Chapter 4

SGI

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).

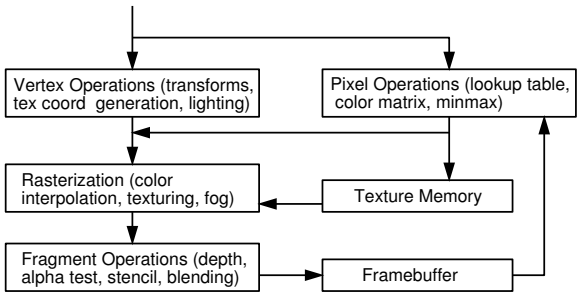


Figure 1: A simplified block diagram of the OpenGL architecture. Geometric data passes through the vertex operations, rasterization, and fragment operations to the framebuffer. Pixel data (either from the host or the framebuffer) passes through the pixel operations and on to either texture memory or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

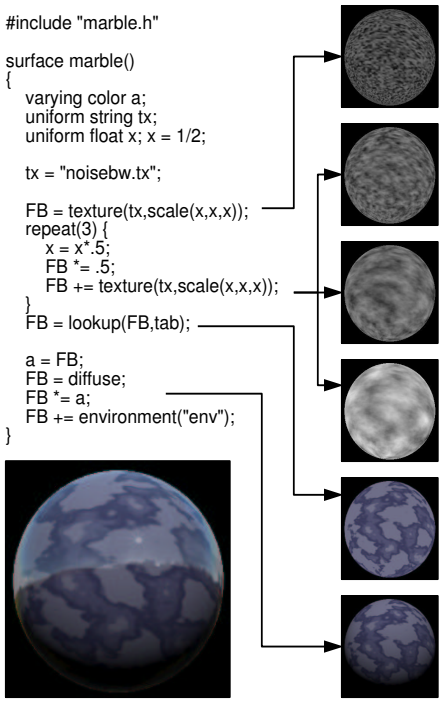


Figure 2: SIMD Computation of a Shader. Some of the different passes for the shader written in ISL listed on the left are shown as thumbnails down the right column. The result of the complete shader is shown on the lower left.

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.

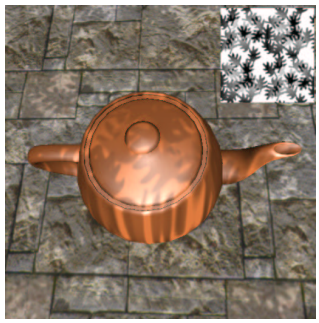
The scene graph passes information to the compiler including the matrix to transform from the object's coordinate system into camera space and the screen space footprint for the geometry. The footprint is computed during the ShadeAction by projecting a 3D bounding box of the geometry into screen space 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.

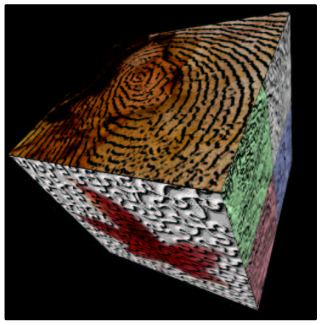
3 EXAMPLE: INTERACTIVE SL

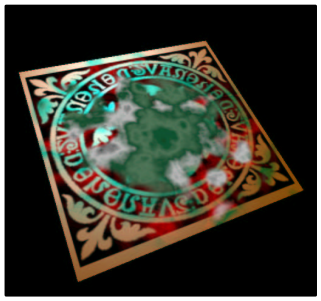
We have developed a constrained shading language, called ISL (for Interactive Shading Language) [25] and an ISL compiler to demonstrate our method on current hardware. ISL is similar in spirit to the RenderMan Shading Language in that it provides a `C`-like syntax to specify per-pixel shading calculations, 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

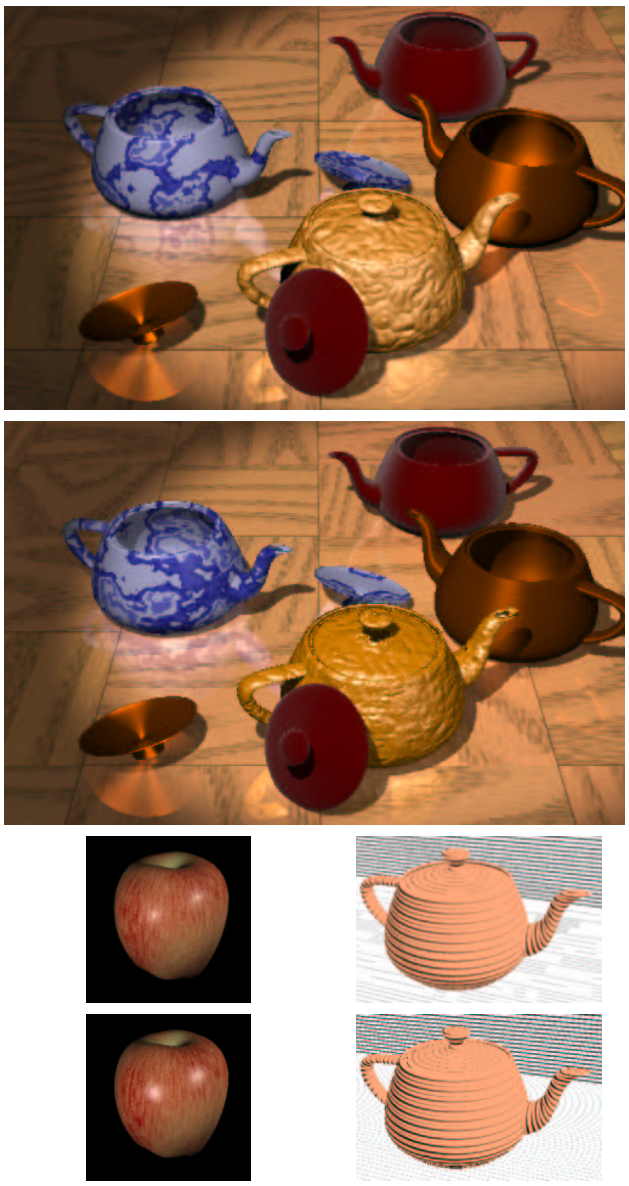


Figure 4: RenderMan SL Examples. The top and bottom images of each pair were rendered with PhotoRealistic RenderMan from Pixar and our multi-pass OpenGL renderer, respectively. No shaders use image maps, except for the reflection and depth shadow maps generated on the fly. The wood floor, blue marble, 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.

1 OpenGL Shader

OpenGL Shader compiles shading programs described in its *Interactive Shading Language* (ISL), into multiple rendering passes. The general technique is described in the "Interactive Multi-Pass Programmable Shading", originally published in SIGGRAPH 2000 and included with these notes. In contrast to the systems in the previous two chapters, ISL is conceived as a higher-level cross-platform language for describing shading.

The higher-level aspects mean that the shading language includes high-level constructs like `if`'s and loops. The cross-platform aspect means that any ISL shader will run and produce similar results on any supported platform. In the case of ISL, the common platform is OpenGL 1.1 or later with an assumed subset of the standard *imaging extensions*. Features that cannot be supported by *any* platform meeting the minimum constraints are not included in the language.

OpenGL Shader can and does map operations in the language to many places in the OpenGL pipeline. For example, a single multiply expressed in ISL may be mapped to the OpenGL units for texture environment, lighting, blend, or scale and bias. Further, on platforms with the appropriate extensions, that same multiply may also map to a multitexture blend or a register combiner operation. The basic premise of ISL is that shaders are written as if every operation were a rendering pass, and it is the compiler's job to stuff as many of those simple operations into a single pass as possible.

Since OpenGL Shader is not part of the graphics driver, its shading API sits above the graphics API. It does its work using ordinary OpenGL calls. When an object needs to be rendered (as may happen multiple times when shading using multi-pass rendering), OpenGL Shader calls a geometry-drawing callback function. This allows the application itself to render the object using whatever data structures and OpenGL calls it would normally use for unshaded objects.

However, OpenGL Shader is in the shading language section of this course, not the API section. For more details on the OpenGL Shader API, see the OpenGL Shader man pages or *Real-Time Shading* [3].

In the following sections, we will step through the construction of the example shaders.

2 Shiny Bump Map

The shiny bump map example highlights the run-everywhere nature of ISL. While many hardware platforms support the dependent texturing or pixel texture extensions necessary to do a full environment map based on bumps from a texture, not all do. To maintain the "every shader runs everywhere" requirement, ISL only supports 1D dependent texturing. This can be cast as any of dependent texture, pixel texture or a color table lookup.

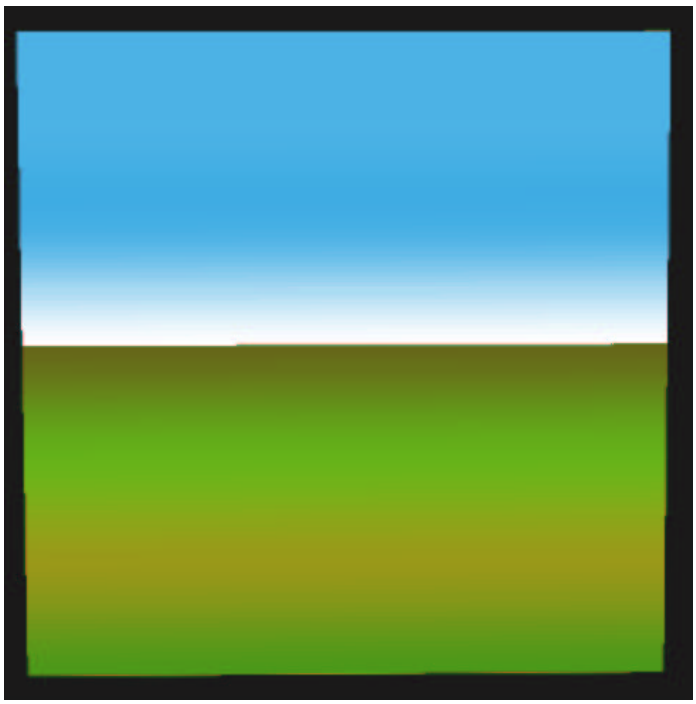


Figure 1: Simple environment

2.1 Environment

Fortunately, it's enough to get a shiny bump map effect with an environment with only one degree of variation: blue above, blending to white near the horizon, then switching to shades of brown and green below the horizon. This 1D map gives the effect of a shiny object, but limits the types of environments that can be used. It is possible to factor some environments into 1D factors, but that is not shown here.

So, the bump mapped environment starts with the 1D environment shown in Figure 1, in this case, defined procedurally by this ISL code:

```
// build 1D reflection map
uniform color groundsky[128];
uniform float i=0; uniform float h=64;
// ground = first h entries
repeat(h) {
    // color spline for ground
    groundsky[i] = spline(i/(h-1),{
        color(.3,.6,.1,1),
        color(.3,.6,.1,1),
        color(.6,.6,.1,1),
        color(.4,.7,.1,1),
```

```
        color(.4,.4,.1,1),
        color(.3,.3,.1,1)});
    i = i+1;
}
// sky = last h entries
repeat(h) {
    // color spline for sky
    groundsky[i] = spline((i-h)/(h-1),{
        color(1.,1.,1.,1),
        color(1.,1.,1.,1),
        color(.3,.7,.9,1),
        color(.3,.7,.9,1),
        color(.3,.7,.9,1),
        color(.3,.7,.9,1)});
    i = i+1;
}
```

Assuming the viewer is sufficiently far away from the object, we can just use the vertical component of the bumped normals as the index into this environment map.

2.2 Bump

For this example, we chose to use the *normal map* style of bump mapping[1]. First, we created textures for the normal as well as S and T tangent vectors (`normalize(dPds)` and `normalize(dPdt)` in RenderMan notation). This was done with a modified draw function that used the `S` and `T` texture coordinates as position and the normal or tangent vectors as color. In effect, unwrapping the object into a the parametric domain to create s normal-map texture patch. These maps for a torus are shown flat and applied to the object in Figures 2, 3 and 4

The next step is to create a bumped normal map. For this purpose, I used the bump map shown in Figure 5. To create a bumped normal map, we must shift the normal at each texel in the S and T tangent directions by an amount proportional to the bump map gradient. This could be done as a loop of computations over the texels, but I chose to use a simple *imgtcl* script (part of the SGI ImageVision Tools package). The results of this script are shown in Figure 6. The script is:

```
set progname [file tail $argv0]

if {$argc != 6} {
    puts stderr "Usage: $progname bump.bw bumpScale norm.rgb dPds.rgb dPdt.rgb bumpnorm.rgb
    Create textures to use for bump mapping from a source image
    The source image, bump.bw, must be a single channel (luminance) image
    The bump scale bumpScale must be a float
    Color images norm, dPds and dPdt contain surface normals, s-tangents and
        t-tangents, with -1..1 vectors components scaled to 0..255
    The output, bumpnorm contains the bumped version of norm.rgb"
```

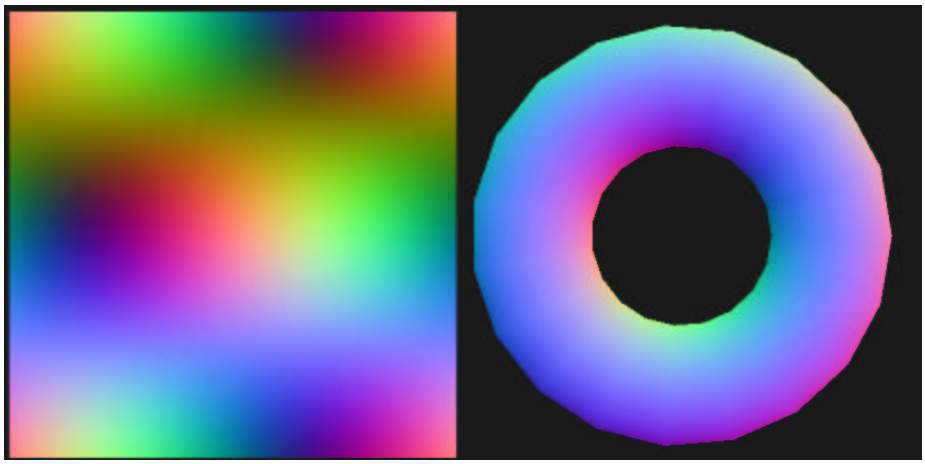


Figure 2: Simple normal map

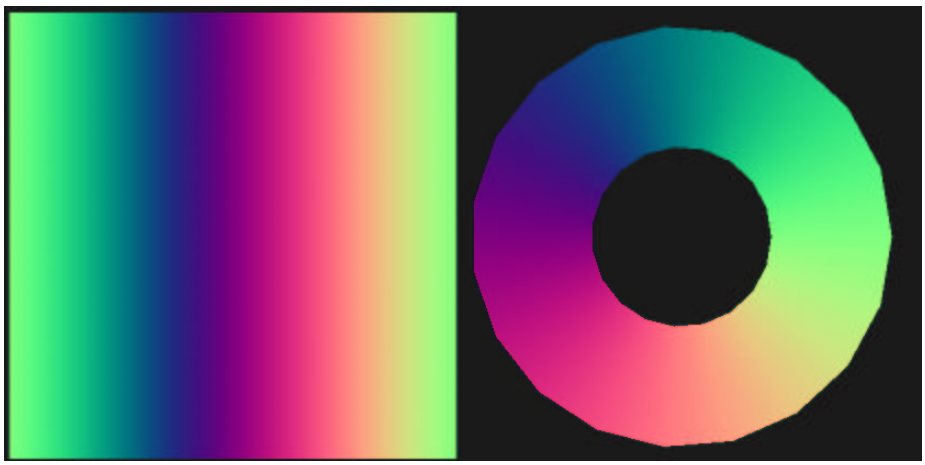


Figure 3: S Tangent map

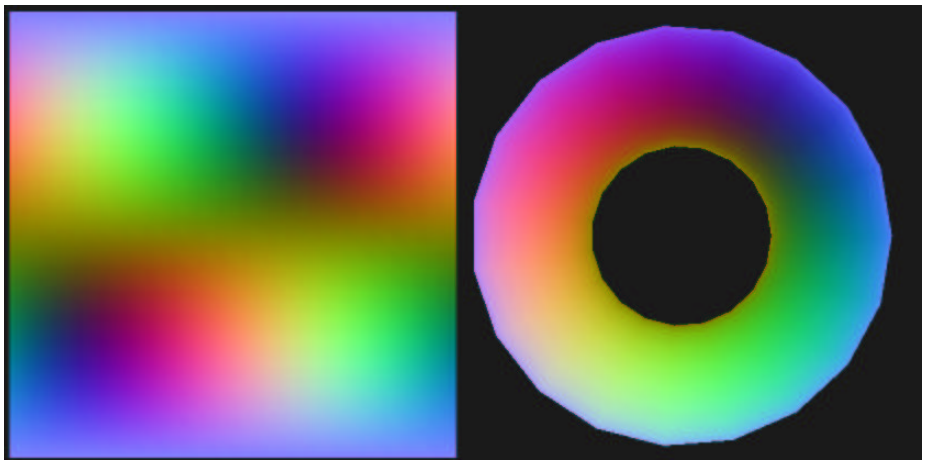


Figure 4: T Tangent map

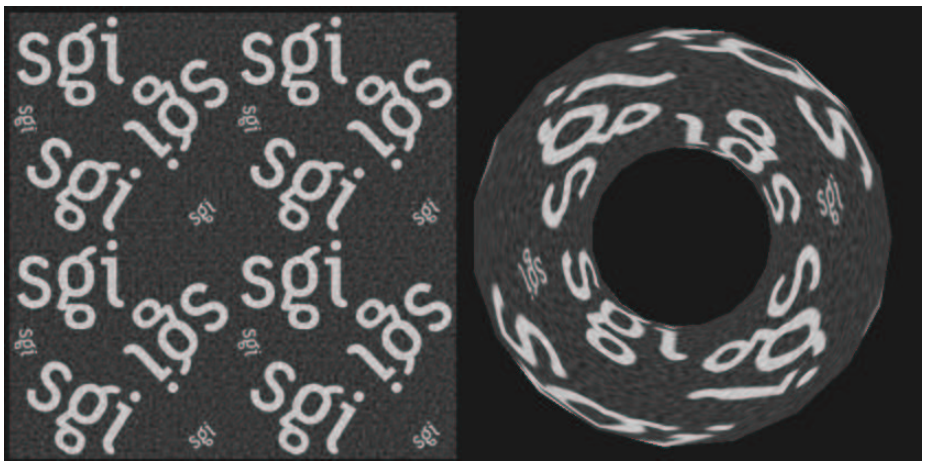


Figure 5: Bump map

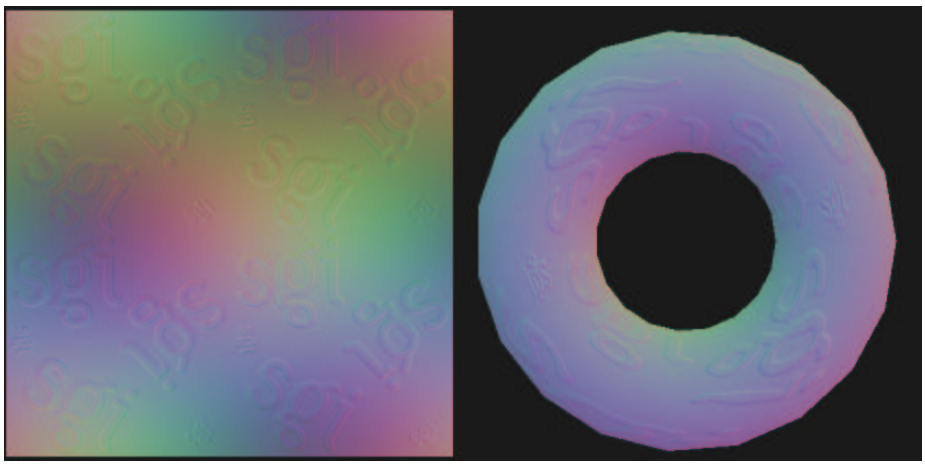


Figure 6: Bump mapped normal map

```
    exit 1
}


set bumpImg   [lindex $argv 0]
set bumpScale [lindex $argv 1]
set normImg   [lindex $argv 2]
set dPdsImg   [lindex $argv 3]
set dPdtImg   [lindex $argv 4]
set nOutImg   [lindex $argv 5]

# open input files
ilFileImgOpen bump $bumpImg
ilFileImgOpen norm $normImg
ilFileImgOpen dPds $dPdsImg
ilFileImgOpen dPdt $dPdtImg

# rescale bump to 0-1
ilScaleImg bumpF bump
bumpF setRange 0 1
bumpF setDataType iflFloat

# x & y components of gradient
new float deriv {3} = "[expr -$bumpScale] 0 $bumpScale"
ilSepKernel sDeriv iflFloat $deriv 3 NULL 1
ilSepKernel tDeriv iflFloat NULL 1 $deriv 3
ilConvImg sSub bumpF sDeriv 0 ilWrap
ilConvImg tSub bumpF tDeriv 0 ilWrap

# rescale norm, dPds and dPdt to -1 to 1
ilScaleImg normF norm
normF setRange -1 1
normF setDataType iflFloat

ilScaleImg dPdsF dPds
dPdsF setRange -1 1
dPdsF setDataType iflFloat

ilScaleImg dPdtF dPdt
dPdtF setRange -1 1
dPdtF setDataType iflFloat

# bumped normal =
# norm + sSub*dPds + tSub*dPdt
ilMultiplyImg sComp sSub dPdsF
ilMultiplyImg tComp tSub dPdtF
```

```
ilAddImg stComp sComp tComp
ilAddImg bumped normF stComp

# recast to 0-255
ilScaleImg nOut bumped
nOut setRange 0 255
nOut setDataType iflUChar

# write as new file
ilFileImgCreate outFile $nOutImg nOut
outFile copy nOut
outFile closeFile

bump closeFile
norm closeFile
dPds closeFile
dPdt closeFile

exit 0
```

2.3 Bump + Environment

The final step is to put the normal map and environment together. First, we must transform the normals in the normal map into world space where our environment map should be applied. We can do this using the ISL transform operation, as seen in this snippet of ISL code:

```
////////////////////
// lookup normal vector
FB=texture(nmap);

////////////////////
// transform normal vector

// rescale normal vectors from 0..1 to -1..1 and back
uniform matrix nScale = translate(-.5,-.5,-.5)*scale(2,2,2);
uniform matrix nUnscale = scale(.5,.5,.5)*translate(.5,.5,.5);

// transform -1..1 normal from object to world space
parameter matrix nm = inverse(affine(shadermatrix));

// set rgb to y (vertical) component and alpha to z (into screen)
// so one lookup will do both color=environment map and alpha=Fresnel
uniform matrix gggb = matrix(0,0,0,0,
                             1,1,1,0,
                             0,0,0,1,
```

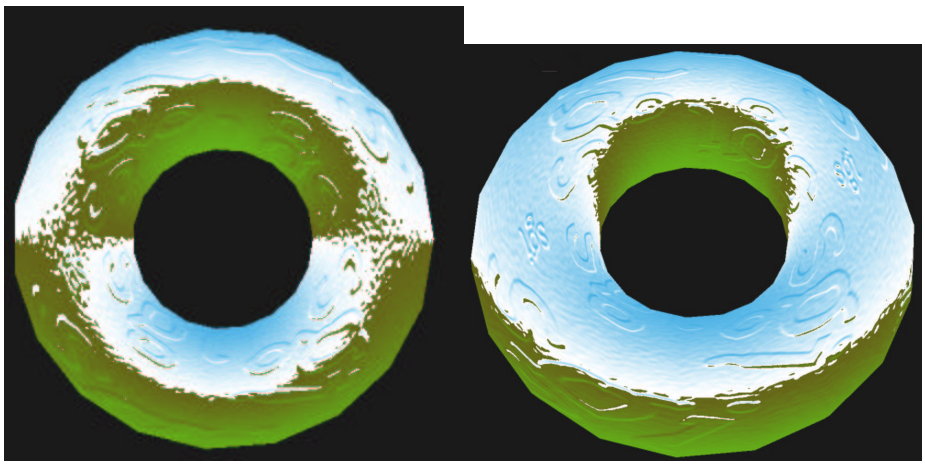


Figure 7: Final shiny/bumpy shader results

```
                                    0,0,0,0);

// transform normal so rgb=y component of world space normal
// a=z compenent of world space normal
FB=transform(nScale * nm * nUnscale * gggb);
```

Finally, we look up the result in the environment map, giving the results shown in Figure 7 from this final shader:

```
surface reflbump(uniform string nmap = "torus_normmap.rgb")
{
    ////////////////////
    // lookup normal vector
    FB=texture(nmap);

    ////////////////////
    // transform normal vector

    // rescale normal vectors from 0..1 to -1..1 and back
    uniform matrix nScale = translate(-.5,-.5,-.5)*scale(2,2,2);
    uniform matrix nUnscale = scale(.5,.5,.5)*translate(.5,.5,.5);

    // transform -1..1 normal from object to world space
    parameter matrix nm = inverse(affine(shadermatrix));

    // set rgb to y (vertical) component and alpha to z (into screen)
    // so one lookup will do both color=environment map and alpha=Fresnel
    uniform matrix gggb = matrix(0,0,0,0,
```

```
                                1,1,1,0,
                                0,0,0,1,
                                0,0,0,0);

    // transform normal so rgb=y component of world space normal
    // a=z compenent of world space normal
    FB=transform(nScale * nm * nUnscale * gggb);


    ////////////////////
    // build and lookup environment map

    // build 1D reflection map
    uniform color groundsky[128];
    uniform float i=0;
    uniform float n=128;
    uniform float h=n/2;
    // ground = first h entries
    repeat(h) {
        // color spline for ground
        groundsky[i] = spline(i/(h-1),{
            color(.3,.6,.1,1),
            color(.3,.6,.1,1),
            color(.6,.6,.1,1),
            color(.4,.7,.1,1),
            color(.4,.4,.1,1),
            color(.3,.3,.1,1)});
        i = i+1;
    }
    // sky = last h entries
    repeat(h) {
        // color spline for sky
        groundsky[i] = spline((i-h)/(h-1),{
            color(1.,1.,1.,1),
            color(1.,1.,1.,1),
            color(.3,.7,.9,1),
            color(.3,.7,.9,1),
            color(.3,.7,.9,1),
            color(.3,.7,.9,1)});
        i = i+1;
    }

    // lookup in map, color=reflection, alpha=fresnel reflectance
    FB = lookup(groundsky);
}
```

3 Homomorphic BRDF Factorization

The shader for the run-time portion of the homomorphic factorization [2] is almost trivial. It uses the *texture code* feature of the ISL texture lookup call to indicate that a different set of texture coordinates is needed for each lookup. In this case, the texture coordinates must be computed on a per-vertex basis. The shader is:

```
surface BRDF(uniform string brdfP = "brdf_p.rgb";
             uniform string brdfQ = "brdf_q.rgb";
     uniform color brdfC = color(1,1,1,1))
{
    FB = diffuse();

    // 1st 1 = identity texture transform matrix
    // 2nd 1 = 1st special texture coordinate set, based on L
    FB *= texture(brdfP, 1, 1);

    // 2 = 2nd special texture coordinate set, based on H
    FB *= texture(brdfQ, 1, 2);

    // 3 = 3rd special texture coordinate set, based on V
    FB *= texture(brdfP, 1, 3);

    FB *= brdfC;
}
```

The extra texture coordinate set number is passed directly to the application's own geometry drawing callback, leaving the application in charge of computing L, H and V in the local tangent space for the three lookups. This is shown as sample application code in the `brdf_viewer` example that is included with OpenGL Shader or using the vertex operation API in the `geometry` code shared by the `viewer_lib` and `editor_iv` examples.

Final results with a basic paint BRDF applied to a car, and the same with a Fresnel-modulated environment layer are shown in Figure 8. The same shader used to render fabric is shown in Figure 9.

4 Procedural Wood

The final example is a procedural wood. Since this wood should have procedural control over a repeating band structure, I started with one of the most basic repeating elements in ISL — the repeating texture. The simple 1D triangle ramp texture shown in the left side of Figure 10, when projected onto an object gives the regular pattern shown in the right side of Figure 10.

This repeating pattern can be used to pick between two basic colors of wood (result shown in Figure 11):

Figure 8: Car with BRDF-based paint (and Fresnel-modulated enviornment layer)

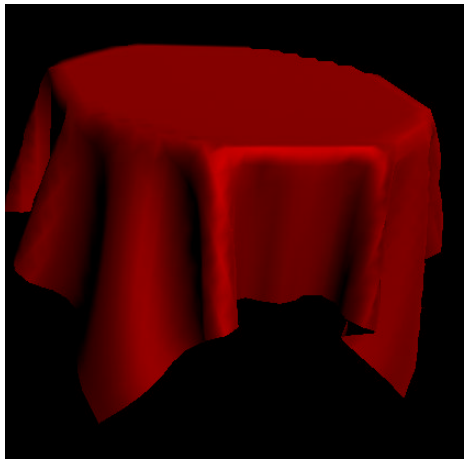


Figure 9: Fabric rendered with factorized BRDF

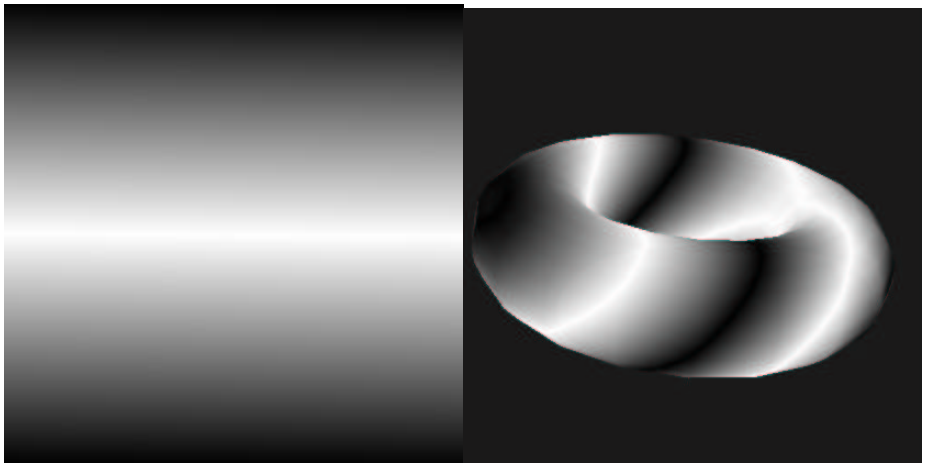


Figure 10: Ramp texture map and pattern when projected onto an object

Figure 11: Using ramp to make simple parameterized color choice

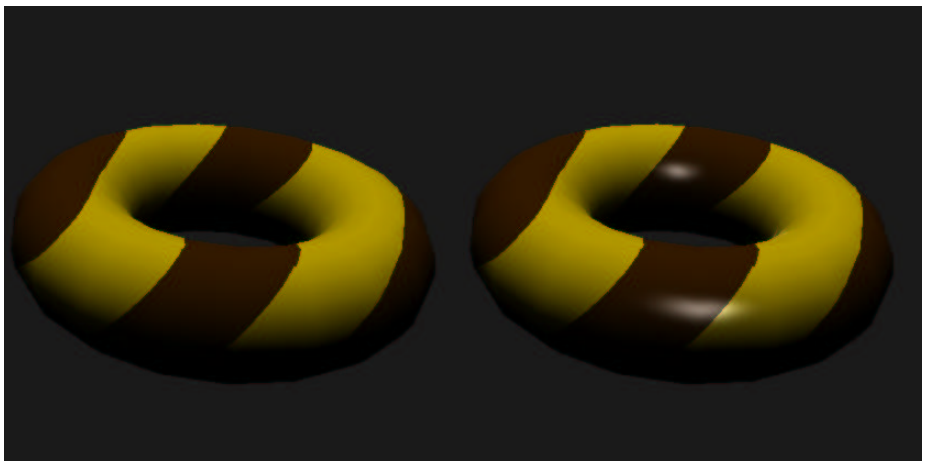


Figure 12: Differing diffuse and specular for each color band

```
FB = project("wave.bw",
     // project in object space
            inverse(shadermatrix)*
     // control over position of rings
            translate(ringCenter[0],ringCenter[1],ringCenter[2])*
     // control over angle of rings
            rotate(ringRotAxis[0],ringRotAxis[1],ringRotAxis[2],
                   ringRotAngle)*
     // control over size of rings
            scale(ringScale,ringScale,ringScale)*
     // center in texture
            translate(.5,.5,.5));

// dark rings
if (FB[0] < lightToDark) {
    FB = darkWood;
}
else {
    FB = lightWood;
}
```

To this, we can add different characteristics for diffuse and specular characteristics in each band (Figure 12):

```
{
    // diffuse color (saved for later)
    FB = diffuse();
    varying color dif=FB;
```

```
    // specular contribution (saved for later)
    FB = environment("highlight.bw");
    varying color spec=FB;

    FB = project("wave.bw",
 inverse(shadermatrix)*
 translate(ringCenter[0],ringCenter[1],ringCenter[2])*
 rotate(ringRotAxis[0],ringRotAxis[1],ringRotAxis[2],
ringRotAngle)*
 scale(ringScale,ringScale,ringScale)*
 translate(.5,.5,.5));

    // dark rings
    if (FB[0] < lightToDark) {
// diffuse color
FB = darkWood;
FB *= dif;
varying color a = FB;

  // specular gloss
FB = darkGloss;
FB *= spec;
FB += a;
    }
    // light rings
    else {
// diffuse color
FB = lightWood;
FB *= dif;
varying color a = FB;

  // specular gloss
FB = lightGloss;
FB *= spec;
FB += a;
    }
}
```

Note that by adjusting darkToLight, we can change the width of dark and light bands on the fly. The boundary between the bands still seems a bit too smooth and regular. This can be alleviated with a repeating turbulence texture, projected at a slight angle to the original band texture. The turbulence texture and its projection are shown in Figure 13

When added to the basic intensity ramp that determines the noise, it makes a nice variation to the band boundaries (Figure 14). I used code like the following to allow

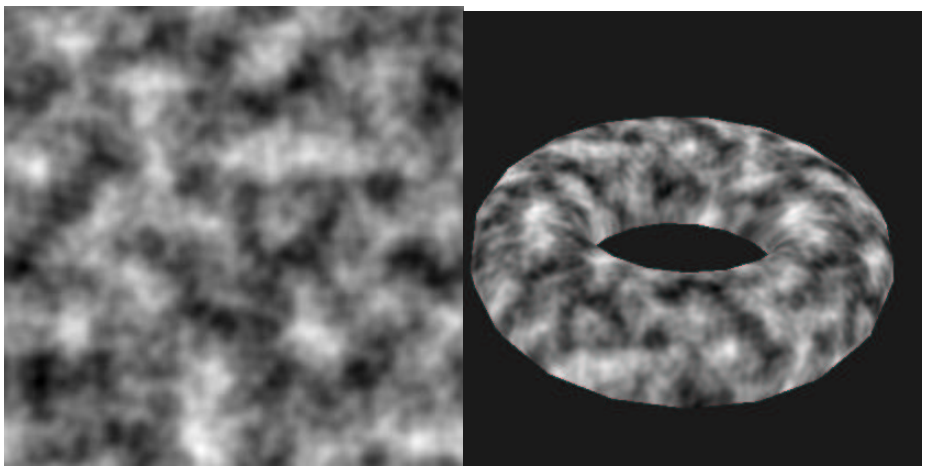


Figure 13: Turbulence texture

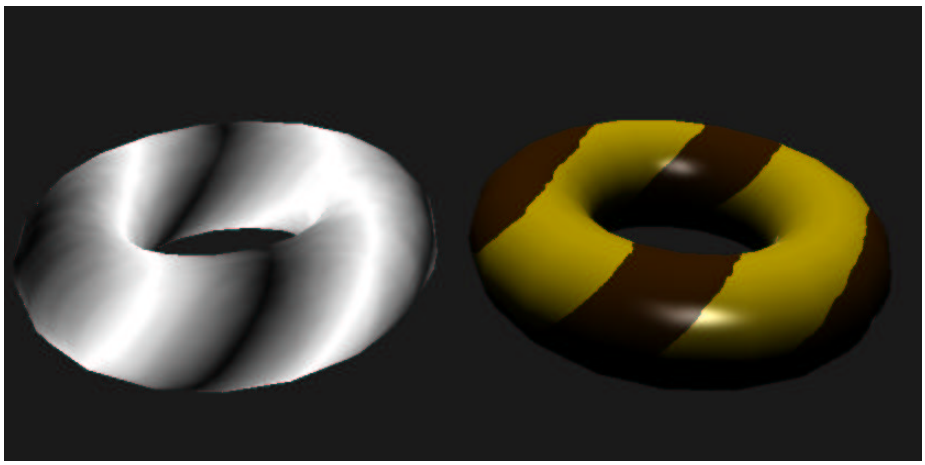


Figure 14: Wood with turbulence added to band boundary

control over the amount of turbulence applied:

```
    // general ring structure: turbulence + triangle wave
    // rings are divided bright vs dark in this structure
    FB = project("turbulence.bw",
 inverse(shadermatrix)*
 scale(ringNoiseScale,ringNoiseScale,ringNoiseScale)*
 translate(ringCenter[0],ringCenter[1],ringCenter[2])*
 rotate(ringRotAxis[0],ringRotAxis[1],ringRotAxis[2],
ringRotAngle+15)*
 scale(ringScale,ringScale,ringScale)*
 translate(.5,.5,.5));
    FB *= ringNoiseStrength;
    FB += project("wave.bw",
 inverse(shadermatrix)*
 translate(ringCenter[0],ringCenter[1],ringCenter[2])*
 rotate(ringRotAxis[0],ringRotAxis[1],ringRotAxis[2],
ringRotAngle)*
 scale(ringScale,ringScale,ringScale)*
 translate(.5,.5,.5));
```

As a final addition, we'll add a fine grain noise for both to both diffuse and specularity within each band. We could use another `if`, but for demonstration purposes, I've chosen to use an alpha blend instead this time. For the fine grain, I'm using a simple noise texture, projected along the same direction as the ring structure, but stretched more along the rings than across (Figure 15). Results shown in Figure 16. Here's the final shader:

```
surface procwood(
```

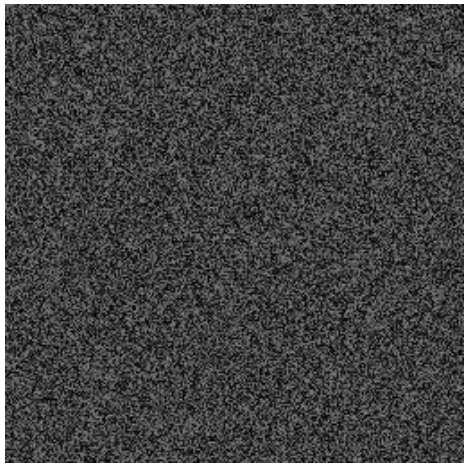


Figure 15: Simple noise texture

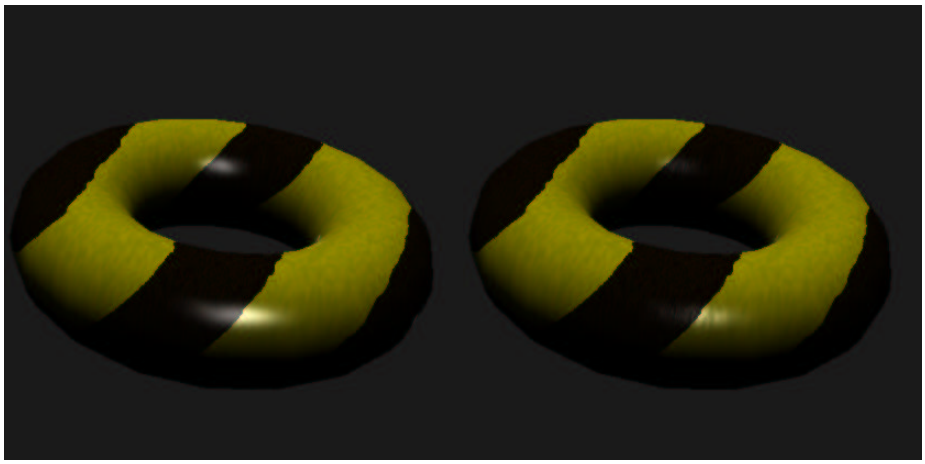


Figure 16: Final wood

```
    parameter float ringScale = 1;
    parameter color ringCenter = color(.5,.5,0,1);
    parameter color ringRotAxis = color(1,0,0,1);
    parameter float ringRotAngle = 15;
    parameter float ringNoiseScale = .6;
    parameter float ringNoiseStrength = .1;

    parameter float lightToDark = .5;

    parameter color darkWood = color(.2,.1,0,1);
    parameter color darkGrain = color(0,0,0,1);
    parameter float darkGloss = .45;
    parameter float darkGrainLong = .25;
    parameter float darkGrainShort = 1;
    parameter float darkGrainGloss = 0;

    parameter color lightWood = color(.7,.6,0,1);
    parameter color lightGrain = color(.5,.5,0,1);
    parameter float lightGloss = .75;
    parameter float lightGrainLong = .1;
    parameter float lightGrainShort = .2;
    parameter float lightGrainGloss = .25
    )
{
    // diffuse color (saved for later)
    FB = diffuse();
    varying color dif=FB;

    // specular contribution (saved for later)
    FB = environment("highlight.bw");
    varying color spec=FB;

    // general ring structure: turbulence + triangle wave
    // rings are divided bright vs dark in this structure
    FB = project("turbulence.bw",
 inverse(shadermatrix)*
 scale(ringNoiseScale,ringNoiseScale,ringNoiseScale)*
 translate(ringCenter[0],ringCenter[1],ringCenter[2])*
 rotate(ringRotAxis[0],ringRotAxis[1],ringRotAxis[2],
ringRotAngle+15)*
 scale(ringScale,ringScale,ringScale)*
 translate(.5,.5,.5));
    FB *= ringNoiseStrength;
    FB += project("wave.bw",
 inverse(shadermatrix)*
 translate(ringCenter[0],ringCenter[1],ringCenter[2])*
```

```
 rotate(ringRotAxis[0],ringRotAxis[1],ringRotAxis[2],
ringRotAngle)*
 scale(ringScale,ringScale,ringScale)*
 translate(.5,.5,.5));

    // dark rings
    if (FB[0] < lightToDark) {
// diffuse color
FB = darkWood;
FB.a = project("noise.bw",
     inverse(shadermatrix)*
     scale(darkGrainLong,darkGrainShort,1)*
     translate(ringCenter[0],ringCenter[1],ringCenter[2])*
     scale(ringScale,ringScale,ringScale)*
 translate(.5,.5,.5));
FB = over(darkGrain);
FB *= dif;
varying color a = FB;

// specular gloss
FB = darkGloss;
FB.a = project("noise.bw",
     inverse(shadermatrix)*
     scale(darkGrainLong,darkGrainShort,1)*
     translate(ringCenter[0],ringCenter[1],ringCenter[2])*
     scale(ringScale,ringScale,ringScale)*
 translate(.5,.5,.5));
FB = over(darkGrainGloss);
FB *= spec;
FB += a;
    }
    // light rings
    else {
// diffuse color
FB = lightWood;
FB.a = project("noise.bw",
     inverse(shadermatrix)*
     scale(lightGrainLong,lightGrainShort,1)*
     translate(ringCenter[0],ringCenter[1],ringCenter[2])*
     scale(ringScale,ringScale,ringScale)*
 translate(.5,.5,.5));
FB = over(lightGrain);
FB *= dif;
varying color a = FB;

// specular gloss
```

```
FB = lightGloss;
FB.a = project("noise.bw",
     inverse(shadermatrix)*
     scale(lightGrainLong,lightGrainShort,1)*
     translate(ringCenter[0],ringCenter[1],ringCenter[2])*
     scale(ringScale,ringScale,ringScale)*
 translate(.5,.5,.5));
FB = over(lightGrainGloss);
FB *= spec;
FB += a;
    }
}
```

References

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

[2] MCCOOL, M. D., ANG, J., AND AHMAD, A. Homomorphic factorization of brdfs for high-performance rendering. In *Proc. ACM SIGGRAPH* (Aug. 2001).

[3] OLANO, M., HART, J., HEIDRICH, W., AND MCCOOL, M. *Real-Time Shading*. AK Peters, 2002.