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.

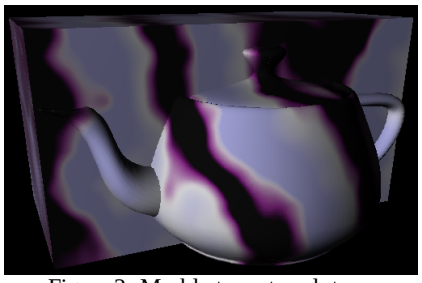


Figure 2: Marble teapot sculpture.

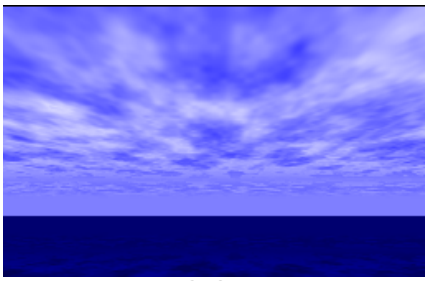


Figure 3: Seascape.

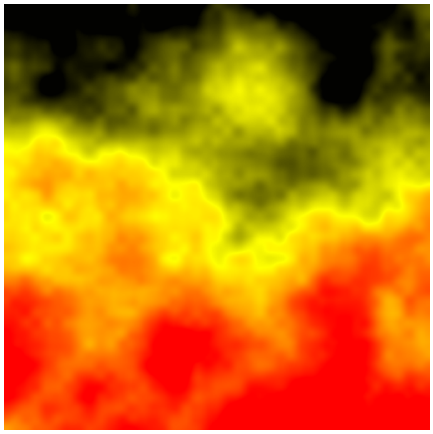


Figure 4: Fire.

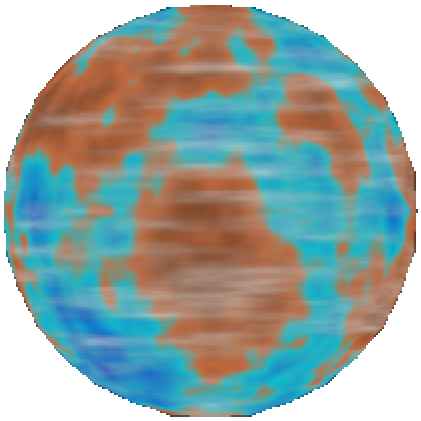


Figure 5: Planet.

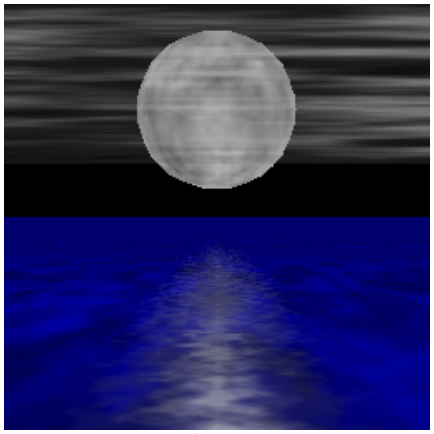


Figure 6: Moonrise.

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

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

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

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

which can more conveniently be represented homogeneously as

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

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

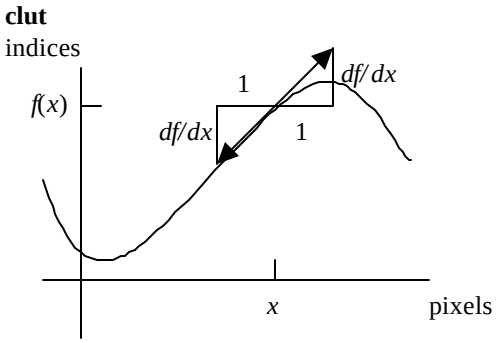


Figure 7: The derivative df/dx approximates the extent of the color map indices one pixel in either direction. Half of the derivative estimates the variation in color map indices half of a pixel in either direction.

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.

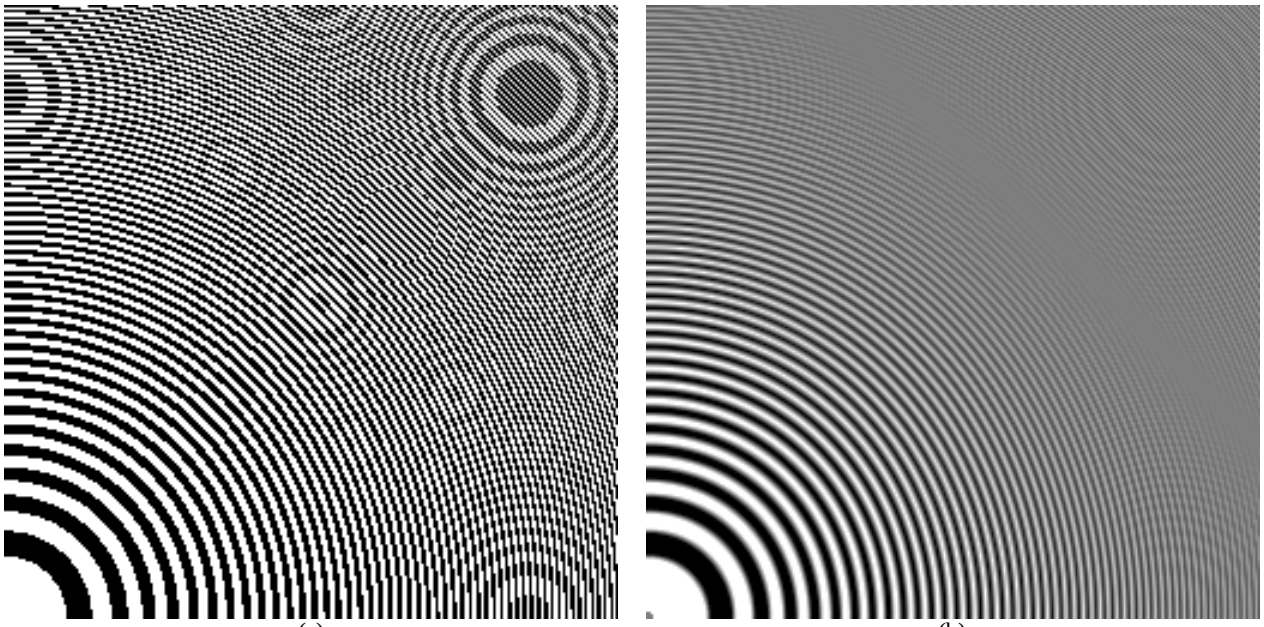
Instead, we analyze the function $\mathbf{p}(\mathbf{x})$ that textures pixels to determine the width of a box filter that would eliminate the aliasing frequencies from the spectrum of the synthesized texture. Several have described techniques for antialiasing procedural textures by antialiasing the 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

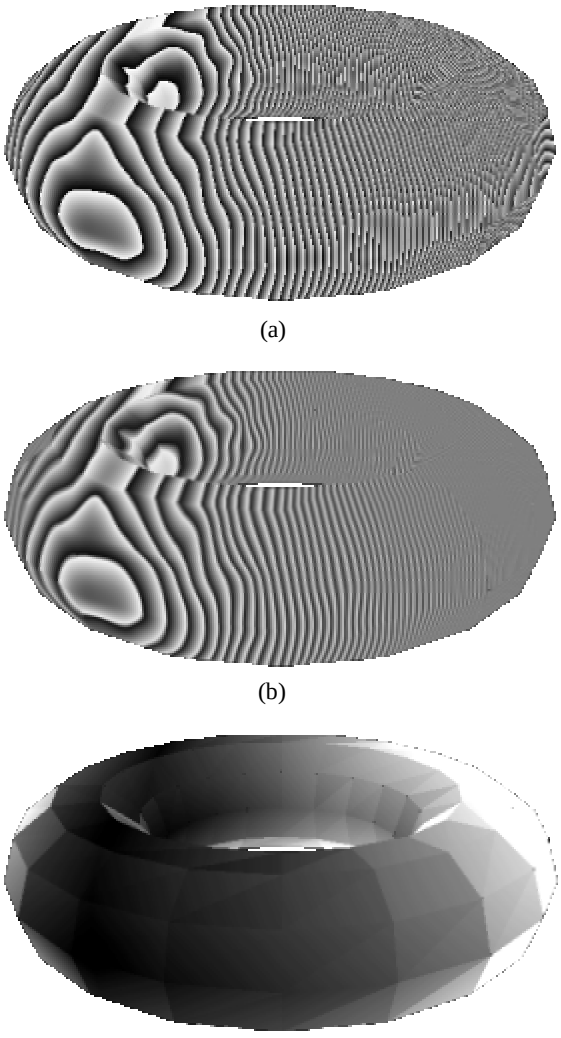
The filtering of color map values can be evaluated efficiently using either a color table MIP map or a summed area color table.

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 **csat**[i] = **csat**[i-1] + **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: **csat**[N-1]/N.
- $f + w/2 \geq N$:
  mod: (**csat**[$f + w/2$ – N] + **csat**[N-1] – **csat**[$f – w/2$ - 1])/w.
  clamp: ((f+w/2–(N-1))**clut**[N-1] + **csat**[N-1] – **csat**[f–w/2-1])/w.
- $f - w/2 < 0$:
  mod: (**csat**[$f + w/2$] + **csat**[N-1] - **csat**[$N + f - w/2$ - 1])/w.
  clamp: (-($f - w/2$) **clut**[0] + **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

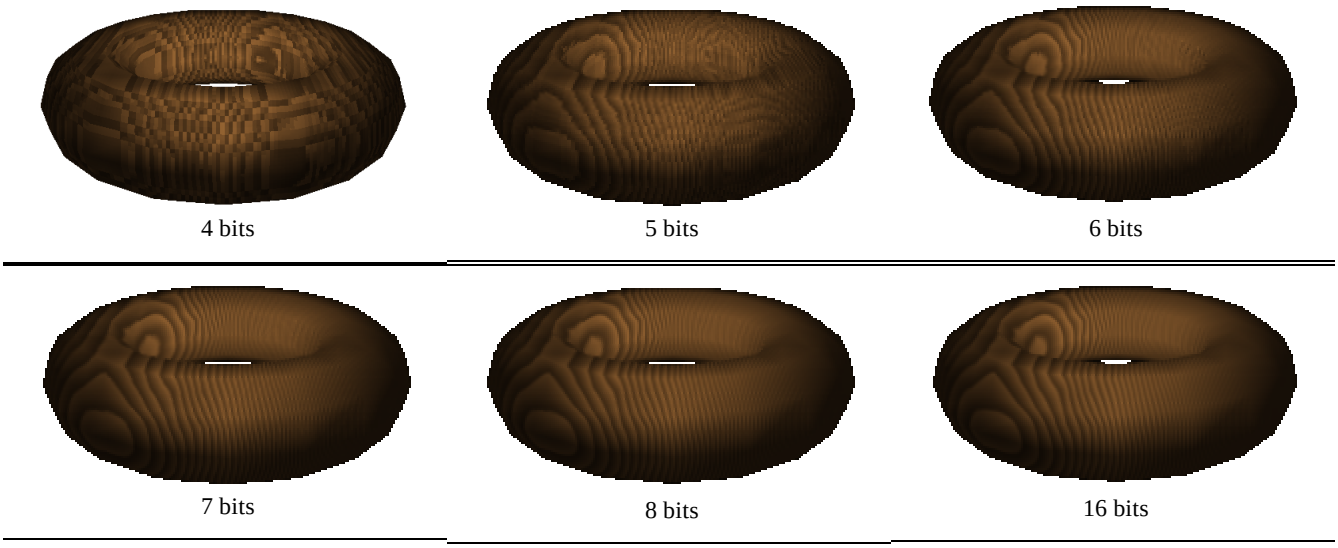


Figure 10: The effect of numerical precision on texture appearance.

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.

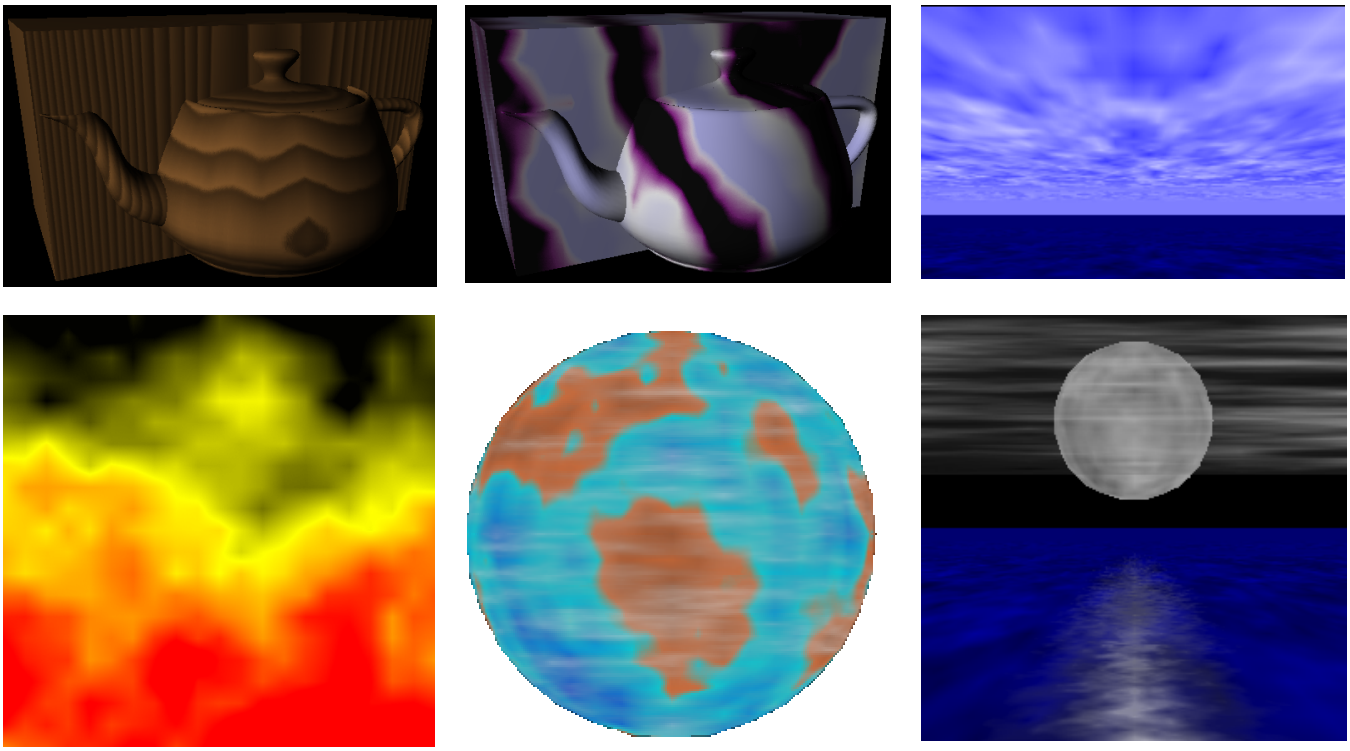


Figure 11: 100,000 gate simulations of Figure 1 through Figure 6.

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.