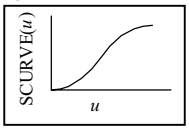
Because the Perlin noise function has become a ubiquitous but expensive tool in texture synthesis, it has been implemented in highly optimized forms on a variety of general and special purpose platforms.

Several fast host-processor methods exist for synthesizing Perlin noise. Goehring *et al.* [4] 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.* [10] used streaming SIMD instructions that forward differenced a linearly interpolated noise function for fast rasterization of procedurally textured triangles.

One can also generate solid noise with a 3-D texture array of random values [13], using hardware trilinear interpolation to correlate the random lattice values stored in the volumetric texture. Fractal turbulence functions can be created using multitexture/multipass modulate and sum operations. A texture atlas of solid texture coordinates would then be replaced with noise samples using the OpenGL pixel texture extension, ala [9].

The vertex-shader programming model found in Direct3D 8.0 [12] and the recent NVIDIA OpenGL vertex shader extension [16] can support procedural solid texturing. A Perlin noise function has been implemented as a vertex program [14]. But a per-vertex procedural texture produces vertex colors that are Gouraud interpolated across faces, such that the frequency of the noise function must be at, or less than half, the frequency of the mesh vertices. This would severely restrict the use of turbulence resulting from $1/f$ sums of noise. Hence the Perlin noise vertex shader is limited to low-frequency displacement mapping or other noise effects that can be mesh frequency bound.

Our favorite implementation of the Perlin noise function is from the Rayshade ray tracer [24]. This implementation created its own pseudorandom numbers by hashing integer solid texture coordinates with a scalar function Hash3d(i,j,k), then interpolated these random values with a simple smooth cubic interpolant SCURVE(u) = $3u^2 - 2u^3$ to yield the final result.



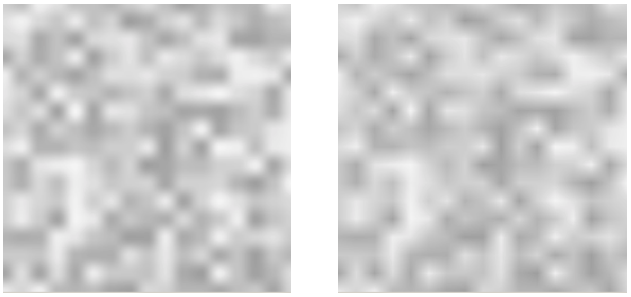
Given solid texture coordinates s,t,r, the Rayshade noise function effectively returned noise as the value

$$\sum_{k=0}^{1}\sum_{j=0}^{1}\sum_{i=0}^{1} \text{Hash3d}(\lfloor s \rfloor + i, \lfloor t \rfloor + j, \lfloor r \rfloor + k) w(s,i) w(t,j) w(r,k)$$

where

$$w(s,i) = \text{SCURVE}(s - \lfloor s \rfloor)^i \, (1 - \text{SCURVE}(s - \lfloor s \rfloor))^{1-i}$$

is a weighting function. Hence, the noise function returns a weighted sum of the random values at the eight corners of the integer lattice cube containing s,t,r.



(a) (b)

Figure 2. Result of the Rayshade implementation of the Perlin noise function, using cubic interpolation (a) and linear interpolation (b) of corner lattice random values.

Figure 2 demonstrates the result of the Rayshade implementation of the Perlin noise function. The random values result from the drand48() function of the standard C math library. Noise is defined on an integer coordinate lattice, which results in the strong horizontal and vertical correlation.

We will use this sample as a reference to compare our pixel-shader implementations of the Perlin noise function. The average brightness of the (s,t) slice of the noise is due to the fixed r coordinate. This average intensity will differ from across implementations, resulting in variations in brightness for a given (s,t) slice of the three-dimensional noise field.

3. A Multipass Noise Algorithm

We based our real-time implementation of the Perlin noise function on the concise Rayshade implementation. We implemented a per-pixel noise function using multipass rendering onto a texture atlas initialized with solid texture coordinates stored as pixel colors.

The Perlin noise function is defined on a field of real values, where the integer subset of its domain defines the base frequency of the noise. Implementation of the noise function requires coordinates s,t,r to range over multiple integers, though color components only range over [0,1]. Hence, given three channels (R,G,B) each with a depth of b bits[1], we use a fixed-point representation with b_i integer bits and b_f fractional bits, $b = b_i + b_f$.

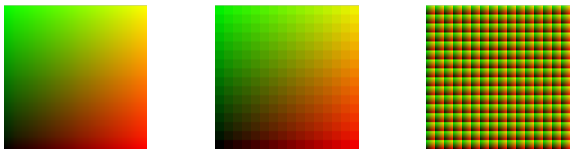
Following the form of the Rayshade noise implementation, the algorithm in Figure 3 computes a random value in [0,1] at the integer lattice points, and linearly interpolates these random values across the cells of the lattice.

[1] 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.

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

The input to the algorithm is an image solid_map whose R,G,B colors consist of solid texture coordinates. The first half of the algorithm decomposes solid_map into its integer part solid_int shifted right b_f times and a fractional part weight shifted left b_i times.



(a) (b) (c)

Figure 4. Solid texture coordinates solid_map (a), tex_int shifted left by b_f (b) and weight (fractional part shifted left by b_f) (c).

Figure 4 shows a sample texture map as a plane of two-dimensional solid texture coordinates spanned by s and t. We set b_f = 4 bits. The solid texture coordinates s,t,r range from (0.0,0.0,0.0) to (15.9375,15.9375,0.0) and are represented in the solid texture coordinate texture map Figure 4(a) with RGB colors from (0,0,0) to (1,1,0). Internally in the 24bpp framebuffer, these RGB colors range from (0,0,0) to (255,255,0). These coordinates are shifted right by b_f to form tex_int, which is shown Figure 4(b) shifted left by b_f to increase contrast and brightness. Subtracting (b) from (a) leaves tex_frac, which is shifted left by b_f to create a normalized weight function Figure 4(c).

The color (R,G,B) of each pixel (x,y) in solid_map corresponds to a solid texture point $(s=R,t=G,r=B)$ that falls within some lattice cell. The corner of this cell is given by the coordinates in the corresponding pixel (x,y) stored in solid_int. The opposite corner of this cell is found in the corresponding pixel in solid_intpp (whose colors are increments of those in solid_int).

Each of the eight corners of the cell can be found by combinations of the coordinates in solid_int and solid_intpp. The second half of the algorithm iterates over all eight corners, creating a random value indexed by the integer value at that corner. These random values are weighted by the fractional portion of the solid texture coordinates found in weight or its additive inverse. Summing the products of these weights for each of the eight corners performs a trilinear interpolation of the random values at the corners, resulting in result of the noise function.

We will spend the next two sections implementing this algorithm using the available accelerated features of two different graphics architectures. These implementations are each divided into two sections, on implementing the logical shift operations needed for the first half of the algorithm, and the random value synthesis needed for the second half.

4. SGI Implementation

The SGI graphics accelerators have focused on high-end real-time rendering for the scientific visualization and entertainment production communities. Hence accelerated features have included scientific imaging functions that support algebraic and lookup-table operations on pixels.

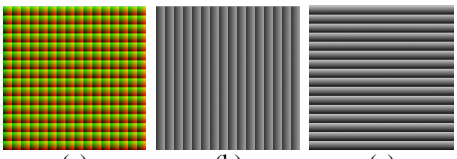
We focused our implementation on low end and midline SGI workstations, which are commonly deployed for digital content creation and design in both the videogame and animation communities.

4.1 PixelTransfer and PixelMap

We implemented the noise function in multipass OpenGL on SGI workstations using accelerated PixelTransfer[3] and PixelMap functions. The PixelTransfer function performs a per-component scale and bias, whereas PixelMap performs a per-component lookup into a predefined table of values.

We defined an assembly language of useful PixelTransfer functions. Specifically, the function setPixelTransfer(a,b) sets OpenGL to perform an $ax + b$ operation during the next image transfer operation, where x represents each component of the RGBA color. The function setPixelMap(table) uses PixelMap to replace colors channels with their corresponding entries in a lookup table. We also defined a blendtex(i) operation that draws the texture image corresponding to texture index i. The instruction savetex(i) saves the current framebuffer as texture image i.

Unlike the previous section, the SGI implementation begins with three luminance images tex_s, tex_t and tex_r instead of a single RGB image solid_map. We could perform all of the decompositions on a single texture, but we would later need to break its red, green and blue channels into individual luminance textures, and we found it impossible to perform this efficiently with the OpenGL extension set available to low-end and midline SGI workstations that lacked the color_matrix extension.



(a) (b) (c)

Figure 5. RGB image weight (a) is equal to (1,0,0) * luminance image tex_s (b) + (0,1,0) * luminance image tex_t (c) + (0,0,1) * luminance image tex_r (not shown).

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

[3] Following the convention of the OpenGL ARB, we avoid the use of the "gl" prefix for functions and the "GL_" prefix for tokens when describing elements of the OpenGL API.

4.2 Logical Shift Operations

The task of decomposing a texture map of fixed point solid texture coordinates into integer and fractional textures used PixelTransfer multiplication to achieve shifting operations. We defined an integer shift = $1 << b_f$. We modulated the texture by shift to perform a logical shift left by b_f, and by 1/shift to perform a logical shift right. (Some hardware required us to round instead of truncate, which was performed by a PixelTransfer bias of -0.5/255.0.) We also defined fracshift as $255.0/((1 << b_f) - 1)$. This allowed us to scale our fractional portions into normalized weights.

The following code fragment demonstrates the decomposition of the *s* coordinate. Similar decompositions need to be performed on tex_t and tex_r as well.

```
// shift s right to remove fractional part, save as si
blendtex(tex_s);
setPixelTransfer(1.0/shift, 0.0 /* or –0.5/255.0 */);
savetex(tex_si);
resetPixelTransfer();

// shift si back left
blendtex(tex_si);
setPixelTransfer(shift, 0.0);
CopyPixels(0,0,HRES,VRES,COLOR);
resetPixelTransfer();

// subtract si (floor of s) from s to get fractional part of s
Enable(BLEND);
BlendEquation(SUBTRACT);
BlendFunc(1, 1);
blendtex(tex_s);
Disable(BLEND);

// scale fractional part into normalized weight in [0,1]
setPixelTransfer(fracshift, 0.0);
savetex(tex_sf);
resetPixelTransfer();
```

4.3 Random Value Synthesis

We implemented randomization using a lookup table. This lookup table was accessed using the accelerated PixelMap OpenGL function. Recall the value k ranges from 0 to 7 denoting the current corner. The following code fragment synthesizes a random field based on the *s* coordinate.

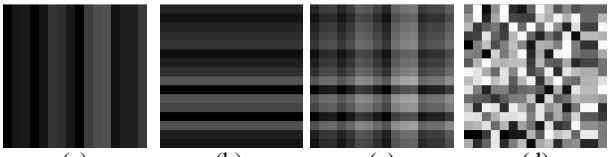
```
// tex_sin = random(si) or random(si++)
blendtex(tex_si);
setPixelTransferf(1.0, (k&1) ? 1.0/255.0 : 0.0);
setPixelMap(sran);
savetex(tex_sin);
```

Similar code fragments apply to the *t* and *r* coordinates, using (k&2) and (k&4) in the PixelTransfer, respectively. At this point tex_sin, tex_tin and tex_rin contain random values indexed by the *s*,*t*,*r* values at the *k*th corner of the cell. The following code fragment combines these three random values into a single random value.

```
// now tex_sin, tex_tin and tex_rin are random
// add them up into a single random number[4]
blendtex(tex_sin);
Enable(BLEND); BlendFunc(ONE,ONE);
blendtex(tex_tin);
blendtex(tex_rin);
Disable(BLEND);
```

This combination of random values is highly correlated due to the componentwise combination of random values. We reduce this correlation with an additional randomization pass.

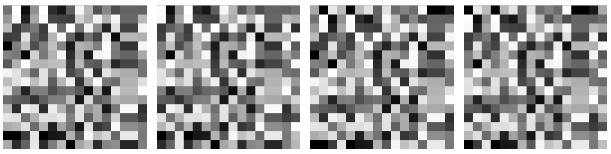
```
// one more randomization (in place)
setPixelMap(nran);
CopyPixels(0,0,HRES,VRES,COLOR);
resetPixelTransfer();
```



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

Figure 6. The sum of random numbers indexed by *s* (a). and *t* (b) is highly correlated (c). This correlation is reduced by indexing into a final randomization (d).

The random number tables sran, tran and rran are uniform random number distributions over the range [0,1/3]. These three random values are added to form the final distribution, which is slightly non-uniform and heavily coordinate correlated, as shown in Figure 6(c). An additional randomization reduces this correlation as shown in Figure 6(d).



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

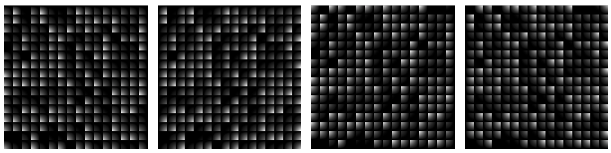
Figure 7. The random values at integer lattice locations for corners ($\lfloor s \rfloor$,$\lfloor t \rfloor$) (a), ($\lfloor s \rfloor+1$,$\lfloor t \rfloor$) (b), ($\lfloor s \rfloor$,$\lfloor t \rfloor+1$) (c) and ($\lfloor s \rfloor+1$,$\lfloor t \rfloor+1$) (d).

Figure 7 shows the random values generated at the four corners of the lattice. Note that in this example these are all translates of each other.

The random value is then weighted by the fractional part of the original texture coordinates s,t,r. Note that we have broken out the original RGB image weight from the previous section into three luminance images tex_sf, tex_tf and tex_rf. We also use the built-in additive complement blending operation to invert the weight appropriately depending on the cell corner.

```
// displayed texture now random value at corner k
// weight this contribution by fractional parts of s,t,r
Enable(BLEND);
BlendFunc(0, (k&1) ? SRC : 1 - SRC);
blendtex(tex_sf);
blendFunc(0, (k&2) ? SRC : 1 - SRC);
blendtex(tex_tf);
BlendFunc(0, (k&4) ? SRC : 1 - SRC);
blendtex(tex_rf);
```

[4] Note the addition of the component random values introduces a slight Gaussian bias to the resulting noise. This could be eliminated if an accelerated exclusive-or blending mode was available.



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

Figure 8. Random values scaled by the weight functions (1 - tex_sf)(1-tex_tf) (a), tex_sf(1-tex_tf) (b), (1-tex_sf)tex_tf (c) and tex_sf tex_tf (d).

Figure 8 shows the random values at the corners (Figure 7) scaled by the product of weighting functions tex_sf and tex_tf. These weighting functions are luminance textures corresponding to the individual channels of Figure 4(c), such that weight = (tex_sf, tex_tf, tex_rf).

The resulting weighted random value corresponding to the current corner is then added into a running total, as show in the following fragment.

```
// add noise component into noise sum
BlendFunc(1,1);
blendtex(tex_noise);
Disable(BLEND);

// keep track of sum
savetex(tex_noise);
```

The texture tex_noise is initialized to black. After all eight corners have been visited, tex_noise contains the final noise values corresponding to the solid texture coordinates in the input luminance images tex_s, tex_t and tex_r.

Figure 9. Noise function resulting from the sum of Figure 8 (a-d).

4.4 Results

Figure 9 shows the final noise function resulting from summing the images in Figure 8. The correlation from Figure 6(c) was reduced by the randomization in Figure 6(d) but is still evident, particularly in the final interpolated version, as strong horizontal and vertical tendencies in the noise. However, this correlation is also found in the reference noise implementation in Figure 2, and is primarily due to the integer lattice of noise values.

We implemented this algorithm at a resolution of 256^2 on a SGI Solid Impact, a SGI Octane, and an NVidia GeForce2. The SGI workstations are designed for advanced imaging applications and have hardware accelerated PixelTransfer and PixelMap operations whereas the NVidia card designed for mainstream consumer applications does not. The execution times are given in Table 1.

Implementation	Execution Time (Rate)
SGI Octane	0.4 sec. (2.5 Hz)
SGI Solid Impact	0.75 sec. (1.3 Hz)
NVidia GeForce 256	5 sec. (0.2 Hz)

Table 1. Execution results for the multipass noise algorithm.

5. NVidia Implementation

We also implemented a noise function for consumer-level accelerators using the NVidia chipset. The NVidia products have been designed to accelerate commodity personal computer graphics, especially videogames. Hence the drivers did not accelerate PixelTransfer and PixelMap. We instead used register combiners to shift, randomize and isolate/combine components.

5.1 Register Combiners

Register combiners support very powerful per-pixel operations by combining multitextured lookups in a variety of manners. They support the addition, subtraction and component-wise multiplication (and even a dot product) of RGB vectors. They also support conditional operations based on the high-bit of the alpha channel of one of the inputs. They support signed byte arithmetic with a full 9 bits per channel, though can only store 8 bit results. They also provide several mapping functions for signed/unsigned conversion, and the ability to modulate output values by one-half, two and four.

The Direct3D 8.0 specification includes a register-combiner based assembly language [12]. However, our implementation sought to squeeze the best possible performance out of the NVidia chipset. We chose instead to use the OpenGL register combiner extensions, which provide complete, though device dependent, access to the graphics accelerator.

Figure 10 illustrates the register combiner functionality used in this paper. The register combiner has four inputs A,B,C,D that can be any combination of the incoming fragment, a pixel from multitexture unit 0 or 1, and the contents of a scratch register called Spare0. The constants zero and one (via a special unsigned invert operation) can also be used as inputs, and other constant values can also be loaded via special registers.

The outputs of the register combiners include A*B, C*D, A*B + C*D and the special A*B | C*D. This latter output yields A*B if the alpha component of the register Spare0 is less than 0.5, otherwise the output yields C*D. These outputs can also be optionally scaled by ½, 2 or 4. For this paper, it is safe to assume the output is always contained in the register Spare0. The register combiner has separate but comparable functions for the RGB values and the alpha values of the inputs and registers.

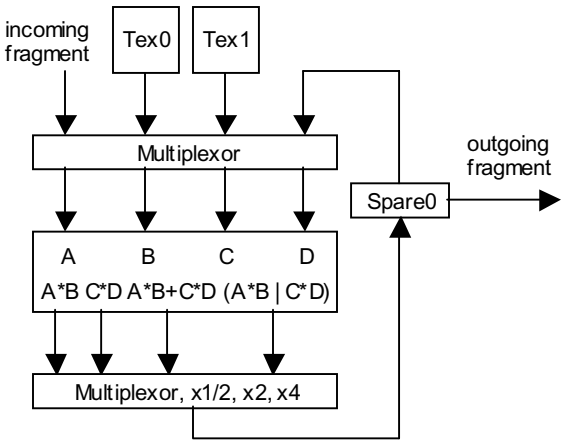


Figure 10. Partial block diagram of the register combiner functionality used in this paper.

There can be any number of register combiners that form a pipeline, using the temporary registers such as Spare0 to hold data between stages. The GeForce2 used to implement the pixel shaders in this paper contains two register combiners which allow two register combiner operations per pass. The GeForce3 is expected to have eight register combiners.

5.2 Logical Shift Operations

In order to perform the decomposition of the input solid texture coordinate image into integer and fractional components, we developed a logical shift left register routine. This routine used the modulate-by-two output mapping, but this causes values greater than one half to clamp to one. We avoided this overflow by using the conditional mode of the register combiners. The following example sets up the register combiners to perform such a logical shift left on a luminance value (R=G=B) in multitexture unit 0.

```
// first stage
// spare[α] = texture0[b]
A[α] = texture0[b]
B[α] = 1 (zero with unsigned_invert)
spare0[α] = A[α]*B[α]
// spare0 rgb = texture0 less its high bit (or zero if less than ½)
A[rgb] = texture0[rgb]
B[rgb] = white (zero with unsigned_invert)
spare0[rgb] = A[rgb]*B[rgb] - 0.5 // via bias_by_negative_one_half

// second stage
// spare0 rgb = (spare0[α] < 0.5 ? texture0[rgb] : spare0[rgb]) << 1
A[rgb] = texture0[rgb]
B[rgb] = white
C[rgb] = spare0[rgb]
D = white
spare0[rgb] = 2*(spare0[α]<0.5 ? A[rgb]*B[rgb] : C[rgb]*D[rgb])
```

We could also generate a register combiner to perform a logical shift right using the scale_by_one_half mode, but found it was much simpler to perform a multitextured modulate-mode blend with a texture consisting of the single pixel containing the RGB color (0.5,0.5,0.5).

5.3 Random Value Synthesis

Randomization on the NVidia controller was particularly difficult. The driver (and presumably the hardware) accelerated neither pixel transfer/mapping operations, nor logical operations like exclusive-or.

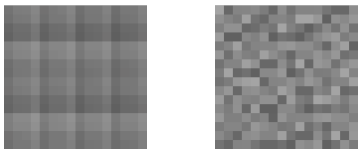
We instead implemented a register combiner random number generator by shifting each of the components of the integer values of the coordinates left one bit at a time. All four bits of each of the three components are at one point the high bit in multitexture unit 0. We then used the register combiner's conditional mode to display one of two colors depending on the high bit of the current texel of multitexture unit 0. The following code fragment implements this technique.

```
for (kk = 0; kk < 4; kk++) {
   for (comp = 0; comp < 3; comp++) {
      // display either tex_ranzero or tex_ranone
      // depending on hi bit of tex_comp
      setupblendhibit(ranzero[comp][kk],ranone[comp][kk]);
      blend2tex(tex_comp[comp],tex_corran);
      savetex(tex_corran);
      if (kk < 3) {
         // shift tex_comp left one
         setupshift1();
         blendtex(tex_comp[comp]);
         savetex(tex_comp[comp]);
      }
   }
}
```

The operation blend2tex(tex_a,tex_b) displays a multitextured image with tex_a as multitexture unit 0 and tex_b as multitexture unit 1.

The arrays ranzero and ranone were initialized with random luminances. These random luminances were used as input to the function setupblendhibit(rgba0,rgba1). This function set up a register combiner that would display either constant color rgba0 or rgba1 depending on the high bit of texture0, and would blend the color (rgba0 or rgba1) with texture1.

We found that setting the alpha channel of rgba0 and rgba1 to 1/8 provided a reasonable balance of colors after twelve successive blending operations. These blends were accumulated in tex_corran (corner random). Note that this loop involves 12 randoms + 9 shifts = 21 passes, which expands to 168 passes for all eight corners.



(a) (b)

Figure 11. Heavily correlated random values generated by blending random colors depending on the bits of the integer lattice value (a). Using (a) to index into a random value reduces the correlation (b).

The resulting tex_corran still exhibited some coordinate correlation, which we reduced with an additional eight single-bit randomizations on tex_corran, yielding tex_corranran. This step resulted in an additional 8 randoms + 7 shifts = 15 passes per corner for a total of 120 passes.

Due to the successive blending, the register combiner noise function is Gaussian distributed. A normal distribution could be recovered through a histogram equalization step, though such operations are not yet accelerated on consumer-level hardware.

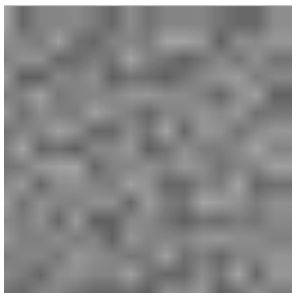


Figure 12. Noise function resulting from register combiners.

5.4 Results

The register combiner implementation resulted in 375 passes, but runs in .77 seconds at a resolution of 256^2 on a GeForce2 using version 12.0 of the "developer" driver. This results in a 1.3 Hz performance, which is suitable for interactive applications but is not yet real-time. A discussion of the reasons why the performance is slower than necessary is given later in Section 6.2.

The resulting noise is shown in Figure 12. The NVidia implementation blended random colors, yielding Gaussian noise, whereas the reference and SGI implementations produced white noise. If desired, one could redistribute the Gaussian noise into white noise with a fixed histogram equalization step, though no such operation is currently accelerated on NVidia GPUs.

6. Discussion

The implementation of the Perlin noise function on SGI and NVidia GPUs has been successful in that we found it was feasible, but disappointing in that subtle hardware limitations prevent truly efficient implementations. These limitations included the limited precision available in the 8 bit per component framebuffer, the delay in performing a CopyTexSubImage transfer from the framebuffer to the texture memory, and the lack of acceleration of loginal operation blend modes such as exclusive-or. The process has also been illuminating, and has inspired us with several ideas for further advancement in hardware design to overcome these limitations and better support efficient multipass pixel shading.

6.1 Limited Precision

Most of the per-pixel operations need only a single channel, and set R=G=B since this is the most efficient mode of operation. The register combiners can be implemented to a higher precision, but their input and output precision is limited to the framebuffer precision.

The register combiners currently support a conversion between 8-bit unsigned external values and 9-bit signed internal values. These conversions perform the function $f(x) = 2x - 1$ on an input, and $f^{-1}(x) = 0.5x + 0.5$ on the output, where x is each of the components of an RGBA pixel.

We could likewise create a packed luminance conversion to the input and output of the register combiners. The input mapping would perform the function L = R << 16 | G << 8 | B yielding a 24-bit luminance value on which one could perform scalar register combiner operations. Internally, the register combiner could maintain a 16.8 fixed-point format, and support operations such as addition, subtraction, multiplication and division using the extended range and precision of the new format. Once the operation is completed, the result may then be unpacked into the 8-bit framebuffer with the output mapping R = L >> 16, G = (L>>8)&0xff and B & 0xff.

6.2 Swizzle-Blits

Given the number of passes required, the register combiner performance was astounding, currently 1.3 Hz on a GeForce2 graphics accelerator at a resolution of 256x256. Profiling the code revealed that the main bottleneck was the time it took to save the framebuffer to a texture, adding an average of 2 ms per pass for 354 of the passes. OpenGL currently does not support rendering directly to texture, and the register combiner does not allow the framebuffer to be used as an input.

Whereas framebuffer memory is organized in scanline order, modern texture memory is organized into blocks and other patterns to better capitalize on spatial coherence. This coherence allows texture pixels to be more effectively cached during texture mapping operation. However, in this case the layout of texture memory is counterproductive. The cost to "swizzle" the memory into the clustered arrangement when saving a framebuffer image to texture memory dominates the execution time of iterative multipass shaders.

We have verifies this delay with a profile of the code, revealing that our CopyTexSubImage operations were taking longer than any other component of our shader. We also experimented with various resolutions and found a direct 1:1 correspondence between the number of pixels and the execution time.

Perhaps a mode can be incorporated into the graphics accelerator state that optionally defeats the spatial-coherent clustering of texture memory. This mode could be enabled during multipass shader evaluation, to eliminate the shuffled memory delay incurred during the CopyTexSubImage operations.

Alternatively, upcoming modes that support rendering directly to texture may also ameliorate this problem.

6.3 Logical Blend Modes

Blending modes such as exclusive-or and logical shifts left and right are extremely valuable when generating random values. Unfortunately these operations are not accelerated under current graphics drivers. Such operations are of the simplest to implement in hardware, and we suspect they will become accelerated as demand for them increases.

7. Conclusion

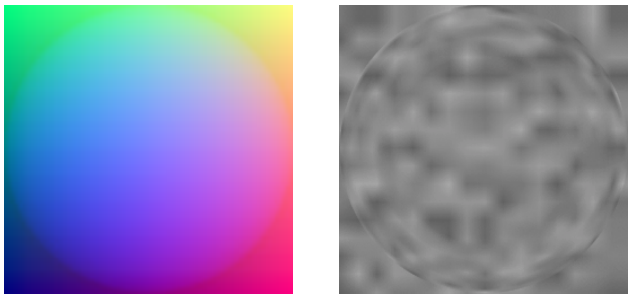
We have investigated the implementation of the Perlin noise function as a multipass pixel shader. We have developed a general algorithm and implemented it using the accelerated features from two different manufacturers.

The SGI implementation based on PixelTransfer and PixelMap operations remains faster than the NVidia implementation based on register combiners. However, we expect the additional register combiner stages available in the upcoming GeForce3 will close this gap.

The process of implementing a general-purpose procedure using GPU accelerated operations has been illuminating. We are excited by the prospect of using the GPU as a SIMD-based supercomputer. However, this vision has been stifled by the low precision available in the buffers and processors, and the latency due to slow framebuffer-to-texture memory transfers. We believe both problems can be solved with moderate changes to existing graphics accelerator architectures, and have suggested possible solution implementations.

Our noise implementation uses linear interpolation of random values on an integer lattice. One can also implement cubic interpolation at the expense of four extra passes. The function SCURVE(u) = $3u^2 - 2u^3$ can also be expressed as $uu(3-2u)$. The function 1/4 SCURVE(u) can be implement by modulating the images u, u and 3/4 – 1/2 u. Note the latter is necessarily scaled by ¼ to fall within the legal [0,1] OpenGL range. This result can then be scaled by 4 (either through PixelTransfer or a register combiner) to yield SCURVE(u).

We have investigated numerous methods for enhancing the performance of these multipass pixel shaders. The 2-D s-t plane examples suggested that image processing applications such as translation and convolution could be applied, but such techniques would not work for arbitrarily shaped objects in the solid texture coordinate image, such as in Figure 13.



(a) (b)

Figure 13. Application of the noise function (b) on a sphere of solid texture coordinates (a).

The source code and an executable for both implementations of the Perlin noise pixel shader can be found at:

http://graphics.cs.uiuc.edu/~jch/mpnoise.zip

Acknowledgments

Conversations with Pat Hanrahan and Henry Moreton were helpful in determining the cause of the 3ms CopyTexSubImage delay. This research was supported in part by a grant from the Evans & Sutherland Computer Corp. Thanks also to Nate Carr for proofreading the paper.

References

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

[2] Carr, N.A. and J.C. Hart. Real-Time Procedural Solid Texturing. Manuscript, in review. Apr. 2001.

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

[4] 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

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

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

[7] 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.

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

[9] Heidrich, W., R. Westermann, H-P Seidel and T. Ertl. Applications of Pixel Textures in Visualization and Realistic Image Synthesis. Proc. ACM Sym. on Interactive 3D Graphics, Apr. 1999, pp. 127-134.

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

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

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

[13] Mine, A. and F. Neyret. Perlin Textures in Real Time using OpenGL. Research Report #3713, INRIA, 1999. http://www-imagis.imag.fr/Membres/Fabrice.Neyret/publis/RR-3713-eng.html

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

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

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

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

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

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

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

[21] Proudfoot, K., W.R. Mark, S. Tzvetkov and P. Hanrahan. A real-time programmable shading system for programmable graphics hardware. Proc. SIGGRAPH 2001, Aug. 2001, to appear.

[22] 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.

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

[24] Skinner, R. and C.E. Kolb. noise.c component of the Rayshade ray tracer, 1991.