A Shading Language on Graphics Hardware: The PixelFlow Shading System

Marc Olano[†] Anselmo Lastra[‡]

University of North Carolina at Chapel Hill

Abstract

Over the years, there have been two main branches of computer graphics image-synthesis research; one focused on interactivity, the other on image quality. Procedural shading is a powerful tool, commonly used for creating high-quality images and production animation. A key aspect of most procedural shading is the use of a shading language, which allows a high-level description of the color and shading of each surface. However, shading languages have been beyond the capabilities of the interactive graphics hardware community. We have created a parallel graphics multicomputer, PixelFlow, that can render images at 30 frames per second using a shading language. This is the first system to be able to support a shading language in real-time. In this paper, we describe some of the techniques that make this possible.

CR Categories and Subject Descriptors: D.3.2 [Language Classifications] Specialized Application Languages; I.3.1 [Computer Graphics] Hardware Architecture; I.3.3 [Computer Graphics] Picture/Image Generation; I.3.6 [Computer Graphics] Methodologies and Techniques; I.3.7 [Computer Graphics] Three-dimensional Graphics and Realism.

Additional Keywords: real-time image generation, procedural shading, shading language.

1 INTRODUCTION

We have created a SIMD graphics multicomputer, PixelFlow, which supports *procedural shading* using a shading language. Even a small (single chassis) PixelFlow system is capable of rendering scenes with procedural shading at 30 frames per second or more. Figure 1 shows several examples of shaders that were written in our shading language and rendered on PixelFlow.

In procedural shading, a user (someone other than a system designer) creates a short procedure, called a *shader*, to determine the final color for each point on a surface. The shader is responsible for color variations across the surface and the interaction of light with the surface. Shaders can use an assortment of input *appearance parameters*, usually including the surface normal, texture coordinates, texture maps, light direction and colors.

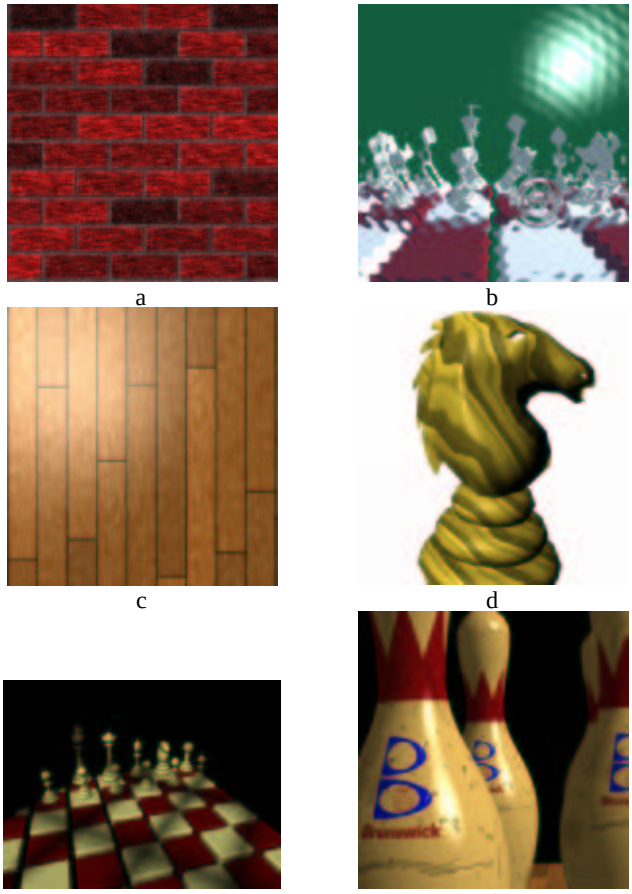


Figure 1: Some PixelFlow surface shaders. a) brick. b) mirror with animated ripple. c) wood planks. d) a volume-based wood. e) light shining through a paned window. f) view of a bowling scene.

Procedural shading is quite popular in the production industry where it is commonly used for rendering in feature films and commercials. The best known examples of this have been rendered using Pixar's PhotoRealistic RenderMan software [Upstill90]. A key aspect of RenderMan is its shading language. The shading language provides a high-level description of each procedural shader. Shaders written in the RenderMan shading

[†] Now at Silicon Graphics, Inc., 2011 N. Shoreline Blvd., M/S #590, Mountain View, CA 94043 (email: olano@engr.sgi.com)

[‡] UNC Department of Computer Science, Sitterson Hall, CB #3175, Chapel Hill, NC 27599 (email: lastra@cs.unc.edu)

language can be used by any compliant renderer, no matter what rendering method it uses.

There are several reasons to provide procedural shading instead of just image texturing on a real-time graphics system:

- It is easy to add noise and random variability to make a surface look more realistic.
- It can be easier to create a procedural shader for a complicated surface than to try to eliminate the distortions caused by wrapping a flat, scanned texture over the surface.
- It is easier to "tweak" a procedural shader than to rescan or repaint an image texture.
- It is often easier to create detail on an object using a procedural shader instead of modifying the object geometry.
- A procedurally shaded surface can change with time, distance, or viewing angle.

Usually procedural shading is associated with images that take a while to generate – from a few minutes to a day or so. Recently, graphics hardware reached the point where image texture mapping was not just possible, but common; now hardware is reaching the point where shading languages for interactive graphics are possible.

We have produced a shading language and shading language compiler for our high-end graphics machine, PixelFlow. This language is called *pfman* (*pf* for PixelFlow, *man* because it is similar to Pixar's RenderMan shading language). One of the great advantages of a shading language for procedural shading, particularly on a complex graphics engine, is that it effectively hides the implementation details from the shader-writer. The specifics of the graphics architecture are hidden in the shading language compiler, as are all of the tricks, optimizations, and special adaptations required by the machine. In this paper, we describe shading on PixelFlow, the pfman language, and the optimizations that were necessary to make it run in real-time.

Section 2 is a review of the relevant prior work. Section 3 covers features of the pfman shading language, paying particular attention to the ways that it differs from the RenderMan shading language. Section 4 describes our extensions to the OpenGL API [Neider93] to support procedural shading. Section 5 gives a brief overview of the PixelFlow hardware. Section 6 covers our implementation and the optimizations that are done by PixelFlow and the pfman compiler. Finally, Section 7 has some conclusions.

2 RELATED WORK

Early forms of programmable shading were accomplished by rewriting the shading code for the renderer (see, for example, [Max81]). Whitted and Weimer specifically allowed this in their testbed system [Whitted81]. Their *span buffers* are an implementation of a technique now called *deferred shading*, which we use on PixelFlow. In this technique, the parameters for shading are scan converted for a later shading pass. This allowed them to run multiple shaders on the same scene without having to re-render. Previous uses of deferred shading for interactive graphics systems include [Deering88] and [Ellsworth91].

More recently, easier access to procedural shading capabilities has been provided to the graphics programmer. Cook's *shade trees* [Cook84] were the base of most later shading works. He turned simple expressions, describing the shading at a point on the surface, into a parse tree form, which was interpreted. He introduced the name *appearance parameters* for the parameters that affect the shading calculations. He also proposed an orthogonal subdivision of types of programmable functions into displacement, surface shading, light, and atmosphere trees.

Perlin's image synthesizer extends the simple expressions in Cook's shade trees to a full language with control structures [Perlin85]. He also introduced the powerful Perlin noise function, which produces random numbers with a band-limited frequency spectrum. This style of noise plays a major role in many procedural shaders.

The RenderMan shading language [Hanrahan90][Upstill90] further extends the work of Cook and Perlin. It suggests new procedures for transformations, image operations, and volume effects. The shading language is presented as a standard, making shaders portable to any conforming implementation.

In addition to the shading language, RenderMan also provides a geometry description library (the RenderMan API) and a geometric file format (called RIB). The reference implementation is Pixar's PhotoRealistic RenderMan based on the *REYES* rendering algorithm [Cook87], but other implementations now exist [Slusallek94][Gritz96].

The same application will run on all of these without change. RenderMan effectively hides the details of the implementation. Not only does this allow multiple implementations using completely different rendering algorithms, but it means the user writing the application and shaders doesn't need to know anything about the rendering algorithm being used. Knowledge of basic graphics concepts suffices.

Previous efforts to support user-written procedural shading on a real-time graphics system are much more limited. The evolution of graphics hardware is only just reaching the point where procedural shading is practical. The only implementation to date was Pixel-Planes 5, which supported a simple form of procedural shading [Rhoades92]. The language used by this system was quite low level. It used an assembly-like interpreted language with simple operations like copy, add, and multiply and a few more complex operations like a Perlin noise function. The hardware limitations of Pixel-Planes 5 limited the complexity of the shaders, and the low-level nature of the language limited its use.

Lastra et. al. [Lastra95] presents previous work on the PixelFlow shading implementation. It analyzes results from a PixelFlow simulator for hand-coded shaders and draws a number of conclusions about the hardware requirements for procedural shading. At the time of that paper, the shading language compiler was in its infancy, and we had not addressed many of the issues that make a real-time shading language possible. [Lastra95] is the foundation on which we built our shading language.

3 SHADING LANGUAGE

A surface shader produces a color for each point on a surface, taking into account the color variations of the surface itself and the lighting effects. As an example, we will show a shader for a brick wall. The wall is rendered as a single polygon with texture coordinates to parameterize the position on the surface.

The shader requires several additional parameters to describe the size, shape, and color of the brick. These are the width and height of the brick, the width of the mortar, and the colors of the mortar and brick (Figure 2). These parameters are used to wrap

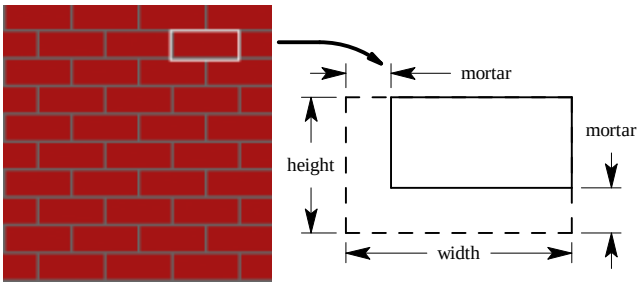


Figure 2: Example bricks and the size and shape parameters for the brick shader.

```
    // figure out which row of bricks this is (row is 8-bit integer)
fixed<8,0> row = tt / height;
    // offset even rows by half a row
if (row % 2 == 0) ss += width/2;
    // wrap texture coordinates to get "brick coordinates"
ss = ss % width;
tt = tt % height;
    // pick a color for the brick surface
float surface_color[3] = brick_color;
if (ss < mortar  ||  tt < mortar)
    surface_color = mortar_color;
```

Figure 3: Code from a simple brick shader

the texture coordinates into *brick coordinates* for each brick. These are (0,0) at the lower left corner of each brick, and are used to choose either the brick or mortar color. A portion of the brick shader is shown in Figure 3. The brick image in Figure 2 was generated with this shader.

One advantage of procedural shading is the ease with which shaders can be modified to produce the desired results. Figure 1a shows a more realistic brick that resulted from small modifications to the simple brick shader. It includes a simple procedurally-defined bump map to indent the mortar, high-frequency band-limited noise to simulate grains in the mortar and brick, patches of color variation within each brick to simulate swirls of color in the clay, and variations in color from brick to brick.

The remainder of this section covers some of the details of the pfman shading language and some of the differences between it and the RenderMan shading language. These differences are

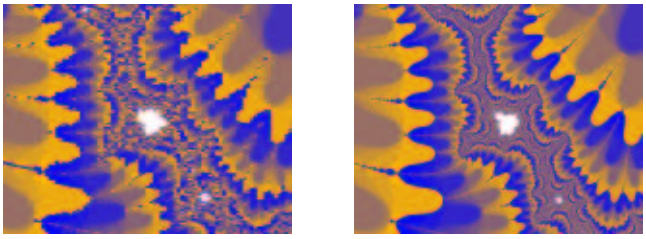
1. the introduction of a fixed-point data type,
2. the use of arrays for points and vectors,
3. the introduction of transformation attributes,
4. the explicit listing of all shader parameters, and
5. the ability to link with external functions.

Of these changes, 1 and 2 allow us to use the faster and more efficient fixed-point math on our SIMD processing elements. The third covers a hole in the RenderMan standard that has since been fixed. The fourth was not necessary, but simplified the implementation of our compiler. Finally, item 5 is a result of our language being compiled instead of interpreted (in contrast to most off-line renderer implementations of RenderMan).

3.1 Types

As with the RenderMan shading language, variables may be declared to be either `uniform` or `varying`. A `varying` variable is one that might vary from pixel to pixel – texture coordinates for example. A `uniform` variable is one that will never vary from pixel to pixel. For the brick shader presented above, the width, height and color of the bricks and the thickness and color of the mortar are all uniform parameters. These control the appearance of the brick, and allow us to use the same shader for a variety of different styles of brick.

RenderMan has one representation for all numbers: floating-point. We also support floating-point (32-bit IEEE single precision format) because it is such a forgiving representation. This format has about 10^{-7} relative error for the entire range of numbers from 10^{-38} to 10^{38}. However, for some quantities used in shading this range is overkill (for colors, an 8 to 16 bit fixed-point representation can be sufficient [Hill97]). Worse, there are cases where floating-point has too much range but not enough precision. For example, a Mandelbrot fractal shader has an insatiable appetite for precision, but only over the range [–2,2] (Figure 4). In this case, it makes much more sense to use a fixed-



a b

Figure 4: Fixed-point vs. floating-point comparison.
a) Mandelbrot set computed using floating-point.
b) Mandelbrot set computed using fixed-point

point format instead of a 32 bit floating-point format: the floating-point format wastes one of the four bytes for an exponent that is hardly used. In general, it is easiest to prototype a shader using floating-point, then change to fixed-point as necessary for memory usage, precision, and speed. Our fixed-point types may be signed or unsigned and have two parameters: the size in bits and an exponent, written `fixed<size,exponent>`. Fixed-point behaves like floating-point where the exponent is a compile-time constant. Small exponents can be interpreted as the number of fractional bits: a two byte integer is `fixed<16,0>`, while a two byte pure fraction is `fixed<16,16>`.

Like recent versions of the RenderMan shading language [Pixar97], pfman supports arrays of its basic types. However, where RenderMan uses separate types for points, vectors, normals, and colors, pfman uses arrays with *transformation attributes*. By making each point be an array of floating-point or fixed-point numbers, we can choose the appropriate representation independently for every point. A transformation attribute indicates how the point or vector should be transformed. For example, points use the regular transformation matrix, vectors use the same transformation but without translation, and normals use the adjoint or inverse without translation. We also include a transformation attribute for texture coordinates, which are transformed by the OpenGL texture transformation matrix.

3.2 Explicit Shader Parameters

RenderMan defines a set of *standard parameters* that are implicitly available for use by every surface shader. The surface shader does not need to declare these parameters and can use them as if they were global variables. In pfman, these parameters must be explicitly declared. This allows us to construct a transfer map (discussed later in Section 6) that contains only those parameters that are actually needed by the shader.

In retrospect, we should have done a static analysis of the shader function to decide which built-in parameters are used. This would have made pfman that much more like RenderMan, and consequently that much easier for new users already familiar with RenderMan.

3.3 External Linking

Compiling a pfman shader is a two-stage process. The pfman compiler produces C++ source code. This C++ code is then compiled by a C++ compiler to produce an object file for the shader. The function definitions and calls in pfman correspond directly to C++ function definitions and calls. Thus, unlike most RenderMan implementations, we support calling C++ functions from the shading language and vice versa. This facility is limited to functions using types that the shading language supports.

Compiling to C++ also provides other advantages. We ignore certain optimizations in the pfman compiler since the C++ com-

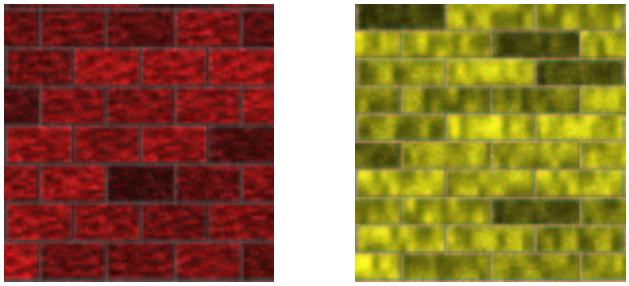


Figure 5: Instances of a brick surface shader.

piler does them. One could also use the generated C++ code as a starting point for a hand-optimized shader. Such a hand-optimized shader would no longer be portable, and performing the optimization would require considerable understanding of the PixelFlow internals normally hidden by the shading language. Not surprisingly, no one has done this yet.

4 API

The RenderMan standard [Upstill90] defines not only the shading language, but also a graphics application program interface (API). This is a library of graphics functions that the graphics application can call to describe the scene to the renderer. We elected to base our API on OpenGL [Neider93] instead of RenderMan. OpenGL is a popular API for interactive graphics applications, supported on a number of graphics hardware platforms. It provides about the same capabilities as the RenderMan API, with a similar collection of functions, but with more focus on interactive graphics. By using OpenGL as our base we can easily port applications written for other hardware.

We extended OpenGL to support procedural shading [Leech98]. We required that the procedural shading extensions have no impact on applications that do not use procedural shading. We also endeavored to make them fit the framework and philosophy of OpenGL. Our efforts to extend OpenGL should be readily usable by future real-time shading language systems.

Following the OpenGL standard, all of our extensions have the suffix `EXT`. We will follow that convention here to help clarify what is already part of OpenGL and what we added. OpenGL functions also include suffix letters (`f`, `i`, `s`, etc.) indicating the operand type. For brevity, we omit these in the text.

4.1 Loading Functions

Procedural surface shaders and lights are written as pfman functions. The new API call, `glLoadExtensionCodeEXT`, loads a shader. Currently, we do not support dynamic linking of surface or light functions, so this call just declares which shaders will be used. In the future, we do plan to dynamically load shaders.

4.2 Shading Parameters

On PixelFlow, the default shader implements the OpenGL shading model. Applications that do not "use" procedural shading use this default *OpenGL shader* without having to know any of the shading extensions to OpenGL.

We set the values for shading parameters using the `glMaterial` call, already used by OpenGL to set parameters for the built-in shading model. Parameters set in this fashion go into the OpenGL global state, where they may be used by any shader. Any shader can use the same parameters as the OpenGL shader simply by sharing the same parameter names, or it can define its own new parameter names.

OpenGL also has a handful of other, parameter-specific, calls. `glColor` can be set to change any of several possible color parameters, each of which can also be changed with `glMaterial`. We created similar parameter name equivalents for `glNormal` and `glTexCoord`. Other shaders may use these names to access the normals set with `glNormal` and texture coordinates from `glTexcoord`.

4.3 Shader Instances

The RenderMan API allows some parameter values to be set when a shader function is chosen. Our equivalent is to allow certain *bound* parameter values. A shading function and its bound parameters together make a *shader instance* (or sometimes just *shader*) that describes a particular type of surface. Because the character of a shader is as much a product of its parameter settings as its code, we may create and use several instances of each shading function. For example, given the brick shading function of Figure 3, we can define instances for fat red bricks and thin yellow bricks by using different bound values for the width, height, and color of the bricks (Figure 5).

To set the bound parameter values for an instance, we use a `glBoundMaterialEXT` function. This is equivalent to `glMaterial`, but operates only on bound parameters.

We create a new instance with a `glNewShaderEXT`, `glEndShaderEXT` pair. This is similar to the way OpenGL defines other objects, for example display list definitions are bracketed by calls to `glNewList` and `glEndList`. `glNewShaderEXT` takes the shading function to use and returns a *shader ID* that can be used to identify the instance later. Between the `glNewShaderEXT` and `glEndShaderEXT` we use `glShaderParameterBindingEXT`, which takes a parameter ID and one of `GL_MATERIAL_EXT` or `GL_BOUND_MATERIAL_EXT`. This indicates whether the parameter should be set by calls to `glMaterial` (for ordinary parameters) or `glBoundMaterialEXT` (for bound parameters).

To choose a shader instance, we call `glShaderEXT` with a shader ID. Primitives drawn after the `glShaderEXT` call will use the specified shader instance.

4.4 Lights

OpenGL normally supports up to eight lights, `GL_LIGHT0` through `GL_LIGHT7`. New light IDs beyond these eight are created with `glNewLightEXT`. Lights are turned on and off through calls to `glEnable` and `glDisable`. Parameters for the lights are set with `glLight`, which takes the light ID, the parameter name, and the new value. As with surface shaders, we have a built-in OpenGL light that implements the OpenGL lighting model. The eight standard lights are pre-loaded to use this function.

The OpenGL lighting model uses multiple colors for each light, with a different color for each of the ambient, diffuse and specular shading computations. In contrast, the RenderMan lighting model has only one color for each light. We allow a mix of these two styles. The only constraint is that surface shaders that use three different light colors can only be used with lights that provide three light colors. Surface shaders that follow the RenderMan model will use only the diffuse light color from lights that follow the OpenGL model.

5 PIXELFLOW

We implemented the pfman shading language on PixelFlow, a high-performance graphics machine. The following sections give

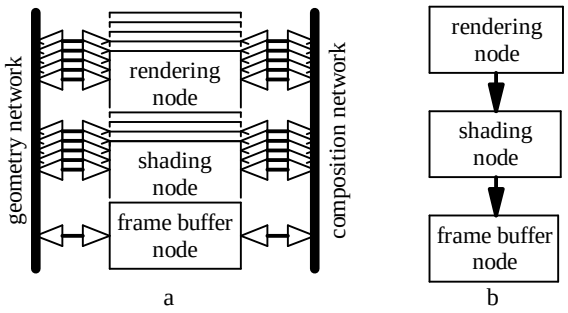


Figure 6: PixelFlow: a) machine organization.
b) simplified view of the system.

a brief overview of PixelFlow. For more details, refer to [Molnar92] or [Eyles97]

5.1 Low-level View

A typical PixelFlow system consists of a host, a number of rendering nodes, a number of shading nodes, and a frame buffer node (Figure 6a). The rendering nodes and shading nodes are identical, so the balance between rendering performance and shading performance can be decided for each application. The frame buffer node is also the same, though it includes an additional *daughter card* to produce video output. The host is connected through a daughter card on one of the rendering nodes.

The pipelined design of PixelFlow allows the rendering performance to scale linearly with the number of rendering nodes and the shading performance to scale linearly with the number of shading nodes.

Each rendering node is responsible for an effectively random subset of the primitives in the scene. The rendering nodes handle one 128x64 pixel *region* at a time. More precisely, the region is 128x64 image samples. When antialiasing, the image samples are blended into a smaller block of pixels after shading. For brevity, we will continue to use the word "pixel", with the understanding that sometimes they may be image samples instead of actual pixels.

Since each rendering node has only a subset of the primitives, a region rendered by one node will have holes and missing polygons. The different versions of the region are merged using a technique called *image composition*. PixelFlow includes a special high-bandwidth *composition network* that allows image composition to proceed at the same time as pixel data communication. As all of the rendering nodes simultaneously transmit their data for a region, the hardware on each node compares, pixel-by-pixel, the data it is transmitting with the data from the upstream nodes. It sends the closer of each pair of pixels downstream. By the time all of the pixels reach their destination, one of the system's shading nodes, the composition is complete.

Once the shading node has received the data, it does the surface shading for the entire region. In a PixelFlow system with n shading nodes, each shades every n^{th} region. Once each region has been shaded, it is sent over the composition network (without compositing) to the frame buffer node, where the regions are collected and displayed.

Each node has two RISC processors (HP PA-8000's), a custom SIMD array of pixel processors, and a texture memory store. Each processing element of the SIMD array has 256 bytes of memory, an 8-bit ALU with support for integer multiplication, and an enable flag indicating the active processors. All enabled processors in the 128x64 array simultaneously compute, on their own data, the result of any operation. This provides a speedup of up to 8192 times the rate of a single processing element.

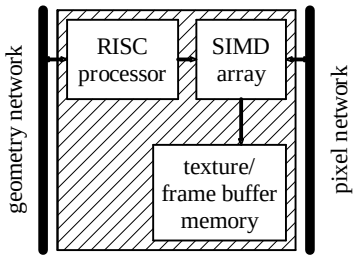


Figure 7: Simple block diagram of a PixelFlow node.

5.2 High-level View

The hardware and basic system software handle the details of scheduling primitives for the rendering nodes, compositing pixel samples from these nodes, assigning them to shading nodes, and moving the shaded pixel information to the frame buffer. Consequently, it is possible to take the simplified view of PixelFlow as a simple pipeline (Figure 6b). This view is based on the passage of a single displayed pixel through the system. Each displayed pixel arrives at the frame buffer, having been shaded by a single shading node. We can ignore the fact that displayed pixels in other regions were shaded by different physical shading nodes. Before arriving at the shading node, the pixel was part of a primitive on just one of the rendering nodes. We can ignore the fact that other pixels may display different primitives from different rendering nodes.

Only the rendering nodes make use of the second RISC processor. The primitives assigned to the node are split between the processors. We can take the simplified view that there is only one processor on the node, and let the lower level software handle the scheduling between the physical processors. Figure 7 is simple block diagram of a PixelFlow node with these simplifications. Each node is connected to two communication networks. The geometry network (800 MB/s in each direction), handles information about the scene geometry, bound parameter values, and other data bound for the RISC processors. It is 32 bits wide, operating at 200 MHz. The composition network (6.4 GB/s in each direction) handles transfers of pixel data from node to node. It is 256 bits wide, also operating at 200 MHz. Since our simplified view of the PixelFlow system hides the image composition, it is reasonable to simply refer to the composition network as a pixel network.

6 IMPLEMENTATION

Implementation of a shading language on PixelFlow requires optimizations. Some are necessary to achieve the targeted interactive rates of 20-30 frames per second, whereas others are necessary just to enable shaders to run on PixelFlow. The three scarce resources impact our PixelFlow implementation: time, communication bandwidth, and memory. In this section, we present optimizations to address each.

6.1 Execution Optimizations

Our target frame rate of 30 frames per second translates to 33 ms per frame. The system pipelining means that most of this time is actually available for shading. Each shading node can handle one 128x64 region at a time and a 1280x1024 screen (or 640x512 screen with 4-sample antialiasing) contains 160 such regions. On a system with four shading nodes, each is responsible for 40 regions and can take an average of 825 μs shading each region. On a larger system with 16 shading nodes, each is responsible for

shader	bytes free	execution time
brick	46	613.15 μs
ripple reflection	59	1058.07 μs
planks	105	532.30 μs
bowling pin	86	401.96 μs
nanoManipulator 1	75	567.95 μs
nanoManipulator 2	1	2041.44 μs
nanoManipulator 3	51	1638.67 μs

Table 1: Memory and performance summary.

10 regions and can spend an average of 3.3 ms shading a region. Table 1 shows per-region execution times fore some sample shaders. The first four shaders appear in Figure 1. The other shaders were written by the UNC nanoManipulator project for surface data visualization.

6.1.1 Deferred Shading

Deferred shading is the technique of performing shading computations on pixels only after the visible pixels have been determined [Whitted81][Deering88][Ellsworth91]. It provides several advantages for the execution of surface shading functions. First, no time is wasted on shading computations for pixels that will not be visible. Second, our SIMD array can simultaneously evaluate a single surface shader instance on every primitive that uses it in a 128x64 region. Finally, it decouples the rendering performance and shading performance of the system. To handle more complex shading, add more shading hardware. To handle more complex geometry, add more rendering hardware.

6.1.2 Uniform and Varying

RenderMan has *uniform* and *varying* types (Section 3.1), in part for the efficiency of their software renderer. A *uniform expression* uses only uniform operands and has a uniform result; a *varying expression* may have both uniform and varying operands but has a varying result. As Pixar's prman renderer evaluates the shading on a surface, it computes uniform expressions only once, sharing the results with all of the surface samples, but loops over the surface samples to compute the varying expressions.

We can use a similar division of labor. The microprocessor on each PixelFlow node can compute the result of a single operation much faster than the SIMD array; but the microprocessor produces one result, while the SIMD array can produce a different result on each of the 8K pixel processing elements. If the value is the same everywhere, it is faster for the microprocessor to compute and broadcast the result to the pixel processors. If the value is different at different pixel processors, it is much faster to allow the SIMD array to compute all of the results in parallel.

Since uniform expressions do not vary across the pixels, it is much more efficient to compute them using the microprocessor and store them in microprocessor memory. In contrast, varying expressions are the domain of the pixel processors. They can potentially have different values at every pixel, so must exist in pixel memory. They are fast and efficient because their storage and operations are replicated across the SIMD array. This same distinction between shared (*uniform*) and SIMD array (*varying*) memory was made by Thinking Machines for the Connection Machine [ThinkingMachines89], though they called them *mono* and *poly*, and by MasPar for the MP-1 [MasPar90], though their terms were *singular* and *plural*.

Operation	16-bit fixed	32-bit fixed	32-bit float
`+`	0.07 μs	0.13 μs	3.08 μs
`*`	0.50 μs	2.00 μs	2.04 μs
`/`	1.60 μs	6.40 μs	7.07 μs
`sqrt`	1.22 μs	3.33 μs	6.99 μs
`noise`	5.71 μs	—	21.64 μs

Table 2: Fixed-point and floating-point execution times for 128x64 SIMD array.

6.1.3 Fixed-point

We can achieve significant speed improvements by using fixed-point operations for varying computations instead of floating-point. Our pixel processors do not support floating-point in hardware: every floating-point operation is built from basic integer math operations. These operations consist of the equivalent integer operation with bitwise shifts to align the operands and result. Fixed-point numbers may also require shifting to align the decimal points, but the shifts are known at compile-time. The timings of some fixed-point and floating-point operations are shown in Table 2. These operations may be done by as many as 8K pixel processors at once, yet we would still like them to be as fast as possible.

6.1.4 Math Functions

We provide floating-point versions of the standard math library functions. An efficient SIMD implementation of these functions has slightly different constraints than a serial implementation. Piece-wise polynomial approximation is the typical method to evaluate transcendental math functions.

This approach presents a problem on PixelFlow due to the handling of conditionals on a SIMD array. On a SIMD array, the condition determines which processing elements are enabled. The true part of an `if`/`else` is executed with some processing elements enabled, the set of enabled processors is flipped and the false part is executed. Thus the SIMD array spends the time to execute both branches of the `if`.

This means that using a table of 32 polynomials takes as much time as a single polynomial with 32 times as many terms covering the entire domain. Even so, a polynomial with, say, 160 terms is not practical. For each PixelFlow math function, we reduce the function domain using identities, but do not reduce it further. For example, the log of a floating-point number, $m*2^e$, is `e*log(2)+log(m)`. We fit `log(m)` with a single polynomial. Each polynomial is chosen to use as few terms as possible while remaining accurate to within the floating-point precision. Thus, we still do a piece-wise fit, but fit a single large piece with a polynomial of relatively high degree.

While we provide accurate versions of the math functions, often shaders do not really need the "true" function. With the ripple reflection shader in Figure 1b, it is not important that the ripples **be** sine waves. They just need to **look like** sine waves. For that reason, we also provide faster, visually accurate but numerically poor, versions of the math functions. The fast versions use simpler polynomials, just matching value and first derivative at each endpoint of the range fit by the more exact approximations. This provides a function that appears visually cor-

function	exact	fast
sin	81.36 μs	45.64 μs
`cos`	81.36 μs	48.77 μs
`tan`	93.25 μs	52.65 μs
`asin, acos`	78.52 μs	47.50 μs
`atan`	66.41 μs	35.34 μs
`atan2`	66.17 μs	35.15 μs
`exp`	53.37 μs	37.86 μs
`exp2`	51.09 μs	35.58 μs
`log`	57.76 μs	21.57 μs
`log2`	57.68 μs	21.49 μs

Table 3: Execution times for floating-point math functions on 128x64 SIMD array.

```
// setup, compute base surface color
illuminance() {
    // add in  the contribution of one light
}
// wrap-up
```

Figure 8: Outline of a typical surface shader.

rect but executes in about half the time.

6.1.5 Combined Execution

Many shading functions have similar organizations. Combining the execution of the common sections of code in multiple shaders can lead to large gains in performance. In the next few sections, we will discuss some of these methods. The easiest and most automatic of this class of optimizations is combined execution of lights for all surface shaders. For some of the more traditional surface shaders, involving image texture lookups and Phong shading, we can do further overlapped computation.

6.1.5.1 Lights

One of the jobs of a surface shader is to incorporate the effects of each light in the scene. As in the RenderMan shading language, this is accomplished through the `illuminance` construct, which behaves like a loop over the active lights (Figure 8). This means that each surface shader effectively includes a loop over every light. For `m` shaders and `n` lights, this results in `m*n` light executions. This can be quite expensive since the lights themselves are procedural, and could be arbitrarily complex. Since the lights are the same for each of the `m` shaders, we compute each light just once and share its results among all of the shaders, resulting in only `n` light executions. We do this by interleaving the execution of all of the lights and shaders.

We accomplish this interleaving by having each surface shader generate three instruction streams for the SIMD array. The first stream, which we call *`pre-illum`*, contains only the setup code (until the `illuminance` in Figure 8). The second stream contains the body of the `illuminance` construct. We call this the *`illum`* stream. Finally, the *`post-illum`* stream contains everything after the `illuminance`. The lights themselves create their own stream of SIMD commands. The interleaving pattern of these streams is shown in Figure 9.

The SIMD memory usage of the surfaces and lights must be chosen in such a way that each has room to operate, but none conflict. The surface shaders will not interfere with each other since any one pixel can only use one surface shader. Different surface shaders already use different pixel memory maps. Lights, however, must operate in an environment that does not disturb any surface shader, but provides results in a form that all surface

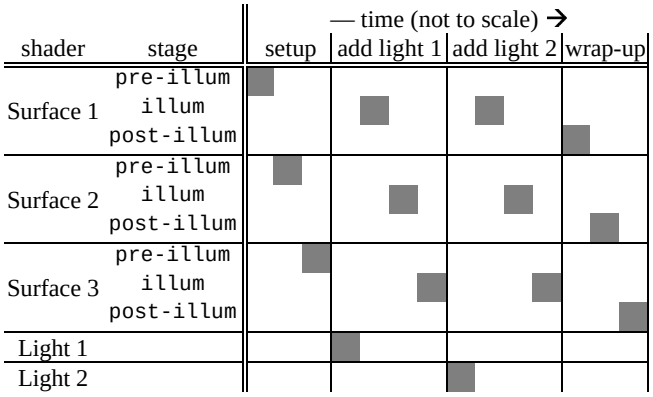


Figure 9: Interleaving of surface shaders and lights.

shaders can use. The results of the lighting computation, the color and direction of the light hitting each pixel, are stored in a special communications area to be shared by all surface shaders. The light functions themselves operate in the SIMD memory left over by the retained result of the greediest of the surface shader `pre-illum` stages. Above this *high water mark*, the light can freely allocate whatever memory it needs. The `illum`, and `post-illum` streams of all shaders can use all available memory without interfering with either the other surfaces or the lights.

6.1.5.2 Surface Position

For image composition, every pixel must contain the Z-buffer depth of the closest surface visible at that pixel. This Z value, along with the position of the pixel on the screen, is sufficient to compute where the surface sample is in 3D. Since the surface position can be reconstructed from these pieces of information, we do not store the surface position in pixel memory during rendering or waste composition bandwidth sending it from the rendering nodes to the shading nodes. Instead, we compute it on the shading nodes in a phase we call `pre-shade`, which occurs before any shading begins. Thus, we share the execution time necessary to reconstruct the surface position. We also save memory and bandwidth early in the graphics pipeline, helping with the other two forms of optimization, to be mentioned later.

6.1.5.3 Support for Traditional Shaders

Some optimizations have been added to assist in cases that are common for forms of the OpenGL shading model. Unlike the earlier execution optimizations, these special-purpose optimizations are only enabled by setting flags in the shader.

Surface shaders that use only the typical Phong shading model can use a shared `illum` stream. This allows shaders to set up different parameters to the Phong shader, but the code for the Phong shading model runs only once.

Surface shaders that use a certain class of texture lookups can share the lookup computations. These shaders know what texture they want to look up in the `pre-illum` phase, but don't require the results until the `post-illum` phase. The PixelFlow hardware does not provide any significant improvement in actual lookup time for shared lookups, but this optimization allows the SIMD processors to perform other operations while the lookup is in progress. To share the lookup processing, they place their *texture ID* and *texture coordinates* in special shared "magic" parameters. The results of the lookup are placed in another shared magic parameter by the start of the `post-illum` stage.

6.1.6 Cached Instruction Streams

On PixelFlow, the microprocessor code computes the uniform expressions and all of the uniform control flow (`if`'s with uniform conditions, `while`'s, `for`'s, etc.), generating a stream of SIMD processor instructions. This SIMD instruction stream is buffered for later execution. The set of SIMD instructions for a shader only changes when some uniform parameter of the shader changes, so we cache the instruction stream and re-use it. Any parameter change sets a flag that indicates that the stream must be regenerated. For most non-animated shaders, this means that the uniform code executes only once, when the application starts.

6.2 Bandwidth Optimizations

Communication bandwidth is another scarce resource on PixelFlow. As mentioned in Section 5, there are two communication paths between nodes in the PixelFlow system, the geometry net

and composition net. We are primarily concerned with the composition net bandwidth. While its total bandwidth is 6.4 GB/s, four bytes of every transfer are reserved for the pixel depth, giving an effective bandwidth of 5.6 GB/s.

Since PixelFlow uses deferred shading, the complete set of varying shading parameters and the shader ID must be transferred across the composition network. The final color must also be transferred from the shader node to the frame buffer. However, the design of the composition network allows these two transfers to be overlapped, so we really only pay for the bandwidth to send data for each visible pixel from the rendering nodes to shading nodes. At 30 frames per second on a 1280x1024 screen, the maximum communication budget is 142 bytes per pixel. To deal with this limited communication budget, we must perform some optimizations to reduce the number of parameters that need to be sent from renderer node to shader node.

6.2.1 Shader-Specific Maps

Even though each 128x64 pixel region is sent as a single transfer, every pixel could potentially be part of a different surface. Rather than use a transfer that is the union of all the parameters needed by all of those surface shaders, we allow each to have its own tailored transfer map. The first two bytes in every map contain the *shader ID*, which indicates what transfer map was used and which surface shader to run.

6.2.2 Bound Parameters

The bound parameters of any shader instance cannot change from pixel to pixel (Section 4.3), so they are sent over the geometry network directly to the shading nodes. Since the shader nodes deal with visible pixels without any indication of when during the frame they were rendered, we must restrict bound parameters to only change between frames. Bound uniform parameters are used directly by the shading function running on the microprocessor. Any bound varying parameters must be loaded into pixel memory. Based on the shader ID stored in each pixel, we identify which pixels use each shader instance and load their bound varying parameters into pixel memory before the shader executes.

Any parameter that is bound in every instance of a shader should probably be uniform, since this gives other memory and execution time gains. However, it is occasionally helpful to have bound values for varying shading parameters. For example, our brick shader may include a `dirtiness` parameter. Some brick walls will be equally dirty everywhere. Others will be dirtiest near the ground and clean near the top. The instance used in one wall may have `dirtiness` as a bound parameter, while the instance used in a second wall allows `dirtiness` to be set using `glMaterial` with a different value at each vertex.

However, the set of parameters that should logically be bound in some instances and not in others is small. Allowing bound values for varying parameters would be only a minor bandwidth savings, were it not for another implication of deferred shading. Since bound parameters can only change once per frame, we find parameters that would otherwise be uniform are being declared as varying solely to allow them to be changed with `glMaterial` from primitive to primitive (instead of requiring hundreds of instances). This means that someone writing a PixelFlow shader may make a parameter varying for flexibility even though it will never actually vary across any primitives. Allowing instances to have bound values for all parameters helps counter the resulting explosion of pseudo-varying parameters.

6.3 Memory Optimizations

The most limited resource when writing shaders on PixelFlow is pixel memory. The texture memory size (64 megabytes) affects the size of image textures a shader can use in its computations, but does not affect the shader complexity. The microprocessor memory (128 megabytes), is designed to be sufficient to hold large geometric databases. For shading purposes it is effectively unlimited. However, the pixel memory, at only 256 bytes, is quite limited. From those 256 bytes, we further subtract the shader input parameters and an area used for communication between the light shaders and surface shaders. What is left is barely enough to support a full-fledged shading language. The memory limitations of Pixel-Planes 5 were one of the reasons that, while it supported a form of procedural shading, it could not handle a true shading language. In this section we highlight some of the pfman features and optimizations made by the pfman compiler to make this limited memory work for real shaders.

6.3.1 Uniform vs. Varying

We previously mentioned uniform and varying parameters in the context of execution optimizations. Bigger gains come from the storage savings: uniform values are stored in the large main memory instead of the much more limited pixel memory.

6.3.2 Fixed-point

PixelFlow can only allocate and operate on multiples of single bytes, yet we specify the size of our fixed-point numbers in bits. This is because we can do a much better job of limiting the sizes of intermediate results in expressions with a more accurate idea of the true range of the values involved. For example, if we add two two-byte integers, we need three bytes for the result. However, if we know the integers really only use 14 bits, the result is only 15 bits, which still fits into two bytes.

A two-pass analysis determines the sizes of intermediate fixed-point results. A *bottom-up* pass determines the sizes necessary to keep all available precision. It starts with the sizes it knows (e.g. from a variable reference) and combines them according to simple rules. A *top-down* pass limits the fixed-point sizes of the intermediate results to only what is necessary.

6.3.3 Memory Allocation

The primary feature that allows shaders to have any hope of working on PixelFlow is the memory allocation done by the compiler. Since every surface shader is running different code, we use a different memory map for each. We spend considerable compile-time effort creating these memory maps.

Whereas even the simplest of shaders may define more than 256 bytes of varying variables, most shaders do not use that many variables at once. We effectively treat pixel memory as one giant register pool, and perform register allocation on it during compilation. This is one of the most compelling reasons to use a compiler when writing surface shaders to run on graphics hardware. It is possible to manually analyze which variables can coexist in the same place in memory, but it is not easy. One of the authors did just such an analysis for the Pixel-Planes 5 shading code. It took about a month. With automatic allocation, it suddenly becomes possible to prototype and change shaders in minutes instead of months.

The pfman memory allocator performs variable lifetime analysis by converting the code to a static single assignment (SSA) form [Muchnick97][Briggs92] (Figure 10). First, we go through the shader, creating a new temporary variable for the result of every assignment. This is where the method gets its name: we do

```
i = 1;              i1 = 1;              i1 = 1;
i = i + 1;          i2 = i1 + 1;         i2_3 = i1 + 1;
if (i > j)          if (i2 > j1)         if (i2_3 > j1)
    i = 5;              i3 = 5;              i2_3 = 5;
j = i;              j2 = φ(i2,i3);       j2 = i2_3;
      a                   b                    c
```

Figure 10: Example of lifetime analysis using SSA. a) original code fragment. b) code fragment in SSA form. Note the new variables used for every assignment and the use of the φ-function for the ambiguous assignment. c) final code fragment with φ-functions merged.

a static analysis, resulting in one and only one assignment for every variable. In some places, a variable reference will be ambiguous, potentially referring to one of several of these new temporaries. During the analysis, we perform these references using a *φ-function*. The φ-function is a pseudo-function-call indicating that, depending on the control flow, one of several variables could be referenced. For example, the value of `i` in the last line of Figure 10b, could have come from either `i2` or `i3`. In these cases, we merge the separate temporaries back together into a single variable. What results is a program with many more variables, but each having as short a lifetime as possible.

Following the SSA lifetime analysis, we make a linear pass through the code, mapping these new variables to free memory as soon as they become live, and unmapping them when they are no longer live. Variables can only become live at assignments and can only die at their last reference. As a result of these two passes, variables with the same name in the user's code may shift from memory location to memory location. We only allow these shifts when the SSA name for the variable changes. One of the most noticeable effects of the this analysis is that a variable that is used independently in two sections of code does not take space between execution of the sections.

Table 4 shows the performance of the memory allocation on an assortment of shaders. Table 1 shows the amount of memory left after the shading parameters, shader, light, and all overhead have been factored out.

shader	total (uniform + varying)	varying only	varying with allocation
simple brick	171	97	16
fancy brick	239	175	101
ripple reflection	341	193	137
wood planks	216	152	97

Table 4: Shader memory usage in bytes.

7 CONCLUSIONS

We have demonstrated an interactive graphics platform that supports procedural shading through a shading language. With our system, we can write shaders in a high-level shading language, compile them, and generate images at 30 frames per second or more. To accomplish this, we modified a real-time API to support procedural shading and an existing shading language to include features beneficial for a real-time implementation.

Our API is based on OpenGL, with extensions to support the added flexibility of procedural shading. We believe the decision to extend OpenGL instead of using the existing RenderMan API was a good one. Many existing interactive graphics applications are already written in OpenGL, and can be ported to PixelFlow with relative ease. Whereas the RenderMan API has better support of curved surface primitives important for its user community, OpenGL has better support for polygons, triangle strips and display lists, important for interactive graphics hardware.

Our shading language is based on the RenderMan shading language. Of the differences we introduced, only the fixed-point data type was really necessary. We expect that future hardware-assisted shading language implementations may also want similar fixed-point extensions. The other changes were either done for implementation convenience or to fill holes in the RenderMan shading language definition that have since been addressed by more recent versions of RenderMan. If we were starting the project over again today, we would just add fixed-point to the current version of the RenderMan shading language.

We have only addressed surface shading and procedural lights. RenderMan also allows other types of procedures, all of which could be implemented on PixelFlow, but have not been. We also do not have derivative functions, an important part of the RenderMan shading language. Details on how these features could be implemented on PixelFlow can be found in [Olano98]

We created a shading language compiler, which hides the details of our hardware architecture. The compiler also allows us to invisibly do the optimizations necessary to run on our hardware. We found the most useful optimizations to be those that happen automatically. This is consistent with the shading language philosophy of hiding system details from the shader writer.

Using a compiler and shading language to mask details of the hardware architecture has been largely successful, but the hardware limitations do peek through as shaders become short on memory. Several of our users have been forced to manually convert portions of their large shaders to fixed-point to allow them to run. Even after such conversion, one of the shaders in Table 1 has only a single byte free. If a shader exceeds the memory resources after it is converted to fixed-point, it cannot run on PixelFlow. If this becomes a problem, we can add the capability to spill pixel memory into texture memory, at a cost in execution speed.

Any graphics engine capable of real-time procedural shading will require significant pixel-level parallelism, though this parallelism may be achieved through MIMD processors instead of SIMD as we used. For the near future, this level of parallelism will imply a limited per-pixel memory pool. Consequently, we expect our memory optimization techniques to be directly useful for at least the next several real-time procedural-shading machines. Our bandwidth optimization techniques are somewhat specific to the PixelFlow architecture, though should apply to other deferred shading systems since they need to either transmit or store the per-pixel appearance parameters between rendering and shading. Deferred shading and our experience with function approximation will be of interest for future SIMD machines. The other execution optimizations, dealing with tasks that can be done once instead of multiple times, will be of lasting applicability to anyone attempting a procedural shading machine.

There is future work to be done extending some of our optimization techniques. In particular, we have barely scratched the surface of automatic combined execution of portions of different shaders. We do only the most basic of these optimizations automatically. Some others we do with hints from the shader-writer, whereas other possible optimizations are not done at all. For example, we currently run every shader instance independently. It would be relatively easy to identify and merge instances of the same shader function that did not differ in any uniform parameters. For a SIMD machine like ours, this would give linear speed improvement with the number of instances we can execute together. Even more interesting would be to use the techniques of [Dietz92] and [Guenter95] to combine code within a shader and between shader instances with differing uniform parameter values.

Creating a system that renders in real-time using a shading language has been richly rewarding. We hope the experiences we have outlined here will benefit others who attempt real-time procedural shading.

8 ACKNOWLEDGMENTS

PixelFlow was a joint project of the University of North Carolina and Hewlett-Packard and was supported in part by DARPA order numbers A410 and E278, and NSF grant numbers MIP-9306208 and MIP-9612643.

The entire project team deserves recognition and thanks; this work exists by virtue of their labors. We would like to single out Voicu Popescu for his work on pfman memory allocation as well as the other project members who worked on the pfman compiler, Peter McMurry and Rob Wheeler. Thanks to Steve Molnar and Yulan Wang for their early work on programmable shading on PixelFlow. Thanks to Jon Leech for his work on the OpenGL extensions. We would also like to express special thanks to the other people who worked on the PixelFlow shading system and the API extensions: Dan Aliaga, Greg Allen, Jon Cohen, Rich Holloway, Roman Kuchkuda, Paul Layne, Carl Mueller, Greg Pruett, Brad Ritter, and Lee Westover.

Finally, we would like to gratefully acknowledge the help and patience of those who have used pfman, and provided several of the shaders used in this paper. They are Arthur Gregory, Chris Wynn, and members of the UNC nanoManipulator project, under the direction of Russ Taylor (Alexandra Bokinsky, Chun-Fa Chang, Aron Helser, Sang-Uok Kum, and Renee Maheshwari).

References

[Briggs92] Preston Briggs, *Register Allocation via Graph Coloring*, PhD Dissertation, Department of Computer Science, Rice University, Houston, Texas, 1992.

[Cook84] Robert L. Cook, “Shade Trees”, Proceedings of SIGGRAPH 84 (Minneapolis, Minnesota, July 23–27, 1984). In *Computer Graphics*, v18n3. ACM SIGGRAPH, July 1984. pp. 223–231.

[Cook87] Robert L. Cook, “The Reyes Image Rendering Architecture”, Proceedings of SIGGRAPH 87 (Anaheim, California, July 27–31, 1987). In *Computer Graphics*, v21n4. ACM SIGGRAPH, July 1987. pp. 95–102.

[Deering88] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy and Neil Hunt, “The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics”, Proceedings of SIGGRAPH 88 (Atlanta, Georgia, August 1–5, 1988). In *Computer Graphics*, v22n4, ACM SIGGRAPH, August 1988. pp. 21–30.

[Dietz92] Henry G. Dietz, “Common Subexpression Induction”, Proceedings of the 1992 International Conference on Parallel Processing (Saint Charles, Illinois, August 1992). pp. 174–182.

[Ellsworth91] David Ellsworth, “Parallel Architectures and Algorithms for Real-time Synthesis of High-quality Images using Deferred Shading”. Workshop on Algorithms and Parallel VLSI Architectures (Pont-á-Mousson, France, June 12, 1990).

[Eyles97] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, Nick England and Lee Westover, “PixelFlow: The Realization”, Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware (Los Angeles, California, August 3–4, 1992). ACM SIGGRAPH, August 1997. pp. 57–68.

[Gritz96] Larry Gritz and James K. Hahn, “BMRT: A Global Illumination Implementation of the RenderMan Standard”, *Journal of Graphics Tools*, v1n3, 1996. pp. 29–47.

[Guenter95] Brian Guenter, Todd B. Knoblock and Erik Ruf, “Specializing Shaders”, Proceedings of SIGGRAPH 95 (Los Angeles, California, August 6–11, 1995). In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, 1995. pp. 343–348.

[Hanrahan90] Pat Hanrahan and Jim Lawson, “A Language for Shading and Lighting Calculations”, Proceedings of SIGGRAPH 90 (Dallas, Texas, August 6–10, 1990). In *Computer Graphics*, v24n4. ACM SIGGRAPH, August 1990. pp. 289–298.

[Hill97] B. Hill, Th. Roger and F. W. Vorhagen, “Comparative Analysis of the Quantization of Color Spaces on the Basis of the CIELAB Color-Difference Formula”, *ACM Transactions on Graphics*, v16n2. ACM, April 1997. pp. 109–154.

[Lastra95] Anselmo Lastra, Steven Molnar, Marc Olano and Yulan Wang, “Real-time Programmable Shading”, Proceedings of the 1995 Symposium on Interactive 3D Graphics (Monterey, California, April 9–12, 1995). ACM SIGGRAPH, 1995. pp. 59–66.

[Leech98] Jon Leech, “OpenGL Extensions and Restrictions for PixelFlow”, Technical Report TR98-019, Department of Computer Science, University of North Carolina at Chapel Hill.

[MasPar90] MasPar Computer Corporation, MasPar Parallel Application Language (MPL) User Guide, 1990.

[Max81] Nelson L. Max, “Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset”, Proceedings of SIGGRAPH 81 (Dallas, Texas, July 1981). In *Computer Graphics*, v15n3. ACM SIGGRAPH, August 1981. pp. 317–324.

[Molnar92] Steven Molnar, John Eyles and John Poulton, “PixelFlow: High-speed Rendering Using Image Composition”, Proceedings of SIGGRAPH 92 (Chicago, Illinois, July 26–31, 1992). In *Computer Graphics*, v26n2. ACM SIGGRAPH, July 1992. pp. 231–240.

[Muchnick97] Steven Muchnick, *Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[Neider93] Jackie Neider, Tom Davis and Mason Woo, *OpenGL Programming Guide: the official guide to learning OpenGL release 1*., Addison-Wesley, 1993.

[Olano98] Marc Olano, *A Programmable Pipeline for Graphics Hardware*, PhD Dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, 1998.

[Perlin85] Ken Perlin, “An Image Synthesizer”, Proceedings of SIGGRAPH 85 (San Francisco, California, July 22–26, 1985). In *Computer Graphics*, v19n3. ACM SIGGRAPH, July 1985. pp. 287–296.

[Pixar97] Pixar Animation Studios, *PhotoRealistic RenderMan 3.7 Shading Language Extensions*. Pixar animation studios, March 1997.

[Rhoades92] John Rhoades, Greg Turk, Andrew Bell, Andrei State, Ulrich Neumann and Amitabh Varshney, “Real-time procedural textures”, Proceedings of the 1992 Symposium on Interactive 3D Graphics (Cambridge, Massachusetts, March 29–April 1, 1992). In *Computer Graphics* special issue. ACM SIGGRAPH, March 1992. pp. 95–100.

[Slusallek94] Philipp Slusallek, Thomas Pflaum and Hans-Peter Seidel, “Implementing RenderMan–Practice, Problems and Enhancements”, Proceedings of Eurographics ’94. In *Computer Graphics Forum*, v13n3, 1994. pp. 443–454.

[ThinkingMachines89] Thinking Machines Corporation, *Connection Machine Model CM-2 Technical Summary*. Thinking Machines Corporation, Version 5.1, May 1989.

[Upstill90] Steve Upstill, *The RenderMan Companion*, Addison-Wesley, 1990.

[Whitted81] T. Whitted and D. M. Weimer, “A software test-bed for the development of 3-D raster graphics systems”, Proceedings of SIGGRAPH 81 (Dallas, Texas, July 1981). In *Computer Graphics*, v15n3. ACM SIGGRAPH, August 1981. pp. 271–277.

Graphics Hardware (2003)
M. Doggett, W. Heidrich, W. Mark, A. Schilling (Editors)

Automatic Shader Level of Detail

Marc Olano,* Bob Kuehne† and Maryann Simmons†

* University of Maryland, Baltimore County
† SGI

Abstract

Current graphics hardware can render procedurally shaded objects in real-time. However, due to resource and performance limitations, interactive shaders can not yet approach the complexity of shaders written for film production and software rendering, which may stretch to thousands of lines. These constraints limit not only the complexity of a single shader, but also the number of shaded objects that can be rendered at interactive rates. This problem has many similarities to the rendering of large models, the source of extensive research in geometric simplification and level of detail. We introduce an analogous process for shading : shader simplification. Starting from an initial detailed shader, shader simplification automatically produces a set of simplified shaders or a single new shader with extra level-of-detail parameters that control the shader execution. The resulting level-of-detail shader can automatically adjust its rendered appearance based on measures of distance, size, or importance, as well as physical limits such as rendering time budget or texture usage. We demonstrate shader simplification with a system that automatically creates shader levels of detail to reduce the number of texture accesses, one common limiting factor for current hardware.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image generation I.3.6 [Computer Graphics]: Methodology and Techniques I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism
Keywords: *Interactive Rendering, Rendering Systems, Hardware Systems, Procedural Shading, Languages, Multi-Pass Rendering, Level of Detail, Simplification, Computer Games, Reflectance & Shading Models*

1. Introduction

Procedural shading is a powerful technique, first explored for software rendering in work by Cook[11] and Perlin[35], and popularized by the RenderMan Shading Language[19]. A shader is a procedure written in a special purpose high-level language that controls some aspect of the appearance of an object to which it is applied. The term *shader* is used generically to refer to any such procedures, whether they compute surface color, attenuation of light through a volume (as with fog), light color and direction, fine changes to the surface position, transformation of control points or vertices, or any combination of these factors.

Recent graphics hardware can render procedurally shaded objects in real-time, through shaders defined in a low-level assembly language[4, 30, 40] or a high-level shading language[27, 33, 34, 36]. Even though the hardware is capable of rendering these shaders interactively, the number of texture units, total texture memory used, number of instructions, or other factors can affect overall performance or prevent a shader from running at all. Even on programmable PC graphics hardware, it is easy to exceed the hardware's abilities for rendering of a single object. Such shaders may be rendered using multiple passes through the graphics pipeline, but choosing the partitioning into passes is a dif-

Figure 1: *Shader simplification applied to a leather shader.*

ficult compilation task and the final number of passes has a direct impact on performance[5, 8, 34].

Consider the leather shader in Figure 1. With a bump map requiring three texture accesses per light and homomorphic BRDF factorization[28] requiring two texture accesses per light plus one additional texture access, it is complex enough to benefit from the several automatically generated simplification steps shown. An even more realistic leather shader might include multiple measured BRDFs for worn and unworn areas, bumps for the stitching, dust collected in the crevices, scuff marks, and changes in color due to variations in the leather. The options are limited only by the imagination and skill of the shader writer. But even though such a complex shader might look good applied to a single closely examined chair, it is overkill as you move away to see the rest of the room, all the other furniture in the room, other buildings, trees, cars and pedestrians — all using shaders of similar or greater complexity.

In this paper, we introduce the automatic generation of level-of-detail shaders (LOD shaders) from arbitrarily complex shaders. Our examples use SGI OpenGL Shader running on an SGI Octane[41]. From each input shader, our system automatically creates a single parameterized level-of-detail shader that can adjust the shading complexity (and thus number of rendering passes produced) based on a level-of-detail input parameter. The application sets the level parameter to control detail for the current viewing conditions and resource limits, thus allowing both interactive performance and high-quality shading of many objects in the same scene. The methods described in this paper could also be applied to produce a series of shaders for an application to select, or could be adapted for simplification of shaders on commodity graphics hardware as suggested by Vlachos[45].

1.1. Background

Automatic transformation of non-procedural surface appearance has been explored by a number of researchers. Kajiya was the first to pose the problem of converting large-scale surface characteristics to a bump map then lighting model[22]. Fournier used nonlinear optimization to fit a bump map to a sum of several standard Phong *peaks*[13]. Cabral, et al. addressed the conversion from bump map to BRDF through a numerical integration pre-process[7], and Becker and Max solved it for conversion from RenderMan-based displacement maps to bump maps and then to a BRDF representation[6]. Kautz approached the problem in reverse, creating bump maps to statistically match a chosen fractal micro-facet BRDF[24].

Fewer researchers have attempted to tackle automatic antialiasing of arbitrary shading language code. The primary form of antialiasing provided in the RenderMan shading language is manual transformation of the shader, relying on the shader-writer's knowledge to effectively remove high-frequency components of the shader or smooth the sharp transitions from an `if`, by instead using a `smoothstep` (cubic spline interpolation between two values) or `filterstep` (smoothstep across the current sample width)[12]. Perlin describes automatic use of blending wherever `if` is used in the shading code[12]. Heidrich et al. also did automatic antialiasing, using affine arithmetic to estimate the frequency and error while computing shading results[20].

Thus far, creation of shaders at multiple levels of detail for rendering speed or computational efficiency has been primarily a manual process. Goldman manually created multiple independently written level-of-detail versions of a fur shader for movie production[17]. Apodaca and Gritz describe several general options for manually creating shaders with multiple complexity levels[3]. Olano and Kuehne provided a set of building block functions with manually created levels of detail, so shaders using these building blocks inherit those levels of detail[32]. Guenter et al. automatically created *specialized shaders*, when only some shader parameters were expected to vary[18]. Expressions using other parameters were evaluated into textures.

While most of this prior work is in the context of off-line rendering systems like RenderMan, our work is set specifically within the context of recent advances in *interactive* shading languages. The first interactive shading system was a low-level assembly-like language for the Pixel-Planes 5 machine at UNC[38]. Later work at UNC developed a full interactive shading language on UNC's PixelFlow system[33]. Peercy and coworkers at SGI created a shading language that runs using multiple OpenGL Rendering passes[34]. The Real-Time Shading group at Stanford has created another high-level shading language, RTSL, that can be compiled into one or more rendering passes on SGI, NVIDIA, or ATI hardware[8, 36]. Many aspects of these research efforts appear in the several recent commercial shading languages and compilers, NVIDIA's Cg language, Microsoft's HLSL, ATI's Ashli, and the OpenGL shading language[27, 29, 5, 25].

Several aspects of interactive shading languages motivate the need for shader simplification and level-of-detail shaders. The languages they use share some features with traditional shading languages like RenderMan, but tend to be simpler, with operations that graphics hardware can and cannot do a major factor in their design. Hardware limits bound shader complexity and encourage the use of results precomputed into textures. Both of these factors make the simplification problem more tractable. Additionally, the desire to have the appearance of high-quality shaders on every object creates the need for shaders that can transition smoothly from high quality to fast rendering while maintaining interactive frame rates.

2. Automatic Simplification

Shader simplification automatically creates multiple levels of detail from an arbitrary source shader. Our simplification

process draws on two major areas of prior work — geometric simplification and compiler optimization.

Specifically, our shader simplification strategy is modeled after operations from the topology-preserving geometric level-of-detail literature. Schroeder and Turk both performed early work in automatic mesh simplification using a series of local operations, each resulting in a smaller total polygon count for the entire model[39, 44]. Hoppe used the collapse of an edge to a single vertex as the basic local simplification operation. He also introduced progressive meshes, where all simplified versions of a model are stored in a form that can be reconstructed to any level at run-time[21]. These ideas have had a large influence on more recent polygonal simplification work[26].

From this work we take several desired properties for our shader simplification algorithm. It should perform only local simplification operations for computational efficiency. Each operation should move monotonically toward the goal. Each simplification operation has an associated cost and the simplification of lowest remaining cost should be chosen at each step. The outline of our algorithm becomes:

```
for each candidate simplification site
  find simplification cost
while (simplifications remain)
  choose site with lowest remaining cost
  perform simplification
  re-compute costs for area local to site
```

The second area we draw on in developing our simplification algorithm is classic compiler *peephole optimization*[1]. Peephole optimization occurs toward the end of the compilation process when the program has already been reduced to blocks of simple instructions. The optimizer looks at small windows of instructions for certain patterns to replace.

Peephole optimization performs only local operations. If several optimizations overlap the optimizer will choose between them based on a set of costs. The golden rule for optimizations is to never change the program output. Shader simplification is in effect an optimization process but is one that may or may not break this rule. We can classify the simplifications into three categories:

Lossless: Obeying the strict compiler definition of optimization. This can be expanded to include some specializing shader-like optimizations[18] where certain non-constant parameters are assumed to be constant for the simplification. The geometric level of detail equivalent is simplification of highly tessellated, but flat regions of a model.

Resolution-specific lossless: Producing numerically or visually identical results but only at a specific resolution. This would include the majority of specializing shader simplifications and any others that evaluate results into textures. The equivalence is dependent on the texture resolution and minimum viewing distance. It also includes simplifications that replace textures with computed results, where the computed results fit a specific MIP-level of the texture. In geometric simplification, the equivalent is Simplification Envelopes or Appearance Preserving Simplification, with strong guarantees on geometric deviation[9, 10]

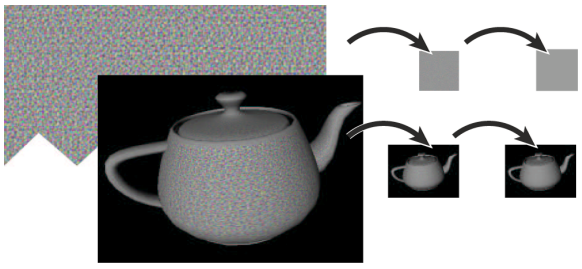


Figure 2: *Band-limited noise texture, noise **almost** blended away at a distance, and noise replaced with average value.*

Lossy: Not identical, but not noticeably or objectionably different. This includes many approximations that would never be considered for traditional optimization, but produce visually similar results at lower cost. Most geometric simplifications would fall in this category, as they minimize visual impact without making any guarantees, and assume slight changes in shape for distant objects are acceptable in exchange for interactive performance.

2.1. Simplifications

One of the most severe restrictions of current hardware is the cost of each texture access, with limits on either accesses or active textures per rendering pass in the tens at most.

Shaders that make heavy use of textures for precomputed expressions, for math and shading functions, and as actual textures can easily exceed these limits[8]. We have chosen the reduction of texture accesses as our simplification goal. Reduction by multiples of the single-pass hardware texturing limit provides an obvious speedup by reducing the number of passes required, but reductions by less than the single pass limit can also be beneficial as some hardware has higher rendering rates if fewer texture units are used, and fewer texture accesses indirectly leads to fewer operations and fewer active textures.

Many geometric simplifications use a single simplification rule, for example collapsing an edge to a point. We proceed using a choice of two simplification rules. The first is a lossy simplification that replaces a texture access with a simple non-texture-based approximation (Figure 2); the second is a lossless simplification that replaces one or more textures accesses and other operations with a single texture (Figure 3).

Texture Removal: Our first simplification rule clearly moves us toward the goal as it results in the direct removal of one texture access at each application. Our measure of

error for this simplification is the least-squares difference between the texture and non-texture approximation at each MIP-MAP level. The use of scale-based MIP filtering introduces a frequency and distance factor, while the least-squares error provides a measure of the contrast between pre- and post-simplification representations.

In the work presented here, we only approximate a texture by its average color. The most blurred level of a MIP map is just the average color, so applying this simplification switches to this constant color earlier than would be done by standard MIP filtering. For example, replacing

```
FB *= texture("marble.tx");
```

with

```
FB *= color(.612,.618,.607,1);
```

The least-squares error between texture and average color is the standard deviation of the texture, but we prefer the least-squares error interpretation since it generalizes more easily for future approximations. For example, an environment map may be well approximated by one or more light sources using the built-in Phong model, with light sources located at peaks of the environment map. The least-squares error between environment map and Phong lighting measures the error in this approximation, with lower error expected at larger MIP levels due to the closer fit of the approximation to the texture.

While the texture removal operation alone is theoretically sufficient to eventually remove all texturing operations from any complex shader, it does not always reduce texture uses as quickly as should be possible. The problem is not the local error metric, but with the global effect of the introduced error. Subsequent operations using the removed texture may amplify or attenuate the computed error. Directly computing this propagation of error can be done, as was shown in the sampling of RenderMan shaders by Heidrich et al.[20]. In our experience, global error measures were not necessary, even for shaders that did have fairly significant amplification of texture results like the watermelon in Plate 2(a). We attribute this to the common practice of writing complex shaders in layers[3].

Texture Collapse: Our second simplification rule is the collapse of one or more textures and the operations performed onto them into a single new texture. We guarantee to never increase the total texture accesses by including at least one existing texture in any set of collapsed operations.

Transformation of textures within a collapse, as illustrated in Figure 3, introduces resolution-specific error through resampling of the source textures into the collapse texture. In our current version, we support only lossless collapse (with no relative rotation or scaling of either texture).

By limiting ourselves to lossless collapse, texture collapse operations will happen immediately, at no additional cost.

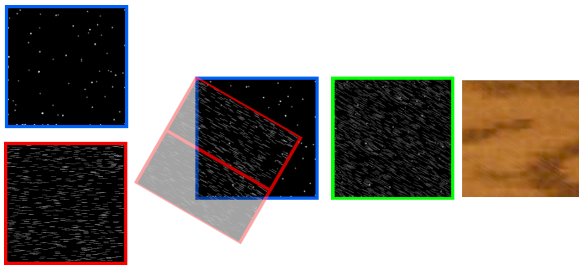


Figure 3: *An illustration of the collapse of two textures and the associated computations into a single texture. Left to right: the original dust (top) and scratch (bottom) textures; the textures as transformed and overlaid by the shader (the scratch texture is compressed, rotated and repeated, only two copies of the repeated texture are shown); the collapsed single-texture result; and an example of the collapsed texture in use as dust and scratch wood detail.*

However, texture removals at one scale may enable further collapses. For example:

```
FB =  texture("silk.tx");
FB *=  texture("cone.tx");
FB += color(0.1,0,0,0);
FB *= environment("flowers.env");
```

is reduced first by texture collapse creating a new temporary texture `loctx_0_silk.tx` (naming of generated textures is explained in Section 3.1), producing

```
FB = texture("loctx_0_silk.tx");
FB *= environment("flowers.env");
```

then by texture removal to

```
FB = texture("loctx_0_silk.tx");
FB *= color(.264,.238,.279,1);
```

then by a second collapse creating new temporary texture `loctx_1_loctx_0_silk.tx` producing

```
FB = texture("loctx_1_loctx_0_silk.tx");
```

and finally by texture removal to

```
FB = color(.111,.076,.090,1);
```

2.2. LOD Shader Representation

While each simplified block could be provided as a single stand-alone unit, we assemble all simplified blocks for a shader into a single unit, the LOD shader. We replace the full shader with an `if`. The true branch is the original shader while the false branch is the shader after one step of simplification. We iterate the simplification process on this false branch producing an LOD shader of the following form:

```
if(autoLOD < threshold0)
  original_shader
else
```

```
if(autoLOD < threshold1)
  simplified_once
else
  simplified_twice
```

Within OpenGL Shader, such parameter-based conditionals control which portions of the shader are executed by the hardware. The threshold levels monotonically increase with each level of simplification and provide a simple means to choose between levels of detail within the shader itself. The resulting LOD shader can be directly substituted as a replacement for the original shader. If `autoLOD` is not set, the original shader will be executed every time, but if `autoLOD` is set the appropriate level of detail will be used instead.

The existence of level-control parameters are the one aspect that distinguishes the interface to an LOD shader from other shaders. We control our LOD levels through the single parameter, `autoLOD`. This parameter represents the degree of texture scaling and is a function of object size, object parameterization, and object distance. As with geometric level of detail, other parameter choices are possible, including object importance, distance, size, time budget, or any of the hardware resources mentioned above. Several of these parameters could be combined into more complex conditionals selecting simplified blocks, collected into a single aggregate parameter, or controlled through an optimization function as done by Funkhouser and Séquin[15].

3. System Design

The bulk of this paper has focused on shader simplification and creation of levels of detail for arbitrary shaders. These capabilities must fit into the larger context of a shading system. In this section of the paper we will explore how our shading system architecture has been modified to allow both generation and usage of automatic level of detail.

3.1. Compile and Simplify

The first step for using any interactive shader is to compile it into a form executable by the graphics hardware. During the compilation we also perform any simplifications. Simplifications are only performed on shaders that define an *autoLOD* parameter. The existence of this parameter triggers use of the simplifier.

The simplification process also needs to know the size and contents of each texture, information not normally needed during shader compilation. Our system leaves the application in control of all aspects of texture loading and paging, so we require an application-provided image data *callback* function to get this data. The simplifier may call the image data callback during compilation to get a copy of texture data to analyze. Textures are identified to this callback by their string name. The callback can return the texture data or an error code indicating that the texture is unknown or dynamic and cannot be removed or simplified.

Texture collapse operations may require new textures to hold the combined textures and operations. Since the application is in charge of texture allocation and paging, we ask for image data for a *local texture* with a name beginning `loctx_%d_`. For example `loctx_1_stone` would be a copy of a texture named `stone` that the simplifier is free to write and replace. Later requests for `loctx_1_stone` should return the modified data (for analysis for texture removal or further collapse). Since a texture collapse may build on a previous collapse, these names may also build, so `loctx_5_loctx_1_stone` is a writable copy of `loctx_1_stone`.

3.2. Between Frames

The LOD shader may use different active textures on different frames depending on which level of detail is in use. This is not inconsistent with our goal of reducing texture accesses rather than global texture memory use, but many applications already use enough textures to require some form of texture paging. Adding an additional set of generated textures to that burden may be a problem for these applications.

We provide an optional snapshot function that an application can call between frames. The snapshot evaluates all run-time parameters and conditionals in the shader to provide a frozen version the application can store and use. In the process of building the snapshot, the application can find out exactly which textures will be used for a given set of run-time parameters, including the `autoLOD` setting. The application does not need to use the frozen shader that results if it only wants to know future texture usage. It can take a snapshot just one frame in advance or compute several speculatively to page textures for possible future views.

3.3. Draw and Shade

The final shaded object is drawn by the same mechanisms as any unsimplified object. If the application does not set the `autoLOD` parameter, it assumes the default value which triggers the full unsimplified shader. If the application does set an `autoLOD` value, the appropriate level of detail will be selected and executed. Applications using frozen snapshots must set their `autoLOD` values before taking the snapshot.

During the drawing of the shaded object, different textures may need to be loaded and bound to texture units for rendering. The draw action indicates which textures to load by calling an application provided *texture bind* callback function. Like the image data function, this function identifies textures by their string name. The texture bind callback also indicates the texture unit to bind to the texture (if the hardware supports multiple textures in each rendering pass). It is then the application's responsibility to load or page in the texture if necessary and prepare it for use. The texture names may be one of the names from the original shader source code or one of the generated `loctx` textures.

LOD	*Active*	*Accesses*	*Reduction*	*Speedup*
0	14	45	0.00	1.0
1	11	23	0.49	1.8
2	5	9	0.80	1.9
3	0	0	1.00	2.3

Table 1: *Results for test scene:* ***LOD****: A selection of simplification levels for this scene, from most detailed (0) to all constant colors (3).* ***Active****: number of active, unique textures.* ***Accesses****: number of texture accesses.* ***Reduction****: percentage of texture accesses removed.* ***Speedup****: framerate speedup factor.*

4. Results

We ran the automatic simplification on a number of shaders, all of which were written independently from our work on shader LOD. Once the user enables simplification by including the `autoLOD` parameter, the process is entirely automatic.

Results are shown in Plates 1 and 2, with performance results for Plate 1(a) shown in Table 1. As these results show, the automatically generated levels of detail are visually comparable to the fully detailed version at the appropriate viewing distances, at a significant reduction in texture accesses. Even further reductions could be achieved within the current framework by allowing more aggressive texture collapse.

5. Discussion

Using a single LOD shader that encapsulates the progression of levels of detail provides many of the advantages for simplified shaders that progressive meshes provide for geometry. In this section, we directly echo the points from Hoppe's original progressive mesh paper[21]. Not only does this place our current system in context, but it also suggests some logical extensions and more ambitious future work.

- *Shader simplification*: The LOD shader can be generated automatically from an initial complex shader using automatic tools. Our shader simplifier operates with the sole goal of reducing the number of texture accesses. Other valid simplification goals may include texture memory used, instruction count, balance between direct textures and dependent textures, or a weighted combination of these. Reducing texture accesses also indirectly reduces the number of active textures and instruction count, and so is relevant across a wide range of hardware.
- *LOD approximation*: Like a progressive mesh, an LOD shader contains all levels of detail. Thus it could include the shader equivalent of Hoppe's *geomorphs* to smoothly transition from one level to the next. Within OpenGL Shader, we have implemented continuous, per-pixel LOD at the cost of an additional pass that renders the object texture-mapped with a special MIP LOD texture that approximates the sampling rate of the shader[43]. The result is read back and used to set a per-pixel LOD level, that can also be used to smoothly blend between levels.
- *Selective Refinement*: Selective refinement for meshes refers to simplifying some portions of the mesh more than others based on current viewing conditions, encompassing both variation across the object and a guided decision on which of the stored simplifications to apply. Within OpenGL Shader, we can treat per-pixel LOD as noted above[43]. Programmable PC hardware does not realize any benefit from shading variations across a single object, but a single LOD shader will present a high quality appearance on some surfaces while using a lower quality for others, based on distance, viewing angle or other factors. The LOD shader could also apply certain simplifications and not others based on pressure from hardware resource limits, though our current implementation does not. For example, if available texture memory is low, texture-reducing simplification steps may be applied in one part of the shader while leaving more computation-heavy portions of the shader to be rendered at full detail.
- *Retargetability*: Retargetability is not found in mesh simplification. Since shading simplification can be built into a shading compiler, it gains the advantages of the compiler framework. Compilers consist of a sequence of modules that perform a simple operation on an intermediate representation of the shader. Since simplification can be dropped in as one or more modules in the chain, it is easy to add to existing shading compilers and easy to add new simplification modules. Further, since the shading compiler can be retargeted through multiple compiler *backends* to different shading hardware, it is easy to create simplifications for one hardware platform and use them on another.

Many of these points depend on the storage of an LOD shader. Our choice to combine all levels into a single LOD shader would work well for most of the points mentioned, with the added advantage that LOD shaders can easily be dropped in as replacements for their non-LOD counterparts.

6. Conclusions and Future Work

We have presented a method for automatic simplification of complex procedural shaders designed for use on graphics hardware. The resulting LOD shaders automatically adjust their level of shading detail for interactive rendering. We presented a general strategy for shader simplification, a specific example for reducing texture accesses, and a system that provides a shader compiler and shader simplification to an application.

6.1. Other Simplification Goals

The two simplifications discussed are specific to our goal to reduce the number of texture accesses. Future work may

optimize other simplification goals, including the previously suggested options of reducing total number of instructions or texture memory used. We have not fully explored simplification operations appropriate for these other simplification goals, but some directions inspired by prior research appear particularly promising.

Texture-based simplification for both shaders and geometry provides examples of ways to move computations into an increased number of textures. Guenter, Knoblock and Ruf[18] replaced static sequences of shading operations with pre-generated textures[18]. Heidrich has analyzed texture sizes and sampling rates necessary for accurate evaluation of shaders into texture[31]. In a related vein, texture-impostor based simplification techniques replace geometry with pre-rendered textures, either for indoor scenes as has been done by Aliaga[2] or outdoor scenes as by Shade et al.[42].

The body of BRDF approximation methods also suggests approaches to reduce computation at the cost of increased numbers of textures. Like shading functions, BRDFs are positive everywhere. Fournier used singular value decomposition (SVD) to fit a BRDF to sums of products of functions of light direction and view direction for use in radiosity[14]. Kautz and McCool presented a similar method for real-time BRDF rendering, computing functions of view direction, light direction, or other basis as textures using either SVD or a simpler normalized integration method[23]. McCool, Ang and Ahmad's homomorphic factorization uses only products of 2D texture lookups, fit using least-squares[28]. In a related area, Ramamoorthi and Hanrahan used a common set of spherical harmonic basis textures for reconstructing irradiance environment maps[37]. Many of these could be generalized to approximate blocks of shading code, which can be seen as a black-box producing a result from an arbitrary number of input variables.

6.2. Going Further

There are other promising overall research directions for shader simplification. Following the lead of texture-based simplification researchers like Aliaga and Shade et al., we could generate new textures for run-time parameter-dependent texture collapse or other simplification on the fly, warping them for use over several frames or updating when they become too different[2, 42].

Since rendering with LOD shaders will usually be accompanied by geometric level of detail, the two should be more closely linked. Cohen et al.[9], Garland and Heckbert[16] and others have shown that geometric simplification can be driven by appearance. Shader simplification should also be affected by geometric level of detail, with a trade-off between performing the same operation per-vertex or per-pixel depending on object tessellation.

Finally, our error metric measures the actual error in each replacement but provides no hard guarantees on the perceptual fidelity of our simplifications. Many geometric simplification algorithms have been successful without providing exact error metrics or bounds. However, algorithms such as simplification envelopes by Cohen et al.[10] provide hard bounds on the amount of error introduced by a simplification — guarantees that are important for some users. Further investigation is necessary to bound the error introduced by shader simplification.

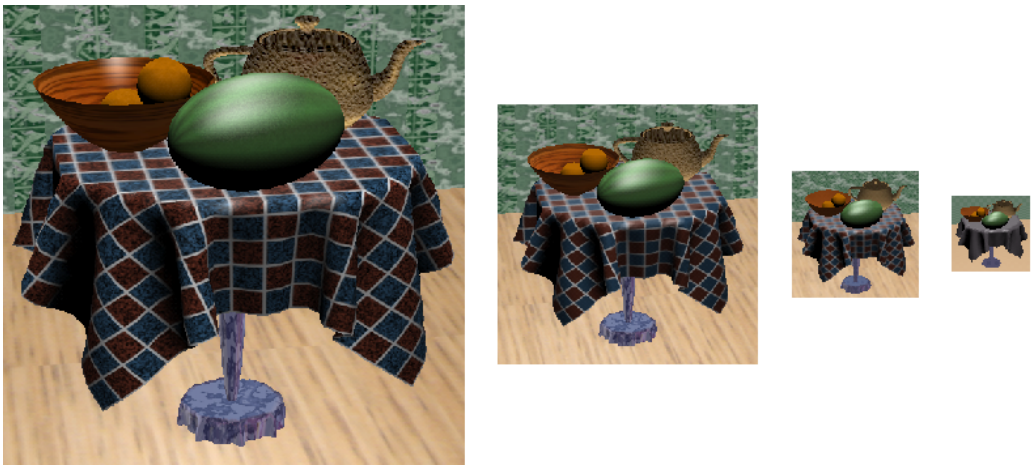
7. Acknowledgments

The leather BRDF was fit by homomorphic factorization by Michael McCool to data from the Columbia-Utrecht Reflectance and Texture Database. The car paint BRDF is also from Michael McCool, fit to data for Dupont Cayman lacquer from the Ford Motor Company and measured at Cornell University. The Porsche model was distributed by 3dcafe.com.

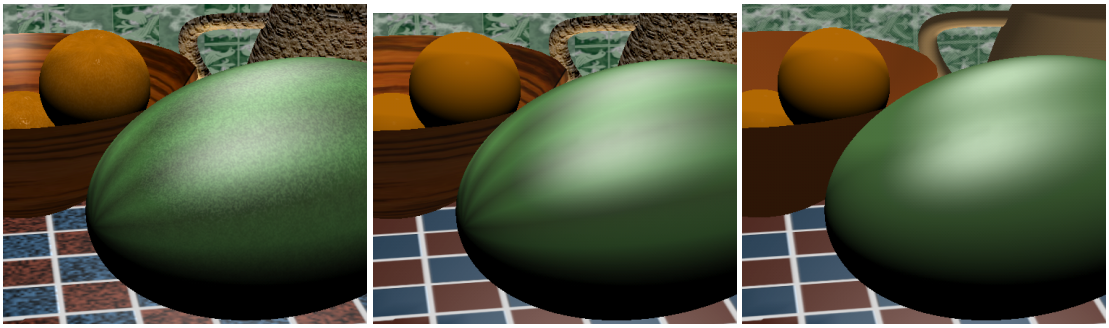
References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. D. G. Aliaga. Visualization of complex models using dynamic texture-based simplification. In *IEEE Visualization '96*, pages 101–106, October 1996.
3. A. A. Apodaca and L. Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann, first edition, 2000.
4. ATI. *ATI OpenGL Extensions Specifications*, 2001.
5. ATI. Ashli demo. http://www.ati.com, 2003.
6. B. G. Becker and N. L. Max. Smooth transitions between bump rendering algorithms. In *Proceedings of SIGGRAPH 93*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 183–190, August 1993.
7. B. Cabral, N. Max, and R. Springmeyer. Bidirectional reflection functions from surface bump maps. In *ACM Computer Graphics (Proceedings of SIGGRAPH 87)*, pages 273–281, July 1987.
8. E. Chan, R. Ng, P. Sen, K. Proudfoot, and P. Hanrahan. Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In *Graphics Hardware 2002*. ACM SIGGRAPH / Eurographics, August 2002.
9. J. Cohen, M. Olano, and D. Manocha. Appearance-preserving simplification. In *Proceedings of SIGGRAPH 98*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 115–122, July 1998.
10. J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks Jr., and W. Wright. Simplification envelopes. In *Proceedings of SIGGRAPH 96*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 119–128, August 1996.
11. R. L. Cook. Shade trees. In *ACM Computer Graphics (Proceedings of SIGGRAPH 84)*, pages 223–231, July 1984.

12. D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, second edition, 1998.
13. A. Fournier. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination*, pages 45–52, May 1992.
14. A. Fournier. Separating reflection functions for linear radiosity. In *Proceedings of Eurographics Workshop on Rendering*, pages 296–305, June 1995.
15. T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of SIGGRAPH 93*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 247–254, August 1993.
16. M. Garland and P. S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *IEEE Visualization '98*, pages 263–270, October 1998.
17. D. B. Goldman. Fake fur rendering. In *Proceedings of SIGGRAPH 97*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 127–134, August 1997.
18. B. Guenter, T. B. Knoblock, and E. Ruf. Specializing shaders. In *Proceedings of SIGGRAPH 95*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 343–350, August 1995.
19. P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *ACM Computer Graphics (Proceedings of SIGGRAPH 90)*, pages 289–298, August 1990.
20. W. Heidrich, P. Slusallek, and H. Seidel. Sampling procedural shaders using affine arithmetic. 17(3):158–176, July 1998.
21. H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH 96*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 99–108, August 1996.
22. J. T. Kajiya. Anisotropic reflection models. In *ACM Computer Graphics (Proceedings of SIGGRAPH 85)*, pages 15–21, July 1985.
23. J. Kautz and M. D. McCool. Interactive rendering with arbitrary BRDFs using separable approximations. In *Eurographics Rendering Workshop*, June 1999.
24. J. Kautz and H. Seidel. Towards interactive bump mapping with anisotropic shift-variant BRDFs. pages 51–58. ACM SIGGRAPH / Eurographics / ACM Press, August 2000.
25. J. Kessenich, D. Baldwin, and R. Rost. *The OpenGL Shading Language*. 3Dlabs, Inc. Ltd., February 2003. Version 1.05.
26. D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann / Elsevier Science, 2003.
27. W. R. Mark, S. Glanville, and K. Akeley. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003)*, 22(3), August 2003.
28. M. D. McCool, J. Ang, and A. Ahmad. Homomorphic factorization of BRDFs for high-performance rendering. In *Proceedings of SIGGRAPH 2001*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 171–178, August 2001.
29. Microsoft. *DirectX Graphics Programmers Guide*. Microsoft Developers Network Library, DirectX 9 edition, 2002.
30. NVIDIA. *NVIDIA OpenGL Extensions Specifications*, March 2001.
31. M. Olano, J. C. Hart, W. Heidrich, E. Lindholm, M. McCool, B. Mark, and K. Perlin. Real-time shading. In *ACM SIGGRAPH 2001 Course Notes*, August 2001.
32. M. Olano, J. C. Hart, W. Heidrich, and M. McCool. *Real-time Shading*. AK Peters, 2002.
33. M. Olano and A. Lastra. A shading language on graphics hardware: The PixelFlow shading system. In *Proceedings of SIGGRAPH 98*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 159–168, July 1998.
34. M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In *Proceedings of SIGGRAPH 2000*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 425–432, July 2000.
35. K. Perlin. An image synthesizer. In *ACM Computer Graphics (Proceedings of SIGGRAPH 85)*, pages 287–296, July 1985.
36. K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of SIGGRAPH 2001*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 159–170, August 2001.
37. R. Ramamoorthi and P. Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of SIGGRAPH 2001*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 497–500, August 2001.
38. J. Rhoades, G. Turk, A. Bell, A. State, U. Neumann, and A. Varshney. Real-time procedural textures. In *1992 Symposium on Interactive 3D Graphics*, pages 95–100. ACM, March 1992.
39. W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. In *ACM Computer Graphics (Proceedings of SIGGRAPH 92)*, pages 65–70, July 1992.
40. M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification*. SGI, 2002.
41. SGI. OpenGL shader. http://www.sgi.com/software/shader, 2003. Version 3.0.
42. J. Shade, D. Lischinski, D. H. Salesin, T. D. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proceedings of SIGGRAPH 96*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 75–82, August 1996.
43. M. Simmons and D. Shreiner. Per-pixel smooth shader level of detail. In *Computer Graphics (Conference Abstracts and Applications SIGGRAPH 2003)*, 2003.
44. G. Turk. Re-tiling polygonal surfaces. In *ACM Computer Graphics (Proceedings of SIGGRAPH 92)*, pages 55–64, July 1992.
45. A. Vlachos. Designing a portable shader library for current and future APIs. In *Game Developers Conference Presentation*, March 2003. http://www.ati.com/developer/gdc/GDC2003-ShaderLib.pdf.



(a) A selection of LOD Levels for this scene (0-3) at typical viewing distances. Performance numbers in Table 1.



(b) Close-up of level 0: Highest level of detail.

(c) Close-up of level 1: Various noise and highlight details have been removed.

(d) Close-up of level 2: Extra detail on wall, bowl, watermelon and teapot removed.

Plate 1: *A simple scene showing the interaction of multiple automatically simplified level-of-detail shaders.*

(a) A slightly different watermelon shader

(b) A car shader

(c) A tile shader

Plate 2: *Individual examples of shader simplification.*