SIGGRAPH 2002 Course 17 Notes

State of the Art in Hardware Shading

Presenters

Marc Olano
Member of Technical Staff
SGI

Chas Boyd
DirectX Graphics
Microsoft Corp.

Bill Mark
NVIDIA Corp.

Michael McCool
Associate Professor
University of Waterloo

Jason L. Mitchell
Project Team Leader
3D Application Research Group
ATI Research

Randi Rost
Senior Manager, Driver Development
3Dlabs, Inc.

Course Description:

In the past couple of years, simple programmable shading capabilities have appeared on a wide range of graphics hardware. This has sparked great interest across the spectrum of developers, from games to visualization, including several SIGGRAPH papers on using the new features. While the capabilities are similar, there are significant differences in the methods provided program the shading hardware and the interface with an application program. This course provides a practical comparison between hardware shading alternatives and shading APIs. Different platforms are compared using a series of common examples, with tutorial-level details and live demos on each platform.

Prerequisites:

This course assumes working knowledge of a modern real-time graphics API like OpenGL or DirectX. The participants are also assumed to be familiar with the concepts of procedural shading.

Syllabus

I. Shading Hardware
   A. Introduction (Olano - 30 min)
      1. Why do we want real-time procedural shading?
      2. Shading hardware issues
      3. Overview of common examples for all presenters
         a. Shiny bump map
         b. McCool Homomorphic BRDF factorization
         c. Parameterized Wood
   B. NVIDIA (Bill Mark - 60 min)
      1. NVIDIA Shading overview
      2. Details on implementing each of the common examples
      3. NVIDIA specific examples
      4. Demos
   C. ATI (Jason L. Mitchell - 60 min)
      1. ATI Shading overview (15 min)

---

break

---

      2. Details on implementing each of the common examples
      3. ATI specific examples
      4. Demos
   D. SGI (Marc Olano - 60 min)
      1. SGI OpenGL Shader overview
      2. Shading as a library above the API
      3. Details on implementing each of the common examples
      4. SGI OpenGL Shader specific examples
      5. Demos

---

break

II. APIs
   A. DirectX (Chas Boyd - 60 min)
      1. How shading fits into DirectX
      2. Details on implementing each of the common examples
      3. DirectX specific examples
      4. Demos
   B. OpenGL 2.0 (Randi Rost - 60 min)
      1. How shading fits into OpenGL 2.0
      2. Details implementing on each of the common examples
      3. OpenGL 2.0 specific examples

---

break

---

4. Demos (15 min)

C. API design issues (McCool - 60 min)
    1. SMASH: A conceptual design for future programmable graphics accelerators
    2. Flexible parameter binding mechanism
    3. Virtual stack machine shader execution conceptual model
    4. Use of metaprogramming techniques in host language

III. Panel-style Q&A (All - 30 min)

Contents

Chapter 1

Introduction

Marc Olano

1 About This Course

Or, "why do we want to do real-time shading, and why offer a course on it?"

Over the years of graphics hardware development, there have been obvious strides in the geometric complexity of objects that can be rendered in real-time. The first statistic quoted for any new piece of graphics hardware is the number of polygons it can render per second. However, there has also been a steady pace of improvement in appearance for objects rendered in real-time (Figure 1). These improvements are harder to benchmark and tend to come in jumps across the industry. Nonetheless, no one today would seriously consider buy a new graphics system that did only flat shading only.

Compare this to software rendering, where techniques like procedural shading have been in use for 15-20 years [3, 5, 8, 10]. Procedural shading is popular in a large part because of the power it provides to customize the appearance of everything you render by changing the procedures that control that appearance.

In the past few years, we've begun to see graphics hardware that can do some form of procedural shading in real-time. This new freedom in expressing the appearance of rendered objects has excited the imagination of people across the spectrum of interactive graphics users, including everyone from game developers to car designers.

However, the capabilities and ease of use of new real-time shading hardware varies widely. This course is designed to provide a solid comparison of many of the latest offerings. Even as we offer this course, hardware capabilities and software interfaces for them are improving. Any attempt to show the "state of the art", is at best a snapshot. As such, these notes will be but one snapshot, and the course presentations another. While the notes may serve as a starting point, we encourage you to check the web sites of the various course presenters for the latest developments at and after SIGGRAPH.

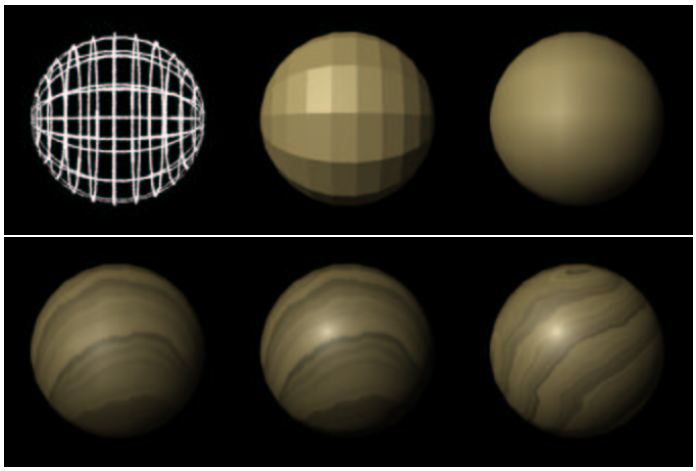


Figure 1: Progression of hardware-accelerated appearance: vector, flat shading, Gouraud shading, 2D Texture + Gouraud shading, 2D Texture + per-vertex Phong, 3D Procedural Shader

The course itself is divided into two major sections. The morning presenters (Bill Mark from NVIDIA, Jason Mitchell from ATI and Marc Olano from SGI) will focus on the how the *shaders* that determine surface appearance are described. The afternoon presenters (Chas Boyd from Microsoft on DirectX, Randi Rost from 3DLabs on OpenGL 2.0 and Michael McCool from the University of Waterloo on API Design and SMASH) will focus on API issues. That is, on the interface for using shaders and shaded objects within an application.

2 The Examples

To provide a common ground for comparison, each presenter in both sections will show three common examples on their latest and greatest system. These will be supplemented by their additional examples for each presenter to show off other important features of their system.

The common examples are not so much a benchmark of performance as a benchmark of ease of use and understanding. By using the same set of examples, course participants will be able to compare the different hardware and software interfaces. On the other hand, we haven't attempted to define the examples too precisely since each system has its strengths and weaknesses. If we'd defined every detail of the examples, we'd run the danger of giving a false comparison by the chance overlap with strengths for some systems and weaknesses for others. Instead, the examples are more roughly defined, giving each presenter the option to target their strengths — they way you'd do it if you were writing the shaders.

Note that not all chapters in these notes show implementation of the examples — in some cases major changes are expected between the course notes deadline and SIGGRAPH. In those cases, these notes include reference material that may continue to prove useful for those platforms.

The following sections describe each of the three common examples, including the problem statement given to the presenters at the outset of the course.

2.1 Shiny Bump Map

> Environment mapped bump mapping ... dependent texturing, everybody seems to like it

The first example combines environment mapping, a common technique for simulating reflection, with bump mapping, a common technique for simulating fine-scale surface features through shading without changing the surface geometry. These two are interesting when put together since both bump map and environment map are results of texturing operations. Put together, they require the results of one texture lookup to influence the texture coordinates used in a second lookup.

It's also included because bumpy-shiny things have become a trite examples on recent graphics hardware, being applied to practically every object in some cases.

The problem statement intentionally avoids specifying exactly how the bump map is computed. The traditional formulation originally proposed by Blinn uses a bump

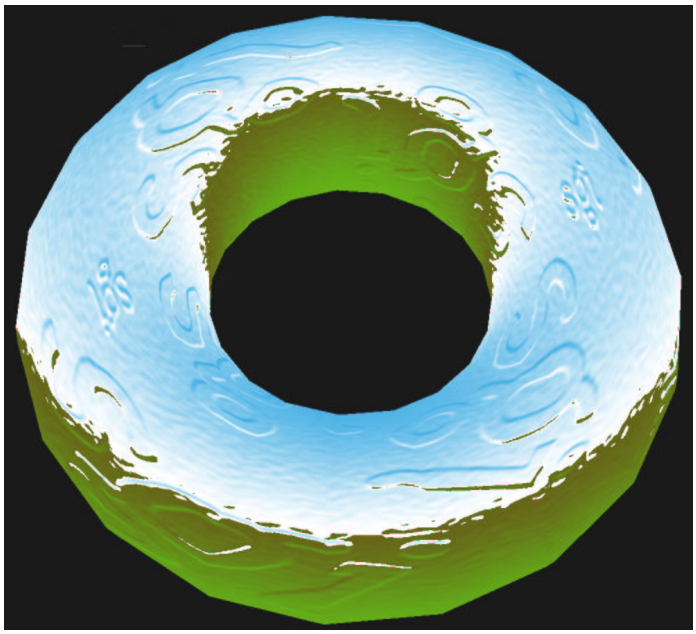


Figure 2: Shiny Bump Map on a low-tessellation torus

Figure 3: Car rendered with Homomorphic BRDF factorization for paint from [7], and again with environment-map based Fresnel reflectance layer on top of BRDF-based paint

texture representing a grey-scale height map of the surface [1]. Changes to the shading normal are determined from the gradients of this bump map texture. Another formulations by Cabral, subtracts shifted versions of the texture in a technique similar to 3D image embossing to compute the bump gradients [2]. Yet another formulation by Fournier uses a texture map containing surface normals (a *normal map*) instead of computing perturbations to the original shading normals [4]. Any of these or other method of computing the bumped surface normals could be used. Also, the shader could compute some other related quantity rather than the bumped normal if it seems more efficient.

The problem statement also avoids specifying how the environment map is stored. Once again, there are many options that may make more or less sense for certain implementations. All systems in this course can represent environment maps in *sphere map* form, as an image of a reflective sphere. Some can also use *cube map* form, mapping reflection vectors onto the faces of a cube, or *parabolic map* form, as images of two reflective paraboloids [6].

2.2 Homomorphic BRDF Factorization

non-standard texgen, realistic surfaces

```
texture("p",V) * texture("q",H) * texture("p",L)
* diffuse * color
```

This is the run-time aspect of McCool, Ang and Ahmad's 2001 SIGGRAPH paper [7]. The bulk of this paper dealt with numerical factorization of arbitrary bidirectional reflectance distribution functions (BRDFs) into combinations of 2D textures.

A BRDF is a 4D function that encodes the reflectance of a surface based on both the direction of view (`V` = 2 dimensions) and the incoming light direction (`L` = 2 dimensions). Equipment exists to measure the BRDF of a real surfaces, typically at a large number of discrete locations for both light and view directions. Given this BRDF, we

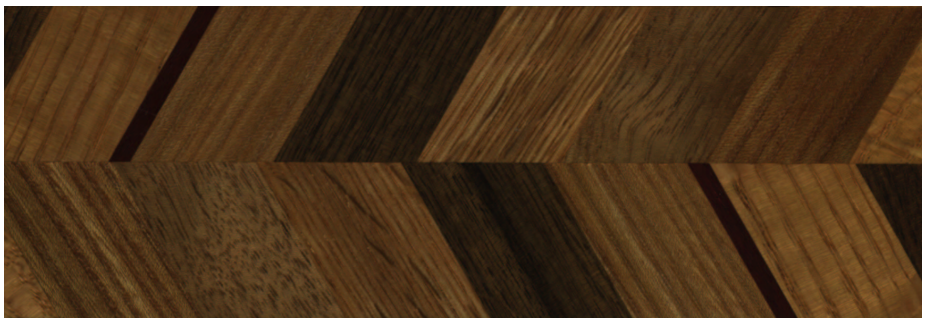


Figure 4: Scan of wood

can create realistic renderings of many surfaces. However, the nature of the BRDF as a 4D function prevent its direct use for real-time rendering.

The homomorphic factorization method computes a least-squares fit to a full 4D BRDF by a product of 2D textures, each with a unique set of texture coordinates dependent on both V and L. The method doesn't constrain the choice of texture coordinates for each 2D texture, but good results were obtained in the original paper using one texture lookup indexed by V, one indexed by L (actually the same texture used over again) and one indexed by $H = V + L$, all expressed in the local tangent coordinates. This set of textures makes some physical sense relative to "microfacet" BRDF models that model the surface as a distribution of microscopic perfectly reflective facets. The H texture can be interpreted as the probability any microfacet will have the given orientation, H. The V and L textures can be interpreted as shadowing and masking of some microfacets by other facets.

This is a good choice for an example since it requires non-standard texture coordinate generation (and hence application or vertex-level computation). It also gives more realistic appearance than is typically seen in real-time rendering, even on systems with full shading support.

2.3 Procedural Wood

Should be able to morph between 3D versions of the different wood samples in [Figure 4]. ...a good basis is the wood shader in The RenderMan Companion [9]. It should be parameterized for dark and light bands (color, width and transition) and also different fine grain in the dark and light bands (color, frequency and specularity). I'll try to get a better scan to show the last effect – there is a variation in the specular highlight intensity that correlates well with the fine grain of some of the wood. Since it should be parameterized for all of these things, a simple 3D wood texture won't cut it, but feel free to use 1D, 2D or 3D textures for other things if it makes it easier.

This example is intended to be more complex than the typical real-time shader

(though not as complex as some of the 1000-line shaders used in software rendering). A single wood shader, with a great degree of parameterization, should give all of us a reasonable challenge in comparison to the relative simplicity of the first example.

References

[1] BLINN, J. F., AND NEWELL, M. E. Texture and reflection in computer generated images. *Communications of the ACM 19* (1976), 542–546.

[2] CABRAL, B. K., PEERCY, M. S., AND AIREY, J. M. Method, system, and computer program product for bump mapping in tangent space. US Patent 5,949,424, 1999.

[3] COOK, R. L. Shade trees. In *Proc. ACM SIGGRAPH* (July 1984), pp. 223–231.

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

[5] HANRAHAN, P., AND LAWSON, J. A language for shading and lighting calculations. In *Computer Graphics (SIGGRAPH '90 Proceedings)* (Aug. 1990), pp. 289–298.

[6] HEIDRICH, W., AND SEIDEL, H.-P. View-independent environment maps. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware* (1998), pp. 39–45.

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

[8] PERLIN, K. An image synthesizer. vol. 19, pp. 287–296.

[9] UPSTILL, S. *The RenderMan companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley, 1990.

[10] WHITTED, T., AND WEIMER, D. M. A software testbed for the development of 3D raster graphics systems. *ACM Transactions on Graphics 1*, 1 (January 1982), 43–57.

Chapter 2

NVIDIA

Bill Mark

State of the Art in Hardware Shading – NVIDIA

SIGGRAPH 2002 Course Notes

William R. Mark

April 4, 2002

Programmable graphics hardware, APIs, and shading languages are evolving towards greater generality and performance at a rate that is extremely rapid even by the standards of the computer industry. Because course notes must be prepared almost four months in advance of the SIGGRAPH conference, we can't include truly state-of-the-art material in the course notes. Instead, we will prepare a web site at `http://www.nvidia.com/siggraph2002` that will provide course attendees with material to complement the course presentation.

We are including in these course notes a copy of NVIDIA's OpenGL extension specification for the interface to the programmable vertex hardware in the GeForce3 and GeForce4. The `NV_vertex_program` extension is similar to assembly-language-level interfaces that will provide access to the capabilities of future NVIDIA GPUs.

Name

NV_vertex_program

Name Strings

GL_NV_vertex_program

Contact

Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)

Notice

IP Status

NVIDIA Proprietary.

Status

Version 1.6

Version

NVIDIA Date: February 25, 2002
$Date$ $Revision$
$Id: //sw/main/docs/OpenGL/specs/GL_NV_vertex_program.txt#16 $

Number

233

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification and requires OpenGL 1.2.1.

Requires support for the ARB_multitexture extension with at least two texture units.

EXT_point_parameters affects the definition of this extension.

EXT_secondary_color affects the definition of this extension.

EXT_fog_coord affects the definition of this extension.

EXT_vertex_weighting affects the definition of this extension.

ARB_imaging affects the definition of this extension.

Overview

Unextended OpenGL mandates a certain set of configurable per-vertex computations defining vertex transformation, texture coordinate generation and transformation, and lighting. Several extensions have added further per-vertex computations to OpenGL. For example, extensions have defined new texture coordinate generation modes (ARB_texture_cube_map, NV_texgen_reflection, NV_texgen_emboss), new vertex transformation modes (EXT_vertex_weighting), new lighting modes (OpenGL 1.2's separate specular and rescale normal functionality), several modes for fog distance generation (NV_fog_distance), and

eye-distance point size attenuation (EXT_point_parameters).

Each such extension adds a small set of relatively inflexible per-vertex computations.

This inflexibility is in contrast to the typical flexibility provided by the underlying programmable floating point engines (whether micro-coded vertex engines, DSPs, or CPUs) that are traditionally used to implement OpenGL's per-vertex computations. The purpose of this extension is to expose to the OpenGL application writer a significant degree of per-vertex programmability for computing vertex parameters.

For the purposes of discussing this extension, a vertex program is a sequence of floating-point 4-component vector operations that determines how a set of program parameters (defined outside of OpenGL's begin/end pair) and an input set of per-vertex parameters are transformed to a set of per-vertex output parameters.

The per-vertex computations for standard OpenGL given a particular set of lighting and texture coordinate generation modes (along with any state for extensions defining per-vertex computations) is, in essence, a vertex program. However, the sequence of operations is defined implicitly by the current OpenGL state settings rather than defined explicitly as a sequence of instructions.

This extension provides an explicit mechanism for defining vertex program instruction sequences for application-defined vertex programs. In order to define such vertex programs, this extension defines a vertex programming model including a floating-point 4-component vector instruction set and a relatively large set of floating-point 4-component registers.

The extension's vertex programming model is designed for efficient hardware implementation and to support a wide variety of vertex programs. By design, the entire set of existing vertex programs defined by existing OpenGL per-vertex computation extensions can be implemented using the extension's vertex programming model.

Issues

What should this extension be called?

RESOLUTION: NV_vertex_program. DirectX 8 refers to its similar functionality as "vertex shaders". This is a confusing term because shaders are usually assumed to operate at the fragment or pixel level, not the vertex level.

Conceptually, what the extension defines is an application-defined program (admittedly limited by its sequential execution model) for processing vertices so the "vertex program" term is more accurate.

Additionally, some of the API machinery in this extension for describing programs could be useful for extending other OpenGL operations with programs (though other types of programs would likely look very different from vertex programs).

What terms are important to this specification?

vertex program mode - when vertex program mode is enabled, vertices are transformed by an application-defined vertex program.

conventional GL vertex transform mode - when vertex program mode is disabled (or the extension is not supported), vertices are

transformed by GL's conventional texgen, lighting, and transform state.

provoke - the verb that denotes the beginning of vertex transformation by either vertex program mode or conventional GL vertex transform mode. Vertices are provoked when either glVertex or glVertexAttribNV(0, ...) is called.

program target - a type or class of program. This extension supports two program targets: the vertex program and the vertex state program. Future extensions could add other program targets.

vertex program - an application-defined vertex program used to transform vertices when vertex program mode is enabled.

vertex state program - a program similar to a vertex program. Unlike a vertex program, a vertex state program runs outside of a glBegin/glEnd pair. Vertex state programs do not transform a vertex. They just update program parameters.

vertex attribute - one of 16 4-component per-vertex parameters defined by this extension. These attributes alias with the conventional per-vertex parameters.

per-vertex parameter - a vertex attribute or a conventional per-vertex parameter such as set by glNormal3f or glColor3f.

program parameter - one of 96 4-component registers available to vertex programs. The state of these registers is shared among all vertex programs.

What part of OpenGL do vertex programs specifically bypass?

Vertex programs bypass the following OpenGL functionality:

- Normal transformation and normalization
- Color material
- Per-vertex lighting
- Texture coordinate generation
- The texture matrix
- The normalization of AUTO_NORMAL evaluated normals
- The modelview and projection matrix transforms
- The per-vertex processing in EXT_point_parameters
- The per-vertex processing in NV_fog_distance
- Raster position transformation
- Client-defined clip planes

Operations not subsumed by vertex programs

- The view frustum clip
- Perspective divide (division by w)

- The viewport transformation
- The depth range transformation
- Clamping the primary and secondary color to [0,1]
- Primitive assembly and subsequent operations
- Evaluator (except the AUTO_NORMAL normalization)

How specific should this specification be about precision?

RESOLUTION: Reasonable precision requirements are incorporated into the specification beyond the often vague requirements of the core OpenGL specification.

This extension essentially defines an instruction set and its corresponding execution environment. The instruction set specified may find applications beyond the traditional purposes of 3D vertex transformation, lighting, and texture coordinate generation that have fairly lax precision requirements. To facilitate such possibly unexpected applications of this functionality, minimum precision requirements are specified.

The minimum precision requirements in the specification are meant to serve as a baseline so that application developers can write vertex programs with minimal worries about precision issues.

What about when the "execution environment" involves support for other extensions?

This extension assumes support for functionality that includes a fog distance, secondary color, point parameters, and multiple texture coordinates.

There is a trade-off between requiring support for these extensions to guarantee a particular extended execution environment and requiring lots of functionality that everyone might not support.

Application developers will desire a high baseline of functionality so that OpenGL applications using vertex programs can work in the full context of OpenGL. But if too much is required, the implementation burden mandated by the extension may limit the number of available implementations.

Clearly we do not want to require support for 8 texture units even if the machinery is there for it. Still multitexture is a common and important feature for using vertex programs effectively. Requiring at least two texture units seems reasonable.

What do we say about the alpha component of the secondary color?

RESOLUTION: When vertex program mode is enabled, the alpha component of csec used for the color sum state is assumed always zero. Another downstream extension may actually make the alpha component written into the COL1 (or BFC1) vertex result register available.

Should client-defined clip planes operate when vertex program mode is enabled?

RESOLUTION. No.

OpenGL's client-defined clip planes are specified in eye-space. Vertex programs generate homogeneous clip space positions. Unlike the conventional OpenGL vertex transformation mode, vertex program mode requires no semantic equivalent to eye-space.

Applications that require client-defined clip planes can simulate OpenGL-style client-defined clip planes by generating texture coordinates and using alpha testing or other per-fragment tests such as NV_texture_shader's CULL_FRAGMENT_NV program to discard fragments. In many ways, these schemes provide a more flexible mechanism for clipping than client-defined clip planes.

Unfortunately, vertex programs used in conjunction with selection or feedback will not have a means to support client-defined clip planes because the per-fragment culling mechanisms described in the previous paragraph are not available in the selection or feedback render modes. Oh well.

Finally, as a practical concern, client-defined clip planes greatly complicate clipping for various hardware rasterization architectures.

How are edge flags handled?

RESOLUTION: Passed through without the ability to be modified by a vertex program. Applications are free to send edge flags when vertex program mode is enabled.

Should vertex attributes alias with conventional per-vertex parameters?

RESOLUTION. YES.

This aliasing should make it easy to use vertex programs with existing OpenGL code that transfers per-vertex parameters using conventional OpenGL per-vertex calls.

It also minimizes the number of per-vertex parameters that the hardware must maintain.

See Table X.2 for the aliasing of vertex attributes and conventional per-vertex parameters.

How should vertex attribute arrays interact with conventional vertex arrays?

RESOLUTION: When vertex program mode is enabled, a particular vertex attribute array will be used if enabled, but if disabled, and the corresponding aliased conventional vertex array is enabled (assuming that there is a corresponding aliased conventional vertex array for the particular vertex array), the conventional vertex array will be used.

This matches the way immediate mode per-vertex parameter aliasing works.

This does slightly complicate vertex array validation in program mode, but programmers using vertex arrays can simply enable vertex program mode without reconfiguring their conventional vertex arrays and get what they expect.

Note that this does create an asymmetry between immediate mode and vertex arrays depending on whether vertex program mode is

enabled or not. The immediate mode vertex attribute commands operate unchanged whether vertex program mode is enabled or not. However the vertex attribute vertex arrays are used only when vertex program mode is enabled.

Supporting vertex attribute vertex arrays when vertex program mode is disabled would create a large implementation burden for existing OpenGL implementations that have heavily optimized conventional vertex arrays. For example, the normal array can be assumed to always contain 3 and only 3 components in conventional OpenGL vertex transform mode, but may contain 1, 2, 3, or 4 components in vertex program mode.

There is not any additional functionality gained by supporting vertex attribute arrays when vertex program mode is disabled, but there is lots of implementation overhead. In any case, it does not seem something worth encouraging so it is simply not supported. So vertex attribute arrays are IGNORED when vertex program mode is not enabled.

Ignoring VertexAttribute commands or treating VertexAttribute commands as an error when vertex program mode is enabled would likely add overhead for such a conditional check. The implementation overhead for supporting VertexAttribute commands when vertex program mode is disabled is not that significant. Additionally, it is likely that setting persistent vertex attribute state while vertex program mode is disabled may be useful to applications. So vertex attribute immediate mode commands are PERMITTED when vertex program mode is not enabled.

Colors and normals specified as ints, uints, shorts, ushorts, bytes, and ubytes are converted to floating-point ranges when supplied to core OpenGL as described in Table 2.6. Other per-vertex attributes such as texture coordinates and positions are not converted. How does this mix with vertex programs where all vertex attributes are supposedly treated identically?

RESOLUTION: Vertex attributes specified as bytes and ubytes are always converted as described in Table 2.6. All other formats are not converted according to Table 2.6 but simply converted directly to floating-point.

The ubyte type is converted because those types seem more useful for passing colors in the [0,1] range.

If an application desires a conversion, the conversion can be incorporated into the vertex program itself.

This also applies to vertex attribute arrays. However, by enabling a color or normal vertex array and not enabling the corresponding aliased vertex attribute array, programmers can get the conventional conversions for color and normal arrays (but only for the vertex attribute arrays that alias to the conventional color and normal arrays and only with the sizes/types supported by these color and normal arrays).

Should programs be C-style null-terminated strings?

RESOLUTION: No. Programs should be specified as an array of GLubyte with an explicit length parameter. OpenGL has no precedent for passing null-terminated strings into the API (though glGetString returns null-terminated strings). Null-terminated strings are problematic for some languages.

Should all existing OpenGL transform functionality and extensions be implementable as vertex programs?

RESOLUTION: Yes. Vertex programs should be a complete superset of what you can do with OpenGL 1.2 and existing vertex transform extensions.

To implement EXT_point_parameters, the GL_VERTEX_PROGRAM_POINT_SIZE_NV enable is introduced.

To implement two-sided lighting, the GL_VERTEX_PROGRAM_TWO_SIDE_NV enable is introduced.

How does glPointSize work with vertex programs?

RESOLUTION: If GL_VERTEX_PROGRAM_POINT_SIZE_NV is disabled, the size of points is determine by the glPointSize state. If enabled, the point size is determined per-vertex by the clamped value of the vertex result PSIZ register.

Can the currently bound vertex program object name be deleted or reloaded?

RESOLUTION. Yes. When a vertex program object name is deleted or reloaded when it is the currently bound vertex program object, it is as if a rebind occurs after the deletion or reload.

In the case of a reload, the new vertex program object will be used from then on. In the case of a deletion, the current vertex program object will be treated as if it is nonexistent.

Should program objects have a mechanism for managing program residency?

RESOLUTION: Yes. Vertex program instruction memory is a limited hardware resource. glBindProgramNV will be faster if binding to a resident program. Applications are likely to want to quickly switch between a small collection of programs.

glAreProgramsResidentNV allows the residency status of a group of programs to be queried. This mimics glAreTexturesResident.

Instead of adopting the glPrioritizeTextures mechanism, a new glRequestResidentProgramsNV command is specified instead. Assigning priorities to textures has always been a problematic endeavor and few OpenGL implementations implemented it effectively. For the priority mechanism to work well, it requires the client to routinely update the priorities of textures.

The glRequestResidentProgramsNV indicates to the GL that a set of programs are intended for use together. Because all the programs are requesting residency as a group, drivers should be able to attempt to load all the requested programs at once (and remove from residency programs not in the group if necessary). Clients can use glAreProgramsResidentNV to query the relative success of the request.

glRequestResidentProgramsNV should be superior to loading programs on-demand because fragmentation can be avoided.

What happens when you execute a nonexistent or invalid program?

RESOLUTION: glBegin will fail with a GL_INVALID_OPERATION if the currently bound vertex program is nonexistent or invalid. The same applies to glRasterPos and any command that implies a glBegin.

Because the glVertex and glVertexAttribNV(0, ...) are ignored outside of a glBegin/glEnd pair (without generating an error) it is impossible to provoke a vertex program if the current vertex program is nonexistent or invalid. Other per-vertex parameters (for examples those set by glColor, glNormal, and glVertexAttribNV when the attribute number is not zero) are recorded since they are legal outside of a glBegin/glEnd.

For vertex state programs, the problem is simpler because glExecuteProgramNV can immediately fail with a GL_INVALID_OPERATION when the named vertex state program is nonexistent or invalid.

What happens when a matrix has been tracked into a set of program parameters, but then glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, addr, GL_NONE, GL_IDENTITY_NV) is performed?

RESOLUTION: The specified program parameters stop tracking a matrix, but they retain the values of the matrix they were last tracking.

Can rows of tracked matrices be queried by querying the program parameters that track them?

RESOLUTION: Yes.

Discussing matrices is confusing because of row-major versus column-major issues. Can you give an example of how a matrix is tracked?

```
GLfloat matrix[16] = { 1, 5, 9,  13,
                       2, 6, 10, 14,
                       3, 7, 11, 15,
                       4, 8, 12, 16 };
GLfloat row1[4], row2[4];

glMatrixMode(GL_MATRIX0_NV);
glLoadMatrixf(matrix);
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 4, GL_MATRIX0_NV, GL_IDENTITY_NV);
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 8, GL_MATRIX0_NV, GL_TRANSPOSE_NV);
glGetProgramParameterfvNV(GL_VERTEX_PROGRAM_NV, 5,
  GL_PROGRAM_PARAMETER_NV, row1);
/* row1 is now [ 2 6 10 14 ] */
glGetProgramParameterfvNV(GL_VERTEX_PROGRAM_NV, 9,
  GL_PROGRAM_PARAMETER_NV, row2);
/* row2 is now [ 5 6 7 8 ] because the tracked matrix is transposed */
```

Should evaluators be extended to evaluate arbitrary vertex attributes?

RESOLUTION: Yes. We'll support 32 new maps (16 for MAP1 and 16 for MAP2) that take priority over the conventional maps that they might alias to (only when vertex program mode is enabled).

These new maps always evaluate all four components. The rationale for this is that if we supported 1, 2, 3, or 4 components, that would add 128 (16*4*2) enumerants which is too many. In addition, if you wanted to evaluate two 2-component vertex attributes, you could instead generate one 4-component vertex attribute and use

the vertex program with swizzling to treat this as two-components.

Moreover, we are assuming 4-component vector instructions so less than 4-component evaluations might not be any more efficient than 4-component evaluations. Implementations that use vector instructions such as Intel's SSE instructions will be easier to implement since they can focus on optimizing just the 4-component case.

How should GL_AUTO_NORMAL work with vertex programs?

RESOLUTION: GL_AUTO_NORMAL should NOT guarantee that the generated analytical normal be normalized. In vertex program mode, the current vertex program can easily normalize the normal if required.

This can lead to greater efficiency if the vertex program transforms the normal to another coordinate system such as eye-space with a transform that preserves vector length. Then a single normalize after transform is more efficient than normalizing after evaluation and also normalizing after transform.

Conceptually, the normalize mandated for AUTO_NORMAL in section 5.1 is just one of the many transformation operations subsumed by vertex programs.

Should the new vertex program related enables push/pop with GL_ENABLE_BIT?

RESOLUTION: Yes. Pushing and popping enable bits is easy. This includes the 32 new evaluator map enable bits. These evaluator enable bits are also pushed and popped using GL_EVAL_BIT.

Should all the vertex attribute state push/pop with GL_CURRENT_BIT?

RESOLUTION: Yes. The state is aliased with the conventional per-vertex parameter state so it really should push/pop.

Should all the vertex attrib vertex array state push/pop with GL_CLIENT_VERTEX_ARRAY_BIT?

RESOLUTION: Yes.

Should all the other vertex program-related state push/pop somehow?

RESOLUTION: No.

The other vertex program doesn't fit well with the existing bits. To be clear, GL_ALL_ATTRIB_BITS does not push/pop vertex program state other than enables.

Should we generate a GL_INVALID_OPERATION operation if updating a vertex attribute greater than 15?

RESOLUTION: Yes.

The other option would be to mask or modulo the vertex attribute index with 16. This is cheap, but it would make it difficult to increase the number of vertex attributes in the future.

If we check for the error, it should be a well predicted branch for immediate mode calls. For vertex arrays, the check is only required at vertex array specification time.

Hopefully this will encourage people to use vertex arrays over immediate mode.

Should writes to program parameter registers during a vertex program be supported?

RESOLUTION. No.

Writes to program parameter registers from within a vertex program would require the execution of vertex programs to be serialized with respect to each other. This would create an unwarranted implementation penalty for parallel vertex program execution implementations.

However vertex state programs may write to program parameter registers (that is the whole point of vertex state programs).

Should we support variously sized immediate mode byte and ubyte commands? How about for vertex arrays?

RESOLUTION. Only support the 4ub mode.

There are simply too many glVertexAttribNV routines. Passing less than 4 bytes at a time is inefficient. We expect the main use for bytes to be for colors where these will be unsigned bytes. So let's just support 4ub mode for bytes. This applies to vertex arrays too.

Should we support integer, unsigned integer, and unsigned short formats for vertex attributes?

RESOLUTION: No. It's just too many immediate mode entry points, most of which are not that useful. Signed shorts are supported however. We expect signed shorts to be useful for passing compact texture coordinates.

Should we support doubles for vertex attributes?

RESOLUTION: Yes. Some implementation of the extension might support double precision. Lots of math routines output double precision.

Should there be a way to determine where in a loaded program string the first parse error occurs?

RESOLUTION: Yes. You can query PROGRAM_ERROR_POSITION_NV.

Should program objects be shared among rendering contexts in the same manner as display lists and texture objects?

RESOLUTION: Yes.

How should this extension interact with color material?

RESOLUTION: It should not. Color material is a conventional OpenGL vertex transform mode. It does not have a place for vertex programs. If you want to emulate color material with vertex programs, you would simply write a program where the material parameters feed from the color vertex attribute.

Should there be a glMatrixMode or glActiveTextureARB style selector for vertex attributes?

RESOLUTION: No. While this would let us reduce a lot of enumerants down, it would make programming a hassle in lots of cases. Consider having to change the vertex attribute mode to enable a set of vertex arrays.

How should gets for vertex attribute array pointers?

RESOLUTION: Add new get commands. Using the existing calls would require adding 4 sets of 16 enumerants stride, type, size, and pointer. That's too many gets.

Instead add glGetVertexAttribNV and glGetVertexAttribPointervNV. glGetVertexAttribNV is also useful for querying the current vertex attribute.

glGet and glGetPointerv will not return vertex attribute array pointers.

Why is the address register numbered and why is it a vector register?

In the future, A0.y and A0.z and A0.w may exist. For this extension, only A0.x is useful. Also in the future, there may be more than one address register.

There's a nice consistency in thinking about all the registers as 4-component vectors even if the address register has only one usable component.

Should vertex programs and vertex state programs be required to have a header token and an end token?

RESOLUTION: Yes.

The "!!VP1.0" and "!!VSP1.0" tokens start vertex programs and vertex state programs respectively. Both types of programs must end with the "END" token.

The initial header token reminds the programmer what type of program they are writing. If vertex programs and vertex state programs are ever read from disk files, the header token can serve as a magic number for identifying vertex programs and vertex state programs.

The target type for vertex programs and vertex state programs can be distinguished based on their respective grammars independent of the initial header tokens, but the initial header tokens will make it easier for programmers to distinguish the two program target types.

We expect programs to often be generated by concatenation of program fragments. The "END" token will hopefully reduce bugs due to specifying an incorrectly concatenated program.

It's tempting to make these additional header and end tokens optional, but if there is a sanity check value in header and end tokens, that value is undermined if the tokens are optional.

What should be said about rendering invariances?

RESOLUTION: See the Appendix A additions below.

The justification for the two rules cited is to support multi-pass rendering when using vertex programs. Different rendering passes will likely use different programs so there must be some means of

guaranteeing that two different programs can generate particular identical vertex results between different passes.

In practice, this does limit the type of vertex program implementations that are possible.

For example, consider a limited hardware implementation of vertex programs that uses a different floating-point implementation than the CPU's floating-point implementation. If the limited hardware implementation can only run small vertex programs (say the hardware provides on 4 temporary registers instead of the required 12), the implementation is incorrect and non-conformant if programs that only require 4 temporary registers use the vertex program hardware, but programs that require more than 4 temporary registers are implemented by the CPU.

This is a very important practical requirement. Consider a multi-pass rendering algorithm where one pass uses a vertex program that uses only 4 temporary registers, but a different pass uses a vertex program that uses 5 temporary registers. If two programs have instruction sequences that given the same input state compute identical resulting vertex positions, the multi-pass algorithm should generate identically positioned primitives for each pass. But given the non-conformant vertex program implementation described above, this could not be guaranteed.

This does not mean that schemes for splitting vertex program implementations between dedicated hardware and CPUs are impossible. If the CPU and dedicated vertex program hardware used IDENTICAL floating-point implementations and therefore generated exactly identical results, the above described could work.

While these invariance rules are vital for vertex programs operating correctly for multi-pass algorithms, there is no requirement that conventional OpenGL vertex transform mode will be invariant with vertex program mode. A multi-pass algorithm should not assume that one pass using vertex program mode and another pass using conventional GL vertex transform mode will generate identically positioned primitives.

Consider that while the conventional OpenGL vertex program mode is repeatable with itself, the exact procedure used to transform vertices is not specified nor is the procedure's precision specified. The GL specification indicates that vertex coordinates are transformed by the modelview matrix and then transformed by the projection matrix. Some implementations may perform this sequence of transformations exactly, but other implementations may transform vertex coordinates by the composite of the modelview and projection matrices (one matrix transform instead of two matrix transforms in sequence). Given this implementation flexibility, there is no way for a vertex program author to exactly duplicate the precise computations used by the conventional OpenGL vertex transform mode.

The guidance to OpenGL application programs is clear. If you are going to implement multi-pass rendering algorithms that require certain invariances between the multiple passes, choose either vertex program mode or the conventional OpenGL vertex transform mode for your rendering passes, but do not mix the two modes.

What range of relative addressing offsets should be allowed?

RESOLUTION: -64 to 63.

Negative offsets are useful for accessing a table centered at zero without extra bias instructions. Having the offsets support much larger magnitudes just seems to increase the required instruction widths. The -64 to 63 range seems like a reasonable compromise.

When EXT_secondary_color is supported, how does the GL_COLOR_SUM_EXT enable affect vertex program mode?

RESOLUTION: The GL_COLOR_SUM_EXT enable has no affect when vertex program mode is enabled.

When vertex program mode is enabled, the color sum operation is always in operation. A program can "avoid" the color sum operation by not writing the COL1 (or BFC1 when GL_VERTEX_PROGRAM_TWO_SIDE_NV) vertex result registers because the default values of all vertex result registers is (0,0,0,1). For the color sum operation, the alpha value is always assumed zero. So by not writing the secondary color vertex result registers, the program assures that zero is added as part of the color sum operation.

If there is a cost to the color sum operation, OpenGL implementations may be smart enough to determine at program bind time whether a secondary color vertex result is generated and implicitly disable the color sum operation.

Why must RCP of 1.0 always be 1.0?

This is important for 3D graphics so that non-projective textures and orthogonal projections work as expected. Basically when q or w is 1.0, things should work as expected.

Stronger requirements such as "RCP of -1.0 must always be -1.0" are encouraged, but there is no compelling reason to state such requirements explicitly as is the case for "RCP of 1.0 must always be 1.0".

What happens when the source scalar value for the ARL instruction is an extremely positive or extremely negative floating-point value? Is there a problem mapping the value to a constrained integer range?

RESOLUTION: It is not a problem. Relative addressing can by offset by a limited range of offsets (-64 to 63). Relative addressing that falls outside of the 0 to 95 range of program parameter registers is automatically mapped to (0,0,0,0).

Clamping the source scalar value for ARL to the range -64 to 160 inclusive is sufficient to ensure that relative addressing is out of range.

How do you perform a 3-component normalize in three instructions?

```
#
# R1 = (nx,ny,nz)
#
# R0.xyz = normalize(R1)
# R0.w   = 1/sqrt(nx*nx + ny*ny + nz*nz)
#
DP3 R0.w, R1, R1;
RSQ R0.w, R0.w;
MUL R0.xyz, R1, R0.w;
```

How do you perform a 3-component cross product in two instructions?

```
  #
  # Cross product |  i     j     k   | into R2.
  #               | R0.x  R0.y  R0.z |
  #               | R1.x  R1.y  R1.z |
  #
  MUL R2, R0.zxyw, R1.yzxw;
  MAD R2, R0.yzxw, R1.zxyw, -R2;
```

How do you perform a 4-component vector absolute value in one instruction?

```
  #
  # Absolute value is the maximum of the negative and positive
  # components of a vector.
  #
  # R1 = abs(R0)
  #
  MAX R1, R0, -R0;
```

How do you compute the determinant of a 3x3 matrix in three instructions?

```
  #
  # Determinant of | R0.x  R0.y  R0.z | into R3
  #                | R1.x  R1.y  R1.z |
  #                | R2.x  R2.y  R2.z |
  #
  MUL R3, R1.zxyw, R2.yzxw;
  MAD R3, R1.yzxw, R2.zxyw, -R3;
  DP3 R3, R0, R3;
```

How do you transform a vertex position by a 4x4 matrix and then perform a homogeneous divide?

```
  #
  # c[20] = modelview row 0
  # c[21] = modelview row 1
  # c[22] = modelview row 2
  # c[23] = modelview row 3
  #
  # result = R5
  #
  DP4 R5.w, v[OPOS], c[23];
  DP4 R5.x, v[OPOS], c[20];
  DP4 R5.y, v[OPOS], c[21];
  DP4 R5.z, v[OPOS], c[22];
  RCP R11, R5.w;
  MUL R5,R5,R11;
```

How do you perform a vector weighting of two vectors using a single weight?

```
  #
  # R2        = vector 0
  # R3        = vector 1
  # v[WGHT].x = scalar weight to blend vectors 0 and 1
  # result    = R2 * v[WGHT].x + R3 * (1-v[WGHT])
  #
  # this is because A*B + (1-A)*C = A*(B-C) + C
  #
  ADD R4, R2, -R3;
  MAD R4, v[WGHT].x, R4, R3;
```

How do you reduce a value to some fundamental period such as 2*PI?

```
#
# c[36] = (1.0/(2*PI), 2*PI, 0.0, 0.0)
#
# R1.x = input value
# R2   = result
#
MUL R0, R1, c[36].x;
EXP R4, R0.x;
MUL R2, R4.y, c[36].y;
```

How do you implement a simple specular and diffuse lighting computation with an eye-space normal?

```
!!VP1.0
#
# c[0-3]  = modelview projection (composite) matrix
# c[4-7]  = modelview inverse transpose
# c[32]   = normalized eye-space light direction (infinite light)
# c[33]   = normalized constant eye-space half-angle vector (infinite viewer)
# c[35].x = pre-multiplied monochromatic diffuse light color & diffuse material
# c[35].y = pre-multiplied monochromatic ambient light color & diffuse material
# c[36]   = specular color
# c[38].x = specular power
#
# outputs homogenous position and color
#
DP4   o[HPOS].x, c[0], v[OPOS];
DP4   o[HPOS].y, c[1], v[OPOS];
DP4   o[HPOS].z, c[2], v[OPOS];
DP4   o[HPOS].w, c[3], v[OPOS];
DP3   R0.x, c[4], v[NRML];
DP3   R0.y, c[5], v[NRML];
DP3   R0.z, c[6], v[NRML];           # R0 = n' = transformed normal
DP3   R1.x, c[32], R0;               # R1.x = Lpos DOT n'
DP3   R1.y, c[33], R0;               # R1.y = hHat DOT n'
MOV   R1.w, c[38].x;                 # R1.w = specular power
LIT   R2, R1;                        # Compute lighting values
MAD   R3, c[35].x, R2.y, c[35].y;    # diffuse + emissive
MAD   o[COL0].xyz, c[36], R2.z, R3;  # + specular
END
```

Can you perturb transformed vertex positions with a vertex program?

Yes. Here is an example that performs an object-space diffuse lighting computations and perturbs the vertex position based on this lighting result. Do not take this example too seriously.

```
!!VP1.0
#
# c[0-3]  = modelview projection (composite) matrix
# c[32]   = normalized light direction in object-space
# c[35]   = yellow diffuse material, (1.0, 1.0, 0.0, 1.0)
# c[64].x = 0.0
# c[64].z = 0.125, a scaling factor
#
# outputs diffuse illumination for color and perturbed position
#
DP3   R0, c[32], v[NRML];     # light direction DOT normal
MUL   o[COL0].xyz, R0, c[35];
MAX   R0, c[64].x, R0;
MUL   R0, R0, v[NRML];
```

```
    MUL   R0, R0, c[64].z;
    ADD   R1, v[OPOS], -R0;       # perturb object space position
    DP4   o[HPOS].x, c[0], R1;
    DP4   o[HPOS].y, c[1], R1;
    DP4   o[HPOS].z, c[2], R1;
    DP4   o[HPOS].w, c[3], R1;
    END
```

What if more exponential precision is needed than provided by the builtin EXP instruction?

A sequence of vertex program instructions can be used refine the initial EXP approximation. The pseudo-macro below shows an example of how to refine the EXP approximation.

The psuedo-macro requires 10 instructions, 1 temp register, and 2 constant locations.

```
CE0 = { 9.61597636e-03, -1.32823968e-03, 1.47491097e-04, -1.08635004e-05 };
CE1 = { 1.00000000e+00, -6.93147182e-01, 2.40226462e-01, -5.55036440e-02 };

/* Rt != Ro && Rt != Ri */
EXP_MACRO(Ro:vector, Ri:scalar, Rt:vector) {
   EXP Rt, Ri.x;                  /* Use appropriate component of Ri */
   MAD Rt.w, c[CE0].w, Rt.y, c[CE0].z;
   MAD Rt.w, Rt.w,Rt.y, c[CE0].y;
   MAD Rt.w, Rt.w,Rt.y, c[CE0].x;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].w;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].z;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].y;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].x;
   RCP Rt.w, Rt.w;
   MUL Ro, Rt.w, Rt.x;            /* Apply user write mask to Ro */
}
```

Simulation gives |max abs error| < 3.77e-07 over the range (0.0 <= x < 1.0). Actual vertex program precision may be slightly less accurate than this.

What if more exponential precision is needed than provided by the builtin LOG instruction?

The pseudo-macro requires 10 instructions, 1 temp register, and 3 constant locations.

```
CL0 = { 2.41873696e-01, -1.37531206e-01, 5.20646796e-02, -9.31049418e-03 };
CL1 = { 1.44268966e+00, -7.21165776e-01, 4.78684813e-01, -3.47305417e-01 };
CL2 = { 1.0, NA, NA, NA };

/* Rt != Ro && Rt != Ri */
LOG_MACRO(Ro:vector, Ri:scalar, Rt:vector) {
   LOG Rt, Ri.x;                  /* Use appropriate component of Ri */
   ADD Rt.y, Rt.y, -c[CL2].x;
   MAD Rt.w, c[CL0].w, Rt.y, c[CL0].z;
   MAD Rt.w, Rt.w, Rt.y,c[CL0].y;
   MAD Rt.w, Rt.w, Rt.y,c[CL0].x;
   MAD Rt.w, Rt.w, Rt.y,c[CL1].w;
   MAD Rt.w, Rt.w, Rt.y,c[CL1].z;
   MAD Rt.w, Rt.w, Rt.y,c[CL1].y;
   MAD Rt.w, Rt.w, Rt.y,c[CL1].x;
   MAD Ro, Rt.w, Rt.y, Rt.x;      /* Apply user write mask to Ro */
}
```

```
        Simulation gives |max abs error| < 1.79e-07 over the range (1.0
        <= x < 2.0).  Actual vertex program precision may be slightly
        less accurate than this.

New Procedures and Functions

    void BindProgramNV(enum target, uint id);

    void DeleteProgramsNV(sizei n, const uint *ids);

    void ExecuteProgramNV(enum target, uint id, const float *params);

    void GenProgramsNV(sizei n, uint *ids);

    boolean AreProgramsResidentNV(sizei n, const uint *ids,
                                  boolean *residences);

    void RequestResidentProgramsNV(sizei n, uint *ids);

    void GetProgramParameterfvNV(enum target, uint index,
                                 enum pname, float *params);
    void GetProgramParameterdvNV(enum target, uint index,
                                 enum pname, double *params);

    void GetProgramivNV(uint id, enum pname, int *params);

    void GetProgramStringNV(uint id, enum pname, ubyte *program);

    void GetTrackMatrixivNV(enum target, uint address,
                            enum pname, int *params);

    void GetVertexAttribdvNV(uint index, enum pname, double *params);
    void GetVertexAttribfvNV(uint index, enum pname, float *params);
    void GetVertexAttribivNV(uint index, enum pname, int *params);

    void GetVertexAttribPointervNV(uint index, enum pname, void **pointer);

    boolean IsProgramNV(uint id);

    void LoadProgramNV(enum target, uint id, sizei len,
                       const ubyte *program);

    void ProgramParameter4fNV(enum target, uint index,
                              float x, float y, float z, float w)
    void ProgramParameter4dNV(enum target, uint index,
                              double x, double y, double z, double w)

    void ProgramParameter4dvNV(enum target, uint index,
                               const double *params);
    void ProgramParameter4fvNV(enum target, uint index,
                               const float *params);

    void ProgramParameters4dvNV(enum target, uint index,
                                uint num, const double *params);
    void ProgramParameters4fvNV(enum target, uint index,
                                uint num, const float *params);

    void TrackMatrixNV(enum target, uint address,
                       enum matrix, enum transform);

    void VertexAttribPointerNV(uint index, int size, enum type, sizei stride,
                               const void *pointer);
```

```
    void VertexAttrib1sNV(uint index, short x);
    void VertexAttrib1fNV(uint index, float x);
    void VertexAttrib1dNV(uint index, double x);
    void VertexAttrib2sNV(uint index, short x, short y);
    void VertexAttrib2fNV(uint index, float x, float y);
    void VertexAttrib2dNV(uint index, double x, double y);
    void VertexAttrib3sNV(uint index, short x, short y, short z);
    void VertexAttrib3fNV(uint index, float x, float y, float z);
    void VertexAttrib3dNV(uint index, double x, double y, double z);
    void VertexAttrib4sNV(uint index, short x, short y, short z, short w);
    void VertexAttrib4fNV(uint index, float x, float y, float z, float w);
    void VertexAttrib4dNV(uint index, double x, double y, double z, double w);
    void VertexAttrib4ubNV(uint index, ubyte x, ubyte y, ubyte z, ubyte w);

    void VertexAttrib1svNV(uint index, const short *v);
    void VertexAttrib1fvNV(uint index, const float *v);
    void VertexAttrib1dvNV(uint index, const double *v);
    void VertexAttrib2svNV(uint index, const short *v);
    void VertexAttrib2fvNV(uint index, const float *v);
    void VertexAttrib2dvNV(uint index, const double *v);
    void VertexAttrib3svNV(uint index, const short *v);
    void VertexAttrib3fvNV(uint index, const float *v);
    void VertexAttrib3dvNV(uint index, const double *v);
    void VertexAttrib4svNV(uint index, const short *v);
    void VertexAttrib4fvNV(uint index, const float *v);
    void VertexAttrib4dvNV(uint index, const double *v);
    void VertexAttrib4ubvNV(uint index, const ubyte *v);

    void VertexAttribs1svNV(uint index, sizei n, const short *v);
    void VertexAttribs1fvNV(uint index, sizei n, const float *v);
    void VertexAttribs1dvNV(uint index, sizei n, const double *v);
    void VertexAttribs2svNV(uint index, sizei n, const short *v);
    void VertexAttribs2fvNV(uint index, sizei n, const float *v);
    void VertexAttribs2dvNV(uint index, sizei n, const double *v);
    void VertexAttribs3svNV(uint index, sizei n, const short *v);
    void VertexAttribs3fvNV(uint index, sizei n, const float *v);
    void VertexAttribs3dvNV(uint index, sizei n, const double *v);
    void VertexAttribs4svNV(uint index, sizei n, const short *v);
    void VertexAttribs4fvNV(uint index, sizei n, const float *v);
    void VertexAttribs4dvNV(uint index, sizei n, const double *v);
    void VertexAttribs4ubvNV(uint index, sizei n, const ubyte *v);

New Tokens

    Accepted by the <cap> parameter of Disable, Enable, and IsEnabled,
    and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv,
    and GetDoublev, and by the <target> parameter of BindProgramNV,
    ExecuteProgramNV, GetProgramParameter[df]vNV, GetTrackMatrixivNV,
    LoadProgramNV, ProgramParameter[s]4[df][v]NV, and TrackMatrixNV:

        VERTEX_PROGRAM_NV                               0x8620

    Accepted by the <cap> parameter of Disable, Enable, and IsEnabled,
    and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv,
    and GetDoublev:

        VERTEX_PROGRAM_POINT_SIZE_NV                    0x8642
        VERTEX_PROGRAM_TWO_SIDE_NV                      0x8643

    Accepted by the <target> parameter of ExecuteProgramNV and
    LoadProgramNV:

        VERTEX_STATE_PROGRAM_NV                         0x8621
```

Accepted by the <pname> parameter of GetVertexAttrib[dfi]vNV:

ATTRIB_ARRAY_SIZE_NV	0x8623
ATTRIB_ARRAY_STRIDE_NV	0x8624
ATTRIB_ARRAY_TYPE_NV	0x8625
CURRENT_ATTRIB_NV	0x8626

Accepted by the <pname> parameter of GetProgramParameterfvNV
and GetProgramParameterdvNV:

PROGRAM_PARAMETER_NV	0x8644

Accepted by the <pname> parameter of GetVertexAttribPointervNV:

ATTRIB_ARRAY_POINTER_NV	0x8645

Accepted by the <pname> parameter of GetProgramivNV:

PROGRAM_TARGET_NV	0x8646
PROGRAM_LENGTH_NV	0x8627
PROGRAM_RESIDENT_NV	0x8647

Accepted by the <pname> parameter of GetProgramStringNV:

PROGRAM_STRING_NV	0x8628

Accepted by the <pname> parameter of GetTrackMatrixivNV:

TRACK_MATRIX_NV	0x8648
TRACK_MATRIX_TRANSFORM_NV	0x8649

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
GetFloatv, and GetDoublev:

MAX_TRACK_MATRIX_STACK_DEPTH_NV	0x862E
MAX_TRACK_MATRICES_NV	0x862F
CURRENT_MATRIX_STACK_DEPTH_NV	0x8640
CURRENT_MATRIX_NV	0x8641
VERTEX_PROGRAM_BINDING_NV	0x864A
PROGRAM_ERROR_POSITION_NV	0x864B

Accepted by the <matrix> parameter of TrackMatrixNV:

NONE	
MODELVIEW	
PROJECTION	
TEXTURE	
COLOR (if ARB_imaging is supported)	
MODELVIEW_PROJECTION_NV	0x8629
TEXTUREi_ARB	

where i is between 0 and n-1 where n is the number of texture units
supported.

Accepted by the <matrix> parameter of TrackMatrixNV and by the
<mode> parameter of MatrixMode:

MATRIX0_NV	0x8630
MATRIX1_NV	0x8631
MATRIX2_NV	0x8632
MATRIX3_NV	0x8633
MATRIX4_NV	0x8634

```
    MATRIX5_NV                                  0x8635
    MATRIX6_NV                                  0x8636
    MATRIX7_NV                                  0x8637

    (Enumerants 0x8638 through 0x863F are reserved for further matrix
    enumerants 8 through 15.)

Accepted by the <transform> parameter of TrackMatrixNV:

    IDENTITY_NV                                 0x862A
    INVERSE_NV                                  0x862B
    TRANSPOSE_NV                                0x862C
    INVERSE_TRANSPOSE_NV                        0x862D

Accepted by the <array> parameter of EnableClientState and
DisableClientState, by the <cap> parameter of IsEnabled, and by
the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and
GetDoublev:

    VERTEX_ATTRIB_ARRAY0_NV                     0x8650
    VERTEX_ATTRIB_ARRAY1_NV                     0x8651
    VERTEX_ATTRIB_ARRAY2_NV                     0x8652
    VERTEX_ATTRIB_ARRAY3_NV                     0x8653
    VERTEX_ATTRIB_ARRAY4_NV                     0x8654
    VERTEX_ATTRIB_ARRAY5_NV                     0x8655
    VERTEX_ATTRIB_ARRAY6_NV                     0x8656
    VERTEX_ATTRIB_ARRAY7_NV                     0x8657
    VERTEX_ATTRIB_ARRAY8_NV                     0x8658
    VERTEX_ATTRIB_ARRAY9_NV                     0x8659
    VERTEX_ATTRIB_ARRAY10_NV                    0x865A
    VERTEX_ATTRIB_ARRAY11_NV                    0x865B
    VERTEX_ATTRIB_ARRAY12_NV                    0x865C
    VERTEX_ATTRIB_ARRAY13_NV                    0x865D
    VERTEX_ATTRIB_ARRAY14_NV                    0x865E
    VERTEX_ATTRIB_ARRAY15_NV                    0x865F

Accepted by the <target> parameter of GetMapdv, GetMapfv, GetMapiv,
Map1d and Map1f and by the <cap> parameter of Enable, Disable, and
IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv,
GetFloatv, and GetDoublev:

    MAP1_VERTEX_ATTRIB0_4_NV                    0x8660
    MAP1_VERTEX_ATTRIB1_4_NV                    0x8661
    MAP1_VERTEX_ATTRIB2_4_NV                    0x8662
    MAP1_VERTEX_ATTRIB3_4_NV                    0x8663
    MAP1_VERTEX_ATTRIB4_4_NV                    0x8664
    MAP1_VERTEX_ATTRIB5_4_NV                    0x8665
    MAP1_VERTEX_ATTRIB6_4_NV                    0x8666
    MAP1_VERTEX_ATTRIB7_4_NV                    0x8667
    MAP1_VERTEX_ATTRIB8_4_NV                    0x8668
    MAP1_VERTEX_ATTRIB9_4_NV                    0x8669
    MAP1_VERTEX_ATTRIB10_4_NV                   0x866A
    MAP1_VERTEX_ATTRIB11_4_NV                   0x866B
    MAP1_VERTEX_ATTRIB12_4_NV                   0x866C
    MAP1_VERTEX_ATTRIB13_4_NV                   0x866D
    MAP1_VERTEX_ATTRIB14_4_NV                   0x866E
    MAP1_VERTEX_ATTRIB15_4_NV                   0x866F

Accepted by the <target> parameter of GetMapdv, GetMapfv, GetMapiv,
Map2d and Map2f and by the <cap> parameter of Enable, Disable, and
IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv,
GetFloatv, and GetDoublev:
```

MAP2_VERTEX_ATTRIB0_4_NV	0x8670
MAP2_VERTEX_ATTRIB1_4_NV	0x8671
MAP2_VERTEX_ATTRIB2_4_NV	0x8672
MAP2_VERTEX_ATTRIB3_4_NV	0x8673
MAP2_VERTEX_ATTRIB4_4_NV	0x8674
MAP2_VERTEX_ATTRIB5_4_NV	0x8675
MAP2_VERTEX_ATTRIB6_4_NV	0x8676
MAP2_VERTEX_ATTRIB7_4_NV	0x8677
MAP2_VERTEX_ATTRIB8_4_NV	0x8678
MAP2_VERTEX_ATTRIB9_4_NV	0x8679
MAP2_VERTEX_ATTRIB10_4_NV	0x867A
MAP2_VERTEX_ATTRIB11_4_NV	0x867B
MAP2_VERTEX_ATTRIB12_4_NV	0x867C
MAP2_VERTEX_ATTRIB13_4_NV	0x867D
MAP2_VERTEX_ATTRIB14_4_NV	0x867E
MAP2_VERTEX_ATTRIB15_4_NV	0x867F

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

-- Section 2.10 "Coordinate Transformations"

Add this initial discussion:

"Per-vertex parameters are transformed before the transformation results are used to generate primitives for rasterization, establish a raster position, or generate vertices for selection or feedback.

Each vertex's per-vertex parameters are transformed by one of two vertex transformation modes. The first vertex transformation mode is GL's conventional vertex transformation model. The second mode, known as 'vertex program' mode, transforms the vertex's per-vertex parameters by an application-supplied vertex program.

Vertex program mode is enabled and disabled, respectively, by

```
void Enable(enum target);
```

and

```
void Disable(enum target);
```

with target equal to VERTEX_PROGRAM_NV. When vertex program mode is enabled, vertices are transformed by the currently bound vertex program as discussed in section 2.14."

Update the original initial paragraph in the section to read:

"When vertex program mode is disabled, vertices, normals, and texture coordinates are transformed before their coordinates are used to produce an image in the framebuffer. We begin with a description of how vertex coordinates are transformed and how the transformation is controlled in the case when vertex program mode is disabled. The discussion that continues through section 2.13 applies when vertex program mode is disabled."

-- Section 2.10.2 "Matrices"

Change the first paragraph to read:

"The projection matrix and model-view matrix are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

```
  void MatrixMode(enum mode);
```

which takes one of the pre-defined constants TEXTURE, MODELVIEW, COLOR, PROJECTION, or MATRIXi_NV as the argument. In the case of MATRIXi_NV, i is an integer between 0 and n-1 indicating one of n tracking matrices where n is the value of the implementation defined constant MAX_TRACK_MATRICES_NV. TEXTURE is described later in section 2.10.2, and COLOR is described in section 3.6.3. The tracking matrices of the form MATRIXi_NV are described in section 2.14.5. If the current matrix mode is MODELVIEW, then matrix operations apply to the model-view matrix; if PROJECTION, then they apply to the projection matrix."

Change the last paragraph to read:

"The state required to implement transformations consists of a n-value integer indicating the current matrix mode (where n is 4 + the number of tracking matrices supported), a stack of at least two 4x4 matrices for each of COLOR, PROJECTION, and TEXTURE with associated stack pointers, n stacks (where n is at least 8) of at least one 4x4 matrix for each MATRIXi_NV with associated stack pointers, and a stack of at least 32 4x4 matrices with an associated stack pointer for MODELVIEW. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial matrix mode is MODELVIEW."

-- NEW Section 2.14 "Vertex Programs"

"The conventional GL vertex transformation model described in sections 2.10 through 2.13 is a configurable but essentially hard-wired sequence of per-vertex computations based on a canonical set of per-vertex parameters and vertex transformation related state such as transformation matrices, lighting parameters, and texture coordinate generation parameters.

The general success and utility of the conventional GL vertex transformation model reflects its basic correspondence to the typical vertex transformation requirements of 3D applications.

However when the conventional GL vertex transformation model is not sufficient, the vertex program mode provides a substantially more flexible model for vertex transformation. The vertex program mode permits applications to define their own vertex programs.

2.14.1 The Vertex Program Execution Model

A vertex program is a sequence of floating-point 4-component vector operations that operate on per-vertex attributes and program parameters. Vertex programs execute on a per-vertex basis and operate on each vertex completely independently from the processing of other vertices. Vertex programs execute a finite fixed sequence of instructions with no branching or looping. Vertex programs execute without data hazards so results computed in one operation can be used immediately afterwards. The result of a vertex program is a set of vertex result vectors that becomes the transformed vertex parameters used by primitive assembly.

Vertex programs use a specific well-defined instruction set, register set, and operational model defined in the following sections.

The vertex program register set consists of five types of registers described in the following five sections.

2.14.1.1 The Vertex Attribute Registers

The Vertex Attribute Registers are sixteen 4-component vector floating-point registers containing the current vertex's per-vertex attributes. These registers are numbered 0 through 15. These registers are private to each vertex program invocation and are initialized at each vertex program invocation by the current vertex attribute state specified with VertexAttribNV commands. These registers are read-only during vertex program execution. The VertexAttribNV commands used to update the vertex attribute registers can be issued both outside and inside of Begin/End pairs. Vertex program execution is provoked by updating vertex attribute zero. Updating vertex attribute zero outside of a Begin/End pair is ignored without generating any error (identical to the Vertex command operation).

The commands

```
void VertexAttrib{1234}{sfd}NV(uint index, T coords);
void VertexAttrib{1234}{sfd}vNV(uint index, T coords);
void VertexAttrib4ubNV(uint index, T coords);
void VertexAttrib4ubvNV(uint index, T coords);
```

specify the particular current vertex attribute indicated by index. The coordinates for each vertex attribute are named x, y, z, and w. The VertexAttrib1NV family of commands sets the x coordinate to the provided single argument while setting y and z to 0 and w to 1. Similarly, VertexAttrib2NV sets x and y to the specified values, z to 0 and w to 1; VertexAttrib3NV sets x, y, and z, with w set to 1, and VertexAttrib4NV sets all four coordinates. The error INVALID_VALUE is generated if index is greater than 15.

No conversions are applied to the vertex attributes specified as type short, float, or double. However, vertex attributes specified as type ubyte are converted as described by Table 2.6.

The commands

```
void VertexAttribs{1234}{sfd}vNV(uint index, sizei n, T coords[]);
void VertexAttribs4ubvNV(uint index, sizei n, GLubyte coords[]);
```

specify a contiguous set of n vertex attributes. The effect of

```
VertexAttribs{1234}{sfd}vNV(index, n, coords)
```

is the same as the command sequence

```
#define NUM k  /* where k is 1, 2, 3, or 4 components */
int i;
for (i=n-1; i>=0; i--) {
  VertexAttrib{NUM}{sfd}vNV(i+index, &coords[i*NUM]);
}
```

VertexAttribs4ubvNV behaves similarly.

The VertexAttribNV calls equivalent to VertexAttribsNV are issued in reverse order so that vertex program execution is provoked when index is zero only after all the other vertex attributes have first been specified.

2.14.1.2 The Program Parameter Registers

The Program Parameter Registers are ninety-six 4-component floating-point vector registers containing the vertex program parameters. These registers are numbered 0 through 95. This

relatively large set of registers is intended to hold parameters such as matrices, lighting parameters, and constants required by vertex programs. Vertex program parameter registers can be updated in one of two ways: by the ProgramParameterNV commands outside of a Begin/End pair or by a vertex state program executed outside of a Begin/End pair (vertex state programs are discussed in section 2.14.3).

The commands

```
  void ProgramParameter4fNV(enum target, uint index,
                            float x, float y, float z, float w)
  void ProgramParameter4dNV(enum target, uint index,
                           double x, double y, double z, double w)
```

specify the particular program parameter indicated by index. The coordinates values x, y, z, and w are assigned to the respective components of the particular program parameter. target must be VERTEX_PROGRAM_NV.

The commands

```
  void ProgramParameter4dvNV(enum target, uint index, double *params);
  void ProgramParameter4fvNV(enum target, uint index, float *params);
```

operate identically to ProgramParameter4fNV and ProgramParameter4dNV respectively except that the program parameters are passed as an array of four components.

The commands

```
  void ProgramParameters4dvNV(enum target, uint index,
                              uint num, double *params);
  void ProgramParameters4fvNV(enum target, uint index,
                              uint num, float *params);
```

specify a contiguous set of num program parameters. The effect is the same as

```
  for (i=index; i<index+num; i++) {
    ProgramParameter4{fd}vNV(i, params + i*4);
  }
```

The program parameter registers are shared to all vertex program invocations within a rendering context. ProgramParameterNV command updates and vertex state program executions are serialized with respect to vertex program invocations and other vertex state program executions.

Writes to the program parameter registers during vertex state program execution can be maskable on a per-component basis.

The error INVALID_VALUE is generated if any ProgramParameterNV has an index is greater than 95.

The initial value of all ninety-six program parameter registers is (0,0,0,0).

2.14.1.3 The Address Register

The Address Register is a single 4-component vector signed 32-bit integer register though only the x component of the vector is accessible. The register is private to each vertex program invocation

and is initialized to (0,0,0,0) at every vertex program invocation. This register can be written during vertex program execution (but not read) and its value can be used for as a relative offset for reading vertex program parameter registers. Only the vertex program parameter registers can be read using relative addressing (writes using relative addressing are not supported).

See the discussion of relative addressing of program parameters in section 2.14.1.9 and the discussion of the ARL instruction in section 2.14.1.10.1.

2.14.1.4 The Temporary Registers

The Temporary Registers are twelve 4-component floating-point vector registers used to hold temporary results during vertex program execution. These registers are numbered 0 through 11. These registers are private to each vertex program invocation and initialized to (0,0,0,0) at every vertex program invocation. These registers can be read and written during vertex program execution. Writes to these registers can be maskable on a per-component basis.

2.14.1.5 The Vertex Result Register Set

The Vertex Result Registers are fifteen 4-component floating-point vector registers used to write the results of a vertex program. Each register value is initialized to (0,0,0,1) at the invocation of each vertex program. Writes to the vertex result registers can be maskable on a per-component basis. These registers are named in Table X.1 and further discussed below.

Vertex Result Register Name	Description	Component Interpretation
HPOS	Homogeneous clip space position	(x,y,z,w)
COL0	Primary color (front-facing)	(r,g,b,a)
COL1	Secondary color (front-facing)	(r,g,b,a)
BFC0	Back-facing primary color	(r,g,b,a)
BFC1	Back-facing secondary color	(r,g,b,a)
FOGC	Fog coordinate	(f,*,*,*)
PSIZ	Point size	(p,*,*,*)
TEX0	Texture coordinate set 0	(s,t,r,q)
TEX1	Texture coordinate set 1	(s,t,r,q)
TEX2	Texture coordinate set 2	(s,t,r,q)
TEX3	Texture coordinate set 3	(s,t,r,q)
TEX4	Texture coordinate set 4	(s,t,r,q)
TEX5	Texture coordinate set 5	(s,t,r,q)
TEX6	Texture coordinate set 6	(s,t,r,q)
TEX7	Texture coordinate set 7	(s,t,r,q)

Table X.1: Vertex Result Registers.

HPOS is the transformed vertex's homogeneous clip space position. The vertex's homogeneous clip space position is converted to normalized device coordinates and transformed to window coordinates as described at the end of section 2.10 and in section 2.11. Further processing (subsequent to vertex program termination) is responsible for clipping primitives assembled from vertex program-generated vertices as described in section 2.10 but all client-defined clip planes are treated as if they are disabled when vertex program mode is enabled.

Four distinct color results can be generated for each vertex.

COL0 is the transformed vertex's front-facing primary color.
COL1 is the transformed vertex's front-facing secondary color.
BFC0 is the transformed vertex's back-facing primary color. BFC1 is the transformed vertex's back-facing secondary color.

Primitive coloring may operate in two-sided color mode. This behavior is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_TWO_SIDE_NV. The selection between the back-facing colors and the front-facing colors depends on the primitive of which the vertex is a part. If the primitive is a point or a line segment, the front-facing colors are always selected. If the primitive is a polygon and two-sided color mode is disabled, the front-facing colors are selected. If it is a polygon and two-sided color mode is enabled, then the selection is based on the sign of the (clipped or unclipped) polygon's signed area computed in window coordinates. This facingness determination is identical to the two-sided lighting facingness determination described in section 2.13.1.

The selected primary and secondary colors for each primitive are clamped to the range [0,1] and then interpolated across the assembled primitive during rasterization with at least 8-bit accuracy for each color component.

FOGC is the transformed vertex's fog coordinate. The register's first floating-point component is interpolated across the assembled primitive during rasterization and used as the fog distance to compute per-fragment the fog factor when fog is enabled. However, if both fog and vertex program mode are enabled, but the FOGC vertex result register is not written, the fog factor is overridden to 1.0. The register's other three components are ignored.

Point size determination may operate in program-specified point size mode. This behavior is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_POINT_SIZE_NV. If the vertex is for a point primitive and the mode is enabled and the PSIZ vertex result is written, the point primitive's size is determined by the clamped x component of the PSIZ register. Otherwise (because vertex program mode is disabled, program-specified point size mode is disabled, or because the vertex program did not write PSIZ), the point primitive's size is determined by the point size state (the state specified using the PointSize command).

The PSIZ register's x component is clamped to the range zero through either the hi value of ALIASED_POINT_SIZE_RANGE if point smoothing is disabled or the hi value of the SMOOTH_POINT_SIZE_RANGE if point smoothing is enabled. The register's other three components are ignored.

If the vertex is not for a point primitive, the value of the PSIZ vertex result register is ignored.

TEX0 through TEX7 are the transformed vertex's texture coordinate sets for texture units 0 through 7. These floating-point coordinates are interpolated across the assembled primitive during rasterization and used for accessing textures. If the number of texture units supported is less than eight, the values of vertex result registers that do not correspond to existent texture units are ignored.

2.14.1.6 Semantic Meaning for Vertex Attributes and Program Parameters

One important distinction between the conventional GL vertex transformation mode and the vertex program mode is that per-vertex

parameters and other state parameters in vertex program mode do not have dedicated semantic interpretations the way that they do with the conventional GL vertex transformation mode.

For example, in the conventional GL vertex transformation mode, the Normal command specifies a per-vertex normal. The semantic that the Normal command supplies a normal for lighting is established because that is how the per-vertex attribute supplied by the Normal command is used by the conventional GL vertex transformation mode. Similarly, other state parameters such as a light source position have semantic interpretations based on how the conventional GL vertex transformation model uses each particular parameter.

In contrast, vertex attributes and program parameters for vertex programs have no pre-defined semantic meanings. The meaning of a vertex attribute or program parameter in vertex program mode is defined by how the vertex attribute or program parameter is used by the current vertex program to compute and write values to the Vertex Result Registers. This is the reason that per-vertex attributes and program parameters for vertex programs are numbered instead of named.

For convenience however, the existing per-vertex parameters for the conventional GL vertex transformation mode (vertices, normals, colors, fog coordinates, vertex weights, and texture coordinates) are aliased to numbered vertex attributes. This aliasing is specified in Table X.2. The table includes how the various conventional components map to the 4-component vertex attribute components.

Vertex Attribute Register Number	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Conventional Component Mapping
0	vertex position	Vertex	x,y,z,w
1	vertex weights	VertexWeightEXT	w,0,0,1
2	normal	Normal	x,y,z,1
3	primary color	Color	r,g,b,a
4	secondary color	SecondaryColorEXT	r,g,b,1
5	fog coordinate	FogCoordEXT	fc,0,0,1
6	-	-	-
7	-	-	-
8	texture coord 0	MultiTexCoord(GL_TEXTURE0_ARB, ...)	s,t,r,q
9	texture coord 1	MultiTexCoord(GL_TEXTURE1_ARB, ...)	s,t,r,q
10	texture coord 2	MultiTexCoord(GL_TEXTURE2_ARB, ...)	s,t,r,q
11	texture coord 3	MultiTexCoord(GL_TEXTURE3_ARB, ...)	s,t,r,q
12	texture coord 4	MultiTexCoord(GL_TEXTURE4_ARB, ...)	s,t,r,q
13	texture coord 5	MultiTexCoord(GL_TEXTURE5_ARB, ...)	s,t,r,q
14	texture coord 6	MultiTexCoord(GL_TEXTURE6_ARB, ...)	s,t,r,q
15	texture coord 7	MultiTexCoord(GL_TEXTURE7_ARB, ...)	s,t,r,q

Table X.2: Aliasing of vertex attributes with conventional per-vertex parameters.

Only vertex attribute zero is treated specially because it is the attribute that provokes the execution of the vertex program; this is the attribute that aliases to the Vertex command's vertex coordinates.

The result of a vertex program is the set of post-transformation vertex parameters written to the Vertex Result Registers. All vertex programs must write a homogeneous clip space position, but the other Vertex Result Registers can be optionally written.

Clipping and culling are not the responsibility of vertex programs because these operations assume the assembly of multiple vertices into a primitive. View frustum clipping is performed subsequent to vertex program execution. Clip planes are not supported in vertex program mode.

2.14.1.7 Vertex Program Specification

Vertex programs are specified as an array of ubytes. The array is a string of ASCII characters encoding the program.

The command

```
LoadProgramNV(enum target, uint id, sizei len,
              const ubyte *program);
```

loads a vertex program when the target parameter is VERTEX_PROGRAM_NV. Multiple programs can be loaded with different names. id names the program to load. The name space for programs is the positive integers (zero is reserved). The error INVALID_VALUE occurs if a program is loaded with an id of zero. The error INVALID_OPERATION is generated if a program is loaded for an id that is currently loaded with a program of a different program target. Managing the program name space and binding to vertex programs is discussed later in section 2.14.1.8.

program is a pointer to an array of ubytes that represents the program being loaded. The length of the array is indicated by len.

A second program target type known as vertex state programs is discussed in 2.14.4.

At program load time, the program is parsed into a set of tokens possibly separated by white space. Spaces, tabs, newlines, carriage returns, and comments are considered whitespace. Comments begin with the character "#" and are terminated by a newline, a carriage return, or the end of the program array.

The Backus-Naur Form (BNF) grammar below specifies the syntactically valid sequences for vertex programs. The set of valid tokens can be inferred from the grammar. The token "" represents an empty string and is used to indicate optional rules. A program is invalid if it contains any undefined tokens or characters.

```
<program>              ::= "!!VP1.0" <instructionSequence> "END"

<instructionSequence>  ::= <instructionSequence> <instructionLine>
                         | <instructionLine>

<instructionLine>      ::= <instruction> ";"

<instruction>          ::= <ARL-instruction>
                         | <VECTORop-instruction>
                         | <SCALARop-instruction>
                         | <BINop-instruction>
                         | <TRIop-instruction>

<ARL-instruction>      ::= "ARL" <addrReg> "," <scalarSrcReg>

<VECTORop-instruction> ::= <VECTORop> <maskedDstReg> "," <swizzleSrcReg>

<SCALARop-instruction> ::= <SCALARop> <maskedDstReg> "," <scalarSrcReg>
```

```
<BINop-instruction>    ::= <BINop> <maskedDstReg> ","
                           <swizzleSrcReg> "," <swizzleSrcReg>

<TRIop-instruction>    ::= <TRIop> <maskedDstReg> ","
                           <swizzleSrcReg> "," <swizzleSrcReg> ","
                           <swizzleSrcReg>

<VECTORop>             ::= "MOV"
                         | "LIT"

<SCALARop>             ::= "RCP"
                         | "RSQ"
                         | "EXP"
                         | "LOG"

<BINop>                ::= "MUL"
                         | "ADD"
                         | "DP3"
                         | "DP4"
                         | "DST"
                         | "MIN"
                         | "MAX"
                         | "SLT"
                         | "SGE"

<TRIop>                ::= "MAD"

<scalarSrcReg>         ::= <optionalSign> <srcReg> <scalarSuffix>

<swizzleSrcReg>        ::= <optionalSign> <srcReg> <swizzleSuffix>

<maskedDstReg>         ::= <dstReg> <optionalMask>

<optionalMask>         ::= ""
                         | "." "x"
                         | "."     "y"
                         | "." "x" "y"
                         | "."         "z"
                         | "." "x"     "z"
                         | "."     "y" "z"
                         | "." "x" "y" "z"
                         | "."             "w"
                         | "." "x"         "w"
                         | "."     "y"     "w"
                         | "." "x" "y"     "w"
                         | "."         "z" "w"
                         | "." "x"     "z" "w"
                         | "."     "y" "z" "w"
                         | "." "x" "y" "z" "w"

<optionalSign>         ::= "-"
                         | ""

<srcReg>               ::= <vertexAttribReg>
                         | <progParamReg>
                         | <temporaryReg>

<dstReg>               ::= <temporaryReg>
                         | <vertexResultReg>

<vertexAttribReg>      ::= "v" "[" vertexAttribRegNum "]"

<vertexAttribRegNum>   ::= decimal integer from 0 to 15 inclusive
```

```
                          | "OPOS"
                          | "WGHT"
                          | "NRML"
                          | "COL0"
                          | "COL1"
                          | "FOGC"
                          | "TEX0"
                          | "TEX1"
                          | "TEX2"
                          | "TEX3"
                          | "TEX4"
                          | "TEX5"
                          | "TEX6"
                          | "TEX7"

<progParamReg>          ::= <absProgParamReg>
                          | <relProgParamReg>

<absProgParamReg>       ::= "c" "[" <progParamRegNum> "]"

<progParamRegNum>       ::= decimal integer from 0 to 95 inclusive

<relProgParamReg>       ::= "c" "[" <addrReg> "]"
                          | "c" "[" <addrReg> "+" <progParamPosOffset> "]"
                          | "c" "[" <addrReg> "-" <progParamNegOffset> "]"

<progParamPosOffset>    ::= decimal integer from 0 to 63 inclusive

<progParamNegOffset>    ::= decimal integer from 0 to 64 inclusive

<addrReg>               ::= "A0" "." "x"

<temporaryReg>          ::= "R0"
                          | "R1"
                          | "R2"
                          | "R3"
                          | "R4"
                          | "R5"
                          | "R6"
                          | "R7"
                          | "R8"
                          | "R9"
                          | "R10"
                          | "R11"

<vertexResultReg>       ::= "o" "[" vertexResultRegName "]"

<vertexResultRegName>   ::= "HPOS"
                          | "COL0"
                          | "COL1"
                          | "BFC0"
                          | "BFC1"
                          | "FOGC"
                          | "PSIZ"
                          | "TEX0"
                          | "TEX1"
                          | "TEX2"
                          | "TEX3"
                          | "TEX4"
                          | "TEX5"
                          | "TEX6"
                          | "TEX7"
```

```
<scalarSuffix>          ::= "." <component>

<swizzleSuffix>         ::= ""
                          | "." <component>
                          | "." <component> <component>
                                <component> <component>

<component>             ::= "x"
                          | "y"
                          | "z"
                          | "w"
```

The <vertexAttribRegNum> rule matches both register numbers 0 through 15 and a set of mnemonics that abbreviate the aliasing of conventional the per-vertex parameters to vertex attribute register numbers. Table X.3 shows the mapping from mnemonic to vertex attribute register number and what the mnemonic abbreviates.

Mnemonic	Vertex Attribute Register Number	Meaning
"OPOS"	0	object position
"WGHT"	1	vertex weight
"NRML"	2	normal
"COL0"	3	primary color
"COL1"	4	secondary color
"FOGC"	5	fog coordinate
"TEX0"	8	texture coordinate 0
"TEX1"	9	texture coordinate 1
"TEX2"	10	texture coordinate 2
"TEX3"	11	texture coordinate 3
"TEX4"	12	texture coordinate 4
"TEX5"	13	texture coordinate 5
"TEX6"	14	texture coordinate 6
"TEX7"	15	texture coordinate 7

Table X.3: The mapping between vertex attribute register numbers, mnemonics, and meanings.

A vertex programs fails to load if it does not write at least one component of the HPOS register.

A vertex program fails to load if it contains more than 128 instructions.

A vertex program fails to load if any instruction sources more than one unique program parameter register.

A vertex program fails to load if any instruction sources more than one unique vertex attribute register.

The error INVALID_OPERATION is generated if a vertex program fails to load because it is not syntactically correct or for one of the semantic restrictions listed above.

The error INVALID_OPERATION is generated if a program is loaded for id when id is currently loaded with a program of a different target.

A successfully loaded vertex program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 2.14.1.10.

A successfully loaded program replaces the program previously assigned to the name specified by id. If the OUT_OF_MEMORY error is generated by LoadProgramNV, no change is made to the previous contents of the named program.

Querying the value of PROGRAM_ERROR_POSITION_NV returns a ubyte offset into the last loaded program string indicating where the first error in the program. If the program fails to load because of a semantic restriction that cannot be determined until the program is fully scanned, the error position will be len, the length of the program. If the program loads successfully, the value of PROGRAM_ERROR_POSITION_NV is assigned the value negative one.

2.14.1.8 Vertex Program Binding and Program Management

The current vertex program is invoked whenever vertex attribute zero is updated (whether by a VertexAttributeNV or Vertex command). The current vertex program is updated by

```
BindProgramNV(enum target, uint id);
```

where target must be VERTEX_PROGRAM_NV. This binds the vertex program named by id as the current vertex program. The error INVALID_OPERATION is generated if id names a program that is not a vertex program (for example, if id names a vertex state program as described in section 2.14.4).

Binding to a nonexistent program id does not generate an error. In particular, binding to program id zero does not generate an error. However, because program zero cannot be loaded, program zero is always nonexistent. If a program id is successfully loaded with a new vertex program and id is also the currently bound vertex program, the new program is considered the currently bound vertex program.

The INVALID_OPERATION error is generated when both vertex program mode is enabled and Begin is called (or when a command that performs an implicit Begin is called) if the current vertex program is nonexistent or not valid. A vertex program may not be valid for reasons explained in section 2.14.5.

Programs are deleted by calling

```
void DeleteProgramsNV(sizei n, const uint *ids);
```

ids contains n names of programs to be deleted. After a program is deleted, it becomes nonexistent, and its name is again unused. If a program that is currently bound is deleted, it is as though BindProgramNV has been executed with the same target as the deleted program and program zero. Unused names in ids are silently ignored, as is the value zero.

The command

```
void GenProgramsNV(sizei n, uint *ids);
```

returns n previously unused program names in ids. These names are marked as used, for the purposes of GenProgramsNV only, but they become existent programs only when the are first loaded using LoadProgramNV. The error INVALID_VALUE is generated if n is negative.

An implementation may choose to establish a working set of programs on which binding and ExecuteProgramNV operations (execute programs are

explained in section 2.14.4) are performed with higher performance. A program that is currently part of this working set is said to be resident.

The command

```
boolean AreProgramsResidentNV(sizei n, const uint *ids,
                              boolean *residences);
```

returns TRUE if all of the n programs named in ids are resident, or if the implementation does not distinguish a working set. If at least one of the programs named in ids is not resident, then FALSE is returned, and the residence of each program is returned in residences. Otherwise the contents of residences are not changed. If any of the names in ids are nonexistent or zero, FALSE is returned, the error INVALID_VALUE is generated, and the contents of residences are indeterminate. The residence status of a single named program can also be queried by calling GetProgramivNV with id set to the name of the program and pname set to PROGRAM_RESIDENT_NV.

AreProgramsResidentNV indicates only whether a program is currently resident, not whether it could not be made resident. An implementation may choose to make a program resident only on first use, for example. The client may guide the GL implementation in determining which programs should be resident by requesting a set of programs to make resident.

The command

```
void RequestResidentProgramsNV(sizei n, const uint *ids);
```

requests that the n programs named in ids should be made resident. While all the programs are not guaranteed to become resident, the implementation should make a best effort to make as many of the programs resident as possible. As a result of making the requested programs resident, program names not among the requested programs may become non-resident. Higher priority for residency should be given to programs listed earlier in the ids array. RequestResidentProgramsNV silently ignores attempts to make resident nonexistent program names or zero. AreProgramsResidentNV can be called after RequestResidentProgramsNV to determine which programs actually became resident.

2.14.1.9 Vertex Program Register Accesses

There are 17 vertex program instructions. The instructions and their respective input and output parameters are summarized in Table X.4.

Opcode	Inputs (scalar or vector)	Output (vector or replicated scalar)	Operation
ARL	s	address register	address register load
MOV	v	v	move
MUL	v,v	v	multiply
ADD	v,v	v	add
MAD	v,v,v	v	multiply and add
RCP	s	ssss	reciprocal
RSQ	s	ssss	reciprocal square root
DP3	v,v	ssss	3-component dot product
DP4	v,v	ssss	4-component dot product
DST	v,v	v	distance vector
MIN	v,v	v	minimum

MAX	v,v	v	maximum
SLT	v,v	v	set on less than
SGE	v,v	v	set on greater equal than
EXP	s	v	exponential base 2
LOG	s	v	logarithm base 2
LIT	v	v	light coefficients

Table X.4: Summary of vertex program instructions. "v" indicates a vector input or output, "s" indicates a scalar input, and "ssss" indicates a scalar output replicated across a 4-component vector.

Instructions use either scalar source values or swizzled source values, indicated in the grammar (see section 2.14.1.7) by the rules <scalarSrcReg> and <swizzleSrcReg> respectively. Either type of source value is negated when the <optionalSign> rule matches "-".

Scalar source register values select one of the source register's four components based on the <component> of the <scalarSuffix> rule. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. The indicated component is used as a scalar for the particular source value.

Swizzled source register values may arbitrarily swizzle the source register's components based on the <swizzleSuffix> rule. In the case where the <swizzleSuffix> matches (ignoring whitespace) the pattern ".????" where each question mark is one of "x", "y", "z", or "w", this indicates the ith component of the source register value should come from the component named by the ith component in the sequence. For example, if the swizzle suffix is ".yzzx" and the source register contains [2.0, 8.0, 9.0, 0.0] the swizzled source register value used by the instruction is [8.0, 9.0, 9.0, 2.0].

If the <swizzleSuffix> rule matches "", this is treated the same as ".xyzw". If the <swizzleSuffix> rule matches (ignoring whitespace) ".x", ".y", ".z", or ".w", these are treated the same as ".xxxx", ".yyyy", ".zzzz", and ".wwww" respectively.

The register sourced for either a scalar source register value or a swizzled source register value is indicated in the grammar by the rule <srcReg>. The <vertexAttribReg>, <progParamReg>, and <temporaryReg> sub-rules correspond to one of the vertex attribute registers, program parameter registers, or temporary register respectively.

The vertex attribute and temporary registers are accessed absolutely based on the numbered register. In the case of vertex attribute registers, if the <vertexAttribRegNum> corresponds to a mnemonic, the corresponding register number from Table X.3 is used.

Either absolute or relative addressing can be used to access the program parameter registers. Absolute addressing is indicated by the grammar by the <absProgParamReg> rule. Absolute addressing accesses the numbered program parameter register indicated by the <progParamRegNum> rule. Relative addressing accesses the numbered program parameter register plus an offset. The offset is the positive value of <progParamPosOffset> if the <progParamPosOffset> rule is matched, or the offset is the negative value of <progParamNegOffset> if the <progParamNegOffset> rule is matched, or otherwise the offset is zero. Relative addressing is available only for program parameter registers and only for reads (not writes). Relative addressing reads outside of the 0 to 95 inclusive range always read the value (0,0,0,0).

The result of all instructions except ARL is written back to a

masked destination register, indicated in the grammar by the rule <maskedDstReg>.

Writes to each component of the destination register can be masked, indicated in the grammar by the <optionalMask> rule. If the optional mask is "", all components are written. Otherwise, the optional mask names particular components to write. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. For example, an optional mask of ".xzw" indicates that the x, z, and w components should be written but not the y component. The grammar requires that the destination register mask components must be listed in "xyzw" order.

The actual destination register is indicated in the grammar by the rule <dstReg>. The <temporaryReg> and <vertexResultReg> sub-rules correspond to either the temporary registers or vertex result registers. The temporary registers are determined and accessed as described earlier.

The vertex result registers are accessed absolutely based on the named register. The <vertexResultRegName> rule corresponds to registers named in Table X.1.

2.14.1.10 Vertex Program Instruction Set Operations

The operation of the 17 vertex program instructions are described in this section. After the textual description of each instruction's operation, a register transfer level description is also presented.

The following conventions are used in each instruction's register transfer level description. The 4-component vector variables "t", "u", and "v" are assigned intermediate results. The destination register is called "destination". The three possible source registers are called "source0", "source1", and "source2" respectively.

The x, y, z, and w vector components are referred to with the suffixes ".x", ".y", ".z", and ".w" respectively. The suffix ".c" is used for scalar source register values and c represents the particular source register's selected scalar component. Swizzling of components is indicated with the suffixes ".c***", ".*c**", ".**c*", and ".***c" where c is meant to indicate the x, y, z, or w component selected for the particular source operand swizzle configuration. For example:

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
```

This example indicates that t should be assigned the swizzled version of the source0 operand based on the source0 operand's swizzle configuration.

The variables "negate0", "negate1", and "negate2" are booleans that are true when the respective source value should be negated. The variables "xmask", "ymask", "zmask", and "wmask" are booleans that are true when the destination write mask for the respective component is enabled for writing.

Otherwise, the register transfer level descriptions mimic ANSI C syntax.

The idiom "IEEE(expression)" represents the s23e8 single-precision result of the expression if evaluated using IEEE single-precision

floating point operations. The IEEE idiom is used to specify the maximum allowed deviation from IEEE single-precision floating-point arithmetic results.

The following abbreviations are also used:

+Inf	floating-point representation of positive infinity
-Inf	floating-point representation of negative infinity
+NaN	floating-point representation of positive not a number
-NaN	floating-point representation of negative not a number
NA	not applicable or not used

2.14.1.10.1 ARL: Address Register Load

The ARL instruction moves value of the source scalar into the address register. Conceptually, the address register load instruction is a 4-component vector signed integer register, but the only valid address register component for writing and indexing is the x component. The only use for A0.x is as a base address for program parameter reads. The source value is a float that is truncated towards negative infinity into a signed integer.

```
t.x = source0.c;
if (negate0) t.x = -t.x;
A0.x = floor(t.x);
```

2.14.1.10.2 MOV: Move

The MOV instruction moves the value of the source vector into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
if (xmask) destination.x = t.x;
if (ymask) destination.y = t.y;
if (zmask) destination.z = t.z;
if (wmask) destination.w = t.w;
```

2.14.1.10.3 MUL: Multiply

The MUL instruction multiplies the values of the two source vectors into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
```

```
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = t.x * u.x;
    if (ymask) destination.y = t.y * u.y;
    if (zmask) destination.z = t.z * u.z;
    if (wmask) destination.w = t.w * u.w;
```

2.14.1.10.4 ADD: Add

The ADD instruction adds the values of the two source vectors into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = t.x + u.x;
    if (ymask) destination.y = t.y + u.y;
    if (zmask) destination.z = t.z + u.z;
    if (wmask) destination.w = t.w + u.w;
```

2.14.1.10.5 MAD: Multiply and Add

The MAD instruction adds the value of the third source vector to the product of the values of the first and second two source vectors, writing the result to the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
```

```
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    v.x = source2.c***;
    v.y = source2.*c**;
    v.z = source2.**c*;
    v.w = source2.***c;
    if (negate2) {
      v.x = -v.x;
      v.y = -v.y;
      v.z = -v.z;
      v.w = -v.w;
    }
    if (xmask) destination.x = t.x * u.x + v.x;
    if (ymask) destination.y = t.y * u.y + v.y;
    if (zmask) destination.z = t.z * u.z + v.z;
    if (wmask) destination.w = t.w * u.w + v.w;
```

2.14.1.10.6 RCP: Reciprocal

The RCP instruction inverts the value of the source scalar into the destination register. The reciprocal of exactly 1.0 must be exactly 1.0.

Additionally the reciprocal of negative infinity gives [-0.0, -0.0, -0.0, -0.0]; the reciprocal of negative zero gives [-Inf, -Inf, -Inf, -Inf]; the reciprocal of positive zero gives [+Inf, +Inf, +Inf, +Inf]; and the reciprocal of positive infinity gives [0.0, 0.0, 0.0, 0.0].

```
    t.x = source0.c;
    if (negate0) {
      t.x = -t.x;
    }
    if (t.x == 1.0f) {
      u.x = 1.0f;
    } else {
      u.x = 1.0f / t.x;
    }
    if (xmask) destination.x = u.x;
    if (ymask) destination.y = u.x;
    if (zmask) destination.z = u.x;
    if (wmask) destination.w = u.x;
```

where

```
    | u.x - IEEE(1.0f/t.x) | < 1.0f/(2^22)
```

for 1.0f <= t.x <= 2.0f. The intent of this precision requirement is that this amount of relative precision apply over all values of t.x.

2.14.1.10.7 RSQ: Reciprocal Square Root

The RSQ instruction assigns the inverse square root of the absolute value of the source scalar into the destination register.

Additionally, RSQ(0.0) gives [+Inf, +Inf, +Inf, +Inf]; and both RSQ(+Inf) and RSQ(-Inf) give [0.0, 0.0, 0.0, 0.0];

```
    t.x = source0.c;
    if (negate0) {
```

```
      t.x = -t.x;
    }
    u.x = 1.0f / sqrt(fabs(t.x));
    if (xmask) destination.x = u.x;
    if (ymask) destination.y = u.x;
    if (zmask) destination.z = u.x;
    if (wmask) destination.w = u.x;
```

where

```
    | u.x - IEEE(1.0f/sqrt(fabs(t.x))) | < 1.0f/(2^22)
```

for 1.0f <= t.x <= 4.0f. The intent of this precision requirement is that this amount of relative precision apply over all values of t.x.

2.14.1.10.8 DP3: Three-Component Dot Product

The DP3 instruction assigns the three-component dot product of the two source vectors into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
    }
    v.x = t.x * u.x + t.y * u.y + t.z * u.z;
    if (xmask) destination.x = v.x;
    if (ymask) destination.y = v.x;
    if (zmask) destination.z = v.x;
    if (wmask) destination.w = v.x;
```

2.14.1.10.9 DP4: Four-Component Dot Product

The DP4 instruction assigns the four-component dot product of the two source vectors into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
```

```
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
v.x = t.x * u.x + t.y * u.y + t.z * u.z + t.w * u.w;
if (xmask) destination.x = v.x;
if (ymask) destination.y = v.x;
if (zmask) destination.z = v.x;
if (wmask) destination.w = v.x;
```

2.14.1.10.10 DST: Distance Vector

The DST instructions calculates a distance vector for the values of two source vectors. The first vector is assumed to be [NA, d*d, d*d, NA] and the second source vector is assumed to be [NA, 1.0/d, NA, 1.0/d], where the value of a component labeled NA is undefined. The destination vector is then assigned [1,d,d*d,1.0/d].

```
t.y = source0.*c**;
t.z = source0.**c*;
if (negate0) {
  t.y = -t.y;
  t.z = -t.z;
}
u.y = source1.*c**;
u.w = source1.***c;
if (negate1) {
  u.y = -u.y;
  u.w = -u.w;
}
if (xmask) destination.x = 1.0;
if (ymask) destination.y = t.y*u.y;
if (zmask) destination.z = t.z;
if (wmask) destination.w = u.w;
```

2.14.1.10.11 MIN: Minimum

The MIN instruction assigns the component-wise minimum of the two source vectors into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
if (xmask) destination.x = (t.x < u.x) ? t.x : u.x;
if (ymask) destination.y = (t.y < u.y) ? t.y : u.y;
if (zmask) destination.z = (t.z < u.z) ? t.z : u.z;
```

```
    if (wmask) destination.w = (t.w < u.w) ? t.w : u.w;
```

2.14.1.10.12 MAX: Maximum

The MAX instruction assigns the component-wise maximum of the two source vectors into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = (t.x >= u.x) ? t.x : u.x;
    if (ymask) destination.y = (t.y >= u.y) ? t.y : u.y;
    if (zmask) destination.z = (t.z >= u.z) ? t.z : u.z;
    if (wmask) destination.w = (t.w >= u.w) ? t.w : u.w;
```

2.14.1.10.13 SLT: Set On Less Than

The SLT instruction performs a component-wise assignment of either 1.0 or 0.0 into the destination register. 1.0 is assigned if the value of the first source vector is less than the value of the second source vector; otherwise, 0.0 is assigned.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = (t.x < u.x) ? 1.0 : 0.0;
    if (ymask) destination.y = (t.y < u.y) ? 1.0 : 0.0;
    if (zmask) destination.z = (t.z < u.z) ? 1.0 : 0.0;
    if (wmask) destination.w = (t.w < u.w) ? 1.0 : 0.0;
```

2.14.1.10.14 SGE: Set On Greater or Equal Than

The SGE instruction performs a component-wise assignment of either 1.0 or 0.0 into the destination register. 1.0 is assigned if the value of the first source vector is greater than or equal the value of the second source vector; otherwise, 0.0 is assigned.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
if (xmask) destination.x = (t.x >= u.x) ? 1.0 : 0.0;
if (ymask) destination.y = (t.y >= u.y) ? 1.0 : 0.0;
if (zmask) destination.z = (t.z >= u.z) ? 1.0 : 0.0;
if (wmask) destination.w = (t.w >= u.w) ? 1.0 : 0.0;
```

2.14.1.10.15 EXP: Exponential Base 2

The EXP instruction generates an approximation of the exponential base 2 for the value of a source scalar. This approximation is assigned to the z component of the destination register. Additionally, the x and y components of the destination register are assigned values useful for determining a more accurate approximation. The exponential base 2 of the source scalar can be better approximated by destination.x*FUNC(destination.y) where FUNC is some user approximation (presumably implemented by subsequent instructions in the vertex program) to 2^destination.y where 0.0 <= destination.y < 1.0.

Additionally, EXP(-Inf) or if the exponential result underflows gives [0.0, 0.0, 0.0, 0.0]; and EXP(+Inf) or if the exponential result overflows gives [+Inf, 0.0, +Inf, 1.0].

```
t.x = source0.c;
if (negate0) {
  t.x = -t.x;
}
q.x = 2^floor(t.x);
q.y = t.x - floor(t.x);
q.z = q.x * APPX(q.y);
if (xmask) destination.x = q.x;
if (ymask) destination.y = q.y;
if (zmask) destination.z = q.z;
if (wmask) destination.w = 1.0;
```

where APPX is an implementation dependent approximation of exponential

base 2 such that

```
    | exp(q.y*log(2.0))-APPX(q.y) | < 1/(2^11)
```

for all 0 <= q.y < 1.0.

The expression "2^floor(t.x)" should overflow to +Inf and underflow to zero.

2.14.1.10.16 LOG: Logarithm Base 2

The LOG instruction generates an approximation of the logarithm base 2 for the absolute value of a source scalar. This approximation is assigned to the z component of the destination register. Additionally, the x and y components of the destination register are assigned values useful for determining a more accurate approximation. The logarithm base 2 of the absolute value of the source scalar can be better approximated by destination.x+FUNC(destination.y) where FUNC is some user approximation (presumably implemented by subsequent instructions in the vertex program) of log2(destination.y) where 1.0 <= destination.y < 2.0.

Additionally, LOG(0.0) gives [-Inf, 1.0, -Inf, 1.0]; and both LOG(+Inf) and LOG(-Inf) give [+Inf, 1.0, +Inf, 1.0].

```
    t.x = source0.c;
    if (negate0) {
      t.x = -t.x;
    }
    if (fabs(t.x) != 0.0f) {
      if (fabs(t.x) == +Inf) {
        q.x = +Inf;
        q.y = 1.0;
        q.z = +Inf;
      } else {
        q.x = Exponent(t.x);
        q.y = Mantissa(t.x);
        q.z = q.x + APPX(q.y);
      }
    } else {
      q.x = -Inf;
      q.y = 1.0;
      q.z = -Inf;
    }
    if (xmask) destination.x = q.x;
    if (ymask) destination.y = q.y;
    if (zmask) destination.z = q.z;
    if (wmask) destination.w = 1.0;
```

where APPX is an implementation dependent approximation of logarithm base 2 such that

```
    | log(q.y)/log(2.0) - APPX(q.y) | < 1/(2^11)
```

for all 1.0 <= q.y < 2.0.

The "Exponent(t.x)" function returns the unbiased exponent between -126 and 127. For example, "Exponent(1.0)" equals 0.0. (Note that the IEEE floating-point representation maintains the exponent as a biased value.) Larger or smaller exponents should generate +Inf or -Inf respectively. The "Mantissa(t.x)" function returns a value in the range [1.0f, 2.0). The intent of these functions is that fabs(t.x) is approximately "Mantissa(t.x)*2^Exponent(t.x)".

2.14.1.10.17 LIT: Light Coefficients

The LIT instruction is intended to compute ambient, diffuse, and specular lighting coefficients from a diffuse dot product, a specular dot product, and a specular power that is clamped to (-128,128) exclusive. The x component of the source vector is assumed to contain a diffuse dot product (unit normal vector dotted with a unit light vector). The y component of the source vector is assumed to contain a Blinn specular dot product (unit normal vector dotted with a unit half-angle vector). The w component is assumed to contain a specular power.

An implementation must support at least 8 fraction bits in the specular power. Note that because 0.0 times anything must be 0.0, taking any base to the power of 0.0 will yield 1.0.

```
t.x = source0.c***;
t.y = source0.*c**;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.w = -t.w;
}
if (t.w < -(128.0-epsilon)) t.w = -(128.0-epsilon);
else if (t.w > 128-epsilon) t.w = 128-epsilon;
if (t.x < 0.0) t.x = 0.0;
if (t.y < 0.0) t.y = 0.0;
if (xmask) destination.x = 1.0;
if (ymask) destination.y = t.x;
if (zmask) destination.z = (t.x > 0.0) ? EXP(t.w*LOG(t.y)) : 0.0;
if (wmask) destination.w = 1.0;
```

where EXP and LOG are functions that approximate the exponential base 2 and logarithm base 2 with the identical accuracy and special case requirements of the EXP and LOG instructions. epsilon is 1.0/256.0 or approximately 0.0039 which would correspond to representing the specular power with a s8.8 representation.

2.14.1.11 Vertex Program Floating Point Requirements

All vertex program calculations are assumed to use IEEE single precision floating-point math with a format of s1e8m23 (one signed bit, 8 bits of exponent, 23 bits of magnitude) or better and the round-to-zero rounding mode. The only exceptions to this are the RCP, RSQ, LOG, EXP, and LIT instructions.

Note that (positive or negative) 0.0 times anything is (positive) 0.0.

The RCP and RSQ instructions deliver results accurate to 1.0/(2^22) and the approximate output (the z component) of the EXP and LOG instructions only has to be accurate to 1.0/(2^11). The LIT instruction specular output (the z component) is allowed an error equivalent to the combination of the EXP and LOG combination to implement a power function.

The floor operations used by the ARL and EXP instructions must operate identically. Specifically, the EXP instruction's floor(t.x) intermediate result must exactly match the integer stored in the address register by the ARL instruction.

Since distance is calculated as (d^2)*(1/sqrt(d^2)), 0.0 multiplied by anything must be 0.0. This affects the MUL, MAD, DP3, DP4, DST, and LIT instructions.

Because if/then/else conditional evaluation is done by multiplying by 1.0 or 0.0 and adding, the floating point computations require:

```
  0.0 * x = 0.0    for all x (including +Inf, -Inf, +NaN, and -NaN)
  1.0 * x = x      for all x (including +Inf and -Inf)
  0.0 + x = x      for all x (including +Inf and -Inf)
```

Including +Inf, -Inf, +NaN, and -NaN when applying the above three rules is recommended but not required. (The recommended inclusion of +Inf, -Inf, +NaN, and -NaN when applying the first rule is inconsistent with IEEE floating-point requirements.)

For the purpose of comparisons performed by the SGE and SLT instructions, -0.0 is less than +0.0. (This is inconsistent with IEEE floating-point requirements).

No floating-point exceptions or interrupts are generated. Denorms are not supported; if a denorm is input, it is treated as 0.0 (ie, denorms are flushed to zero).

Computations involving +NaN or -NaN generate +NaN, except for the requirement that zero times +NaN or -NaN must always be zero. (This exception is inconsistent with IEEE floating-point requirements).

2.14.2 Vertex Program Update for the Current Raster Position

When vertex programs are enabled, the raster position is determined by the current vertex program. The raster position specified by RasterPos is treated as if they were specified in a Vertex command. The contents of vertex result register set is used to update respective raster position state.

Assuming an existent program, the homogeneous clip-space coordinates are passed to clipping as if they represented a point and assuming no client-defined clip planes are enabled. If the point is not culled, then the projection to window coordinates is computed (section 2.10) and saved as the current raster position and the valid bit is set. If the current vertex program is nonexistent or the "point" is culled, the current raster position and its associated data become indeterminate and the raster position valid bit is cleared.

2.14.3 Vertex Arrays for Vertex Attributes

Data for vertex attributes in vertex program mode may be specified using vertex array commands. The client may specify and enable any of sixteen vertex attribute arrays.

The vertex attribute arrays are ignored when vertex program mode is disabled. When vertex program mode is enabled, vertex attribute arrays are used.

The command

```
  void VertexAttribPointerNV(uint index, int size, enum type,
                             sizei stride, const void *pointer);
```

describes the locations and organizations of the sixteen vertex attribute arrays. index specifies the particular vertex attribute to be described. size indicates the number of values per vertex

that are stored in the array; size must be one of 1, 2, 3, or 4. type specifies the data type of the values stored in the array. type must be one of SHORT, FLOAT, DOUBLE, or UNSIGNED_BYTE and these values correspond to the array types short, int, float, double, and ubyte respectively. The INVALID_OPERATION error is generated if type is UNSIGNED_BYTE and size is not 4. The INVALID_VALUE error is generated if index is greater than 15. The INVALID_VALUE error is generated if stride is negative.

The one, two, three, or four values in an array that correspond to a single vertex attribute comprise an array element. The values within each array element at stored sequentially in memory. If the stride is specified as zero, then array elements are stored sequentially as well. Otherwise points to the ith and (i+1)st elements of an array differ by stride basic machine units (typically unsigned bytes), the pointer to the (i+1)st element being greater. pointer specifies the location in memory of the first value of the first element of the array being specified.

Vertex attribute arrays are enabled with the EnableClientState command and disabled with the DisableClientState command. The value of the argument to either command is VERTEX_ATTRIB_ARRAYi_NV where i is an integer between 0 and 15; specifying a value of i enables or disables the vertex attribute array with index i. The constants obey VERTEX_ATTRIB_ARRAYi_NV = VERTEX_ATTRIB_ARRAY0_NV + i.

When vertex program mode is enabled, the ArrayElement command operates as described in this section in contrast to the behavior described in section 2.8. Likewise, any vertex array transfer commands that are defined in terms of ArrayElement (DrawArrays, DrawElements, and DrawRangeElements) assume the operation of ArrayElement described in this section when vertex program mode is enabled.

When vertex program mode is enabled, the ArrayElement command transfers the ith element of particular enabled vertex arrays as described below. For each enabled vertex attribute array, it is as though the corresponding command from section 2.14.1.1 were called with a pointer to element i. For each vertex attribute, the corresponding command is VertexAttrib[size][type]v, where size is one of [1,2,3,4], and type is one of [s,f,d,ub], corresponding to the array types short, int, float, double, and ubyte respectively.

However, if a given vertex attribute array is disabled, but its corresponding aliased conventional per-vertex parameter's vertex array (as described in section 2.14.1.6) is enabled, then it is as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element i. In this case, the corresponding command is determined as described in section 2.8's description of ArrayElement.

If the vertex attribute array 0 is enabled, it is as though VertexAttrib[size][type]v(0, ...) is executed last, after the executions of other corresponding commands. If the vertex attribute array 0 is disabled but the vertex array is enabled, it is as though Vertex[size][type]v is executed last, after the executions of other corresponding commands.

2.14.4 Vertex State Programs

Vertex state programs share the same instruction set as and a similar execution model to vertex programs. While vertex program are executed implicitly when a vertex transformation is provoked, vertex state programs are executed explicitly, independently of any vertices.

Vertex state programs can write program parameter registers, but may not write vertex result registers.

The purpose of a vertex state program is to update program parameter registers by means of an application-defined program. Typically, an application will load a set of program parameters and then execute a vertex state program that reads and updates the program parameter registers. For example, a vertex state program might normalize a set of unnormalized vectors previously loaded as program parameters. The expectation is that subsequently executed vertex programs would use the normalized program parameters.

Vertex state programs are loaded with the same LoadProgramNV command (see section 2.14.1.7) used to load vertex programs except that the target must be VERTEX_STATE_PROGRAM_NV when loading a vertex state program.

Vertex state programs must conform to a more limited grammar than the grammar for vertex programs. The vertex state program grammar for syntactically valid sequences is the same as the grammar defined in section 2.14.1.7 with the following modified rules:

```
<program>              ::= "!!VSP1.0" <instructionSequence> "END"

<dstReg>               ::= <absProgParamReg>
                         | <temporaryReg>

<vertexAttribReg>      ::= "v" "[" "0" "]"
```

A vertex state program fails to load if it does not write at least one program parameter register.

A vertex state program fails to load if it contains more than 128 instructions.

A vertex state program fails to load if any instruction sources more than one unique program parameter register.

A vertex state program fails to load if any instruction sources more than one unique vertex attribute register (this is necessarily true because only vertex attribute 0 is available in vertex state programs).

The error INVALID_OPERATION is generated if a vertex state program fails to load because it is not syntactically correct or for one of the other reasons listed above.

A successfully loaded vertex state program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 2.14.1.10.

Executing vertex state programs is legal only outside a Begin/End pair. A vertex state program may not read any vertex attribute register other than register zero. A vertex state program may not write any vertex result register.

The command

```
ExecuteProgramNV(enum target, uint id, const float *params);
```

executes the vertex state program named by id. The target must be VERTEX_STATE_PROGRAM_NV and the id must be the name of program loaded

with a target type of VERTEX_STATE_PROGRAM_NV. params points to an array of four floating-point values that are loaded into vertex attribute register zero (the only vertex attribute readable from a vertex state program).

The INVALID_OPERATION error is generated if the named program is nonexistent, is invalid, or the program is not a vertex state program. A vertex state program may not be valid for reasons explained in section 2.14.5.

2.14.5 Tracking Matrices

As a convenience to applications, standard GL matrix state can be tracked into program parameter vectors. This permits vertex programs to access matrices specified through GL matrix commands.

In addition to GL's conventional matrices, several additional matrices are available for tracking. These matrices have names of the form MATRIXi_NV where i is between zero and n-1 where n is the value of the MAX_TRACK_MATRICES_NV implementation dependent constant. The MATRIXi_NV constants obey MATRIXi_NV = MATRIX0_NV + i. The value of MAX_TRACK_MATRICES_NV must be at least eight. The maximum stack depth for tracking matrices is defined by the MAX_TRACK_MATRIX_STACK_DEPTH_NV and must be at least 1.

The command

```
TrackMatrixNV(enum target, uint address, enum matrix, enum transform);
```

tracks a given transformed version of a particular matrix into a contiguous sequence of four vertex program parameter registers beginning at address. target must be VERTEX_PROGRAM_NV (though tracked matrices apply to vertex state programs as well because both vertex state programs and vertex programs shared the same program parameter registers). matrix must be one of NONE, MODELVIEW, PROJECTION, TEXTURE, TEXTUREi_ARB (where i is between 0 and n-1 where n is the number of texture units supported), COLOR (if the ARB_imaging subset is supported), MODELVIEW_PROJECTION_NV, or MATRIXi_NV. transform must be one of IDENTITY_NV, INVERSE_NV, TRANSPOSE_NV, or INVERSE_TRANSPOSE_NV. The INVALID_VALUE error is generated if address is not a multiple of four.

The MODELVIEW_PROJECTION_NV matrix represents the concatenation of the current modelview and projection matrices. If M is the current modelview matrix and P is the current projection matrix, then the MODELVIEW_PROJECTION_NV matrix is C and computed as

$$C = P M$$

Matrix tracking for the specified program parameter register and the next consecutive three registers is disabled when NONE is supplied for matrix. When tracking is disabled the previously tracked program parameter registers retain the state of their last tracked values. Otherwise, the specified transformed version of matrix is tracked into the specified program parameter register and the next three registers. Whenever the matrix changes, the transformed version of the matrix is updated in the specified range of program parameter registers. If TEXTURE is specified for matrix, the texture matrix for the current active texture unit is tracked. If TEXTUREi_ARB is specified for matrix, the <i>th texture matrix is tracked.

Matrices are tracked row-wise meaning that the top row of the transformed matrix is loaded into the program parameter address,

the second from the top row of the transformed matrix is loaded into the program parameter address+1, the third from the top row of the transformed matrix is loaded into the program parameter address+2, and the bottom row of the transformed matrix is loaded into the program parameter address+3. The transformed matrix may be identical to the specified matrix, the inverse of the specified matrix, the transpose of the specified matrix, or the inverse transpose of the specified matrix, depending on the value of transform.

When matrix tracking is enabled for a particular program parameter register sequence, updates to the program parameter using ProgramParameterNV commands, a vertex program, or a vertex state program are not possible. The INVALID_OPERATION error is generated if a ProgramParameterNV command is used to update a program parameter register currently tracking a matrix.

The INVALID_OPERATION error is generated by ExecuteProgramNV when the vertex state program requested for execution writes to a program parameter register that is currently tracking a matrix because the program is considered invalid.

2.14.6 Required Vertex Program State

The state required for vertex programs consists of:

- a bit indicating whether or not program mode is enabled;
- a bit indicating whether or not two-sided color mode is enabled;
- a bit indicating whether or not program-specified point size mode is enabled;
- 96 4-component floating-point program parameter registers;
- 16 4-component vertex attribute registers (though this state is aliased with the current normal, primary color, secondary color, fog coordinate, weights, and texture coordinate sets);
- 24 sets of matrix tracking state for each set of four sequential program parameter registers, consisting of a n-valued integer indicated the tracked matrix or GL_NONE (where n is 5 + the number of texture units supported + the number of tracking matrices supported) and a four-valued integer indicating the transformation of the tracked matrix;
- an unsigned integer naming the currently bound vertex program
- and the state must be maintained to indicate which integers are currently in use as program names.

Each existent program object consists of a target, a boolean indicating whether the program is resident, an array of type ubyte containing the program string, and the length of the program string array. Initially, no program objects exist.

Program mode, two-sided color mode, and program-specified point size mode are all initially disabled.

The initial state of all 96 program parameter registers is (0,0,0,0).

The initial state of the 16 vertex attribute registers is (0,0,0,1) except in cases where a vertex attribute register aliases to a conventional GL transform mode vertex parameter in which case

the initial state is the initial state of the respective aliased conventional vertex parameter.

The initial state of the 24 sets of matrix tracking state is NONE for the tracked matrix and IDENTITY_NV for the transformation of the tracked matrix.

The initial currently bound program is zero.

The client state required to implement the 16 vertex attribute arrays consists of 16 boolean values, 16 memory pointers, 16 integer stride values, 16 symbolic constants representing array types, and 16 integers representing values per element. Initially, the boolean values are each disabled, the memory pointers are each null, the strides are each zero, the array types are each FLOAT, and the integers representing values per element are each four."

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

-- Section 3.3 "Points"

Change the first paragraph to read:

"When program vertex mode is disabled, the point size for rasterizing points is controlled with

```
void PointSize(float size);
```

size specifies the width or diameter of a point. The initial point size value is 1.0. A value less than or equal to zero results in the error INVALID_VALUE. When vertex program mode is enabled, the point size for rasterizing points is determined as described in section 2.14.1.5."

-- Section 3.9 "Color Sum"

Change the first paragraph to read:

"At the beginning of color sum, a fragment has two RGBA colors: a primary color cpri (which texturing, if enabled, may have modified) and a secondary color csec. If vertex program mode is disabled, csec is defined by the lighting equations in section 2.13.1. If vertex program mode is enabled, csec is the fragment's secondary color, obtained by interpolating the COL1 (or BFC1 if the primitive is a polygon, the vertex program two-sided color mode is enabled, and the polygon is back-facing) vertex result register RGB components for the vertices making up the primitive; the alpha component of csec when program mode is enabled is always zero. The components of these two colors are summed to produce a single post-texturing RGBA color c. The components of c are then clamped to the range [0,1]."

-- Section 3.10 "Fog"

Change the initial sentences in the second paragraph to read:

"This factor f may be computed according to one of three equations:

```
f = exp(-d*c)                                   (3.24)
f = exp(-(d*c)^2)                               (3.25)
f = (e-c)/(e-s)                                 (3.26)
```

If vertex program mode is enabled, then c is the fragment's fog coordinate, obtained by interpolating the FOGC vertex result register values for the vertices making up the primitive. When vertex program

mode is disabled, the c is the eye-coordinate distance from the eye, (0,0,0,1) in eye-coordinates, to the fragment center." ...

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

-- Section 5.1 "Evaluators"

Add the following lines to the end of table 5.1 (page 165):

target	k	values
MAP1_VERTEX_ATTRIB0_4_NV	4	x, y, z, w vertex attribute 0
MAP1_VERTEX_ATTRIB1_4_NV	4	x, y, z, w vertex attribute 1
MAP1_VERTEX_ATTRIB2_4_NV	4	x, y, z, w vertex attribute 2
MAP1_VERTEX_ATTRIB3_4_NV	4	x, y, z, w vertex attribute 3
MAP1_VERTEX_ATTRIB4_4_NV	4	x, y, z, w vertex attribute 4
MAP1_VERTEX_ATTRIB5_4_NV	4	x, y, z, w vertex attribute 5
MAP1_VERTEX_ATTRIB6_4_NV	4	x, y, z, w vertex attribute 6
MAP1_VERTEX_ATTRIB7_4_NV	4	x, y, z, w vertex attribute 7
MAP1_VERTEX_ATTRIB8_4_NV	4	x, y, z, w vertex attribute 8
MAP1_VERTEX_ATTRIB9_4_NV	4	x, y, z, w vertex attribute 9
MAP1_VERTEX_ATTRIB10_4_NV	4	x, y, z, w vertex attribute 10
MAP1_VERTEX_ATTRIB11_4_NV	4	x, y, z, w vertex attribute 11
MAP1_VERTEX_ATTRIB12_4_NV	4	x, y, z, w vertex attribute 12
MAP1_VERTEX_ATTRIB13_4_NV	4	x, y, z, w vertex attribute 13
MAP1_VERTEX_ATTRIB14_4_NV	4	x, y, z, w vertex attribute 14
MAP1_VERTEX_ATTRIB15_4_NV	4	x, y, z, w vertex attribute 15

Replace the four paragraphs on pages 167-168 that explain the operation of EvalCoord:

"EvalCoord operates differently depending on whether vertex program mode is enabled or not. We first discuss how EvalCoord operates when vertex program mode is disabled.

When one of the EvalCoord commands is issued and vertex program mode is disabled, all currently enabled maps (excluding the maps that correspond to vertex attributes, i.e. maps of the form MAPx_VERTEX_ATTRIBn_4_NV). ..."

Add a paragraph before the initial paragraph discussing AUTO_NORMAL:

"When one of the EvalCoord commands is issued and vertex program mode is enabled, the evaluation and the issuing of per-vertex parameter commands matches the discussion above, except that if any vertex attribute maps are enabled, the corresponding VertexAttribNV call for each enabled vertex attribute map is issued with the map's evaluated coordinates and the corresponding aliased per-vertex parameter map is ignored if it is also enabled, with one important difference. As is the case when vertex program mode is disabled, the GL uses evaluated values instead of current values for those evaluations that are enabled (otherwise the current values are used). The order of the effective commands is immaterial, except that Vertex or VertexAttribNV(0, ...) (the commands that issue provoke vertex program execution) must be issued last. Use of evaluators has no effect on the current vertex attributes or conventional per-vertex parameters. If a vertex attribute map is disabled, but its corresponding conventional per-vertex parameter map is enabled, the conventional per-vertex

parameter map is evaluated and issued as when vertex program mode is not enabled."

Replace the two paragraphs discussing AUTO_NORMAL with:

"Finally, if either MAP2_VERTEX_3 or MAP2_VERTEX_4 is enabled or if both MAP2_VERTEX_ATTRIB0_4_NV and vertex program mode are enabled, then the normal to the surface is computed. Analytic computation, which sometimes yields normals of length zero, is one method which may be used. If automatic normal generation is enabled, then this computed normal is used as the normal associated with a generated vertex (when program mode is disabled) or as vertex attribute 2 (when vertex program mode is enabled). Automatic normal generation is controlled with Enable and Disable with the symbolic constant AUTO_NORMAL. If automatic normal generation is disabled and vertex program mode is enabled, then vertex attribute 2 is evaluated as usual. If automatic normal generation and vertex program mode are disabled, then a corresponding normal map, if enabled, is used to produce a normal. If neither automatic normal generation nor a map corresponding to the normal per-vertex parameter (or vertex attribute 2 in program mode) are enabled, then no normal is sent with a vertex resulting from an evaluation (the effect is that the current normal is used). For MAP_VERTEX3, let q=p. For MAP_VERTEX_4 or MAP2_VERTEX_ATTRBI0_4_NV, let q = (x/w, y/w, z/w) where (x,y,z,w)=p. Then let

m = (partial q / partial u) cross (partial q / partial v)

Then when vertex program mode is disabled, the generated analytic normal, n, is given by n=m/||m||. However, when vertex program mode is enabled, the generated analytic normal used for vertex attribute 2 is simply (mx,my,mz,1). In vertex program mode, the normalization of the generated analytic normal can be performed by the current vertex program."

Change the respective sentences of the last paragraph discussing required evaluator state to read:

"The state required for evaluators potentially consists of 9 conventional one-dimensional map specifications, 16 vertex attribute one-dimensional map specifications, 9 conventional two-dimensional map specifications, and 16 vertex attribute two-dimensional map specifications indicating which are enabled. ... All vertex coordinate maps produce the coordinates (0,0,0,1) (or the appropriate subset); all normal coordinate maps produce (0,0,1); RGBA maps produce (1,1,1,1); color index maps produce 1.0; texture coordinate maps produce (0,0,0,1); and vertex attribute maps produce (0,0,0,1). ... If any evaluation command is issued when none of MAPn_VERTEX_3, MAPn_VERTEX_4, or MAPn_VERTEX_ATTRIB0_NV (where n is the map dimension being evaluated) are enabled, nothing happens."

-- Section 5.4 "Display Lists"

Add to the list of commands not compiled into display lists in the third to the last paragraph:

"AreProgramsResidentNV, IsProgramNV, GenProgramsNV, DeleteProgramsNV, VertexAttribPointerNV"

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

-- Section 6.1.12 "Saving and Restoring State"

Only the enables and vertex array state introduced by this extension can be pushed and popped.

See the attribute column in table X.5 for determining what vertex program state can be pushed and popped with PushAttrib, PopAttrib, PushClientAttrib, and PopClientAttrib.

The new evaluator enables in table 6.22 can also be pushed and popped.

-- NEW Section 6.1.13 "Vertex Program Queries"

"The commands

```
  void GetProgramParameterfvNV(enum target, uint index,
                               enum pname, float *params);
  void GetProgramParameterdvNV(enum target, uint index,
                               enum pname, double *params);
```

obtain the current program parameters for the given program target and parameter index into the array params. target must be VERTEX_PROGRAM_NV. pname must be PROGRAM_PARAMETER_NV. The INVALID_VALUE error is generated if index is greater than 95. Each program parameter is an array of four values.

The command

```
  void GetProgramivNV(uint id, enum pname, int *params);
```

obtains program state named by pname for the program named id in the array params. pname must be one of PROGRAM_TARGET_NV, PROGRAM_LENGTH_NV, or PROGRAM_RESIDENT_NV. The INVALID_OPERATION error is generated if the program named id does not exist.

The command

```
  void GetProgramStringNV(uint id, enum pname,
                          ubyte *program);
```

obtains the program string for program id. pname must be PROGRAM_STRING_NV. n ubytes are returned into the array program where n is the length of the program in ubytes. GetProgramivNV with PROGRAM_LENGTH_NV can be used to query the length of a program's string. The INVALID_OPERATION error is generated if the program named id does not exist.

The command

```
  void GetTrackMatrixivNV(enum target, uint address,
                          enum pname, int *params);
```

obtains the matrix tracking state named by pname for the specified address in the array params. target must be VERTEX_PROGRAM_NV. pname must be either TRACK_MATRIX_NV or TRACK_MATRIX_TRANSFORM_NV. If the matrix tracked is a texture matrix, TEXTUREi_ARB is returned (never TEXTURE) where i indicates the texture unit of the particular tracked texture matrix. The INVALID_VALUE error is generated if address is not divisible by four and is not less than 96.

The commands

```
  void GetVertexAttribdvNV(uint index, enum pname, double *params);
```

```
  void GetVertexAttribfvNV(uint index, enum pname, float *params);
  void GetVertexAttribivNV(uint index, enum pname, int *params);
```

obtain the vertex attribute state named by pname for the vertex attribute numbered index. pname must be one of ATTRIB_ARRAY_SIZE_NV, ATTRIB_ARRAY_STRIDE_NV, ATTRIB_ARRAY_TYPE_NV, or CURRENT_ATTRIB_NV. Note that all the queries except CURRENT_ATTRIB_NV return client state. The INVALID_VALUE error is generated if index is greater than 15, or if index is zero and pname is CURRENT_ATTRIB_NV.

The command

```
  void GetVertexAttribPointervNV(uint index,
                                 enum pname, void **pointer);
```

obtains the pointer named pname in the array params for vertex attribute numbered index. pname must be ATTRIB_ARRAY_POINTER_NV. The INVALID_VALUE error is generated if index greater than 15.

The command

```
  boolean IsProgramNV(uint id);
```

returns TRUE if program is the name of a program object. If program is zero or is a non-zero value that is not the name of a program object, or if an error condition occurs, IsProgramNV returns FALSE. A name returned by GenProgramsNV but not yet loaded with a program is not the name of a program object."

-- NEW Section 6.1.14 "Querying Current Matrix State"

"Instead of providing distinct symbolic tokens for querying each matrix and matrix stack depth, the symbolic tokens CURRENT_MATRIX_NV and CURRENT_MATRIX_STACK_DEPTH_NV in conjunction with the GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev return the respective state of the current matrix given the current matrix mode.

Querying CURRENT_MATRIX_NV and CURRENT_MATRIX_STACK_DEPTH_NV is the only means for querying the matrix and matrix stack depth of the tracking matrices described in section 2.14.5."

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

Add the following rule:

"Rule X Vertex program and vertex state program instructions not relevant to the calculation of any result must have no effect on that result.

Rules X+1 Vertex program and vertex state program instructions relevant to the calculation of any result must always produce the identical result. In particular, the same instruction with the same source inputs must produce the identical result whether executed by a vertex program or a vertex state program.

Instructions relevant to the calculation of a result are any instructions in a sequence of instructions that eventually determine the source values for the calculation under consideration.

There is no guaranteed invariance between vertices transformed by conventional GL vertex transform mode and vertices transformed by vertex program mode. Multi-pass rendering algorithms that require rendering invariances to operate correctly should not mix conventional

GL vertex transform mode with vertex program mode for different rendering passes. However such algorithms will operate correctly if the algorithms limit themselves to a single mode of vertex transformation."

Additions to the AGL/GLX/WGL Specifications

Program objects are shared between AGL/GLX/WGL rendering contexts if and only if the rendering contexts share display lists. No change is made to the AGL/GLX/WGL API.

Dependencies on EXT_vertex_weighting

If the EXT_vertex_weighting extension is not supported, there is no aliasing between vertex attribute 1 and the current vertex weight. Replace the contents of the last three columns in row 5 of table X.2 with dashes.

Dependencies on EXT_point_parameters

When EXT_point_parameters is supported, the amended discussion of point size determination should be further amended with the language from the EXT_point_parameters specification though the point parameters functionality only applies when vertex program mode is disabled.

Even if the EXT_point_parameters extension is not supported, the PSIZ vertex result register must operate as specified.

Dependencies on ARB_multitexture

ARB_multitexture is required to support NV_vertex_program and the value of MAX_TEXTURE_UNITS_ARB must be at least 2. If more than 8 texture units are supported, only the first 8 texture units can be assigned texture coordinates when vertex program mode is enabled. Texture units beyond 8 are implicitly disabled when vertex program mode is enabled.

Dependencies on EXT_fog_coord

If the EXT_fog_coord extension is not supported, there is no aliasing between vertex attribute 5 and the current fog coordinate. Replace the contents of the last three columns in row 5 of table X.2 with dashes.

Even if the EXT_fog_coord extension is not supported, the FOGC vertex result register must operate as specified. Note that the FOGC vertex result register behaves identically to the EXT_fog_coord extension's FOG_COORDINATE_SOURCE_EXT being FOG_COORDINATE_EXT. This means that the functionality of EXT_fog_coord is required to implement NV_vertex_program even if the EXT_fog_coord extension is not supported.

If the EXT_fog_coord extension is supported, the state of FOG_COORDINATE_SOURCE_EXT only applies when vertex program mode is disabled and the discussion in section 3.10 is further amended by the discussion of FOG_COORDINATE_SOURCE_EXT in the EXT_fog_coord specification.

Dependencies on EXT_secondary_color

If the EXT_secondary_color extension is not supported, there is no aliasing between vertex attribute 4 and the current secondary color.

Replace the contents of the last three columns in row 4 of table X.2 with dashes.

Even if the EXT_secondary_color extension is not supported, the COL1 and BFC1 vertex result registers must operate as specified. These vertex result registers are required to implement OpenGL 1.2's separate specular mode within a vertex program.

GLX Protocol

Forty-five new GL commands are added.

The following thirty-five rendering commands are sent to the sever as part of a glXRender request:

BindProgramNV

2	12		rendering command length
2	4180		rendering command opcode
4	ENUM		target
4	CARD32		id

ExecuteProgramNV

2	12+4*n		rendering command length
2	4181		rendering command opcode
4	ENUM		target
	0x8621	n=4	GL_VERTEX_STATE_PROGRAM_NV
	else	n=0	command is erroneous
4	CARD32		id
4*n	LISTofFLOAT32		params

RequestResidentProgramsNV

2	8+4*n	rendering command length
2	4182	rendering command opcode
4	INT32	n
n*4	CARD32	programs

LoadProgramNV

2	16+n+p	rendering command length
2	4183	rendering command opcode
4	ENUM	target
4	CARD32	id
4	INT32	len
n	LISTofCARD8	n
p		unused, p=pad(n)

ProgramParameter4fvNV

2	32	rendering command length
2	4184	rendering command opcode
4	ENUM	target
4	CARD32	index
4	FLOAT32	params[0]
4	FLOAT32	params[1]
4	FLOAT32	params[2]
4	FLOAT32	params[3]

ProgramParameter4dvNV

2	44	rendering command length
2	4185	rendering command opcode
4	ENUM	target
4	CARD32	index
8	FLOAT64	params[0]
8	FLOAT64	params[1]
8	FLOAT64	params[2]

```
    8           FLOAT64         params[3]

ProgramParameters4fvNV
    2           16+16*n         rendering command length
    2           4186            rendering command opcode
    4           ENUM            target
    4           CARD32          index
    4           CARD32          n
    16*n        FLOAT32         params

ProgramParameters4dvNV
    2           16+32*n         rendering command length
    2           4187            rendering command opcode
    4           ENUM            target
    4           CARD32          index
    4           CARD32          n
    32*n        FLOAT64         params

TrackMatrixNV
    2           20              rendering command length
    2           4188            rendering command opcode
    4           ENUM            target
    4           CARD32          address
    4           ENUM            matrix
    4           ENUM            transform

VertexAttribPointerNV is an entirely client-side command

VertexAttrib1svNV
    2           12              rendering command length
    2           4189            rendering command opcode
    4           CARD32          index
    2           INT16           v[0]
    2                           unused

VertexAttrib2svNV
    2           12              rendering command length
    2           4190            rendering command opcode
    4           CARD32          index
    2           INT16           v[0]
    2           INT16           v[1]

VertexAttrib3svNV
    2           12              rendering command length
    2           4191            rendering command opcode
    4           CARD32          index
    2           INT16           v[0]
    2           INT16           v[1]
    2           INT16           v[2]
    2                           unused

VertexAttrib4svNV
    2           12              rendering command length
    2           4192            rendering command opcode
    4           CARD32          index
    2           INT16           v[0]
    2           INT16           v[1]
    2           INT16           v[2]
    2           INT16           v[3]

VertexAttrib1fvNV
    2           12              rendering command length
    2           4193            rendering command opcode
```

```
    4           CARD32          index
    4           FLOAT32         v[0]

VertexAttrib2fvNV
    2           16              rendering command length
    2           4194            rendering command opcode
    4           CARD32          index
    4           FLOAT32         v[0]
    4           FLOAT32         v[1]

VertexAttrib3fvNV
    2           20              rendering command length
    2           4195            rendering command opcode
    4           CARD32          index
    4           FLOAT32         v[0]
    4           FLOAT32         v[1]
    4           FLOAT32         v[2]

VertexAttrib4fvNV
    2           24              rendering command length
    2           4196            rendering command opcode
    4           CARD32          index
    4           FLOAT32         v[0]
    4           FLOAT32         v[1]
    4           FLOAT32         v[2]
    4           FLOAT32         v[3]

VertexAttrib1dvNV
    2           16              rendering command length
    2           4197            rendering command opcode
    4           CARD32          index
    8           FLOAT64         v[0]

VertexAttrib2dvNV
    2           24              rendering command length
    2           4198            rendering command opcode
    4           CARD32          index
    8           FLOAT64         v[0]
    8           FLOAT64         v[1]

VertexAttrib3dvNV
    2           32              rendering command length
    2           4199            rendering command opcode
    4           CARD32          index
    8           FLOAT64         v[0]
    8           FLOAT64         v[1]
    8           FLOAT64         v[2]

VertexAttrib4dvNV
    2           40              rendering command length
    2           4200            rendering command opcode
    4           CARD32          index
    8           FLOAT64         v[0]
    8           FLOAT64         v[1]
    8           FLOAT64         v[2]
    8           FLOAT64         v[3]

VertexAttrib4ubvNV
    2           12              rendering command length
    2           4201            rendering command opcode
    4           CARD32          index
    1           CARD8           v[0]
    1           CARD8           v[1]
```

```
    1           CARD8           v[2]
    1           CARD8           v[3]

VertexAttribs1svNV
    2           12+2*n+p        rendering command length
    2           4202            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    2*n         INT16           v
    p                           unused, p=pad(2*n)

VertexAttribs2svNV
    2           12+4*n          rendering command length
    2           4203            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    4*n         INT16           v

VertexAttribs3svNV
    2           12+6*n+p        rendering command length
    2           4204            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    6*n         INT16           v
    p                           unused, p=pad(6*n)

VertexAttribs4svNV
    2           12+8*n          rendering command length
    2           4205            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    8*n         INT16           v

VertexAttribs1fvNV
    2           12+4*n          rendering command length
    2           4206            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    4*n         FLOAT32         v

VertexAttribs2fvNV
    2           12+8*n          rendering command length
    2           4207            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    8*n         FLOAT32         v

VertexAttribs3fvNV
    2           12+12*n         rendering command length
    2           4208            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    12*n        FLOAT32         v

VertexAttribs4fvNV
    2           12+16*n         rendering command length
    2           4209            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    16*n        FLOAT32         v

VertexAttribs1dvNV
    2           12+8*n          rendering command length
```

```
        2           4210            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        8*n         FLOAT64         v

    VertexAttribs2dvNV
        2           12+16*n         rendering command length
        2           4211            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        16*n        FLOAT64         v

    VertexAttribs3dvNV
        2           12+24*n         rendering command length
        2           4212            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        24*n        FLOAT64         v

    VertexAttribs4dvNV
        2           12+32*n         rendering command length
        2           4213            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        32*n        FLOAT64         v

    VertexAttribs4ubvNV
        2           12+4*n          rendering command length
        2           4214            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        4*n         CARD8           v
```

The remaining twelve commands are non-rendering commands. These commands are sent separately (i.e., not as part of a glXRender or glXRenderLarge request), using the glXVendorPrivateWithReply request:

```
    AreProgramsResidentNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           4+n             request length
        4           1293            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           INT32           n
        n*4         LISTofCARD32    programs
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           (n+p)/4         reply length
        4           BOOL32          return value
        20                          unused
        n           LISTofBOOL      programs
        p                           unused, p=pad(n)

    DeleteProgramsNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           4+n             request length
        4           1294            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           INT32           n
        n*4         LISTofCARD32    programs
```

```
GenProgramsNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           4               request length
    4           1295            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           n
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           n               reply length
    24                          unused
    n*4         LISTofCARD322   programs

GetProgramParameterfvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1296            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           FLOAT32         params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofFLOAT32   params

GetProgramParameterdvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1297            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n*2)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    8           FLOAT64         params
```

```
    8                           unused

    otherwise this follows:

    16                          unused
    n*8         LISTofFLOAT64   params

GetProgramivNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1298            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           CARD32          id
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params

GetProgramStringNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1299            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           CARD32          id
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           (n+p)/4         reply length
    4                           unused
    4           CARD32          n
    16                          unused
    n           STRING          program
    p                           unused, p=pad(n)

GetTrackMatrixivNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1300            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          address
    4           ENUM            pname
  =>
    1           1               reply
```

```
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params

Note that ATTRIB_ARRAY_SIZE_NV, ATTRIB_ARRAY_STRIDE_NV, and
ATTRIB_ARRAY_TYPE_NV may be queried by GetVertexAttribNV but
return client-side state.

GetVertexAttribdvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1301            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n*2)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    8           FLOAT64         params
    8                           unused

    otherwise this follows:

    16                          unused
    n*8         LISTofFLOAT64   params

GetVertexAttribfvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1302            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:
```

```
            4           FLOAT32         params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofFLOAT32   params

        GetVertexAttribivNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           1303            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            4           INT32           params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofINT32     params

        GetVertexAttribPointervNV is an entirely client-side command

        IsProgramNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           1304            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           0               reply length
            4           BOOL32          return value
            20                          unused
```

Errors

The error INVALID_VALUE is generated if VertexAttribNV is called where index is greater than 15.

The error INVALID_VALUE is generated if any ProgramParameterNV has an index is greater than 95.

The error INVALID_VALUE is generated if VertexAttribPointerNV is called where index is greater than 15.

The error INVALID_VALUE is generated if VertexAttribPointerNV
is called where size is not one of 1, 2, 3, or 4.

The error INVALID_VALUE is generated if VertexAttribPointerNV
is called where stride is negative.

The error INVALID_OPERATION is generated if VertexAttribPointerNV
is called where type is UNSIGNED_BYTE and size is not 4.

The error INVALID_VALUE is generated if LoadProgramNV is used to load a
program with an id of zero.

The error INVALID_OPERATION is generated if LoadProgramNV is used
to load an id that is currently loaded with a program of a different
program target.

The error INVALID_OPERATION is generated if the program passed to
LoadProgramNV fails to load because it is not syntactically correct
based on the specified target. The value of PROGRAM_ERROR_POSITION_NV
is still updated when this error is generated.

The error INVALID_OPERATION is generated if LoadProgramNV has a
target of VERTEX_PROGRAM_NV and the specified program fails to
load because it does not write the HPOS register at least once.
The value of PROGRAM_ERROR_POSITION_NV is still updated when this
error is generated.

The error INVALID_OPERATION is generated if LoadProgramNV has a target
of VERTEX_STATE_PROGRAM_NV and the specified program fails to load
because it does not write at least one program parameter register.
The value of PROGRAM_ERROR_POSITION_NV is still updated when this
error is generated.

The error INVALID_OPERATION is generated if the vertex program
or vertex state program passed to LoadProgramNV fails to load
because it contains more than 128 instructions. The value of
PROGRAM_ERROR_POSITION_NV is still updated when this error is
generated.

The error INVALID_OPERATION is generated if a program is loaded with
LoadProgramNV for id when id is currently loaded with a program of
a different target.

The error INVALID_OPERATION is generated if BindProgramNV attempts
to bind to a program name that is not a vertex program (for example,
if the program is a vertex state program).

The error INVALID_VALUE is generated if GenProgramsNV is called
where n is negative.

The error INVALID_VALUE is generated if AreProgramsResidentNV is
called and any of the queried programs are zero or do not exist.

The error INVALID_OPERATION is generated if ExecuteProgramNV executes
a program that does not exist.

The error INVALID_OPERATION is generated if ExecuteProgramNV executes
a program that is not a vertex state program.

The error INVALID_OPERATION is generated if Begin, RasterPos, or a
command that performs an explicit Begin is called when vertex program
mode is enabled and the currently bound vertex program writes program
parameters that are currently being tracked.

The error INVALID_OPERATION is generated if ExecuteProgramNV is called and the vertex state program to execute writes program parameters that are currently being tracked.

The error INVALID_VALUE is generated if TrackMatrixNV has a target of VERTEX_PROGRAM_NV and attempts to track an address is not a multiple of four.

The error INVALID_VALUE is generated if GetProgramParameterNV is called to query an index greater than 95.

The error INVALID_VALUE is generated if GetVertexAttribNV is called to query an <index> greater than 15, or if <index> is zero and <pname> is CURRENT_ATTRIB_NV.

The error INVALID_VALUE is generated if GetVertexAttribPointervNV is called to query an index greater than 15.

The error INVALID_OPERATION is generated if GetProgramivNV is called and the program named id does not exist.

The error INVALID_OPERATION is generated if GetProgramStringNV is called and the program named <program> does not exist.

The error INVALID_VALUE is generated if GetTrackMatrixivNV is called with an <address> that is not divisible by four and not less than 96.

The error INVALID_VALUE is generated if AreProgramsResidentNV, DeleteProgramsNV, GenProgramsNV, or RequestResidentProgramsNV are called where <n> is negative.

The error INVALID_VALUE is generated if LoadProgramNV is called where <len> is negative.

The error INVALID_VALUE is generated if ProgramParameters4dvNV or ProgramParameters4fvNV are called where <count> is negative.

The error INVALID_VALUE is generated if VertexAttribs{1,2,3,4}{d,f,s}vNV is called where <count> is negative.

New State

update table 6.22 (page 212) so that all the "9"s are "25"s because there are 9 conventional map targets and 16 vertex attribute map targets making a total of 25.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
VERTEX_PROGRAM_NV	B	IsEnabled	False	vertex program enable	2.10	enable
VERTEX_PROGRAM_POINT_SIZE_NV	B	IsEnabled	False	program-specified point size mode	2.14.1.5	enable
VERTEX_PROGRAM_TWO_SIDE_NV	B	IsEnabled	False	two-sided color mode	2.14.1.5	enable
PROGRAM_ERROR_POSITION_NV	Z	GetIntegerv	-1	last program error position	2.14.1.7	-
PROGRAM_PARAMETER_NV	96xR4	GetProgramParameterNV	(0,0,0,0)	program parameters	2.14.1.2	-
CURRENT_ATTRIB_NV	16xR4	GetVertexAttribNV but zero cannot be queried, aliased with per-vertex parameters	see 2.14.6	vertex attributes	2.14.1.1	current
TRACK_MATRIX_NV	24xZ8+	GetTrackMatrixivNV	NONE	track matrix	2.14.5	-
TRACK_MATRIX_TRANSFORM_NV	24xZ8+	GetTrackMatrixivNV	IDENTITY_NV	track matrix transform	2.14.5	-
VERTEX_PROGRAM_BINDING_NV	Z+	GetIntegerv	0	bound vertex program	2.14.1.8	-
VERTEX_ATTRIB_ARRAYn_NV	16xB	IsEnabled	False	vertex attrib array enable	2.14.3	vertex-array
ATTRIB_ARRAY_SIZE_NV	16xZ	GetVertexAttribNV	4	vertex attrib array size	2.14.3	vertex-array

ATTRIB_ARRAY_STRIDE_NV	16xZ+	GetVertexAttribNV	0	vertex attrib array stride	2.14.3	vertex-array
ATTRIB_ARRAY_TYPE_NV	16xZ4	GetVertexAttribNV	FLOAT	vertex attrib array type	2.14.3	vertex-array

Table X.5. New State Introduced by NV_vertex_program.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
PROGRAM_TARGET_NV	Z2	GetProgramivNV	0	program target	6.1.13	-
PROGRAM_LENGTH_NV	Z+	GetProgramivNV	0	program length	6.1.13	-
PROGRAM_RESIDENT_NV	Z2	GetProgramivNV	False	program residency	6.1.13	-
PROGRAM_STRING_NV	ubxn	GetProgramStringNV	""	program string	6.1.13	-

Table X.6. Program Object State.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
-	12xR4	-	(0,0,0,0)	temporary registers	2.14.1.4	-
-	15xR4	-	(0,0,0,1)	vertex result registers	2.14.1.4	-
	Z4	-	(0,0,0,0)	vertex program address register	2.14.1.3	-

Table X.7. Vertex Program Per-vertex Execution State.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_MATRIX_STACK_DEPTH_NV	m*Z+	GetIntegerv	1	current stack depth	6.1.14	-
CURRENT_MATRIX_NV	m*n*xM^4	GetFloatv	Identity	current matrix	6.1.14	-

Table X.8. Current matrix state where m is the total number of matrices including texture matrices and tracking matrices and n is the number of matrices on each particular matrix stack. Note that this state is aliased with existing matrix state.

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_TRACK_MATRIX_STACK_DEPTH_NV	Z+	GetIntegerv	1	maximum tracking matrix stack depth	2.14.5	-
MAX_TRACK_MATRICES_NV	Z+	GetIntegerv	8 (not to exceed 32)	maximum number of tracking matrices	2.14.5	-

Table X.9. New Implementation-Dependent Values Introduced by NV_vertex_program.

Revision History

Version 1.1:

Added normalization example to Issues.

Fix explanation of EXP and ARL floor equivalence.

Clarify that vertex state programs fail if they load more than one vertex attribute (though only one is possible).

Version 1.2

Add GLX protocol for VertexAttrib4ubvNV and VertexAttribs4ubvNV

Add issue about TrackMatrixNV transform behavior with example

Fix the C code specifying VertexAttribsvNV

Version 1.3

Dropped support for INT typed vertex attrib arrays.

Clarify that when ArrayElement is executed and vertex program mode is enabled and the vertex attrib 0 array is enabled, the vertex attrib 0 array command is executed last. However when ArrayElement is executed and vertex program mode is enabled and the vertex attrib 0 array is disabled and the vertex array is enabled, the vertex array command is executed last.

Version 1.4

  Allow TEXTUREi_ARB for the track matrix. This allows matrix
  tracking of a particular texture matrix without reference to active
  texture (set by glActiveTextureARB) state.

  Early NVIDIA drivers (prior to October 5, 2001) have a bug
  in their handling of tracking matrices specified with TEXTURE.
  Rather than tracking the particular texture matrix indicated
  by the active texture state when TrackMatrixNV is called, these
  early drivers incorrectly track matrix the active texture's texture
  matrix _at track matrix validation time_. In practice this means,
  every tracked matrix defined with TEXTURE tracks the same matrix
  values; you cannot track distinct texture matrices at the same
  time and the texture matrix you actually track depends on the
  active texture matrix at validation time. This is a driver bug.

  Drivers after October 5, 2001 properly track the texture matrix
  specified by active texture when TrackMatrix is called.

  The new correct drivers can be distinguished from the old drivers
  at run time with the following code:

```
    while (glGetError() != GL_NO_ERROR);  // Clear any pre-existing OpenGL errors.
    glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 8, GL_TEXTURE0_ARB, GL_IDENTITY_NV);
    if (glGetError() != GL_NO_ERROR) {
      // Old buggy pre-version 1.4 drivers with GL_TEXTURE
      // glTrackMatrixNV bug.
    } else {
      // Correct new version 1.4 drivers (or later) with GL_TEXTURE
      // glTrackMatrixNV bug fixed and GL_TEXTUREi_NV support.

      // Note: you may want to untrack the matrix at this point.
    }
```

Version 1.5

  Earlier versions of this specification claimed for
  GetVertexAttribARB that it is an error to query any vertex attrib
  state for vertex attrib array zero. In fact, it should only be
  an error to query the CURRENT_ATTRIB_ARB state for vertex attrib
  zero; the size, stride, and type of vertex attrib array zero may
  be queried. Version 1.5 specifies the correct behavior.

  Early NVIDIA drivers (prior to January 11, 2002) did not implement
  generate error when querying vertex attrib array zero state (ie,
  did the right thing for size, stride, and type) but not create an
  error when querying the current attribute values for vertex attrib
  array zero either.

Version 1.6

  GLX opcodes and vendorpriv values assigned.

Chapter 3

ATI

Jason L. Mitchell

Pixel Shading with DirectX 8.1 and the ATI RADEON™ 8500

Jason L. Mitchell
JasonM@ati.com
3D Application Research Group
ATI Research

Introduction

Programmable shaders are a powerful way to describe the interaction of surfaces with light, as evidenced by the success of programmable shading models like RenderMan and others. As graphics hardware evolves beyond the traditional "fixed function" pipeline, hardware designers are looking to programmable models to empower the next generation of real-time content. To allow content to interface with current programmable pixel shading hardware, we have designed the 1.4 pixel shader model (ps.1.4) exposed in DirectX 8.1 and supported by the ATI RADEON™ 8500. In these notes, we will outline the structure of the programming model and present some illustrative examples. In the companion notes distributed at SIGGRAPH, we will show implementations of the common example shaders used throughout this course (bumped cubic environment mapping, McCool BRDF and parameterized volumetric wood) as well as a new programming model which goes beyond ps.1.4. Soft copies of these notes and the supplemental material distributed at SIGGRAPH 2002 are available at http://www.ati.com/developer.

The ps.1.4 Programming Model

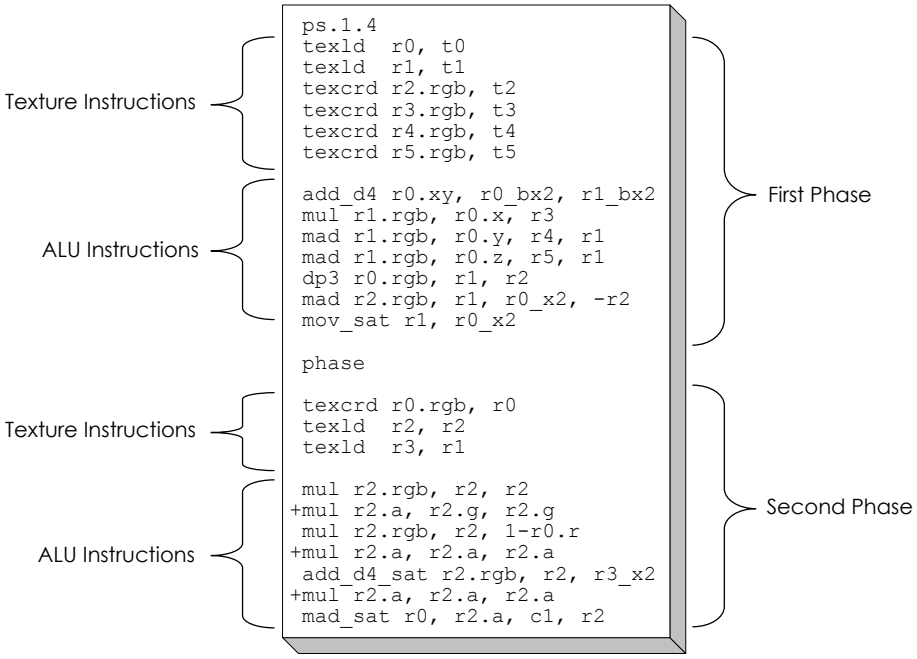
The 1.4 pixel shader programming model (ps.1.4), introduced in DirectX 8.1 in late 2001, advances the previously available programming model by applying a RISC approach. That is, the same micro operations which can be applied to colors can also be applied to texture addresses. This allows a wider variety of pixel shading affects to be achieved, as well as backward compatibility with previously available CISC models.

Inputs and Outputs

The pixel shader may take as inputs the data from interpolated texture coordinates, samples from texture maps, constant colors, the diffuse interpolator or the specular interpolator. There are six sets of texture coordinates (`t0`-`t5`), which may be used as extra interpolated data or as texture coordinates for sampling texture maps. There are six texture maps available in the ps.1.4 model and eight read-only constant registers (`c0`-`c7`). The low-precision diffuse (`v0`) and specular (`v1`) interpolators may also be used as arguments to ALU operations. There are six read-write temp registers (`r0`-`r5`) available in the ps.1.4 model. The contents of the `r0` temp register are considered the RGBA output of the pixel shader.

Shader Structure

A ps.1.4 shader may contain one or two *phases*, each of which begins with up to 6 texture instructions and ends with up to 8 ALU instructions. Each of the ALU instructions may be *co-issued*.




```
ps.1.4
texld  r0, t0
texld  r1, t1
texcrd r2.rgb, t2
texcrd r3.rgb, t3
texcrd r4.rgb, t4
texcrd r5.rgb, t5

add_d4 r0.xy, r0_bx2, r1_bx2
mul r1.rgb, r0.x, r3
mad r1.rgb, r0.y, r4, r1
mad r1.rgb, r0.z, r5, r1
dp3 r0.rgb, r1, r2
mad r2.rgb, r1, r0_x2, -r2
mov_sat r1, r0_x2

phase

texcrd r0.rgb, r0
texld  r2, r2
texld  r3, r1

 mul r2.rgb, r2, r2
+mul r2.a, r2.g, r2.g
 mul r2.rgb, r2, 1-r0.r
+mul r2.a, r2.a, r2.a
 add_d4_sat r2.rgb, r2, r3_x2
+mul r2.a, r2.a, r2.a
 mad_sat r0, r2.a, c1, r2
```

The shader shown above has two phases. The first phase uses six texture instructions (the maximum) and 7 ALU instructions. The second phase uses three texture instructions (two of which are *dependent reads*) and 4 ALU instructions (the first three of which are co-issued). The *phase* instruction marks the boundary between the phases.

ALU Instructions

The instruction set available for ALU operations is a fairly traditional set of arithmetic operations and comparators as listed below.

`add d, s0, s1`	`s0 + s1`
`sub d, s0, s1`	`s0 - s1`
`mul d, s0, s1`	`s0 * s1`
`mad d, s0, s1, s2`	`s0 * s1 + s2`
`lrp d, s0, s1, s2`	`s2 + s0*(s1-s2)`
`mov d, s0`	`d = s0`
`cnd d, s0, s1, s2`	`d = (s2 > 0.5) ? s0 : s1`
`cmp d, s0, s1, s2`	`d = (s2 >= 0) ? s0 : s1`
`dp3 d, s0, s1`	`s0·s1 replicated to d.rgba`
`dp4 d, s0, s1`	`s0·s1 replicated to d.rgba`
`bem d, s0, s1, s2`	`Macro for EMBM`

The inputs to the ALU instructions may be any of the temporary registers (`r0-r5`) or constant registers (`c0-c7`). The diffuse interpolator (`v0`) and specular interpolator (`v1`) may be inputs to ALU instructions in the second phase of the shader.

Argument Modifiers

As shown in the sample shader on the previous page, arguments to ALU instructions may have modifications made to them prior to the operation of the ALU instruction. There are five argument modifiers which can be used to perform operations such as negation, inversion, scaling and conversion from the [0..1] range to the [-1..1] range.

rn`_bias`	Bias
`1 - `*rn*	Invert
`-`*rn*	Negate
rn`_x2`	Scale by 2
rn`_bx2`	Signed Scaling

Source Register Selectors

It is often useful to think of the individual components of an RGBA vector as independent scalars. With source register selectors, it is possible to extract these scalars from an argument register and replicate them across all channels of the argument. The four source register selectors are shown below.

`.r`	Replicate Red
`.g`	Replicate Green
`.b`	Replicate Blue
`.a`	Replicate Alpha

Arbitrary Write Masks

It is often desirable to write to only a subset of the channels of a destination register. In ps.1.4, destination write masks can be used in any combination as long as the masks are ordered r, g, b, a. This allows the shader to execute a sequence of ALU operations which write to different components of the same destination register. This is especially useful when computing texture coordinates to be used in dependent texture reads, as we will illustrate later.

Instruction Modifiers

In some cases, we wish to modify the result of an ALU instruction as it is written into the destination register. In the ps.1.4 model, we can use instruction modifiers to perform shifts and saturates on the results of ALU operations. There are six shift (multiplier or divider, depending on the direction of the shift) operations that we can perform. Additionally, ALU results may be explicitly saturated to the [0..1] range. Saturation and shifting may be performed on the same ALU instruction.

instr`_x2`	Multiply by 2
instr`_x4`	Multiply by 4
instr`_x8`	Multiply by 8
instr`_d2`	Divide by 2
instr`_d4`	Divide by 4
instr`_d8`	Divide by 8
instr`_sat`	Saturate (clamp from 0 and 1)

Co-Issue

Pairing or *co-issuing* of ps.1.4 instructions is indicated by a plus sign (+) preceding the second instruction of the pair. The first instruction of the pair is a vector instruction which may write to any or all of r, g and b of the destination register. The second instruction of the pair is a scalar which writes into the alpha channel of the destination register. As an example, consider the following instructions:

```
 mul r0.rgb, t0, v0  // Component-wise multiply of the colors
+add r1.a,   r1, c2  // Add an alpha component at the same time
```

The dot product instructions may not be executed in the alpha pipeline, as they are always vector instructions.

Texture Instructions

The two most common texture instructions are the `texcrd` and `texld` instructions. The `texcrd` instruction is used to specify that a given temporary register (`r0`-`r5`) is to contain interpolated data. The `texld` instruction uses the specified texture coordinates to sample data from a texture map into the destination register. For example, the following `texcrd` instruction causes `r0` to contain interpolated data from the 0^{th} set of texture coordinates:

```
texcrd r0.rgb, t0
```

The following `texld` instruction causes `r1` to contain sampled data from the 1^{st} texture using the 1^{st} set of texture coordinates:

```
texld  r1, t1
```

The following `texld` instruction causes `r2` to be loaded with sampled data from the 2^{nd} texture using the contents of `r3` as texture coordinates:

```
texld  r2, r3
```

Using the contents of a temporary register as texture coordinates (the second argument of a `texld` instruction) is the definition of a dependent read because these texture coordinates *depend* upon the earlier ALU ops used to compute them (in this case `r3`). Naturally, a dependent read can only be used at the top of the second phase.

The `texkill` instruction can be used to kill pixels based upon results computed in a pixel shader. This is similar to alpha-testing, but more general in that multiple conditions may be tested with the `texkill` instruction. Multiple texkill instructions may appear in a single shader.

The final texture instruction is the `texdepth` instruction, which causes the current pixel's z to be replaced with the contents of a given register component. This instruction can be used to implement z-sprites, z-correct bump mapping and other effects. Naturally, only one `texdepth` instruction may be present in a given pixel shader.

Texture Projection

Any `texld` instruction may be modified to express a projected texture access. This includes projective *dependent* reads, which are fundamental to doing reflection and refraction mapping of things like water surfaces. Syntax looks like this:

```
texld r3, r3_dz  or
texld r3, r3_dw
```

Projective loads are useful for projective textures like refraction maps or for doing a divide, as we will show later in the skin shader [Vlachos02].

Example ps.1.4 Shaders

Now that we have introduced the structure and syntax of 1.4 pixel shaders, we will illustrate their usage in a variety of practical applications.

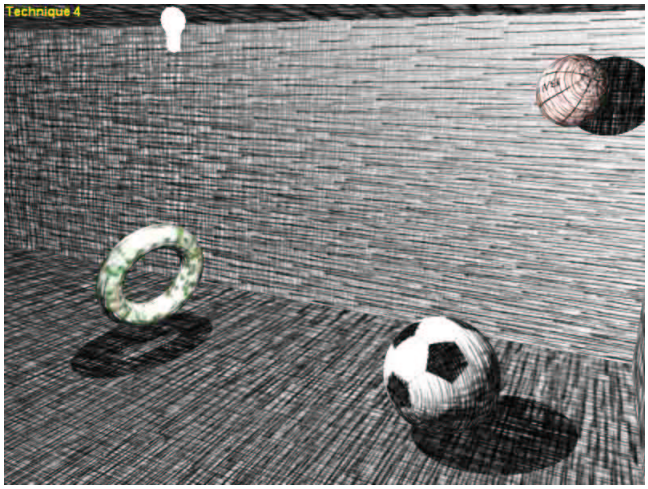
Real-Time Hatching

The first application of the 1.4 pixel shading model that we will illustrate is the Real-Time Hatching technique shown at SIGGRAPH last year [Praun01]. The general goal of this pixel shader is to compute the linear combination of 6 channels of a Tonal Art Map (TAM). The coefficients defining this linear combination have been computed in the vertex shader as a function of N·L with respect to a given light source and are stored in the r, g and b components of the 1st and 2nd texture coordinates.

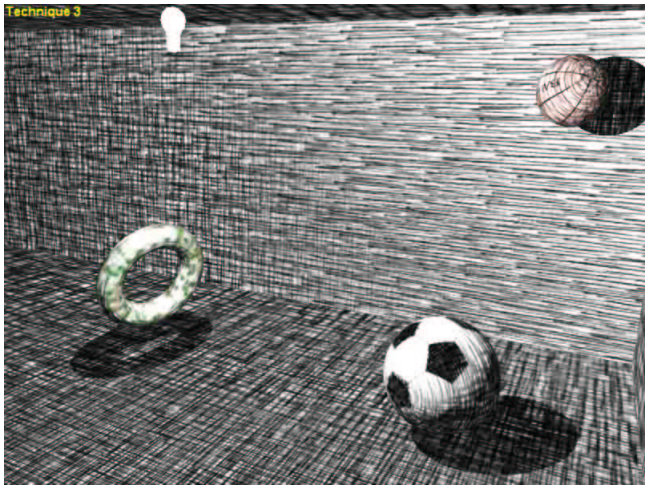
```
ps.1.4
texld  r0, t0          ; sample the first three channels of the TAM
texld  r1, t0          ; sample the second three channels of the TAM
texcrd r2.rgb, t1.xyz ; get the 123 TAM weights and place in register 2
texcrd r3.rgb, t2.xyz ; get the 456 TAM weights and place in register 3
dp3_sat r0, 1-r0, r2  ; dot the reg0 (TAM values) with reg2 (TAM weights)
dp3_sat r1, 1-r1, r3  ; dot the reg1 (TAM values) with reg3 (TAM weights)
add_sat r0, r0, r1    ; add reg 0 and reg1
mov_sat r0, 1-r0      ; complement and saturate
```

Real-Time Hatching with Per-Vertex TAM weights

One side effect of this approach is inaccurate lighting due to the fact that the TAM weights are computed at the vertices and interpolated. This can cause artifacts when the light source is close to a large polygon. The two-polygon wall in the image on the left side of the figure below seems to have its hatches grayed out as it transitions from the top right corner of near white, to the other corners which are near black. The wall in the image on the right shows the effect of per-pixel TAM weights, correctly transitioning between the intermediate hatching levels across the polygon.

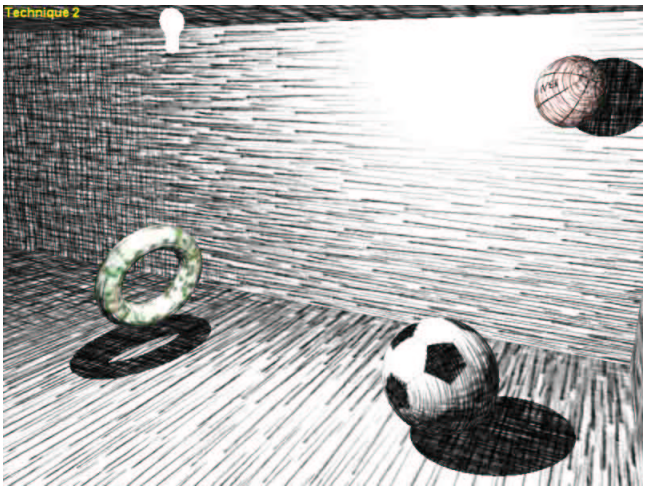


Per-Vertex TAM Weights



Per-Pixel TAM Weights

Another dramatic improvement that can be made to the hatching shader is inclusion of a per-pixel distance attenuation term as shown below.



Per-Pixel Distance Attenuation and TAM weight computation

The improved shader interpolates N·L, modulates it with per-pixel distance attenuation and uses this scalar as a texture coordinate to look up the per-pixel TAM weights. The two 1D RGB function textures used to look up the TAM weights based on N·L are shown here:

Two 1D RGB textures used to determine Per-Pixel TAM Weights from N · L

After computing the 6-term linear combination of hatching patterns in the TAM as before, the color is tinted to match a base texture map color.

```
ps.1.4
def c0, 1.00f, 1.00f, 1.00f, 1.00f
def c1, 0.30f, 0.59f, 0.11f, 0.00f  ; RGB to luminance conversion weights

texcrd r1.rgb, t2     ; N·L
texld  r4, t3         ; Intensity map looked up from light space position
texld  r5, t0         ; Base Texture

mul_x2 r4, r4.r, r1.r ; N·L * attenuation
add    r4, r4, c2     ; += ambient
dp3    r3, r5, c1     ; Intensity of base map
mul    r5, r4, r5     ; Modulate base map by light
mul    r4, r4, r3     ; Modulate light by base map intensity

phase

texld r0, t1          ; sample the first three channels of the TAM
texld r1, t1          ; sample the second three channels of the TAM
texld r2, r4          ; Get weights for 123
texld r3, r4          ; Get weights for 456

dp3_sat r0, 1-r0, r2  ; dot the reg0 (TAM values) with reg2 (TAM weights)
dp3_sat r1, 1-r1, r3  ; dot the reg1 (TAM values) with reg3 (TAM weights)
add_sat r0, r0, r1    ; add reg0 and reg1
mul r0.rgb, 1-r5, r0  ; Color hatches with base texture
mov_sat r0, 1-r0      ; complement and saturate
```

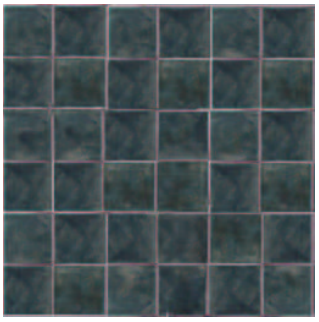
Real-Time Hatching with Per-Pixel TAM weights, distance attenuation and color tinting

Per-pixel Variable Specular power

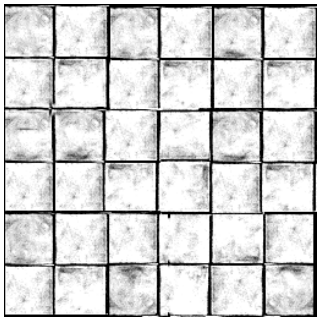
In the preceding example, we have illustrated the ability to migrate one type of per-vertex computation (TAM weight calculation) to the pixel level in order to improve rendering quality. We will now show how to implement per-pixel material properties (in this case, specular exponent) by using arbitrary register write masks and dependent texture reads in ps.1.4. We will use three different texture maps in this shader:

1. Albedo / Gloss map
2. Normal / k map
3. N·H × k map (function look up)

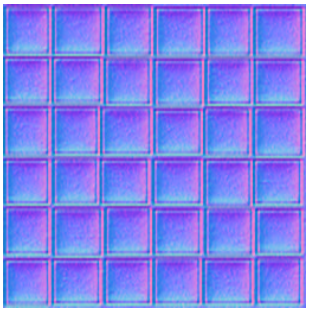
The first two of these maps are shown below. The images on the left are the RGB channels of the maps and the images on the right are the alpha channels. In the first map, we store albedo and gloss for the tile material. The second map stores the x, y and z components of the tangent-space normal in RGB and the specular exponent (k) in alpha. Note that the artist has given each tile in this texture map a different specular exponent to simulate neighboring tiles of disparate material properties. Being able to simply paint the quantity k into a texture map channel is both convenient and empowering to an artist.



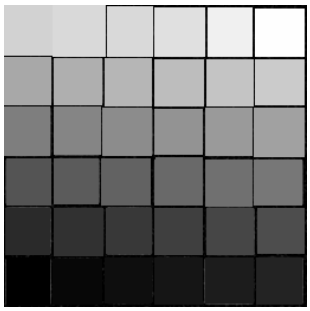
Albedo in RGB



Gloss in alpha



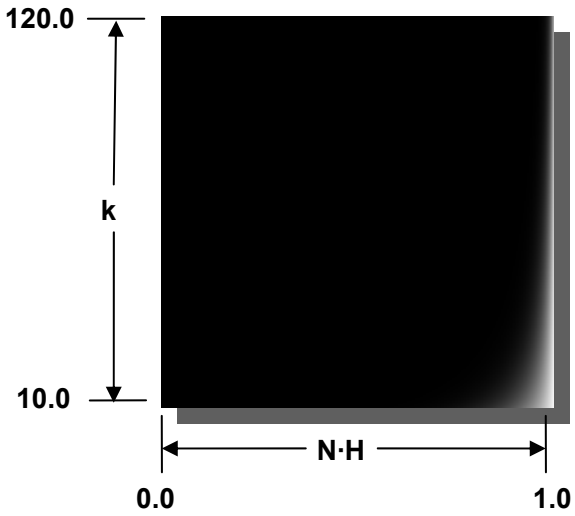
Normals in RGB



***k* in alpha**

Material maps for per-pixel specular exponent shader

The third texture we will use in this shader is a function lookup which will be used to raise N·H to the k^{th} power via a dependent texture read. Each row of this 2D texture can be thought of as an exponential function which is selected by the alpha channel of the Normal / *k* map shown above. In this way, we are able to select different specular exponents for different regions within the same texture map. For our purposes, we have found a dynamic range of 10 to 120 is reasonable for *k*:



Function look-up map for per-pixel specular exponent shader

In the shader code below, we sample the tangent space normal from the first map and dot this quantity with interpolated L and H vectors. N·H is stored in the red channel of `r2` and the specular exponent is moved into the green channel using write masks. This 2D texture coordinate is then used to access the function look-up map shown above via a dependent read. The instructions in the second phase composite the results into a final color.

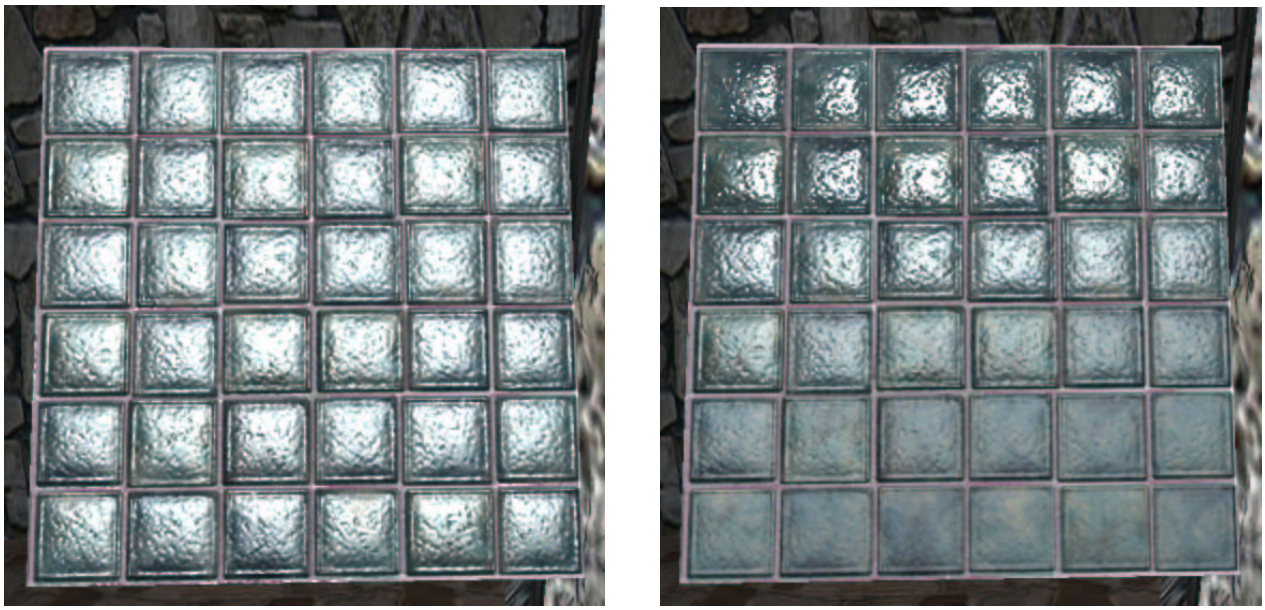
```
ps.1.4
texld   r1, t0                  ; Normal
texld   r2, t1                  ; Cubic Normalized Tangent Space Light Direction
texcrd  r3.rgb, t2              ; Tangent Space Halfangle vector

dp3_sat r5.xyz, r1_bx2, r2_bx2  ; N.L
dp3_sat r2.xyz, r1_bx2, r3      ; N.H
mov     r2.y, r1.a              ; K = Specular Exponent

phase
texld   r0, t0                  ; Base
texld   r3, r2                  ; Specular NHxK map

add     r4.rgb, r5, c7          ; += ambient
mul     r0.rgb, r0, r4          ; base * (ambient + N.L))
+mul_x2 r0.a, r0.a, r3.b        ; Gloss map * specular highlight
add     r0.rgb, r0, r0.a        ; (base*(ambient+N.L)) + (Gloss*Highlight)
```

Output from this shader is shown on the right side of the figure below. The left side shows the result of using the same normal map and a constant specular exponent for the whole object. The image on the right shows how different materials can be represented with the same map by migrating material calculations to the pixel level.



Constant specular power and per-pixel specular power using ps.1.4

Human Skin

The skin shader used in the Rachel demo uses nearly the maximum number of instructions to implement per-pixel diffuse and specular illumination for two lights. The shader computes the following equation to calculate the lighting per-pixel.

$$I_{RGB} = C_{base}(I_a + I_{d0}(\boldsymbol{N} \cdot \boldsymbol{L_0}) + I_{d1}(\boldsymbol{N} \cdot \boldsymbol{L_1})) + gI_s(I_{d0}\,|\boldsymbol{N} \cdot \boldsymbol{H_0}|^k + I_{d1}\,|\boldsymbol{N} \cdot \boldsymbol{H_1}|^k)$$

where
- C_{base} is the base color sampled from a texture map
- I_a is the light source ambient coefficient
- I_{dn} are light source diffuse coefficients
- I_s is the light source specular coefficient
- $\boldsymbol{N}$ is the normal to the surface
- $\boldsymbol{L}$ is the light vector
- $\boldsymbol{H}$ is the halfway vector
- g is the gloss factor
- k is the specular exponent.

A per-pixel variable specular exponent similar to the preceding example is used in this shader but is further improved by using a dependent projective texture fetch as a way to perform a division [Vlachos02].


```
 ps.1.4
 texld r0, t0
 texcrd r1.xyz, t3              // tangent space H0
 texcrd r2.xyz, t5              // tangent space H1
 dp3_sat r4.r, r0_bx2, r1       // N·H0
 dp3_sat r4.b, r1, r1           // H0·H0
 mul_sat r4.g, r4.b, c0.a       // c0.a*(H0·H0)
 mul r4.r, r4.r, r4.r           // (N·H0)^2
 dp3_sat r5.r, r0_bx2, r2       // N·H1
 dp3_sat r5.b, r2, r2           // H1·H1
 mul_sat r5.g, r5.b, c0.a       // c0.a*(H1·H1)
 mul r5.r, r5.r, r5.r           // (N·H1)^2
 phase
 texld r0, t0                   // fetch again to get spec map to use as gloss
 texld r1, t0                   // Cbase
 texld r2, t2                   // tangent space L0
 texld r3, t4                   // tangent space L1
 texld r4, r4_dz                // ((N·H)^2 /(H·H))^k @= |N·H|^k
 texld r5, r5_dz                // ((N·H)^2 /(H·H))^k @= |N·H|^k
 dp3_sat r2.r, r2_bx2, r0_bx2   // N·L0
+mul r2.a, r0.a, r4.r            // g * |N·H0|^k <- Gloss specular highlight 0
 dp3_sat r3.r, r3_bx2, r0_bx2   // N·L1
+mul r3.a, r0.a, r5.r            // g * |N·H1|^k <- Gloss specular highlight 1
 mul r0.rgb, r2.a, c2           // Id0*g*|N·H0|^k
 mad_x2 r0.rgb, r3.a, c3, r0    // Id0*g*|N·H0|^k + Id1*g*|N·H1|^k
 mad r2.rgb, r2.r, c2, c1       // Ia + Id0*(N·L)
 mad r2.rgb, r3.r, c3, r2       // Ia + Id0*(N·L) + Id1*(N·L)
 mul r0.rgb, r0, c4             // Is * (Id0*g*|N·H0|^k + Id1*g*|N·H1|^k)
 mad_x2_sat r0.rgb, r2, r1, r0 // Cbase * (Ia + Id0*(N·L) + Id1*(N·L))
                                // + Id0*g*|N·H0|^k + Id1*g*|N·H1|^k
+mov r0.a, c0.z
```

Conclusion

We've outlined the behavior of the 1.4 pixel shading model which is available in DirectX 8.1 and is implemented by the ATI RADEON™ 8500. Three key examples have been presented to illustrate the properties of this programming model and the effects that can be achieved today on commodity graphics hardware.

References

[Praun01] Emil Praun, Hugues Hoppe, Matthew Webb and Adam Finkelstein, "Real-Time Hatching." *Proceedings of SIGGRAPH 2001*, pages 579-584.

[Vlachos02] Alex Vlachos, John Isidoro and Christopher Oat, "Textures as Lookup Tables for Per-Pixel Lighting Computations," in *Game Programming Gems 3*, Dante Treglia editor, 2002

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).

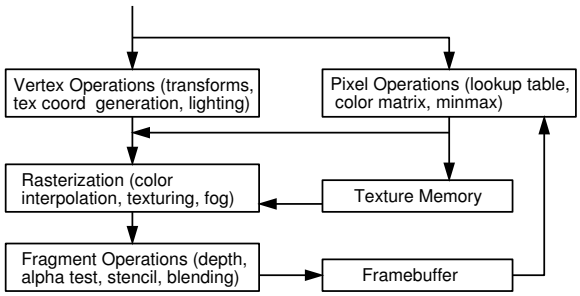


Figure 1: A simplified block diagram of the OpenGL architecture. Geometric data passes through the vertex operations, rasterization, and fragment operations to the framebuffer. Pixel data (either from the host or the framebuffer) passes through the pixel operations and on to either texture memory or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

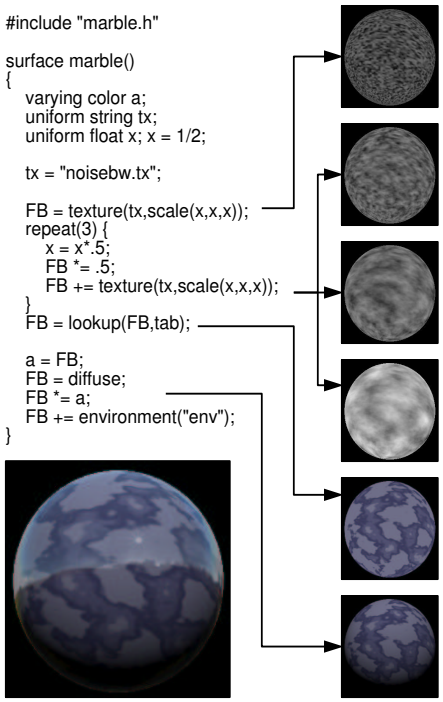


Figure 2: SIMD Computation of a Shader. Some of the different passes for the shader written in ISL listed on the left are shown as thumbnails down the right column. The result of the complete shader is shown on the lower left.

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.

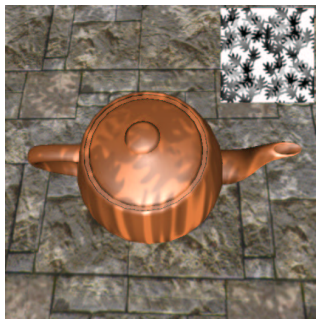
The scene graph passes information to the compiler including the matrix to transform from the object's coordinate system into camera space and the screen space footprint for the geometry. The footprint is computed during the ShadeAction by projecting a 3D bounding box of the geometry into screen space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

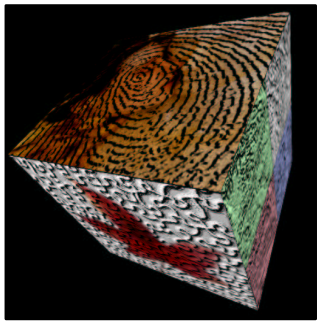
3 EXAMPLE: INTERACTIVE SL

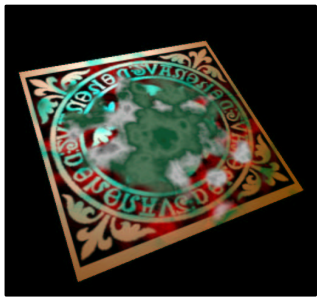
We have developed a constrained shading language, called ISL (for Interactive Shading Language) [25] and an ISL compiler to demonstrate our method on current hardware. ISL is similar in spirit to the RenderMan Shading Language in that it provides a `C`-like syntax to specify per-pixel shading calculations,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

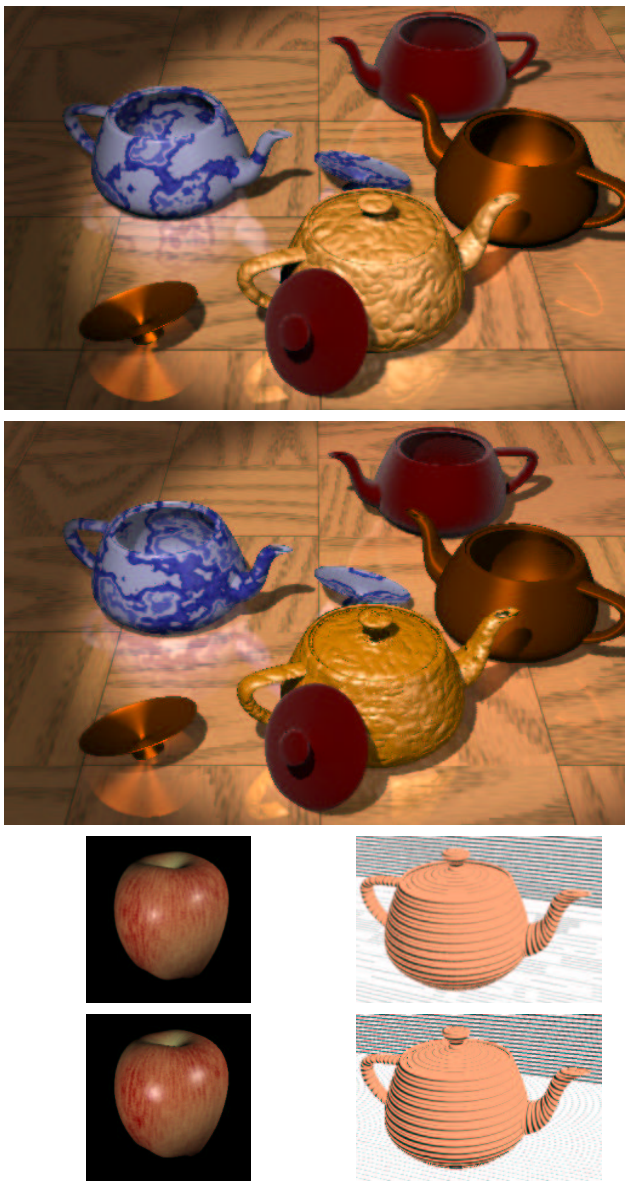


Figure 4: RenderMan SL Examples. The top and bottom images of each pair were rendered with PhotoRealistic RenderMan from Pixar and our multi-pass OpenGL renderer, respectively. No shaders use image maps, except for the reflection and depth shadow maps generated on the fly. The wood floor, blue marble,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

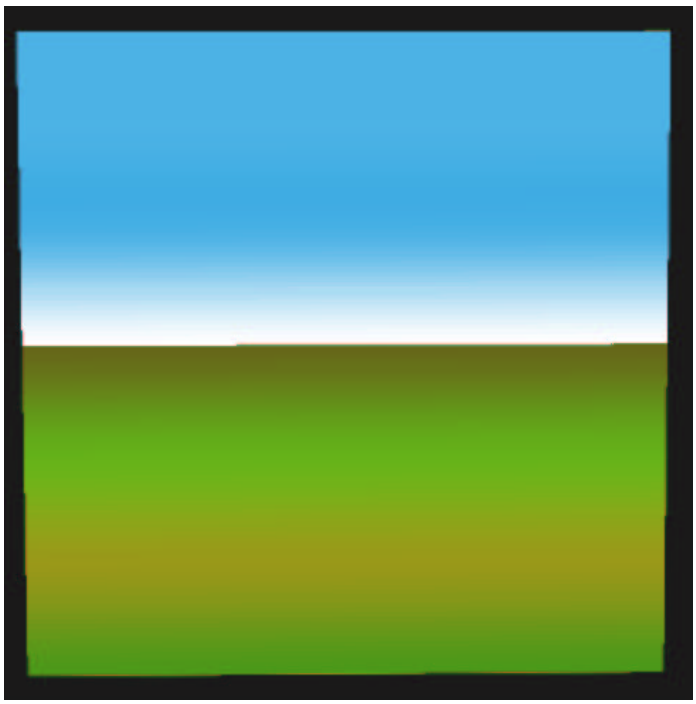


Figure 1: Simple environment

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"
```

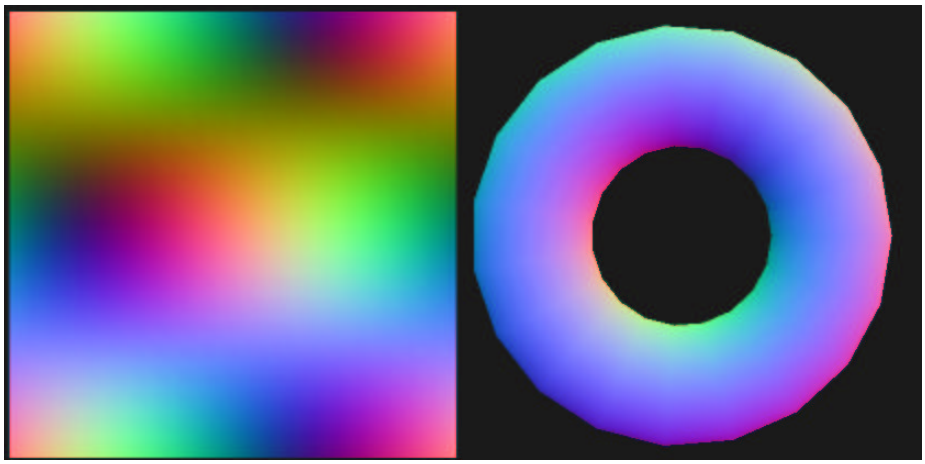


Figure 2: Simple normal map

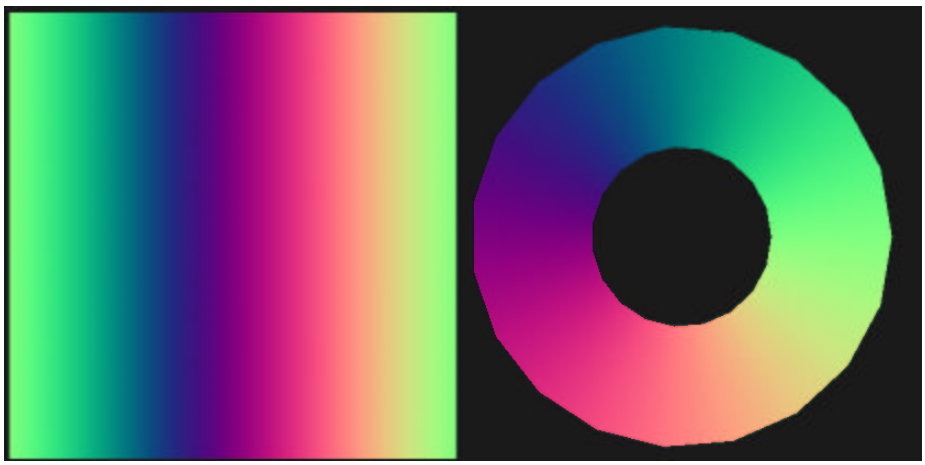


Figure 3: S Tangent map

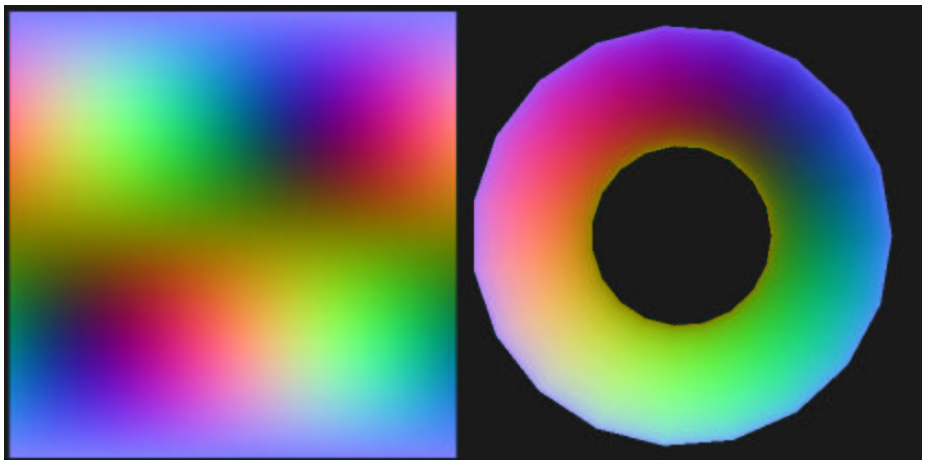


Figure 4: T Tangent map

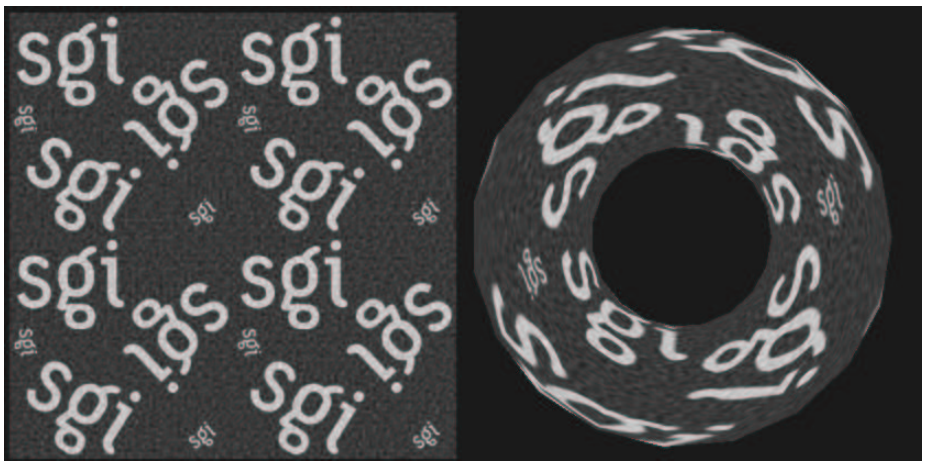


Figure 5: Bump map

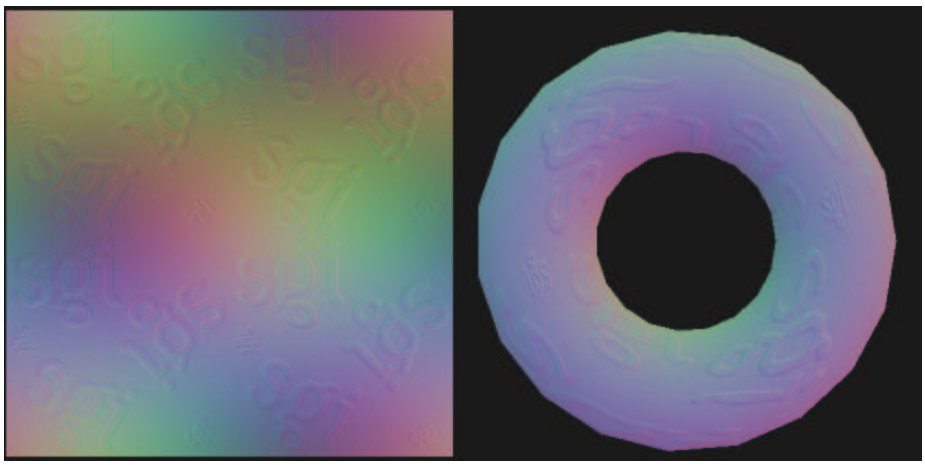


Figure 6: Bump mapped normal map

```
    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,
```

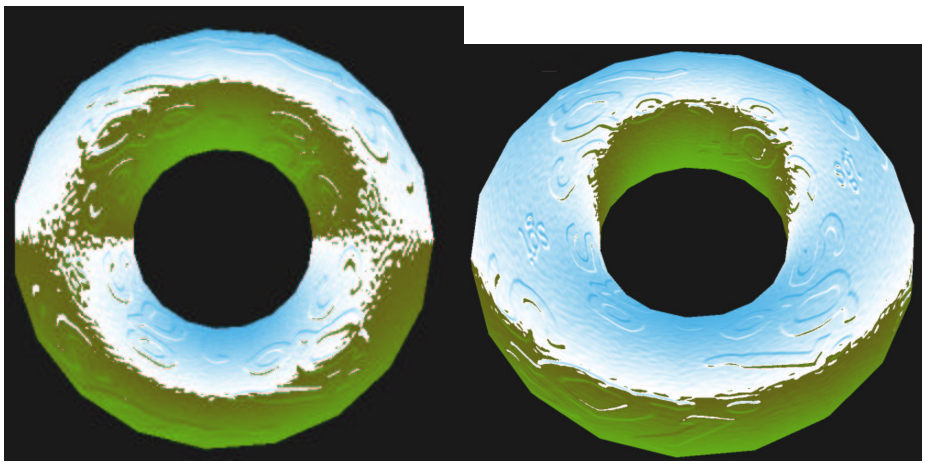


Figure 7: Final shiny/bumpy shader results

```
                                    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)

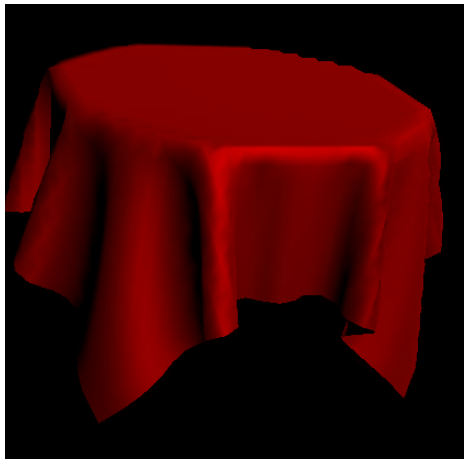


Figure 9: Fabric rendered with factorized BRDF

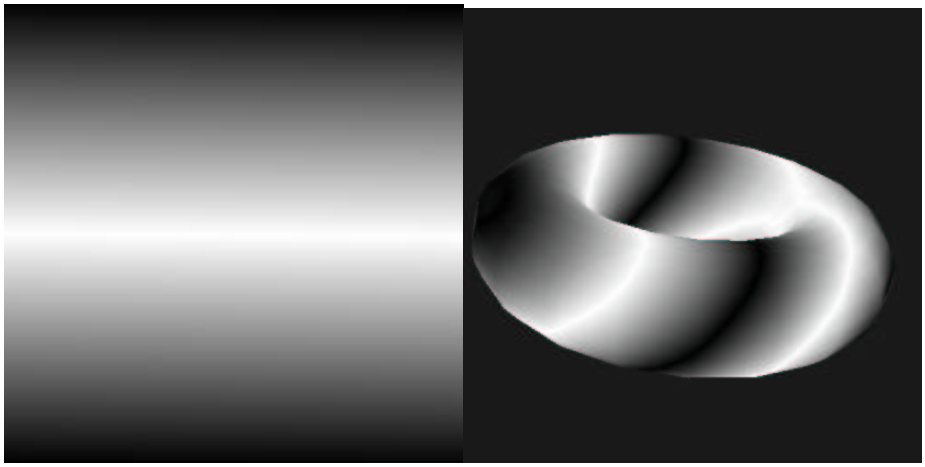


Figure 10: Ramp texture map and pattern when projected onto an object

Figure 11: Using ramp to make simple parameterized color choice

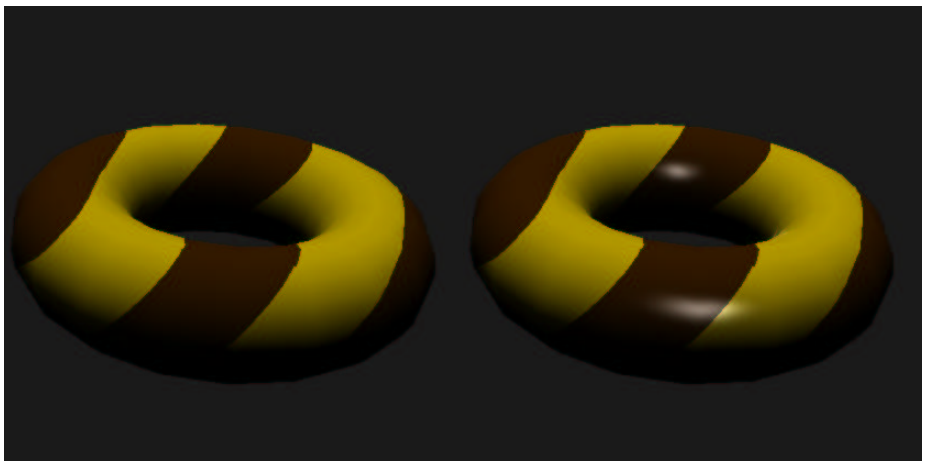


Figure 12: Differing diffuse and specular for each color band

```
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

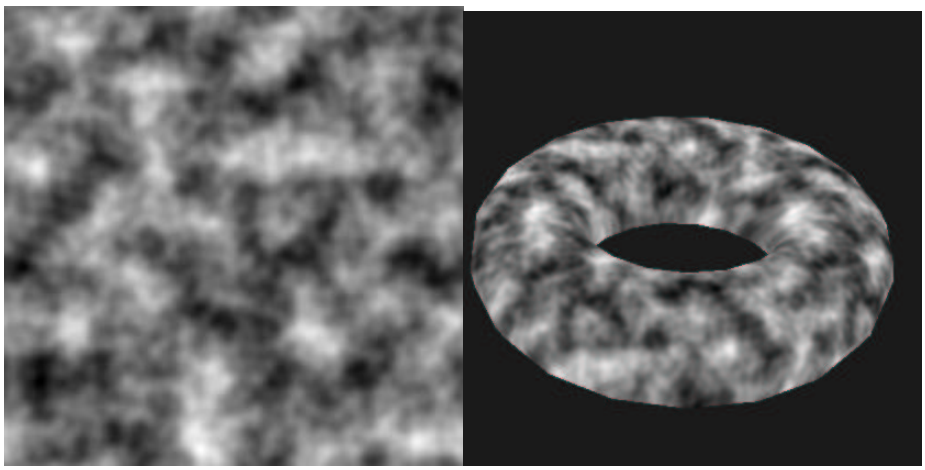


Figure 13: Turbulence texture

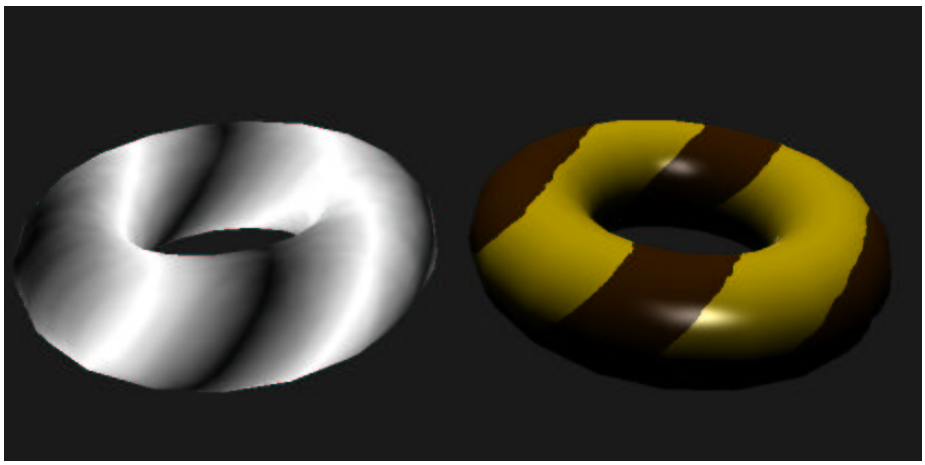


Figure 14: Wood with turbulence added to band boundary

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(
```

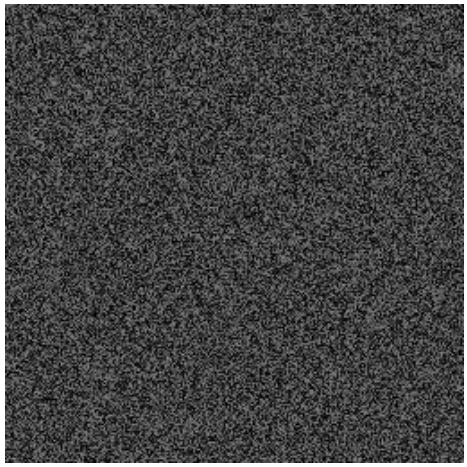


Figure 15: Simple noise texture

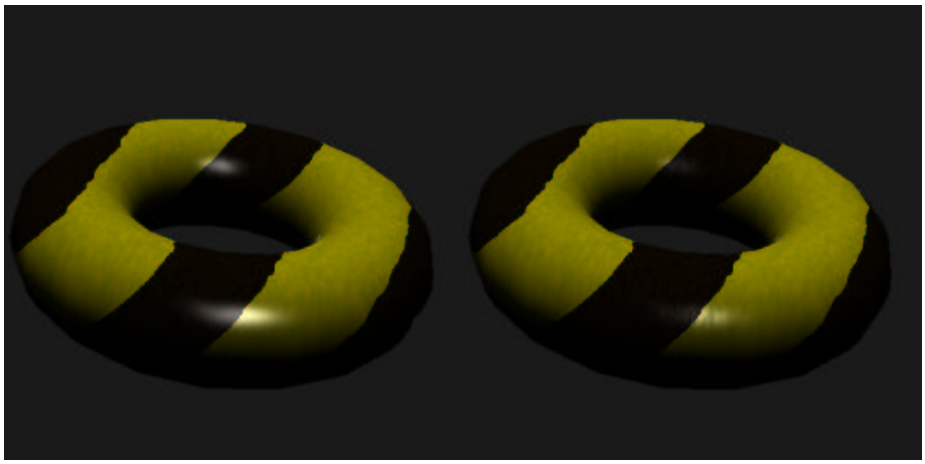


Figure 16: Final wood

```
    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.

Chapter 5

DirectX

Chas Boyd

Chapter 5 Hardware Shading with Direct3D

This white paper provides a short history and overview of the hardware accelerated shading models and the mechanisms for unifying them that are provided by the DirectX graphics library known as D3DX for use with the Direct3D low-level API.

Goals

Direct3D was developed to meet requirements for game programmers. They have two primary needs:
1) Deliver novel visual experiences based on the increasing features and performance available from graphics accelerator fabrication processes.
2) Support enough total systems to enable a reasonable sales volume.

This last requirement means the API needs to span hardware implementations along two axes. It must not only support multiple manufacturers, but also the different generations of hardware from each manufacturer. Due to the rapid rate of innovation used to deliver novel experiences, there are often multiple generations of a given brand of hardware in the consumer marketplace, and while the latest version of a given accelerator can be very important to a game developer to target, the publishers of each game title would like to take advantage of a larger installed base by including earlier hardware.

While creating an API that can help applications span multiple brands of hardware is difficult, helping them span consecutive generations is actually the tougher problem. Direct3D has typically sorted the features differences into those that can be accommodated by relatively localized code branches (for which capability bits are provided), and those that represent generational changes.

Much of the recent work on DirectX Graphics has been dedicated to helping applications with this latter task. A key result of this work is the D3DX Effect Framework, a mechanism for helping applications manage different rendering techniques enabled by the various generations of hardware.

Effects

With the change in hardware model for multi-texture, it became clear that a successful game would need to provide different code paths for each of the 2 or 3 generations of hardware it targeted. This is because each new generation of hardware has not only more advanced features, but also improved performance. This makes it very difficult to emulate the newer generation's features on older hardware. The performance of emulating a technique is usually worse than the true technique, yet that older hardware already has less performance.

For example, many multi-texture blending operations can be emulated by multi-pass frame-buffer blending operations. However, the older hardware that requires multi-pass

emulation is so much slower that to provide acceptable performance it really should be used for fewer passes, not more.

As a result, separate code paths and in some cases separate art content (texures and models) are often needed for each generation of hardware. This level of impact on application architecture is non-trivial, however, many applications began to be implemented using this type of architecture

Starting in DirectX7, support was provided for effects in the D3DX library of extensions to Direc3D. Effects are a way of architecting rendering code that isolates and manages implementation-specific details from the application, allowing it to more easily span generations of hardware, or other variations in implementations. An effect object is a software abstraction that gathers together a collection of different techniques which could be used to implement a common logical visual effect.

There is very little policy associated with them. The different techniques used to implement an effect can be selected based on any criterion, not just feature set. Criteria such as performance, distance from the camera, user preferences etc. are also commonly used, and sometimes combined together.

Effect objects can be loaded from and persisted in .fx files. This enables them to serve as collections of binding code that map various rendering techniques into applications.

Because of the efficiency of the effect loading process, they can be dynamically loaded while an application is running. This is extremely useful for efficient application development. When an effect is modified while the application is running, it not-only saves the time required to reload the app that would be required using a compile-based process, but more importantly, the time for the app to reload all the models and textures it is using.

Like OpenGL, Direct3D does not have a concept of geometric objects to which shaders are assigned to directly. The API is designed to be a flexible substrate for implementation of higher-level object models such as hierarchical scene graphs, cell-portal graphs, BSPs, etc. The data model is based on vertices, textures, and the state that controls the device, of which shaders are a part.

This flexibility is preserved in the Effect binding mechanism. Effects manage only the state information, so the application is free to render its content using any low-level rendering calls it considers appropriate.

Effects provide a clean way to make an application scale from single-texture to multi-texture to programmable shaders. Or to manage hardware vs software vertex transformation pipelines.

The following listings show the evolution of the hardware and API generations using multi-texture, assembly level shaders, and the C-level language.

DirectX 6 Multi-Texture

The following listing shows the code required to implement a specular per-pixel bump map using the dependent read mechanism introduced in late 1998 in DirectX 6.0, and supported in hardware on the Matrox G400 in 1999, the ATI Radeon in 2000, and the nVidia GeForce 3 in 2001, among others.

```
/*  Render()
Implements the following specular bump mapping rendering using multitexture syntax

0   MODULATE( EarthTexture, Diffuse );   // light the base texture
1   BUMPENVMAP( BumpMap, _ );            // sample bump map (other arg ignored)
2   ADD( EnvMapTexture, Current );       // sample envt map using bumped texcoords
                                         // and add to result
*/
HRESULT CMyD3DApplication::Render()
{
    m_pd3dDevice->Clear( 0L, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
                         0x00000000, 1.0f, 0L );

    m_pd3dDevice->BeginScene();

    m_pd3dDevice->SetRenderState( D3DRS_WRAP0, D3DWRAP_U | D3DWRAP_V );

    m_pd3dDevice->SetTexture( 0, m_pEarthTexture );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 1 );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_MODULATE );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );

    m_pd3dDevice->SetTexture( 1, m_psBumpMap );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 1 );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP,   D3DTOP_BUMPENVMAP );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT00,   F2DW(0.5f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT01,   F2DW(0.0f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT10,   F2DW(0.0f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT11,   F2DW(0.5f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVLSCALE,  F2DW(4.0f) );
    m_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVLOFFSET, F2DW(0.0f) );

    m_pd3dDevice->SetTexture( 2, m_pEnvMapTexture );
    m_pd3dDevice->SetTextureStageState( 2, D3DTSS_TEXCOORDINDEX, 0 );
    m_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLOROP,   D3DTOP_ADD );
    m_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLORARG1, D3DTA_TEXTURE );
    m_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLORARG2, D3DTA_CURRENT );


    m_pd3dDevice->SetStreamSource( 0, m_pEarthVB, 0, sizeof(BUMPVERTEX) );
    m_pd3dDevice->SetFVF( BUMPVERTEX::FVF );

    if( FAILED( m_pd3dDevice->ValidateDevice( &dwNumPasses ) ) )
    {
        // The right thing to do when device validation fails is to try
        // a different rendering technique.  This sample just warns the user.
```

```
        m_bDeviceValidationFailed = TRUE;
    }
    else
    {
        m_bDeviceValidationFailed = FALSE;
    }

    // Finally, draw the Earth
    m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, m_dwNumSphereVertices );

    return;
}
```

DirextX 8 Pixel Shaders

The following listing shows an effect file containing a single technique that implements the homomorphic factorization method of McCool, Ang, and Ahmad for rendering BRDFs. It uses DirectX 8 pixel shader version 1.1 and vertex shader version 1.1.

```
//////////////////////////////////////////////////////////////////////////
// BRDF Effect File
// Copyright (c) 2000-2002 Microsoft Corporation. All rights reserved.
//

vector lhtR;            // Light Direction from app
vector MaterialColor;   // Object Diffuse Material Color


matrix mWld;     // World
matrix mTot;     // Total


texture BRDFTexture1;
texture BRDFTexture2;
texture BRDFTexture3;
texture ObjectTexture;

// These strings are for the app loading the fx file

// Technique name for display in viewer window:
string tec0 = "BRDF Shader";

// Background Color
DWORD  BCLR = 0xff0000ff;

// model to load
string XFile = "sphere.x";

// BRDF technique
technique tec0
{
    pass p0
    {
        //load matrices
        VertexShaderConstant[0] = <mWld>;           // World Matrix
        VertexShaderConstant[4] = <mTot>;           // World*View*Proj Matrix

        //Material properties of object
        VertexShaderConstant[9]  = <MaterialColor>;

       // Light Properties.
       // lhtR, the light direction, is input from the shader app
       // for BRDFs, these color constants are built into the texture maps
        VertexShaderConstant[16] = <lhtR>;            // light direction

        Texture[0] = <BRDFTexture1>;
        Texture[1] = <BRDFTexture2>;
        Texture[2] = <BRDFTexture3>;
```

```
Texture[3] = <ObjectTexture>;

// Only one colour being used
ColorOp[1]   = Disable;
AlphaOp[1]   = Disable;

AddressU[0] = clamp;       // set up clamping for cube maps
AddressV[0] = clamp;
AddressW[0] = clamp;

AddressU[1] = clamp;
AddressV[1] = clamp;
AddressW[1] = clamp;

AddressU[2] = clamp;
AddressV[2] = clamp;
AddressW[2] = clamp;

// object's detail texture
AddressU[3] = wrap;
AddressV[3] = wrap;


// Definition of the vertex shader, declarations then assembly.
VertexShader =
decl
{
    stream 0;
    float v0[3];        // Position
    float v3[3];        // Normal
    float v7[3];        // Texture Coord1
    float v8[3];        // Tangent
}
asm
{
    vs.1.1              // version number

    m4x4 oPos, v0, c4   // transform point to projection space
    m4x4 r4,v0,c0       // transform point to world space
    m3x3 r0,v3,c0       // transform Normal to World Space, result in r0
    m3x3 r1,v8,c0       // transform Tangent to World Space

    mul r2,-r1.zxyw,r0.yzxw;      // comptue binorm with cross product
    mad r2,-r1.yzxw,r0.zxyw,-r2;

    // get negative view vector
    add r4.xyz,-r4.xyz,c10
    dp3 r5.x,r4.xyz,r4.xyz
    rsq r5.x,r5.x
    mul r4.xyz,r5.xxx,r4.xyz

    //compute the half angle and normalize
    add r5.xyz,r4.xyz,-c16
    dp3 r6.x,r5.xyz,r5.xyz
    rsq r6.x,r6.x
    mul r5.xyz,r6.xxx,r5.xyz
```

```
            //now inverse transform everything
            //r1 = tangent
            //r2 = binormal
            //r0 = normal

            //t0 = view
            dp3 oT0.x,r4,r1
            dp3 oT0.y,r4,r2
            dp3 oT0.z,r4,r0

            //t1 = -light
            dp3 oT1.x,-c16,r1
            dp3 oT1.y,-c16,r2
            dp3 oT2.z,-c16,r0

            //t2 = half angle
            dp3 oT2.x,r5,r1
            dp3 oT2.y,r5,r2
            dp3 oT2.z,r5,r0

            //object might have its own texture
            mov oT3.xy,v7.xy

            //move diffuse color in
            mov oD0,c9
        };

        pixelshader =
        asm
        {
            ps.1.1
            tex t0
            tex t1
            tex t2
            tex t3

            mul r0,t0,t1     // combine 1st 2 brdf factors
            mul r0,r0,t2     // multiply in 3rd factor
            mul r0,r0,t3     // apply detail texture
            mul r0,r0,v0     // light using diffuse primary color
        };
    }
}
```

DirectX 9 High Level Shading Language

The following is a very simple wood shader from the Renderman Companion. Implemented as a DirectX 9.0 effect file, it demonstrates the integration of the "C"-like high-level shading language into the effect framework. This should run in 1 pass of the pixel shader 2.0 model supported in DirectX 9.0.

```
fx.2.0

//////////////////////////////////////////////////////////////////////////////
// Effect parameters /////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////

float ringscale = 10.0f;
float point_scale = 1.0f, turbulence = 1.0f;

vec3 lightwood = vec3(0.3f, 0.12f, 0.03f);
vec3 darkwood  = vec3(0.05f, 0.01f, 0.005f);

float Ka = 0.2;
float Kd = 0.4;
float Ks = 0.6;
float roughness = 0.1;

vec3 ambient_color;
vec3 diffuse_color;
vec3 specular_color;

volumetexture noise;

vec3 L;
vec3 I;


//////////////////////////////////////////////////////////////////////////////
// Renderman-like helper functions ///////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////

float smoothstep(float a, float b, float s)
{
    float s2 = s * s;
    float s3 = s * s2;

    float sV1 = 2.0f * s3 - 3.0f * s2 + 1.0f;
    float sT1 = s3 - 2.0f * s2 + s;
    float sV2 = -2.0f * s3 + 3.0f * s2;
    float sT2 = s3 - s2;

    return s < 0.0f ? a : s > 1.0f ? b : sV1 * a + sT1 + sV2 * b + sT2;
}

vec3 diffuse(vec3 N)
{
    return diffuse_color * dot(N, L)
}
```

```
vec3 specular(vec3 N, vec3 eye, float roughness)
{
    vec3 H = (L + eye) / len(L + eye);
    return specular_color * pow(dot(H, N), 1 / roughness);
}



////////////////////////////////////////////////////////////////////////////
// Wood Shader ////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

struct PS_INPUT
{
    vec3 Color    : COLOR0;
    vec3 Position : TEXCOORD0;
    vec3 Normal   : TEXCOORD1;
};

struct PS_OUTPUT
{
    vec4 Color : COLOR0;
};



PS_OUTPUT Wood(const PS_INPUT v)
{
    /* Perturb P to add irregularity */
    vec3 PP = v.Position * point_scale;
    PP += sample3d(noise, PP) * turbulence;

    /* Compute radial distance r from PP to axis of tree */
    float r = sqrt(PP.y * PP.y + PP.z * PP.z);

    /* Map radial distance r into ring position [0,1] */
    r *= ringscale;
    r += abs(sample1d(noise, r) * turbulence);
    r  = frc(r);

    /* Use r to select wood color */
    r = smoothstep(0.0f, 0.8f, r) - smoothstep(0.83f, 1.0f , r);

    /* Shade using r to vary brightness of wood grain */
    PS_OUTPUT o;

    o.Color.rgb = (ambient_color * Ka + diffuse(v.Normal) * Kd) *
        lerp(lightwood, darkwood, r) * v.Color.a +
        specular(v.Normal, -I, roughness) * (Ks * (0.3f * r + 0.7f));

    o.Color.a = v.Color.a;

    return o;
};
```

Additional Resources

Chas. Boyd
chasb@microsoft.com

DirectX email contact for applications to beta program:
directx@microsoft.com

Developer web site with links, whitepapers, and SDK downloads:
http://msdn.microsoft.com/DirectX

Presentations on techniques at previous conferences:
http://www.microsoft.com/mscorp/corpevents/meltdown2001/presentations.asp

http://www.microsoft.com/mscorp/corpevents/gdc2001/developer_day.asp

Per-Pixel Lighting

Philip Taylor

Microsoft Corporation

November 13, 2001

Download the source code for this article.

Note This download requires DirectX 8.1.

This column is based on material contained in the "Per-Pixel Lighting" talk, developed and delivered by Dan Baker and Chas Boyd at GDC 2001. In the interests of space, I am not going to cover several advanced topics covered in the slide material (available at http://www.microsoft.com/mscorp/corpevents/meltdown2001/presentations.asp and http://www.microsoft.com/corpevents/gdc2001/developer_day.asp) like anisotropic lighting, and per-pixel environment mapping.

Instead, this column will focus on the fundamentals of pixel lighting, the standard models, the process of defining new models, and provide an example of defining and implementing a new lighting model using pixel shaders. It's in the area of custom or "do-it-yourself" lighting models that pixel shaders really shine—but lets not get ahead of ourselves.

Fundamentals of Per-Pixel Lighting

First, I assume everyone is familiar with basic diffuse and specular lighting. This assumes a physical model, like that shown in Figure 1. Let's examine the standard lighting model, and define the system and the terminology.

Figure 1 below is a diagram showing the standard lighting setup used to describe Direct3D's fixed-function lighting. There is a vertex, defined by the position P, a Light, defined by the L vector, the View position defined by the V vector, and the Normal defined by the N vector. In addition, the diagram shows the "half-vector" H, part of the Blinn simplification to Phong shading. These elements are sufficient to describe both the diffuse and specular reflectance lighting system.

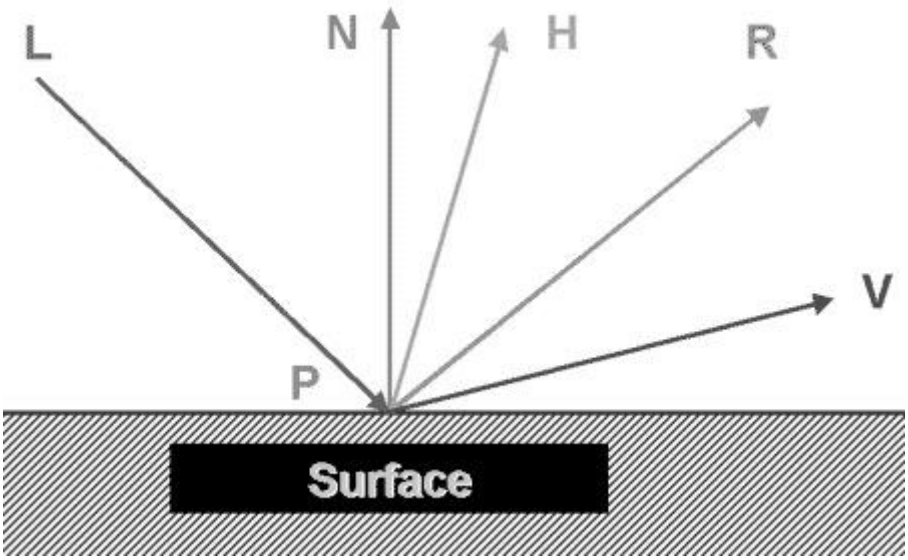


Figure 1. Standard lighting diagram

P = Vertex position
N = unit normal vector of Vertex P
L = unit vector from Light to the Vertex position
V = unit vector from Vertex P to the View position V
R = unit vector representing light reflection R
H = unit vector halfway H, between L and V, used for Blinn simplification

Diffuse lighting uses the relationship N dot L, where dot is the dot product, to determine the diffuse contribution to color. The dot product represents the cosine of the angle between the two vectors, so when:

The angle is acute, or small, the cosine value is large, and so this component contribution to the final color is larger.

The angle is obtuse, or large, the cosine value is small, and so this components contribution to the final color is smaller.

The Phong formulation for specular lighting uses a reflected vector R, representing the direction the light is reflected, with the Light vector L, R dot V, raised to a power n. The power value n allows simulation of a variety of surfaces, so when:

The power value n is large, the resulting highlight is tight and shiny, simulating a glossy surface.

The power value n is small, the resulting highlight is large and dull, simulating a less glossy surface.

The Blinn simplification replaces the R vector with the vector H halfway between V and L, and modifies the power value n to produce a result sufficiently similar to the more complex calculation for good image

quality purposes—but at a significantly lower computational cost, since H is much cheaper to calculate than R.

Direct3D lighting, whether in hardware or software, uses these equations at the vertex level. Unfortunately, vertex lighting can have two undesired properties:

1. Vertex lighting requires extra tessellation to look good; otherwise the coarseness of the underlying geometry will be visible.
2. Vertex lighting causes all applications that use it to have a similar look.

Tessellation becomes critical for vertex lighting to look good, since the triangle rasterizer linearly interpolates the vertices without a deep understanding of local geometry. If the geometry is too coarse, or the geometry contains a lot of variation in a short distance, then the hardware can have a problem producing values that result in good image quality. Increasing tessellation, however, reduces performance. Couple that with the fact that vertex lighting always has a telltale visual signature, and it's rarely compelling. Exceptions are when vertex lighting is used for global ambient, or in addition to per-pixel lighting.

Now, with that understanding of lighting, it is easier to see what all the fuss is about with respect to per-pixel lighting. Of course everyone wants per-pixel lighting, as it really is that much better.

There are two approaches for per-pixel lighting:

1. Pixel lighting in world space.
2. Pixel lighting in tangent space.

Now, in looking at approach two, you may say, "Wait a minute there, Phil, what is this 'tangent space,' and where did that come from?"

Per-pixel lighting uses texture mapping to calculate lighting. That's not new, as light mapping using textures has been utilized in games for years to generate lighting effects. What is new, however, is that in pixel shaders, the values in the texture map can be used to perform a lighting calculation, as opposed to a texture-blend operation.

Now that a lighting calculation is involved, great care must be taken to ensure that the lighting computation is done in the correct 3d basis (also called a coordinate space, or "space" for short). All lighting math must be calculated in the same coordinate space. If a normal is in a different space than the light direction, any math between them is bogus. It would be like multiplying feet by meters; it just doesn't make sense.

With this requirement in mind, any lighting approach needs to manipulate the source values to make sure all components of the calculations are in the same space. In our case here, there are two sets of inputs:

1. Normal and bump maps, stored in texture or "tangent" space.
2. Light directions and environment maps, stored in object or world space.

For normal maps, the texels in a texture map don't represent colors, but vectors. Figure 2 below shows the coordinate space the normals are in. The standard u and v directions along the textures width and height are joined with a "w" direction that is normal to the surface of the texture, to finish the basis (u,v,w). That is, a texel of 1,0,0 actually translates to 1 component of the u vector, 0 of the v vector, and 0 of the w vector. If u, v, and w are perpendicular to each other, this is called an orthonormal basis. This generates a texture space basis, which by convention is called the "tangent space basis."

One thing you may have wondered about u, v, and w, is where in the world are these vectors? At first, it looks like they are part of the texture. In reality, these vectors get pasted onto the real world object that is being textured. At each vertex, u, v, and w might point in an entirely different direction! So, two places on a mesh that have the same piece of texture might have completely different vectors, since u, v, and w are different. Remember, an x, y, z vector in the normal map really means x*u + y*v + z*w. If u, v, w are different, then the vectors are different even if x, y and z are the same. The hard part is finding u, v, and w; after that, everything else is pretty straightforward.

Figure 3 shows the relative coordinate spaces for both objects and texture maps. In either approach to pixel lighting, all source data needs to be moved into one space or the other, either from tangent space into world space, or from world space into tangent space.

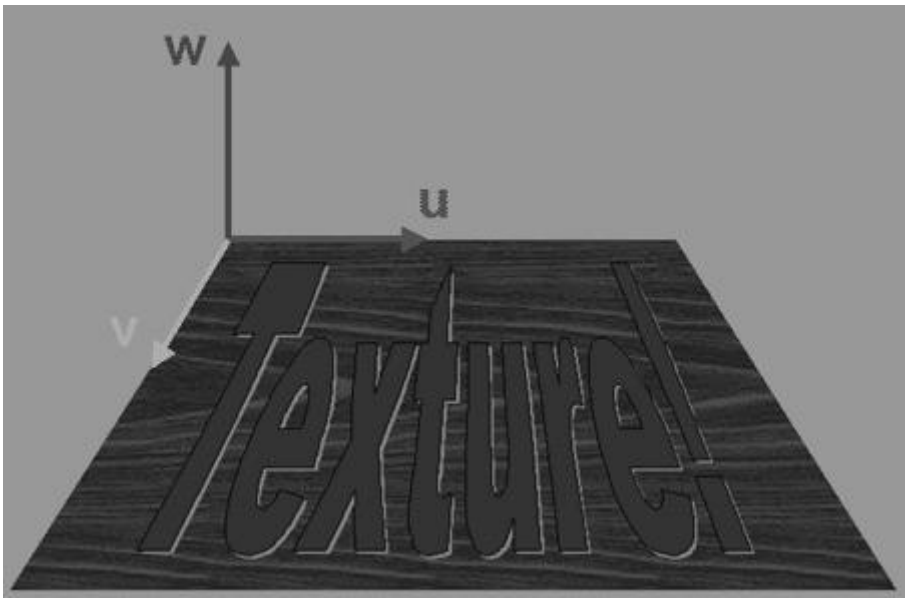


Figure 2. Texture coordinate system

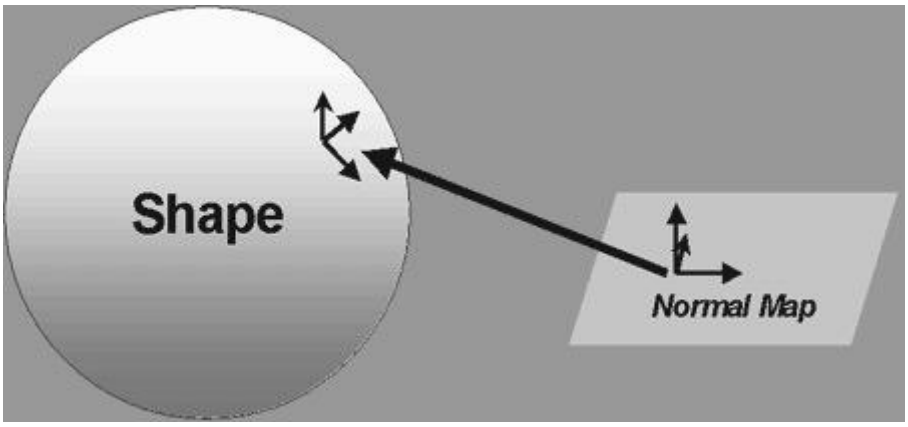


Figure 3. Basis computation diagram

Now that a tangent space (u,v,w) has been defined (more on how to find these vectors later), it's time to look at world-space pixel lighting. In world-space pixel lighting, it's necessary to transform the texture data—for example, the texture map texels into world space. Here the light vectors are not transformed in the vertex shader; they are left in world space, as it's the texture data that needs transformation. Simply pass the tangent basis transform into the pixel shader as three texture coordinates. Note that it's possible to generate the binormal by a cross product to save bandwidth. The iterators will interpolate this matrix to each pixel. Then use the pixel-shader and the tangent space basis to perform per-pixel lighting.

Tangent space pixel lighting is the reverse. Here it's necessary to transform the light and environment map data into tangent space. The light vectors are transformed in the vertex shader, and then passed down to the pixel shader. The environment map data is transformed in the pixel shader, and then the pixel shader performs per-pixel lighting.

Let me discuss one other terminology convention. The normal, the w, and the z axis vectors are all defined to be the same vector; the tangent is the u-axis vector, and the binormal is the v-axis vector. It's possible, and necessary, to pre-compute the tangent vector u as the vector that points along the u-axis of the texture at every point.

Now it's time to examine the tangent space transformation process.

First, recognize this is an inverse transformation. From linear algebra, if one has an orthonormal basis, the inverse matrix is the transpose. So, given (u,v,w) are the basis vectors of the surface, and L is the light, the transform is:

```
   [U.x U.y U.z] * [-L.x]
```

```
[V.x V.y V.z] * [-L.y]

[w.x W.y W.z] * [-L.z]
```

Expanding out, this becomes

```
L.x' = DOT3(U,-L)

L.y' = DOT3(V,-L)

L.z' = DOT3(W,-L)
```

where

U = the tangent along the x-axis of the texture;
V = the normal;
W = the binormal U x V (cross product).

Note that one performs the dot product with the negative of the Light direction.

Computing a tangent space vector is just like computing a normal. Remember that w is defined as the normal, so we should already have that piece of information. Now, we need to generate u and v. To generate the tangent space basis vectors (u and v), use the following equation:

```
Vec1 = Vertex3 – Vertex2

Vec2 = Vertex1 – Vertex2

DeltaU1 = Vertex3.u - Vertex2.u

DeltaU2 = Vertex1.u - Vertex2.u

DirectionV = |DeltaU2*Vec1-DeltaU1*Vec2|

DirectionU = |DirectionV x Vertex.N|

DirectionW = |DirectionU x DirectionV|
```

Where

X indicates taking a cross product;
|| indicates taking a unit vector;
Vertex1-3 are the vertices of the current triangle.

Usually, tangents and normals are calculated during the authoring process, and the binormal is computed in the shader (as a cross product). So, the only field we are adding to our vertex format is an extra u vector. Additionally, if we assume the basis is orthonormal, we don't need to store v either, since it is just u cross w.

A couple of points to keep in mind: Tangents need to be averaged—and be careful about texture wrapping (since it modifies u and v values). Look for D3DX in DirectX 8.1 to include new methods to help with tangent space operations; check the documentation.

For character animation, as long as you skin the normals and the tangents, this technique works fine. Again, generate the binormal in the shader after skinning, so you can skin two vectors instead of three. This will work just fine with indexed palette skinning and 2/4-matrix skinning, as well as with vertex animation (morphing). In terms of performance, tangent space lighting is good, since the transform can be done at a per-vertex level. It's less clocks than vertex lighting, and the per-pixel dp3 dot product is as fast as any pixel operation, so there is no loss of performance there either. To perform diffuse and specular in the same pass, compute the light vector and the half-angle vector, and transform both into tangent space. The perturbation and dot product are then done in the pixel pipeline, either in the pixel shader or by using multi-texture.

Below is a section of a vertex shader that shows how to generate a tangent space light vector:

```
// v3 is normal vector
// v8 is tangent vector
// c0-c3 is World Transform
// c12 is light dir

//tangent space basis generation
m3x3 r3,v8,c0      // transform tan to world space
m3x3 r5,v3,c0      // transform norm to world space

mul r0,r3.zxyw,r5.yzxw   // cross prod to generate binormal
mad r4,r3.yzxw,r5.zxyw,-r0

dp3 r6.x,r3,-c12      // transform the light vector,
dp3 r6.y,r4,-c12      // by resulting matrix
dp3 r6.z,r5,-c12      // r6 is light dir in tan space
```

This can simply be repeated for any vector that needs to be transformed to tangent space.

One other thing to be aware of is the range of values required by dot product lighting. Dot product operations contain data represented in the range [-1,1] to perform lighting operations, or signed data. Standard texture formats contain data represented in the range [0,1] to map color values, and thus contain unsigned data. Both pixel shaders and the fixed-function pipeline define modifiers that remap texture data to bridge this gap so texture data can be used effectively in dot product operations. Pixel shaders define the **_bx2** argument modifier, that remaps input data from unsigned to signed. So the input arguments to the dot product operation usually have this modifier applied to them. It's also useful to clamp the results of the dot product to black using the **_sat** instruction modifier. Here is a typical dot product pixel shader instruction:

```
dp3_sat r0, t0_bx2, v0_bx2    // t0 is normal map, v0 is light dir
```

For the fixed-function pipeline, the similar process is done with the texture argument modifier flag **D3DTA_COMPLEMENT** for texture inputs, and the texture operators **D3DTOP_ADDSIGNED** for results in the range [-0.5,0.5], and **D3DTOP_ADDSIGNED2X** for results in the range [-1.0,1.0] range.

With this understanding of the basics of per-pixel lighting, it's time to examine the standard lighting models, and how diffuse and specular lighting work in pixel shaders.

Standard Lighting Models

The standard lighting models include diffuse and specular lighting. Each lighting model can be done with both pixel shaders and fixed-function multi-texture fallback techniques. Understanding these techniques and the fallbacks allows development of a shader strategy that can cope with the differing generations of graphics cards. DirectX 6.0 generation cards are multi-texture capable—almost all can do subtractive blending, and some can do dot product blending. Examples include TNT2, Rage128, Voodoo 5, and G400. DirectX 7.0 generation cards are both hardware transform and multi-texture capable, and almost all can do both subtractive and dot product blending. Examples include geForce2 and Radeon. All DirectX 8.0 cards can do vertex and pixel shaders in hardware. Examples include geForce3 and Radeon8500.

Per-pixel diffuse is consistent with standard lighting models with no specular. It's nice, because there is no need to modulate against another lighting term; each pixel is correctly lit after the per-pixel diffuse calculation. **Note that filtering can be a major problem, since normals cannot be filtered for a variety of reasons.** Below is a vertex shader fragment to perform setup for per-pixel diffuse lighting, including calculating the light direction in tangent space, biasing it for the per-pixel dot product, and setting up the texture coordinates for the pixel shader.

```
//v0 = position

//v3 = normal (also the w vector)
```

```
//v7 = texture coordinate
//v8 = tangent (u vector)

vs.1.1
//tranform position
m4x4 oPos,v0,c4

//tsb generation
m3x3 r3,v8,c0          //gen normal
m3x3 r5,v3,c0          //gen tangent

//gen binormal via Cross product
mul r0,-r3.zxyw,r5.yzxw;
mad r4,-r3.yzxw,r5.zxyw,-r0;

//diffuse, transform the light vector
dp3 r6.x,r3,-c16
dp3 r6.y,r4,-c16
dp3 r6.z,r5,-c16

//light in oD0
mad oD0.xyz,r6.xyz,c20,c20   //multiply by a half then add half

//tex coords
mov oT0.xy, v7.xy
mov oT1.xy, v7.xy
```

Next, a typical diffuse pixel shader is shown below:

```
ps.1.1
```

```
tex t0                  //sample texture

tex t1                  //sample normal


//diffuse

dp3_sat r1,t1_bx2,v0_bx2          //dot(normal,light)


//assemble final color

mul r0,t0,    r1             //modulate against base color
```

This is prototypical usage of dp3 to calculate N dot L. Conceptually, this is a good way to lay out the pixel shader. Figure 4 contains a screenshot of this shader in action. Notice the separation of the calculation and final color assembly. The following renderstates are used, shown in effects file format syntax:

```
VertexShaderConstant[0]  = World Matrix (transpose)

VertexShaderConstant[8]  = Total Matrix (transpose)

VertexShaderConstant[12] = Light Direction;

VertexShaderConstant[20] = (.5f,.5f,.5f,.5f)


Texture[0]          = normal map;

Texture[1]          = color map;
```

Note how simple the pixel shader is. Figure 4 shows an example diffuse per-pixel lighting image. The take-away from this is that with per-pixel diffuse lighting, it is easy to get good-looking results. All pixel shader cards support the **dp3** operator, so this technique is good to go on pixel shader hardware.

For previous generation cards, two primary fallbacks exist. The first fallback is to use the **D3DTOP_DOTPRODUCT3** fixed-function operator, which some of the better previous generation cards support, since this operator was first enabled in DirectX 6.0. Be sure to check the **D3DTEXOPCAPS_DOTPRODUCT3** capability bits for support of this multi-texture capability. Using **ValidateDevice()** is also a good idea. Below is the multi-texture setup (using the effects framework syntax) for a **D3DTOP_DOTPRODUCT3** fixed-function operation

```
ColorOp[0]   = DotProduct3;

ColorArg1[0] = Texture;
```

```
ColorArg2[0] = Diffuse;


ColorOp[1]   = Modulate;
ColorArg1[1] = Texture;
ColorArg2[1] = Current;


VertexShaderConstant[0]  = World Matrix (transpose)
VertexShaderConstant[8]  = Total Matrix (transpose)
VertexShaderConstant[12] = Light Direction;


Texture[0]   = normal map
Texture[1]   = color map;
```

Where colorop indicates the texture stage operation, and colorarg[n] indicates the texture stage arguments. MIP mapping and filtering need to be set as well, but I ignore these settings due to space considerations. Remember, the **D3DTOP_DOTPRODUCT3** operator in the fixed-function pipeline automatically applies the **_sat** and **_bx2** operations automatically, which means:

You must use biased art for the normal maps for **_bx2** to generate correct results.

The automatic use of **_sat** (clamping) means no signed result can be generated.

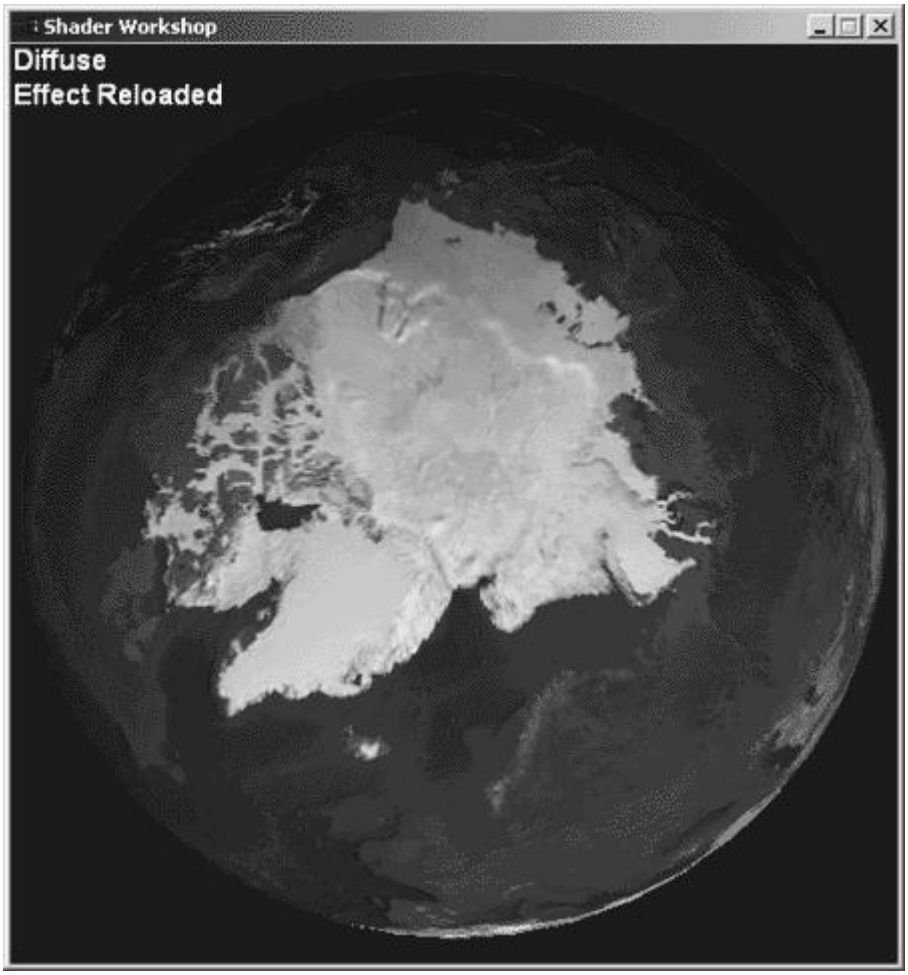


Figure 4. Diffuse per-pixel lighting

The second fallback is to use emboss bump mapping. The only hardware requirement is for a dual texture unit with a subtract operation, as shown by the presence of the **D3DTEXOPCAPS_SUBTRACT** capability bit. Again, whenever using the fixed-function multi-texture pipeline, it's a good idea to use **ValidateDevice()**. Emboss bump-mapping works by shifting a height map in the direction of the light vector, and subtracting this from the base map. The results can be very convincing, but can take quite an effort to fine tune. A vertex shader fragment for a typical emboss operation is shown below:

```
//v0 = position

//v3 = normal (also the w vector)

//v7 = texture coordinate

//v8 = tangent (u vector)
```

```
vs.1.1          // for emboss
m4x4 oPos, v0,c08      // generate output position

//diffuse
m3x3 r3, v8, c0       // transform tan to world space
m3x3 r5, v3, c0      // transform norm to world space

mul r0,r3.zxyw,r5.yzxw  // cross prod to generate binormal
mad r4,r3.yzxw,r5.zxyw,-r0

dp3 r6.x,r3,c12      // tangent space light in r6
dp3 r6.y,r4,c12
//   dp3 r6.z,r5,c12  don't need this
//                    -only x and y shifts matter

// set up the texture, based on light direction:
mul r1.xy,  r6.xy, -c24.xy
mov oT0.xy, v7.xy          // copy the base height map
add oT1.xy, v7.xy,  r1.xy    // offset the normal height map

// simple dot product to get global darkening effects:
dp3 oD0.xyz,v3.xyz,c12.xyz
```

Next is the multi-texture state setup (again using the effects framework syntax) for a **D3DTOP_ADDSIGNED** fixed-function operation, using the complement input argument modifier flag:

```
ColorOp[0]   = SelectArg1;
ColorArg1[0] = Texture;

ColorOp[1]   = AddSigned;
```

```
ColorArg1[1] = Texture | Complement;

ColorArg2[1] = Current;


VertexShaderConstant[0]  = World Matrix (transpose)

VertexShaderConstant[8]  = Total Matrix (transpose)

VertexShaderConstant[12] = Light Direction;

VertexShaderConstant[24] = Offset Constant


Texture[0]   = base height map;

Texture[1]   = normal height map;
```

Again, MIP mapping and filtering need to be set as well, but I ignore these settings due to space considerations. In conclusion, emboss-style bump mapping can be used for a diffuse fallback technique for hardware that does not support the dot product multi-texture operator. This includes most DirectX 6.x generation cards—a huge percentage of the installed base. For ideal results, this technique requires modified artwork, and textures should be brightened on load. An alternative is to use a **D3DTOP_MODULATE2X** operator to scale the result up, which has the visual effect of brightening. Also note that filtering can be applied to this technique more easily than to normal maps, so this technique may result in a better appearance than the dot product multi-texture technique, even on hardware that supports dot product operations.

Per-pixel specular is similar to diffuse, but requires a pixel shader. Instead of the light direction, an interpolated half-angle vector H is used; which is computed in the vertex shader. In the pixel shader, the H is dotted with the pixel normal, and then raised to a pre-determined power. The specular result is added to the other passes. Also, remember that there is only one output value of a pixel shader, in **r0**, so make sure to add the specular result into **r0**.

One question you may be asking at this point: How is exponentiation performed? Two techniques are used, multiply-based and table-based. One is simpler and is acceptable for small exponents, and one is more work, but looks nicer for higher exponents. Both techniques I cover here use the following renderstates (again using effects framework syntax):

```
VertexShaderConstant[0] = World Matrix

VertexShaderConstant[8] = Total Matrix
```

```
VertexShaderConstant[12] = Light Direction

VertexShaderConstant[14] = Camera Position (World)

VertexShaderConstant[33] = (.5f,.5f,.5f,.5f)


Texture[0]   = normal map

Texture[1]   = color map
```

Now, it's time to examine the two pixel shader specular techniques, starting with the multiply-based exponentiation technique. Below is a vertex shader fragment that shows (in addition to the diffuse actions of calculating the light direction in tangent space, biasing it for the per-pixel dot product, and setting up the texture coordinates for the pixel shader) the actions of computing the half vector, using the view direction and the light direction, and scaling/biasing it for the dot product calculations used by multiply-based exponentiation:

```
vs.1.1

//tranform position

m4x4 oPos,v0,c4


//tsb generation

m3x3 r3,v8,c0            //gen normal

m3x3 r5,v3,c0            //gen tangent


//gen binormal via Cross product

mul r0,-r3.zxyw,r5.yzxw;

mad r4,-r3.yzxw,r5.zxyw,-r0;


//specular

m4x4 r2,v0,c0            //transform position

//get a vector toward the camera

add r2,-r2,c24
```

```
dp3 r11.x,  r2.xyz,r2.xyz   //load the square into r11
rsq r11.xyz,r11.x         //get the inverse of the square
mul r2.xyz, r2.xyz,r11.xyz   //multiply, r0 = -(camera vector)
add r2.xyz,r2.xyz,-c16      //get half angle

//normalize
dp3 r11.x,r2.xyz,r2.xyz      //load the square into r1
rsq r11.xyz,r11.x         //get the inverse of the square
mul r2.xyz,r2.xyz,r11.xyz   //multiply, r2 = HalfAngle

//transform the half angle vector
dp3 r8.x,r3,r2
dp3 r8.y,r4,r2
dp3 r8.z,r5,r2

//half-angle in oD1
mad oD1.xyz,   r8.xyz,c20,c20   //mutiply by a half, add half

//tex coords
mov oT0.xy,   v7.xy
mov oT1.xy,   v7.xy
```

Below is a pixel shader fragment that shows per-pixel specular, using the multiply-based exponentiation technique:

```
ps.1.1            // pow2 by multiplies
tex t0            // color map
tex t1            // normal map

//  specular lighting dotproduct
```

```
dp3_sat r0,t1_bx2,v1_bx2   // bias t0 and v1 (light color)


mul r1,r0,r0            //  2nd power
mul r0,r1,r1            //  4th power
mul r1,r0,r0            //  8th power
mul r0,r1,r1            // 16th power!


//assemble final color
mul r0,r0,t0         // modulate by color map
```

Note the use of the **_bx2** modifier. Again, this is to enable the input data to be processed as a signed quantity, while reserving dynamic range (used by the specular calculation) before overflow clamping that can occur on implementations limited to the range [-1, 1]. Figure 5 shows an image generated using multiply-based specular exponentiation. Notice the banding in the highlight. This is due to loss of precision in the calculations, since each result channel is only 8-bits. Here is where higher precision texture formats and higher precision for internal calculations will increase image quality. In conclusion, multiply-based per-pixel specular is easy to implement, but can involve precision problems, so don't try to use the technique for powers greater than 16. On graphics chips with higher precision, this may not be an issue.

The next technique is table-lookup based specular exponentiation. The example used here performs this operation with a 3x2 table. The texture is used as a table of exponents, storing the function y = pow (x).

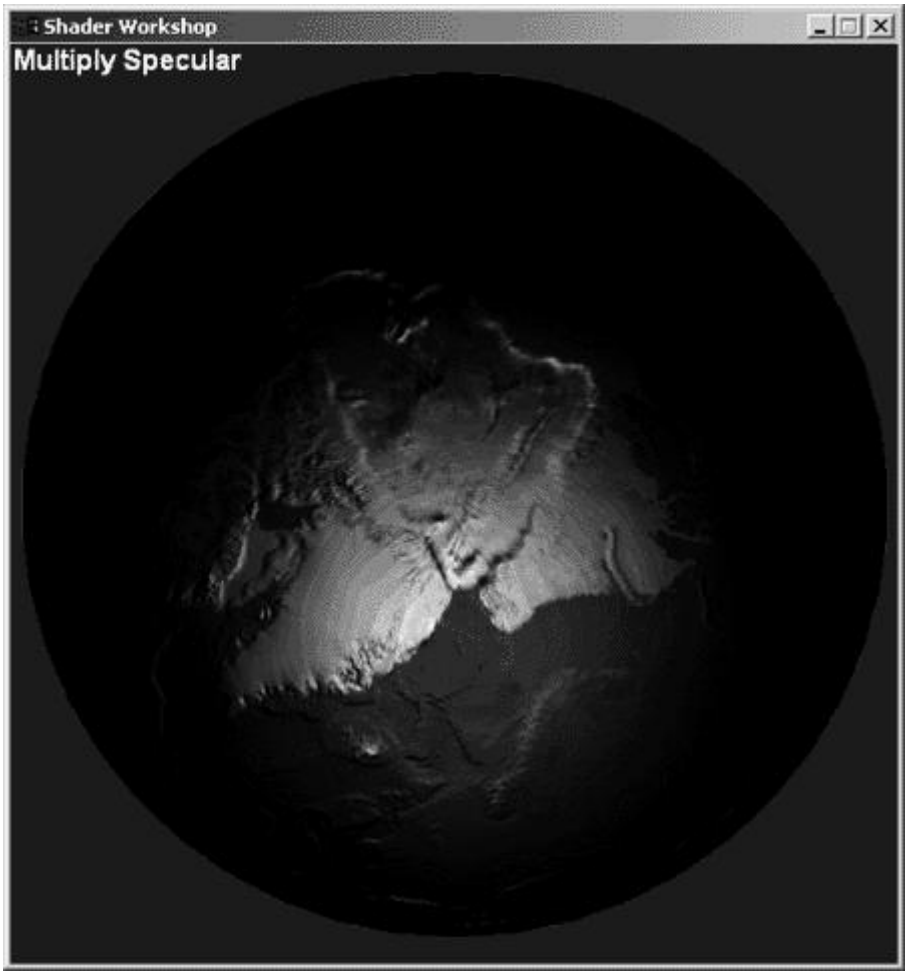


Figure 5. Multiply-based specular exponentiation

This technique also uses the dependent texture read capability of the `texm3x2tex` instruction. Note the 3x2 multiply is also 2 dot products, so this technique can do specular and diffuse, or two light sources, simultaneously. Below is the vertex shader for this technique:

```
vs.1.1
//tranform position
m4x4 oPos,v0,c4

//tsb generation
m3x3 r3,v8,c0             //transform normal
m3x3 r5,v3,c0             //and tangent
```

```
//Cross product
mul r0,-r3.zxyw,r5.yzxw;
mad r4,-r3.yzxw,r5.zxyw,-r0;

//specular
m4x4 r2,v0,c0            //transform position

//get a vector toward the camera
add r2,-r2,c24

dp3 r11.x,r2.xyz,r2.xyz   //load the square into r11
rsq r11.xyz,r11.x         //get the inverse of the square
mul r2.xyz,r2.xyz,r11.xyz //multiply, r0 = -(camera vector)

add r2.xyz,r2.xyz,-c16    //get half angle

//normalize
dp3 r11.x,r2.xyz,r2.xyz   //load the square into r1
rsq r11.xyz,r11.x         //get the inverse of the square
mul r2.xyz,r2.xyz,r11.xyz //multiply, r2 = HalfAngle

//transform the half angle vector
dp3 r8.x,r3,r2
dp3 r8.y,r4,r2
dp3 r8.z,r5,r2

//tex coords
mov oT0.xy, v7.xy         //coord to samp normal from
```

```
mov oT1.xyz,r8            //Not a tex coord, but half

mov oT2.xyz,r8            //angle

mov oT3.xy, v7.xy
```

The table-lookup vertex shader is identical to the multiply-based vertex shader through the half-angle normalization calculation. From there, this technique uses texture coordinates to pass down vectors, as well as true texture coordinates used to index the color and normal maps. The half-angle is passed down as texture coordinates for stage 2, then texture coordinates for stage 1 are used to pass down the light direction, and texture coordinates for stage 0 and stage 3 are used for the normal and color maps respectively.

Next is shown the 3x2 table-lookup specular lighting pixel shader:

```
ps.1.1          // exponentiation by table lookup


//  texcoord t1      // the diffuse light direction

//  texcoord t2      // half-angle vector

//  texture at stage t2 is a table lookup function


tex t0          // sample the normal map

texm3x2pad t1, t0_bx2     // 1st row of mult, 1st dotproduct=u

texm3x2tex t2, t0_bx2     // 2nd row of mult, 2nd dotproduct=v


//asemble final color

mov r0,t2         // use (u,v) above to get intensity

mul r0,r0,t3            //blend terms
```

The key detail of this shader is the use of the **texm3x2tex** instruction with the **texm3x2pad** instruction to perform a dependent read. The **texm3x2pad** instruction is used in conjunction with other texture address operators to perform 3x2 matrix multiplies. It is used to represent the stage where only the texture coordinate is used, so there is no texture bound at this stage (in this shader, that is, **t1**). The input argument, **t0**, should still be specified.

The **texm3x2pad** instruction takes the specified input color (**t0** here) and multiplies that by the subsequent stages' (**t1** here) texture coordinates (u, v, and w) to calculate the 1st row of the multiply (a dot product) to generate a u coordinate. Then the **texm3x2tex** instruction takes the specified input color (**t0** again) and the texture coordinates of the stage specified (**t2** here) to calculate the second row of the multiply (again a dot product) to generate the v coordinate. Lastly, this stage's texture (**t2** here) can be used to sample the texture by a dependent read at (u, v) to produce the final color.

That leaves the question of how to generate the lookup-table texture. Using D3DX, one could use the following code fragment to generate the table-lookup texture:

```
void LightEval(D3DXVECTOR4 *col, D3DXVECTOR2 *input
                          D3DXVECTOR2 *sampSize, void *pfPower)
{
    float fPower = (float) pow(input->y,*((float*)pfPower));
    col->x = fPower;
    col->y = fPower;
    col->z = fPower;
    col->w = input->x;
}

D3DXCreateTexture(m_pd3dDevice, 256,256, 0,0,
   D3DFMT_A8R8G8B8, D3DPOOL_MANAGED, &pLightMap100);

float fPower = 100;
D3DXFillTexture(m_pLightMap100,LightEval,&fPower);
```

Figure 6 below shows the results of table-lookup specular exponentiation. A careful examination does indeed show less banding effects and a better-looking image. This technique will support exponents greater than 100, which is important for some visual effects. There is higher precision in the texture read unit, and the table-lookup texture ends up nicely filtered, so that the banding is reduced to a tolerable level. Note that other functions can be used besides exponents.

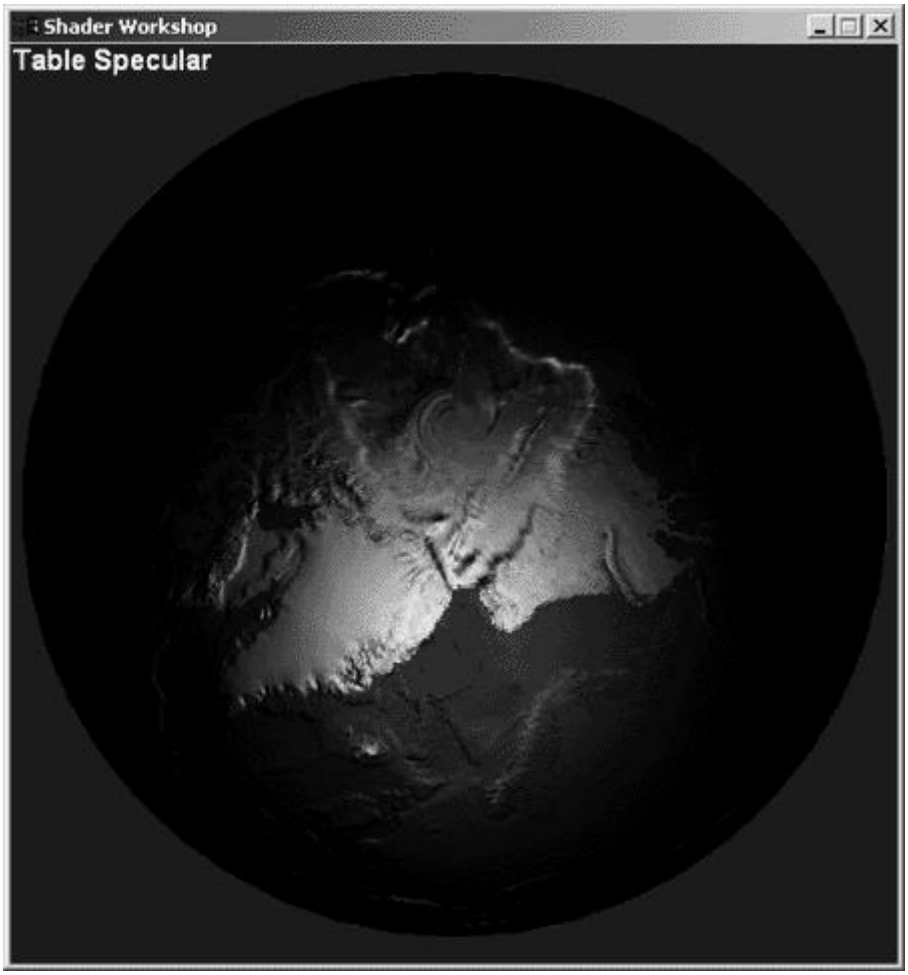


Figure 6. Table lookup specular exponentiation

The fallback using the fixed-function multi-texture pipeline is an analog of emboss for the specular term, where the light map highlight is subtracted from the height-field normal map. Specular highlights do need to be manually positioned, but that's not hard to do in the vertex shader. Then composite the values in the pixel shader using subtract, and add the result as the per-pixel specular term. Figure 7 shows a diagram of the two textures and how the subtract result gives the desired effect.

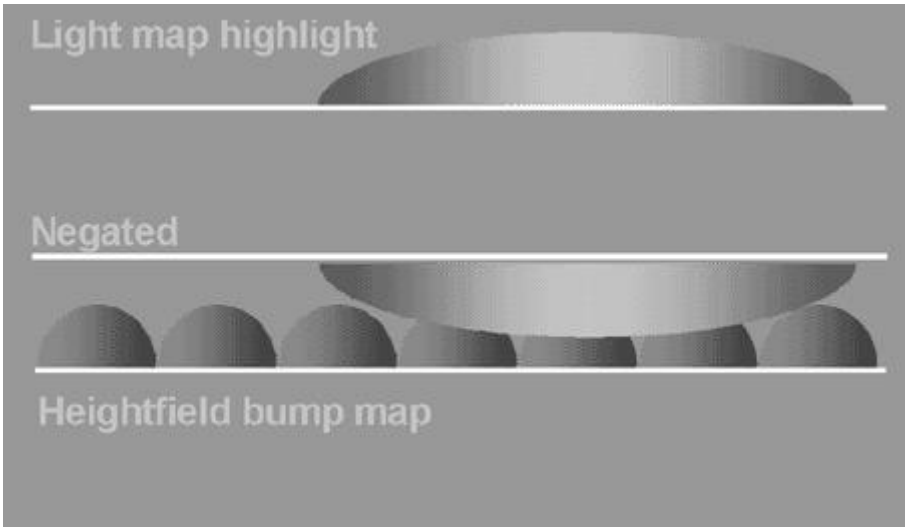


Figure 7. Specular emboss fallback diagram

In the interests of space, I am skipping the implementation of the specular fallback; you should be getting the idea. That's the end of the coverage of the standard diffuse and specular per-pixel lighting models, and how to realize them in pixel shaders. Now it's on to "do-it-yourself" lighting models, where I show how to develop your own, custom lighting models and implement them.

Custom Per-Pixel Lighting

With this summary of techniques for the legacy—that is, with standard lighting models behind us—it's now time to consider custom, "do-it-yourself" lighting models, or DIY lighting. The beauty of pixel shader hardware is that it frees the developer from being stuck with the legacy lighting model, and opens up a brave new world of custom lighting. Remember, the legacy lighting models are just a set of equations someone came up with at some point in time that did a reasonable job of modeling the physics of lighting. There is no magic there. So do not be restricted by the basic lighting model. While diffuse and specular are easy to calculate, and generally produce acceptable results, they do have the drawback of producing a result that looks overly familiar. When the goal of advanced lighting is to stand out, looking the same is a major drawback.

So what does DIY lighting mean? It means not being bound by the basic lighting model, and instead using creativity and an understanding of the principles of the basic lighting tasks. Using the SIGGRAPH literature, hardware vendor Web sites, and information about game engine rendering approaches, there are huge amounts of material available as a guide to interesting alternative approaches to the basic diffuse and specular lighting model.

The key is to understand the process of the basic lighting model, and to use a process in developing a lighting model. This allows a consistent basis upon which to evaluate lighting ideas. This discussion focuses on local lighting, since there isn't space to cover attenuation effects, atmospheric effects, or anything else.

The process of the basic lighting model is based on the physics of lighting. Figure 8 shows the physical model the calculations are intended to reproduce. There is a light source generating light, a surface geometry upon which the incident light energy is received, and upon which reflected light departs (called the incident and reflectant geometry), and a camera to record the lighting.

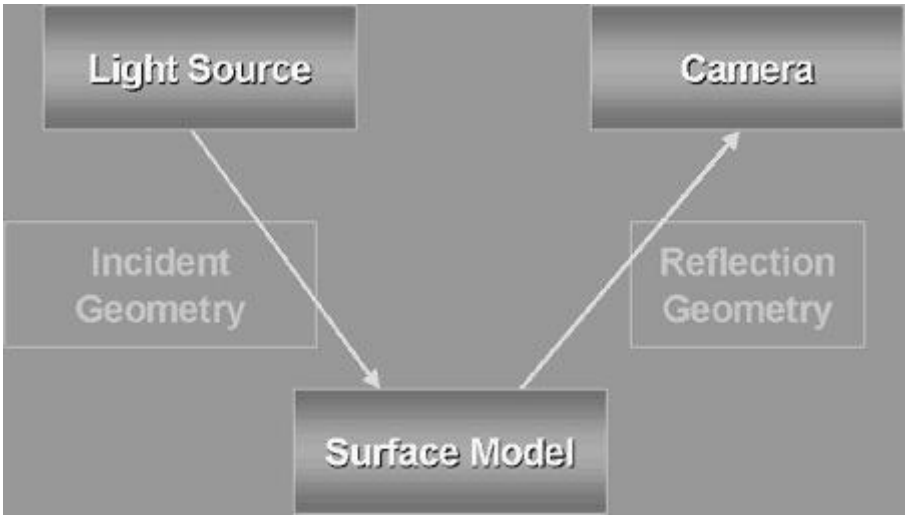


Figure 8. Lighting Process

Incident angle and reflection angle play a key part in the lighting calculation. The geometries' interaction with light is also controlled by a surface model that defines the physical properties of the surface that lighting is incident on and reflected from. There are many different elements that can be used in surface models. The most common model is the Lambert model for the diffuse light term. The Lambert model defines microfacets across the surface where intensity depends on input energy and the area of the microfacets perpendicular to the light direction. Lambert diffuse is easily calculated using the dot product. Figure 9 illustrates the Lambert model. The key point is that there is a physical basis for this model, from which the calculations can be identified.

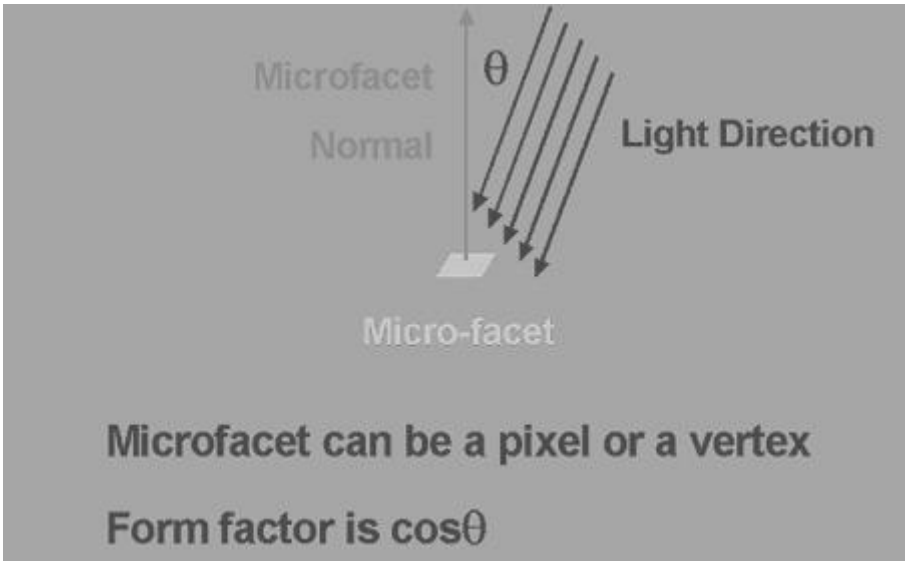


Figure 9. Lambert Model

With an understanding of how this works in the basic lighting model, lets begin an exploration of 'DIY" Lighting. In a simple "DIY" evaluation process, first, there is an identification stage. Here you analyze what the key effects you wish to generate in each scene are. Then these are prioritized. Once this has been accomplished, that data can be used to characterize the lighting environment for each scene and each effect.

How many light sources are needed for these effects? What shape are they? What shadows, and what reflections result? Now with the key lighting effects identified and characterized, algorithms to implement the effect need to be generated. In those algorithms, terms that make important contributions to the final result are kept, and terms that don't contribute significantly to image quality are dropped. Experimentation is usually need to determine this, but in some cases, running limit calculations can lead to an expectation that a terms contribution is meaningful or meaningless. These pieces can often be defined and tested independently, but at some point the individual parts will need to be blended together for a "final" evaluation of the model. Another part of the process consists of determining whether a piece of the calculation is object-based (author time calculation), vertex-based (vertex shader time calculation), or pixel-based (pixel shader time calculation).

Finally, understanding the range of values that serve as input to a model, and understanding the expected range of output values, can lead to finding equivalencies substitutions in making calculations, where the substitute calculation is simpler conceptually, simpler in cost, or simply good enough. This is an important point, to not be hide-bound by convention, and instead keep an open mathematical mind to take advantage of whatever one can.

So, with that in mind, it's time to walk through the process for a simple yet effective 'DIY" lighting model. Here, local lighting is the focus, attenuation effects, atmospheric effects, and others are not considered. The model shown is a variation of an area or distributed lighting effect. It's primarily a diffuse effect, but with a more interesting ambient term. It provides "shadow detail," and shows that an object is grounded in a real-world scene by providing for an influence factor from the world itself. This is in essence an image-based lighting technique.

Figure 10 is a Lightwave image showing a target scene for this model, where how close to the target the model comes provides an evaluation of the model. Many ray-trace tools support a similar model, where the renderer integrates at each point, casts rays from the microfacet pixel to all points of the hemisphere, and accumulates color from ray intersections from other parts of the same object and the environment map image. This can take hours. The goal here is to get a significant percent of the same quality in real-time.

Figure 10. Target image for evaluating DIY model

The distributed lighting model used here, shown in Figure 11, works on large area light sources; there is no single direction vector. Energy is based on a number of directions, the fraction of possible directions that can illuminate the microfacet. When using this model for outdoor scenes, there are two light sources, the sun and the sky. The sun is a directional light source, throwing sharp shadows. The sky is an omni-directional light source, throwing soft shadows. It's useful to consider the area light source as an enclosing

hemisphere, and then the lighting of objects inside the scene reduces to considering what possible percent of the hemisphere can shine on the object.

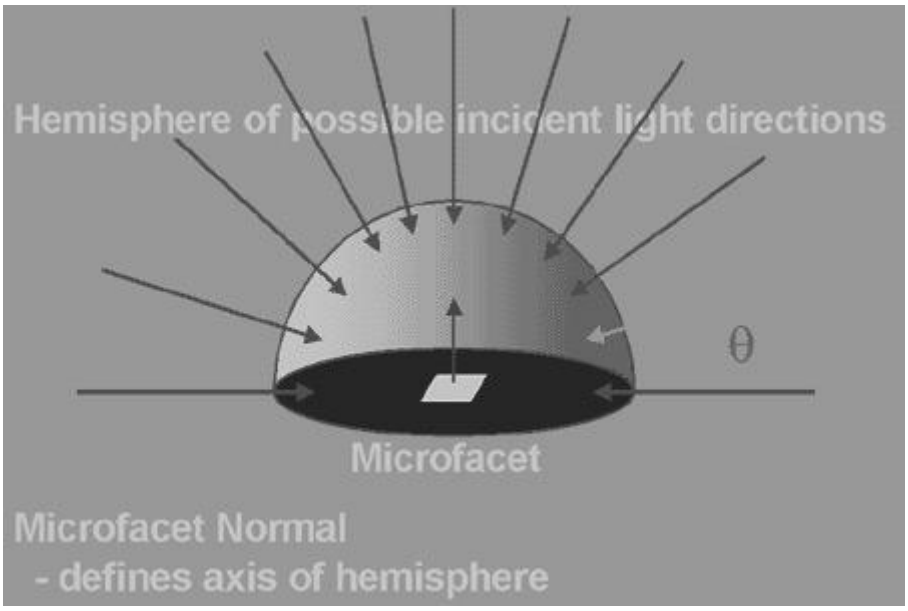


Figure 11. Distributed lighting model

Figures 12, 13, and 14 illustrate this for a cube, a hemisphere, and a sphere. It's pretty easy to see how objects in scenes that use this model are lit. Figure 12 contains a cube, and the lighting intensity is highest on the top face, and gradually decreases down the side faces. Figure 13 contains a hemisphere, and the lighting intensity is greatest at the top polar region, decreasing down to the equator. Figure 14 contains a sphere, and lighting intensity is again greatest at the top polar region, and decreases towards the bottom polar region.

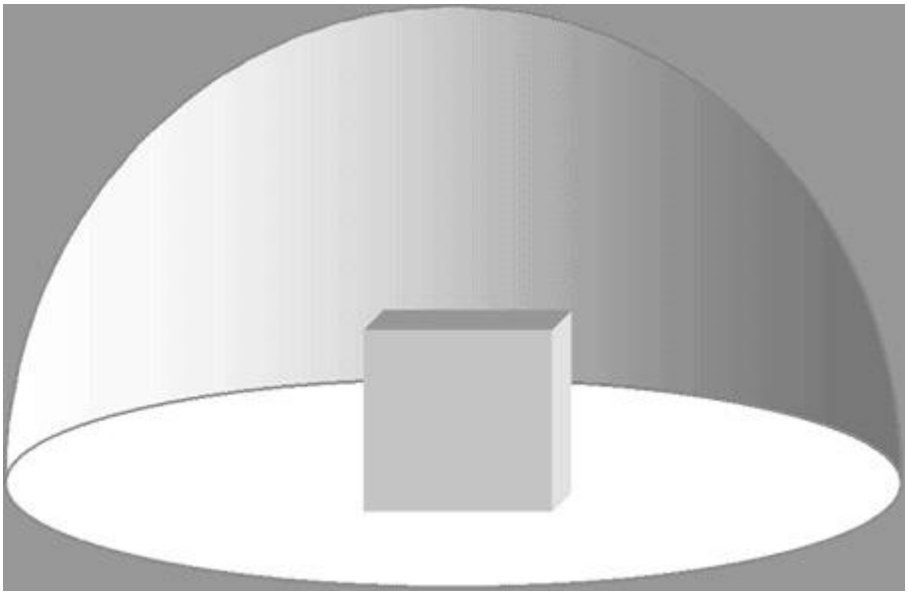


Figure 12. "Hemisphere" lighting for a cube

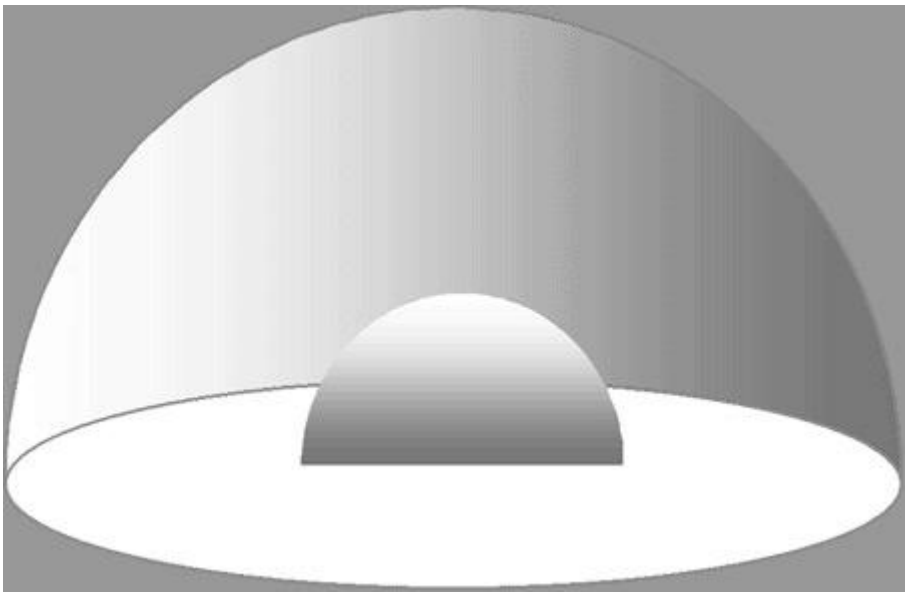


Figure 13. "Hemisphere" lighting for a hemisphere

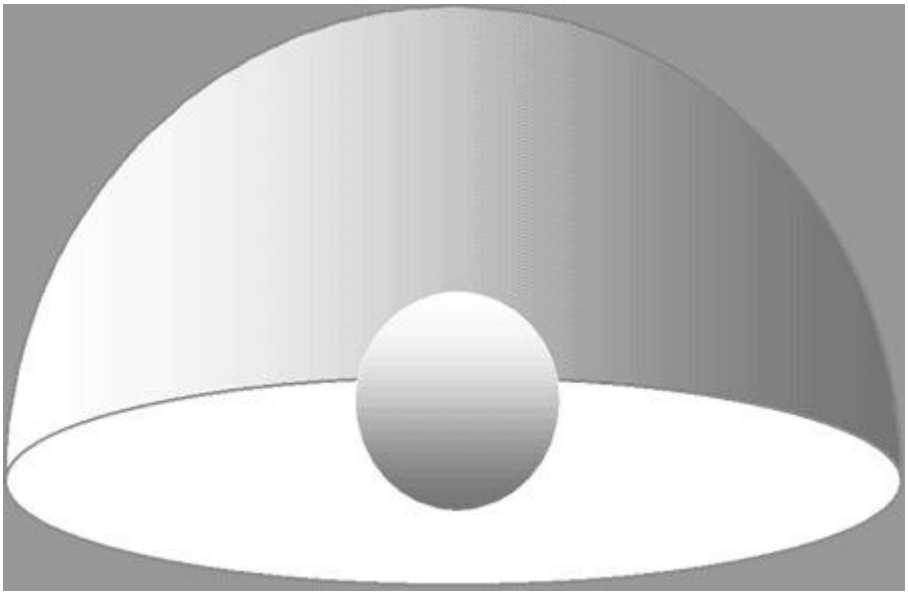


Figure 14. "Hemisphere" lighting for a sphere

Effectively, the light near the plane of the microfacet contributes less energy, so we can use the form factor cos(q) term to scale energy down. Integrating L = 1/p S cos(q) dd for this models' irradiance term, the light source is the far field. Integrating the environment map to get that term is the usual technique. This will work even on DirectX 7-class hardware.

A cube map and its corresponding integral are shown in Figure 15. Notice that the environment map integral is mostly two colors, sky color and ground color. The exact integral calculation is two slow for interactive applications, indicating an authoring time process would be required, and that the integral could not change during the game. That's less than ideal, but can anything be done about that?

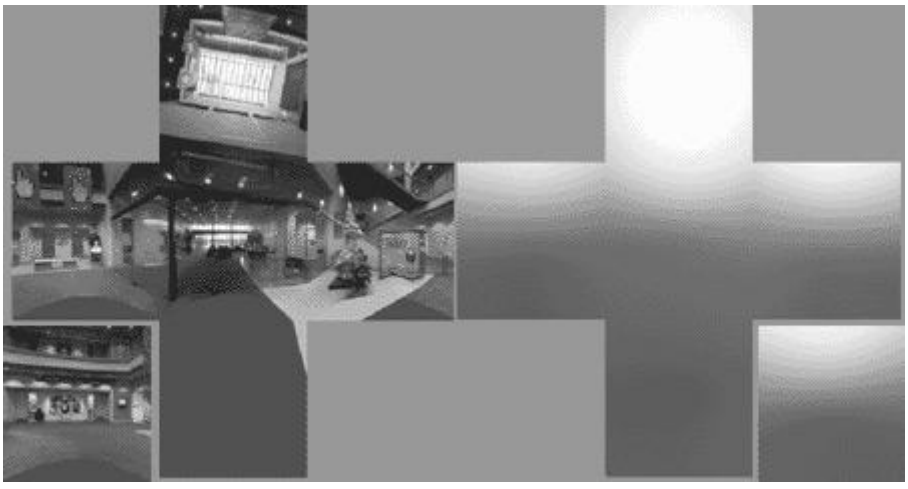


Figure 15. Cubemap and its integral

Let's consider this 2-hemisphere model, where our current understanding of the model is as a calculation with only two important terms—a sky term and a ground term. Figure 16 shows this 2-hemisphere model, and Figure 17 contains a process block diagram.

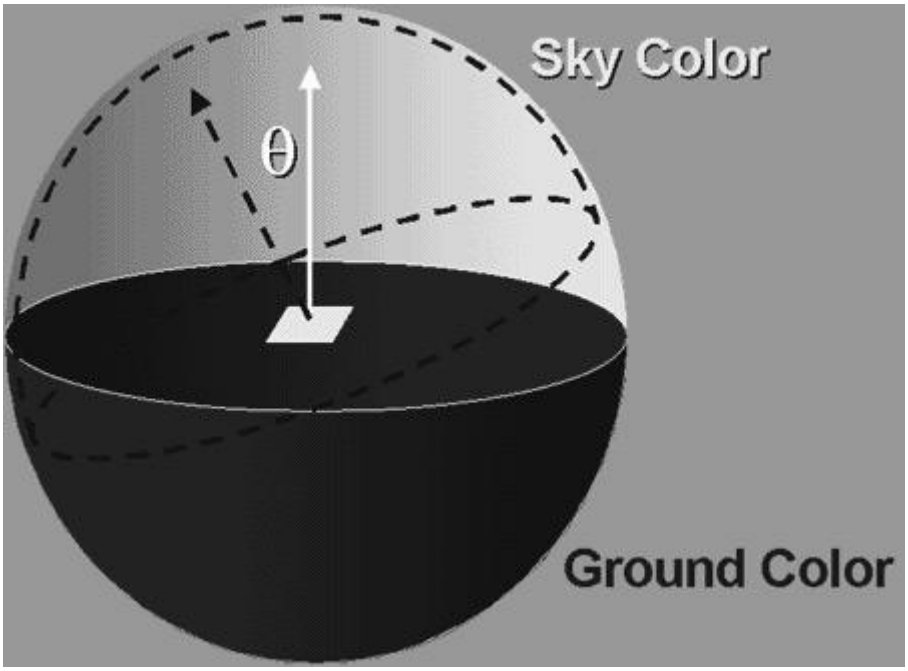


Figure 16. 2-Hemisphere model

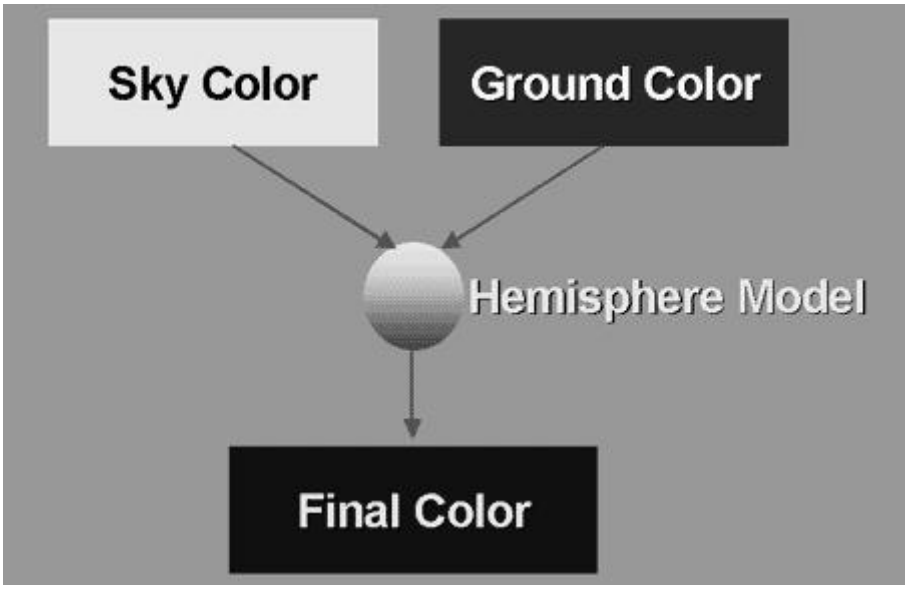


Figure 17. 2-Hemisphere model elements

The actual integral is:

color = a*Skycolor+(1-a)*GroundColor

Where

a = 1-0.5*sin(q) for q<90
a = 0.5*sin(q) for q > 90

Or, if instead of that, the simpler form

a = 0.5+0.5*cos(q)

is used. The resulting simplified integral versus the actual integral is shown in Figure 18. Notice the shape of the curve is the same, but mirrored, and similar amounts of light and dark regions appear below both curves. There is a general equivalency, even if it's not exact.

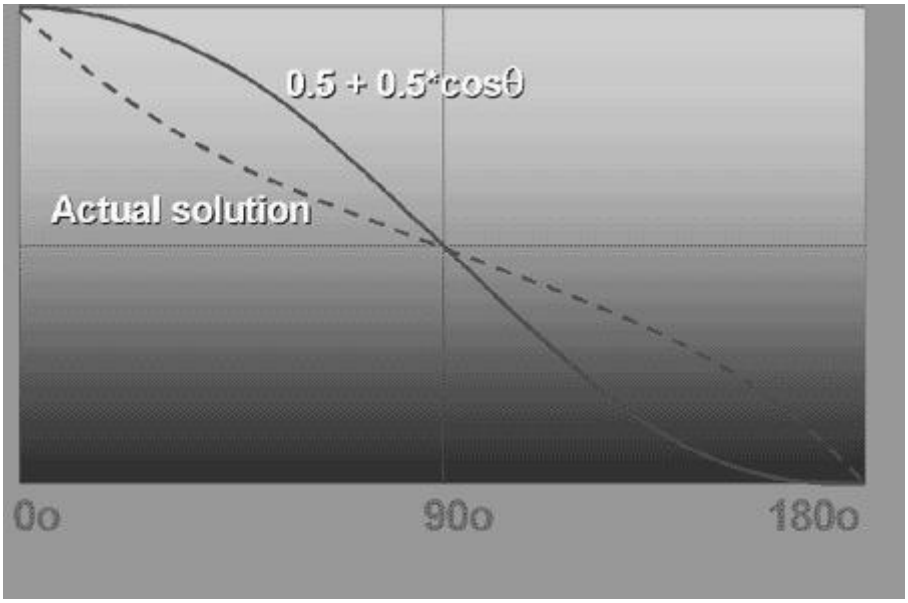


Figure 18. Integral comparison

Herein lies one of the secrets of the shader game: It's okay to substitute similar calculations if they are generally equivalent, because at real-time frame rates the small delta in image quality is usually unnoticeable. If this simplification provides enough of a performance gain, it can be preferred. In this case, the simplification takes 2 clocks, since it uses a **`dp3`** and **`mad`**. While it's not visually identical (this solution provides more bump detail along the equator and less bump detail facing the light) it's good enough to produce the desired effect at a significant gain in performance because, in this case, the actual calculation was too slow to do in real-time versus this simplification with its 2-clock cost. That's a huge win both in terms of clocks and in terms of grasping the essence and beauty of DIY lighting and the shader calculation gestalt.

The 2-term calculation boils down to what percent of incident energy has what color. The far field hemisphere is composed of 2 colors, sky, and ground, in a simple proportion. Even with the environment simplified to two colors like this, the model still allows for dynamic updates, like when a car enters a canyon or a tunnel, and then leaves it; so the pavement or ground color would change, or the sky or roof color would change. The hemisphere implementation can also be done either per-vertex or per-pixel.

The per-vertex implementation can pass the hemisphere axis in as a light direction and use the standard tangent space basis vertex shader that transforms the hemi axis into tangent space. A vertex shader implementation would look something like:

```
vs.1.1          // vertex hemisphere shader


m4x4 oPos,v0,c8   // transform position
m3x3 r0, v3,c0   // transform normal


dp3 r0,r0,c40   // c40 is sky vector
mov r1,c33      // c33 is .5f in all channels
mad r0,r0,c33,r1   // bias operation


mov r1,c42      // c42 is ground color
sub r1,c41,r1      // c41 is sky color
mad r0,r1,r0,c42   // lerp operation


//c44 = (1,1,1,1)
sub r1,c44,v7.zzz   // v7.zzz = occlusion term
mul r0,r0,r1
mul oD0,r0,c43
```

A per-pixel fragment implementation would look like:

```
//   v0.rgb is hemi axis in tangent space
// v0.a is occlusion ratio from vshader
```

```
tex t0          // normal map
tex t1          // base texture


dp3_d2_sat r0,v0_bx2,t0_bx2   // dot normal with hemi axis
add r0,r0,c5      // map into range, not _sat
lrp r0,r0,c1,c2
mul r0,r0,t1      // modulate base texture
```

With that in mind, how does this look? Figure 19 shows the 2-term approach. While this is interesting, there are issues here. The combination of two colors is getting there, but there is obviously too much light in certain areas, like the eye sockets, the nostrils, and behind the teeth.

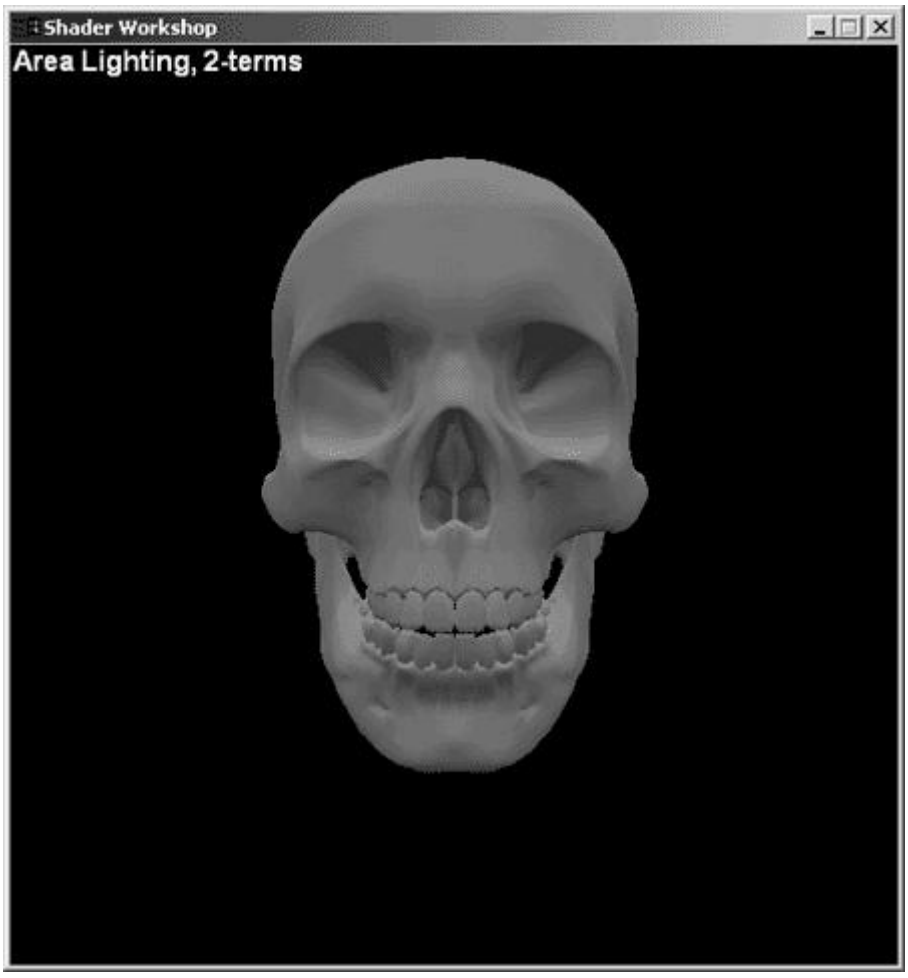


Figure 19. 2-term DIY image

Time to refine the DIY model. How is the model refined? With the addition of another term, of course. What term would that be? Well, the first attempt did not take object self-shadowing into account, and that is the basis of the resulting image being brighter in areas where it shouldn't be. So adding an occlusion term is necessary.

Figure 20 shows a block diagram of this updated DIY lighting model. This calculation can be done vertex-to-vertex by firing a hemisphere of rays from each normal, storing the result as vertex colors; or pixel-to-pixel by firing rays in a height field and storing the result in the alpha channel of a normal map; or both by firing rays from vertices and pixels and storing the result in a texture map.

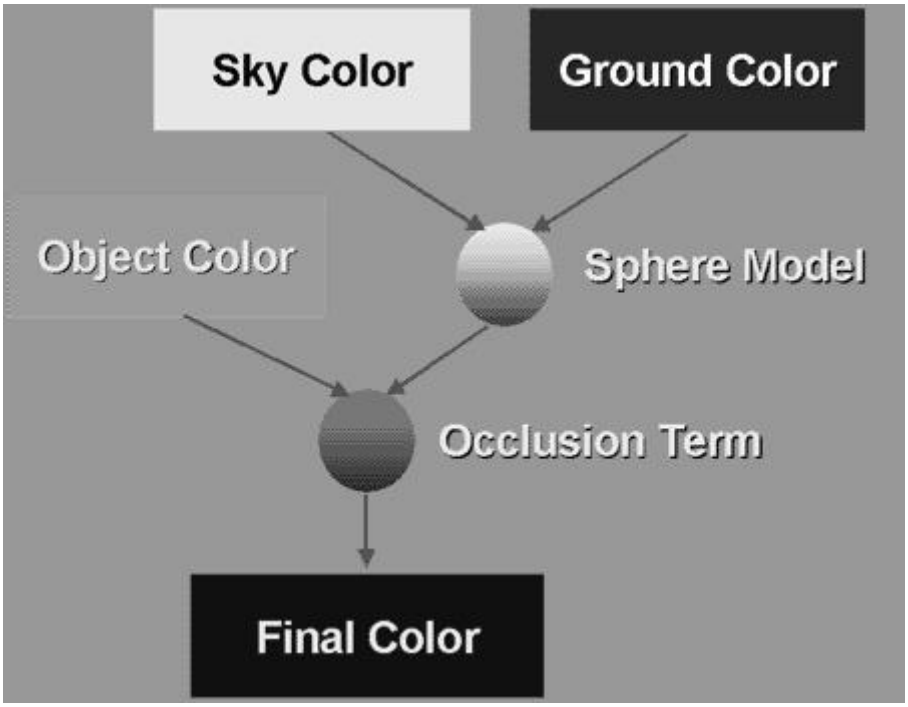


Figure 20. Updated DIY model elements

The sample shown here used an offline rendering process to calculate this occlusion data. Considering the vertex-to-vertex case, the calculation answers the question, "How much do adjacent polygons shadow each other?" The result can be stored in a vertex attribute, and should handle object level effects. Note that looking only at neighbor vertices might be okay.

Considering the pixel-to-pixel case, the calculation similarly answers the question, "How much do adjacent pixels shadow each other?" An example is a bump-mapped earth, where the geometry provides no self-occlusion, since a sphere is everywhere convex. This means all occlusion can be done in a bump map. Figure 21 shows the resulting image with a 3-term DIY lighting model. This is a big improvement, with the problem areas of the eye sockets, nostrils, and interior of the mouth looking much better.

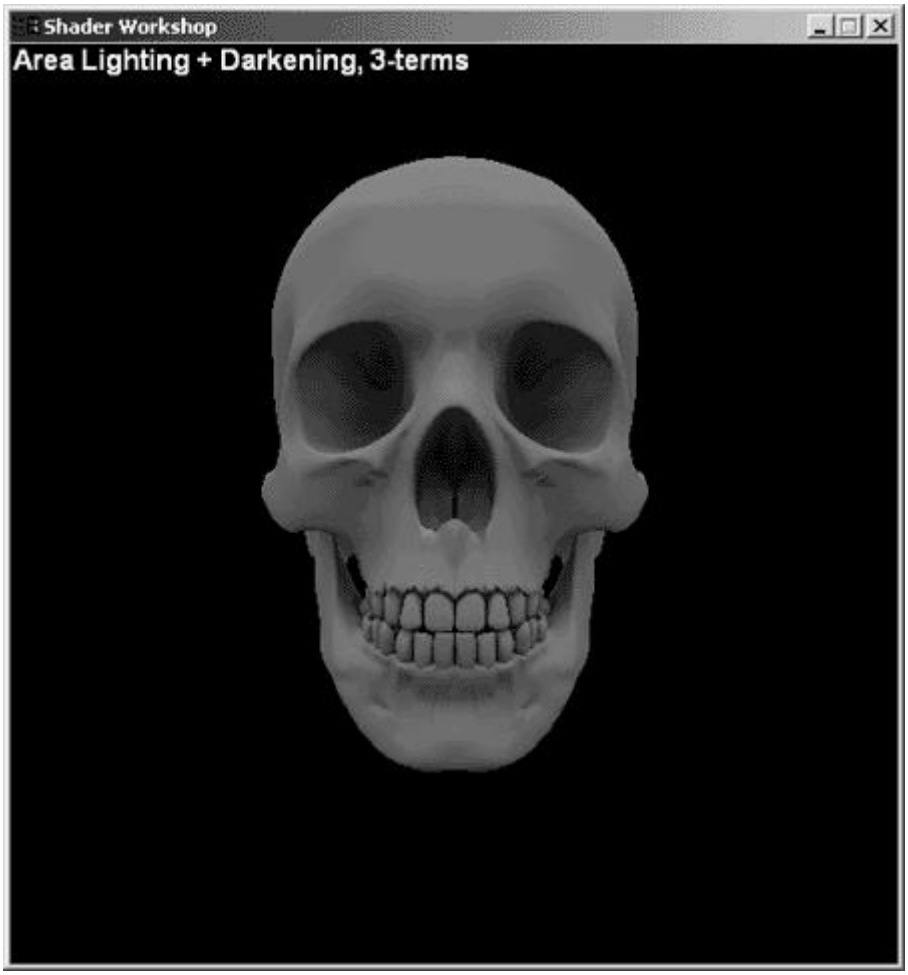


Figure 21. 3-term DIY lighting model image

Finally, combine this with a directional light, as shown in Figure 22, and an amazingly realistic image results for such a relatively simple lighting model.

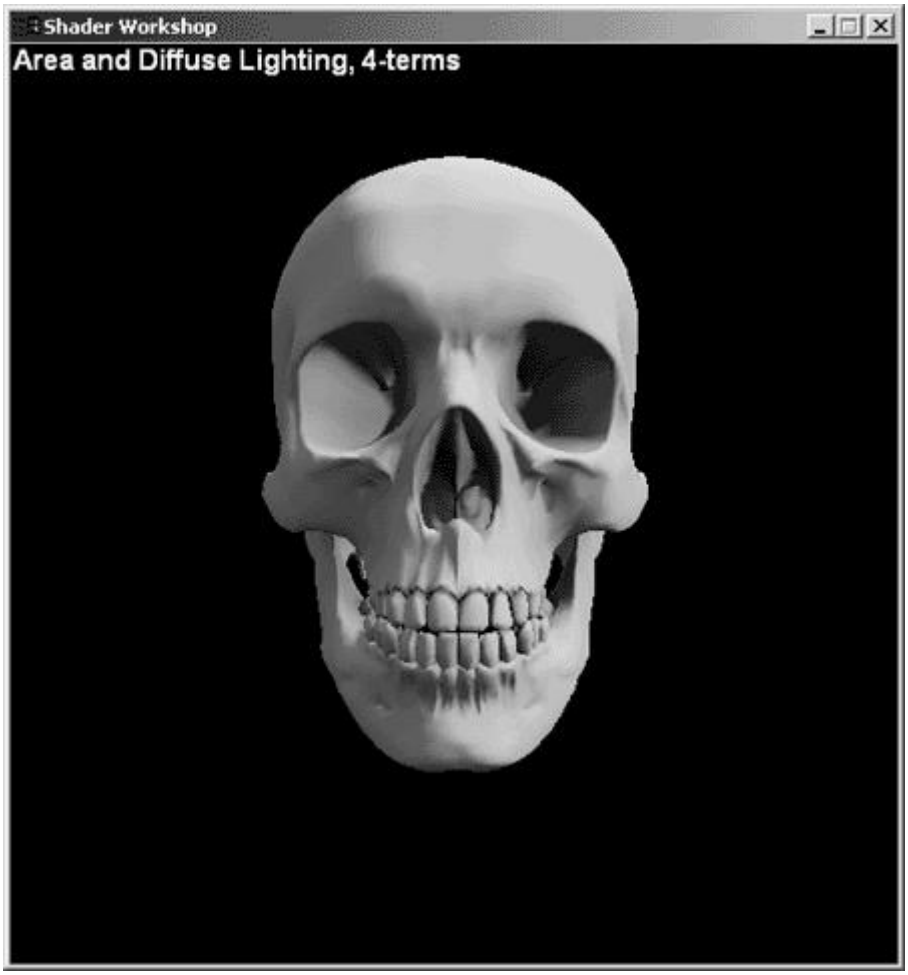


Figure 22. 3-term DIY lighting model image

Now clearly, the process of defining, evaluating, and refining a DIY lighting model is iterative, but it isn't all that difficult, as this article has shown. It's just a matter of clearly thinking through the process of what sort of illumination is necessary to get the desired effect, and working through the iterations until it looks good enough.

There are two lessons here: First, understanding the tangent space basis, so you can correctly perform DOT3 based diffuse and specular lighting, and secondly, understanding how to perform "do-it-yourself" lighting. Each of these lessons is important, and DIY lighting builds on the knowledge of and correct usage of the tangent space basis, but the real kicker is understanding the "lighting black box" and functional equivalencies, so that you feel comfortable using your own approaches to lighting when you want something that appears just a little different.

Last Word

I'd like to acknowledge the help of **Chas Boyd, Dan Baker, Tony Cox,** and **Mike Burrows** (Microsoft) in producing this column. Thanks to Lightwave for the Lightwave images, and Viewpoint Datalabs for the models.

Your feedback is welcome. Feel free to drop me a line at the address below with your comments, questions, topic ideas, or links to your own variations on topics the column covers. Please, though, don't expect an individual reply or send me support questions.

Remember, Microsoft maintains active mailing lists as forums for like-minded developers to share information:

DirectXAV for audio and video issues at http://DISCUSS.MICROSOFT.COM/archives/DIRECTXAV.html.

DirectXDev for graphics, networking, and input at http://DISCUSS.MICROSOFT.COM/archives/DIRECTXDEV.html.

Philip Taylor is the PM for the DirectX SDK, Managed DirectX, the Windows® XP 3D screensavers, and a few more bits and bobs. Previously at Microsoft he was senior engineer in the DirectX evangelism group for DirectX 3.0 to DirectX 8.0, and helped many game ISVs with DirectX. He has worked with DirectX since the first public beta of the GameSDK (DirectX 1.0), and, once upon a time, actually shipped DirectX 2.0 games. In his spare time, he can be found lurking on many 3-D graphics programming mailing lists and Usenet newsgroups. You can reach him at msdn@microsoft.com.

Chapter 6

OpenGL 2.0

Randi Rost

COURSE 17: STATE-OF-THE-ART IN SHADING HARDWARE

CHAPTER 6: THE OPENGL SHADING LANGUAGE

Randi J. Rost

3Dlabs, Inc.

SIGGRAPH 2002 Course Notes

5-April-2002

6.1 Introduction

For just about as long as there has been graphics hardware, there has been programmable graphics hardware. Over the years, building flexibility into graphics hardware designs has been a necessary way of life for hardware developers. Graphics APIs continue to evolve, and since a hardware design can take two years or more from start to finish, the only way to guarantee a hardware product that can support the then-current graphics APIs at its release is to build in some degree of programmability from the very beginning.

Until recently, the realm of programming graphics hardware belonged to just a few people, mainly researchers and graphics hardware driver developers. Research into programmable graphics hardware has been taking place for many years, but the point of this research has not been to produce viable hardware and software for application developers and end users. The graphics hardware driver developers have focused on the immediate task of providing support for the important graphics APIs of the time: PHIGS, PEX, Iris GL, OpenGL, Direct3D, and so on. Until recently, none of these APIs exposed the programmability of the underlying hardware, so application developers have been forced into using the fixed functionality provided by a traditional graphics APIs.

Hardware companies have not exposed the programmable underpinnings of their products because there is a high cost of educating and supporting customers to use low-level, device-specific interfaces and because these interfaces typically change quite radically with each new generation of graphics hardware. Application developers who use such a device-specific interface to a piece of graphics hardware face the daunting task of updating their software for each new generation of hardware that comes along. And forget about supporting the application on hardware from multiple vendors!

As we move into the 21st century, some of these fundamental tenets about graphics hardware are being challenged. Application developers are pushing the envelope as never before, and demanding a variety of new features in hardware in order to create more and more sophisticated on-screen effects. As a result, new graphics hardware designs are more programmable than ever before. Standard graphics APIs have been challenged to keep up with the pace of hardware innovation. For OpenGL, the result has been a spate of extensions to the core API as hardware vendors struggle to support a range of interesting new features that their customers are demanding.

So we are standing today at a crossroads for the graphics industry. A paradigm shift is occurring that is taking us from the world of rigid, fixed-functionality graphics hardware and graphics APIs to a brave new world where the visual processing unit, or VPU (i.e., graphics hardware) is as flexible and as important as the central processing unit, or CPU. The VPU will be optimized for processing dynamic media such as 3D graphics and video. Highly parallel processing of floating point data will be the primary task for VPUs, and the flexibility of the VPU will mean that it can also be used to process data other than a stream of traditional graphics commands. Applications can take advantage of the capabilities of both the CPU and the VPU, utilizing the strengths of each to optimally perform the task at hand.

This paper talks about how we are attempting to expose the programmability of graphics hardware to the leading cross-platform 3D graphics API: OpenGL. This effort is called OpenGL 2.0, and it brings a lot of new and exciting features to OpenGL while retaining compatibility so that existing applications will continue to run. For the purpose of this paper and this course, we will concentrate on presenting the OpenGL Shading Language, a high-level shading language built into OpenGL that allows applications to take total control over per-vertex and per-fragment calculations.

At the time of this writing, the features of OpenGL 2.0 are solidifying and implementation efforts are beginning. It is not currently possible to execute all of the OpenGL 2.0 shaders described in this paper and get screen shots of them, but they all have been compiled with our OpenGL Shading Language compiler. By the time SIGGRAPH 2002 rolls around, we expect to be able to run all of these shaders and many more. You are invited to check out the 3Dlabs web site at http://www.3dlabs.com and follow the link to our OpenGL 2.0 page. There you will find a version of this paper updated for SIGGRAPH, the latest versions of all of the OpenGL 2.0 white papers, and information about obtaining and working with the 3Dlabs OpenGL 2.0 implementation.

6.2 Additions to OpenGL

The effort to define OpenGL began in the early 1990's and the first version of the OpenGL specification was approved in 1992. Since that time, great strides have been made in both system architecture and graphics hardware architecture. Today, graphics hardware is changing rapidly from the model of a fixed function state machine (as originally targeted by OpenGL) to that of a highly flexible and programmable machine. It is becoming much more difficult for hardware developers to support new features demanded by application developers than to just expose the underlying programmability and let application developers do things themselves.

The OpenGL Shading Language

The OpenGL 2.0 effort adds two programmable processors to the fixed geometry processing pipeline previously defined by OpenGL. These two processors are called the *vertex processor* and the *fragment processor*. The vertex processor is capable of executing a program called a *vertex shader* on each and every vertex that is presented to it. The fragment processor is capable of executing a program called a *fragment shader* on each and every fragment that results from primitive rasterization.

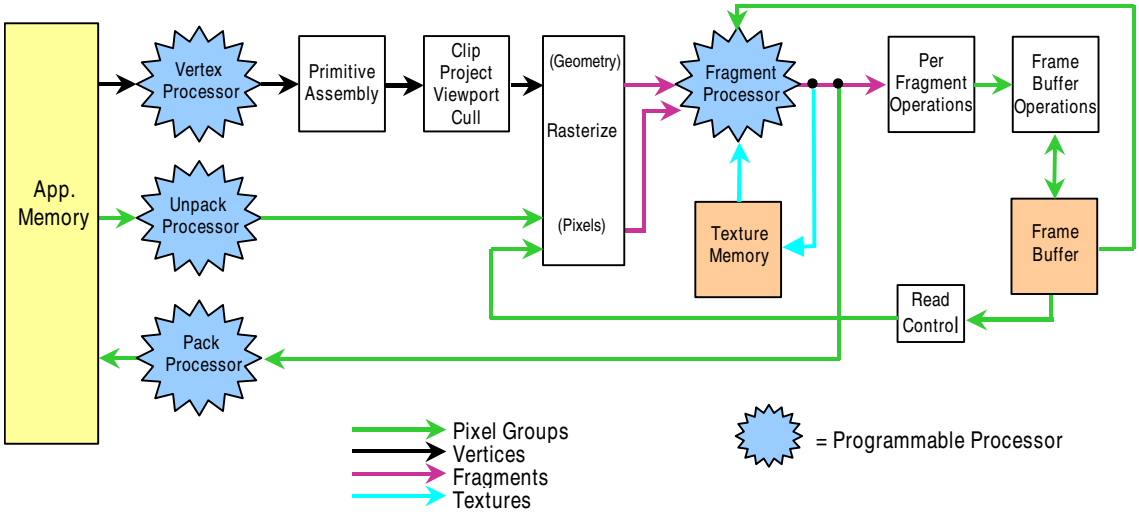
The high level language that is defined as part of the OpenGL 2.0 effort is called the *OpenGL Shading Language*. Some of the significant features of this language are:

- It is integrated intimately with OpenGL 1.3
- It allows incremental replacement of the OpenGL 1.3 fixed functionality
- It is a high level language, accessible to all applications
- It is based on C and C++ and includes support for scalar, vector and matrix types
- It virtualizes most of the graphics hardware pipeline resources
- The same language is used for both vertex and fragment shaders
- OpenGL state is directly accessible from the language
- It has a rich feature set, including numerous built-in functions for common operations
- It is hardware independent and implementable on a variety of graphics hardware architectures
- It is (will be) a standard part of OpenGL

The OpenGL 2.0 environment

The diagram below is a simplified version of the logical diagram for OpenGL 2.0. It is based on the logical diagram called The OpenGL Machine published with the OpenGL Reference Manual (blue book). It illustrates some of the primary differences between OpenGL 2.0 and OpenGL 1.3. It should not be interpreted as an implementation diagram, it is a logical diagram that illustrates the state blocks, processing modules, and data paths in OpenGL 2.0.

The OpenGL 2.0 state machine is very similar to that of OpenGL 1.3, but several areas of fixed functionality have been augmented by programmable processors. This diagram shows only the programmable units and not the fixed functionality that they replace.



The newly defined programmable units are shown as blue (shaded) stars. State blocks that remain the same as in OpenGL 1.3 are shown in white rectangles. To simplify the diagram, some of the OpenGL 1.3 functionality that remains the same in OpenGL 2.0 is grouped together in a single white rectangle (e.g., clip, project, viewport, cull). Memory that is under control of the application is shown on the left, and memory that is controlled by OpenGL is shown in orange (shaded) boxes. The arrows represent the primary flow of data. For immediate mode geometry commands, vertex data starts out in application memory and is sent down what is referred to as the geometry pipeline, generating pixels that are ultimately written into frame buffer memory. The application sends pixel data to the unpack processor and after rasterization, it is written to either the frame buffer or texture memory. The fragment processor can read texture memory during subsequent rendering operations. The application can read back pixels from the frame buffer, optionally passing them through the fragment processor for processing, and then packing them in application memory under the control of the pack processor.

Vertex Processor

The vertex processor is a programmable unit that processes vertex data in the OpenGL pipeline. When the vertex processor is active, the following fixed functionality of OpenGL is disabled:

- Vertex transformation
- Normal transformation and normalization
- Texture coordinate generation
- Texture coordinate transformation
- Lighting
- Color material application
- Clamping of colors

Even when the vertex processor is active, the following OpenGL functions are still performed by the fixed functionality within OpenGL. Most of these operations work on several vertices at a time and involve topological knowledge:

- Perspective divide and viewport mapping
- Primitive assembly
- Frustum and user clipping
- Backface culling
- Two-sided lighting selection
- Polymode processing
- Polygon offset

Vertex shaders that intend to perform computations similar to the fixed functionality of OpenGL 1.3 are responsible for writing the code for all of the functionality in the first list above. For instance, it is not possible to use the existing fixed functionality to perform the vertex and normal transformation but have a specialized lighting function. The shader must be written to perform all three functions. Existing OpenGL state is available to a vertex shader in the form of predefined variables (e.g., gl_ModelViewMatrix). Any OpenGL state used by the shader is automatically tracked and made available to the shader. This automatic state tracking mechanism allows applications to use existing OpenGL state commands for state management and have the current values of such state automatically available for use in the vertex shader.

The vertex processor operates on one vertex at a time. This programmable unit does not have the capability of reading from texture memory or the frame buffer. The design of this unit is focused on the functionality needed to transform and light a single vertex. The vertex shader must compute the homogeneous position of the coordinate, and it may also compute color, texture coordinates, and other arbitrary values to be passed to the fragment processor. The output of the vertex processor is sent through subsequent stages of processing that are defined exactly the same as they are for OpenGL 1.3: primitive assembly, user clipping, frustum clipping, perspective projection, viewport mapping, polygon offset, polygon mode, shade mode, and culling.

After all this processing, vertex data arrives at the rasterization stage. OpenGL defines rasterization as consisting of two parts. The first part of rasterization is to determine which squares of an integer grid in window coordinates the primitive occupies. This portion of the rasterization process remains unchanged for OpenGL 2.0. The second part of rasterization, assigning color, depth, and stencil values to each square, is almost completely replaced in OpenGL 2.0 as described in the next section.

Fragment Processor

The fragment processor (defined in the *OpenGL 2.0 Shading Language* white paper) is designed to operate on the fragments produced by the rasterization stage. In OpenGL, a fragment is defined as a grid square (i.e., an x/y position) and its assigned color, depth, and texture coordinates. In OpenGL 1.3, there are very strict rules for how the fragment's associated data can be modified. Additional rules have been added through extensions, exposing the greater flexibility of current hardware. In OpenGL 2.0, a programmable unit (the fragment processor) is defined to allow applications to use a high-level programming language to express how the fragment's associated data is computed. Mechanisms for creating, compiling, linking, using, and deleting programs for this processor are the same as for the vertex processor.

When the fragment processor is active, the following fixed functionality relating to geometry processing is replaced:

- Interpolation of vertex data across the primitive
- Texture access
- Texture application
- Fog
- Color sum

and the following fixed functionality relating to pixel processing is also replaced:

- Pixel zoom
- Scale and bias
- Color table lookup
- Convolution
- Color matrix

Even when the fragment processor is active, the following OpenGL functions are still performed by the fixed functionality within OpenGL:

- Shading model
- Coverage
- Pixel ownership test
- Scissor
- Stipple
- Alpha test
- Depth test
- Stencil test
- Alpha blending
- Logical ops
- Dithering
- Plane masking
- Pixel packing/unpacking

Related OpenGL state is also automatically tracked if used by the fragment shader. With the exception of a few built-in functions, the language used to program the fragment processor is nearly identical to the language used to program the vertex processor. A fragment shader can change a fragment's depth, color, and stencil value, but not its x/y position. To support parallelism at the fragment processing level, fragment shaders are written in a way that expresses the computation required for a single fragment, and access to neighboring fragments is not allowed. A fragment shader is free to read multiple values from a single texture, or multiple values from multiple textures. It is not allowed to directly write either texture memory or frame buffer memory, but it can directly read frame buffer memory at the current pixel location.

The OpenGL 1.3 parameters for texture maps are carried forward to OpenGL 2.0 and continue to define the behavior of the filtering operation, borders, and wrapping. These operations are applied when a texture is accessed. The fragment shader is free to use the resulting texel however it chooses. It is possible for a fragment shader to read multiple values from a texture and perform a custom filtering operation. It is also

possible to use a 1D texture to perform a lookup table operation. In both cases the texture should have its texture parameters set so that nearest neighbor filtering is applied on the texture access operations.

For each fragment, the fragment shader can compute color, depth, stencil, or one or more arbitrary data values. A fragment shader must compute at least one of these values. The color, depth, and stencil values will remain as computed by the previous stages in the OpenGL pipeline if the fragment shader does not modify them.

The results of the fragment shader are then sent on for further processing. The remainder of the OpenGL pipeline remains as defined in OpenGL 1.3. Fragments are submitted to coverage application, pixel ownership testing, scissor testing, alpha testing, stencil testing, depth testing, blending, dithering, logical operations, and masking before ultimately being written into the frame buffer..

Creating and compiling shaders

A shader is created using:

```
GLhandle glCreateShaderObject (GLenum shaderType)
```

The shader object is empty when it is created. The shaderType argument specifies the type of shader object to be created, and should be one of GL_VERTEX_SHADER or GL_FRAGMENT_SHADER. If the shader object is created successfully, a non-zero handle that can be used to reference it is returned.

Source code for the shader is specified with the commands:

```
void glLoadShader (GLhandle shaderID,
                   const GLuint nseg,
                   const GLubyte **seg)

void glAppendShader (GLhandle shaderID, const GLubyte *seg)
```

The glLoadShader command sets the source code for the specified shader object to the text strings in the *seg* array. If the object previously had source code loaded into it, it is completely replaced. The glAppendShader command adds the specified string onto the end of the existing source code for the shader. The *shaderID* argument selects the shader to be modified. The *seg* argument is an array of pointers to one or more null terminated character strings that make up the program. The number of strings in the array is given in *nseg*. The multiple strings interface provides:

- A way to organize common pieces of source code.
- A way to share prefix strings (analogous to header files) between programs as an aid to compatibility between the output of the vertex shader to the input of a fragment shader.
- A way of supplying external #define values to control the compilation process.
- A way for including user defined or third party library functions.
- Allows for each string to be treated as a compilation unit so functions can be made private, hence may provide better support for 'programming in the large' or libraries.

A shader is compiled with the command:

```
GLboolean glCompileShader (GLhandle shaderID)
```

This function returns TRUE if the shader was compiled without errors and is ready for use, and FALSE otherwise. A read-only string containing information about the compilation can be obtained with the command:

```
const GLubyte *glGetInfoLog (shaderID)
```

If a shader compiled successfully the info log will either be an empty string or it will contain information about the compilation. The info log is only useful during application development and an application should not expect different OpenGL implementations to produce identical descriptions of error.

Once a shader object is no longer needed, it can be deleted using one of the object deletion commands:

```
void glDeleteObject (GLhandle objectID)
void glDeleteObjects (GLuint numObjs, GLhandle *objectIDs)
```

Linking and using shaders

In order to use shaders, they must be attached to a program object and linked. A program object is created with the command:

```
GLhandle glCreateProgramObject ()
```

The program object is empty when it is created. If the program object is created successfully, a non-zero handle that can be used to reference it is returned. Program objects can be deleted just like shader object with the glDeleteObject and glDeleteObjects calls.

Shader objects are attached to a program object with the command:

```
GLboolean glAttachShaderObject (GLhandle programID, GLhandle shaderID)
```

This function returns TRUE if *programID* represents a valid program object and *shaderID* represents a valid shader object and the attach operation is successful, and FALSE otherwise. It is permissible to attach shader objects to program objects before source code has been loaded into the shader object, or before the shader object has been compiled. It is permissible to attach multiple shader objects of the same type to a single program object, and it is permissible to attach a shader object to more than one program object. The generic detach object function is used to detach a shader object from a program object.

```
void glDetachObject (GLhandle programID, GLhandle shaderID)
```

In order to use the shaders contained in a program object, the program object must be linked and the program object must be made part of the current rendering state. This is accomplished with the following commands:

```
GLboolean glLinkProgram (GLhandle programID)
GLboolean glUseProgramObject (GLhandle programID)
```

The link command attempts to create a valid executable program for the programmable environment of OpenGL 2.0, and it returns TRUE if successful, FALSE otherwise. Information about the link operation can be obtained by calling glGetInfoLog with the ID of the program object. If the program object was linked successfully the info log will either be an empty string or it will contain information about the link operation.

If a valid executable is created, the program object can be made part of the current rendering state by calling glUseProgramObject.

The link command attempts to create a valid program for each of the programmable processors in OpenGL 2.0 from the attached shaders. All of the attached shader objects of a particular type are linked together to form the executable program for the corresponding OpenGL 2.0 processor. For instance, all of the shader objects of type GL_VERTEX_SHADER are linked together to form a single executable for the vertex processor. If the program object does not contain any shader objects of a particular type, the processor corresponding to that shader type will be inactive and the corresponding fixed functionality of OpenGL will be used instead. (For instance, if a shader of type GL_VERTEX_SHADER is provided, but no shader of type GL_FRAGMENT_SHADER is provided, the vertex processor will be active but the fragment processor will not. Fixed functionality will be used to process fragments in this case.)

If the link operation is successful, the program object contains executable code that is made part of the rendering state when glUseProgramObject is called. glUseProgramObject will install the executable code as part of current rendering state and return TRUE if the specified program object contains valid executable code. It will return FALSE and leave the current rendering state unmodified if the specified program object does not contain valid executable code. If glUseProgramObject is called with a handle of 0, all of the OpenGL 2.0 processors are disabled and the OpenGL 1.3 fixed functionality will be used instead.

The program object that is currently in use can be obtained by calling glGetHandle with the parameter name GL_PROGRAM_OBJECT. While a program object is in use, applications are free to modify attached shader objects, compile attached shader objects, attach additional shader objects, and detach shader objects.

Specifying vertex attributes

Vertex data is passed to OpenGL in the same way as always. One way vertex data is passed to OpenGL is by calling glBegin, followed by some sequence of glColor/glNormal/glVertex/etc. A call to glEnd terminates this method of specifying vertex data.

These calls continue to work in the OpenGL 2.0 environment. As before, a call to glVertex indicates that the data for an individual vertex is complete and should be processed. However, if a valid vertex shader has been installed by calling glUseProgramObject, the vertex data will be processed by that vertex shader instead of by the usual fixed functionality of OpenGL. A vertex shader can access the standard types of vertex data passed to OpenGL using the following built-in variables:

```
attribute vec4 gl_Vertex;
attribute vec4 gl_Color;
attribute vec3 gl_Normal;
attribute vec4 gl_MultiTexCoord0; // glTexCoord also updates this.
attribute vec4 gl_MultiTexCoord1;
attribute vec4 gl_MultiTexCoord2;
attribute vec4 gl_MultiTexCoord3;
attribute vec4 gl_MultiTexCoord4;
attribute vec4 gl_MultiTexCoord5;
attribute vec4 gl_MultiTexCoord6;
attribute vec4 gl_MultiTexCoord7;
attribute vec4 gl_SecColor;
attribute vec3 gl_FrontMaterial[5];
attribute vec3 gl_BackMaterial[5];
```

OpenGL's vertex-at-a-time interface is simple and powerful, but on today's systems it is definitely not the highest performance way of rendering. Whenever possible, applications should use the vertex array interface instead. This interface allows you to store vertex data in arrays and set pointers to those arrays. Instead of drawing a vertex at a time, you can draw a whole set of primitives at a time. It is even possible that vertex arrays are stored in graphics memory to allow for maximum performance (see the OpenGL 2.0 white paper *Minimizing Data Movement and Memory Management*).

The vertex array interface also works the same way in OpenGL 2.0 as it did previously. When a vertex array is rendered, the vertex data is processed one vertex at a time, just like the vertex-at-a-time interface. If a vertex shader is active, each vertex will be processed by the vertex shader rather than by the fixed functionality of OpenGL.

However, the brave new world of programmability means that applications no longer need to be limited to the 22 standard attributes defined by OpenGL. There are many additional per-vertex attributes that applications would like to pass into a vertex shader. It is easy to imagine that applications will want to specify per-vertex data such as tangents, temperature, pressure, and who knows what else. How do we allow applications to pass "non-traditional" attributes to OpenGL and operate on them in vertex shaders?

Then answer is to imagine that the hardware has a small number of generic locations for passing in vertex attributes. Each location is numbered. An implementation that supports 16 attribute locations will have them numbered from 0 to 15. An application can pass a vertex attribute into any of the numbered slots using one of the following functions:

```
void glVertexAttribute{1234}{sfd} (GLuint location, T value)
void glVertexAttribute{1234}{sfd}v (GLuint location, T *value)
void glVertexAttribute4ub (GLuint index, T values)
void glVertexAttribute4ubv (GLuint index, T *values)
```

which load the given value(s) into the attribute indicated by *location*. The attributes follow the OpenGL rules for the substitution of default values for missing components. Signed integers are converted into the range -1.0…1.0 and unsigned integers are converted into the range 0.0…1.0. The above notation is a shorthand way of expressing a number of different entry points. The {1234} notation indicates that there is a separate version of the function for expressing data with 1, 2, 3, or 4 components. The {sfd} notation indicates that there is a separate version of the function for expressing data as a short, a float, or a double. The T indicates the data type for the data to be based and it corresponds to the selection of short, float, or double in the {sfd} notation. The v indicates that the function is a pointer to a "vector" (array) of data. So for example, the actual function that you would use to supply a three-component float as a vector would be defined as:

```
void glVertexAttribute3fv(GLuint location, GLfloat *value)
```

This solves the question of how user-defined vertex data is passed into OpenGL, but how do we access that data from within a vertex shader? We don't want to refer to these numbered locations in our shader since this is not very descriptive and is prone to errors. Instead, we bind a numbered location to a variable name that we will use in our vertex shader by using the following function:

```
void glBindAttributeLocation (GLhandle program,
                              GLuint location,
```

```
                                    const ubyte *name)
```

Using these functions, we can create a vertex shader that contains a variable named “temperature” that is used directly in the lighting calculations. We can decide that we want to pass per-vertex temperature values in attribute location 11, and set up the proper binding with the following line of code:

```
temperatureLoc = 11;
glBindAttributeLocation(myProgram, temperatureLoc, “temperature”);
```

Subsequently, we can call glVertexAttribute to pass a temperature value at every vertex in the following manner:

```
glVertexAttribute1f(temperatureLoc, temperature);
```

It is also possible to pass several values simultaneously using these new functions:

```
void glVertexAttributes{1234}{sfd}v (GLuint location,
                                     GLuint count,
                                     T *values)

void glVertexAttributes4ubv (uint location,
                             uint count,
                             T *values)
```

The glVertexAttribute* calls are all designed for use between glBegin and glEnd. As such, they offer replacements for the standard OpenGL calls such as glColor, glNormal, etc. But as we have already pointed out, this is not the best way to do things if graphics performance is a concern.

The vertex array interface has also been extended to include the notion of user-defined data. New array types GL_USER_ATTRIBUTE_ARRAY0 through GL_USER_ATTRIBUTE_ARRAY*n*-1 are defined, where *n* is the number of attribute locations supported by the implementation. These arrays can be used to pass user-defined data to the vertex array interface. In our example, the temperature values could be stored in the GL_USER_ATTRIBUTE_ARRAY11 array, and passed to OpenGL by calling glDrawArray. The glBindAttributeLocation function defined above would be used in the same way to bind vertex attribute location 11 to the variable named “temperature” in our vertex shader.

Specifying other attributes

As described in the previous section, attribute variables are used to provide per-vertex data to the shader. Data that is constant across the primitive being rendering can be specified by using uniform variables. Uniform variables declared within a shader can be loaded directly by the application. This gives applications the ability to provide any type of arbitrary data to a shader. Applications can modify these values as often as every primitive in order to modify the behavior of the shader (although performance may suffer if this is done).

The OpenGL Shading Language also defines a number of built-in variables that track OpenGL state. Applications can continue using OpenGL to manage state through existing OpenGL calls, and use these built-in uniform variables in custom shaders. Of course, if you want something that isn’t already supported

directly by OpenGL, it is a simple matter to define your own uniform variable and supply the value to your shader.

When a program object is made current, user-defined uniform variables have undefined values, and built-in uniform variables that track GL state are initialized to the current value of that GL state. Subsequent calls that modify a GL state value will cause the built-in uniform variable that tracks that state value to be updated as well. The following commands are used to load uniform parameters into the program object that is currently in use.

```
void glLoadUniform{1234}f (GLint uniformLoc, T value)
void glLoadUniform{1234}fv (GLint uniformLoc, T value)
void glLoadUniformInt(GLint uniformLoc, int value)
void glLoadUniformBool(GLint uniformLoc, int value)
void glLoadUniformArray{1234}fv (GLint uniformLoc,
                                 GLint start, GLuint count, T value)
void glLoadUniformMatrix{234}fv (GLint uniformLoc, T value)
void glLoadUniformMatrixArray{234}fv (GLint uniformLoc, GLint start,
                                      GLuint count, T value)
```

Application code for creating/compiling shaders

Now that we've covered the basics, let's look at some sample application code that will set things up for rendering with a custom vertex shader and a custom fragment shader. It is convenient to store OpenGL shaders in their own files for easier maintenance. The first thing we'll do is call a function to read each shader from a file and store it as a string:

```
//
// Read the source code
//
if (!readShader(fileName, EVertexShader, vertexShaderString, vSize))
    return 0;

if (!readShader(fileName, EFragmentShader, fragmentShaderString, fSize))
    return 0;
```

This results in a string called vertexShaderString containing our entire vertex shader, and a string called fragmentShaderString containing our entire fragment shader. Next, we need to create three OpenGL objects: a shader object that will be used to store and compile our vertex shader, a second shader object that will be used to store and compile our fragment shader, and a program object to which our shaders will be attached.

```
//
// Create shader and program objects. Note, you should
// really check if the handles returned are not null.
//
programObject        = glCreateProgramObject();
vertexShaderObject   = glCreateShaderObject(GL_VERTEX_SHADER);
fragmentShaderObject = glCreateShaderObject(GL_FRAGMENT_SHADER);
```

The strings that hold our two shaders can now be passed on to our two newly-created shader objects. There are two functions for doing this, and either one can be used.

```
//
// Hand the source to OpenGL. Use either load or append,
// it doesn't matter.
//
glLoadShader(vertexShaderObject, 1, &vertexShaderString);
glAppendShader(fragmentShaderObject, fragmentShaderString);
```

The shaders are ready to be compiled. For each shader, we call glCompileShader and then call glGetInfoLog in order to see what transpired. glCompileShader will return TRUE if it succeeded and FALSE otherwise. Regardless of whether the compilation suceeded or failed, we print out what was contained in the info log for the shader. If the compilation was unsuccessful, this log will have information about the compilation errors. If the compilation was successful, this log may still have useful information that would help us improve the shader in some way. You would typically only check the info log during application development, or if you're running a shader for the first time on a new platform. We're going to bail out if the compilation of either shader failed.

```
//
// Compile the vertex and fragment shader, and print out
// the compiler log file.
//
compiled = glCompileShader(vertexShaderObject);
pInfoLog = glGetInfoLog(vertexShaderObject);
printf("%s\n\n", pInfoLog);

compiled &= glCompileShader(fragmentShaderObject);
pInfoLog = glGetInfoLog(fragmentShaderObject);
printf("%s\n\n", pInfoLog);

if (!compiled)
    return 0;
```

At this point the shaders have been compiled successfully, and we're almost ready to try them out. The shader objects are attached to our program object so that they can be linked.

```
//
// Populate the program object with the two compiled shaders
//
glAttachShaderObject(programObject, vertexShaderObject);
glAttachShaderObject(programObject, fragmentShaderObject);
```

Our two shaders are linked together to form an executable by calling glLinkProgram. Again, we look at the info log of the program object regardless of whether the link succeeded or failed. There may be useful information for us if we've never tried this shader before. If we make it to the end of this code, we have a valid program object that can be made part of current state simply by calling glUseProgramObject.

```
//
// Link the whole thing together and print out the linker log file
//
linked = glLinkProgram(programObject);
pInfoLog = glGetInfoLog(programObject);
printf("%s\n\n", pInfoLog);
```

```
  if (compiled && linked)
      return programObject;
  else
      return 0;
```

Application code for installing and using a shader

Each shader is going to be a little bit different. Each vertex shader may use a different set of attributes, different uniforms, attributes may be bound to different locations, etc. In our demo program, we just have an "install" function for each set of shaders that we want to install and use. The following example code is used to make the wood shader from section 6.7 part of current state. In this case, "installing" consists of calling glUseProgramObject on the program object that was previously created to contain the wood shader and then loading the uniform variables with the values that will be used for this execution of the wood shader.

```
  GLhandle installWoodShader()
  {
  //
  // Creates some 3D wood grain with diffuse lighting
  //
  GLhandle woodProgramObject;

  woodProgramObject = installShader("wood");

  glUseProgramObject(woodProgramObject);
  {
      float colorSpread[3] = { 0.3, 0.15, 0.0 };
      float lightPosition[3] = { 0.0, 0.0, 4.0 };
      float darkColor[3] = { 0.6, 0.3, 0.1};
      float ringSize = 1.0;
      float scale = 2.0;

      glLoadUniform1fv(glGetUniformLocation(woodProgramObject,
             "RingSize"), &ringSize);
      glLoadUniform1fv(glGetUniformLocation(woodProgramObject,
             "Scale"), &scale);
      glLoadUniform3fv(glGetUniformLocation(woodProgramObject,
             "LightPosition"), &lightPosition[0]);
      glLoadUniform3fv(glGetUniformLocation(woodProgramObject,
             "DarkColor"), &darkColor[0]);
      glLoadUniform3fv(glGetUniformLocation(woodProgramObject,
             "ColorSpread"),&colorSpread[0]);
  }

  return woodProgramObject;
}
```

6.3 Overview of the OpenGL Shading Language

We've talked all about how to create, load, compile, link, use, and get data into an OpenGL shader. But we haven't yet talked about the programming language for writing those shaders. For the rest of this paper, we'll be looking at example shaders and seeing how things work in the OpenGL 2.0 environment.

The OpenGL Shading Language has its roots in C and has features similar to RenderMan and other shading languages. It has a rich set of types, including vectors and matrices. An extensive set of built-in functions operates just as easily on vectors as on scalars. The language includes support for loops, subroutine calls, and conditional expressions. It is assumed that hardware vendors will be able to use compiler technology to translate this high level language into machine code that will execute to within a few percent of the performance of hand-written machine code.

A thorough description of the OpenGL Shading Language can be found in the white paper *OpenGL 2.0 Shading Language*, available at the 3Dlabs web site (http://www.3dlabs.com). In this paper, we focus more on programming examples, but here is a brief summary of this language.

- The language is high level and the same language is used for both vertex and fragment shaders.
- It is based on C and C++ and uses many of the same syntax and semantic rules.
- It naturally supports vector operations as these are inherent to many graphics algorithms.
- There are no obvious limits on a shader's size or complexity.

Data types

The OpenGL Shading Language supports the following data types:

bool	a conditional type, taking on values of true or false
int	a 16-bit signed integer
float	a single floating point scalar
vec2	a two component floating point vector
vec3	a three component floating point vector
vec4	a four component floating point vector
mat2	a 2x2 floating point matrix
mat3	a 3x3 floating point matrix
mat4	a 4x4 floating point matrix

These data types are modeled after the data types in the C programming language. Unlike C, there are no user defined structures or pointers.

Constructors

The constructor syntax from C++ is used to make variables of the correct type before assignment or during an expression. A constructed value is created by setting its components to a sequence of comma-separated values enclosed in parentheses. Data type conversion is performed as necessary. If there is a single value within parentheses, this single value is used to initialize the constructed value. If there are multiple values, they will be assigned in order, from left to right, to the components of the constructed value. Once a constructed item has values assigned to each of its components, any extra values in the list will be ignored.

All the variable types can have one (and only one) qualifier before the type keyword. If no qualifier is present then the variable is just like any variable defined in C and can be read and written as expected.

Type qualifiers

Named constants can be declared using the *const* qualifier. Any variables qualified with the keyword *const* are read-only variables for that shader. Declaring variables as constant allows more descriptive shaders than using hard-wired numerical constants. The const qualifier can be used with any of the fundamental data types. It is an error to write to a const variable outside of the declaration and they must be initialized as part of the declaration. An example is:

```
const vec3 zAxis = vec3 (0, 0, 1);
```

The keyword *attribute* is used to qualify variables that are passed to a vertex shader from OpenGL on a per-vertex basis. It is an error to declare an attribute variable in any type of shader other than a vertex shader. Attribute variables are read-only as far as the vertex shader is concerned. Values for attribute variables are passed to a vertex shader through the OpenGL vertex API or as part of a vertex array. They convey vertex attributes to the vertex shader and are expected to change on every vertex shader run. The attribute qualifier can be used only with the data types float, vec2, vec3, vec4, mat2, mat3, and mat4.

A declaration looks like:

```
attribute vec4 position;
attribute vec3 normal;
attribute vec2 texCoord;
```

All the standard OpenGL vertex attributes have built-in variable names to allow easy integration between user programs and OpenGL vertex functions. The built-in attribute names are listed later in this section.

It is expected that graphics hardware will have a small number of fixed locations for passing vertex attributes. Therefore, the OpenGL Shading language defines each attribute variable as having space for up to four floating-point values (i.e., a vec4). There is an implementation dependent limit on the number of attribute variables that can be used and if this is exceeded it will cause a link error. (Declared attribute variables that are not used do not count against this limit.) A float attribute counts the same amount against this limit as a vec4, so applications may want to consider packing groups of four unrelated float attributes together into a vec4 to better utilize the capabilities of the underlying hardware. A mat4 attribute will use up the equivalent of 4 vec4 attribute variable locations, a mat3 will use up the equivalent of 3 attribute variable locations, and a mat2 will use up 1 attribute variable location

The keyword *uniform* is used to qualify variables that remain constant for the entire primitive being processed. Typically, variables qualified as being uniform will be constant over a number of primitives/vertices/fragments. Uniform variables are read-only as far as the shader is concerned and are initialized directly by an application via API commands, or indirectly because of state tracking. The uniform qualifier can be used with any of the fundamental data types.

An example declaration is:

```
uniform vec4 lightPosition;
```

There is an implementation dependent limit on the amount of uniform storage that can be used for each type of shader and if this is exceeded will cause a compile time error. (Uniform variables that are declared but not used do not count against this limit.) The number of user-defined uniform variables and the number of built-in uniform variables that are used within a shader are added together to determine whether available uniform

storage has been exceeded. If two shaders that are linked together refer to a uniform variable of the same name, they will read the same value.

Varying variables provide the interface between a vertex shader and a fragment shader. These variable are write-only for the vertex shader and read-only for the fragment shader. You can think of these variables as allocating the hardware interpolators that will be used when rendering a primitive. At link time, the varying variables actually used in the vertex shader are compared to the varying variables actually used in the fragment shader. If the two do not match, a link error will result. The built-in varying variable *gl_Position* holds the homogeneous vertex position and must be written to by the vertex shader otherwise a compiler error will be generated.

An example declaration looks like:

```
varying vec3 normal;
```

Attribute variables, uniform variables, and varying variables all share the same name space so they must have unique names. To avoid namespace confusion within a shader, all attribute, uniform, and varying variables must be declared at a global scope (outside any functions).

Vector components

The names of the components of a vector are denoted by a single letter. As a notational convenience, several letters are associated with each component based on common usage of position, color or texture coordinate vectors. A generic vector component notation is also provided. The individual components of a vector can be selected by following the variable name with a '.' and the component name.

The component names supported are:

{*x, y, z, w*}	useful when accessing vectors that represent points or normals
{*r, g, b, a*}	useful when accessing vectors that represent colors
{*s, t, p, q*}	useful when accessing vectors that represent texture coordinates
{*0, 1, 2, 3*}	useful when accessing vectors that contain generic or unnamed values

The component names *x, r, s* and *0* are, for example, synonyms for the same (first) component in a vector.

Matrix Components

The individual components of a matrix can be accessed by following the variable with a '.' and the component two digit row column number: 00, 01, 02, 03, 10, …, 33. 00 is the top left corner of the matrix. The first digit represents the row to be accessed, and the second digit represents the column to be accessed. Attempting to access a component outside the bounds of a matrix (e.g., component 33 of a mat3) will generate a compiler error. An entire row or column of a matrix can be accessed by using the "_" in place of a specific row or column number. Thus, ._0 will access column 0 and .0_ will access row 0, etc.

```
mat3   m;
m.00 = 1;// set the first element to 1.
m.0_ = vec3 (1, 2, 3);// sets the first row to 1, 2, 3
```

Arrays

Uniform or varying variables can be aggregated into arrays (normal read/write variables cannot to discourage overuse of temporary storage and because it is difficult for a compiler in determine register reuse) using the C [] operator. The size must be specified. Some examples are:

```
uniform vec4 lightPosition[4];
varying vec3 material[8];
```

There is no mechanism (or need) to be able to initialize these arrays in the shaders as the initialization is done by API commands.

Array elements are accessed as in C:

```
diffuseColor += lightIntensity[3] * NdotL;
```

The array index starts from zero and the index is an integer constant or an integer variable. An array can be passed by reference to a function by just using the array name without any index.

Basic language constructs

White space and indenting are only mandatory to the extent that they separate tokens in the language, otherwise they are ignored.

Comments may be denoted using either the C /* */ syntax or the C++ // comment syntax.

Expressions in the shading language include the following:

- constants: floating point (e.g. 2.0, 3, -3.24, 3.6e6) and named constants (from the const declaration).
- vector and matrix constructors, for example vec3 (0, 1, 0).
- variable references, array subscripting, and vector/matrix component selection.
- the binary operators +, -, *, /, and %.
- the unary - (negation) operator, as well as pre- and post- increment and decrement (--) and (++).
- the relational operators (>, >=, <, <=), the equality operators (==, !=) and logical operators (&&, || and !). All these operations result as bool. Logical operators operate on type bool and auto-convert their operands to be bool.
- the comma operator (,) as in C
- built in-function calls
- user defined function calls

Operator precedence rules are as in C and parentheses can be used to change precedence.

Assignments are made using '=' as in C and the variable must be of the same type as finally produced by the expression, i.e. no implied casting is done. However, an int will be automatically promoted to a float. Expressions on the left of an assignment are evaluated before expressions on the right of the assignment. The +=, -=, *=, /=, and %= assignment operators in C are also supported.

Control Flow

The fundamental control flow features of the OpenGL Shading Language are:

- block structured statement grouping

- conditionals
- looping constructs
- function calls

These constructs are all modeled on their counterparts in the C programming language.

A statement is an expression or declaration terminated by a semicolon.

Braces are used to group declarations and statements together into a compound statement or block so that they are syntactically equivalent to a single statement. Any variables declared within a brace-delimited group of statements are only visible within that block. In other words, the variable scoping rules are the same as in C.

The conditional expression is supported and is written using the ternary operator "?:". In the expression:

```
expression ? true-expression : false-expression
```

expression is evaluated first. If it is true or non-zero then *true-expression* is evaluated and that is the value of the conditional expression. Otherwise, *false-expression* is evaluated and that is the value. Constructors should be used to obtain compatible types. If the types of *true-expression* and *false-expression* differ, a compiler error will be generated.

Conditionals in the shading language are the same as in C:

```
if (expression)
    true-statement
```

or

```
if (expression)
    true-statement
else
    false-statement
```

The expression is evaluated, and if it is true (that is, if *expression* is true or a non-zero value), *true-statement* is executed. If it is false (*expression* is false or zero) and there is an else part then *false-statement* is executed instead.

The relational operators ==, !=, <, <=, > and >= may be used in the expression and be combined using the logical operators &&, || and !. If the conditional is operating on a vector or a matrix then all the components of the vector or matrix must be true for *true-statement* to be executed.

Conditionals can be nested.

For, while, and do loops are allowed as in C:

```
for (init-expression; condition-expression; loop-expression)
    statement;

while (condition-expression)
    statement;

do
```

```
    statement;
while (condition-expression)
```

Constructs for early loop termination are also the same as in C:

```
break;
continue;
```

Integer and boolean expressions are preferred for *condition-expression*, as this will ensure all the vertices or fragments in a primitive have the same execution path whereas with a float as a control variable this cannot be guaranteed.

The break statement can be used to break out of the loop early. In the for loop, the continue statement can be used to skip the remainder of the statement and execute *loop-expression* and *condition-expression* again.

Loops can be nested.

A valid shader contains exactly one function named main. This function takes no arguments and returns no value. Therefore, the skeleton of all valid shaders will look like this:

```
void main(void)
{
...
}
```

User defined functions

Shaders can be partitioned into functions as an aid to writing clearer programs and reusing common operations. An implementation can inline the calls or do true subroutines. Recursion is not allowed.

A function is defined similar to C:

```
// prototype
returnType functionName (type0 arg0, type1 arg1, …, typen argn);

// definition
returnType functionName (type0 arg0, type1 arg1, …, typen argn)
{
// do some computation
return returnValue;
}
```

All functions must be defined before they are called or have prototypes.

Arguments are always passed by reference and are read-only unless the output keyword is used to qualify an argument. For example:

```
float myfunc (float f, // you cannot assign to f
       output float g) // you can assign to g
```

Aliasing of arguments is not allowed.

A constant may be passed in as an argument even though a reference to a constant doesn't really exist. A constant will generate a compile error if qualified by the output keyword. The scope rules for variables declared within the body of the function are the same as in C.

Functions that return no value use void to signify this. Functions that accept no input arguments need not use void; as in C++, since prototypes are required, there is no ambiguity when an empty argument list "()" is declared.

Functions can be overloaded as in C++. This allows the same function name to be used for multiple functions, as long as the argument list types differ. This is used heavily in the built-in functions. When overloaded functions (or indeed any functions) are resolved, an exact match for the function's signature is looked for. Other than auto-promotion of int to float, as needed, no promotion or demotion of the return type or input argument types is done. All expected combination of inputs and outputs must be defined as separate functions.

Built-in functions

The OpenGL Shading Language defines an assortment of built-in convenience functions for scalar and vector operations.

The built-in functions basically fall into three categories:

- They expose some necessary hardware functionality in a convenient way such as accessing a texture map. There is no way in the language for these functions to be emulated by the user.
- They represent a trivial operation (clamp, mix, etc.) which are very simple for the user to write, but they are very common and may have direct hardware support. It is a very hard problem for the compiler to map expressions to 'complex' assembler instructions.
- They represent an operation which we hope to accelerate at some point. The trigonometry functions fall into this category.

Many of the functions are similar to the same named ones in C, but they support vector input as well as the more traditional scalar input. For operations on vectors, the output will have the same type as the input and the function will have been applied component-by-component. The OpenGL Shading Language includes the following built-in functions:

Trigonometry functions:

- Radians
- Degrees
- Sine
- Cosine
- Tangent
- Arcsine
- Arccosine
- Arctangent

Exponential functions:

- Power
- Exponential

- Log
- Square Root
- Inverse Square Root

Common functions:

- Absolute Value
- Sign
- Floor
- Ceiling
- Fraction
- Modulo
- Min
- Max
- Clamp
- Mix
- Step
- Smoothstep

Geometric functions:

- Length
- Distance
- Normalize
- Face Forward
- Reflect
- Dot Product
- Cross Product

Matrix functions:

- Matrix Multiply

Vertex processing functions are available only in shaders intended for use on the vertex processor:

- Element

Fragment processing functions are only available in shaders intended for use on the vertex processor:

- Texture Access
- Derivative
- Level-of-Detail
- Kill
- Noise

Built-in variables

The following attribute names are built into the OpenGL Shading Language and can be used from within a vertex shader to access the current values of attributes defined by OpenGL:

```
attribute vec4  gl_Color;
attribute vec3  gl_Normal;
attribute vec4  gl_Vertex;
attribute vec4  gl_MultiTexCoord0;
attribute vec4  gl_MultiTexCoord1;
attribute vec4  gl_MultiTexCoord2;
attribute vec4  gl_MultiTexCoord3;
attribute vec4  gl_MultiTexCoord4;
attribute vec4  gl_MultiTexCoord5;
attribute vec4  gl_MultiTexCoord6;
attribute vec4  gl_MultiTexCoord7;
attribute vec4  gl_SecColor;
attribute vec3  gl_FrontMaterial[5];
attribute vec3  gl_BackMaterial[5];
```

As an aid to accessing the various components of the material arrays, the following built-in constants are also built into the OpenGL Shading Language:

```
constant  int   gl_kEmmisiveColor    = 0;
constant  int   gl_kAmbientColor     = 1;
constant  int   gl_kDiffuseColor     = 2;
constant  int   gl_kSpecularColor    = 3;
constant  int   gl_kDiffuseAlpha     = 4; // .x
constant  int   gl_kSpecularExponent = 4; // .y
```

As an aid to accessing OpenGL vertex processing state, the following uniform variables are built into the OpenGL Shading Language and are accessible from within a vertex shader:

```
uniform mat4  gl_ModelViewMatrix;
uniform mat4  gl_ModelViewProjectionMatrix;
uniform mat3  gl_NormalMatrix;
uniform mat4  gl_TextureMatrix[8];
uniform vec3  gl_SceneAmbient;
uniform mat4  gl_TexGen[8];
uniform float gl_NormalScale;
uniform vec3  gl_Light0[8];
uniform vec3  gl_Light1[8];
uniform vec3  gl_Light2[8];
uniform vec3  gl_Light3[8];
uniform vec3  gl_Light4[8];
uniform vec3  gl_Light5[8];
uniform vec3  gl_Light6[8];
uniform vec3  gl_Light7[8];

// Additional lights would follow here if an implementation exports
// more than the minimum of 8.
```

As an aid to accessing the various components of the light arrays from within a vertex shader, the following built-in constants are also built into the OpenGL Shading Language:

```
// Define the layout of the vec3 light parameters in the Light arrays.
// Note the two scalar spotlight values are stored in a vec3 for
// convenience.
const int gl_kAmbientIntensity   = 0;
const int gl_kDiffuseIntensity   = 1;
const int gl_kSpecularIntensity  = 2;
const int gl_kPosition          = 3;
const int gl_kHalfVector         = 4;
const int gl_kAttenuation        = 5;
const int gl_kSpotlightDirection = 6;
const int gl_kSpotlight          = 7; // x = CutoffAngle, y = Exponent
```

As an aid to accessing OpenGL fragment processing state, the following uniform variables are built into the OpenGL Shading Language and are accessible from within a fragment shader:

```
uniform vec4  gl_TextureEnvColor[8];
uniform float gl_FogDensity;
uniform float gl_FogStart;
uniform float gl_FogEnd;
uniform float gl_FogScale; // = 1 / (gl_FogEnd - gl_FogStart)
uniform vec3  gl_FogColor;

// Variables used in the pixel processing and imaging operations.
uniform vec4  gl_ColorScaleFactors;  // RED/GREEN/BLUE/ALPHA_SCALE
uniform vec4  gl_ColorBiasFactors;   // RED/GREEN/BLUE/ALPHA_BIAS
uniform float gl_DepthScaleFactor;   // DEPTH_SCALE
uniform float gl_DepthBiasFactor;    // DEPTH_SCALE
uniform float gl_IndexShift;         // equals 2^INDEX_SHIFT
uniform float gl_IndexOffset;        // INDEX_OFFSET
uniform mat4  gl_ColorMatrix;        // set up to track state
uniform vec4  gl_PostColorMatrixScaleFactors;
uniform vec4  gl_PostColorMatrixBiasFactors;
```

The vertex shader has access to the following built-in varying variables which direct clipping and rasterization activities:

```
varying vec4  gl_Position; // must be written to
varying float gl_PointSize;
varying vec4  gl_ClipVertex;
```

The fragment shader has access to the following built-in varying variables, which holds the x, y, z, and 1/w values for the fragment being processed:

```
varying vec4 gl_FragCoord;
```

The fragment shader has access to the following built-in variables:

```
bool  gl_FrontFacing;
vec4  gl_FragColor;
float gl_FragDepth;
```

```
float gl_FragStencil;
vec4  gl_FragData0-n       // n is implementation-dependent

// Frame Buffer
vec4  gl_FBColor;
float gl_FBDepth;
float gl_FBStencil;
vec4  gl_FBDatan;
```

The following built-in varying variables can be written to from within a vertex shader and can be read from within a fragment shader:

```
varying vec4  gl_FrontColor;
varying vec4  gl_BackColor;
varying vec4  gl_TexCoord0;
varying vec4  gl_TexCoord1;
varying vec4  gl_TexCoord2;
varying vec4  gl_TexCoord3;
varying vec4  gl_TexCoord4;
varying vec4  gl_TexCoord5;
varying vec4  gl_TexCoord6;
varying vec4  gl_TexCoord7;
varying vec4  gl_FrontSecColor;
varying vec4  gl_BackSecColor;
varying float gl_EyeZ;
```

6.4 Brick Shader

Now that we have the basics of the OpenGL Shading Language in hand, let's look at a simple example. In this example, we'll be applying a brick pattern to an object. The brick pattern will be calculated entirely within a fragment shader.

Application Setup

The input to the vertex shader will be a single light position stored as a uniform variable and a normal and a position for each vertex that are supplied through the usual OpenGL mechanisms. There is no need to supply color or texture coordinates since these will be computed algorithmically in the fragment shader.

The uniform variables for the fragment shader are mortarThickness, brickColor, mortarColor, brickMortarWidth, brickMortarHeight, mwf (mortar width fraction), and mhf (mortar height fraction).

The uniform variables are initialized by the application using the following C code:

```
float bc[3] = { 1.0, 0.3, 0.2 };
float mc[3] = { 0.85, 0.86, 0.84 };
float bmw = 0.22;
float bmh = 0.15;
float mwf = 0.94;
float mhf = 0.90;
float lightPosition[3] = { 0.0, 0.0, 4.0 };

glLoadUniform3fv(glGetUniformLocation(brickProgramObject,
                        "brickColor"), &bc[0]);
glLoadUniform3fv(glGetUniformLocation(brickProgramObject,
                        "mortarColor"),&mc[0]);
glLoadUniform1fv(glGetUniformLocation(brickProgramObject,
                        "brickMortarWidth"), &bmw);
glLoadUniform1fv(glGetUniformLocation(brickProgramObject,
                        "brickMortarHeight"), &bmh);
glLoadUniform1fv(glGetUniformLocation(brickProgramObject,
                        "mwf"), &mwf);
glLoadUniform1fv(glGetUniformLocation(brickProgramObject,
                        "mhf"), &mhf);
glLoadUniform3fv(glGetUniformLocation(brickProgramObject,
                        "LightPosition"), &lightPosition[0]);
```

Vertex shader

The code below is the vertex shader that is needed for our brick shader. This shader demonstrates some of the capabilities of the OpenGL Shading Language:

```
varying float LightIntensity;
varying vec3  Position;
uniform vec3  LightPosition;

const float specularContribution = 0.7;
const float diffuseContribution  = (1 - specularContribution);
```

```
void main(void)
{
    vec4 pos        = gl_ModelViewMatrix * gl_Vertex;
    Position        = vec3(gl_Vertex);
    vec3 tnorm      = normalize(gl_NormalMatrix * gl_Normal);
    vec3 lightVec   = normalize(LightPosition - vec3(pos));
    vec3 reflectVec = reflect(lightVec, tnorm);
    vec3 viewVec    = normalize(vec3(pos));

    float spec = clamp(dot(reflectVec, viewVec), 0, 1);
    spec = spec * spec;
    spec = spec * spec;
    spec = spec * spec;
    spec = spec * spec;

   LightIntensity = diffuseContribution * dot(lightVec, tnorm) +
                    specularContribution * spec;

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

The purpose of this shader is to produce three values that will be interpolated across each primitive and ultimately used by the fragment shader that will be described in the next section. These three values are light intensity, vertex position in object space, and the transformed vertex position. These values are represented in our vertex shader by the varying variables *LightIntensity* and *Position* (which are defined as varying variables in the first two lines of the shader), and *gl_Position* (which is built in to the OpenGL Shading Language and must be computed by every vertex shader).

Once we enter the main function, the first thing we do is compute the position of the vertex in world coordinates. This is done by declaring a local variable *pos* and initializing it by multiplying the current OpenGL model-view matrix and the incoming vertex value.

Next we use a constructor to convert the incoming vertex value to a vec3 and store it in our varying variable *Position*. The underlying hardware will have a fixed number of interpolators for us to use, so there's no sense interpolating the *w* coordinate if we're not going to need it in the fragment shader. As you can see, there's nothing to prevent a vertex shader from reading an input value (*gl_Vertex*) more than once.

After this, we need to compute the values needed for the light intensity calculation. The incoming normal (*gl_Normal*) is transformed by the OpenGL normal transformation matrix and then normalized by calling the built-in function normalize(). The light direction is computed by subtracting the world coordinate position (*pos*) from the uniform variable that holds our light position in world coordinates. This vector is also normalized. The reflection vector is computed by calling the built-in function reflect(), and the return value from this function is also normalized. Finally, the viewing direction vector is computed by normalizing the world coordinate position (without the *w* coordinate).

A specular value is then computed by using the built-in function dot() to compute the dot product of the reflection vector and the view vector. At the time of this writing, the built-in power function (pow()) has not been implemented, so we simply multiply the specular value by itself four times in order to achieve the same

effect as pow(spec, 16). Finally, the light intensity is computed by taking 30% of the diffuse value and 70% of the specular value.

Vertex shaders must compute a value for *gl_Position* and that is taken care of in the last line of code. The current model-view-projection matrix is obtained from OpenGL state and is used to multiply the incoming vertex value. The resulting transformed coordinate is stored in *gl_Position* as required by the OpenGL Shading Language. This vertex value will subsequently be combined with others to construct a primitive, and the resulting primitive will then be clipped, culled, and sent on for rasterization.

Fragment shader

The fragment shader below works with the vertex shader in the previous section to render objects with a brick pattern.

```
uniform vec3brickColor;
uniform vec3mortarColor;

uniform floatbrickMortarWidth;
uniform floatbrickMortarHeight;
uniform floatmwf;
uniform floatmhf;

varying vec3  Position;
varying float LightIntensity;

void main (void)
{
    vec3ct;
    floatss, tt, w, h;

    ss = Position.x / brickMortarWidth;
    tt = Position.z / brickMortarHeight;

    if (fract (tt * 0.5) > 0.5)
        ss += 0.5;

    ss = fract (ss);
    tt = fract (tt);

    w = step (mwf, ss) - step (1 - mwf, ss);
    h = step (mhf, tt) - step (1 - mhf, tt);

    ct = clamp(mix(mortarColor, brickColor, w*h)*LightIntensity, 0, 1);

    gl_FragColor = vec4 (ct, 1.0);
}
```

This shader starts off by defining a few more uniform variables than did the vertex shader. The brick pattern that will be rendered on our geometry is parameterized in order to make it easier to modify. The parameters that are constant across an entire primitive can be stored as uniform variables and initialized (and later

modified) by the application. This makes it easy to expose these controls to the end user for modification through user interface elements such as sliders.

We want our brick pattern to be applied in a consistent way to our geometry in order to have the object look the same no matter where it is placed in the scene or how it is rotated. The key to determining the placement of the brick pattern is the value that is passed in the varying variable *Position*. This variable was computed at each vertex by the vertex shader in the previous section, and it is interpolated across the primitive and made available to the fragment shader at each fragment location. Our fragment shader can use this information to determine where the fragment location is in relation to the algorithmically defined brick pattern.

The first step is to divide the object's x position by the brick+mortar width and the z position by the brick+mortar height. This gives us a "brick row number" (*tt*) and a "brick number" within that row (*ss*). Keep in mind that these are signed, floating point values, so it is perfectly reasonable to have negative row and brick numbers as a result of this computation.

The purpose of the next line is to offset every other row of bricks by half a brickwidth. The "brick row number" (*tt*) is multiplied by 0.5 and the result is compared against 0.5. Half the time (or every other row) this comparison will be true, and the "brick number" value is in incremented by 0.5 to offset the entire row.

Next, the built-in math function fract() is then used to obtain the fractional parts of our width and height values. We then compute two values that tell us whether we are in the brick or in the mortar in the horizontal direction (*w*) and in the vertical direction (*h*). The built-in function step() is used to produce a value of 1 if the brick color is to be used, and 0 if the mortar color is to be used. If the surface of our brick is parameterized to go from 0.0-1.0 in both width and height, then we can imagine that our brick pattern is the brick color from 0.0-*mwf* in the width direction and 0.0-*mhf* in the height direction. The values of *mwf* and *mhf* can be computed by the application to give a uniform mortar width in both directions based on the ratio of brick+mortar width to brick+mortar height, or they can be chosen aribitrarily to give a mortar appearance that "looks right."

Finally, we compute the color of the fragment and store it in a temporary variable. The built-in function mix() is used to choose the brick color or the mortar color, depending on the value of *w***h.* Since *w* and *h* can only have values of 0.0 (mortar) or 1.0 (brick), we will chose the brick color only if both values are 1.0, otherwise we will choose the mortar color. The resulting value is then multiplied by the light intensity, and that result is clamped to the range [0.0, 1.0] and stored in a temprorary variable (*ct*). This temporary variable is a vec3, so we create our final color value by using a constructor to add a 1.0 as the fourth element of a vec4 and assign the result to our built-in variable *gl_FragColor.*

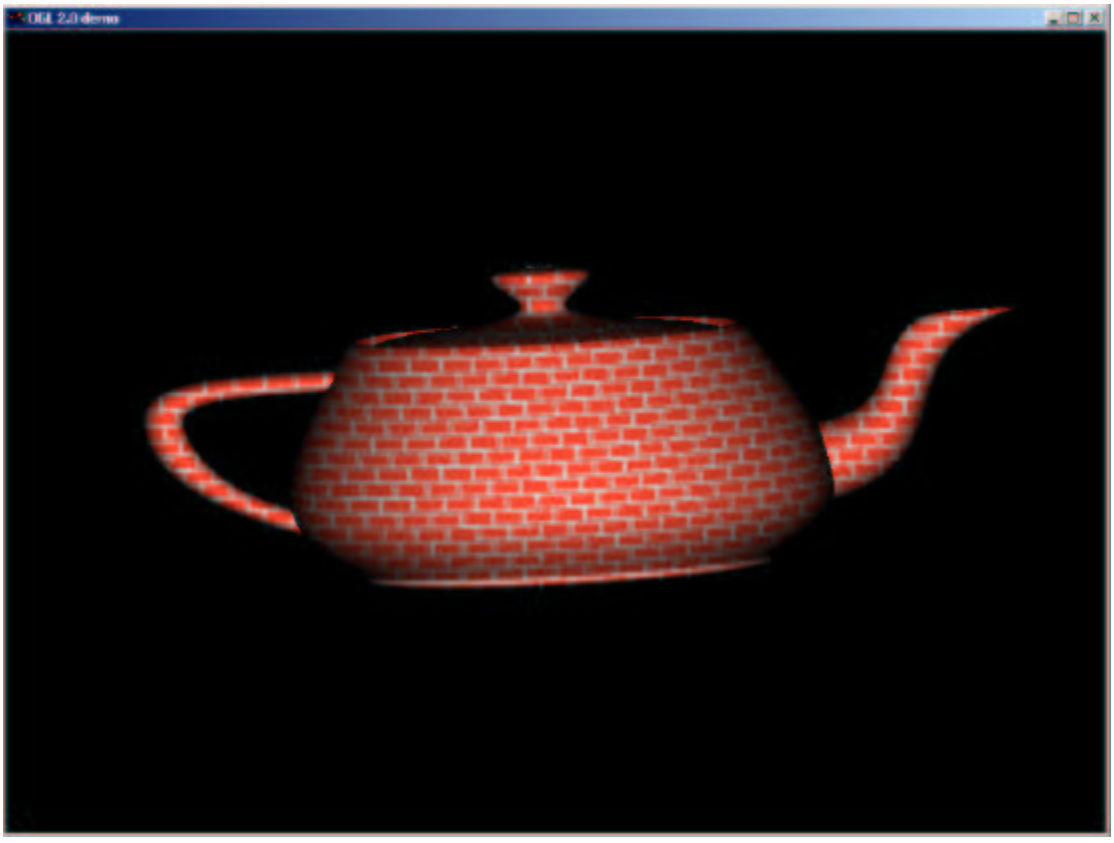
One possible improvement to this shader is to notice that we never actually used the *y* component of our varying variable position. This value was interpolated but never used. Since hardware implementations will have a fixed number of interpolators available, we would be making better use of our scarce resources by passing the *x* and *z* components as a varying vec2 variable. This would make the code more cryptic, but it may be worth doing if your program requires a large number of varying variables for other things.

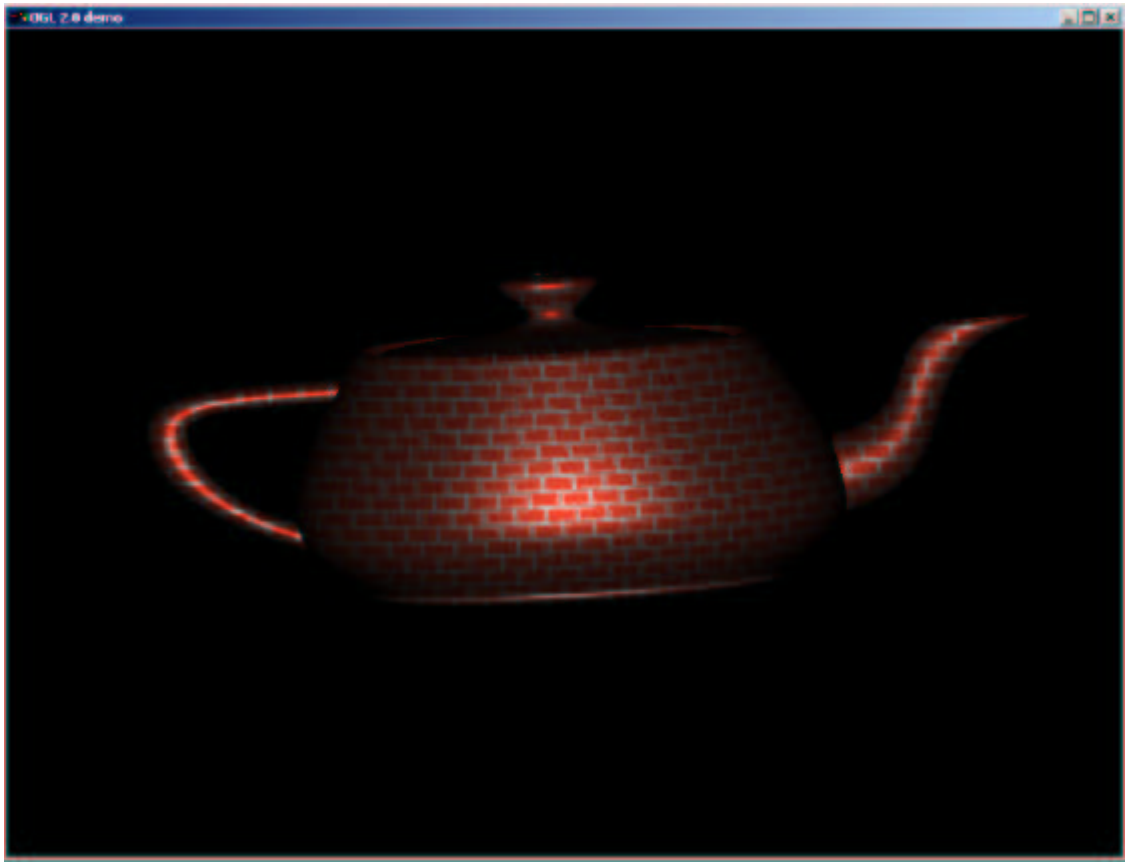
When comparing this shader to the vertex shader in the previous example, we notice one of the key features of the OpenGL shading language, namely that the language used to write these two shaders is almost identical. Both shaders have a main function, some uniform variables, some local variables, expressions are the same, built-in functions are called in the same way, constructors are used in the same way, and so on. The only perceptible difference exhibited by these two shaders is that the varying variables were write-only

for the vertex shader described in the previous section, and they are read-only for the fragment shader described in this section.

Screen shots

The screen shots below show the results of our brick shaders. The first case shows diffuse lighting only and was achieved simply by setting *specularContribution* to 0 and *diffuseContribution* to 1. The second image shows the result of running the shaders as shown in the listings above.





6.5 Bump Mapping and Environment Mapping

A variety of interesting effects can be applied to a surface using a technique called *bump mapping*. Bump mapping involves modulating the surface normal before lighting is applied. The modulation can be done algorithmically to apply a regular pattern, it can be done by adding noise to the components of a normal, or it can be done by looking up a perturbation value in a texture map. This technique does not truly alter the surface being shaded, it merely "tricks" the lighting calculations. Therefore, the "bumping" will not show up on the silhouette edges of an object. Imagine modeling the moon as a sphere, and shading it with a bump map so that it appears to have craters. The silhouette of the sphere will always be perfectly round, even if the "craters" (bumps) go right up to the silhouette edge. In real life, you would expect the craters on the silhouette edges to prevent the silhouette from looking perfectly round. For this reason, it is a good idea to use bump mapping to only apply "small" effects to a surface (at least relative to the size of the surface). Wrinkles on an orange, embossed logos, and pitted bricks are all good examples of things that can be successfully bump mapped.

A technique that is used to model reflections in a complex environment without resorting to ray-tracing is called *environment mapping*. In this technique, one or more texture maps are used to simulate the reflections in the environment that is being rendered. It is best used when rendering objects that have mirror-like qualities. One way to perform environment mapping is by using what is called a *cube map*. A cube map is a texture that is made up of six textures, corresponding to top, bottom, and sides of the environment to be modeled. Imagine placing a mirror-like object in the middle of a room. Texture maps can be created to simulate the floor, ceiling, and walls of the room. To shade the mirror-like object, you need only use the reflection vector to access the appropriate face of the cube map and return the value at that location. This value can be combined with the surface color in order to shade the object.

The following OpenGL 2.0 shaders combine bump mapping and environment mapping in order to render an object.

Application Setup

(Not available at the time of this writing.)

Vertex Shader

(Not available at the time of this writing.)

Fragment Shader

(Note: at the time of this writing, this example has NOT been run on a working OpenGL 2.0 implementation. See the OpenGL 2.0 location on the 3Dlabs web site at http://www.3dlabs.com to get an up-to-date version of this paper, along with any corrections to the shader source below.)

In this example a normal is read from a texture map and transformed by the tangent space local coordinate system (which is iterated across the primitive). The transformed normal together with an iterated eye vector is used to calculate a reflected vector. The transformed normal then accesses the diffuse cube map and the reflected vector accesses the specular cube map. The diffuse and specular light intensities from the cube maps are combined with a base color and gloss value (another texture map) and the final color calculated.

There is a bit more setup work to perform prior to exeucting this fragment shader. Before we can run it, we need to make sure the following have been set up using the normal OpenGL mechanisms:

- A cube map containing the diffuse reflection values that is loaded and bound to texture unit 0
- A cube map containing the specular reflection values that is loaded and bound to texture unit 1
- A regular texture map containing normals to be applied to the object
- A regular texture map that contains the base color and gloss for the object that is loaded and bound to texture unit 3

```
// Define the interpolators and their binding to the values loaded by
// the vertex shader.
varying mat3    lcm;                    // local coordinate matrix
varying vec3    eye;
varying vec2    baseST;

void main (void)
{
    // Declarations
    vec3        N, reflect, normal;
    vec4        base;                   // base color and gloss

    // Define names for the texture units to which the texture
    // objects are bound. We could just use numbers in the texture
    // function, but this is more descriptive.
    const int diffuseCube  = 0;         // rgb
    const int specularCube = 1;         // rgb
    const int normalMap    = 2;
    const int baseColorMap = 3;         // rgb, gloss

    // Fetch normal from normal map and transform it using the
    // interpolated local coordinate matrix.
    N = lcm * texture3 (normalMap, baseST);

    // Reflect the eye about the normal.
    reflect =    2 * N * dot (N, eye) - eye * dot (N ,  N);

    // Calculate the color. Base map holds gloss value
    // in alpha.
    base = texture4 (baseColorMap, baseST);

    vec3 color;
    color = texture3(diffuseCube, N) * vec3 (base.r, base.g, base.b) +
                texture3(specularCube, reflect) * base.a;

    gl_FragColor = vec4 (color, 1.0);
}
```

Screen Shots

(Not available at the time of this writing.)

6.6 BRDF Shader

In order to model more physically realistic surfaces, we must go beyond the simplistic lighting/reflection model that is built into OpenGL. For some time, computer graphics researchers have been rendering images with a more realistic reflection model called the bidirectional reflectance distribution function, or BRDF. Instruments have been developed to measure the BRDF of real materials. Michael McCool of the University of Waterloo and others have developed methods for factorizing this measured data in order to produce texture maps that can be used to render BRDF surfaces on the current generation of graphics hardware. This section describes the OpenGL Shading Language shaders that can be used to render McCool-style BRDF surfaces using the BRDF textures available from the University of Waterloo. This section describes shaders that implement the V*H*W parameterization described at the University of Waterloo web site (http://www.cgl.uwaterloo.ca/Projects/rendering/Papers/index.html.) These programs should work with the sample V*H*W BRDF textures that are available there.

(Note: At the time of this writing, we do not have enough of OpenGL 2.0 implemented to execute these shaders. Before doing anything with the shaders described in this section, please check the OpenGL 2.0 section of the 3Dlabs web site at http://www.3dlabs.com to obtain a more up-to-date version of this paper. The shaders described in this section have been compiled and have been reviewed by others, but have NOT yet been executed on OpenGL 2.0-capable hardware.)

Application Setup

To render BRDF surfaces using the following shaders, the application needs to set up a few uniform variables. The vertex shader must provide the values for uniform variables that describe the eye position and the position of a single light source in object space. (The shaders were developed this way for similarity to the demo program available at the University of Waterloo web site.) The fragment shader requires the application to provide values for two uniform variables: a color for a single light source, and a scale factor that is used to rescale the values retrieved from the BRDF texture maps. (Each pair of BRDF texture maps has been prescaled in order to preserve precision, and so must be rescaled using the rescale factors specific to that set of BRDF texture maps.)

The application is expected to provide five attributes for every vertex. Three of them are standard OpenGL attributes and need not be defined by our vertex program: gl_Vertex (position), gl_Normal (normal), and gl_TexCoord0 (texture coordinate). The other two attributes are a tangent vector and a binormal vector which are assumed to be computed by the application. These two attributes should be provided to OpenGL using either the glVertexAttribute* function or by using a vertex array of type GL_USER_ATTRIBUTE_ARRAY*n*. The location to be used for these user-defined attributes can be bound to the appropriate attribute in our vertex shader by calling glBindAttributeLocation. For instance, if we choose to pass the tangent values in at vertex attribute location 0 and the binormal values in at vertex attribute location 1, we would set up the binding using these lines of code:

```
glBindAttributeLocation(myProgramID, 0, "tangent");
glBindAttributeLocation(myProgramID, 1, "binormal");
```

Prior to rendering, the application should also set up three texture maps: two cube maps that contain the BRDF factorization data, and a regular texture that describes the surface color of the object. The BRDF "P" texture map is assumed to be bound to texture unit 0, the BRDF "Q" texture map is assumed to be bound to

texture unit 1, and the surface color texture is assumed to be bound to texture unit 2. The BRDF texture maps should be set up so that the edge of each texture map extends to infinity, not to wrap.

Vertex Shader

Our BRDF vertex shader is shown below. It is based on the vertex program that comes with the BRDF demo program from the University of Waterloo. The purpose of this shader is to produce six varying values:

- *gl_Position*, as required by every vertex shader
- *gl_TexCoord0*, which will be used to access a texture map to get the base color of the surface
- *inDirection*, a vec3 representing the incoming light direction that is used to access the factorized BRDF textures
- *haDirection*, a vec3 representing the half angle between the light direction and the viewing direction
- *outDirection*, a vec3 representing the outgoing light direction that is used to access the factorized BRDF textures
- *lightIntensity*, a float that is used to store the cosine of the angle between the incoming surface normal and the light source direction.

The incoming vertex position is transformed as required in the first line of code. Light direction, view direction, and half-angle are normalized direction vectors that are computed in the next three lines. A diffuse light intensity is then calculated based on the incoming normal and the light position. The following nine lines of code compute orthonormal surface frames for each of the direction vectors. Finally, the incoming texture coordinate is copied to a built-in varying variable so that it can be interpolated and used in the fragment shader.

A couple of interesting language features are demonstrated in this shader. As you can see, incoming attribute values can be read multiple times. The incoming vertex value is accessed three times and the incoming normal is accessed four times. Variables can be declared when they are first used as shown by the local variables *lightDir*, *viewDir*, and *halfangleDir*.

```
uniform vec3   lightPosition;    // light position in object space
uniform vec3   eyePosition;      // viewing position in object space

attribute vec3 tangent;          // tangent for the specified vertex
attribute vec3 binormal;         // binormal for the specified vertex

varying vec3   inDirection;
varying vec3   outDirection;
varying vec3   haDirection;
varying float  lightIntensity;

void main(void)
{
    // transform incoming vertex
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    // compute light direction vector
    vec3 lightDir = normalize(lightPosition - vec3 (gl_Vertex));

    // compute viewing direction vector
```

```
    vec3 viewDir = normalize(eyePosition - vec3 (gl_Vertex));

    // compute half angle direction vector
    vec3 halfangleDir = normalize(lightDir + viewDir);

    // compute diffuse reflection, clamp to [0,1]
    lightIntensity = clamp(dot(gl_Normal, lightDir), 0, 1);

    // compute outgoing direction
    outDirection.x = dot(viewDir, tangent);
    outDirection.y = dot(viewDir, binormal);
    outDirection.z = dot(viewDir, gl_Normal);

    // compute half-angle direction
    haDirection.x = dot(halfangleDir, tangent);
    haDirection.y = dot(halfangleDir, binormal);
    haDirection.z = dot(halfangleDir, gl_Normal);

    // compute incoming direction
    inDirection.x = dot(lightDir, tangent);
    inDirection.y = dot(lightDir, binormal);
    inDirection.z = dot(lightDir, gl_Normal);

    // copy texture coord into a standard varying variable
    gl_TexCoord0 = gl_MultiTexCoord0;
}
```

Fragment Shader

The fragment shader for our BRDF surface rendering is based on the background information provided at http://www.cgl.uwaterloo.ca/Projects/rendering/Shading/database.html. The incoming direction vectors are used to perform two texture lookups in the BRDF "P" cube map, and one texture lookup in the BRDF "Q" cube map. The results of these lookups are multiplied together and scaled by a scaling factor that is specific to the BRDF textures being used. Another texture lookup is performed to obtain the base color of the surface, and the diffuse reflection from a single light source is also computed. These results are multiplied by the previous result, and the final fragment color value is stored in gl_FragColor.

```
uniform vec3   scaleFactor;       // BRDF scaling factors
uniform vec3   lightColor;        // diffuse light color

varying vec3   inDirection;
varying vec3   outDirection;
varying vec3   haDirection;
varying float  lightIntensity;

const   int    brdfP = 0;         // BRDF cube map "P" uses texture unit 0
const   int    brdfQ = 1;         // BRDF cube map "Q" uses texture unit 1
const   int    surfaceColor = 2;  // Surface color texture uses texture unit 2

void main(void)
{
```

```
    vec3   color;

    // Compute p(Q(v))
    color  = texture3(brdfP, inDirection);

    // Compute q(Q(h))
    color *= texture3(brdfQ, haDirection);

    // Compute p(Q(l))
    color *= texture3(brdfP, outDirection);

    // Multiply result by the scaling factor for the BRDF textures
    color *= scaleFactor;

    // Multiply result by the texture-mapped surface color
    color *= texture3(surfaceColor, gl_TexCoord0);

    // Multiply result by the light intensity
    color *= lightColor * vec3 (lightIntensity);

    // Clamp and add alpha component to produce final color
    gl_FragColor = vec4 (clamp(color, 0.0, 1.0), 1.0);
}
```

Screen Shots

(Not available at the time of this writing.)

6.7 Wood Shader

Our first working OpenGL 2.0 shader was the brick pattern discussed in 6.4. Our second working shader was the wood shader discussed in this section. We didn't start with an existing RenderMan wood shader, but instead came up with our own way of thinking about wood and how we could implement the necessary OpenGL Shading Language programs given the constraints of our implementation. Still, our model of wood ended up being remarkably similar to the wood shader defined in Advanced RenderMan

Our "theory" of wood is as follows:

- The wood is composed of light and dark areas alternating in concentric cylinders surrounding a central axis.
- Noise is added to warp the cylinders to create a more natural-looking pattern
- The center of the "tree" is taken to be the y-axis
- Throughout the wood there is a high-frequency grain pattern of dark streaks. These streaks model the look of an open-grained wood. These streaks are roughly parallel to the y-axis.

Application Setup

The wood shaders don't require too much from the application. The application is expected to pass in a vertex position and a normal per vertex using the usual OpenGL entry points. In addition, the vertex shader takes a light position and a scale factor that are passed in as uniform variables. The fragment shader takes a grain size, a color for the dark wood, and a color spread value that are also passed in as uniform variables.

The uniform variables needed for the wood shaders are initialized with the following C code:

```
float colorSpread[3] = { 0.3, 0.15, 0.0 };
float lightPosition[3] = { 0.0, 0.0, 4.0 };
float darkColor[3] = { 0.6, 0.3, 0.1};
float ringSize = 1.0;
float scale = 2.0;

glLoadUniform1fv(glGetUniformLocation(woodProgramObject,
       "RingSize"), &ringSize);
glLoadUniform1fv(glGetUniformLocation(woodProgramObject,
       "Scale"), &scale);
glLoadUniform3fv(glGetUniformLocation(woodProgramObject,
       "LightPosition"), &lightPosition[0]);
glLoadUniform3fv(glGetUniformLocation(woodProgramObject,
       "DarkColor"), &darkColor[0]);
glLoadUniform3fv(glGetUniformLocation(woodProgramObject,
       "colorSpread"),&colorSpread[0]);
```

Vertex Shader

The wood vertex shader is actually pretty simple. To compute the reflected light intensity, we need to transform the vertex into world coordinates as shown in the first line of code. Then we apply a scale factor to the incoming vertex value. This is done to scale the object so that the grain pattern will appear at a visually pleasing size in the final rendering. The scale factor can be changed for each object rendered, to make the

object bigger or smaller relative to our “tree”. *Position* will be varied across the primitive and used in our fragment shader for the position-dependent shading calculations.

The incoming normal is transformed by the current OpenGL normal matrix and then normalized. This vector, the light position, and the view direction are used to compute the light intensity from a single white light source. The light intensity is scaled by a factor of 1.5 in order to light the scene more fully. (The fragment shader will ultimately be responsible for clamping the final colors to the range [0,1].) Finally, the transformed vertex position is computed and stored in *gl_Position* as required by the OpenGL Shading Language.

```
varying float LightIntensity;
varying vec3 Position;
uniform vec3 LightPosition;
uniform float Scale;

void main(void)
{
    vec4 pos = gl_ModelViewMatrix * gl_Vertex;
    Position = vec3(gl_Vertex) * Scale;
    vec3 tnorm = normalize(gl_NormalMatrix * gl_Normal);
    LightIntensity =
        dot(normalize(LightPosition - vec3(pos)), tnorm) * 1.5;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Fragment Shader

Most of the interesting work in our wood shader is done by the fragment shader. In this shader, we use the object position and our wood parameter values in order to determine the color of the fragment. First we define our uniform variables:

```
uniform float RingSize;
uniform vec3  DarkColor;
uniform vec3  ColorSpread;
```

RingSize represents the distance in the x-z plane from the middle of one dark band to the next. In order to avoid doing a square root calculation (for performance reasons and because it hasn’t been implemented at the time of this writing), RingSize is not a constant but a value that can be modified interactively to provide a reasonable effect. *DarkColor* is the starting point for the wood color: we will add to this value to build up the color of the lighter wood. *ColorSpread* is the amount of variation that we’ll have between the dark wood and the light wood. Now we define the varying variables to match those in the vertex shader:

```
varying float LightIntensity;
varying vec3 Position;
```

As described in the previous section, the *Position* variable will be the foundation for our fragment calculations. Our shader needs to have the property that it will always produce the same output for a particular value of *Position*, so everything we do to determine our wood pattern algorithmically needs to be

based on this value. The *lightIntensity* value will be used in the final stage of the shader to add in the contribution from the light source. Once our uniform and varying variables have been declared, we begin the main function:

```
void main (void)
{
    vec3 location = Position;
```

The first thing we do in our main function is to declare a variable called *location* and store in it the value of *Position*. The OpenGL Shading Language says that varying variables are read-only as far as the fragment shader is concerned. We will be modifying this interpolated value, so we need to store it in a local variable for our own use.

```
    vec3 floorvec =
          vec3(floor(Position.x * 10), 0, floor(Position.z * 10));
    vec3 noise = Position * 10 - floorvec - 0.5;
    noise *= noise;
    location += noise * 0.12;
```

Now we are getting to some interesting stuff. We want a method of creating some noise in our pattern that is repeatable for a given *Position* value. For performance reasons (and because we haven't implemented the noise() function at the time of this writing), we've come up with a simple method that we can use to perturb our location in a repeatable way. We're simply going to use some of the low-order bits from each of the *x* and *z* components of *Position*.

To do this, we create a vector called floorvec that contains the integer part of the x and z values multiplied by 10. (We don't need to waste cycles doing a floor() function on the y component because noise along the tree axis is irrelevant for our purposes.) Once floorvec is computed, we subtract it from the *Position* vector multiplied by 10. The result will be that the x and z components each contain a value between 0 and 1, and y will contain 0. If we subtract 0.5 from each component, we will have values that range from -0.5 to 0.5.

This noise function has a bit of a problem. As the input values (*Position.x* and *Position.z*) change, the output values will go from -0.5 to 0.5, then it will drop suddenly back to -0.5. This discontinuity will cause jarring visual artifacts, so we smooth it out by squaring the noise value as shown in the next line of code. We now have noise values that vary from 0 to 0.25 and the function is continuous. This noise vector is then multiplied by an empirically chosen scale factor of 0.12 and added to the local variable that contains the position. In this way, we are "warping" the current position by a small amount in order to sample the algorithmically defined texture at a different location.

After this, we need to compute the distance from the tree's axis in order to see where we fall relative to our pattern of concentric circles. We do this in the *x*-*z* plane by squaring the values of the *x* and *z* components. To determine where we are in the ring pattern, we take this distance and divide by *RingSize* (which we've already squared to avoid performing a square root). The integer part of the result (*ring*) will tell us which ring we're in, and the fractional part will tell us how far we are into that ring:

```
    float dist = location.x * location.x + location.z * location.z;
    float ring = dist / RingSize;
```

We base the "brightness" to be added on the fractional portion of our variable *ring*. This brightness value will range from 0 to 1 as we move across a ring. If we do nothing further, we will have sharp contrast in the wood color as our brightness goes from very near 1 back down to 0. Because of our sampling method, this will result in severe aliasing artifacts. A cheap solution is to subtract the brightness from 1.0 once it gets higher than 0.5. Now the brightness will be 0 at the start of a ring, increase to 0.5 at the center of the ring, then decrease back to 0 at the start of the next ring. To get the initial value for our color we multiply the brightness by 0.5 and by the *ColorSpread* uniform variable, then add this to our *DarkColor* value:

```
float brightness = fract(ring);
if (brightness > 0.5)
    brightness = (1.0 - brightness);
vec3 color = DarkColor + 0.5 * brightness * ColorSpread;
```

To add more interest to the pattern, we compute another brightness factor based on a value of *ring* multiplied by 7. This time, the computed brightness is subtracted from our accumulated color. The simple antialiasing is applied in the same fashion. There are better ways to do antialiasing, but this method provides acceptable results in terms of quality versus performance.

```
brightness = fract(ring*7);
if (brightness > 0.5)
    brightness = 1.0 - brightness;
color -= 0.5 * brightness * ColorSpread;
```

Finally, we add a third level of brightness variation based on a value of ring multiplied by 47. (Although the brightness pattern repeats, the values of 7 and 47 that we've used make it pretty difficult to discern the repeated pattern.) This time, we simulate an open-grained pattern in the wood with tiny streaks that are parallel to the y-axis. Every so often, the conditional test in the fourth line of code below will be true, and a some amount of brightness will be subtracted from our accumulated color. This computation is not antialiased like the previous ones, so in some orientations of the object we will see these streaks start off nearly the same color as the surrounding wood, then get darker and darker, and then disappear entirely. These streaks will show up more or less often if you very the constants in the third and fourth line of code below.

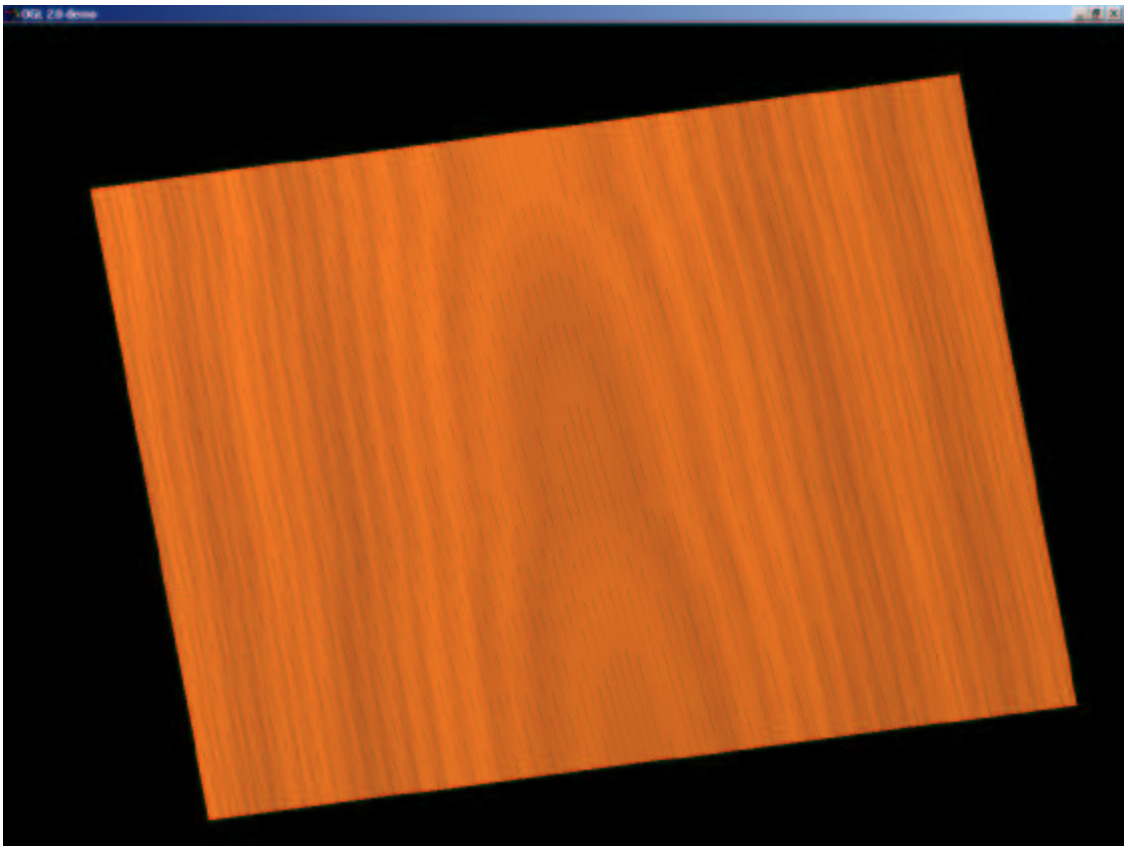
```
brightness = fract(ring*47);
float line = fract(Position.z + Position.x);
float snap = floor(line * 30) * (1.0/30.0);
if (line < snap + 0.004)
    color -= 0.5 * brightness * ColorSpread;
```

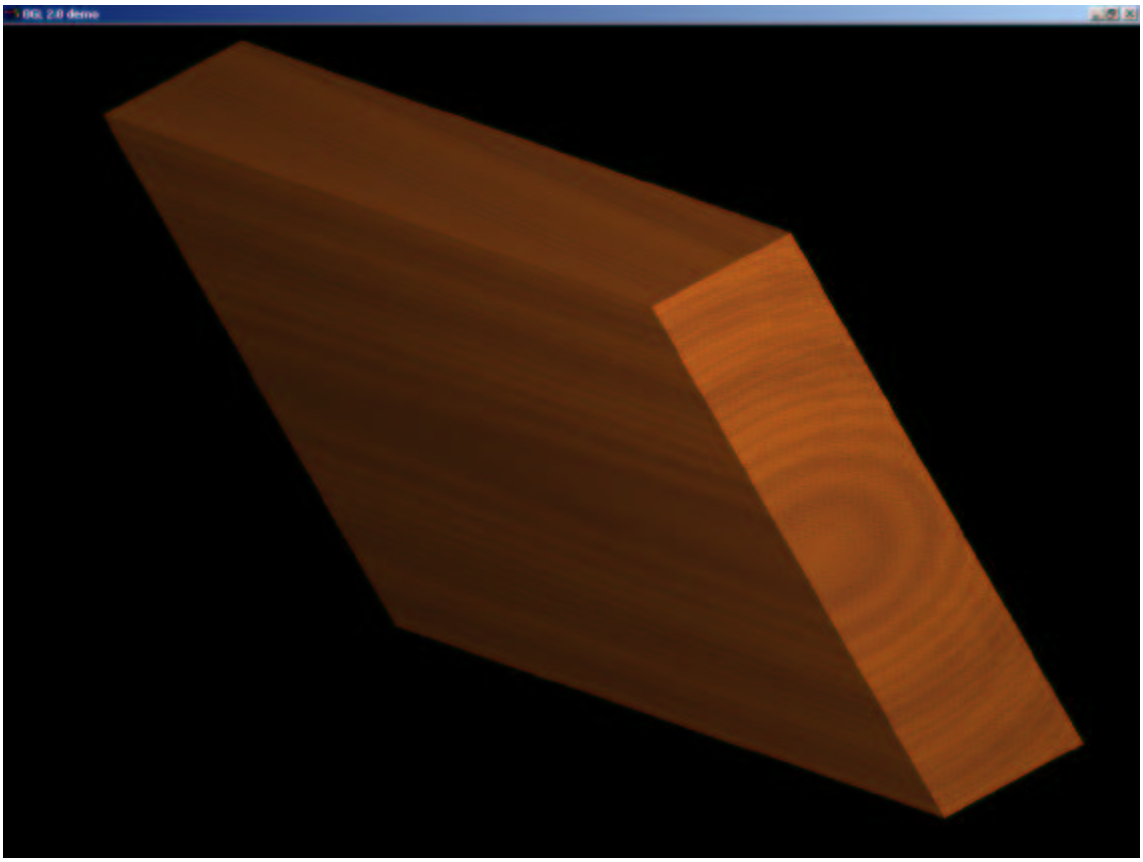
At this point, we've created the wood color for the fragment. We multiply it by the varying variable LightIntensity in order to simulate diffuse reflection, and then we clamp the color to the range [0,1]. All that's needed now is to add an alpha component of 1.0 and assign the result to the built-in variable *gl_FragColor*:

```
    color *= LightIntensity;
    color  = clamp(color, 0, 1);

    gl_FragColor = vec4(color, 1.0);
}
```

Screen shots





6.8 Acknowledgements

I would like to thank my colleagues at 3Dlabs for their assistance with the OpenGL 2.0 effort in general and for help in developing this paper. Specifically, John Kessenich, Barthold Lichtenbelt, and Steve Koren have been doing amazing work implementing the OpenGL Shading Language compiler, linker, and object support in the OpenGL ICD. John developed the wood shader shown in section 6.7 and supplied the screen shots. Barthold developed an OpenGL Shading Language demo program, excerpts of which are included in sections 6.2 and 6.4. Steve worked on the brick shader and supplied the screen shots for it.

Dave Baldwin of 3Dlabs was the original architect of the OpenGL Shading Language. Very little in the language has changed since his original proposal. Dale Kirkland, Jeremy Morris, Phil Huxley, and Antonio Tejada of 3Dlabs have been involved in many of the OpenGL 2.0 discussions and have provided a wealth of good ideas and encouragement as we have moved forward. Other members of the 3Dlabs driver development teams in Egham, U.K., Huntsville, AL, and Austin, TX have contributed as well. The 3Dlabs executive staff should be commended for having the vision to move forward with the OpenGL proposal and the courage to allocate resources to its development. Thanks to Osman Kent, Neil Trevett, Jerry Peterson, and John Schimpf in particular.

Numerous other people have been involved in the OpenGL 2.0 discussions. We would like to thank our colleagues and fellow ARB representatives at ATI, SGI, NVIDIA, Intel, Microsoft, Evans & Sutherland, IBM, Sun, Apple, Imagination Technologies, Dell, Compaq, and HP for contributing to discussions and for helping to move the process along.

A big thank you goes to the software developers who have taken the time to talk with us, send us email, or answer survey questions on OpenGL.org. Our ultimate aim is to provide you with the best possible API for doing graphics application development, and the time that you have spent telling us what you need has been invaluable. A few ISVs lobbied long and hard for certain features, and they were able to convince us to make some significant changes to the original OpenGL 2.0 proposal. Thanks, all you software developers, and keep telling us what you need!

Finally, a debt of gratitude is owed to the designers of the C programming language, the designers of RenderMan, and the designers of OpenGL, the three standards that have provided the strongest influence on our efforts. Hopefully, the OpenGL Shading Language will continue their traditions of success and excellence.

6.9 Further information

Our intention is to continue providing developers with the latest and greatest information about the OpenGL 2.0 effort at the 3Dlabs web site, http://www.3dlabs.com. Look for an “OpenGL 2.0” link on our home page. On the OpenGL 2.0 page, you will find the latest versions of the OpenGL 2.0 white papers, slides from recent OpenGL 2.0 presentations, extension specifications, and sample code and shaders. Interested readers should peruse all the white papers, since they go into detail about the many facets of the OpenGL 2.0 effort. This white paper will be updated around the time of SIGGRAPH 2002 in order to fix errors and include updated code and screen shots

Chapter 7

API Design

Michael McCool

SMASH Metaprogramming Shader API

Michael D. McCool, Qin Zheng and Tiberiu Popa

Computer Graphics Lab
Department of Computer Science
University of Waterloo

5th April 2002

1 Introduction

Modern graphics accelerators have embedded programmable components in the form of vertex and fragment shading units. Current APIs permit specification of the programs for these components using an assembly-language level interface. Shader compilers are available [5, 7] but these read in an external string specification, which can be inconvenient. Both the DX9 and OpenGL 2.0 proposals also use an external high-level shading language separate from the host language, with actual shader programs specified either in strings or in separate files.

It is possible, using C++, to define a high-level shading language directly in the API. This would permit more direct interaction with the specification of textures and parameters, simplifies implementation (an LR parser is not required), and permits on-the-fly generation and manipulation of shader programs to specialize them [2] as needed. Precompiled shader programs could still be used simply by compiling and running a C++ program defining an appropriate shader and dumping a precompiled binary representation. However, parameter naming and binding are simplified if the application program and the shader program are compiled together, since objects defining named parameters and textures can be accessed by the shader definition directly.

SMASH supports two levels of API: an OpenGL-like low-level C-compatible API, whose calls are indicated with the prefix `sm`, and a high-level C++ API whose calls and types are indicated with the prefix `Sm`. In this document, we focus on the C++ shader API. The low-level shader API, which the high-level shader API compiles to, is based on the DX9 assembly language but with a function call-based API in the style of ATI's OpenGL vertex shader extensions.

This whitepaper describes a work in progress and the detailed syntax of the final system may differ from what is shown below. Please access our website at

`http://www.cgl.uwaterloo.ca/Projects/rendering/`

for more up to date information. The syntax described here also differs from that documented for earlier versions of SMASH.

2 Metaprogramming Parser

String based shading languages need a separate parsing step, usually based on an LR grammar parser-compiler such as YACC or Bison, to convert the syntax of the shader program to a parse tree. However, using a metaprogramming API, the shader program is specified using a

sequence of function calls originating directly in the application program. The API then interprets this sequence of calls as a set of "tokens" to be used to generate a parse tree which can in turn be compiled by an on-the-fly compiler backend in the API driver library. Expressions in a shading language can be parsed and type-checked at the application program's compile time using operator overloading. To do this, overloaded operator functions are defined that construct symbolic parse trees for the expressions rather than executing computations directly. The "variables" in the shader are in fact smart reference-counting pointers to nodes in directed acyclic graphs representing expressions, and each operator allocates a new node and uses smart pointers to refer to its children. The reference-counting smart pointers implement a simple garbage collection scheme which in this case is adequate to avoid memory leaks. Compiling expressions in this way eliminates a large chunk of the grammar for the shading language; the API gets passed a complete parse tree for expressions directly, and does not have to build it itself by parsing a flat sequence of tokens. Each assignment in sequence is recorded as a statement in the shader program and buffered until the entire sequence of commands has been received. When the shader program is complete, code generation and optimization is performed by the driver, resulting internally in machine language which is prepared for downloading to the specified shader unit when the shader program is bound.

Eventually, when shading units support control constructs, the shading language can be extended with API calls that embed tokens for control keywords in the shader statement sequence: `SmIF(`*`cond`*`)`, `SmWHILE(`*`cond`*`)`, `SmENDIF()`, etc. Complex statements are received by the API as a sequence of such calls/tokens. For instance, a WHILE statement would be presented to the API as a WHILE token (represented by an `SmWHILE(`*`cond`*`)` function call; note the parameter, which refers to an expression parse tree for the condition), a sequence of other statements, and a matching ENDWHILE token. Use of these constructs can be wrapped in macros to make the syntax slightly cleaner (i.e. to hide semicolons and function call parenthesis):

```
#define SM_IF(cond)              SmIF(cond);
#define SM_ELSEIF(cond)          SmELSEIF(cond);
#define SM_ELSE                  SmELSE();
#define SM_ENDIF                 SmENDIF();
#define SM_WHILE(cond)           SmWHILE(cond);
#define SM_ENDWHILE              SmENDWHILE();
#define SM_DO                    SmDO();
#define SM_UNTIL(cond)           SmUNTIL(cond);
#define SM_FOR(init,cond,inc)    SmFOR(init,cond,inc);
#define SM_ENDFOR                SmENDFOR();
```

Since expression parsing (and type checking) is done by C++ at the compile time of the host language, all that is needed to parse structured control constructs is a straightforward recursive-descent parser. This parser will traverse the buffered token sequence when the shader program is complete, generating a full parse tree internally. Code generation can then take place in the usual way.

Although true conditional execution and looping are not yet available in any commercial real-time shading system, such control constructs can theoretically be implemented efficiently in the context of a long texture lookup latency with either a recirculating pipeline or a multithreaded shading processor.

3 Testbed

Our high-level shader API is built on top of SMASH, a testbed we have developed to experiment with possible next-generation graphics hardware features and their implementation.

This system is modular. Pipelines can be built with any number of shading processors or other types of modules (such as rasterizers or displacement units) chained together in sequence or in parallel. The API has to deal with the fact that any given SMASH system might have a variable number of shading units, and that different shading units might have slightly different capabilities (for instance, vertex shaders might not have texture units, and fragment shaders may have a limited number of registers and operations). These restrictions are noted when a system is built and the shader compiler adapts to them.

The API currently identifies shaders by pipeline depth. In the usual case of a vertex shader and a fragment shader, the vertex shader has depth 0 and the fragment shader has depth 1. When a shader program is downloaded, the packet carrying the program information has a counter. If this counter is non-zero, it is decremented and the packet is forwarded to the next unit in the pipeline. Otherwise, the program is loaded and the packet absorbed. Modules in the pipeline that do not understand a certain packet type are also supposed to forward such packets without change. A flag in each packet indicates whether or not packets should be broadcast over parallel streams or not; shader programs are typically broadcast. In this fashion shader programs can be sent to any shader unit in the pipe. Sequences of tokens defining a shader program are defined using a sequence of API calls inside a matched pair of `SmBeginShader(`*`shaderlevel`*`)` and `SmEndShader()` calls. Once defined, a shader can be loaded using the `SmBindShader(`*`shaderobject`*`)` call.

When a program is running on a shader unit, vertex and fragment packets are rewritten by that unit. The system supports packets of length up to 255 words, not counting a header which gives the type and length of each packet. Each word is 32 bits in length, so shaders can have up to 255 single-precision inputs and outputs.[1] Type declarations in shader parameter declaration can be used to implicitly define packing and unpacking of shorter parameters to conserve bandwidth when this full precision is not necessary. Other units, such as the rasterizer and compositing module, also need to have packets formatted in a certain way to be meaningful; in particular, the rasterizer needs the position of a vertex in a certain place in the packet (at the end, consisent with the order of parameter and vertex calls). These units also operate by packet rewriting; for instance, a rasterizer parses sequences of vertices according to the current geometry mode, reconstructs triangles from them, and converts them into streams of fragments.

4 Parameter Binding

It is convenient to support two different techniques for passing parameters to shaders. For semi-constant parameters, the use of named parameters whose values can be changed at any time and in any order is convenient. We will give these parameters the special name of *attributes* and will reserve the word *parameters* for values specified per-vertex. A named attribute is created simply by constructing an object of an appropriate type:

```
// create named transformation attributes
SmAttributeAffXform3x4f modelview;
SmAttributeProjXform4x4f perspective;
// create named light attributes
SmAttributeColor3f light_color;
SmAttributePoint3f light_position;
```

The constructor of these classes makes appropriate calls into the API to allocate state for these attributes, and the destructor makes calls to deallocate this state. Operators overloaded on

[1] In practice, to support antialiasing at the fragment level, extra overhead may also be required to transmit differentials. Using half precision for two differential values per parameter doubles the amount of space required for each parameter and cuts the maximum number of parameters in half.

these classes are used in the shader definition to access these values. When a shader definition uses such an attribute the compiler notes this fact and arranges for the current value of each such attribute to be bound to a constant register in the shader unit when the shader program is loaded. Attributes of all types can be associated with stacks for save/restore. For convenience, operators are overloaded on both the classes themselves and pointers to them so either implicit or explicit allocation and deallocation can be used. Arrays of attributes are supported with special classes as well.

For parameters whose values change at every vertex, for efficiency we have chosen to make the order of specification of these parameters important. In immediate mode, a sequence of generic multidimensional parameter calls simply adds parameters to a packet, which is sent off as a vertex packet when the vertex call is made (after adding the last few parameters given in the vertex call itself). This is actually supported directly in the low-level API. For instance, suppose we want to pass a tangent vector, a normal, and a texture coordinate to a vertex shader at the vertices of a single triangle. In immediate mode we would use calls of the form

```
smBegin(SM_TRIANGLES);
    smVector3fv(tangent[0]);
    smNormal3fv(normal[0]);
    smTexCoord2fv(texcoord[0]);
    smVertex3fv(position[0]);

    smVector3fv(tangent[1]);
    smNormal3fv(normal[1]);
    smTexCoord2fv(texcoord[1]);
    smVertex3fv(position[1]);

    smVector3fv(tangent[2]);
    smNormal3fv(normal[2]);
    smTexCoord2fv(texcoord[2]);
    smVertex3fv(position[2]);
smEnd();
```

The types given above are optional, and are checked at runtime only in a special “test mode”. High-performance runtime mode, which is invoked by linking to a different version of the API library, simply assumes the types match. The generic parameter call `smParam*` can be used in place of `smVector*`, `smNormal*`, etc. Vertex and parameter arrays are of course also supported for greater efficiency.

Declarations inside every shader definition provide the necessary information to enable the system to make sure the necessary named parameters are loaded into the shader and that unnamed parameters are unpacked in the right order for vertices and fragments. The API must also ensure that when a shader is bound that any texture objects it uses are also bound. The C++ API itself also uses classes to wrap low-level texture objects so that within a shader definition a texture lookup can be specified as if it were an array access. As with attributes, operators are overloaded on pointers to texture objects as well as on texture objects themselves. To avoid confusion with the `[]` array-access operator, a special class is defined for arrays of texture objects.

In the rest of the paper we will give a sequence of examples to demonstrate the high-level C++ shader API.

5 Modified Phong Lighting Model

Consider the modified (reciprocal) Blinn-Phong lighting model:

$$L_o \quad = \quad \Big(k_d[\mathbf{u}] + k_s[\mathbf{u}](\hat{\mathbf{n}} \cdot \hat{\mathbf{h}})^q\Big) \max(0, (\hat{\mathbf{n}} \cdot \hat{\mathbf{l}}))I_\ell / r_\ell^2$$

where $\hat{\mathbf{v}}$ is the normalized view vector, $\hat{\mathbf{l}}$ is the normalized light vector, $\hat{\mathbf{h}} = \mathrm{norm}(\hat{\mathbf{v}} + \hat{\mathbf{l}})$ is the normalized half vector, $\hat{\mathbf{n}}$ is the normalized surface normal, I_ℓ is the light source intensity, r_ℓ is the distance to light source, $k_d[\mathbf{u}]$, $k_s[\mathbf{u}]$, and q are parameters of the lighting model, and $\mathbf{u}$ is a 2D surface texture coordinate.

We will implement this using per-pixel computation of the specular lobe and texture mapping of k_d and k_s. In general, the notation $t[\mathbf{u}]$ indicates a filtered and interpolated texture lookup, not just a simple array access (although, if the texture object access modes are set up appropriately, it can be made equivalent to a simple array access).

5.1 Vertex Shader

This shader computes the model-view transformation of position and normal, the projective transformation of view-space position into device space, the halfvector, and the irradiance. These values will be ratiolinearly interpolated by the rasterizer and the interpolated values will be assigned to the fragments it generates. The rasterizer expects the last parameter in each packet to be a device-space 4-component homogeneous point.

```
SmShader phong0 = SmBeginShader(0); {
  // declare vertex parameters, in order given
  SmInputTexCoord2f ui;    // texture coords
  SmInputNormal3f nm;      // normal vector (MCS)
  SmInputPoint3f pm;       // position (MCS)

  // declare outputs, in order sent
  SmOutputVector3f hv;     // half-vector (VCS)
  SmOutputTexCoord2f uo;   // texture coords
  SmOutputNormal3f nv;     // normal (VCS)
  SmOutputColor3f ec;      // irradiance
  SmOutputPoint4f pd;      // position (HDCS)

  // compute VCS position
  SmRegPoint3f pv = modelview * pm;
  // compute DCS position
  pd = perspective * pv;
  // compute normalized VCS normal
  nv = normalize(nm * inverse(modelview));
  // compute normalized VCS light vector
  SmRegVector3f lvv = light_position - pv;
  SmRegParam1f rsq = 1.0/(lvv|lvv);
  lvv *= sqrt(rsq);
  // compute irradiance
  SmRegParam1f ct = max(0,(nv|lvv));
  ec = light_color * rsq * ct;
  // compute normalized VCS view vector
  SmRegVector3f vv = -normalize(pv);
  // compute normalized VCS half vector
```

```
  hv = normalize(lvv + vv);
  // pass through texture coordinates
  uo = ui;
} SmEndShader();
```

We do not need to provide prefixes for the utility functions `normalize`, `sqrt`, etc. since they are distinguished by the type of their arguments. In our examples we will also highlight, using boldface, the use of externally declared attribute and texture objects.

The types `SmInput*` and `SmOutput*` are classes whose constructors call allocation functions in the API. The order in which these constructors are called provides the necessary information to the API on the order in which these values should be unpacked from input packets and packed into output packets. Temporary registers can also be declared explictly as shown, although of course the compiler will declare more internally in order to implement expression evaluation, and will optimize register allocation as well. These "register" declarations, therefore, are really just smart pointers to expression parse trees.

SMASH permits allocation of named transformation matrices in the same manner as other attributes. Matrices come in two varieties, representing affine transformations and projective transformations. When accessing a matrix value, the matrix can be bound either as a transpose, inverse, transpose inverse, adjoint, or transpose adjoint. The adjoint is useful as it is equivalent to the inverse within a scale factor. However, we do not need to declare these bindings explicitly since simply using a object representing a named attribute or matrix stack is enough to bind it to the shader and for the API to arrange for that parameter to be sent to the shader processor when updated. The symbolic functions `transpose`, `inverse`, `adjoint`, etc. cause the appropriate version of the matrix to be bound to the shader.

5.2 Fragment Shader

This shader completes the Blinn-Phong lighting model example by computing the specular lobe and adding it to the diffuse lobe. Both reflection modes are modulated by specular and diffuse colors that come from texture maps using the previously declared texture objects `phong_kd` and `phong_ks`. The rasterizer automatically converts 4D homogenous device space points (specifying the positions of vertices) to normalized 3D device space points (specifying the position of each fragment). The 32-bit floating-point fragment depth z comes first in the output packet to automatically result in the correct packing and alignment for x and y.

The Phong exponent is specified here as a named attribute. Ideally, we would antialias this lighting model by clamping the exponent as a function of distance and curvature [1], but we have not implemented this functionality.

```
SmShader phong1 = SmBeginShader(1); {
  // declare inputs, in order given
  SmInputVector3f hv;      // half-vector (VCS)
  SmInputTexCoord2f u;     // texture coordinates
  SmInputNormal3f nv;      // normal (VCS)
  SmInputColor3f ec;       // irradiance
  SmInputParam1f pdz;      // fragment depth (DCS)
  SmInputParam2us pdxy;    // fragment 2D position (DCS)

  // declare outputs, in order sent
  SmOutputColor3f fc;      // final fragment color
  SmOutputParam1f fpdz;    // final fragment depth
  SmOutputParam2us fpdxy; // final fragment 2D position

  // compute texture-mapped diffuse lobe
```

```
  fc = phong_kd[u];
  // compute texture-mapped specular lobe
  fc += phong_ks[u]
        * pow((normalize(hv)|normalize(nv)),phong_exp);
  // multiply lighting model by irradiance
  fc *= ec;
  // pass through depth and position
  fpdz = pdz;
  fpdxy = pdxy;
} SmEndShader();
```

Since it is not needed for bit manipulation, we use the operator "`|`" to indicate the inner (dot) product between vectors rather than bitwise OR. We also use the operator "`&`" for the cross product, which has the advantage that the triple product can be easily defined. However, parentheses should be always be used around dot and cross products when they are used in other expressions due to the low precendence of these operators.

Matrix multiplications are indicated with the "`*`" operator. In matrix-vector multiplications if the vector appears on the right it is interpreted as a column and if on the left as a row. For the most part this eliminates the need to explicitly specify transposes. Since we have chosen to use "`*`" to represent matrix multiplication and not the more abstract operation of typed transformation application, to transform a normal you have to explicitly specify the use of the inverse and use the normal as a row vector. Use of the "`*`" operator on a pair of tuples of any type results in pairwise multiplication. Use of "`*`" between a 1D scalar value and any nD tuple results in scalar multiplication.

6 Separable BRDFs and Material Mapping

A bidirectional reflection distribution function f is in general a 4D function that relates the differential incoming irradiance to the differential outgoing radiance.

$$L_o(\mathbf{x}, \hat{\omega}_o) \quad = \quad \int_\Omega f(\hat{\omega}_o, \mathbf{x}, \hat{\omega}_i)\ \max(0, \hat{\mathbf{n}} \cdot \hat{\omega}_i)\, L_i(\mathbf{x}, \hat{\omega}_i)\, d\hat{\omega}_i.$$

Relative to a point source, which would appear as an impulse function in the above integral, the BRDF can be used as a lighting model:

$$L_o(\mathbf{x}, \hat{\omega}_o) \quad = \quad f(\hat{\omega}_o, \mathbf{x}, \hat{\omega}_i)\ \max(0, \hat{\mathbf{n}} \cdot \hat{\omega}_i)\, I_\ell / r_\ell^2.$$

In general, it is impractical to tabulate a general BRDF. A 4D texture lookup would be required. Fortunately, it is possible to approximate BRDFs by factorization. A numerical technique called homomorphic factorization [4] can be used to find a separable approximation to any shift-invariant BRDF:

$$f_m(\hat{\omega}_o, \hat{\omega}_i) \quad \approx \quad p_m(\hat{\omega}_o)\, q_m(\hat{\mathbf{h}})\, p_m(\hat{\omega}_i)$$

In this factorization, we have chosen to factor the BRDF into terms dependent directly on incoming direction, outgoing direction, and half vector direction, all expressed relative to the local surface frame. Other parameterizations are possible but this one seems to work well in many circumstances and is easy to compute.

To model the dependence of the reflectance on surface position, we can sum over several BRDFs, using a texture map to modulate each BRDF. We call this *material mapping*:

$$f(\hat{\omega}_o, \mathbf{u}, \hat{\omega}_i) \quad = \quad \sum_{m=0}^{M-1} t_m(\mathbf{u})\, f_m(\hat{\omega}_o, \hat{\omega}_i)$$

$$= \sum_{m=0}^{M-1} t_m(\mathbf{u})\, p_m(\hat{\omega}_o)\, q_m(\hat{\mathbf{h}})\, p_m(\hat{\omega}_i).$$

When storing them in a fixed-point format, we also rescale the texture maps to maximize precision:

$$f(\hat{\omega}_o, \mathbf{u}, \hat{\omega}_i) = \sum_{m=0}^{M-1} \alpha_m t'_m(\mathbf{u}) p'_m(\hat{\omega}_o) q'_m(\hat{\mathbf{h}}) p'_m(\hat{\omega}_i).$$

6.1 Vertex Shader

Here is a vertex shader to set up material mapping using a separable BRDF decomposition for each material.

```
SmShader hf0 = SmBeginShader(0); {
  // declare vertex parameters, in order given
  SmInputTexCoord2f ui;  // texture coords
  SmInputVector3f t1;    // primary tangent
  SmInputVector3f t2;    // secondary tangent
  SmInputPoint3f pm;     // position (MCS)

  // declare output parameters, in order given
  SmOutputVector3f vvs;  // view-vector (SCS)
  SmOutputVector3f hvs;  // half-vector (SCS)
  SmOutputVector3f lvs;  // light-vector (SCS)
  SmOutputTexCoord2f uo; // texture coords
  SmOutputColor3f ec;    // irradiance
  SmOutputPoint4f pd;    // position (HDCS)

  // compute VCS position
  SmRegPoint3f pv = modelview * pm;
  // compute DCS position
  pd = perspective * pv;
  // transform and normalize tangents
  t1 = normalize(modelview * t1);
  t2 = normalize(modelview * t2);
  // compute normal via a cross product
  SmRegNormal3f nv = normalize(t1 & t2);
  // compute normalized VCS light vector
  SmRegVector3f lvv = light_position - pv;
  SmRegParam1f rsq = 1.0/(lvv|lvv);
  lvv *= sqrt(rsq);
  // compute irradiance
  SmRegParam1f ct = max(0,(nv|lvv));
  ec = light_color * rsq * ct;
  // compute normalized VCS view vector
  SmRegVector3f vv = -normalize(pv);
  // compute normalized VCS half vector
  SmRegVector3f hv = norm(lvv + vv);
  // project BRDF parameters onto SCS
  vvs = SmRegVector3f((vvv|t1),(vvv|t2),(vvv|nv));
  hvs = SmRegVector3f((hvv|t1),(hvv|t2),(hvv|nv));
```

```
  lvs = SmRegVector3f((lvv|t1),(lvv|t2),(lvv|nv));
  // pass through texture coordinates
  uo = ui;
} SmEndShader();
```

6.2 Fragment Shader

The fragment shader completes the material mapping shader by using an application program loop (running on the host) to generate an unrolled shader program. Note that a looping construct is not required in the shader program to implement this. In fact, the API does not even see the loop, only the calls it generates.

```
SmShader hf1 = SmBeginShader(1); {
  // declare parameters, in order given
  SmInputVector3f vv;      // view-vector (SCS)
  SmInputVector3f hv;      // half-vector (SCS)
  SmInputVector3f lv;      // light-vector (SCS)
  SmInputTexCoord2f u;     // texture coordinates
  SmInputColor3f ec;       // irradiance
  SmInputParam1f pdz;      // fragment depth (DCS)
  SmInputParam2us pdxy;    // fragment position (DCS)

  // declare outputs, in order sent
  SmOutputColor3f fc;      // fragment color
  SmOutputParam1f fpdz;    // fragment depth
  SmOutputParam2us fpdxy; // fragment position

  // intialize total reflectance
  fc = SmColor3f(0.0,0.0,0.0);
  // sum up contribution from each material
  for (int m = 0; m < M; m++) {
     fc += hf_mat[m][u] * hf_alpha[m]
           * hf_p[m][vv] * hf_q[m][hv] * hf_p[m][lv];
  }
  // multiply by irradiance
  fc *= ec;
  // pass through fragment position and depth
  fpdz = pdz;
  fpdxy = pdxy;
} SmEndShader();
```

Here the texture array objects `hf_mat`, `hf_p`, and `hf_q` should have been previously defined, along with the normalization factor attribute array `hf_alpha`. The texture objects `hf_p` and `hf_q` should have been set up as cube maps so unnormalized direction vectors can be used directly as texture parameters.

7 Marble and Wood

To implement marble, wood, and similar materials, we have used the simple parameterized model for such materials proposed by John C. Hart et al. [3]. This model is given by

$$t(\mathbf{x}) \quad = \quad \sum_{i=0}^{N-1} \alpha_i |n(2^i \mathbf{x})|,$$

$$\begin{aligned} \mathbf{u} &= \mathbf{x}^T \mathsf{A}\mathbf{x} + t(\mathbf{x}), \\ k_d(\mathbf{x}) &= c_d[\mathbf{u}], \\ k_s(\mathbf{x}) &= c_s[\mathbf{u}]. \end{aligned}$$

where n is a bandlimited noise function such as Perlin noise [6], t is the "turbulence" noise function synthesized from it, A is a 4×4 symmetric matrix giving the coefficients of the quadric function $\mathbf{x}^T\mathsf{A}\mathbf{x}$, c_d and c_s are a 1D MIP-mapped texture maps functioning as filtered color lookup tables, and $\mathbf{x}$ is the model-space (normalized homogeneous) position of a surface point. The outputs need to be combined with a lighting model, so we will combine them with the Phong lighting model (we could just as easily have used separable BRDFs and material maps, with one color lookup table for each).

Generally speaking we would use fractal turbulence and would have $\alpha_i = 2^i$; however, for the purposes of this example we will permit the α_i values to vary to permit further per-material noise shaping and will bind them to named attributes. Likewise, various simplifications would be possible if we fixed A (marble requires only a linear term, wood only a cylinder) but we have chosen to give an implementation of the more general model and will bind A to a named attribute.

The low-level SMASH API happens to have support for Perlin noise, generalized fractal noise, and generalized turbulence built in, so we do not have to do anything special to evaluate these noise functions. If we had to compile to a system without noise hardware, we would store a periodic noise function in a texture map and then could synthesize aperiodic fractal noise by including appropriate rotations among octaves in the noise summation.

7.1 Vertex Shader

The vertex shader sets up the Phong lighting model, but also computes half of the quadric as a linear transformation of the model space position. Note that this can be correctly ratiolinearly interpolated.

```
SmShader pnm0 = SmBeginShader(0); {
  // declare vertex parameters, in order given
  SmInputNormal3f nm;  // normal vector (MCS)
  SmInputPoint3f pm;   // position (MCS)

  // declare outputs, in order sent
  SmOutputPoint4f ax;  // coeffs x MCS position
  SmOutputPoint4f x;   // position (MCS)
  SmOutputVector3f hv; // half-vector (VCS)
  SmOutputVector3f nv; // normal (VCS)
  SmOutputColor3f ec;  // irradiance
  SmOutputPoint4f pd;  // position (HDCS)

  // transform position
  SmRegPoint3f pv = modelview * pm;
  pd = perspective * pv;
  // transform normal
  nv = normalize(nm * inverse(modelview));
  // compute normalized VCS light vector
  SmRegVector3f lvv = light_position - pv;
  SmRegParam1f rsq = 1.0/(lvv|lvv);
  lvv *= sqrt(rsq);
  // compute irradiance
```

```
  SmRegParam1f ct = max(0,(nv|lvv));
  ec = light_color * rsq * ct;
  // compute normalized VCS view vector
  SmRegVector3f vv = -normalize(pv);
  // compute normalized VCS half vector
  hv = norm(lvv + vv);
  // pass through texture coordinates
  uo = ui;

  // projectively normalize position
  x = projnorm(pm);
  // compute half of quadric
  ax = quadric_coefficients * x;
} SmEndShader();
```

7.2 Fragment Shader

The fragment shader completes the computation of the quadric and the turbulence function and passes their sum through the color lookup table. Two different lookup tables are used to modulate the specular and diffuse parts of the lighting model, which will permit, for example, dense dark wood to be shinier than light wood (with the appropriate entries in the lookup tables).

```
SmShader pnm1 = SmBeginShader(1); {
  // declare parameters, in order given
  SmInputPoint4f ax;      // coeffs x MCS position
  SmInputPoint4f x;       // position (MCS)
  SmInputVector3f hv;     // half-vector (VCS)
  SmInputVector3f nv;     // normal (VCS)
  SmInputColor3f ec;      // irradiance
  SmInputParam1f pdz;     // fragment depth (DCS)
  SmInputParam2us pdxy;   // fragment 2D position (DCS)

  // declare outputs, in order sent
  SmOutputColor3f fc;     // fragment color
  SmOutputParam1f fpdz;   // fragment depth
  SmOutputParam2us fpdxy; // fragment 2D position

  // compute texture coordinates
  SmRegTexCoord1f u = (x|ax) + turbulence(pnm_alpha,x);
  // compute diffuse and specular colors
  SmRegColor3f kd = pnm_cd[u];
  SmRegColor3f ks = pnm_cs[u];
  // compute Blinn-Phong lighting model
  fc += kd;
  fc += ks * pow((normalize(hv)|normalize(nv)),phong_exp);
  // multiply by irradiance
  fc *= ec;
  // pass through fragment depth and position
  fpdz = pdz;
  fpdxy = pdxy;
} SmEndShader();
```

Acknowledgements

This research was funded by grants from the National Science and Engineering Research Council of Canada (NSERC), the Centre for Information Technology of Ontario (CITO), the Canadian Foundation for Innovation (CFI), the Ontario Innovation Trust (OIT), and finally the Bell University Labs initiative.

References

[1] John Amanatides. Algorithms for the detection and elimination of specular aliasing. In *Proc. Graphics Interface*, pages 86–93, May 1992.

[2] B. Guenter, T. Knoblock, and E. Ruf. Specializing shaders. In *Proc. ACM SIGGRAPH*, pages 343–350, August 1995.

[3] John C. Hart, Nate Carr, Masaki Kameya, Stephen A. Tibbitts, and Terrance J. Coleman. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. In *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 45–53. ACM Press, August 1999. Held in Los Angeles, California.

[4] M. D. McCool, J. Ang, and A. Ahmad. Homomorphic factorization of brdfs for high-performance rendering. In *Proc. SIGGRAPH*, pages 171–178, August 2001.

[5] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In *Proc. SIGGRAPH*, pages 425–432, July 2000.

[6] Ken Perlin. An image synthesizer. In *Proc. SIGGRAPH*, pages 287–296, July 1985.

[7] K. Proudfoot, W. R. Mark, P. Hanrahan, and S. Tzvetkov. A real-time procedural shading system for programmable graphics hardware. In *Proc. ACM SIGGRAPH*, August 2001.