A Framework for Analyzing Real-Time Advanced Shading Techniques

John C. Hart[$]
Washington State University
hart@eecs.wsu.edu

Peter K. Doenges
Evans & Sutherland Computer Corp.
pdoenges@es.com

Abstract

Numerous techniques have been developed to perform advanced shading operations in real time. The real-time versions vary greatly from their original implementations due to the constraints of existing hardware and programming libraries. This paper introduces a grammar for articulating these modern real-time shading pipelines. We survey the existing techniques, using the grammar to classifies them into a taxonomy illustrating the commutations that differentiate the pipelines from the classical version and each other. The taxonomy follows a natural development that concludes with texture shading, which is applied to four advanced shaders to explore its versatility.

1 Introduction

The task of presenting a three-dimensional object on a two-dimensional display relies largely on perceptual cues the brain has evolved for resolving the three-dimensional spatial configuration of a scene from its projection onto the eye's two-dimensional retina. One of these cues is shading: the variation in color and intensity of a surface that indicates its position and orientation within a scene.

Consumer computer graphics has finally outgrown the classic lighting model composed of Lambertian diffuse and Phong/Blinn specular reflection that dominated hardware implementation for the past two decades. [Cook & Torrance, 1982] noted that the standard technique of matching the diffuse component with the surface material color and the specular component with the color of the light source was a good model for plastic, which consists of a suspension of diffusing pigments covered by a thin clear specular membrane. With the support of advanced shaders, consumer computer graphics will finally overcome its cliché plastic appearance.

Procedural shaders generate the color of a point on the surface by running a short program. The Renderman system, which contains a little language specifically developed for procedural shaders, is the current industry standard for procedural shading. Hanrahan suggests that Renderman, while adequate for production-quality shading, may not be the most appropriate choice for specifying real-time shaders [Hanrahan, 1999].

[$]A consultant for the Evans & Sutherland Computer Corp.

Shaders determine the color of light exiting a surface as a function of the light entering a surface and the properties of the surface. Shaders combine the cues of *lighting*, which determines how light uniformly interacts with the surface, and *texturing*, which determines how light nonuniformly interacts with the surface. We use the stationarity of the phenomena as a key differentiator between surface lighting and surface texturing. Hence, "texturing" is a feature parameterized in part by position whereas "lighting" is not.

Section 3 surveys current real-time advanced shading strategies. Finding information on these topics is not easy. The techniques have resulted as much from developers as from researchers, and these techniques appear in tutorials, how-to's, product specifications and reports more often than in journals and conference proceedings. This survey collects these ideas together in one place and presents the techniques in a uniform and organized manner to allow better comparison and understanding of their benefits and drawbacks.

This paper is in part a response to the keynote talk of the Eurographics/SIGGRAPH 1999 Graphics Hardware Workshop [Hanrahan, 1999]. This talk lamented the fact that there are many directions hardware vendors are considering to support advanced shading. This situation was best described by the slide in Figure 1. Section 3 develops a natural progression from the standard graphics pipeline through fragment lighting, multitexturing and multipass eventually concluding with texture shading.

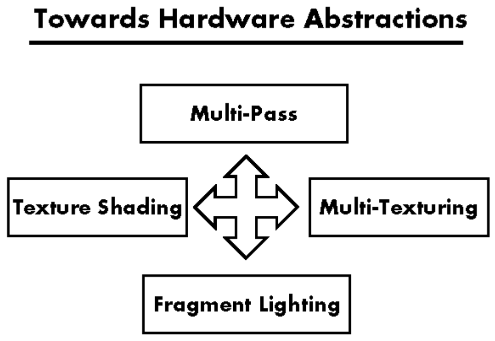

Figure 1. Slide 22 of [Hanrahan, 1999]

Hanrahan recommended that graphics hardware community should investigate solutions to this problem by "commuting the graphics pipeline." The grammar introduced in Section 2 provides a representation where such commutations can be articulated. analyzed and classified. Such grammars are already familiar in the analysis of rendering. A grammar associated with the rendering equation [Kajiya, 1985] has been used to classify the transport of light from its source to the viewer as a sequence of symbols corresponding to emittance, occlusion, and specular and diffuse surface reflections.

This paper also follows up on the ideas of [McCool & Heidrich, 1999], which proposed a texture shader built on a multitexturing stack machine and a wider variety of texture coordinate generation modes. Section 4 begins to look in detail at what kinds of shaders and variations are possible using these advanced programmable shader techniques.

2 A Graphics Pipeline Grammar

This section develops a grammatical representation for the graphics shading pipeline. The symbols used in this grammar are listed in Figure 2.

$\mathbf{x}$	vertex in model coordinates
$\mathbf{u}$	surface parameterization (u,v)
$\mathbf{s}$	shading parameter vector $(N,V,R,H,\ldots)$
π	graphics pipeline from model to viewport coordinates
$\mathbf{x}_s$	pixel in viewport coordinates (x_s,y_s)
δ	rasterization (interpolation and sampling)
$\mathbf{c}$	color vector (R,G,B)
$\oplus$	color combination function
C	frame buffer
T	texture map
$\leftarrow$	assignment

Figure 2: Operators in the shading pipeline grammar.

We denote a two-dimensional surface parameterization as $\mathbf{u} = (u,v)$. We denote the shading parameters as a vector $\mathbf{s}$ that contains light source and surface material constants, as well as the local coordinate frame and the view vector. We use the vector $\mathbf{x}$ to represent a triangle vertex with its position in model coordinates, and $\mathbf{x}_s$ to denote the same point in viewport (screen) coordinates. The 2-D surface texture coordinates are an attribute of the vertex and are denoted $\mathbf{u}\,\mathbf{x}$. Likewise, the shading parameter vector is also a vertex attribute and is denoted $\mathbf{s}\,\mathbf{x}$. Note that since these functions accept a single parameter, we eliminate the use of paranthesis in favor of a grammatical expression.

We denote color $\mathbf{c} = (R,G,B)$. The map $\mathbf{p}$: $\mathbf{s} \rightarrow \mathbf{c}$ denotes a shader, a procedure that maps shading parameters to a color $\mathbf{c}$. The operator T: $\mathbf{u} \rightarrow \mathbf{c}$ is a 2-D texture map that returns a color $\mathbf{c}$ given surface texture coordinates $\mathbf{u}$.

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

Most of the standard graphics pipeline can be decomposed into a general projection π that maps vertices from 3-D model coordinates $\mathbf{x}$ to 2-D screen coordinates $\mathbf{x}_s$, and a rasterization that takes these screen coordinate vertices and fills in the pixels of the polygon they describe using linear interpolation. It will be useful for the analysis of the aliasing artifacts to know exactly when attributes are interpolated across a polygon, as this signals when continuous functions are discretely sampled. We will indicate that a continuous function has been discretely sampled by rasterization with a delta function operator δ.

Hence, $\mathbf{x}$ is a polygon vertex in model-coordinates, $\pi\,\mathbf{x}$ is the screen coordinate corresponding to that point and $\delta\,\pi\,\mathbf{x}$ reminds us that the coordinates of that pixel were discretely interpolated from the screen coordinates of the polygon's vertices. The goal of the next section will be to articulate and analyze various techniques for assigning a value to the screen pixel $C\,\delta\,\pi\,\mathbf{x}$.

3 Procedural Shading Techniques

This section analyzes various graphics shading pipelines, including the standard pipeline, deferred shading, multipass, multitexturing, environment map techniques and texture shading. It also makes explicit shading techniques supported by these pipelines, including Phong mapping and environment mapped bump mapping.

3.1 Standard Graphics Pipeline Shading

The standard implementation of the modern graphics pipeline is dominated by the linear interpolation of vertex attributes at the end of the pipeline.

Gouraud Interpolation. The standard graphics pipeline implementation of lighting is expressed in this notation as

$$\mathrm{C}\,\delta\,\pi\,\mathbf{x} \leftarrow \delta\,\mathbf{p}\,\mathbf{s}\,\mathbf{x}. \tag{1}$$

Lighting is computed per-vertex, and the resulting color is interpolated (using a technique known as Gouraud shading) across the pixels of the screen-space polygon by the rasterization engine.

Texture Mapping. Texturing is performed in screen coordinates and texture coordinates are assigned per-vertex and interpolated across the pixels of the screen-space polygon by the rasterization engine. Interpolated texture coordinates are then index into a texture map to determine a per-pixel texture color

$$\mathrm{C}\,\delta\,\pi\,\mathbf{x} \leftarrow \mathrm{T}\,\delta\,\mathbf{u}\,\mathbf{x}. \tag{2}$$

The aliasing artifacts introduced by texture mapping occur when the sampling rate of the delta function on the LHS of (2) (the resolution of the polygon's screen projection) disagrees with the sampling rate of the delta function on the RHS (the texture's resolution). Methods for resampling the texture map based on the MIP map [Williams, 1983] or the summed-area table [Crow, 1984] fix this problem by adjusting the sampling density on the RHS of (2) to match that of the LHS.

An additional though subtle issue with the δ function on the RHS of (2) is perspective correction. Since the projection π on the LHS performs a perspective divide, then the δ rasterization function on the RHS must also perform a per-pixel perspective divide.

Modulation. The results of lighting and texture mapping are combined using a modulation operator

$$C\ \delta\ \pi\ \mathbf{x} \leftarrow (\delta\ \mathbf{p}\ \mathbf{s}\ \mathbf{x}) \oplus (T\ \delta\ \mathbf{u}\ \mathbf{x}). \qquad (3)$$

In other words, the color of each pixel in the polygon's projection $\pi(\mathbf{x})$ is given by a blending operation of the pixel in the texture map *T* and the interpolated shading of the polygon's vertices.

3.2 Fragment Lighting

Fragment lighting is perhaps the most obvious way to implement lighting. It simply extends the per-vertex lighting currently present in graphics libraries to per-pixel computation. Fragment lighting thus computes the shading of each pixel as it is rasterized

$$C\ \delta\ \pi\ \mathbf{x} \leftarrow \mathbf{p}\ \delta\ \mathbf{s}\ \mathbf{x}. \qquad (4)$$

The shader parameters stored at each vertex are interpolated across the pixels of the polygon's projection, and a procedure is called for each pixel to synthesize the color of that pixel. Methods for Phong shading in hardware [Bishop & Weimer, 1986],[Peercy *et al.*, 1997] are based on fragment lighting, as are a variety of procedural shaders, both production [Hanrahan & Lawson, 1990] and realtime [Hart *et al.*, 1999].

Note that fragment lighting (4), which supports Phong shading interpolation, is a permutation of (1), which supports Gouraud shading interpolation. The juxtaposition of sampling δ and shader evaluation **p** suffices to change the representation from interpolating shader results (color) with shader parameters (e.g. surface normals).

Fragment lighting applies the entire procedure to each pixel before moving to the next. The main drawbacks to this technique is that interpolated vectors, such as the surface normal, need to be renormalized, which requires an expensive per-pixel square root operation. If this renormalization is approximated or ignored, the resulting artifacts can be incorrect shading, and this error increases with the curvature of the surface the polygon approximates.

The second drawback is the amount of per-pixel computation versus the amount of per-pixel time. Assuming a megapixel display and a 500 MIPS computer sustaining a 10 Hz frame rate limits procedures to 50 instructions. While a high-level VLIW instruction set could implement most shaders in 50 instructions, this would be a wasteful investment of resources since most shaders remain static, and the shader processor continue to repeatedly synthesize the same texture albeit for different pixels as the object moves across the screen.

The sampling rate of the δ in the LHS of (4) (the resolution of the polygon's projection) matches the sampling rate of the RHS (the resolution of the polygon sampling the shader). Hence aliasing occurs when this rate insufficiently samples the shader **p**. With the exception of the obvious and expensive supersampling technique, procedural shaders can be antialiased by bandlimiting [Norton, *et al.*, 1982] and a gradient magnitude technique [Rhoades, *et al.*, 1992], [Hart *et al.*, 1999], both which modify the texture procedure **p** to only generate signals properly sampled by the coordinates discretized by the δ.

3.3 Deferred Shading

Deferred shading [Molnar, 1992] implements procedural shading in two phases. In the first phase

$$T\ \delta\ \pi\ \mathbf{x} \leftarrow \delta\ \mathbf{s}\ \mathbf{x} \qquad (5)$$

such that the shading parameters are stored in a fat texture map *T* which is the same resolution as the display, called the fat framebuffer. Once all of the polygons have been scan converted, the second phase makes a single shading pass through every pixel in the frame buffer

$$C\ \mathbf{x}_s \leftarrow \mathbf{p}\ T\ \mathbf{x}_s \qquad (6)$$

replacing the color with the results of the procedure applied to the stored solid texture coordinates. In this matter, the application of **p**, the shader, is deferred until all of the polygons have been projected, such that the shader is only applied to visible sections of the polygons.

Deferred shading applies all of the operations of the shader expression to a pixel before the next pixel is visited, and so suffers the same process limitations as fragment lighting. Unlike fragment lighting, deferred shading has a fixed number of pixels to visit, which provides a constant execution speed regardless of scene complexity.

In fact, the main benefit of deferred shading is the reduction of its shading depth complexity to one. This means a shader is evaluated only once for each pixel, regardless of the number of overlapping objects that project to that pixel. Since some shaders can be quite complex, only applying them to visible pixel can save a significant amount of rendering time.

The main drawbacks for deferred shading is the size of the fat framebuffer *T*. The fat framebuffer contains every variable used by the shader, including surface normals, reflection vectors, and the location/direction and color of each light source.

One possible offset to the large frame buffer is to generate the frame in smaller chunks, trading space for time. It is not yet clear whether the time savings due to deferred shading's unit depth complexity makes up for the multiple renderings necessary for this kind of implementation.

Antialiasing is another drawback of deferred shading since the procedural texture is generated in a separate step of the algorithm than the step where the samples have been recorded from the δ. Deferred shading thus precludes the use of efficient polygon rasterization antialiasing methods such as coverage masks. Unless a significant amount of auxiliary information is also recorded, previous procedural texturing antialiasing algorithms do not easily apply to deferred shading.

However, with the multi-sample antialiasing found in many recent graphics controllers, supersampling appears to be the antialiasing technique of choice, and is certainly the most directly and generally scalable antialiasing solution across all segments of the graphics market. While the deferred shading frame buffer would have to be as large as the sampling space, this still seems to be a feasible and attractive direction for further pursuit.

Since all of the information needed by shader is held in the fat frame buffer per pixel, the channels of the framebuffer would need to be signed and generally of higher precision than the resulting color to prevent error accumulation in complex shaders.

3.4 Environment Map Lighting

There are a variety of texture coordinate modes that implement useful features. Recall that an environment map is an image of the incoming luminance of a point in space.

Spheremap. Environment mapping is most commonly supported in hardware by the spheremap mode. This texture coordinate generation mode assigns a special projection of the reflection vector *R* component of the shading information **s** to the texture coordinate **u** of the vertex **x**

$$\mathbf{u}\,\mathbf{x} \leftarrow \pi^{R}\, R\, \mathbf{s}\, \mathbf{x}. \qquad (7)$$

This projection requires a per-vertex square root that is handled during texture coordinate generation. The texture map consists of a mirror-ball image of the surrounding environment, which is combined with standard lighting by (3). This notation reveals how environment mapping avoids the interpolation of normalized vectors, by instead interpolating and sampling the projection of the reflection vector as a texture coordinate. This inexact approximation can create artifacts on large polygons spanning high-curvature areas.

Phong/Gloss Mapping. One problem with vertex lighting is that since vertex colors are interpolated, specular highlights that peak inside polygon faces do not get properly sampled. Specular highlights can be simulated using (13) to generate an environment spheremap consisting of a black sphere with specular highlights. This allows current graphics API's to support Phong highlights on polygon faces, using (7) for texture coordinate generation and (3) for modulating texturing with lighting. These highlights could be considered the incoming light from a diffused area light source, thereby softening the appearance of real-time shaded surfaces.

One significant advantage of the environment map is that it can contain the incoming light information from any number of light sources. Without it, we have a shading complexity that is linear in the number of light sources, often requiring a separate pass/texture for each light source. Putting all of the light source information into an environment map reduces the complexity to constant in the number of light sources.

Environment Mapped Bump Mapping. Phong mapping can also be used to approximate bump mapping

$$C\,\delta\,\pi\,\mathbf{x} \leftarrow T\,\delta\,((\mathbf{u}\,\mathbf{x}) + (T'\,\delta\,\mathbf{u}'\,\mathbf{x})) \qquad (8)$$

where T' is a bump map texture that, instead of a color, returns a 2-D vector (the partial derivatives of the bump map height field) that offsets the index into the environment map. Comparing EMBM (8) to standard texture mapping (2) shows precisely where the perturbation occurs. This form of bump mapping is supported by Direct3D 6.0, but requires hardware support for the offsetting of texture indices by a texture result. It would be interesting to investigate what other effects are possible by offsetting a texture index with the result of another texture.

3.5 Multipass Rendering

Modern graphics API's have limited resources that are often exhausted implementing a single effect. Multipass algorithms render the scene several times, combining a new effect with the existing result. Each pass can include an additional shader element, including lighting effects and texture layers. Each pass follows the form

$$C\,\delta\,\pi\,\mathbf{x} \leftarrow (C\,\delta\,\pi\,\mathbf{x}) \oplus ((\delta\,\mathbf{p}\,\mathbf{s}^{*}\,\mathbf{x}) \oplus (T^{*}\,\delta\,\mathbf{u}^{*}\,\mathbf{x})) \qquad (9)$$

where the asterisk indicates operators that typically change for each pass. The image composition operators of OpenGL 1.2.1 support a variety of frame combination operators.

Multipass is a SIMD approach that distributes procedural operations across the screen, applying a single operation to every screen pixel before moving to the next operation in the shading expression.

The benefit of multipass rendering is its flexibity. Any number of passes and combinations can be supported, and can be used to support full-featured shading languages [Proudfoot, 1999], [Olano, *et al.*, 2000].

The drawback of multipass rendering is its execution time. Each pass typically requires the re-rasterization of all of the geometry. Furthermore, storage and combination of frame buffer images can be incredibly slow. In many OpenGL implementations, it is faster to display a one-polygon-per-pixel mesh texture mapped with an image than to try to write the image directly to the screen.

Multipass rendering benefits from a retained (scene graph) mode since the input object data rarely changes from pass to pass. If multipass is used from a static viewpoint, then the polygons need only be rasterized once, and each pass can be performed on screen space vertices (π **x**) instead of model space vertices **x**, specifically

$$C\,\delta\,\mathbf{x}_{s} \leftarrow (C\,\delta\,\mathbf{x}_{s}) \oplus ((\delta\,\mathbf{p}\,\mathbf{s}^{*}\,\mathbf{x}) \oplus (T^{*}\,\delta\,\mathbf{u}^{*}\,\mathbf{x})) \qquad (10)$$

Such a system would combine the benefits of deferred shading (compare (5) and (6)) with multipass since the shading would be deferred until after polygon projection. The shader would be executed only on the polygons that survived clipping, but this includes more than just the visible polygons.

As each new frame is composed with a previous frame, a low-pass filter should remove high frequencies from the new frame before composition. It is not yet clear what interference patterns could be created when composing two images with energy at nearly the same frequency. Some situations could cause a beating pattern at a lower, more noticeable frequency.

As the results of the shading expression are accumulated in the frame buffer, the precision of the frame buffer needs to be increased beyond the final color output precision. To best accomodate a variety of individual shading operations, the intermediate frame buffers need to support signed values.

Accumulation Buffer. Perhaps the most obvious multipass technique is the accumulation buffer

$$C\ \delta\ \pi^*\ \mathbf{x} \leftarrow (C\ \delta\ \pi^*\ \mathbf{x}) \oplus (\delta\ \mathbf{p}\ \mathbf{s}\ x) \qquad (11)$$

where π^* indicates that the projection is perturbed. This perturbation supports antialiasing, motion blur, depth of field and, when combined with a shadowing technique, soft shadows.

Shadow Mask. The multipass method for rendering shadows via the shadow mask [Williams, 1978] is given by the following steps

$$C\ \delta\ \pi\ \mathbf{x} \leftarrow \delta\ \mathbf{p}\ \mathbf{s}\ \mathbf{x},$$

$$C'\ \delta\ \pi\ \mathbf{x} \leftarrow \delta\ \mathbf{p}\ \mathbf{s}'\ \mathbf{x},$$

$$C^l\ \delta\ \pi^l\ \mathbf{x} \leftarrow \delta\ \mathbf{x},$$

$$\alpha\ C\ \mathbf{x}_s \leftarrow (z\ C\ \mathbf{x}_s) > (z\ C^l\ \pi^l\ \pi^{-1}\ \mathbf{x}_s),$$

$$C\ \mathbf{x}_s \leftarrow (\alpha\ C\ \mathbf{x}_s)*(C\ \mathbf{x}_s) + (1 - \alpha\ C\ \mathbf{x}_s)*(C'\ \mathbf{x}_s).$$

where **s** contains ambient lighting parameters and **s**' contains diffuse and specular. The superscript *l* indicates a frame buffer C^l and projection π^l for the light source. The expression $\alpha\ C\ \mathbf{x}_s$ returns the alpha channel of the frame buffer C at position $\mathbf{x}_s$, and likewise the z operator returns the depth channel. Analyzing the shadow mask algorithm in this notation reveals several opportunities for special hardware to support parallel operation and pass combination.

Shadow Volumes. Multipass techniques usually rely heavily on the stencil buffer to either restrict the shader's operations to a section of the screen, or to store a temporary result of a shading operation. For example, the shadow volume method can be expressed

$$C\ \delta\ \pi\ \mathbf{x} \leftarrow \delta\ \mathbf{p}\ \mathbf{s}\ \mathbf{x},$$

$$s\ C\ \delta\ \pi\ \mathbf{x} \leftarrow (s\ C\ \delta\ \pi\ \mathbf{x})\ \mathrm{OR}\ ((z\ \delta\ \pi\ \mathbf{x}') > (z\ C\ \delta\ \pi\ \mathbf{x}')),$$

$$C\ \delta\ \pi\ \mathbf{x} \leftarrow (s\ C\ \delta\ \pi\ \mathbf{x})\ ?\ (\delta\ \mathbf{p}\ \mathbf{s}'\ \mathbf{x}).$$

In this example, the object vertices are denoted **x** and the shadow volume vertices are **x**'. The operator s(C) returns the stencil buffer value from the frame buffer C. The vector **s** contains ambient shading parameters whereas **s**' contains diffuse and specular parameters.

3.6 Multitexturing

Multitexturing allows different textures to be combined on a surface. Multitexturing is a SIMD approach that distributes procedural operations across data, performing a single operation on the entire texture before moving to the next operation.

OpenGL 1.2.1 supports chained multitexturing

$$C\delta\pi\mathbf{x} \leftarrow T'''\delta\mathbf{u}'''\mathbf{x} \oplus (T''\delta\mathbf{u}''\mathbf{x} \oplus (T'\delta\mathbf{u}'\mathbf{x} \oplus T\delta\mathbf{u}\mathbf{x})). \qquad (12)$$

where the $\oplus$ symbol denotes one of the OpenGL texture modes, either decal, modulate or blend. Direct3D appears to be extending these modes to allow a larger variety of texture expressions.

Multitexturing avoids the antialiasing roadblocks encountered by deferred shading because multitexturing defers the shading to the texture map, then projects the result onto the screen. This sets up the opportunity for shading aliases, which are more tolerable, without affecting rasterization aliases, which are more distracting.

Antialiasing in a multitexturing system could be accomplished by antialiasing each of the component textures. MIP mapping of multitexture components is one method used to filter the texture components.

Since the textures are used as components to shading equations, higher precision texture maps are needed to accumulate intermediate results, especially if scales greater than one are allowed. Signed texture values are also necessary.

3.7 Texture Shading

Texture shading stores shading information in the texture coordinates and maps. In its simplest form, it is expressed as

$$T\ \delta\ \mathbf{u} \leftarrow \mathbf{p}\ \mathbf{s}\ \delta\ \mathbf{u} \qquad (13)$$

where the texture coordinate vector **u** indexes local illumination and texturing information **s**, and **p** applies a shader to this information, storing the resulting color in the texture map. The texture map is then applied to the surface using (2), which now takes responsibility for both texturing and lighting [Kautz & McCool, 1999] . Such techniques require special texture generation modes such that the texture coordinates contain a portion of the shader expression. These methods are demonstrated in Section 4.

Fat Texture Map. Texture shading occurs on a surface, which is parameterized by a two-dimensional coordinate system. A fat texture map could be considered that stores a vector of shading parameter instead of simply colors

$$C\ \delta\ \pi\ \mathbf{x} \leftarrow \mathbf{p}\ T\ \delta\ \mathbf{u}\ \mathbf{x}. \qquad (14)$$

The parameters stored in the fat texture map might include vectors such as surface normals and tangents, or cosines such as the dot product of the surface normal and the light direction. This model of texture shading is similar to deferred shading, replacing the fat frame buffer with a fat texture map.

Incorporating texture shading into multitexturing replaces the fat texture map with a collection of standard-sized texture maps each containing a sub-expression result of a complex shader expression. McCool proposed a multitexturing algebra based on a stack machine, allowing more complex texture expressions. McCool's proposal for dot products overlooks the sines of the angles between vectors, which could be useful for rendering hair.

It is interesting that the linear interpolation across the polygon interpolates the indices across the parameter vectors stored in texture memory. This allows the interpolation of normals and other shading parameters to be precomputed, such that only the index **u** need be interpolated [Kilgard, 1999].

Solid Mapping. Texture shading was used to perform solid texturing in OpenGL without any extensions [Carr, et al., 2000]. The technique assumed that the mapping **u**: **x**→**u** is one-to-one (such that images of the object's polygons do not overlap in the texture map *T*). The object's polygons are rasterized into the texture map

$$\mathrm{T}\,\delta\,\mathbf{u}\,\mathbf{x} \leftarrow \delta\,\mathbf{s}\,\mathrm{x}, \qquad (15)$$

where the shading parameters, in this case the solid texture coordinates (*s*,*t*,*r*), are stored as a color {*R* = *s* ; *G* = *t* ; *B* = *r*} in the texture map *T*. A second pass

$$\mathrm{T}\,\delta\,\mathbf{u} \leftarrow \mathbf{p}\,\mathrm{T}\,\delta\,\mathbf{u} \qquad (16)$$

replaces the texture map contents (s,t,r) with a color (*R*,*G*,*B*) generated by the procedural shader **p** on the solid texture coordinates. The texture map now contains a procedural solid texture that can be mapped back onto the object using standard texture mapping (2).

4 Applications

In the previous section, we followed a natural progression of techniques to support the real-time implementation of advanced shading models. This progression concluded with texture shading, which, when supported by multitexturing and multipass rendering, provides a powerful tool for implementing advanced shaders, though the full power of this tool is not yet completely understood. We explore the capabilities of texture shading by considering the implementation of a variety of advanced shaders.

These advanced shaders require more information than the standard surface normal and reflection vector currently available. This information can be encoded as dot products, as recommended by [McCool & Heidrich, 1999]. The coordinates and vectors used by these shaders are enumerated in Figure 3.

u	the point on the surface whose illumination properties we are interested in;
N	the unit *surface normal* perpendicular to the tangent plane of the surface at **u**;
T	principal *tangent vector* used to fix the orientation of the coordinate frame at u for anisotropic shading;
L	a light-dependent unit *light vector* anchored at u in the direction of one of possibly many light sources;
V	the view-dependent unit *view vector* anchored at u in the direction of the viewer;
R	the light-dependent unit *light reflection vector* equal to $2(N \cdot L)N - L$;
H	the light- and view-dependent unit *halfway vector* equal to *L*+*V* normalized (constant for orthographic projection and directional light sources);

Figure 3: Shading parameters.

One method for implementing advanced shaders is to precompute its results for all possible inputs. We consider the equivalence classes of the reflectance function of a surface $r(u,v,q_i,f_i,q_r,f_r)$ where *u*,*v* denotes a point on the surface, q_i, f_i are the elevation and azimuth of a light vector *L* on this point, and q_r, f_r are the elevation and azimuth of the viewing direction *V*. (We use the term BRDF although many shaders are not actually bidirectional [Lewis, 1994]). We will denote equivalence classes by replacing parameters of the plenoptic function with the symbol ·, as shown in Figure 4.

$r\,(\cdot,\cdot,q_i,f_i,q_r,f_r)$	BRDF
$r\,(\cdot,\cdot,q_i,\cdot,\cdot,\cdot)$	Diffuse, e.g. Lambert's law
$r\,(\cdot,\cdot,q_i,\cdot,q_r,\cdot)$	Isotropic, e.g. *N·L*, *N·V*
$r\,(\cdot,\cdot,q_i,f_i+\cdot,q_r,f_r+\cdot)$	Specular, e.g. *N·L*, *N·V*, *V·R*
$r\,(\cdot,\cdot,q_i,f_i,\cdot,\cdot)$	Anisotropic diffuse, e.g. *N·L*, *T·L*
$r\,(u,v,\cdot,\cdot,\cdot,\cdot)$	Texturing
$r\,(u,v,q_i,\cdot,\cdot,\cdot)$	Diffuse bump mapping
$r\,(u,v,q_i,f_i+\cdot,q_r,f_r+\cdot)$	Specular bump mapping

Figure 4: Equivalence classes of reflectance functions.

We investigate the various advanced texturing and shading techniques within these equivalence classes and use the classes to determine if precomputation and storage is feasible within the implementation technique.

[Cabral *et al.*, 1999] showed how a general BRDF could be applied through the environment spheremap by assigning to it the reflected luminance instead of the incident luminance. While a technique for interpolating these luminance maps was described, this technique relies on a large number of environment maps discretized over the possible view positions.

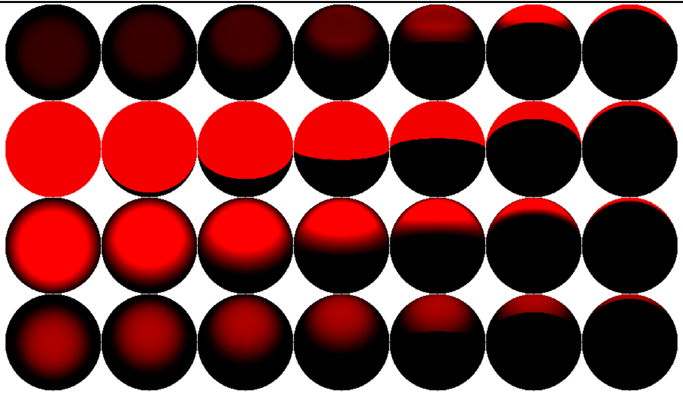
Figure 5: Cook-Torrance.

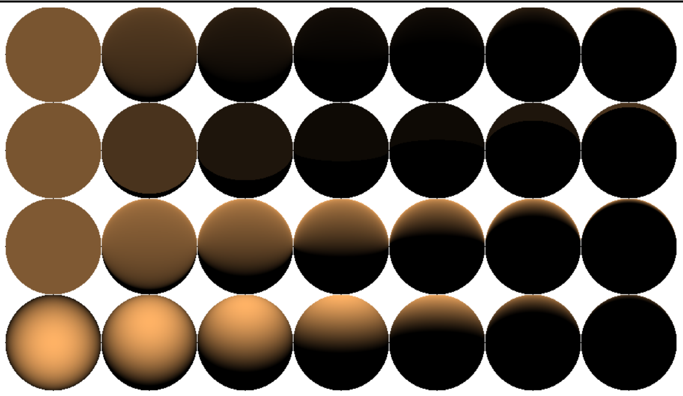
Figure 6: Skin.

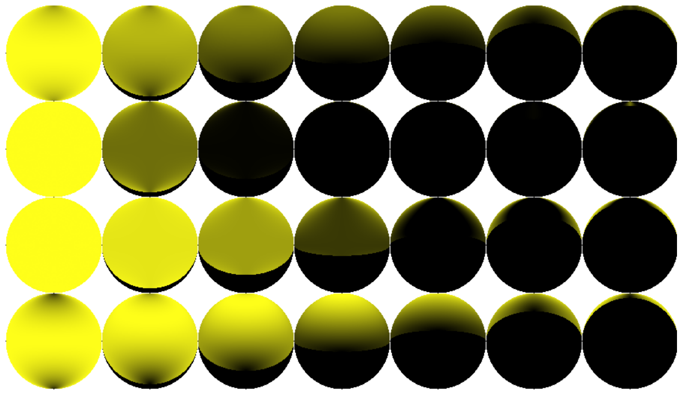
Figure 7: Hair.

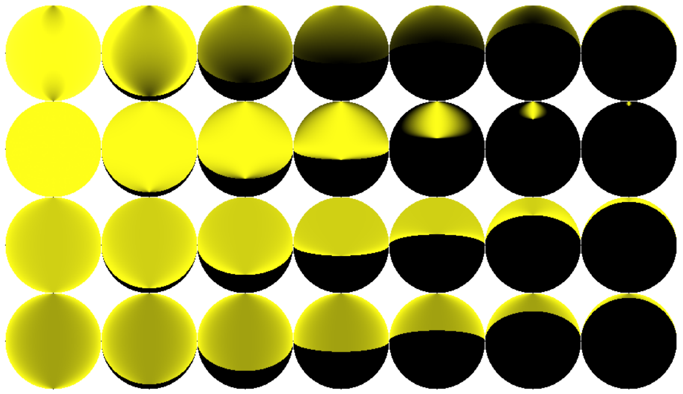
Figure 8: Fake Fur.

Companies such as nVidia have announced interest and support in 3-D texture maps, they are not currently available in an efficient form through current graphics API's. A 3-D texture map would be capable of storing reflectance information for specular reflectance and even diffuse bump mapping.

The advanced shaders we investigated typically use at least four distinct values as their parameters, which precludes the use of a texture map to lookup precomputed results. However, these advanced shaders are created from separable 2-D reflectance functions that can be combined to form the final multidimensional shader. [Kautz & McCool, 1999] decomposed 4-D BRDF's into a sequence of separable functions of 2-D reflectance functions. Basing the separability of shaders on the model instead of a general decomposition has the added benefit of supporting parameterization of the model, requiring recomputation of only the component whose parameter has changed, or even the real-time control of the blending operations between the individual lookup textures.

4.1 Cook-Torrance

The Torrance-Sparrow local illumination model is a highly empirical physically based method that is both experimentally and physically justified. The most common implementation of the Torrance-Sparrow model is the Cook-Torrance approximation [Cook & Torrance, 1982] of the specular component

$$r = \frac{FDG}{p\,(N \cdot V)(N \cdot L)} . \qquad (17)$$

The Fresnel effect is the total reflection of light glancing off of a surface at an angle shallower than the critical angle, which is modeled as

$$F = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left(1 + \frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right) \qquad (18)$$

where $c = V{\cdot}H$ and $g^2=\eta^2+c^2-1$. Computed directly, the divisions and square root would be costly, though feasible, for direct hardware implementation of this term. Alternatively, an approximation or a two-dimensional lookup table indexed by g and c would also suffice. The constants for the Frenel term vary with wavelength, so separate F terms can be computed for each color channel, resulting in a highlight that changes hue with intensity. The Frenel effect is plotted in the second row of Figure 5.

The roughness term is a distribution of the orientation of the microfacets, which is typically modeled by the Beckmann distribution function,

$$D = \frac{1}{4m^2(N \cdot H)^4}\exp\left(\frac{(N \cdot H)^2 - 1}{m^2(N \cdot H)^2}\right), \qquad (19)$$

parameterized by the surface roughness *m*. The Beckmann distribution function for *m*=0.6 is plotted in the fourth row of Figure 5. This could be implemented with a 2-D texture map parameterized by *N·H* and *m*, which would also allow the roughness to vary across a surface.

The geometric attenuation factor *G* accounts for self-shadowing

$$G = \min\left\{1, \frac{2(N\cdot H)}{V\cdot H}(N\cdot V), \frac{2(N\cdot H)}{V\cdot H}(N\cdot L)\right\} \tag{20}$$

as the smaller of one, the inverse of the percentage of blocked incident light, and the inverse of the percentage of blocked reflected light, and is demonstrated in the third row of Figure 5. The geometry term consists of four cosines *N·H*, *V·H*, *N·V* and *N·L*. However, the implementation can be separated into the product of two texture maps. A base 2-D texture map of 2(*N·H*)/(*V·H*), modulated by a 1-D texture maps containing either *N·V* or *N·L*. (If the API supports scaling by the texture coordinate, these 1-D texture maps could be eliminated.)

Note that the full Cook-Torrance implementation, shown in the first row of Figure 5, requires four cosines *N·H*, *N·L*, *V·H*, and *V·L*. Precomputation and storage of the lighting model would result in a four-dimensional table equivalent to the BRDF. Hence, programmable shading remains a more efficient implementation for this lighting model.

4.2 Multilayer Shaders

Multilayer shaders decompose reflected light into a surface scattered component and a sub-surface scattered component at each layer. There many applications of multilayer shaders, including materials such as skin, leaves, tree canopies and shallow ponds.

Whereas Lambertian reflection is constructed from geometric principles, Seeliger's model [Hanrahan & Krueger, 1993] is constructed from first principles in physics as

$$r = \frac{N\cdot L}{(N\cdot L)+(N\cdot V)} \tag{21}$$

It scatters light more uniformly than Lambert's law, providing a softer appearance similar to skin. Compare the fourth row (Lambertian) with the third row (Seeliger) of Figure 6. This lighting model is isotropic (but not bidirectional). It could be precomputed using a two-dimensional texture map indexed by the cosines *N·L* and *N·V*, or even by arithmetic on two texture coordinates.

The Henyey-Greenstein function was used to model the scattering of light by particles in a given layer

$$p(L\cdot V) = \frac{1}{4p}\frac{1-g^2}{(1+g^2-2gL\cdot V)^{3/2}} \tag{22}$$

which is parameterized by the mean cosine of the direction of scattering *g*. This scattering function is plotted as intensity in the second row of Figure 6.

The scattering function was used as a probability distribution function for the Monte Carlo model that constructed a full BRDF by sampling a hemisphere of incoming light and measuring the exiting light on the same hemisphere. However, the Henyey-Greenstein function could also be used as an opacity function for texture layers, as demonstrated in the first row of Figure 6. As such, it can be implemented as a 2-D texture indexed by the cosine *L·V* and the scattering parameter *g*. One possible improvement is to implement the Henyey-Greenstein scattering using the EMBM enhancement (8).

Alternatively, the entire skin reflection function could be implemented as a 3-D specular BRDF, indexed by *N·L*, *N·V* and *L·V*.

4.3 Anisotropic Shaders

Anisotropic lighting models require a grain tangent direction in the reflectance coordinate frame, and must also account for self-shadowing. The most common use for anisotropic reflection is in the simulation of hair and fur, but can also be used for brushed metals and grassy fields.

A BRDF for hair was modeled [Kajiya & Kay, 1989] with a diffuse component given by the sine of the angle between the hair direction and the light vector

$$r_d = \sin(T, L) = \sqrt{1-(T\cdot L)^2} \tag{23}$$

and the specular component as the sum of the products of the sines and cosines of the angle between the hair direction and the light vector and the view vector, raised to a specular exponent

$$r_s = \left((T\cdot L)(T\cdot V) + \sqrt{1-(T\cdot L)^2}\sqrt{1-(T\cdot V)^2}\right)^n. \tag{24}$$

Figure 7 shows these shading models. The fourth row is diffuse. The third row is specular with exponent one and the second row is specular with exponent 8. Note that the tangent dot products may be negative, such that raising to an even power changes the sign. The diffuse and specular components are combined in the first row.

The diffuse reflection can be implemented with as a 1-D texture map, indexed by *T·L*. (The cosine-to-sine conversion is so fundamental that perhaps it bears hardware implementation.) The specular reflection function can be implemented as a 2-D texture map, indexed by *T·L* and *T·V*. Alternatively, for directional light and orthographic views, this can be implemented using the tangent vector *T* as the texture coordinate, and using a texture transformation matrix whose first row is *L* and second row is *V* [Heidrich & Seidel, 1998].

This model was further enhanced for efficient use in the entertainment industry [Goldman, 1997]. The scattering of light by hair and fur is approximated by

$$p = \frac{(T\times L)\cdot(T\times V)}{|T\times L||T\times V|} \tag{25}$$

the cosine of the dihedral angle between the hair-light plane and the hair-view plane. Compare the fourth row of Figure 8, which contains the diffuse and specular terms, with the third row, which plots the scattering function.

An opacity function for hair is given by an inverted Gaussian

$$a(V) = 1 - \exp\left(\frac{-k\sqrt{1-(V\cdot T)^2}}{V\cdot N}\right) \qquad (26)$$

where k is a constant equal to the projected area of each strand of hair times the number of hair strands, both per unit square. This opacity function is plotted (for k=0.1) as intensity in the second row of Figure 8. These terms are collected to form a general reflectance model for hair

$$r_{\text{hair}} = a(V)(1-sa(L))\left(\frac{1+p}{2}k_r + \frac{1-p}{2}k_t\right)(k_d r_d + k_s r_s) \qquad (27)$$

which combines constants of reflection k_r and transmission k_t (backlighting), and diffuse k_d and specular k_s reflection. The fraction s is used to control the degree of self-shadowing of hair. This expression can be implemented as a multipass rendering, or a multitexturing if the API supports the operations. This result is demonstrated in the first row of Figure 8.

4.4 Non-Photorealistic Shaders

While photorealism has been a longstanding goal of computer graphics, a significant amount of attention has also been paid to the use of graphics for illustration and visualization. The fundamental problem in non-photorealistic rendering is silhouette detection. The silhouette of an object occurs where the surface normal is perpendicular to the view vector, which could be indicated by the reflectance

$$r = 1\text{-}(1\text{-}V\cdot N)^n \qquad (28)$$

where the exponent n indicates the crispness of the silhouette.

Shading in illustrations is often performed by hash marks, which often follow the tangent directions of the surface, and hardware shaders based on this form of shading would need the tangent vectors in addition to the surface normal to properly orient a prestored or synthesized hash texture. One could implement such hashing using a hashed spheremap.

5 Conclusion

We tackled the problem of analyzing present shader technology. We introduced a grammar capable of representing the fundamental nature of and differences between real-time shading techniques. We used this grammar to compare features of the standard pipeline with deferred rendering, multipass, multitexturing, texture shading and environment map techniques. We also evaluated these techniques with respect to a variety of advanced shaders.

We found that the natural progression of the real-time shader techniques leads to texture shading supported by multitexturing and multipass. We also found that storage of the BRDF is inefficient, and advanced shading procedures are too complex to implement directly, but they can however be assembled by multitexture components that consist of 2-D texture maps indexed by coordinates generated from dot products of shader vector variables.

5.1 Future Work

Analyzing real-time shading pipelines using the grammar provides a basis for innovation, and makes various commutations easier to consider. We expect this comparison may inspire new techniques based on innovative permutations of the parameterization, shader, projection and interpolation operations.

We also expect the grammar to grow more specific, providing a more detailed view of the specific channels and coordinates used for various shading effects.

Due to the constraints of time, we have omitted bump mapping, procedural texturing and the noise function from this discussion. Half of procedural shading is procedural texturing, though most of the attention on advanced shading has focused on lighting and local illumination models.

5.2 Acknowledgments

This work was supported in full by a consulting contract through the Evans & Sutherland Computer Corp. Conversations with Kurt Akeley, John Buchanan, Nate Carr, Rich Ehlers, Alain Fournier, Eric Haines, Pat Hanrahan, Chuck Hansen, Masaki Kameya, Michael McCool, Marc Olano and Steve Tibbitts were very useful in uncovering the details of many of these real-time shading techniques.

Bibliography

[Banks, 1994] D. C. Banks. Illumination in Diverse Codimensions. Computer Graphics (Proceedings of SIGGRAPH '94), 1994, pp. 327-334.

[Bishop & Weimer, 1986] Gary Bishop and David M. Weimer. Fast Phong Shading, Computer Graphics 20(4), (Proceedings of SIGGRAPH 86), Aug. 1986, pp. 103-106.

[Cabral *et al.*, 1999] Brian Cabral and Marc Olano and Philip Nemec. Reflection Space Image Based Rendering, Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, Aug. 1999, pp. 165-170.

[Carr, et al., 2000] Nate Carr, John Hart and Jerome Maillot. The Solid Map: Methods for Generating a 2-D Texture Map for Solid Texturing. Proc. Western Computer Graphics Symposium, Mar. 2000.

[Cook & Torrance, 1982] R. L. Cook and K. E. Torrance. A Reflectance Model for Computer Graphics, ACM Transactions on Graphics, 1 (1), January 1982, pp. 7-24.

[Crow, 1984] Franklin C. Crow. Summed-area Tables for Texture Mapping, Computer Graphics 18(3), (Proceedings of SIGGRAPH 84), July 1984, pp. 207-212.

[Goldman, 1997] Dan B. Goldman. Fake Fur Rendering, Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, Aug. 1997, pp. 127-134.

[Hanrahan & Lawson, 1990] Pat Hanrahan and Jim Lawson. A Language for Shading and Lighting Calculations, Computer Graphics 24(4), (Proceedings of SIGGRAPH 90), Aug. 1990, pp. 289-298.

[Hanrahan & Krueger, 1993] Pat Hanrahan and Wolfgang Krueger. Reflection from Layered Surfaces Due to Subsurface Scattering, Proceedings of SIGGRAPH 93, Aug. 1993, pp. 165-174.

[Hanrahan, 1999] Patrick Hanrahan. Real Time Shading Languages. Keynote, Eurographics/SIGGRAPH Workshop on Graphics Hardware, Aug. 1999

[Hart *et al.*, 1999] John C. Hart, Nate Carr, Masaki Kameya, Stephen A. Tibbitts and Terrance J. Coleman. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation, 1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware, Aug., 1999, pp. 45-53.

[Heckbert, 1990] Paul S. Heckbert. Adaptive Radiosity Textures for Bidirectional Ray Tracing, Computer Graphics (Proceedings of SIGGRAPH 90), 24 (4), August 1990, pp. 145-154

[Heidrich & Seidel, 1998] W. Heidrich and H.-P. Seidel. Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware. Image and Multi-dimensional Digital Signal Processing Workshop (IMDSP) 1998.

[Kajiya & Kay, 1989] James T. Kajiya and Timothy L. Kay. Rendering Fur with Three Dimensional Textures, Computer Graphics (Proceedings of SIGGRAPH 89), *23 (3)*, July 1989, pp. 271-280.

[Kajiya, 1985] James T. Kajiya. Anisotropic Reflection Models, Computer Graphics (Proceedings of SIGGRAPH 85), *19 (3)*, July 1985, pp. 15-21.

[Kajiya, 1986] James T. Kajiya. The Rendering Equation, Computer Graphics (Proceedings of SIGGRAPH 86), *20(4)*, August 1986, pp. 143-150.

[Kautz & McCool, 1999] Jan Kautz and Michael D. McCool. Interactive Rendering with Arbitrary BRDFs using Separable Approximations, Eurographics Rendering Workshop 1999, June 1999.

[Kilgard, 1999] Mark J. Kilgard. A Practical and Robust Bump-mapping Technique for Today's GPUs. nVidia Technical Report. Feb. 2000.

[Lastra *et al.*, 1995] Anselmo Lastra and Steven Molnar and Marc Olano and Yulan Wang. Real-Time Programmable Shading, 1995 Symposium on Interactive 3D Graphics, April 1995, pp. 59-66.

[Lewis, 1994] R. R. Lewis. Making Shaders More Physically Plausible, Computer Graphics Forum, *13 (2)*, January 1994, pp. 109-120.

[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.

[McCool & Heidrich, 1999] Michael D.McCool and Wolfgang Heidrich. Texture shaders, 1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware, August 1999, pp. 117-126.

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

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

[Olano, *et al.*, 2000] Marc Olano, et al., Interactive Multi-Pass Programmable Shading. To appear: Proc. SIGGRAPH 2000.

[Peercy *et al.*, 1997] Mark Peercy and John Airey and Brian Cabral. Efficient Bump Mapping Hardware, Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, Aug. 1997, pp. 303-306.

[Proudfoot, 1999] Kekoa Proudfoot. Real Time Shading Language Description, Version 4. Nov. 1999.

[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.

[Stalling & Zöckler, 1997] D. Stalling and M. Zöckler and H.-C. Hege Fast Display of Illuminated Field Lines. IEEE Transactions on Visualization and Computer Graphics, 3(2), 1997, pp. 118-128.

[Williams, 1978] Lance Williams. Casting Curved Shadows on Curved Surfaces, Computer Graphics (Proceedings of SIGGRAPH 78), *12(3)*, Aug. 1978, pp. 270-274.

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