Chapter 7

Shading Through Multi-Pass Rendering

Marc Olano

Interactive Multi-Pass Programmable Shading

Mark S. Peercy, Marc Olano, John Airey*, P. Jeffrey Ungar
SGI

Abstract

Programmable shading is a common technique for production animation, but interactive programmable shading is not yet widely available. We support interactive programmable shading on virtually any 3D graphics hardware using a scene graph library on top of OpenGL. We treat the OpenGL architecture as a general SIMD computer, and translate the high-level shading description into OpenGL rendering passes. While our system uses OpenGL, the techniques described are applicable to any retained mode interface with appropriate extension mechanisms and hardware API with provisions for recirculating data through the graphics pipeline.

We present two demonstrations of the method. The first is a constrained shading language that runs on graphics hardware supporting OpenGL 1.2 with a subset of the ARB imaging extensions. We remove the shading language constraints by minimally extending OpenGL. The key extensions are *color range* (supporting extended range and precision data types) and *pixel texture* (using framebuffer values as indices into texture maps). Our second demonstration is a renderer supporting the RenderMan Interface and RenderMan Shading Language on a software implementation of this extended OpenGL. For both languages, our compiler technology can take advantage of extensions and performance characteristics unique to any particular graphics hardware.

CR categories and subject descriptors: I.3.3 [Computer Graphics]: Picture/Image generation; I.3.7 [Image Processing]: Enhancement.

Keywords: Graphics Hardware, Graphics Systems, Illumination, Languages, Rendering, Interactive Rendering, Non-Realistic Rendering, Multi-Pass Rendering, Programmable Shading, Procedural Shading, Texture Synthesis, Texture Mapping, OpenGL.

1 INTRODUCTION

Programmable shading is a means for specifying the appearance of objects in a synthetic scene. Programs in a special purpose language, known as *shaders*, describe light source position and emission characteristics, color and reflective properties of surfaces, or transmittance properties of atmospheric media. Conceptually, these programs are executed for each point on an object as it is being rendered to produce a final color (and perhaps opacity) as seen from a given viewpoint. Shading languages can be quite general, having constructs familiar from general purpose programming languages such as `C`, including loops, conditionals, and functions. The most common is the RenderMan Shading Language [32].

The power of shading languages for describing intricate lighting and shading computations been widely recognized since Cook's seminal shade tree research [7]. Programmable shading has played a fundamental role in digital content creation for motion pictures and television for over a decade. The high level of abstraction in programmable shading enables artists, storytellers, and their technical collaborators to translate their creative visions into images more easily. Shading languages are also used for visualization of scientific data. Special *data shaders* have been developed to support the depiction of volume data [3, 8], and a texture synthesis language has been used for visualizing data fields on surfaces [9]. Image processing scripting languages [22, 31] also share much in common with programmable shading.

Despite its proven usefulness in software rendering, hardware acceleration of programmable shading has remained elusive. Most hardware supports a parametric appearance model, such as Phong lighting evaluated per vertex, with one or more texture maps applied after Gouraud interpolation of the lighting results [29]. The general computational nature of programmable shading, and the unbounded complexity of shaders, has kept it from being supported widely in hardware. This paper describes a methodology to support programmable shading in interactive visual computing by compiling a shader into multiple passes through graphics hardware. We demonstrate its use on current systems with a constrained shading language, and we show how to support general shading languages with only two hardware extensions.

1.1 Related Work

Interactive programmable shading, with dynamically changing shader and scene, was demonstrated on the PixelFlow system [26]. PixelFlow has an array of general purpose processors that can execute arbitrary code at every pixel. Shaders written in a language based on RenderMan's are translated into `C++` programs with embedded machine code directives for the pixel processors. An application accesses shaders through a programmable interface extension to OpenGL. The primary disadvantages of this approach are the additional burden it places on the graphics hardware and driver software. Every system that supports a built-in programmable interface must include powerful enough general computing units to execute the programmable shaders. Limitations to these computing units, such as a fixed local memory, will either limit the shaders that may be run, have a severe impact on performance, or cause the system to revert to multiple passes within the driver. Further, every such system will have a unique shading language compiler as part of the driver software. This is a sophisticated piece of software which greatly increases the complexity of the driver.

Our approach to programmable shading stands in contrast to the programmable hardware method. Its inspiration is a long line of interactive algorithms that follow a general theme: treat the graphics hardware as a collection of primitive operations that can be used

*Now at Intrinsic Graphics

to build up a final solution in multiple passes. Early examples of this model include multi-pass shadows, planar reflections, highlights on top of texture, depth of field, and light maps [2, 10]. There has been a dramatic surge of research in this area over the past few years. Sophisticated appearance computations, which had previously been available only in software renderers, have been mapped to generic graphics hardware. For example, lighting per pixel, general bidirectional reflectance distribution functions, and bump mapping now run in real-time on hardware that supports none of those effects natively [6, 17, 20, 24].

Consumer games like ID Software's Quake 3 make extensive use of multi-pass effects [19]. Quake 3 recognizes that multi-pass provides a flexible method for surface design and takes the important step of providing a scripting mechanism for rendering passes, including control of OpenGL blending mode, alpha test functions, and vertex texture coordinate assignment. In its current form, this scripting language does not provide access to all of the OpenGL state necessary to treat OpenGL as a general SIMD machine.

A team at Stanford has been investigating real-time programmable shading. Their focus is a framework and language that explicitly divides operations into those that are executed at the vertex processing stage in the graphics pipeline and those that are executed at the fragment processing stage [25].

The hardware in all of these cases is being used as a computing machine rather than a special purpose accelerator. Indeed, graphics hardware has been used to accelerate techniques such as back-projection for tomographic reconstruction [5] and radiosity approximations [21]. It is now recognized that some new hardware features, such as multi-texture [24, 29], pixel texture [17], and color matrix [23], are particularly valuable for supporting these advanced computations interactively.

1.2 Our Contribution

In this paper, we embrace and extend previous multi-pass techniques. We treat the OpenGL architecture as a SIMD computer. OpenGL acts as an assembly language for shader execution. The challenge, then, is to convert a shader into an efficient set of OpenGL rendering passes on a given system. We introduce a compiler between the application and the graphics library that can target shaders to different hardware implementations.

This philosophy of placing the shading compiler above the graphics API is at the core of our work, and has a number of advantages. We believe the number of languages for interactive programmable shading will grow and evolve over the next several years, responding to the unique performance and feature demands of different application areas. Likewise, hardware will increase in performance and many new features will be introduced. Our methodology allows the languages, compiler, and hardware to evolve independently because they are cleanly decoupled.

This paper has three main contributions. First, we formalize the idea of using OpenGL as an assembly language into which programmable shaders are translated, and we show how to apply dynamic tree-rewriting compiler technology to optimize the mapping between shading languages and OpenGL (Section 2). Second, we demonstrate the immediate application of this approach by introducing a constrained shading language that runs interactively on most current hardware systems (Section 3). Third, we describe the color range and pixel texture OpenGL extensions that are necessary and sufficient to accelerate fully general shading languages (Section 4). As a demonstration of the viability of this solution, we present a complete RenderMan renderer including full support of the RenderMan Shading Language running on a software implementation of this extended OpenGL. We close the paper with a discussion (Section 5) and conclusion (Section 6).

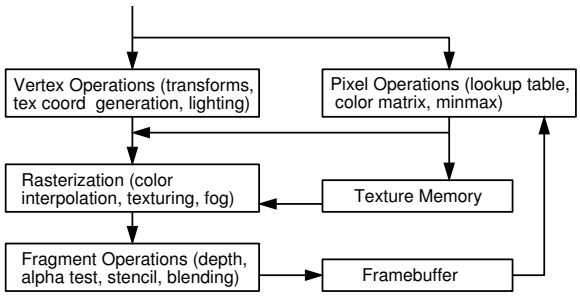


Figure 1: A simplified block diagram of the OpenGL architecture. Geometric data passes through the vertex operations, rasterization, and fragment operations to the framebuffer. Pixel data (either from the host or the framebuffer) passes through the pixel operations and on to either texture memory or through the fragment pipeline to the framebuffer.

2 THE SHADING FRAMEWORK

There is great diversity in modern 3D graphics hardware. Each graphics system includes unique features and performance characteristics. Countering this diversity, all modern graphics hardware also supports the basic features of the OpenGL API standard.

While it is possible to add shading extensions to graphics hardware, OpenGL is powerful enough to support shading with no extensions at all. Building programmable shading on top of standard OpenGL decouples the hardware and drivers from the language, and enables shading on every existing and future OpenGL-based graphics system.

A compiler turns shading computations into multiple passes through the OpenGL rendering pipeline (Figure 1). This compiler can produce a general set of rendering passes, or it can use knowledge of the target hardware to pick an optimized set of passes.

2.1 OpenGL as an Assembly Language

One key observation allows shaders to be translated into multi-pass OpenGL: a single rendering pass is also a general SIMD instruction — the same operations are performed simultaneously for all pixels in an object. At the simplest level, the framebuffer is an accumulator, texture or pixel buffers serve as per-pixel memory storage, blending provides basic arithmetic operations, lookup tables support function evaluation, the alpha test provides a variety of conditionals, and the stencil buffer allows pixel-level conditional execution. A shader computation is broken into pieces, each of which can be evaluated by an OpenGL rendering pass. In this way, we build up a final result for all pixels in an object (Figure 2). There are typically several ways to map shading operations into OpenGL. We have implemented the following:

Data Types: Data with the same value for every pixel in an object are called *uniform*, while data with values that may vary from pixel to pixel are called *varying*. Uniform data types are handled outside the graphics pipeline. The framebuffer retains intermediate varying results. Its four channels may hold one quadruple (such as a homogeneous point), one triple (such as a vector, normal, point, or color) and one scalar, or four independent scalars. We have made no attempt to handle varying data types with more than four channels. The framebuffer channels (and hence independent scalars or

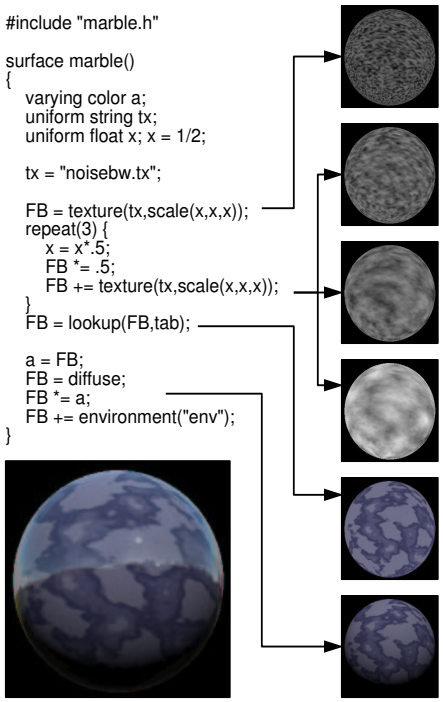


Figure 2: SIMD Computation of a Shader. Some of the different passes for the shader written in ISL listed on the left are shown as thumbnails down the right column. The result of the complete shader is shown on the lower left.

the components of triples and quadruples) can be updated selectively on each pass by setting the write-mask with `glColorMask`.

Variables: Varying global, local, and temporary variables are transferred from the framebuffer to a named texture using `glCopyTexSubImage2D`, which copies a portion of the framebuffer into a portion of a texture. In our system, these textures can be one channel (intensity) or four channels (RGBA), depending on the data type they hold. Variables are used either by drawing a textured copy of the object bounding box or by drawing the object geometry using a projective texture. The relative speed of these two methods will vary from graphics system to graphics system. Intensity textures holding scalar variables are expanded into all four channels during rasterization and can therefore be restored into any framebuffer channel.

Arithmetic Operations: Most arithmetic operations are performed with framebuffer blending. They have two operands: the framebuffer contents and an incoming fragment. The incoming fragment may be produced either by drawing geometry (object color, a texture, a stored variable, etc.) or by copying pixels from the framebuffer and through the pixel operations with `glCopyPixels`. Data can be permuted (*swizzled*) from one framebuffer channel to another or linearly combined more generally using the color matrix during a copy. The framebuffer blending mode, set by `glBlendEquation`, `glBlendFunc`, and `glLogicOp`, supports overwriting, addition, subtraction, multiplication, bit-wise logical operations, and alpha blending. Unextended OpenGL does not have a divide blend mode. We handle divide using multiplication by the reciprocal. The reciprocal is computed like other mathematical functions (see below). More complicated binary operations are reduced to a combination of these primitive operations. For example, a dot product of two vectors is a component-wise multiplication followed by a pixel copy with a color matrix that sums the resulting three components together.

Mathematical and Shader Functions: Mathematical functions with a single scalar operand (e.g. sin or reciprocal) use color or texture lookup tables during a framebuffer-to-framebuffer pixel copy. Functions with more than one operand (e.g. atan2) or a single vector operand (e.g. normalize or color space conversion) are broken down into simpler monadic functions and arithmetic operations, each of which can be supported in a pass through the OpenGL pipeline. Some shader functions, such as texturing and diffuse or specular lighting, have direct correspondents in OpenGL. Often, complex mathematical and shader functions are simply translated to a series of simpler shading language functions.

Flow Control: Stenciling, set by `glStencilFunc` and `glStencilOp`, limits the effect of all operations to only a subset of the pixels, with other pixels retaining their original framebuffer values. We use one bit of the stencil to identify pixels in the object, and additional stencil bits to identify subsets of those pixels that pass varying conditionals (*if-then-else* constructs and loops). One stencil bit is devoted to each level of nesting. Loops with uniform control and conditionals with uniform relations do not need a stencil bit to control their influence because they affect all pixels.

A two step process is used to set the stencil bit for a varying conditional. First, the relation is computed with normal arithmetic operations, such that the result ends up in the alpha channel of the framebuffer. The value is zero where the condition is true and one where it is false. Next, a pixel copy is performed with the alpha $> .5$ test enabled (set by `glAlphaFunc`). Only fragments that pass the alpha test are passed on to the stenciling stage of the OpenGL pipeline. A stencil bit is set for all of these fragments. The stencil remains unchanged for fragments that failed the alpha test. In some cases, the first operation in the body of the conditional can occur in the same pass that sets the stencil.

The passes corresponding to the different blocks of shader code at different nesting levels affect only those pixels that have the proper stencil mask. Because we are executing a SIMD computation, it is necessary to evaluate both branches of *if-then-else* constructs whose relation varies across an object. The stencil compare for the *else* clause simply uses the complement of the stencil bit for the *then* clause. Similarly, it is necessary to repeat a loop with a varying termination condition until all pixels within the object exit the loop. This requires a test that examines all of the pixels within the object. We use the *minmax* function from the ARB imaging extension as we copy the alpha channel to determine if any alpha values are non-zero (signifying they still pass the looping condition). If so, the loop continues.

2.2 OpenGL Encapsulation

We encapsulate OpenGL instructions in three kinds of rendering passes: *GeomPasses*, *CopyPasses*, and *CopyTexPasses*. GeomPasses draw geometry to use vertex, rasterization, and fragment operations. CopyPasses copy a subregion of the framebuffer (via `glCopyPixels`) back into the same place in the framebuffer to use pixel, rasterization, and fragment operations. A stencil allows the CopyPass to avoid operating on pixels outside the object. CopyTexPasses copy a subregion of the framebuffer into a texture object (via `glCopyTexSubImage2D`) and also utilize pixel operations. There are two subtypes of GeomPass. The first draws the object geometry, including normal vectors and texture coordinates. The second draws a screen-aligned bounding rectangle that covers the object using stenciling to limit the operations to pixels on the object. Each pass maintains the relevant OpenGL state for its path

through the pipeline. State changes on drawing are minimized by only setting the state in each pass that is not default and immediately restoring that state after the pass.

2.3 Compiling to OpenGL

The key to supporting interactive programmable shading is a compiler that translates the shading language into OpenGL assembly. This is a CISC-like compiler problem because OpenGL passes are complex instructions. The problem is somewhat simplified due to constraints in the language and in OpenGL as an instruction set. For example, we do not have to worry about instruction scheduling since there is no overlap between rendering passes.

Our compiler implementation is guided by a desire to retarget the compiler to easily take advantage of unique features and performance and to pick the best set of passes for each target architecture. We also want to be able to support multiple shading languages and adapt as languages evolve. To help meet these goals, we built our compiler using an in-house tool inspired by the iburg code generation tool [11], though we use it for all phases of compilation. This tool finds the least-cost covering of a tree representation of the shader based on a text file of patterns.

A simple example can show how the tree-matching tool operates and how it allows us to take advantage of extensions to OpenGL. Part of a shader might be matched by a pair of texture lookups, each with a cost of one, or by a single multi-texture lookup, also with a cost of one. In this case, multi-texture is cheaper because it has a total cost of one instead of two. Using similar matching rules and semantic actions, the compiler can make use of fragment lighting, light texture, noise generation, divide or conditional blends, or any other OpenGL extension [16, 27].

The entire shader is matched at once, giving the set of matching rules that cover the shader with the least total cost. For example, the computations surrounding the above pair of texture lookups expand the set of possible matching rules. Given operation A, texture lookup B, texture lookup C, and operation D, it may be possible to do all of the operations in four separate passes (A,B,C,D), to do the surrounding operations separately while combining the texture lookups into one multi-texture pass for a total cost of three (A,BC,D), or to combine one computation with each texture lookup for a cost of two (AB,CD). By considering the entire shader we can choose the set of matching rules with the least overall cost.

When we use the tool for final OpenGL pass generation, we currently use the number of passes as the cost for each matching rule. For performance optimization, the costs should correspond to predicted rendering speed, so the cost for a GeomPass would be different from the cost for a CopyPass or a CopyTexPass.

The pattern matching happens in two phases, *labeling* and *reducing*. Labeling is done bottom-up through the abstract syntax tree, using dynamic programming to find the least-cost set of pattern match rules. Reducing is done top-down, with one semantic action run before the node's children are reduced and one after. The iburg-like label/reduce tool proved useful for more than just final pass selection. We use it for shader syntax checking, constant folding, and even memory allocation (although most of the memory allocation algorithm is in the code associated with a small number of rules). The ease of changing costs and creating new matching rules allows us to achieve our goal of flexible retargeting of the compiler for different hardware and shading languages.

2.4 Scene Graph Support

Since objects may be rendered multiple times, it is necessary to retain geometry data and to deliver it repeatedly to the graphics hardware. In addition, shaders need to be associated with objects to describe their appearances, and the shaders and objects need to be translated into OpenGL passes to render an image. Our framework supports these operations in a scene graph used by an application through the addition of new scene graph containers and new traversals.

In our implementation, we have extended the Cosmo3D scene graph library [30]. Cosmo3D uses a familiar hierarchical scene graph. Internal nodes describe coordinate transformations, while the leaves are *Shape* nodes, each of which contains a list of *Geometry* and an *Appearance*. Traversals of the scene graph are known as *actions*. A *DrawAction*, for example, is applied to the scene graph to render the objects into a window.

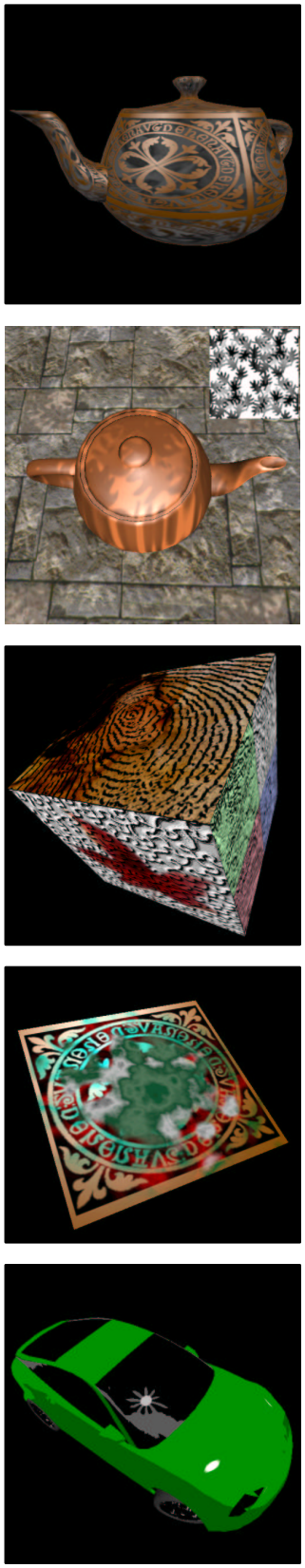
We have implemented a new appearance class that contains shaders. When included in a shape node, this appearance completely describes how to shade the geometry in the shape. The shaders may include a list of active light shaders, a displacement shader, a surface shader, and an atmosphere shader. In addition, we have implemented a new traversal, known as a *ShadeAction*. A ShadeAction converts a scene graph containing shapes with the new appearance into another Cosmo3D scene graph describing the multiple passes for all of the objects in the original scene graph. (The transformation of scene graphs is a powerful, general technique that has been proposed to address a variety of problems [1].) The key element of the ShadeAction is the shading language compiler that converts the shaders into multiple passes. A ShadeAction may treat multiple objects that share the same shader as a single, combined object to minimize overhead. A DrawAction applied to this second scene graph renders the final image.

The scene graph passes information to the compiler including the matrix to transform from the object's coordinate system into camera space and the screen space footprint for the geometry. The footprint is computed during the ShadeAction by projecting a 3D bounding box of the geometry into screen space and computing an axis-aligned 2D bounding box of the eight projected points. Only pixels within the 2D bounding box are copied on a CopyPass or drawn on the quad-GeomPass to minimize unnecessary data movement when shading each object.

We provide support for debugging at the single-step, pass-by-pass level through special hooks inserted into the DrawAction. Each pass is held in an extended Cosmo3D *Group* node, which invokes the debugging hook functions when drawn. Each pass is also tagged with the line of source code that generated it, so everything from shader source-level debugging to pass-by-pass image dumps is possible. Hooks at the per-pass level also let us monitor or estimate performance. At the coarsest level, we can find the number of passes executed, but we can also examine each pass to record details like pixels written or time to draw.

3 EXAMPLE: INTERACTIVE SL

We have developed a constrained shading language, called ISL (for Interactive Shading Language) [25] and an ISL compiler to demonstrate our method on current hardware. ISL is similar in spirit to the RenderMan Shading Language in that it provides a `C`-like syntax to specify per-pixel shading calculations, and it supports separate light, surface, and atmosphere shaders. Data types include varying colors, and uniform floats, colors, matrices, and strings. Local variables can hold both uniform and varying values. Nestable flow control structures include loops with uniform control, and uniform and varying conditionals. There are built-in functions for diffuse and specular lighting, texture mapping, projective textures, environment mapping, RGBA one-dimensional lookup tables, and per-pixel ma-



```
surface celtic() {
    varying color a;
    FB = diffuse;
    FB *= color(.5,.2,0.,1.);
    a = FB;
    FB = specular(30.);
    FB += a;
    FB *= texture("celtic");
    a = FB;
    FB = 1;
    FB −= texture("celtic");
    FB *= texture("silk");
    FB *= .15;
    FB += a;
}
distantlight leaves(uniform string
                map = "leaves", ...) {
    uniform float tx;
    uniform float ty;
    uniform float tz;
    tx = frame*speedx+phasex;
    ty = frame*speedy+phasey;
    tz = frame*speedz+phasez;
    FB = project(map,
            scale(sx,sx,sx)*
            rotate(0,0,1,rx)*
            translate(ax*sin(tx),0,0)*
            shadermatrix);
    FB *= project(map,
            scale(sy,sy,sy)*...);
}
uniform matrix lt = (0,0,0,0,
     0,0,0,0,1,1,1,0,0,0,0,1);
surface bump(uniform string b="";
      uniform string tx =  "") {
    uniform matrix m;
    FB = texture(b);
    m = objectmatrix;
    m[0][3] = m[1][3] = m[2][3] = 0.;
    m[3][3] = m[3][0] = m[3][1] = 0.;
    m[3][2] = 0.;
    m = lt*m*translate(−1,−1,−1)*
            scale(2,2,2);
    FB = transform(FB,m);
    FB *= texture(tx);
}
#include "threshtab.h"
surface shipRockRot(...) {
    varying color a, b, c;
    FB = texture(rot); FB *= .5;
    FB += .32*(1−cos(.08*frame));
    FB = lookup(FB,mtab); c = FB;
    FB = color(1,1,1,1); FB −= c;
    FB *= texture(t1); a = FB;
    FB = texture(t2);
    FB *= texture(rot);
    FB = diffuse;
    FB *= color(.5,.2,0,1); b = FB;
    FB = specular(30.);
    FB += b; FB *= texture(t2);
    FB *= c; FB += a;
}

#include "swizzle.h"
table greentable = { {0,.2,0,1},
                     {0,.4,0,1) };
surface toon(uniform float do = 1.;
     uniform float edge = .25 ) {
    FB = environment("park.env");
    if (do > .5) {
        FB += edge;
        FB =transform(FB,rgba_rrra);
        FB =lookup(FB,greentable);
        FB += environment("sun");
    }
}
```

Figure 3: ISL Examples. ISL shaders are shown to the right of each image. Ellipses denote where parameters and statements have been omitted. Some tables are in header files.

trix transformations. In addition, ISL supports uniform shader parameters and a set of uniform global variables (shader space, object space, time, and frame count).

We have intentionally constrained ISL in a number of ways. First, we only chose primitive operations and built-in functions that can be executed on any hardware supporting base OpenGL 1.2 plus the color matrix extension. Consequently, many current hardware systems can support ISL. (If the color matrix transformation is eliminated, ISL should run anywhere.) This constraint provides the shader writer with insight into how limited precision of current commercial hardware may affect the shader. Second, the syntax does not allow varying expressions of expressions, which ensures that the compiler does not need to create any temporary storage not already made explicit in the shader. As a result, the writer of a shader knows by inspection the worst-case temporary storage required by the shading code (although the compiler is free to use less storage, if possible). Third, arbitrary texture coordinate computation is not supported. Texture coordinates must come either from the geometry or from the standard OpenGL texture coordinate generation methods and texture matrix.

One consequence of these design constraints is that ISL shading code is largely decoupled from geometry. For example, since shader parameters are uniform there is no need to attach them directly to each surface description in the scene graph. As a result, ISL and the compiler can migrate from application to application and scene graph to scene graph with relative ease.

3.1 Compiler

We perform some simple optimizations in the parser. For instance, we do limited constant compression by evaluating at parse time all expressions that are declared uniform. When parameters or the shader code change, we must reparse the shader. In our current system, we do this every time we perform a ShadeAction. A more sophisticated compiler, such as the one implemented for the RenderMan Shading Language (Section 4) performs these optimizations outside the parser.

We expand the parse trees for all of the shaders in an appearance (light shaders, surface shader, and atmosphere shader) into a single tree. This tree is then labeled and reduced using the tree matching compiler tool described in Section 2.3. The costs fed into the labeler instruct the compiler to minimize the total number of passes, regardless of the relative performance of the different kinds of passes.

The compiler recognizes and optimizes subexpressions such as a texture, diffuse, or specular lighting multiplied by a constant. The compiler also recognizes when a local variable is assigned a value that can be executed in a single pass. Rather than executing the pass, storing the result, and retrieving it when referenced, the compiler simply replaces the local variable usage with the single pass that describes it.

3.2 Demonstration

We have implemented a simple viewer on top of the extended scene graph to demonstrate ISL running interactively. The viewer supports mouse interaction for rotation and translation. Users can also modify shaders interactively in two ways. They can edit shader text files, and their changes are picked up immediately in the viewer. Additionally, they can modify parameters by dragging sliders, rotating thumb-wheels, or entering text in a control panel. The viewer creates the control panel on the fly for any selected shader. Changes to the parameters are seen immediately in the window. Examples of the viewer running ISL are given in Figures 2 and 3.

4 EXAMPLE: RENDERMAN SL

RenderMan is a rendering and scene description interface standard developed in the late 1980s [14, 28, 32]. The RenderMan standard includes procedural and bytestream scene description interfaces. It also defines the RenderMan Shading Language, which is the *de facto* standard for programmable shading capability and represents a well-defined goal for anyone attempting to accelerate programmable shading.

The RenderMan Shading Language is extremely general, with control structures common to many programming languages, rich data types, and an extensive set of built-in operators and geometric, mathematical, lighting, and communication functions. The language originally was designed with hardware acceleration in mind, so complicated or user-defined data types that would make acceleration more difficult are not included. It is a large but straightforward task to translate the RenderMan Shading Language into multi-pass OpenGL, assuming the following two extensions:

Extended Range and Precision Data Types: Even the simplest RenderMan shaders have intermediate computations that require data values to extend beyond the range [0-1], to which OpenGL fragment color values are clamped. In addition, they need higher precision than is found in current commercial hardware. With the *color range* extension, color data can have an implementation-specific range to which it is clamped during rasterization and framebuffer operations (including color interpolation, texture mapping, and blending). The framebuffer holds colors of the new type, and the conversion to a displayable value happens only upon video scan-out. We have used the color range extension with an IEEE single precision floating point data type or a subset thereof to support the RenderMan Shading Language.

Pixel Texture: RenderMan allows texture coordinates to be computed procedurally. In this case, texture coordinates cannot be expected to change linearly across a geometric primitive, as required in unextended OpenGL. This general two-dimensional indirection mechanism can be supported with the OpenGL pixel texture extension [17, 18, 27]. This extension allows the (possibly floating point) contents of the framebuffer to be used as texture indices when pixels are copied from the framebuffer. The red, green, blue, and alpha channels are used as texture coordinates s, t, r, and q, respectively. We use pixel texture not only to index two dimensional textures but also to index extremely wide one-dimensional textures. These wide textures are used as lookup tables for mathematical functions such as sin, reciprocal, and sqrt. These can be simple piecewise linear approximations, starting points for Newton iteration, components used to construct the more complex mathematical functions, or even direct one-to-one mappings for a reduced floating point format.

4.1 Scene Graph Support

The RenderMan Shading Language demands greater support from the scene graph library than ISL because geometry and shaders are more tightly coupled. *Varying parameters* can be supplied as four values that correspond to the corners of a surface patch, and the parameter over the surface is obtained through bilinear interpolation. Alternatively, one parameter value may be supplied per control point for a bicubic patch mesh or a NURBS patch, and the parameter is interpolated using the same basis functions that define the surface. We associate a (possibly empty) list of named parameters with each surface to hold any parameters provided when the surface is defined. When the surface geometry is tessellated to form *GeoSets* (triangle strip sets and fan sets, etc.), its parameters are transferred to the GeoSets so that they may be referenced

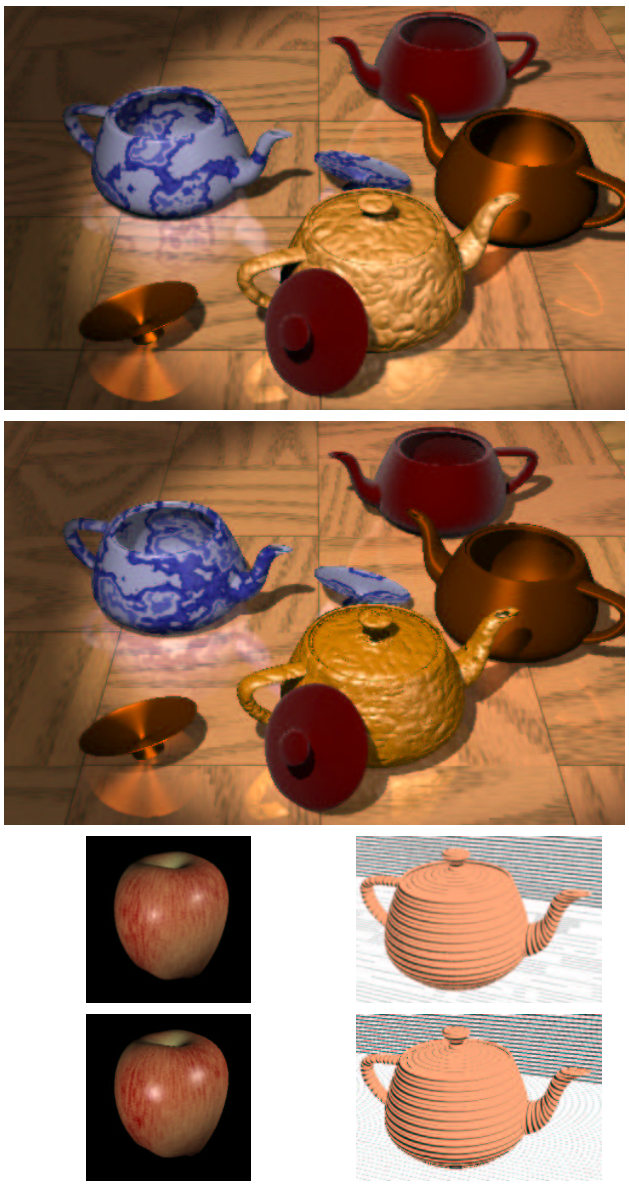


Figure 4: RenderMan SL Examples. The top and bottom images of each pair were rendered with PhotoRealistic RenderMan from Pixar and our multi-pass OpenGL renderer, respectively. No shaders use image maps, except for the reflection and depth shadow maps generated on the fly. The wood floor, blue marble, red apple, and wood block print textures all are generated procedurally. The velvet and brushed metal shaders use sophisticated `illuminance` blocks for their reflective properties. The specular highlight differences are due to Pixar's proprietary specular function; we use the definition from the RenderMan specification. The blue marble, wood floor, and apple do not match because of differences in the `noise` function. Other discrepancies typically are due to limited precision lookup tables used to help evaluate mathematical functions. (Credit: LGParquetPlank by Larry Gritz, SHWvelvet and SHWbrushedmetal by Stephen Westin, DPBlueMarble by Darwin Peachey, eroded from the RenderMan companion, JMredapple by Jonathan Merritt, and woodblockprint by Scott Johnston. Courtesy of the RenderMan Repository `http://www.renderman.org`.)

and drawn as vertex colors by the passes produced by the compiler. Similarly, a shader may require derivatives of surface properties, such as the partial derivatives of the position (*dP/du* and *dP/dv*) either as global variables or through a differential function such as `calculatenormal`. A shader may also use derivatives of user-supplied parameters. The compiler can request from the scene graph any of these quantities evaluated over a surface at the same points used in its tessellation. As with any other parameter, they are computed on the host and stored in the vertex colors for the surface. Where possible, lazy evaluation ensures that the user does not pay in time or space for this support unless requested.

4.2 Compiler

Our RenderMan compiler is based on multiple phases of the tree-matching tool described in Section 2.3. The phases include:

Parsing: convert source into an internal tree representation.
Phase0: detect errors
Phase1: perform context-sensitive typing (e.g. noise, texture)
Phase2: detect and compress uniform expressions
Phase3: compute "difference trees" for Derivatives
Phase4: determine variable usage and live range information
Phase5: identify possible OpenGL instruction optimizations
Phase6: allocate memory for variables
Phase7: generate optimized, machine specific OpenGL

The mapping of RenderMan to OpenGL follows the methodology described in Section 2.1. Texturing and some lighting carry over directly; most math functions are implemented with lookup tables; coordinate transformations are implemented with the color matrix; loops with varying termination condition are supported with minmax; and many built-in functions (including illuminance, solar, and illuminate) are rewritten in terms of simpler operations. Features whose mapping to OpenGL is more sophisticated include:

Noise: The RenderMan SL provides band-limited `noise` primitives that include 1D, 2D, 3D, and 4D operands and single or multiple component output. We use floating point arithmetic and texture tables to support all of these functions.

Derivatives: The RenderMan SL provides access to surface-derivative information through functions that include `Du`, `Dv`, `Deriv`, `area`, and `calculatenormal`. We dedicate a compiler phase to fully implement these functions using a technique similar that described by Larry Gritz [12].

A number of optimizations are supported by the compiler. Uniform expressions are identified and computed once for all pixels. If texture coordinates are linear functions of *s* and *t* or vertex coordinates, they are recognized as a single pass with some combination of texture coordinate generation and texture matrix. Texture memory utilization is minimized by allocating storage based on single-static assignment and live-range analysis [4].

4.3 Demonstration

We have implemented a RenderMan renderer, complete with shading language, bytestream, and procedural interfaces on a software implementation of OpenGL including color range and pixel texture. We experimented with subsets of IEEE single precision floating point. An interesting example was a 16 bit floating point format with a sign bit, 10 bits of mantissa and 5 bits of exponent. This format was sufficient for most shaders, but fell short when computing derivatives and related difference-oriented functions such as `calculatenormal`. Our software implementation supported other OpenGL extensions (cube environment mapping, fragment lighting, light texture, and shadow), but they are not strictly necessary as they can all be computed using existing features.

ISL Image	celtic	leaves	bump	rot	toon
MPix Filled	2.8	4.3	1.2	2.2	1.9
Frames/Second	6.8	7.3	9.6	12.5	4.6
RSL Image	teapots	apple	print		
MPix Filled	500	280	144		

Table 1: Performance for 512x512 images on Silicon Graphics Octane/MXI

The RenderMan bytestream interface was implemented on top of the RenderMan procedural interface. When data is passed to the procedural interface, it is incorporated into a scene graph. Higher order geometric primitives not native to Cosmo3D, such as trimmed quadrics and NURBS patches are accommodated by extending the scene graph library with parametric surface types, which are tessellated just before drawing. At the WorldEnd procedural call, this scene graph is rendered using a ShadeAction that invokes the RenderMan shading language compiler followed by a DrawAction.

To establish that the implementation was correct, over 2000 shading language tests, including point-feature tests, publicly available shaders, and more sophisticated shaders were written or obtained. The results of our renderer were compared to Pixar's commercially available PhotoRealistic RenderMan renderer. While never bit-for-bit accurate, the shading is typically comparable to the eye (with expected differences due, for instance, to the `noise` function). A collection of examples is given in Figure 4. We focused primarily on the challenge of mapping the entire language to OpenGL, so there is considerable room for further optimization.

There are a few notable limitations in our implementation. Displacement shaders are implemented, but treated as bump mapping shaders; surface positions are altered only for the calculation of normals, not for rasterization. True displacement would have to happen during object tessellation and would have performance similar to displacement mapping in traditional software implementations. Transparency is not implemented. It is possible, but requires the scene graph to depth-sort potentially transparent surfaces. Pixel texture, as it is implemented, does not support texture filtering, which can lead to aliasing. Our renderer also does not currently support high quality pixel antialiasing, motion blur, and depth of field. One could implement all of these through the accumulation buffer as has been demonstrated elsewhere [13].

5 DISCUSSION

We measured the performance of several of our ISL and RenderMan shaders (Table 1). The performance numbers for millions of pixels filled are conservative estimates since we counted all pixels in the object's 2D bounding box even when drawing object geometry that touched fewer pixels.

5.1 Drawbacks

Our current system has a number of inefficiencies that impact our performance. First, since we do not use deferred shading, we may spend several passes rendering an object that is hidden in the final image. There are a variety of algorithms that would help (for example, visibility culling at the scene graph level), but we have not implemented any of them.

Second, the bounding box of objects in screen space is used to define the active pixels for many passes. Consequently pixels within the bounding box but not within the object are moved unnecessarily. This taxes one of the most important resources in hardware: bandwidth to and from memory.

Third, we have only included a minimal set of optimization rules in our compiler. Many current hardware systems share framebuffer and texture memory bandwidth. On these systems, storage and retrieval of intermediate results bears a particularly high price. This is a primary motivation for doing as many operations per pass as possible. Our iburg-like rule matching works well for the pipeline of simple units found in standard OpenGL, but more complex units (as found in some new multitexture extensions, for example) require more powerful compiler technology. Two possibilities are surveyed by Harris [15].

5.2 Advantages

Our methodology allows research and development to proceed in parallel as shading languages, compilers, and hardware independently evolve. We can take advantage of the unique feature and performance needs of different application areas through specialized shading languages.

The application does not have to handle the complexities of multipass shading since the application interface is a scene graph. This model is a natural extension of most interactive applications, which already have a retained mode interface of some sort to enable users to manipulate their data. Applications still retain the other advantages of having a scene graph, like occlusion culling and level of detail management.

As mentioned, we have only implemented a few of the many possible compiler optimizations. As the compiler improves, our performance will improve, independent of language or hardware.

Finally, the rapid pace of graphics hardware development has resulted in systems with a diverse set of features and relative feature performance. Our design allows an application to use a shading language on all of the systems, and still take advantage of many of their unique characteristics. Hardware vendors do not need to create the shading compiler and retained data structures since they operate above the level of the drivers. Further, since complex effects can be supported on unextended hardware, designers are free to create fast, simple hardware without compromising on capabilities.

6 CONCLUSION

We have created a software layer between the application and the hardware abstraction layer to translate high-level shading descriptions into multi-pass OpenGL. We have demonstrated this approach with two examples, a constrained shading language that runs interactively on current hardware, and a fully general shading language. We have also shown that general shading languages, like the RenderMan Shading Language, can be implemented with only two additional OpenGL extensions.

There is a continuum of possible languages between ISL and the RenderMan Shading Language with different levels of functionality. We have applied our method to two different shading languages in part to demonstrate its generality.

There are many avenues of future research. New compiler technology can be developed or adapted for programmable shading. There are significant optimizations that we are investigating in our compilers. Research is also needed to understand what hardware features are best for supporting interactive programmable shading. Finally, given examples like the scientific visualization constructs described by Crawfis that are not found in the RenderMan shading language [9], we believe the wide availability of interactive programmable shading will spur exciting developments in new shading languages and new applications for them.

References

[1] BIRCH, P., BLYTHE, D., GRANTHAM, B., JONES, M., SCHAFER, M., SEGAL, M., AND TANNER, C. *An OpenGL++ Specification*. SGI, March 1997.

[2] BLYTHE, D., GRANTHAM, B., KILGARD, M. J., MCREYNOLDS, T., NELSON, S. R., FOWLER, C., HUI, S., AND WOMACK, P. Advanced graphics programming techniques using OpenGL: Course notes. In *Proceedings of SIGGRAPH '99* (July 1999).

[3] BOCK, D. Tech watch: Volume rendering. *Computer Graphics World 22*, 5 (May 1999).

[4] BRIGGS, P. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[5] CABRAL, B., CAM, N., AND FORAN, J. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *1994 Symposium on Volume Visualization* (October 1994), 91–98. ISBN 0-89791-741-3.

[6] CABRAL, B., OLANO, M., AND NEMEC, P. Reflection space image based rendering. *Proceedings of SIGGRAPH 99* (August 1999), 165–170.

[7] COOK, R. L. Shade trees. *Computer Graphics (Proceedings of SIGGRAPH 84) 18*, 3 (July 1984), 223–231. Held in Minneapolis, Minnesota.

[8] CORRIE, B., AND MACKERRAS, P. Data shaders. *Visualization '93 1993* (1993).

[9] CRAWFIS, R. A., AND ALLISON, M. J. A scientific visualization synthesizer. *Visualization '91* (1991), 262–267.

[10] DIEFENBACH, P. J., AND BADLER, N. I. Multi-pass pipeline rendering: Realism for dynamic environments. *1997 Symposium on Interactive 3D Graphics* (April 1997), 59–70.

[11] FRASER, C. W., HANSON, D. R., AND PROEBSTING, T. A. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems 1*, 3 (September 1992), 213–226.

[12] GRITZ, L., AND HAHN, J. K. BMRT: A global illumination implementation of the RenderMan standard. *Journal of Graphics Tools 1*, 3 (1996), 29–47.

[13] HAEBERLI, P. E., AND AKELEY, K. The accumulation buffer: Hardware support for high-quality rendering. *Computer Graphics (Proceedings of SIGGRAPH 90) 24*, 4 (August 1990), 309–318.

[14] HANRAHAN, P., AND LAWSON, J. A language for shading and lighting calculations. *Computer Graphics (Proceedings of SIGGRAPH 90) 24*, 4 (August 1990), 289–298.

[15] HARRIS, M. Extending microcode compaction for real architectures. In *Proceedings of the 20th annual workshop on Microprogramming* (1987), pp. 40–53.

[16] HART, J. C., CARR, N., KAMEYA, M., TIBBITTS, S. A., AND COLEMAN, T. J. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (August 1999), 45–53.

[17] HEIDRICH, W., AND SEIDEL, H.-P. Realistic, hardware-accelerated shading and lighting. *Proceedings of SIGGRAPH 99* (August 1999), 171–178.

[18] HEIDRICH, W., WESTERMANN, R., SEIDEL, H.-P., AND ERTL, T. Applications of pixel textures in visualization and realistic image synthesis. *1999 ACM Symposium on Interactive 3D Graphics* (April 1999), 127–134. ISBN 1-58113-082-1.

[19] JAQUAYS, P., AND HOOK, B. Quake 3: Arena shader manual, revision 10. In *Game Developer's Conference Hardcore Technical Seminar Notes* (December 1999), C. Hecker and J. Lander, Eds., Miller Freeman Game Group.

[20] KAUTZ, J., AND MCCOOL, M. D. Interactive rendering with arbitrary brdfs using separable approximations. *Eurographics Rendering Workshop 1999* (June 1999). Held in Granada, Spain.

[21] KELLER, A. Instant radiosity. *Proceedings of SIGGRAPH 97* (August 1997), 49–56.

[22] KYLANDER, K., AND KYLANDER, O. S. *Gimp: The Official Handbook*. The Coriolis Group, 1999.

[23] MAX, N., DEUSSEN, O., AND KEATING, B. Hierarchical image-based rendering using texture mapping hardware. *Rendering Techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)* (June 1999), 57–62.

[24] MCCOOL, M. D., AND HEIDRICH, W. Texture shaders. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (August 1999), 117–126.

[25] OLANO, M., HART, J. C., HEIDRICH, W., MCCOOL, M., MARK, B., AND PROUDFOOT, K. Approaches for procedural shading on graphics hardware: Course notes. In *Proceedings of SIGGRAPH 2000* (July 2000).

[26] OLANO, M., AND LASTRA, A. A shading language on graphics hardware: The PixelFlow shading system. *Proceedings of SIGGRAPH 98* (July 1998), 159–168.

[27] OPENGL ARB. Extension specification documents. http://www.opengl.org-/Documentation/Extensions.html, March 1999.

[28] PIXAR. *The RenderMan Interface Specification: Version 3.1*. Pixar Animation Studios, September 1999.

[29] SEGAL, M., AKELEY, K., FRAZIER, C., AND LEECH, J. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Silicon Graphics, Inc., 1999.

[30] SGI TECHNICAL PUBLICATIONS. *Cosmo 3D Programmer's Guide*. SGI Technical Publications, 1998.

[31] SIMS, K. Particle animation and rendering using data parallel computation. *Computer Graphics (Proceedings of SIGGRAPH 90) 24*, 4 (August 1990), 405–413.

[32] UPSTILL, S. *The RenderMan Companion*. Addison-Wesley, 1989.

Level-of-Detail Shaders

Marc Olano, Bob Kuehne *
SGI

Abstract

Current graphics hardware can render objects using simple procedural shaders in real-time. However, detailed, high-quality shaders will continue to stress the resources of hardware for some time to come. Shaders written for film production and software renderers may stretch to thousands of lines. The difficulty of rendering efficiently is compounded when there is not just one, but a scene full of shaded objects, surpassing the capability of any hardware to render. This problem has many similarities to the rendering of large models, a problem that has inspired extensive research in geometric level-of-detail and geometric simplification. We introduce an analogous process for shading, *shader simplification*. Starting from an initial detailed shader, shader simplification produces a 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.

CR categories and subject descriptors: I.3.3 [Computer Graphics]: Picture/Image generation — Display algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism — Color, shading, shadowing and texture.

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 and Perlin [10, 35], and popularized by the RenderMan Shading language [20]. A shader is a simple 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 procedures that compute surface color, attenuation of light through a volume (as with fog), light color and direction, fine changes to the surface position, or transformation of control points or vertices.

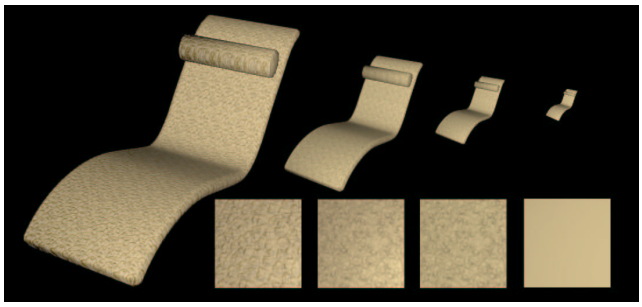


Figure 1: LOD shader upholstering a Le Corbusier chair.

Recent graphics hardware can render simple procedural shaders in real-time [4, 5, 31, 33, 34, 36]. Shaders that exceed the hardware's abilities for rendering of a single object must be rendered using multiple passes through the graphics pipeline. The resulting multi-pass shaders can achieve real-time performance, but many complex shaders in a single scene can easily overwhelm any graphics hardware. Even for shaders that execute in a single rendering pass, the number of textures or combiner stages used can affect overall performance [31].

Consider a realistic shader for a leather chair. Features of this shader may include an overall leather texture or bump map, a couple of measured BRDFs (bidirectional reflectance distribution functions) for worn and unworn areas on the seat, bumps for the stitching, with dust collected in the crevices, scuff marks, changes in color due to variations in the leather, and potentially even more. Such a shader can provide a satisfying interactive rendering of the seat for detailed examination, but is overkill as you move away to see the rest of the room and all the other, buildings, trees and pedestrians using shaders of similar complexity. Figure 1 does not have all the features described, but with a bump map and measured leather BRDF it still exceeds current single pass rendering capabilities.

In this paper, we introduce level-of-detail shaders (LOD shaders) to solve the problem of providing both interactive performance and convincing detailed shading of many objects in a scene. A level-of-detail shader automatically adjusts the shading complexity based on one or more input pa-

*email:{olano,rpk}@sgi.com

rameters, providing only the detail appropriate for the current viewing conditions and resource limits. We present a general framework for creating a level-of-detail shader from a detailed source shader which could be used for automatic LOD shader generation. Finally, we provide details and results from our building-block based level-of-detail shader tools, where the general framework for shader simplification has been manually applied to building-block functions used for writing complex shaders.

1.1 Background

This work is directly inspired by the body of research on geometric simplification. Specifically, many of our shader simplification operations are 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, 41]. 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 reconstructed to any level at run-time [24]. These ideas have had a large influence on more recent polygonal simplification work ([16, 22, 25] and many others).

Many shader simplifications involve generating textures to stand in for one or more other shading operations. Guenter, Knoblock and Ruf replaced static sequences of shading operations with pre-generated textures [19]. Heidrich has analyzed texture sizes and sampling rates necessary for accurate evaluation of shaders into texture [32]. 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. [40].

We also draw on the body of BRDF approximation methods. 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 [13]. Kautz and McCool presented a similar method for real-time BRDF rendering, computing functions of view, light, or other bases as textures using either SVD or a simpler normalized integration method [27]. McCool, Ang and Ahmad's homomorphic factorization uses only products of 2D texture lookups, fit using least-squares [29]. In a related area, Ramamoorthi and Hanrahan used a common set of spherical harmonic basis textures for reconstructing irradiance environment maps [37].

This work is also directly derived from efforts to antialias shaders. The primary form of antialiasing provided in the RenderMan shading language is a 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) [11]. Perlin describes automatic use of blending where `if` is used in the shading code [11]. Heidrich and his collaborators also did automatic antialiasing, using affine arithmetic to compute the shading results and estimate the frequency and error in the results [23].

Finally, there have been several researchers who have done more ambitious shader transformations. Goldman described multiple versions of a fur shader used in several movies, though switches between *realfur* and *fakefur* were only done between shots [18]. Kajiya was the first to pose the problem of converting large-scale surface characteristics to a bump map or BRDF representation [26]. Along this line, Fournier used nonlinear optimization to fit a bump map to a sum of several standard Phong *peaks* [12]. Cabral, Max and Springmeyer 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]. More recently, Apodaca and Gritz manually created a hierarchy of filtered level-of-detail textures [3], while Kautz approached the problem in reverse, creating bump maps to statistically match a chosen fractal micro-facet BRDF [28].

This work is set within the context of recent advances in interactive shading languages, motivating the need for shaders that can transition smoothly from high quality to fast rendering. The first such system by Rhoades et al. was a relatively low-level language for the Pixel-Planes 5 machine at UNC [38]. This was followed by Olano and collaborators with 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 work presented here uses their OpenGL Shader ISL language as the format for both input shaders and LOD shader results.

There are many emerging options for assembler-level interfaces to hardware accelerated shading, including offerings by NVIDIA and ATI as well as a shading interface within DirectX [4, 5, 30, 31]. The shading group at Stanford, led by Kekoa Proudfoot and Bill Mark, created another high-level real-time shading language that can be compiled into either multiple rendering passes or a single pass using NVIDIA or ATI hardware extensions [36]. A group at 3DLabs, led by Randi Rost, is also spearheading an effort to create a high-level shading language for OpenGL version 2.0.

2 USING LOD SHADERS

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. The following directly echos the points from Hoppe's original progressive mesh paper [24].

- *Shader simplification*: The LOD shader can be generated automatically from an initial complex shader using automatic tools (though as in the early days of mesh

simplification, these tools are not yet as automatic as we would like).

- *LOD approximation*: Like a progressive mesh, an LOD shader contains all levels of detail. Thus it can include the shader equivalent of Hoppe's *geomorphs* to smoothly transition from one level to the next.

- *Progressive transmission and compression*: The representation of a shader is much smaller than that of a mesh. Even relatively complex RenderMan shaders are typically only a few thousand lines of code. Shaders for real-time are seldom more complex than several tens of lines of code. Yet a scene with thousands of LOD shaders may still benefit by first storing and sending the simplest levels followed by transmission of the more complex 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. For an LOD shader these aspects are treated independently. Current 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 may also apply certain simplifications and not others based on pressure from hardware resource limits. 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.

Many of these points depend on the storage of an LOD shader. Starting from a complex shader we create a series of simplification operations to produce the most simplified shader, represented as another shader in the source shading language. This combined shader includes all of the levels within a single shading function with additional level control parameters. This provides several practical advantages as the LOD shader is indistinguishable, beyond its additional parameters, from a non-LOD shader. Since OpenGL Shader (and most other shading systems) set shader parameters by name, with default values for unset parameters, LOD shaders are easily interchanged with other shaders. For example, this can allow easy drop in replacement of the covering on a car seat, from a simple stand-in to a non-LOD vinyl shader, an LOD leather shader, or an LOD fabric shader.

The set of level-control parameters are the one aspect that distinguishes the interface to an LOD shader from other shaders. For interchangeable use the parameter set should be agreed upon by both the application and shader simplifier. These parameters are used within the LOD shader to switch

```
FB=diffuse();

FB*=texture("tex");
```

a) basic block

```
FB=diffuse();
if (time<10)
  FB*=texture("tex");
```

b) split blocks

Figure 2: Candidate blocks. a) a single basic block that could be simplified. b) blocks split by a conditional — will not be merged together

and blend between different levels as well as to define the ranges where each level is valid. As with geometric level-of-detail, parameter choices may include distance to the object, approximate screen size of the rendered object, importance of the object, or available time budget. For shading, we may also add budgets for hardware resource limits such as texture memory availability. Many of these parameters could instead be collected into a single aggregate parameter, or controlled through an optimization function as done by Funkhouser and Séquin [15]. All examples in this paper use a single parameter set using a distance metric.

3 SIMPLIFICATION FRAMEWORK

Shader simplification creates an LOD shader from an arbitrary source shader. We describe the simplification process in terms of four stages. First, identify candidate blocks of shader code. Second, produce a set of simplified versions of the candidate blocks. Third, associate level parameters with the simplified blocks, and finally assemble the result into an LOD shader. These stages can be repeated to achieve further simplification, where two or more simplified blocks can be combined into a single larger candidate block for another simplification run.

3.1 Finding Candidate Blocks

The first step toward creating an LOD shader is identifying blocks of shader code that are candidates for simplification. These are like edges for edge-collapse based polygonal simplification. Finding the set of candidate blocks in a shader is slightly more complicated than finding the set of edges in a model, but can be done with a static analysis of the original shader code.

A static analysis is one done before actual execution; it only has access to what can be inferred from the source code itself. In particular, results for conditionals and loops involving compile-time constants are known (*uniform* in ISL parlance), but not ones that might change at run-time (*parameter* in ISL). As a result, choosing a static analysis restricts simplification possibilities to what can be done within a basic block, without crossing a run-time loop or conditional (Figure 2).

Each block within the shader has some variables that are input to the computations within the block and others that are results computed by the block. Expressions within the block form a dependence graph with operations represented as nodes in the graph and variables as edges linking operation to operation. This graph can be partitioned into subgraphs

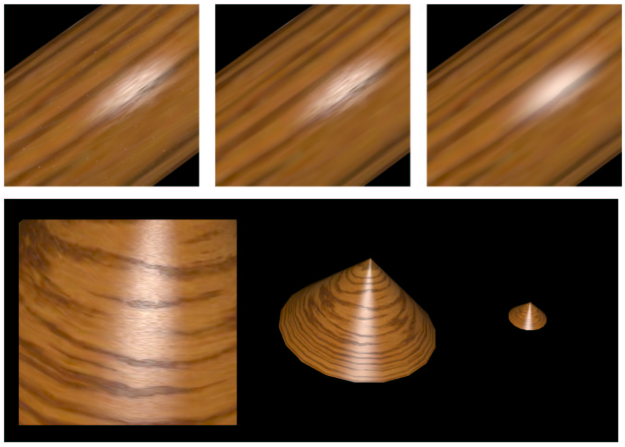


Figure 3: Removal of operations as contributions become imperceptible. Top row, left to right: Close-up of torus mapped with detail dust and scratch textures, with dust and scratches removed, with specular mask removed. Bottom row, left to right: image sequence of the wood applied to a cone with each removal displayed at it's expected switching distance.

where each subgraph computes one block output or intermediate result. These subgraphs are the candidate blocks for simplification. Any basic block can be partitioned in many ways, and the choice of block partitioning is somewhat analogous to choosing edges for mesh simplification.

3.2 Simplifications

Each of the candidate blocks described above computes one result based on a set of inputs. The simplification operations on this block perform a local substitution of a simpler form in place of the original, producing equivalent output while keeping the form of the total shader the same. Simplifications that are not lossy are handled by the shading compiler optimization [19, 33, 34, 36].

Simplifications are chosen by matching a set of heuristic rules. While logically separate, the selection of simplification rules and partitioning of the basic block can be done at the same time using a tool like `iburg` [14]. `Iburg` is a compiler tools designed for use in code generation. Given a piece of code represented as an expression tree, it finds the least cost cover by a set of rules through a bottom-up dynamic programming algorithm.

Finding simplification rule costs for use by `iburg` requires analysis of input textures as well as the shader itself, and application of a rule may require generating a new derived texture as part of the LOD shader generation pre-process.

We classify these rule-based substitutions into one of four forms.

Remove: A candidate block that doesn't contribute enough anymore, or that consists of only high-frequency elements above the Nyquist frequency is replaced by a constant. This effectively removes the effect of portions of the shader that are no longer significant (Figures 3,4).

Collapse: A candidate block consisting of several operations may be merged into a single new operation. For example, a coarse texture and a rotated and repeated detail texture can be combined into a single merged texture of a new size (Figure 5).

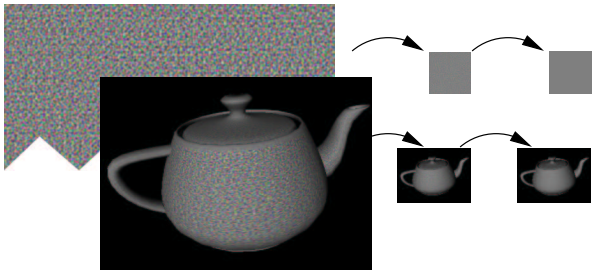


Figure 4: Band-limited Perlin noise texture, noise at a distance, and noise replaced with average value

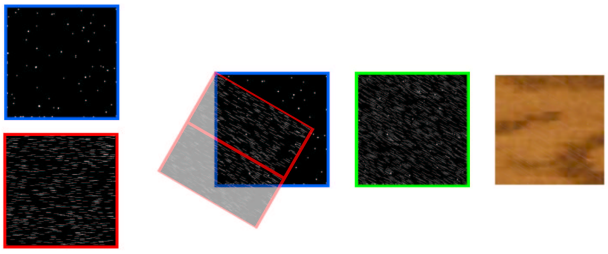


Figure 5: Collapsing two texture operations into a single texture. Left to right, the two initial textures, the two textures transformed and overlaid, the collapsed texture result, and an example of the collapsed texture in use as dust and scratch wood detail.

Substitute: A candidate block identified as implementing a known shading method may be replaced by a simpler method with similar appearance. For example, a bump map can be replaced by a gloss map to modulate the highlight intensity, or a simple texture map (Figure 6). A texture indexed by the surface normal is probably part of a lighting model, and depending on the contents of the texture, may be replaced by the built-in diffuse lighting model. Similarly, a texture indexed by the half angle vector (norm$(V+L)$ for view vector V and light vector L) is a candidate for replacement by one or more applications of the built-in Phong specular model. A texture can be replaced by a smaller low-pass filtered version of the texture and a constant representing the removed high-frequency terms.

Approximate: Approximation rules treat the candidate block as a general function to be approximated. They can theoretically be applied to any block, though not always as effectively as the application-specific rules.

While a variety of function approximation methods are possible, we have focused on ones developed for BRDF approximation [27, 29]. As these methods are texture-based, they are most useful when total texture usage is not the limiting factor. Two issues prevent our approximation rules from being more generally useful, though we believe they are aspects of the approximations we chose to explore and not all applicable function approximation methods.

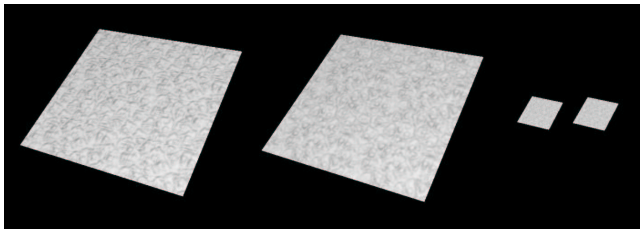


Figure 6: Replacing a bump map with a texture. Left to right, the original bump map, the bump texture at full scale, and the bump map and texture at the expected switching distance.

First, these approximations are based on a factorization into products or sums of products of functions of two variables that can be stored in a texture. In the right coordinate system, BRDFs are well suited to this factorization, usually requiring only one or two terms. Automatic simplification calls for automatic determination of a coordinate system. Arbitrary shading expressions can also be poorly suited to such a factorization in any coordinate system, allowing no acceptable approximation by the homomorphic factorization method, or needing so many SVD terms as to become more expensive than the original expression.

Second, the least squares or singular value decomposition problems are stated in terms of matrices with a number of rows and columns equal to the total number of texels in each approximating texture. Computing these textures rapidly scales to gigabytes, even for modest component texture sizes. Worse, we want to speculatively compute the approximations to evaluate their fitness. The original application to BRDFs limited the component texture sizes to 32x32 or 64x64 resulting in computations with 1024x1024 to 4096x4096 matrices.

3.3 Level Parameters

Selection of simplified verses unsimplified blocks is based on one or several level parameters. For example, switching from a band-limited noise texture to a constant value should happen when the changes in the noise texture are no longer visible (Figure 4). That point can be approximated based either on the distance or screen size of the object. The same transition can also be triggered by a lack of available rendering time, or a lack of available texture memory to store the noise texture.

To manage these different level parameters, we can associate a range for each parameter with each simplified block. Using the noise example above, a constant should be used instead of the noise texture whenever the available texture memory is less than the size of the texture, or there is not enough time to render another texture, or the expected mapping to screen pixels will blur the band-limited noise away.

3.4 Assemble

Given the simplified blocks and level parameter ranges, it is straightforward to assemble them with appropriate conditionals into an LOD shader. Rendering-metric level parameters, like distance or screen coverage, are shared by all blocks in the shader, each emitting a statement of the form

```
if(distance < low_threshold)
  do_simplified_block
else if(distance < high_threshold)
  do_transition_block
else
  do_original_block
```

For resource-accounting level parameters (e.g. available time or texture memory) the blocks are prioritized, and comparisons are emitted for the total consumed by this block and all higher priority blocks.

Shader	Level 1	Level 2	Level 3
Plastic (Collapse)	36.4, 27.6	44.5, 34.4	—, —
Wood (Remove)	18.4, 11.6	18.9, 11.9	19.1, 64.3
Leather (Replace)	25.4, 14.1	43.7, 25.3	79.8, 64.3

Table 1: Result times for test LOD shaders on the 1772 triangle chair model performed on an SGI Octane MXE. Each table entry includes frames-per-second for a small window size, and a large window size with 4x the rendered pixels.

Shader	Level 1	Level 2	Level 3
Plastic (Collapse)	52.9, 33.8	68.2, 42.1	—, —
Wood (Remove)	20.7, 9.2	23.0, 10.0	25.2, 10.7
Leather (Replace)	30.7, 12.3	55.2, 22.8	140.9, 80.3

Table 2: Result times for test LOD shaders on a 3280 triangle draped cloth model consisting of 40 length-82 triangle strips, performed on an SGI Octane MXE. Each table entry includes frames-per-second for a small window size, and a large window size with 4x the rendered pixels.

4 RESULTS

We have described a general theory of shader simplification. Our current results are a modest start within this framework. Specifically, we have produced a set of LOD-aware building block functions for shader construction. This style of shader writing is similar to Abram and Whitted's graphical building-block shader system [1]. Example building-blocks include bump map, a BRDF model, Fresnel reflectance, or noise or turbulence textures with a lookup as used by Hart [21].

Our LOD blocks were created by manually following the steps described in our simplification framework: identify candidate blocks within a building block function, apply one of the simplification rules described in Section 3.2, associate it with a range of an aggregate level parameter, and create conditional blocks for the original code, transition code and simplified code. Despite the manual simplification, we call this semi-automatic because any shaders written using the building blocks, either knowing about level-of-detail or not, become LOD shaders by switching to the LOD building blocks.

Shader	Level 1	Level 2	Level 3
Plastic (Collapse)	9.2, 11.2	11.8, 14.0	—, —
Wood (Remove)	3.6, 5.3	4.1, 5.8	4.5, 6.5
Leather (Replace)	6.4, 8.8	14.7, 18.7	27.7, 35.7

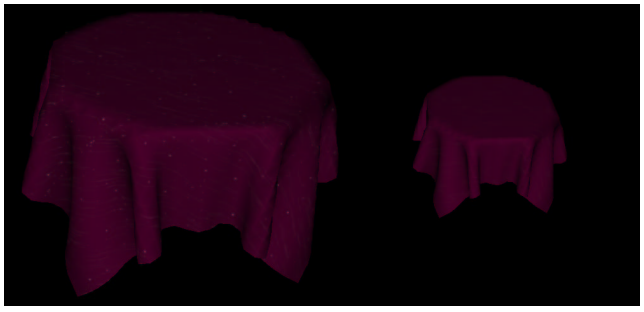
Table 3: Result times for test LOD shaders on the 1772 triangle chair model performed on an SGI O2. Each table entry includes frames-per-second for a small window size, and a large window size with 4x the rendered pixels.

Shader	Level 1	Level 2	Level 3
Plastic (Collapse)	13.6, 15.9	18.2, 20.4	—, —
Wood (Remove)	4.9, 6.9	5.4, 7.6	6.0, 8.5
Leather (Replace)	8.1, 10.3	19.8, 23.9	40.3, 52.3

Table 4: Result times for test LOD shaders on the 3280 triangle draped cloth model performed on an SGI O2. Each table entry includes frames-per-second for a small window size, and a large window size with 4x the rendered pixels.

Tables 1–4 show LOD shader timing in frames per second for several sample LOD shaders. Each shader demonstrates several transitions of specific LOD simplification operations. The Wood shader used in these tests first removes an overlay scratch texture, then removes a specular masking operation, creating three levels-of-detail. Figure 3 shows the removal LOD sequence. The Plastic shader demonstrates the collapse simplification by taking two textures, each applied with its own transformation, and merging these two separate texture passes in a third texture. This resultant texture is then used to shade the object in a single texture for lower levels-of-detail as shown in Figures 5 and 7. The Leather shader demonstrates the replace simplification in the first level-of-detail by replacing a true bump map with a simple texture. The second level in the Leather removes the texture with a simple constant color. Results of this operation sequence are seen in Figure 9.

An overview of the performance results shows much what we would expect — that less detailed shaders result in faster overall rendering. However, as the different results indicate, the shading operations are not purely fill-limited, and rendering nearly 4x fewer pixels in certain cases results in only a modest performance improvement. As certain passes occur, the object's geometry is also re-rendered, yielding a coupling between type of rendering passes constructed for a particular



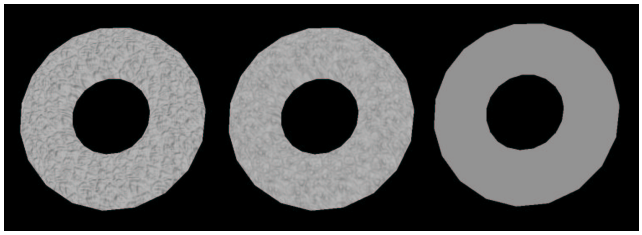


Figure 8: Two replace simplifications in a bumpy leather shader.

shader and that shader's. This implies that LOD shaders can accomplish only part of the task, and should also be accompanied by geometric simplification.

5 CONCLUSIONS AND FUTURE WORK

We have presented LOD shaders: procedural shaders that automatically adjust their level of shading detail for interactive rendering. We also presented a general framework for shader simplification — the process of creating LOD shaders from an ordinary shader. This framework is sufficiently general to serve as a guide for manual shader simplification or as a basis for automatic simplification. Finally, we presented our results for semi-automatic shader simplification using manually generated shading function building blocks for SGI's OpenGL Shader. These LOD shader building blocks implement the same functions as building blocks already provided with OpenGL Shader, but with added level-of-detail parameters to control aspects of their shading complexity.

In the future, we would like to create tools for fully automatic shader simplification. Our current simplification framework also only considers a static analysis of the shader for simplification. Following the lead of texture-based simplification researchers like Aliaga and Shade et al., we could generate new textures on the fly warping them for use over several frames or updating when they become too different [2, 40].

Logically, it should be possible to generalize our remove, collapse and substitution rules into a more widely applicable approximation rule form. Other function fitting methods should be tried to make the approximation rules more useful.

Since rendering with LOD shaders will usually be accompanied by geometric level-of-detail, they should be more closely linked. Cohen et al. Garland and Heckbert and others have shown that geometric simplification can be affected by appearance [8, 17]. Shader simplification should also be affected by geometric level-of-detail (e.g. whether per-vertex Phong shading is a good substitute for a texture-based illumination depends on how the object is tessellated).

Finally, we provide no guarantees on the 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. provide hard bounds on the amount of error introduced by a simplification [9], guarantees that are important for some users. Further investigation is neces-

6 ACKNOWLEDGMENTS

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

We'd also like to thank Dave Shreiner for his helpful comments on the drafts paper.

References

[1] ABRAM, G. D., AND WHITTED, T. Building block shaders. In *Computer Graphics (Proceedings of SIGGRAPH 90)* (Dallas, Texas, August 1990), vol. 24, pp. 283–288. ISBN 0-201-50933-4.

[2] ALIAGA, D. G. Visualization of complex models using dynamic texture-based simplification. In *IEEE Visualization '96* (October 1996), IEEE, pp. 101–106. ISBN 0-89791-864-9.

[3] APODACA, A. A., AND GRITZ, L. *Advanced RenderMan*, first ed. Morgan Kaufmann, 2000.

[4] ATI. *Pixel Shader Extension*, 2000. Specification document, available from http://www.ati.com/online/sdk.

[5] ATI. *Vertex Shader Extension*, 2001. Specification document, available from http://www.ati.com/online/sdk.

[6] BECKER, B. G., AND MAX, N. L. Smooth transitions between bump rendering algorithms. In *Proceedings of SIGGRAPH 93* (Anaheim, California, August 1993), Computer Graphics Proceedings, Annual Conference Series, pp. 183–190. ISBN 0-201-58889-7.

[7] CABRAL, B., MAX, N., AND SPRINGMEYER, R. Bidirectional reflection functions from surface bump maps. In *Computer Graphics (Proceedings of SIGGRAPH 87)* (Anaheim, California, July 1987), vol. 21, pp. 273–281.

[8] COHEN, J., OLANO, M., AND MANOCHA, D. Appearance-preserving simplification. In *Proceedings of SIGGRAPH 98* (Orlando, Florida, July 1998), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 115–122. ISBN 0-89791-999-8.

[9] COHEN, J., VARSHNEY, A., MANOCHA, D., TURK, G., WEBER, H., AGARWAL, P., JR., F. P. B., AND WRIGHT, W. Simplification envelopes. In *Proceedings of SIGGRAPH 96* (New Orleans, Louisiana, August 1996), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 119–128. ISBN 0-201-94800-1.

[10] COOK, R. L. Shade trees. In *Computer Graphics (Proceedings of SIGGRAPH 84)* (Minneapolis, Minnesota, July 1984), vol. 18, pp. 223–231.

[11] EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. *Texturing and Modeling*, second ed. Academic Press, 1998.

[12] FOURNIER, A. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination* (May 1992), Canadian Information Processing Society, pp. 45–52.

[13] FOURNIER, A. Separating reflection functions for linear radiosity. In *Proceedings of Eurographics Workshop on Rendering* (Dublin, Ireland, June 1995), pp. 296–305.

[14] FRASER, C. W., HANSON, D. R., AND PROEBSTING, T. A. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems 1*, 3 (September 1992), 213–226.

[15] FUNKHOUSER, T. A., AND SÉQUIN, C. H. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of SIGGRAPH 93* (Anaheim, California, August 1993), Computer Graphics Proceedings, Annual Conference Series, pp. 247–254. ISBN 0-201-58889-7.

[16] GARLAND, M., AND HECKBERT, P. S. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH 97* (Los Angeles, California, August 1997), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 209–216. ISBN 0-89791-896-7.

[17] GARLAND, M., AND HECKBERT, P. S. Simplifying surfaces with color and texture using quadric error metrics. In *IEEE Visualization '98* (October 1998), IEEE, pp. 263–270. ISBN 0-8186-9176-X.

[18] GOLDMAN, D. B. Fake fur rendering. In *Proceedings of SIGGRAPH 97* (Los Angeles, California, August 1997), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 127–134. ISBN 0-89791-896-7.

[19] GUENTER, B., KNOBLOCK, T. B., AND RUF, E. Specializing shaders. In *Proceedings of SIGGRAPH 95* (Los Angeles, California, August 1995), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 343–350. ISBN 0-201-84776-0.

[20] HANRAHAN, P., AND LAWSON, J. A language for shading and lighting calculations. In *Computer Graphics (Proceedings of SIGGRAPH 90)* (Dallas, Texas, August 1990), vol. 24, pp. 289–298. ISBN 0-201-50933-4.

[21] HART, J. C., CARR, N., KAMEYA, M., TIBBITTS, S. A., AND COLEMAN, T. J. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. In *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (Los Angeles, California, August 1999), ACM SIGGRAPH / Eurographics / ACM Press, pp. 45–53.

[22] HECKBERT, P., ROSSIGNAC, J., HOPPE, H., SCHROEDER, W., SOUCY, M., AND VARSHNEY, A. Multiresolution surface modeling. In *SIGGRAPH 1997 Course Notes* (August 1997), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley.

[23] HEIDRICH, W., SLUSALLEK, P., AND SEIDEL, H.-P. Sampling procedural shaders using affine arithmetic. 158–176. ISSN 0730-0301.

[24] HOPPE, H. Progressive meshes. In *Proceedings of SIGGRAPH 96* (New Orleans, Louisiana, August 1996), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 99–108. ISBN 0-201-94800-1.

[25] HOPPE, H. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH 97* (Los Angeles, California, August 1997), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 189–198. ISBN 0-89791-896-7.

[26] KAJIYA, J. T. Anisotropic reflection models. In *Computer Graphics (Proceedings of SIGGRAPH 85)* (San Francisco, California, July 1985), vol. 19, pp. 15–21.

[27] KAUTZ, J., AND MCCOOL, M. D. Interactive rendering with arbitrary brdfs using separable approximations. In *Eurographics Rendering Workshop 1999* (Granada, Spain, June 1999), Springer Wein / Eurographics.

[28] KAUTZ, J., AND SEIDEL, H.-P. Towards interactive bump mapping with anisotropic shift-variant brdfs. *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (August 2000), 51–58.

[29] MCCOOL, M. D., ANG, J., AND AHMAD, A. Homomorphic factorization of brdfs for high-performance rendering. In *Proceedings of SIGGRAPH 2001* (August 2001), Computer Graphics Proceedings, Annual Conference Series, ACM Press / ACM SIGGRAPH, pp. 171–178. ISBN 1-58113-292-1.

[30] MICROSOFT. *DirectX Graphics Programmers Guide*, directx 8.1 ed. Microsoft Developers Network Library, 2001.

[31] NVIDIA. *NVIDIA OpenGL Extensions Specifications*, March 2001.

[32] OLANO, M., HART, J. C., HEIDRICH, W., LINDHOLM, E., MCCOOL, M., MARK, B., AND PERLIN, K. Real-time shading. In *SIGGRAPH 2001 Course Notes* (August 2001).

[33] OLANO, M., AND LASTRA, A. A shading language on graphics hardware: The pixelflow shading system. In *Proceedings of SIGGRAPH 98* (Orlando, Florida, July 1998), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 159–168. ISBN 0-89791-999-8.

[34] PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. Interactive multi-pass programmable shading. In *Proceedings of SIGGRAPH 2000* (July 2000), Computer Graphics Proceedings, Annual Conference Series, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, pp. 425–432. ISBN 1-58113-208-5.

[35] PERLIN, K. An image synthesizer. In *Computer Graphics (Proceedings of SIGGRAPH 85)* (San Francisco, California, July 1985), vol. 19, pp. 287–296.

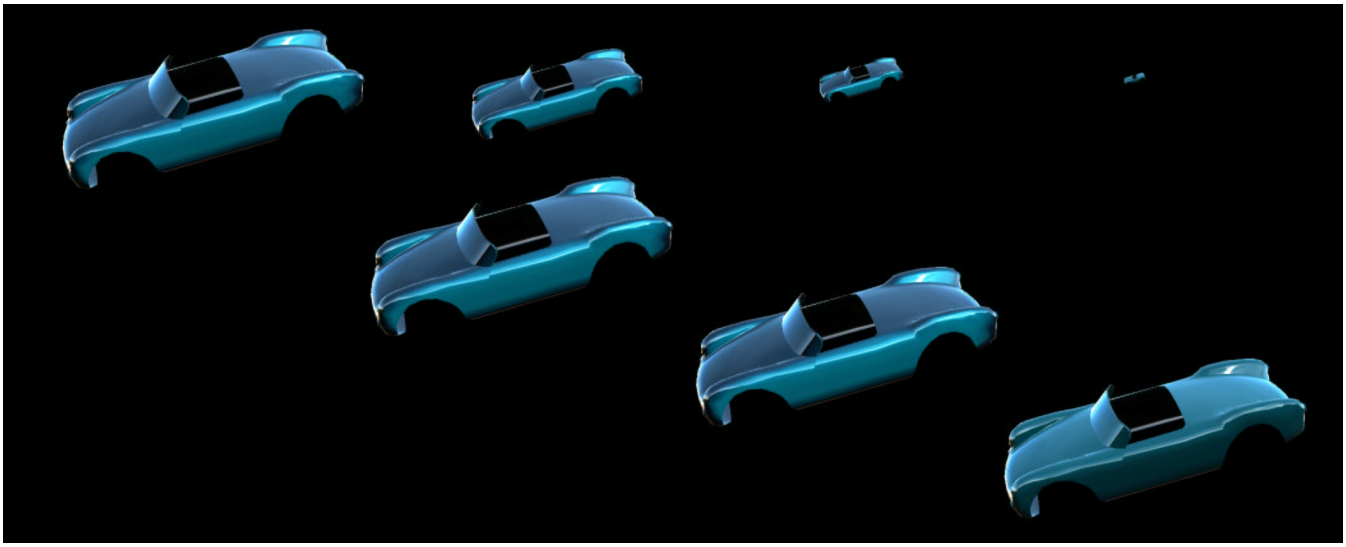


Figure 9: Car paint LOD shader using LOD versions of OpenGL Shader's `microfacetBRDF` and `hdrFresnel` building block functions.

[36] PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of SIGGRAPH 2001* (August 2001), Computer Graphics Proceedings, Annual Conference Series, ACM Press / ACM SIGGRAPH, pp. 159–170. ISBN 1-58113-292-1.

[37] RAMAMOORTHI, R., AND HANRAHAN, P. An efficient representation for irradiance environment maps. In *Proceedings of SIGGRAPH 2001* (August 2001), Computer Graphics Proceedings, Annual Conference Series, ACM Press / ACM SIGGRAPH, pp. 497–500. ISBN 1-58113-292-1.

[38] RHOADES, J., TURK, G., BELL, A., STATE, A., NEUMANN, U., AND VARSHNEY, A. Real-time procedural textures. In *1992 Symposium on Interactive 3D Graphics* (March 1992), vol. 25, pp. 95–100. ISBN 0-89791-467-8.

[39] SCHROEDER, W. J., ZARGE, J. A., AND LORENSEN, W. E. Decimation of triangle meshes. In *Computer Graphics (Proceedings of SIGGRAPH 92)* (Chicago, Illinois, July 1992), vol. 26, pp. 65–70. ISBN 0-201-51585-7.

[40] SHADE, J., LISCHINSKI, D., SALESIN, D. H., DEROSE, T. D., AND SNYDER, J. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proceedings of SIGGRAPH 96* (New Orleans, Louisiana, August 1996), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 75–82. ISBN 0-201-94800-1.

[41] TURK, G. Re-tiling polygonal surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 92)* (Chicago, Illinois, July 1992), vol. 26, pp. 55–64. ISBN 0-201-51585-7.

Interactive Shading Language (ISL) Language Description Version 2.4 March 26, 2002

Contents

I. Introduction

ISL is a shading language designed for interactive display. Like other shading languages, programs written in ISL describe how to find the final color for each pixel on a surface. ISL was created as a simple restricted shading language to help us explore the implications of interactive shading. As such, the language definition itself changes often. While this may be a snapshot specification for ISL, ISL is **not** proposed as a formal or informal language standard. Shading language design for interactive shading is still an open area of research.

A. Features in common with other shading languages

The final pixel color comes from the combined effects of two function types. A *light*

shader computes the color and intensity for a light hitting the surface. Light shaders can be used for ambient, distant and local lights. Several light shaders may be involved in finding the final color for a single pixel. A *surface shader* computes the base surface color and the interaction of the lights with that surface. The term *shader* is used to refer to either of these special types of function.

All shading code is written with a single instruction, multiple data (SIMD) model. ISL shaders are written as if they were operating on a single point on the surface, in isolation. The same operations are performed for all pixels on the surface, but the computed values can be different at every pixel.

Like other shading languages that follow the SIMD model, ISL data may be declared *varying* or *uniform*. Varying values may vary from pixel to pixel, while uniform values must be the same at every pixel on the surface.

B. Major differences from other shading languages

ISL has several differences and limitations that distinguish it from more full-featured shading languages:

- The primary varying data type in ISL is limited to the range [0,1]. Results outside this range are clamped.
- ISL does not allow texture lookups based on computed results.
- ISL does not allow user-defined parameters that vary across the surface. Such parameters must either be computed or loaded as texture.

ISL is also different from most other shading languages in that more than one surface shader may be applied to each surface. The shaders are applied in turn and may composite or blend their results. ISL no longer supports explicit atmosphere shaders. Any light transmission effects between the surface and eye can be handled in the final shader applied to each surface.

II. Files

The appearance of a shaded surface is defined by one or more ISL surface shaders and possibly one or more ISL light shaders. Each shader is defined in its own ISL source files, which should have the file name extension .isl.

A. File contents

Only one shader definition (whether light or surface) can appear in each .isl file. The .isl file may include C preprocessor-like `#include` directives to get access to functions or global variable definitions stored in another file.

Comments in isl may be either C or C++-style (`/*comment*/` or `// comment to end of line`)

B. File compilation

There are two ways to compile a set of ISL files into the rendering passes used to compute surface appearance. The first is to use the ISL run-time library. The second is to use the command line compiler and translator. Both are documented in the shader(1) man page. The ISL Library consists of a set of C++ classes that enable an application to compile that appearance consisting of ISL shaders into an OpenGL stream. The compiled appearance can be associated with geometry from the application, and rendered to an OpenGL rendering context opened by the application. The ISL compiler, islc, converts a set of ISL files into a pass description (.ipf) file. Information on running islc can be found on the islc(1) man page. The pass description file can be converted either to C OpenGL code with the command line translator ipf2ogl (see the ipf2ogl(1) man page), or to a Performer pass file with the command line translator ipf2pf (shipped with Performer 2.4 or later).

III. Data types

All ISL data is classified as either *varying*, *parameter* or *uniform*. Varying data may hold a different value at each pixel. Parameter data must have the same value at every pixel on a surface, but can differ from surface to surface or from frame to frame. Changes to varying or parameter data do not require recompiling the shader. Uniform data also has the same value at every pixel on the surface, but changes to uniform data only take effect when the shader is recompiled.

The complete list of ISL data types is:

uniform float *uf* **parameter float** *pf*	*uf* and *pf* are each a single floating point value
uniform color *uc* **parameter color** *pc*	*uc* and *pc* are each a set of four floating point values, representing a color, vector or point. For colors, the components are ordered red, green, blue and alpha. For points, the components are ordered x,y,z and w.
varying color *vc*	*vc* is a four element color, vector or point that may have different values at each pixel on the surface. Elements of the color are constrained to lie between 0 and 1. Negative values are clamped to zero and values greater than one are clamped to one
uniform matrix *um* **parameter matrix** *pm*	*um* and *pm* are each a set of sixteen floating point values, representing a 4x4 matrix in row-major order (all four elements of first row, all four elements of second row, ...)
uniform string *us*	*us* is a character string, used for texture names.

ISL also allows 1D arrays of all uniform and parameter types, using a C-style specification:

uniform float *ufa*[*n*]	*ufa* is an array with *n* uniform float point elements, *ufa[0]* through *ufa[n-1]*
parameter float *pfa*[*n*]	*ufa* is an array with *n* parameter float point elements, *pfa[0]* through *pfa[n-1]*
uniform color *uca*[*n*]	*uca* is an array with *n* uniform color elements, *uca[0]* through *uca[n-1]*.
parameter color *uca*[*n*]	*pca* is an array with *n* parameter color elements, *pca[0]* through *pca[n-1]*.
uniform matrix *uma*[*n*]	*uma* is an array with *n* uniform matrix elements, *uma[0]* through *uma[n-1]*
parameter matrix *pma*[*n*]	*pma* is an array with *n* parameter matrix elements, *pma[0]* through *pma[n-1]*
uniform string *usa*[*n*]	*usa* is an array with *n* uniform string elements, *usa[0]* through *usa[n-1]*

IV. Variables and identifiers

Identifiers in ISL are used for variable or function names. They begin with a letter, and may be followed by additional letters, underscores or digits. For example a, abc, C93d, and d_e_f are all legal identifiers.

Several variables are predefined with special meaning:

varying color FB	Current frame buffer contents. This is the intermediate result location for almost all varying operations.
parameter matrix shadermatrix	Arbitrary matrix associated with the shader at compile time. This may be used to control the coordinate space where the shader operates.
parameter color lightVector	Within a light shader, the direction the light is shining. This vector may be modified by the light shader. Within a surface shader, the direction of the most recent light.
uniform float pi	The math constant.
uniform float numambientlights	Number of ambient lights in the current islAppearance.
uniform float numdirectlights	Number of direct lights (= both local and distant lights) in the current islAppearance.

V. Uniform operations

In the following, *uf* and *uf0*- *uf15* are uniform floats; *ufa* is an array of uniform floats; *uc*, *uc0* and *uc1* are uniform colors; *uca* is an array of uniform colors; *um*, *um0* and *um1* are uniform matrices; *uma* is an array of uniform matrices; *us*, *us0* and *us1* are uniform strings; *usa* is an array of uniform strings; and *ur*, *ur0* and *ur1* are uniform relations.

A. uniform float

Operations producing a uniform float:

variable reference	Value of uniform float variable.
float constant	One of the following non-case-sensitive patterns: **0x*H*** (hex integer); **0*O*** (octal integer); ***D***; ***D.***; ***.D***; ***D.D***; ***D*e*S*D***; ***D.*e*S*D***; ***.D*e*S*D***; ***D.D*e*S*D*** Where ***H*** = 1 or more hex digits (0-9 or a-f) ***O*** = 1 or more octal digits (0-7) ***D*** = 1 or more decimal digits (0-9) ***S*** = +, - or nothing
(*uf***)**	Grouping intermediate computations.
-*uf*	Negate *uf*
uf0 **+** *uf1*	Add *uf0* and *uf1*
uf0 **-** *uf1*	Subtract *uf1* from *uf0*
uf0 ***** *uf1*	Multiply *uf0* and *uf1*
uf0 **/** *uf1*	Divide *uf0* by *uf1*
*uc***[***uf0***]**	Gives channel *floor(uf0)* of color *uc*, where red is channel 0, green is channel 1, blue is channel 2 and alpha is channel 3.
*um***[***uf0***] [***uf1***]**	Gives element *floor(4*uf0 + uf1)* of matrix *um*
*ufa***[***uf***]**	Element *floor(uf)* of array *ufa* where element 0 is the first element. Behavior is undefined if *floor(uf0)* falls outside the array.
*f***(...)**	Function call to a function returning uniform float result

Uniform float assignments take the following forms, where *lvalue* is either a uniform float variable or a floating point element from a variable (*var[uf0]* for a uniform color or a uniform float array, *var[uf0][uf1]* for a uniform matrix or uniform color array or *var[uf0][uf1][uf2]* for a uniform matrix array):

lvalue **=** *uf*	Simple assignment
lvalue **+=** *uf*	Equivalent to *lvalue = lvalue + uf*
lvalue **-=** *uf*	Equivalent to *lvalue = lvalue – uf*

lvalue ***=** *uf*	Equivalent to *lvalue = lvalue * uf*
lvalue **/=** *uf*	Equivalent to *lvalue = lvalue / uf*

B. uniform color

Operations producing a uniform color:

variable reference	Value of uniform color variable
color(*uf0***,***uf1***,***uf2***,***uf3***)**	*red=uf0*; *green=uf1*; *blue=uf2*; *alpha=uf3*
uf	*color(uf,uf,uf,uf)*
(*uc***)**	Grouping intermediate computations
-*uc* *uc0* **+** *uc1* *uc0* **-** *uc1* *uc0* ***** *uc1* *uc0* **/** *uc1*	Each uniform float operation is applied component-by-component
*um***[***uf***]**	Row *floor(uf)* of matrix *um*
*uca***[***uf***]**	Element *floor(uf)* of array *uca*, where element 0 is the first element. Behavior is undefined if *floor(uf0)* falls outside the array.
*f***(...)**	Function call to a function returning uniform color result

Uniform color assignments take the following forms, where *lvalue* is either a uniform color variable or a color element from a variable (*var[uf0]* for an element of a color array or row of a uniform matrix or *var[uf0][uf1]* for a uniform matrix array):

lvalue **=** *uc*	Simple assignment
lvalue **+=** *uc*	Equivalent to *lvalue = lvalue + uc*
lvalue **-=** *uc*	Equivalent to *lvalue = lvalue – uc*
lvalue ***=** *uc*	Equivalent to *lvalue = lvalue * uc*
lvalue **/=** *uc*	Equivalent to *lvalue = lvalue / uc*

Color elements can also be set individually. See section A above.

C. uniform matrix

Operations producing a uniform matrix:

variable reference	Value of uniform matrix variable

matrix(*uf0*,*uf1*,*uf2*,*uf3*, *uf4*,*uf5*,*uf6*,*uf7*, *uf8*,*uf9*,*uf10*,*uf11*, *uf12*,*uf13*,*uf14*,*uf15***)**	Matrix with rows *(uf0,uf1,uf2,uf3)*, *(uf4,uf5,uf6,uf7)*, *(uf8,uf9,uf10,uf11)* and *(uf12,uf13,uf14,uf15)*
uf	*matrix(uf,0,0,0, 0,uf,0,0, 0,0,uf,0, 0,0,0,uf)*
(*um***)**	Grouping intermediate computations
-*um* *um0* **+** *um1* *um0* **-** *um1*	Each uniform float operation is applied component-by-component
um0 ***** *um1*	Matrix multiplication: *result[i][k] = sum$_{j=0..3}$(um0[i][j] * um1[j][k])*
*uma***[***uf***]**	Element *floor(uf)* of array *uma* where element 0 is the first element. Behavior is undefined if *floor(uf0)* falls outside the array.
*f***(...)**	Function call to a function returning uniform matrix result

Uniform matrix assignments take the following forms, where lvalue is either a uniform matrix variable or one element of a uniform matrix array variable, accessed as *var[uf]*:

lvalue **=** *um*	Simple assignment
lvalue **+=** *um*	Equivalent to *lvalue = lvalue + um*
lvalue **-=** *um*	Equivalent to *lvalue = lvalue – um*
lvalue ***=** *um*	Equivalent to *lvalue = lvalue * um*

Matrix elements can also be set individually. See sections A and B above.

E. uniform string

Operations producing a uniform string:

variable reference	Value of uniform string variable
constant string	String inside double quotes (*"string"*)
*usa***[***uf***]**	Element *floor(uf)* of array *usa* where element 0 is the first element. Behavior is undefined if *floor(uf0)* falls outside the array.

f`(...)`	Function call to a function returning uniform string result

Strings can include escape sequences beginning with '\':

character sequence	name
O	Octal character code
\x*H*	Hex character code
\n	Newline
\t	Tab
\v	Vertical tab
\b	Backspace
\r	Carriage return
\f	Form feed
\a	Alert (bell)
\\	Backslash character
\?	Question mark
\'	Single quote
\"	Embedded double quote

Uniform string assignments take the following forms, where lvalue is either a uniform string variable or one element of an uniform string array variable, accessed by *var[uf]*:

lvalue **=** *us*	Simple assignment

F. uniform relations

Operations producing a uniform relation (used in control statements discussed later):

uf0 **==** *uf1* *uf0* **!=** *uf1* *uf0* **>=** *uf1* *uf0* **<=** *uf1* *uf0* **>** *uf1* *uf0* **<** *uf1*	Traditional comparisons: equal, not equal, greater or equal, less or equal, greater, and less
uc0 **==** *uc1*	True if all elements of *uc0* are equal to the corresponding elements of *uc1*

uc0 **!=** *uc1*	true if any elements of *uc0* does not equal the corresponding element of *uc1*
um0 **==** *um1*	True if all elements of *um0* are equal to the corresponding elements of *um1*
um0 **!=** *um1*	True if any elements of *um0* does not equal the corresponding element of *um1*
us0 **==** *us1* *us0* **!=** *us1*	Traditional string comparison: equal and not equal
(*ur*)	Grouping intermediate computations
ur0 **&&** *ur1*	True if both *ur0* and *ur1* are true
ur0 **||** *ur1*	True if either *ur0* or *ur1* are true
!*ur*	True if *ur* is false

It is not possible to save uniform relation results to a variable.

VI. Parameter operations

In the following, *pf* and *pf0*- *pf15* are parameter floats; *pfa* is an array of parameter floats; *pc*, *pc0* and *pc1* are parameter colors; *pca* is an array of parameter colors; *pm*, *pm0* and *pm1* are parameter matrices; and *pma* is an array of parameter matrices. Also, *uf0* and *uf1* are uniform floats and *uc* is a uniform color as defined above.

A. parameter float

Operations producing a parameter float:

variable reference	Value of parameter float variable.
uf	Convert uniform float to parameter float.
(*pf*)	Grouping intermediate computations.
-*pf*	Negate *pf*
pf0 **+** *pf1*	Add *pf0* and *pf1*
pf0 **-** *pf1*	Subtract *pf1* from *pf0*
pf0 ***** *pf1*	Multiply *pf0* and *pf1*
pf0 **/** *pf1*	Divide *pf0* by *pf1*
pc[*pf0*]	Gives channel *floor(pf0)* of color *pc*, where red is channel 0, green is channel 1, blue is channel 2 and alpha is channel 3.
pm[*pf0*] [*pf1*]	Gives element *floor(4*pf0 + pf1)* of matrix *pm*

pfa[*uf*]	Element *floor(uf)* of array *pfa* where element 0 is the first element. Note that currently the array index must be uniform. Behavior is undefined if *floor(uf0)* falls outside the array.
f(...)	Function call to a function returning parameter float result

Parameter float assignments take the following forms, where *lvalue* is either a parameter float variable or a floating point element from a variable (*var[uf0]* for a parameter float array):

lvalue = *pf*	Simple assignment
lvalue += *pf*	Equivalent to *lvalue = lvalue + pf*
lvalue -= *pf*	Equivalent to *lvalue = lvalue - pf*
lvalue *= *pf*	Equivalent to *lvalue = lvalue * pf*
lvalue /= *pf*	Equivalent to *lvalue = lvalue / pf*

B. parameter color

Operations producing a parameter color:

variable reference	Value of parameter color variable
uc	Convert uniform color to parameter color.
color(*pf0*,*pf1*,*pf2*,*pf3*)	*red=pf0*; *green=pf1*; *blue=pf2*; *alpha=pf3*
pf	*color(pf,pf,pf,pf)*
(*pc*)	Grouping intermediate computations
-*pc* *pc0* + *pc1* *pc0* - *pc1* *pc0* * *pc1* *pc0* / *pc1*	Each parameter float operation is applied component-by-component
pm[*pf*]	Row *floor(pf)* of matrix *pm*
pca[*uf*]	Element *floor(uf)* of array *pca*, where element 0 is the first element. Note that currently the array index must be uniform. Behavior is undefined if *floor(uf0)* falls outside the array.
f(...)	Function call to a function returning parameter color result

Parameter color assignments take the following forms, where *lvalue* is either a parameter color variable or a color element from a variable (*var[uf0]* for an element of

a color array):

lvalue **=** *pc*	Simple assignment
lvalue **+=** *pc*	Equivalent to *lvalue = lvalue + pc*
lvalue **-=** *pc*	Equivalent to *lvalue = lvalue − pc*
lvalue ***=** *pc*	Equivalent to *lvalue = lvalue * pc*
lvalue **/=** *pc*	Equivalent to *lvalue = lvalue / pc*

Unlike uniform colors, parameter colors cannot currently be set by element.

C. parameter matrix

Operations producing a parameter matrix:

variable reference	Value of parameter matrix variable
um	Convert uniform matrix to parameter matrix.
matrix(*pf0***,***pf1***,***pf2***,***pf3***,** *pf4***,***pf5***,***pf6***,***pf7***,** *pf8***,***pf9***,***pf10***,***pf11***,** *pf12***,***pf13***,***pf14***,***pf15***)**	Matrix with rows *(pf0,pf1,pf2,pf3)*, *(pf4,pf5,pf6,pf7)*, *(pf8,pf9,pf10,pf11)* and *(pf12,pf13,pf14,pf15)*
pf	*matrix(pf,0,0,0, 0,pf,0,0, 0,0,pf,0, 0,0,0,pf)*
(*pm***)**	Grouping intermediate computations
-*pm* *pm0* **+** *pm1* *pm0* **-** *pm1*	Each parameter float operation is applied component-by-component
pm0 ***** *pm1*	Matrix multiplication: *result[i][k] = sum$_{j=0..3}$(um0[i][j] * um1[j][k])*
*pma***[***uf***]**	Element *floor(uf)* of array *pma* where element 0 is the first element. Note that currently the array index must be uniform. Behavior is undefined if *floor(uf0)* falls outside the array.
*f***(...)**	Function call to a function returning parameter matrix result

Parameter matrix assignments take the following forms, where lvalue is either a parameter matrix variable or one element of a parameter matrix array variable, accessed as *var[uf]*:

lvalue **=** *pm*	Simple assignment

lvalue **+=** *pm*	Equivalent to *lvalue = lvalue + pm*
lvalue **-=** *pm*	Equivalent to *lvalue = lvalue – pm*
lvalue ***=** *pm*	Equivalent to *lvalue = lvalue * pm*

Unlike uniform matrices, parameter matrices cannot currently be set by element.

D. Parameter relations

Operations producing a parameter relation closely parallel the uniform relations covered earlier. They can be used in control statements discussed later:

pf0 **==** *pf1* *pf0* **!=** *pf1* *pf0* **>=** *pf1* *pf0* **<=** *pf1* *pf0* **>** *pf1* *pf0* **<** *pf1*	Traditional comparisons: equal, not equal, greater or equal, less or equal, greater, and less
pc0 **==** *pc1*	True if all elements of *pc0* are equal to the corresponding elements of *pc1*
pc0 **!=** *pc1*	true if any elements of *pc0* does not equal the corresponding element of *pc1*
pm0 **==** *pm1*	True if all elements of *pm0* are equal to the corresponding elements of *pm1*
pm0 **!=** *pm1*	True if any elements of *pm0* does not equal the corresponding element of *pm1*
(*pr***)**	Grouping intermediate computations
pr0 **&&** *pr1*	True if both *pr0* and *pr1* are true
pr0 **||** *pr1*	True if either *pr0* or *pr1* are true
!*pr*	True if *pr* is false

It is not possible to save parameter relation results to a variable.

VII. Varying operations

In the following, *vc* is a varying color. Also, *pf0* and *pf1* are parameter floats and *pc* is a parameter color as defined above.

A. varying color

Operations producing a varying color:

variable reference	Value of varying color variable Note: when a varying variable is used, *texgen* value of -3 is passed to the application geometry drawing function (see the description under *texture()*). While the geometry drawing function may choose to act on this value, OpenGL Shader will set the texture generation mode appropriately.
pc	Convert parameter color to varying, clamping the resulting color to [0,1]. After this conversion, every pixel has its own copy of the color value.

Possible targets for varying assignments are:

FB	All channels of the framebuffer
FB.*C*	Set only some channels, leaving the others alone. *C* is a channel specification, consisting of some combination of the letters *r*,*g*,*b* and *a* to select the red, green, blue and alpha channels. Each letter can appear at most once, and they must appear in order. This can be used to isolate individual channels: *FB.r*, *FB.g*, *FB.b*, *FB.a*, or to select arbitrary groups of channels: *FB.rgb*, *FB.rb*, *FB.ga*.

Varying assignments into the framebuffer can take the following forms, where *lvalue* is *FB* or *FB.C* (as described above):

FB = *f*(...)	Function call to a function returning varying color result All varying functions also implicitly have access to the value of FB when the function is called. Except for certain built-in functions explicitly noted later, varying functions can **only** be assigned directly into all channels of the framebuffer. To combine the results of a varying function with the existing frame buffer contents, you must save the existing frame buffer into a variable. For example:

NO	OK

`FB.r = f();`	`varying color a = FB;`
	`FB = f();`
	`FB.bga = a;`

lvalue = *vc*	Copy *vc* into *lvalue*

lvalue += *vc* *lvalue* -= *vc* *lvalue* *= *vc*	Add, subtract, or multiply *lvalue* and *vc*, putting the result in *lvalue*.

Assignments into varying variables can only take this form:

variable =	**FB**	Copy framebuffer to variable

B. varying relations

Operations producing a varying relation (used in control statements discussed later):

FB[*vf0***] == *vf1* **FB[***vf0***] != *vf1* **FB[***vf0***] >= *vf1* **FB[***vf0***] <= *vf1* **FB[***vf0***] > *vf1* **FB[***vf0***] < *vf1*	Traditional comparisons: equal, not equal, greater or equal, less or equal, greater, and less Performs per-pixel comparison between frame buffer channel *uf0* and reference value *uf1*. Frame buffer channel 0 is red, channel 1 is green, channel 2 is blue and channel 3 is alpha.

It is not possible to save varying relation results to a variable.

VIII. Built-in functions

The following is the set of provided functions returning uniform results.

uniform float abs(uniform float *x***)** **parameter float abs(parameter float** *x***)**	absolute value of x
uniform float acos(uniform float *x***)** **parameter float acos(parameter float** *x***)**	inverse cosine, radian result is between 0 and pi
uniform float asin(uniform float *y***)**	inverse sine, radian result is between -pi/2 and pi/2

parameter float asin(parameter float *y***)**	
uniform float atan(uniform float *f***)** **parameter float atan(parameter float** *f***)**	inverse tangent, radian result is between -pi/2 and pi/2
uniform float atan(uniform float *y***; uniform float** *x***)** **parameter float atan(parameter float** *y***; parameter float** *x***)**	inverse tangent of y/x, radian result is between -pi and pi
uniform float ceil(uniform float *x***)** **parameter float ceil(parameter float** *x***)**	round x up (smallest integer i >= x)
uniform float clamp(uniform float *x***; uniform float** *a***; uniform float** *b***)** **parameter float clamp(parameter float** *x***; parameter float** *a***; parameter float** *b***)**	clamp x to lie between a and b
uniform float cos(uniform float *r***)** **parameter float cos(parameter float** *r***)**	cosine of *r* radians
uniform float exp(uniform float *x***)** **parameter float exp(parameter float** *x***)**	e^x
uniform float floor(uniform float *x***)** **parameter float floor(parameter float** *x***)**	round x down (largest integer i <= x)
uniform matrix inverse(uniform matrix *m***)**	matrix inverse *m*inverse(m) = inverse(m)*m =* identity matrix

parameter matrix inverse(parameter matrix *m*)	
uniform float log(uniform float *x*) **parameter float log(parameter float** *x*)	natural log of x
uniform float max(uniform float *x*; **uniform float** *y*) **parameter float max(parameter float** *x*; **parameter float** *y*)	maximum of x and y
uniform float min(uniform float *f*; **uniform float** *g*) **parameter float min(parameter float** *f*; **parameter float** *g*)	minimum of x and y
uniform float mod(uniform float *n*; **uniform float** *d*) **parameter float mod(parameter float** *n*; **parameter float** *d*)	Remainder of division *n/d* *n - d*floor(n/d)*
uniform matrix perspective(uniform float *d*) **parameter matrix perspective(parameter float** *d*)	matrix to perform perspective projection looking down the Z axis with a field of view of *d* degrees. *matrix(cotan(d/2),0, 0, 0,* *0, cotan(d/2),0, 0,* *0, 0, 1, 1,* *0, 0, -2,0)*
uniform float pow(uniform float *x*; **uniform float** *y*) **parameter float pow(parameter float** *x*; **parameter float** *y*)	x^y
uniform matrix rotate(uniform float *x*; **uniform float** *y*; **uniform float** *z*; **uniform float** *r*)	rotate *r* radians around axis *(x,y,z)*

parameter matrix **rotate(parameter float** *x***;** **parameter float** *y***;** **parameter float** *z***;** **parameter float** *r***)**	
uniform float round(uniform **float** *x***)** **parameter float** **round(parameter float** *x***)**	round x to the nearest integer
uniform matrix scale(uniform **float** *x***; uniform float** *y***;** **uniform float** *z***)** **parameter matrix** **scale(parameter float** *x***;** **parameter float** *y***;** **parameter float** *z***)**	*matrix(x,0,0,0, 0,y,0,0, 0,0,z,0, 0,0,0,1)*
uniform float sign(uniform **float** *x***)** **parameter float** **sign(parameter float** *x***)**	sign of x: -1, 0 or 1
uniform float sin(uniform **float** *r***)** **parameter float** **sin(parameter float** *r***)**	sine of *r* radians
uniform float **smoothstep(uniform float** *a***; uniform float** *b***;** **uniform float** *x***)** **parameter float** **smoothstep(parameter** **float** *a***; parameter float** *b***; parameter float** *x***)**	smooth transition between 0 and 1 as *x* changes from *a* to *b*. 0 for *x* < *a*, 1 for *x* > *b*
uniform color spline(uniform **float** *x***; uniform color** *c***[])** **uniform float spline(uniform** **float** *x***; uniform float** *c***[])**	evaluate Catmull-Rom spline at *x* based on control point vector, *c*. A Catmull-Rom spline passes through all of the control points. The derivative of the curve at each control point is half the difference between the next and previous control points. The full curve is covered between *x=0* and *x=1*

parameter color **spline(parameter float** *x***;** **parameter color** *c***[])** **parameter float** **spline(parameter float** *x***;** **parameter float** *c***[])**	
uniform float sqrt(uniform **float** *x***)** **parameter float** **sqrt(parameter float** *x***)**	square root of *x*
uniform float step(uniform **float** *a***; uniform float** *x***)** **parameter float** **step(parameter float** *a***;** **parameter float** *x***)**	0 for *x<a* 1 for *x>=a*
uniform float tan(uniform **float** *r***)** **parameter float** **tan(parameter float** *r***)**	tangent of *r* radians
uniform matrix **translate(uniform float** *x***;** **uniform float** *y***; uniform** **float** *z***)** **parameter matrix** **translate(parameter float** *x***; parameter float** *y***;** **parameter float** *z***)**	*matrix(1,0,0,0, 0,1,0,0, 0,0,1,0,* *x,y,z,1)*

The following is the set of provided functions returning varying color results.

varying color texture(**uniform string** *texturename[***;** **parameter matrix** *xform[***;** **uniform float** *texgen]]***)** **varying color texture(** **uniform float**	Map texture onto surface, using texture coordinates defined with object geometry. Versions with array textures are 1D texturing only (using the *s* texture coordinate). Optional float *texgen* (>= 0) is passed to the geometry drawing function so it can generate a different (application defined) set of per-vertex texture coordinates. If *texgen* is not given, a value of 0 will be passed to the geometry drawing function.

<table>
<tr>
<td>
texturearray[][;
parameter matrix
xform[;
uniform float
texgen]])

varying color texture(
uniform color
texturearray[][;
parameter matrix
xform[;
uniform float
texgen]])
</td>
<td>
Optional matrix xform is a matrix for transforming the texture coordinates. If xform is not given, the identity matrix is used (i.e. texture coordinates are used as given).

Note: negative texgen values are used for built-in texture generation modes. These negative values are also passed to the geometry drawing function. While the geometry drawing function may choose to act on these value, OpenGL Shader will set the texture generation mode appropriately.
<table>
<tr><th>texture use</th><th>texgen code</th></tr>
<tr><td>texture()</td><td>>= 0</td></tr>
<tr><td>project()</td><td>-1</td></tr>
<tr><td>environment()</td><td>-2</td></tr>
<tr><td>varying variable use</td><td>-3</td></tr>
</table>
</td>
</tr>
<tr>
<td>
varying color
environment(
uniform string
texturename[;
parameter matrix
xform])

varying color
environment(
uniform float
texturearray[][;
parameter matrix
xform])

varying color
environment(
uniform color
texturearray[][;
parameter matrix
xform])
</td>
<td>
Map texture onto surface, as a spherical environment map. Versions with array textures are 1D texturing only (using the s texture coordinate).

Optional matrix xform is a matrix for transforming the texture coordinates. For example, it can be used to set the map up direction. If xform is not given, the identity matrix is used (i.e. texture coordinates are used as generated).

Note: environment also passes a texgen value of -2 to the application geometry drawing function.
</td>
</tr>
<tr>
<td>
varying color project(
uniform string
texturename[;
parameter matrix
xform])

varying color project(
uniform float
texturearray[][;
</td>
<td>
Project texture onto surface using parallel projection down the Z axis. Versions with array textures are 1D texturing only (using the X coordinate only).

Optional matrix xform is a matrix for transforming before projection. For example, to project in shader space, use inverse(shadermatrix). If xform is not given, the identity matrix is used.
</td>
</tr>
</table>

parameter matrix *xform]*) **varying color project(** **uniform color** *texturearray*[]*[*; **parameter matrix** *xform]*)	Note: *project()* also passes a *texgen* value of -1 to the application geometry drawing function.
varying color transform(parameter matrix *xform*)	Transform the varying color in the frame buffer by the given matrix
varying color lookup(parameter float *lut*[]) **varying color lookup(parameter color** *lut*[])	Lookup each frame buffer channel in the given lookup table. Each channel is handled independently, so the resulting red component of the result comes from the red component *lut[n*FB.r]*. Similarly, for green from *lut[n*FB.g]* and blue from *lut[n*FB.b]*
varying color blend(varying color *v*)	Channel by channel blend: *FB*(1−v) + v = v*(1−FB) + FB*
varying color over(varying color *v*)	Alpha-based blend of *FB* over *v*: *v*(1−FB.a) + FB*FB.a*
varying color under(varying color *v*)	Alpha-based blend of *FB* under *v*: *FB*(1−v.a) + v*v.a*
varying color setupLight(**parameter float** *lightnum*)	Configure a specific light for subsequent diffuse or specular calculations. After being called, the global *lightVector* is set with the current light's position. Light shaders can modify *lightVector* within their body
varying color ambient()	Return sum of ambient light hitting surface
varying color ambient(**uniform float** *lightnum*)	Return result of ambient light *lightnum* If *lightnum*<0 or *lightnum*>=*numambientlights*, ambient() returns black
varying color diffuse()	Return sum of diffuse light hitting surface
varying color diffuse(**uniform float** *lightnum*)	Return result of diffuse contribution from light *lightnum* If *lightnum*<0 or *lightnum*>=*numdirectlights*, diffuse() returns black diffuse(*lightnum*) is equivalent to

	setupLight(lightnum); *runDiffuse(lightVector);*
varying color runDiffuse(parameter color *lvector* **)**	Calculate diffuse effects of previously configured light (configured by using *setupLight*). Accepts a parameter *lvector* to specifiy the light position. Use the global *lightVector* to accept the value set by previous code or the *setupLight* routine. test
varying color specular(parameter float *e***)**	Return sum of specular light hitting surface, using *e* as the exponent in the Phong lighting model
varying color specular(uniform float *lightnum*, **parameter float** *e***)**	Return result of specular contribution from light *lightnum* If *lightnum*<0 or *lightnum*>=*numdirectlights*, specular() returns black specular(*lightnum*, *e*) is equivalent to *setupLight(lightnum);* *runSpecular(e,lightVector);*
varying color runSpecular(parameter float *e***; parameter color** *lvector* **)**	Calculate specular effects of previously configured light (configured by using *setupLight*). Accepts the parameter *e* as the exponent in the Phong lighting model.Accepts a parameter *lvector* to specifiy the light position. Use the global *lightVector* to accept the value set by previous code or the *setupLight* routine.

IX. Variable declarations

A variable declaration is a type name followed by one (and only one) variable name. Each variable name may optionally be followed by an initial value. Some examples:

```
uniform float fvar;
uniform float farray[3];
uniform float fvar = 3;
parameter matrix = 1;
uniform string = "mytexture"
varying color cvar;
```

Variable and functions names are bound using static scoping rules similar to *C*. The same name cannot occur more than once within the same block of statements (bounded by '*{*' and '*}*'), but can be redefined within a nested block:

not legal	legal

{
  uniform float x;
  uniform float x;
}
	{
	  uniform float x;
	  {
	  uniform color x;
	  }
	}

X. Statements

In the following, *uf* is a uniform float, *ur* is a uniform relation and *vr* is a varying relation as defined above.

Legal ISL statements are:

assignment;	Performs assignment
variable declaration;	Creates and possibly initializes variable
{*list of 0 or more statements*}	Executes statements sequentially
if (*ur*) *statement* **if** (*pr*) *statement*	Execute statement only if uniform relation *ur* or parameter relation *pr* is true
if (*ur*) *statement* **else** *statement* **if** (*pr*) *statement* **else** *statement*	Execute first statement if *ur* or *pr* is true, and second statement if *ur* or *pr* is false.
if (*vr*) *statement*	Restricts the currently active set of pixels to those where the given varying relation is true. The active set of pixels starts as all visible pixels within the shaded object, but may be restricted by one or more *if* statements. Note: Any variable of any type assigned inside a varying *if* should only be used inside the *if*. The contents outside the *if* are undefined, and may change from release to release. Assignments into FB are still OK.
if (*vr*) *statement* **else** *statement*	The first statement executes with the same restricted set of pixels as the previous *if* statement. The second statement executes with the active pixels restricted to those that were active when the *if* statement was reached but where the varying relation was false. Note: Any variable of any type assigned inside a varying *if* should only be used inside the *if*. The contents outside the *if* are undefined, and may change from release to release. Assignments into FB are still OK.

repeat (*uf*)
  statement
 	
repeat (*pf*)
  statement
	repeat statement *max(0,floor(uf))* or *max(0,floor(pf))* times.

XI. Functions

Every function has this form:
type function_name(formal_parameters) { body }

The type is one of the ordinary types or a shader type:

surface	Surface appearance. Should compute the base surface color and lighting contribution (though calls to *ambient()*, *diffuse()* and *specular()*).
atmosphere	Equivalent to surface. Atmospheric effects like fog are handled in the last surface shader in the shader list.
ambientlight	Light contributing to *ambient()* function.
distantlight **pointlight**	*distantlight* is a light shining down the z axis. It is transformed by *shadermatrix*, which can be used by the application to point the light in other directions. Within the body of a *distantlight*, *lightVector* gives the light direction. It is initialized to *shadermatrix[2]*, but can be changed by the shader. *pointlight* is a light positioned at the origin. It is transformed by *shadermatrix*, which can be used by the application to point the light in other directions. Within the body of a *pointlight*, *lightVector* gives the light direction. It is initialized to *shadermatrix[3]*, but can be changed by the shader. Distant and point lights return the varying color and intensity of light falling on a surface. They do not compute the interaction of light with the surface itself, that interaction is computed in the surface shader through the *diffuse()* and *specular()* functions, or through *setupLight()* and *runDiffuse()* and *runSpecular*

The set of formal parameter declarations are a semi-colon separated list of uniform variable declarations, with initial values. *Initial values are required for all formal parameters*. For shaders, the initial values are interpreted as defaults for any variable not set explicitly by the application. Arrays in the formal parameter list for a shader are not currently visible to the application. The initial values for parameters of ordinary functions are not currently used, but they are still required.

The body is just a list of statements. The result of each shader is just the value left in *FB* when the shader exits.

The last statement of any function should be the special statement **return** *value;*.

The *return* statement can only appear as the last statement in a function, and the type of *value* should match the function type. For functions returning a varying color, the *return* is optional. If *return* is omitted on a varying color function, the function return value is the value of *FB* at the end of the function.

Surface shaders return a varying color giving the final color of the surface. At the start of the shader, *FB* contains the color of the closest surface previously seen at each pixel. Shaders with transparency should handle any blending with this existing color. In order for surfaces with varying opacity to work, it is also necessary that the application and/or scene graph sort transparent surfaces, and surfaces with varying opacity should be treated as transparent.

Atmosphere shaders start with *FB* set to the final rendered color for each pixel. They return the attenuated color.

An example shader:

```
surface shadertest(
    uniform color c = color(1,0,0,1);
    uniform float f = .25)
{
    FB = diffuse();
    FB *= c*f;
    return FB;
}
```