COURSE 17: STATE-OF-THE-ART IN SHADING HARDWARE

CHAPTER 6: THE OPENGL SHADING LANGUAGE

Randi J. Rost

3Dlabs, Inc.

SIGGRAPH 2002 Course Notes

05-August-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 part of the vision 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.

This paper describes several shaders that were presented at SIGGRAPH 2002. The shaders presented were compiled and demonstrated on the 3Dlabs Wildcat VP870 graphics hardware. At the time of this writing, the features of the OpenGL Shading Language are solidifying and implementation efforts are well underway. A number of software vendors are beginning to use the OpenGL Shading Language to write their own shaders. You are invited to keep checking the 3Dlabs web site at http://www.3dlabs.com/support/developer/ogl2 for the latest information about the OpenGL 2.0 effort..

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.

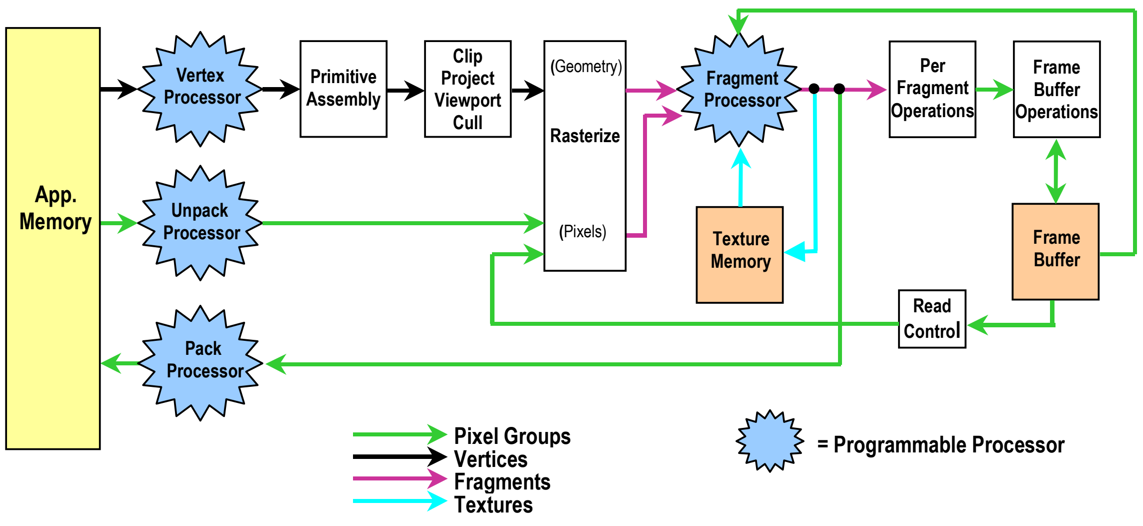
The high level language that is defined as part of the OpenGL 2.0 effort is called the *OpenGL Shading Language*. Some of the significant features of this language are:

- It is integrated intimately with OpenGL 1.4
- It allows incremental replacement of the OpenGL 1.4 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 1.4. 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 1.4, 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 1.4 are shown in white rectangles. To simplify the diagram, some of the OpenGL 1.4 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 1.4 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 1.4: 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 1.4, 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 1.4 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 1.4. 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..

"GL2" Extensions

One of the strengths of OpenGL has been that it has not evolved too quickly or in an incompatible way. Each successive version of OpenGL is effectively a superset of the previous version. Because of this, programs written long ago will work on OpenGL implementations of today.

The process for adding new features to OpenGL has been to create an extension specification for a new feature, implement the extension, get others to implement it, and get software developers to use it in their applications. All new function calls, types, and enumerants defined by the extension are differentiated from core OpenGL through the use of a suffix that is added to function names, types, and enumerants, as well as to the name of the extension itself.

The list of extensions supported by an OpenGL implementation can be determined by calling glGetString with a value of GL_EXTENSIONS. This causes a string to be returned that lists the names of all of the extensions that are supported by the implementation. An extension that is unique to a particular vendor is identified by a suffix that is also unique to that vendor. For instance, extensions that are unique to 3Dlabs use a suffix of "3DL". If two or more vendors agree on a common extension specification, it can be named using the suffix "EXT". This can be used as a signal to applications that the extension is supported by multiple vendors. Using an "EXT" extension rather than a vendor unique extension usually means that the application will be more portable. "EXT" and vendor unique extensions do not need to undergo any review process before they are made available for applications to use.

Once an extension has proved its worth, the body that governs the evolution of OpenGL, the Architecture Review Board (ARB), may vote to change the suffix of the extension to "ARB". The "ARB" suffix indicates that the ARB has reviewed the extension and approves of the way it has been specified. The "ARB" suffix also indicates that anyone implementing the functionality should implement the "ARB" extension rather than any other similar extension. It is often the case that "ARB" extensions are the ones that are given the highest consideration for inclusion into the core OpenGL specification each time the specification is revised. The "ARB" suffix indicates the highest level of portability that can be achieved by an extension prior to it becoming part of the core OpenGL standard. When an ARB extension is added to the core standard, it is often the case that the only change made to the functionality is to drop the "ARB" suffix from function names and enumerants.

The ARB decided to create a new class of extensions as an aid to the implementation and adoption of features that are part of the OpenGL 2.0 vision. This class of extension has the suffix "GL2". The intention is that functionality from the OpenGL 2.0 white papers would be specified and implemented initially as extensions with “GL2” suffixes. This would indicate to application developers that the extension contains capabilities that the ARB thinks are ultimately destined for inclusion in the OpenGL standard. The “GL2” suffix can be used to define extensions that have not yet been implemented or field tested. The intention is to enable both rapid implementation and early adoption. It is expected that “GL2” extensions may include limitations to cope with current hardware, but that these limitations will be removed once the functionality becomes part of core OpenGL. Unlike “ARB” extensions, “GL2” extensions might be revised as issues are found either in specification or in implementation. “GL2” extensions are intended to be a relatively quick way for vendors to come to agreement on extensions that expose new hardware capabilities, as well as a way for software developers to get access to those capabilities in a relatively portable manner.

The API calls that are described in subsequent sections have been defined in “GL2” extension specifications proposed by 3Dlabs and implemented on the 3Dlabs Wildcat VP graphics hardware. At the time of this writing, these “GL2” extensions are being considered by the ARB’s GL2 working group.

Creating and compiling shaders

A shader is created using:

```
GLhandle glCreateShaderObjectGL2 (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_GL2 or GL_FRAGMENT_SHADER_GL2. 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 glLoadShaderGL2 (GLhandle shaderID,
                      GLuint nseg,
                      const GLchar **seg,
                      const GLuint *length)

void glAppendShaderGL2 (GLhandle shaderID,
                        const GLchar *seg,
                        GLuint length)
```

The glLoadShaderGL2 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 *seg* argument is an array of pointers to one or more optionally null-terminated character strings that make up the program. The number of strings in the array is given in *nseg*. The *length* argument is an array of length *nseg* that specifies the length (number of characters) in each string (the null-termination character is not counted as part of the length). Each value in the *length* array can be set to -1 to indicate that the corresponding string is null-terminated. If *length* is 0, all strings in the *seg* array are assumed to be null-terminated.

The glAppendShaderGL2 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 a pointer to an optionally null-terminated character string that is to be added to the end of the specified shader object. The *length* argument has the same meaning as it does for the glLoadShaderGL2 command.

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 glCompileShaderGL2 (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 GLchar *glGetInfoLogGL2 (GLhandle shaderID, GLuint *length)
```

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 typically 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 glDeleteObjectGL2 (GLhandle objectID)
void glDeleteObjectsGL2 (GLsizei 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 glCreateProgramObjectGL2 ()
```

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 objects by using the glDeleteObjectGL2 and glDeleteObjectsGL2 calls.

Shader objects are attached to a program object with the command:

```
GLboolean glAttachShaderObjectGL2 (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 glDetachObjectGL2 (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 glLinkProgramGL2 (GLhandle programID)
GLboolean glUseProgramObjectGL2 (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 glGetInfoLogGL2 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 glUseProgramObjectGL2.

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_GL2 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_GL2 is provided, but no shader of type GL_FRAGMENT_SHADER_GL2 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 glUseProgramObjectGL2 is called. glUseProgramObjectGL2 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 glUseProgramObjectGL2 is called with a handle of 0, all of the OpenGL 2.0 processors are disabled and the OpenGL 1.4 fixed functionality will be used instead.

The program object that is currently in use can be obtained by calling

```
GLhandle glGetHandleGL2(enum pname)
```

with the parameter name GL_PROGRAM_OBJECT_GL2. 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

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 glUseProgramObjectGL2, 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 vec4gl_Vertex;
attribute vec4gl_Color;
attribute vec3gl_Normal;
attribute vec4gl_MultiTexCoord0;// glTexCoord also updates this.
attribute vec4gl_MultiTexCoord1;
attribute vec4gl_MultiTexCoord2;
attribute vec4gl_MultiTexCoord3;
attribute vec4gl_MultiTexCoord4;
attribute vec4gl_MultiTexCoord5;
attribute vec4gl_MultiTexCoord6;
attribute vec4gl_MultiTexCoord7;
attribute vec4gl_SecColor;
attribute vec3gl_FrontMaterial[5];
attribute vec3gl_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 might 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. The number of locations supported by the implementation can be obtained by querying the implementation-dependent constant GL_MAX_VERTEX_ATTRIBS_ARB. Each location is numbered and has room to store up to four components. 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 glVertexAttrib{1234}{sfd}ARB(uint index, T coords);
void glVertexAttrib4NubARB(uint index, T coords)
void glVertexAttrib{123}{sfd}vARB(uint index, T coords);
void glVertexAttrib4{bsifd ubusui}vARB(uint index, T coords);
void glVertexAttrib4N{bsifd ubusui}vARB(uint index, T coords);
```

which load the given value(s) into the attribute slot indicated by *index*. The attributes follow the OpenGL rules for the substitution of default values for missing components. 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 {bsifd ubusui} notation indicates that there is a separate version of the function for expressing data as a byte, short, integer, float, double, unsigned byte, unsigned short, or unsigned int. The T indicates the data type for the data and it corresponds to the selection of short, float, or double in the {sfd} notation and byte, short, integer, float, double, unsigned byte, unsigned short, or unsigned int for the {bsifd ubusui} notation. The v indicates that the function is a pointer to a “vector” (array) of data. Commands containing “N” indicate that values are to be normalized, so signed integers are converted into the range -1.0…1.0 and unsigned integers are converted into the range 0.0…1.0. So for example, the actual function that you would use to supply a three-component float as a vector would be defined as:

```
void glVertexAttrib3fvARB(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 glBindAttributeLocationGL2 (GLhandle program,
                                GLuint location,
                                const ubyte *name,
                                int length)
```

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:

```
glBindAttributeLocation(myProgram, 11, “temperature”, -1);
```

Subsequently, we can call glVertexAttribute to pass a temperature value at every vertex in the following manner:

```
glVertexAttrib1f(11, temperature);
```

The glVertexAttrib* 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 glDrawArrays. The glBindAttributeLocationGL2 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 vertex 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). Typically, uniforms are used to supply state that stays constant for several primitives.

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        = glCreateProgramObjectGL2();
vertexShaderObject   = glCreateShaderObjectGL2(GL_VERTEX_SHADER);
fragmentShaderObject = glCreateShaderObjectGL2(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 both our shaders are defined by a single string, so either one can be used.

```
//
// Hand the source to OpenGL. Use either load or append,
// it doesn't matter.
//
length = strlen(vertexShaderString);
glLoadShaderGL2(vertexShaderObject, 1, &vertexShaderString, &length);
length = strlen(fragmentShaderString);
glAppendShaderGL2(fragmentShaderObject, fragmentShaderString, length);
```

The shaders are now ready to be compiled. For each shader, we call glCompileShaderGL2 and then call glGetInfoLogGL2 in order to see what transpired. glCompileShaderGL2 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 fails.

```
//
// Compile the vertex and fragment shader, and print out
// the compiler log file.
//
compiled = glCompileShaderGL2(vertexShaderObject);
pInfoLog = glGetInfoLogGL2(vertexShaderObject);
printf("%s\n\n", pInfoLog);

compiled &= glCompileShaderGL2(fragmentShaderObject);
pInfoLog = glGetInfoLogGL2(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
//
glAttachShaderObjectGL2(programObject, vertexShaderObject);
glAttachShaderObjectGL2(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 glUseProgramObjectGL2.

```
//
// Link the whole thing together and print out the linker log file
//
linked = glLinkProgramGL2(programObject);
pInfoLog = glGetInfoLogGL2(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 the 3Dlabs' ogl2demo 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 glUseProgramObjectGL2 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 instantiation of the wood shader.

```
int installBrickShader(int change)
{
    //
    // The brick shader example from the OpenGL 2.0 Shading Language
    // White Paper.
    //
    float mt = 0.02f;
    float bc[3] = { 1.0f, 0.3f, 0.2f };
    float mc[3] = { 0.85f, 0.86f, 0.84f };
    float bmw = 0.22f;
    float bmh = 0.15f;
    float mwf = 0.94f;
    float mhf = 0.90f;
    float lightPosition[3] = { 0.0f, 0.0f, 4.0f };

    if (makeShaderActive("brick", change))
    {
        //
        // Find out the locations of the uniforms in our shaders,
        // and load values for them. Note, safer is to do this:
        //
        // location = glGetUniformLocation(ProgramObject, "name")
        // Check for any OpenGL errors. If there are none,
        // glLoadUniform(location, value);
        // Check errors again.
        //
        glLoadUniform3fGL2(glGetUniformLocationGL2(programObject,
                                "brickColor", -1), bc[0], bc[1], bc[2]);
        glLoadUniform3fvGL2(glGetUniformLocationGL2(programObject,
                                "mortarColor", -1),&mc[0]);
        glLoadUniform1fvGL2(glGetUniformLocationGL2(programObject,
                                "brickMortarWidth", -1), &bmw);
        glLoadUniform1fvGL2(glGetUniformLocationGL2(programObject,
                                "brickMortarHeight", -1), &bmh);
        glLoadUniform1fGL2(glGetUniformLocationGL2(programObject,
                                "mwf", -1), mwf);
        glLoadUniform1fGL2(glGetUniformLocationGL2(programObject,
                                "mhf", -1), mhf);
        glLoadUniform3fvGL2(glGetUniformLocationGL2(programObject,
                                "LightPosition", -1), &lightPosition[0]);

        return 1;
    }
    else
        return 0;
}
```

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 *OpenGL Shading Language Specification*, available at the 3Dlabs web site (http://www.3dlabs.com/support/developer/ogl2). 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:

void	for functions that do not return a value
bool	a conditional type, taking on values of **true** or **false**
int	a 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 2×2 floating point matrix for vertex and fragment languages only
mat3	a 3×3 floating point matrix for vertex and fragment languages only
mat4	a 4×4 floating point matrix for vertex and fragment languages only

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

The OpenGL Shading Language defines the following data type qualifiers:

< default >	local read/write memory
const	for variables, a compile time constant; it cannot be written to
const	for function parameters, a read only parameter
attribute	linkage between a vertex shader and OpenGL for per-vertex data
uniform	linkage between a shader and OpenGL for per-primitive data
varying	linkage between a vertex shader and a fragment shader for interpolated data
output	for function parameters for pass by reference

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 vec3zAxis = 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 (color, normal, texture coordinate, etc.) 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 vec4lightPosition;
```

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 determining register reuse) using the C [] operator. The size must be specified. Some examples are:

```
uniform vec4lightPosition[4];
varying vec3material[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. 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)
{
...
}
```

The reserved word *kill* can be used within a fragment shader to cease processing and exit without producing an output value.

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

Fragment processing functions are only available in shaders intended for use on the fragment processor:

- Texture Access
- Derivative
- Level-of-Detail
- 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. Because the texture pattern is generated algorithmically, this type of shader is sometimes referred to as a *procedural* shader.

For this example, the vertex shader will do a simple lighting calculation using a single light source that is specified by the application. The vertex shader will also compute both an eye space, object space, and clip space positions for each vertex. The eye space position will be used within the vertex shader to do the lighting calculation, since the light position will be specified in eye coordinates. The clip position is the transformed vertex position that will be passed on through the rendering pipeline to allow for operations such as clipping, culling, viewport mapping and ultimately, rasterization.

The object space position will be passed on to the fragment shader for use in generating the procedural texture. With procedural textures, it is important to have a repeatable function, so that each location on the object has its texture computed the same way in every frame, no matter how the object is rotated or where the viewpoint is. Using the object space position as the input for our procedural texture function allows us to redraw the object in the same way each frame, and makes it look like the object was carved out of a solid texture.

Within the fragment shader, we'll use the object space position at each fragment to determine how to color the fragment. With this value, and the values that are supplied by the application for brick color, brick height, brick width, mortar color, mortar thickness, and so on, we'll be able to algorithmically compute the brick texture pattern to be used at each fragment. Once the base color is computed, the light intensity value that was computed by the vertex shader and interpolated across the primitive will be applied.

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 application code for setting up the brick shader is shown at the very end of section 6.2.

Once the shaders have been installed and the uniform variables have been provided, the application is expected to send a normal and the vertex position for each vertex that is to be drawn. The current values for the model-view matrix, the model-view-projection matrix, and the normal matrix will all be accessed from within the vertex shader. Because the brick pattern is entirely procedurally generated, the fragment shader does not access any of OpenGL's current state.

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 eye 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 eye coordinate position (*pos*) from the uniform variable that holds our light position in eye 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 eye 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 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."

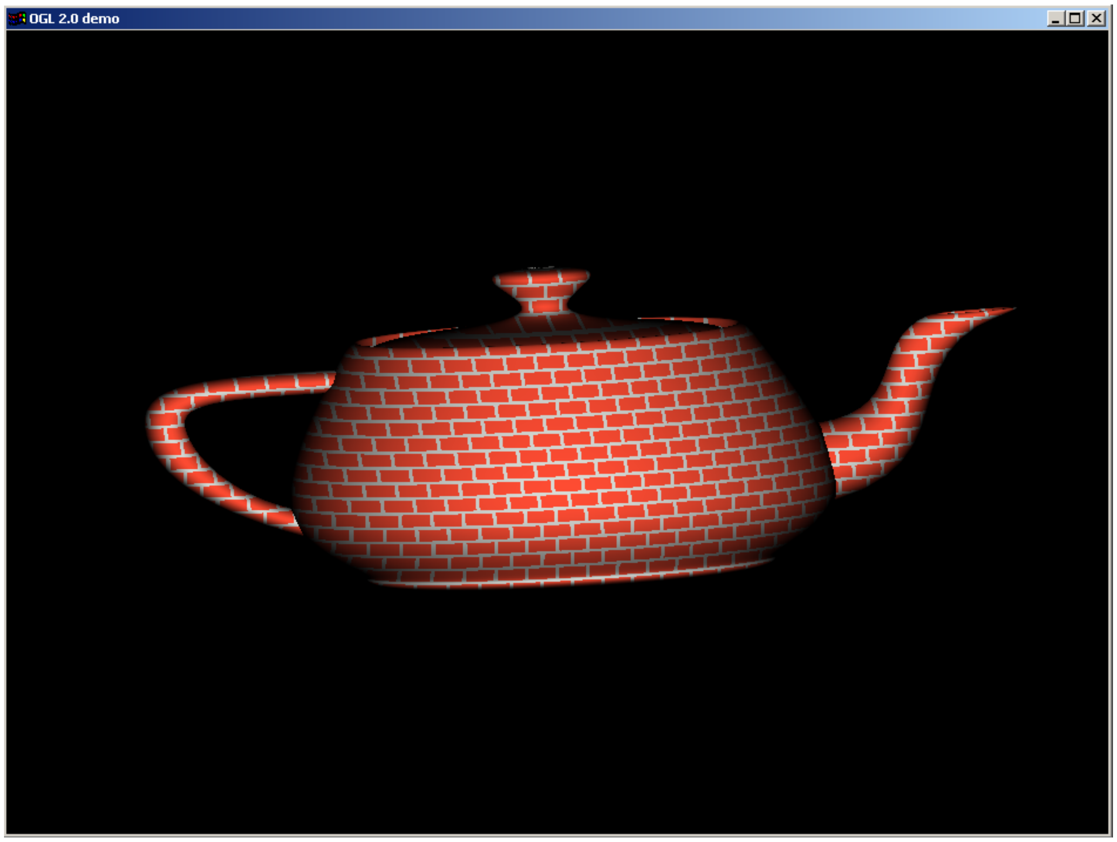
Finally, we compute the color of the fragment and store it in a temporary variable. The built-in function mix() is used to choose the brick color or the mortar color, depending on the value of *w*h.* Since *w* and *h* can only have values of 0.0 (mortar) or 1.0 (brick), we will chose the brick color 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.*

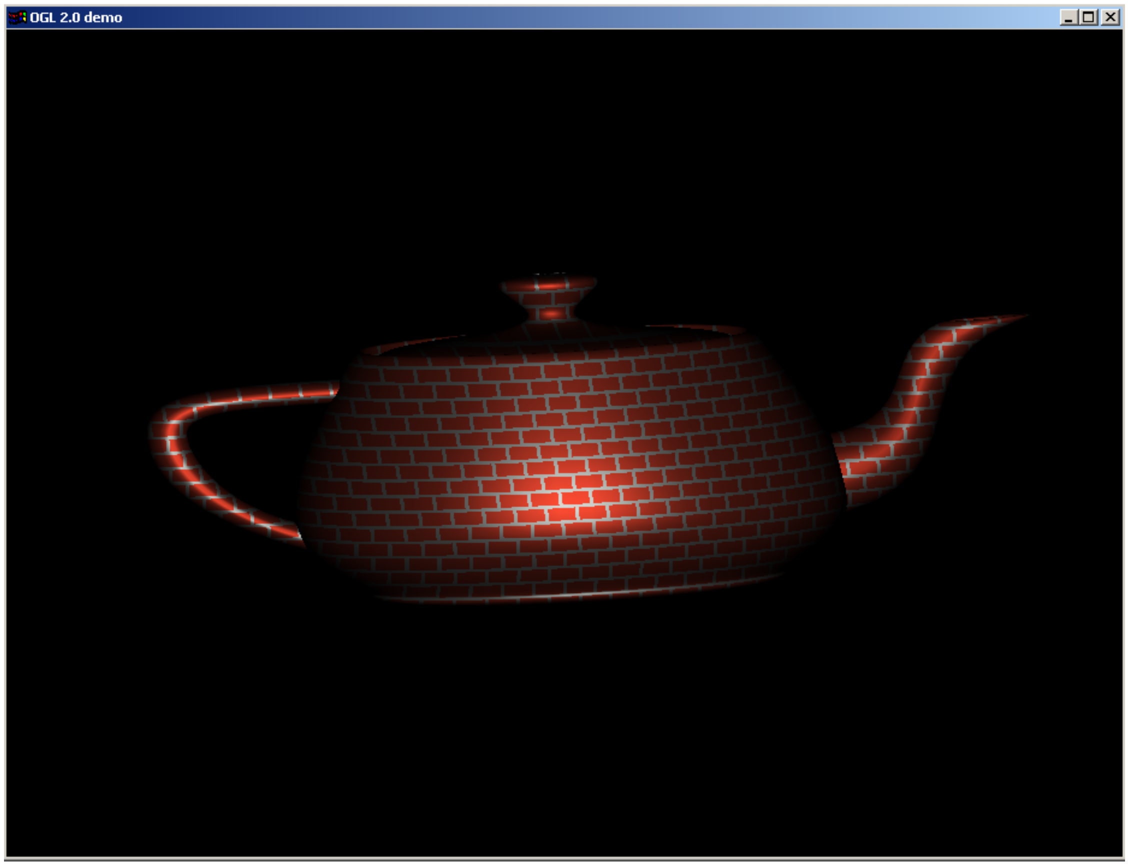
One possible improvement to this shader is to notice that we never actually used the *y* component of our varying variable position. This value was interpolated but never used. Since hardware implementations will have a fixed number of interpolators available, we would be making better use of our scarce 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 main difference between the two shaders is their input and output. Vertex shaders get input from the application in the form of vertex data, and they output all of the data needed to specify a single vertex in the form of varying variables that are write-only for the vertex shader. After rasterization, the fragment processor receives as input the interpolated data at each fragment in the form of read-only varying variables. After processing this data, a fragment shader produces an output value that is can be sent to the back end of the OpenGL rendering pipeline to ultimately be rendered in the frame buffer..

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.

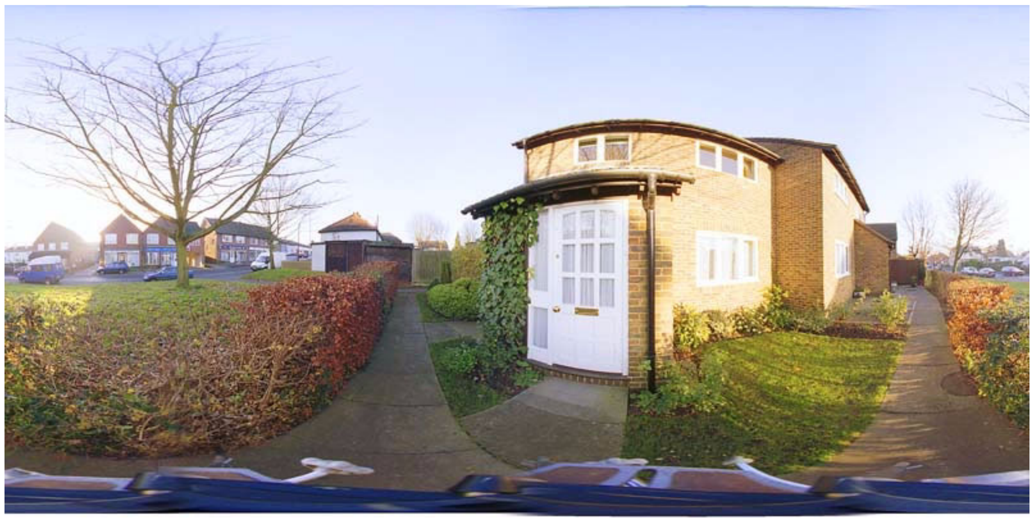



OGL 2.0 demo

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 a single equirectangular texture map. This type of texture can be obtained from a real-world environment using photography techniques, or it can be created to represent a synthetic environment. An example is shown below. (Image courtesy Jerome Dewhurst, jerome@photographica.co.uk, http://www.photographica.co.uk.)



Whatever means is used to obtain an image, the result is a single image that spans 360^o horizontally and 180^o vertically. The image is also distorted as you move up or down from the center of the image. This distortion is done deliberately so that you will see a reasonable representation of the environment if you "shrink-wrap" this texture around the object that you're rendering.

The key to this type of environment mapping is to produce a pair of angles that are used to index into the texture. An altitude angle is computed by determining the angle between the reflection direction and the XZ plane. This altitude angle will vary from 180^o (reflection is straight up) to -180^o (reflection is straight down). The sine of this angle will vary from 1.0 to –1.0, and we'll be able to use this fact to get a texture coordinate in the range of [0,1].

An azimuth angle is determined by projecting the reflection direction onto the XZ plane. The azimuth angle will vary from 0^o to 360^o, and this will give us the key to get a second texture coordinate in the range of [0,1].

The following OpenGL 2.0 shaders combine bump mapping and environment mapping in order to render an object. The altitude and azimuth angles are computed to determine s and t values for indexing into our environment textures. Round "bumps" on the surface are defined procedurally by using the incoming texture coordinate values. If the fragment is within one of these "bumps", the computed s and t values are modified in a non-linear fashion to give the look of a reflection off a rounded bump on the surface.

Application Setup

The application needs to do very little to set up this shader. There is no lighting performed, hence no lighting state is needed. In this example, the environment map will be set up in texture unit 4. For convenience, the X and Y unit vectors that will be used in the dot products within the fragment shader are passed as uniform variables.

```
int installEnvMapShader(int change)
{
    float xunitvec[3] = { 1.0f, 0.0f, 0.0f };
    float yunitvec[3] = { 0.0f, 1.0f, 0.0f };

    glActiveTextureARB(GL_TEXTURE4_ARB);
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, EnvMapName);

    if (makeShaderActive("envmap", change))
    {
        glLoadUniform3fvGL2(glGetUniformLocationGL2(ProgramObject,
                                "Xunitvec", -1), &xunitvec[0]);
        glLoadUniform3fvGL2(glGetUniformLocationGL2(ProgramObject,
                                "Yunitvec", -1), &yunitvec[0]);
        return 1;
    }
    else
        return 0;
}
```

Once the shaders have been installed and the uniform variables have been provided, the application is expected to send a normal, a 2D texture coordinate, and the vertex position for each vertex that is to be drawn. The current values for the model-view matrix, the model-view-projection matrix, and the normal matrix will all be accessed from within the vertex shader. Because the bump pattern is entirely procedurally generated, the fragment shader does not access any of OpenGL's current state.

Vertex Shader

The code below comprises the vertex shader that is used to produce the bumpy/shiny effect.

```
varying vec3 Normal;
varying vec3 EyeDir;

void main(void)
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord0 = gl_MultiTexCoord0;
    Normal = normalize(gl_NormalMatrix * gl_Normal);
    EyeDir = normalize(vec3(gl_ModelViewMatrix * gl_Vertex));
}
```

The goal of this vertex shader is to produce two values that will be interpolated across each primitive: a surface normal and an eye direction. These two values will allow us to compute an accurate reflection direction at each fragment, and from the reflection direction, we will be able to compute the necessary altitude and azimuth angles as well as the distortion caused by our procedural bump pattern.

The transformed position of the vertex is computed in the first line of the program in the usual way. Since we will need the texture coordinate in the fragment shader to produce our procedural bumps, we must copy the value of the standard vertex attribute gl_MultiTexCoord0 into the built-in varying variable gl_TexCoord0. We could have defined our own varying to use instead, but this is just as good. We transform and normalize the incoming normal and then we compute the eye direction based on the current model-view matrix and the incoming vertex value. This eye direction vector is normalized as well.

Fragment Shader

The following code contains the fragment shader that is used to do environment mapping with a procedurally generated bump pattern:

```
uniform vec3 Xunitvec;
uniform vec3 Yunitvec;

varying vec3 Normal;
varying vec3 EyeDir;

const float Density = 6.0;
const float Coeff1 = 5.0;
float Size = 0.15;

void main (void)
{
    // Compute reflection vector. Should really normalize
    // EyeDir and Normal, but this is OK for some objects.

    vec3 reflectDir = reflect(EyeDir, Normal);

    // Compute cosine of altitude and azimuth angles

    vec2 index;
```

```
        index.1 = dot(normalize(reflectDir), Yunitvec);
        reflectDir.y = 0.0;
        index.0 = dot(normalize(reflectDir), Xunitvec);

        // Compute bump locations

        vec2 c = Density * vec2(gl_TexCoord0);
        vec2 offset = fract(c) - vec2(0.5);

        float d = offset.x * offset.x + offset.y * offset.y;
        if (d >= Size)
            offset = vec2(0.0);

        // Translate index values to (0.5, 0.5) and make offset non-linear

        index = (index + 1.0) * 0.5 - Coeff1 * offset * offset * offset;

        // Do a lookup into the environment map.

        vec3 envColor = texture3(4, index);

        gl_FragColor = vec4(envColor, 1.0);
    }
```

The varying variables Normal and EyeDir are the values generated by the vertex shader and then interpolated across the primitive. To get truly precise results, these values should be normalized again in the fragment shader. Skipping the normalization gives us a little better performance, and the quality is acceptable for certain objects.

The application has set up the uniform variables Xunitvec and Yunitvec with the proper values for computing our altitude and azimuth angles. First, we compute our altitude angle by normalizing the reflectionDir vector and performing a dot product with the Yunitvec. Becuase both vectors are unit vectors, this gives us a cosine value for the desired angle that ranges from -1.0 to 1.0. Setting the y component of our reflection vector to 0.0 causes it to be projected onto the X-Z plane. We normalize this new vector to get the cosine of our azimuth angle. Again, this value will range from -1.0 to 1.0.

After this, we use the incoming texture coordinate values as the basis for computing a procedural bump pattern. Our *Density* value tells us how many bumps there are on the surface. By multiplying with our incoming texture coordinate, we can use this value to spread the bumps out further or make them closer together. Next, we determine an offset by taking the fractional part of our modified texture coordinate and subtracting 0.5 from each component. To determine whether or not we're inside a "bump", we compute the distance between the offset and the center of our "bump" ($x^2 + y^2$). If this distance value is greater than the value we've set to represent the bump diameter (*Size*), we'll just set the offset to 0.0.

At this point, we combine our azimuth/altitude values with our bump offset to create an index into our texture map. We add 1.0 to the index values and multiply by 0.5 to get index values that range from 0.0 to 1.0. From this computed index we subtract a somewhat arbitrary function of the offset value that was chosen to give us some non-linearity in the reflection off the bumps. This doesn't produce the correct reflection from rounded bumps, but it's a pretty good fake.

With our index values set, all we need to do is look up the value in the texture map, create a vec4 by adding an alpha value of 1.0, and send the final fragment color on for further processing.

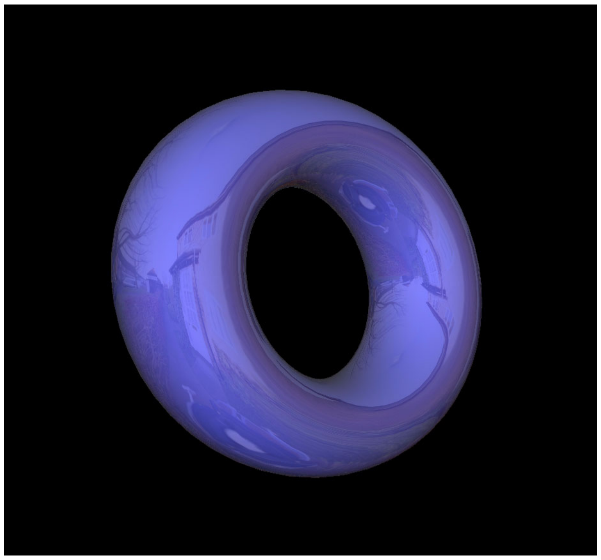
It should be pointed out that a couple of other shortcuts were taken to keep this shader short. First, when we add the offset function, we don't do any clamping. This is fine for the s (azimuth) texture coordinate, since we just wrap around to the other side, and the texture has been set up to enable wrapping. It's also fine outside the bumps, because the offset value is 0.0. But within the bumps, it is possible to wrap the t (altitude) texture coordinate within the bumps. This produces a reflection of the ground near the tops of the bumps on the top of the object, and a reflection of sky on the bottoms of some bumps near the bottom of the object, this could be eliminated by clamping the index values to the range [0,1] after the offset value has been added.

Also, the value for the azimuth angle is not computed correctly. The cosine of the reflection direction is 0.0 when the reflection direction is equal to the eye direction. As the reflection direction changes to 90^o, the cosine value increases to 1.0. As the reflection direction passes 90^o, the cosine value decreases down to 0.0. Instead of treating this properly we just use the cosine value directly. The result is that as the reflection direction approaches 90^o, we use texture values nearer and nearer the edge of the texture. Then, as the angle surpasses 90^o, instead of wrapping to the other side of the texture, we use values near the same edge, gradually getting further from the edge. Furthermore, since the texture covers 360^o in the horizontal direction, we really should multiply the s coordinate by 0.5 (e.g., on the front of the object, we should only see the middle 180^o of the texture, not the whole 360^o).

This could also be treeated properly with a bit more code. I've written another shader that does environment mapping only (no procedural bumps). In this shader, I adjust the azimuth value based on the sign of the z-component of the reflection direction. I multiply the computed value by 0.5, and I do things exactly the same way if the z component of the reflection direction is greater than 0.0. If it's less than 0.0, I multiply my computed value by -0.5 and add 1.0. This gives me values in the range of 0.75 to 1.25. I use these values directly as my texture coordinates, since the values greater than 1.0 will wrap around to the other side of the texture.

Screen Shots

The first image below shows the teapot rendered with the shaders described above. The second image shows a torus rendered with a second set of shaders that combines a diffusely lit base color with the same environment map.



6.6 BRDF Shader

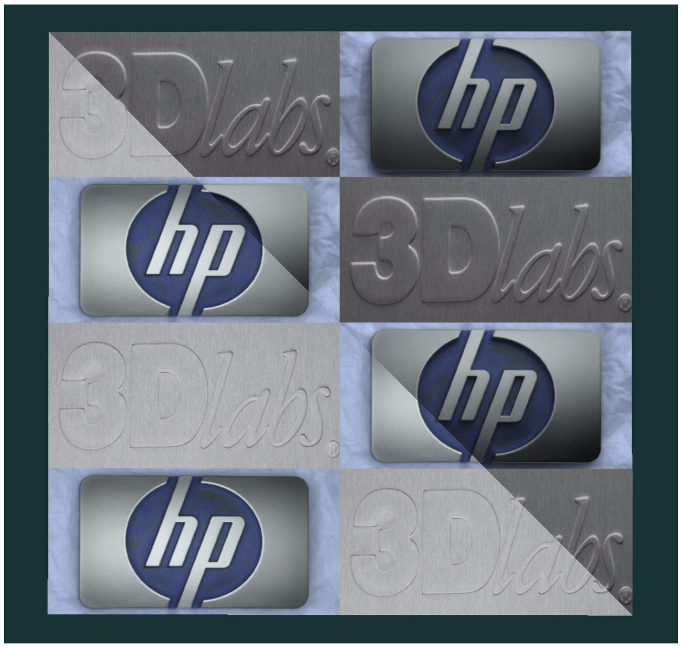
In order to model more physically realistic surfaces, we must go beyond the simplistic lighting/reflection model that is built into OpenGL. For some time, computer graphics researchers have been rendering images with a more realistic reflection model called the bidirectional reflectance distribution function, or BRDF. Instruments have been developed to measure the BRDF of real materials. These BRDF measurements can be sampled to produce texture maps that can be used to reconstruct the BRDF function at run time. A variety of different sampling and reconstruction methods have been devised for rendering BRDF materials.

This section describes the OpenGL Shading Language BRDF shaders that use the Polynomial Texture Mapping (PTM) technique developed by Hewlett-Packard. The shaders presented are courtesy of Brad Ritter, Hewlett-Packard. The BRDF data is from Cornell University. It was obtained by measuring reflections from several types of automotive paints that were supplied by Ford Motor Co.

One of the reasons this type of rendering is important is to achieve realistic rendering of materials whose reflection characteristics vary as a function of view angle and light direction. Such is the case with these automotive paints. To a car designer, it is extremely important to be able to visualize the final “look” of the car even when it is painted with a material whose reflection characteristics vary as a function of view angle and light direction. One of the paint samples tested by Cornell, Mystic Lacquer, has the peculiar property that the color of its specular highlight color changes as a function of viewing angle. This material cannot be adequately rendered using only conventional texture mapping techniques.

PTMs are essentially light-dependent texture maps. PTMs can be used to reconstruct the color of a surface under varying lighting conditions. When a surface is rendered using a PTM, it takes on different illumination characteristics depending on the direction of the light source. Like bump mapping, this helps viewers by providing perceptual clues about the surface geometry. But PTMs are able to go beyond bump maps in that they are capable of capturing surface variations due to self-shadowing and interreflections. PTMs are generated from real materials and are intended to preserve the visual characteristics of the actual materials. Polynomial texture mapping is an image-based technique that does not require bump maps or the modeling of complex geometry.

The image below shows two triangles from a PTM demo developed by Hewlett-Packard. The triangle on the upper right has been rendered using a polynomial texture map, and the triangle on the lower right has been rendered using a conventional 2D texture map. The objects that were used to construct the texture maps were a metallic bezel with the Hewlett-Packard logo on it, and a brushed metal notebook cover with an embossed 3Dlabs logo. As you move the simulated light source in the demo, the conventional texture looks flat and unrealistic, while the PTM texture faithfully reproduces the highlights and surface shadowing that occurs on the real life objects. In the image that I’ve captured, the light source is a bit in front and above the two triangles. The PTM shows realistic reflections, but the conventional texture can only reproduce the lighting effect from a single lighting angle (in this case, as if the light were directly in front of the object).



The PTM technique developed by HP requires as input a set of images of the desired object, with the object illuminated by a light source of a different known direction for each image, all captured from the same viewpoint. For each texel of the PTM, these source images are sampled and a least-squares biquadric curve fit is performed to obtain a polynomial that approximates the lighting function for that texel. This part of the process is partly science and partly art (a bit of manual intervention can improve the end results). The biquadric equation generated in this manner will allow run-time reconstruction of the lighting function for the source material. The coefficients stored in the PTM are A, B, C, D, E, and F as shown in the equation below:

$$Au^2 + Bv^2 + Cuv + Du + Ev + F$$

One use of PTM's is for representing materials with surface properties that vary spatially across the surface. Things like brushed metal, woven fabric, wood, and stone are all materials that reflect light differently depending on the viewing angle and light source direction. They may also have interreflections and self-shadowing. The PTM technique is able to capture these details and reproduce them at runtime. There are two variants for PTMs: luminance (LRGB) and RGB. An LRGB PTM uses the biquadric polynomials to determine the brightness of each rendered pixel. Because each texel in an LRGB PTM has its own bi-quadric polynomial function, the luminance or brightness characteristics of each texel can be unique. An RGB PTM uses a separate biquadric polynomial for each of the three colors, red, green and blue. On today's hardware, RGB PTM's are drawn in three passes, one each for the red, green, and blue component of the final image. This allows objects rendered with an RGB PTM to vary in color as the light position shifts. Thus, color-shifting materials such as holograms can be accurately reconstructed with an RGB PTM.

The key to creating a PTM for these types of spatially varying materials is to capture images of them as lit from a variety of light source directions. Engineers at Hewlett-Packard have developed an instrument to do just that – a dome with multiple light sources and a camera mounted at the top. This device can automatically capture 50 images of the source material from a single fixed camera position as illuminuted by light sources in different positions.

The image data collected with this device is the basis for creating a PTM for the real world texture of a material. These types of PTMs have four degrees of freedom. Two of these will be used to represent the spatially varying characteristics of the material. These two degrees of freedom are controlled using the 2-dimensional texture coordinates. The remaining two degrees of freedom are used to represent the light direction. These are the two independent variables in the biquadric polynomial.

A BRDF PTM is slightly different than a spatially varying PTM. BRDF PTMs are used to model homogeneous materials, that is, they do not vary spatially. BRDF PTMs use two degrees of freedom to represent the light direction and the remaining two degrees of freedom are used to represent the view direction. The parameterized light direction (Lu,Lv) is used for the independent variables of the bi-quadric polynomial and the parameterized view direction (Vu,Vv) is used as the 2D texture coordinate.

There is no single parameterization that works well for all BRDF materials. A further refinement to enhance quality for BRDF PTMs for the materials we are trying to reproduce (automobile paints) is to re-parameterize the light and view vectors as half angle and difference vectors, (Hu,Hv) and (Du,Dv). In the BRDF PTM shader discussed below, (Hu,Hv) is used as the independent variables of the bi-quadric polynomial and (Du,Dv) is used as the 2D texture coordinate. A large part of the function of the vertex shader is to calculate (Hu,Hv) and (Du,Dv).

BRDF PTMs can be created as either LRGB or RGB PTMs. The example below shows how an RGB BRDF PTM is rendered with OpenGL Shading Language Shaders. In addition to the topics already covered, this technique is also an example of multipass rendering.

Application Setup

To render BRDF surfaces using the following shaders, the application needs to set up a few uniform variables. The vertex shader must be provided with values for uniform variables that describe the eye direction (i.e., an infinite viewer) and the position of a single light source (i.e., a local light source). The fragment shader requires the application to provide values for scaling and biasing the six polynomial coefficients. (These values have been prescaled when the PTM was created in order to preserve precision, and so must be rescaled using the scale and bias factors that are specific to that PTM.)

The application is expected to provide four attributes for every vertex. Two of them are standard OpenGL attributes and need not be defined by our vertex program: gl_Vertex (position) and gl_Normal (normal). The other two attributes are a tangent vector and a binormal vector which will 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 glBindAttributeLocationGL2. For instance, if we choose to pass the tangent values in at vertex attribute location 3 and the binormal values in at vertex attribute location 4, we would set up the binding using these lines of code:

```
glBindAttributeLocationGL2(myProgramID, 3, “tangent”, -1);
glBindAttributeLocationGL2(myProgramID, 4, “binormal”, -1);
```

If the variable *tangent* is defined to be an array of three floats, and *binormal* is also defined as an array of three floats, these user-defined vertex attributes can be passed in using the following calls:

```
void glVertexAttribfvARB(3, tangent);
void glVertexAttribfvARB(4, binormal);
```

Alternatively, these values could be passed to OpenGL using user-defined vertex arrays.

Prior to rendering, the application should also set up seven texture maps: three 2D texture maps will hold the A, B, and C coefficients for red, green, and blue compenents of the PTM; three 2D texture maps will hold the D, E, and F coefficients for red, green, and blue components of the PTM; and a 1D texture will hold a lighting function.

This last texture is set up by the application whenever the lighting state is changed. It will be bound to texture unit 3 and just left alone. The light factor texture is solving 4 problems:

- Front facing / back facing discrimination. This texture is indexed with (L dot N) which is positive for front facing vertices and negative for back facing vertices. As a first level of complexity the light texture can solve the front facing / back facing problem by being 1.0 for positive index values and 0.0 for back facing values.
- We'd like to be able to light BRDF PTM shaded objects with colored lights. As a second level of complexity, the light texture (which has three channels, R, G and B) will use a light color instead of 1.0 for positive index values.
- An abrupt transition from front facing to back facing will look awkward and unrealistic on rendered images. As a third level of complexity we apply a gradual transition in the light texture values from 0.0 to 1.0. In the case of the PTM demo from Hewlett-Packard, a sine or cosine curve is used to determine these gradual texture values.
- There is no concept of ambient light for PTM rendering. It can look very unrealistic to render back facing pixels as (0,0,0). Instead of using 0.0 values for negative indices, values such as 0.1 are used.

For each object to be rendered, the application will draw the geometry three times. The first time it will mask off the green and blue channels of the frame buffer, bind the ABC coefficient texture for the red component of the PTM to texture unit 0 , bind the DEF texture for the red component of the PTM to texture unit 0, and draw the geometry. This will cause the red component of the frame buffer to be rendered. The process is repeated to render the green and blue components of the final image.

Vertex Shader

The BRDF PTM vertex shader is shown below. The purpose of this shader is to produce five varying values:

- *gl_Position*, as required by every vertex shader
- *gl_TexCoord0*, which will be used to access our texture maps to get the two sets of polynomial coefficients
- *LdotT*, a float that contains the cosine of the angle between the light direction and the tangent vector
- *LdotB*, a float that contains the cosine of the angle between the light direction and the binormal vector

- *LdotN*, a float that contains the cosine of the angle between the incoming surface normal and the light direction.

```
// VP_PTM_3DLABS_HuHvDuDv_PosInf

uniform vec3 LightPos;
uniform vec3 EyeDir;

attribute vec3 Tangent;
attribute vec3 Binormal;

varying float LdotT;
varying float LdotB;
varying float LdotN;

void main(void)
{
    vec3 lightTemp;
    vec3 halfAngleTemp;
    vec3 tPrime;
    vec3 bPrime;

    // Transform vertex
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    lightTemp = normalize(LightPos - vec3(gl_Vertex));

    // Calculate the Half Angle vector
    halfAngleTemp = normalize(EyeDir + lightTemp);

    // Multiply the Half Angle vector by NOISE_FACTOR
    // to avoid noisy BRDF data
    halfAngleTemp = halfAngleTemp * 0.9;

    // Calculate T' and B'
    //    T' = |T - (T.H)H|
    tPrime = Tangent - (halfAngleTemp * dot(Tangent, halfAngleTemp));
    tPrime = normalize(tPrime);

    //    B' = H x T'
    bPrime = cross(halfAngleTemp, tPrime);

    // Hu = Dot(Light, T')
    // Hv = Dot(Light, B')
    LdotT = dot(lightTemp, tPrime);
    LdotB = dot(lightTemp, bPrime);

    // Du = Dot(HalfAngle, T)
    // Dv = Dot(HalfAngle, B)
    // Remap [-1.0..1.0] to [0.0..1.0]
    gl_TexCoord0.s = dot(Tangent,  halfAngleTemp) * 0.5 + 0.5;
    gl_TexCoord0.t = dot(Binormal, halfAngleTemp) * 0.5 + 0.5;
```

```
      gl_TexCoord0.q = 1.0;

      // "S" Text Coord3: Dot(Light, Normal);
      LdotN = dot(lightTemp, gl_Normal) * 0.5 + 0.5;
}
```

The light source position and eye direction are passed in as uniform variables by the application. In addition to the standard OpenGL vertex and normal vertex attributes, the application is expected to pass in a tangent and a binormal per vertex as described in the previous section. These two user-defined attributes are defined with appropriate names in our vertex shader.

The first line of the vertex shader transforms the incoming vertex value by the current model-view-projection matrix. The next line computes the light source direction for our positional light source by subtracting the vertex position from the light position. Since LightPos is defined as a vec3 and the built-in attribute gl_Vertex is defined as a vec4, we must use the vec3 constructor to obtain the first 3 elements of gl_Vertex prior to doing the vector subtraction operation. The result of the vector subtraction is then normalized and stored as our light direction.

The next line of code computes the half angle by adding together the eye direction vector and the light direction vector and normalizing the result. BRDF data often has noisy data values for extremely large incidence angles (i.e., close to 180^o), so in the next line of code, we avoid the noisy data in a somewhat unscientific manner by applying a scale factor to the half angle. This will effectively cause these values to be ignored.

The next few lines of code compute the 2D parameterization of our half angle and difference vector. The goal here is to compute values for u (LdotT) and v (LdotB) that can be plugged into the biquadric equation in our vertex shader. The technique used here is called Gram-Schmidt orthonormalization and it is described more fully in a paper by Jan Kautz and Michael D. McCool called *Interactive Rendering with Arbitrary BRDF's using Separable Approximations.* (This paper is currently available on the web at http://www.cgl.uwaterloo.ca/Projects/rendering/Papers.) . H (half angle), T' and B' are the orthogonal axis of a coordinate system. T' and B' maintain a general alignment with the original T (tangent) and B (binormal) vectors. Where T and B lie in the plane of the triangle being rendered, T' and B' are in a plane perpendicular to the half angle vector.

Our vertex shader code first computes values for Hu and Hv and places them in the varying variables LdotT and LdotB. These will be plugged into our biquadric equation in the fragment shader as the u and v values. Following this, the values for the built-in varying gl_TexCoord0 are computed. These values hold our parameterized difference vector and will be used to look up the required polynomial coefficients from the texture maps.

Finally, we compute a value that will be used to apply the lighting effect. This value is simply the cosine of the angle between the surface normal and the light direction.

Fragment Shader

The fragment shader for our BRDF PTM surface rendering is shown below:

```
// PP_PTM_3DLABS_RGB
const int ABCtexture   = 0;
const int DEFtexture   = 1;
const int lighttexture = 3;

uniform vec3 ABCscale, ABCbias;
uniform vec3 DEFscale, DEFbias;

varying float LdotT; // passes the computed L*Tangent value
varying float LdotB; // passes the computed L*Binormal value
varying float LdotN; // passes the computed L*Normal value

void main(void)
{
    vec3    ABCcoef, DEFcoef;
    vec3    ptvec;
    float   ptval;

    // Read coefficient values and apply scale and bias factors
    ABCcoef = (texture3(ABCtexture, gl_TexCoord0, 0.0) - ABCbias) * ABCscale;
    DEFcoef = (texture3(DEFtexture, gl_TexCoord0, 0.0) - DEFbias) * DEFscale;

    // Compute polynomial
    ptval = ABCcoef.0 * LdotT * LdotT +
            ABCcoef.1 * LdotB * LdotB +
            ABCcoef.2 * LdotT * LdotB +
            DEFcoef.0 * LdotT +
            DEFcoef.1 * LdotB +
            DEFcoef.2;

    // Turn result into a vec3
    ptvec = vec3 (ptval);

    // Multiply result * light factor
    ptvec *= texture3(lighttexture, LdotN);

    vec3 fragc = clamp(ptvec, vec3 (0.0), vec3 (1.0));

    // Clamp result and assign it to gl_FragColor
    gl_FragColor = vec4 (fragc, 1.0);
}
```

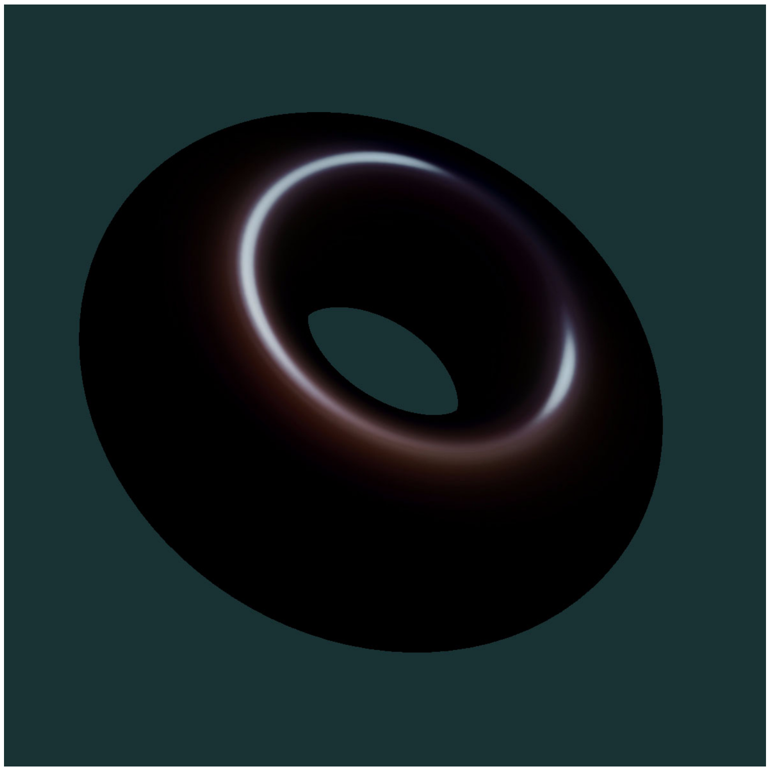
This shader is relatively straightforward if you've digested the information in the previous three sections. The values in the s and t components of gl_TexCoord0 hold a 2D parameterization of the difference vector. This is used to index into both of our coefficient textures and retrieve the values for the A, B, C, D, E, and F coefficients. Since the BRDF PTMs are stored as mipmap textures, we use the version of the texture access function that allows us to specify an LOD bias. However, in this case we're passing an LOD bias of 0.0, so the computed LOD bias will just be used directly. Using vector operations, the six coefficients are scaled and biased by values passed from the application via uniform variables.

These scaled, biased coefficient values are then used together with our parameterized half angle (LdotT and LdotB) in the biquadric polynomial to compute a single floating point value. This value is then replicated to form a vec3. The lighting factor is computed by accessing the 1D light texture using the cosine of the angle between the light and the surface normal. This lighting factor is multiplied by our polynomial vector, the result is clamped to the range [0,1], and an alpha value of 1.0 is provided to produce the final fragment color.

We actually now have the same computed color stored in each of the red, green, and blue components of the fragment color. But depending on which pass we're on, two of the components will be masked out as the fragment is written to the frame buffer and only one will actually be drawn. Doing things in this manner makes it possible for us to use the same fragment shader in each pass.

Screen Shots

The image below shows our BRDF PTM shaders rendering a torus with the BRDF PTM created for the Mystique Lacquer automotive paint. This paint has the peculiar property that its color varies depending on the viewing position. The basic color of this paint is black, but in the orientation captured for the still image below, the specular highlight shows up as mostly white with a reddish-brown tinge on one side of the highlight and a bluish tinge on the other. As the object is moved around, or as the light is moved around, our BRDF PTM shaders will properly render the shifting highlight color.



6.7 Wood Shader

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 to give the appearance of wood that has been sawn, exposing the open grain nature of the wood

Althought the OpenGL Shading Language has a built-in noise function, for this example we are going to use a 3D texture as the source for our noise. There are two main reasons for this: it is faster and it uses fewer instructions (at least on current hardware).

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:

```
int installWoodShader(int change)
{
    //
    // Creates some 3D wood grain with diffuse lighting
    //
    float lightPos[3] = {0.0f, 0.0f, 4.0f};
    float darkwood[3] = {0.4f, 0.2f, 0.07f};
    float lightwood[3]  = {0.6f, 0.3f, 0.1f};
    float scale = 2.0f;
    float ringfreq = 4.0f;
    float lightgrains = 1.0f;
    float darkgrains = 0.0f;
    float grainthreshold = 0.5f;
    float noisescale[3] = {0.5f, 0.1f, 0.1f};
    float noisiness = 3.0;
    float grainscale = 27.0;

    if (!Noise3DTexName)
    {
        glGenTextures(1, &Noise3DTexName);
        make3DNoiseTexture();
        init3DNoiseTexture(Noise3DTexName, Noise3DTexSize, Noise3DTexPtr);
    }

    glActiveTextureARB(GL_TEXTURE6_ARB);
    glBindTexture(GL_TEXTURE_3D_EXT, Noise3DTexName);
    glEnable(GL_TEXTURE_3D);
    if (makeShaderActive("wood", change))
    {
        glLoadUniform1fvGL2(glGetUniformLocationGL2(ProgramObject,
                    "Scale", -1), &scale);
        glLoadUniform3fvGL2(glGetUniformLocationGL2(ProgramObject,
                    "LightPos", -1), &lightPos[0]);
        glLoadUniform3fvGL2(glGetUniformLocationGL2(ProgramObject,
                    "DarkWood", -1), &darkwood[0]);
        glLoadUniform3fvGL2(glGetUniformLocationGL2(ProgramObject,
                    "LightWood", -1), &lightwood[0]);
        glLoadUniform1fvGL2(glGetUniformLocationGL2(ProgramObject,
                    "RingFreq", -1), &ringfreq);
        glLoadUniform1fvGL2(glGetUniformLocationGL2(ProgramObject,
                    "LightGrains", -1), &lightgrains);
        glLoadUniform1fvGL2(glGetUniformLocationGL2(ProgramObject,
                    "DarkGrains", -1), &darkgrains);
        glLoadUniform1fvGL2(glGetUniformLocationGL2(ProgramObject,
                    "GrainThreshold", -1), &grainthreshold);
        glLoadUniform3fvGL2(glGetUniformLocationGL2(ProgramObject,
                    "NoiseScale", -1), &noisescale[0]);
        glLoadUniform1fvGL2(glGetUniformLocationGL2(ProgramObject,
                    "Noisiness", -1), &noisiness);
        glLoadUniform1fvGL2(glGetUniformLocationGL2(ProgramObject,
                    "GrainScale", -1), &grainscale);
        return 1;
    }
    else
        return 0;
}
```

Vertex Shader

The wood vertex shader is actually pretty simple. To compute the reflected light intensity, we need to transform the vertex into eye 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”. An object that has a bounding box of (–1, –1, –1), (1, 1, 1) will need a different scale factor than an object with a bounding box of (0,0,0), (100,100,100) if the procedural texture is expected to look the same on both objects. *Position* will be interpolated across the primitive and used in our fragment shader as the starting point for the position-dependent procedural texture 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 LightPos;
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 = abs(dot(normalize(LightPos - vec3(pos)), tnorm) * 1.5);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Fragment Shader

The fragment shader for our procedurally generated wood is as follows:

```
varying float lightIntensity;
varying vec3 Position;

uniform vec3  LightWood;
uniform vec3  DarkWood;
uniform float RingNoise;
uniform float RingFreq;
uniform float LightGrains;
uniform float DarkGrains;
uniform float GrainThreshold;
uniform vec3  NoiseScale;
uniform float Noisiness;
uniform float GrainScale;

void main (void)
{
    vec3 location;
    vec3 noisevec;
    vec3 color;
    float dist;
    float r;

    noisevec = texture3(6, Position * NoiseScale) * Noisiness;

    location = vec3 (Position + noisevec);

    dist = sqrt(location.x * location.x + location.z * location.z) * RingFreq;

    r = fract(dist + noisevec.0 + noisevec.1 + noisevec.2) * 2.0;

    if (r > 1.0)
        r = 2.0 - r;

    color = mix(LightWood, DarkWood, r);
```

```
        r = fract((Position.x + Position.z) * GrainScale + 0.5);
        noisevec.2 *= r;
        if (r < GrainThreshold)
            color += LightWood * LightGrains * noisevec.2;
        else
            color -= LightWood * DarkGrains * noisevec.2;

        color = clamp(color * lightIntensity, 0.0, 1.0);

        gl_FragColor = vec4(color, 1.0);
    }
```

As you can see, we've parameterized quite a bit of this shader through the use of uniform variables in order to make it easy to manipulate through the application's user interface. As in many procedural shaders, the object position is used as the basis for computing the procedural texture. In this case, the object position is multiplied by NoiseScale (a vec3 that allows us to scale the noise independently in the x, y, and z directions) and the computed value is used as the index into our 3D noise texture. The value that is retrieved from the 3D texture has low-frequency noise in the red channel, noise of twice the frequency (i.e., one octave higher) in the green channel, noise of twice that frequency (i.e., another octave higher) in the blue channel, and noise of twice that frequency in the alpha channel. The noise values obtained from the texture are scaled by the value Noisiness, which allows us to increase or decrease the contribution of the noise.

Our "tree" is assumed to be a series of concentric rings of alternating light wood and dark wood. In order to give some interest to our grain pattern, we'll add the noise vector to our object position. This has the effect of adding our low frequency (first octave) noise to the x coordinate of the position, our second octave noise to the y coordinate, and the third octave noise to the z coordinate. The result will be rings that are still relatively circular but have some variation in width and distance from the center of the tree.

To compute where we are in relation to the center of the tree, we square the x and z components and take the square root of the result. This gives us the distance from the center of the tree. The distance is multiplied by RingFreq, a scale factor that can be used to give the wood pattern more rings or fewer rings.

Following this, we attempt to create a function that goes from 0.0 up to 1.0 and then back down to 0.0. Three octaves of noise are added to the distance value to give more interest to the wood grain pattern. We could compute different noise values here, but the ones we've already obtained will do just fine. Taking the fractional part of the resulting value gives us a function that will range from 0.0 to 1.0. Multiplying this value by 2.0 gives us a function that ranges from 0.0 to 2.0. And finally, by subtracting 1.0 from values that are greater than 1.0, we get our desired function that varies from 0.0 to 1.0 and back to 0.0.

This "sawtooth" function will be used to compute the basic color for the fragment using the built-in mix function. The mix function will do a linear blend of LightWood and DarkWood based on our computed value r.

At this point, we would have a pretty nice result for our wood function, but we attempt to make it a little better by adding a subtle effect to simulate the look of open-grain wood that has been sawn. (You may not be able to see this effect on the screen shots below.)

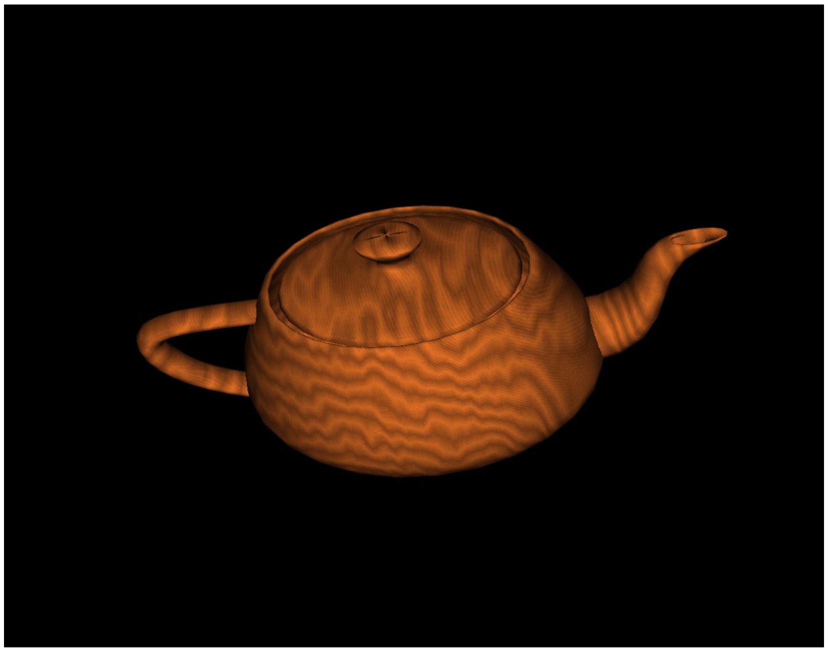
Our desire is to produce streaks that are roughly parallel to the y-axis. This is done by adding the x and z coordinates, multiplying by the GrainScale factor (another uniform variable that can be adjusted to change the frequency of this effect), adding 0.5, and taking the fractional part of the result. Again, this gives us a function that varies from 0.0 to 1.0, but for the default values for GrainScale (27.0) and RingFreq (4.0), this function for r will go from 0.0 to 1.0 much more often than our previous function for r.

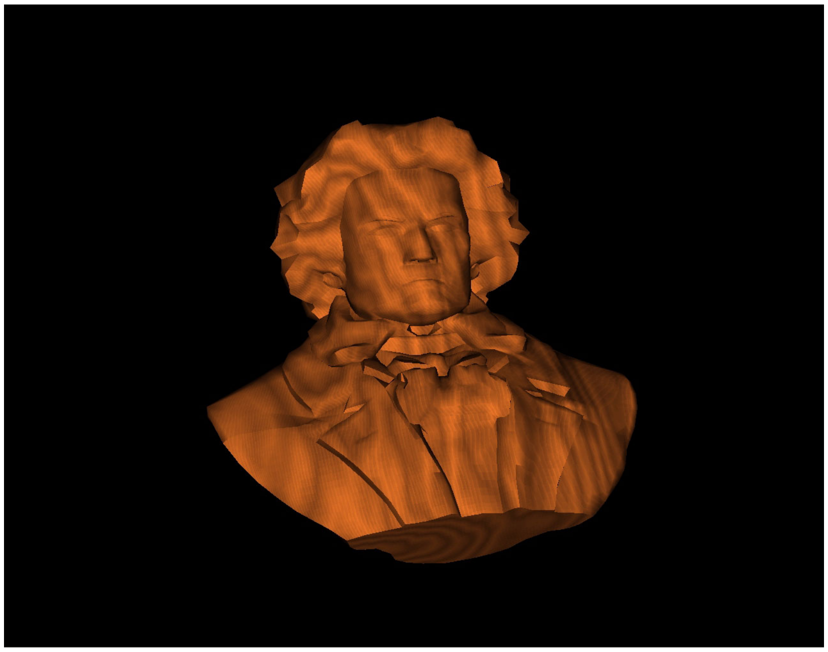
We could just make our “grains” go from light to dark in a linear fashion, but we’ll try to do something a little more subtle. The value of r is multiplied by our third octave noise value to produce a value that increases in a non-linear fashion. Finally, we compare our value of r to the GrainThreshold value (default is 0.5). If the value of r is less than GrainThreshold, we modify our current color by adding to it a value computed by multiplying the LightWood color, the LightGrains color, and our modified noise value. Conversely, if the value of r is greater than GrainThreshold, we modify our current color by subtracting from it a value computed by multiplying the DarkWood color, the DarkGrains color, and our modified noise value. (By default, the value of LightGrains is 1.0 and the value of DarkGrains is 0.0, so we don’t actually see any change if r is greater than GrainThreshold.)

You can play around with this effect and see if it really does help the appearance. It seemed to me that it added to the effect of the wood texture for the default settings I’ve chosen, but there probably is a way to achieve a better effect in a simpler way. Please send me the shader when you do so!

With our final color computed, all that remains is to multiply the color by the interpolated diffuse lighting factor, clamp the color components to the range [0,1], and add an alpha value of 1.0 to produce our final fragment value.

Screen shots





Wildcat VP

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!

As mentioned previously, Brad Ritter of Hewlett-Packard provided the BRDF PTM shaders and provided a description of the process and answers to all my questions. Jerome Dewhurst graciously provided the environment maps that were used in the bumpy/shiny shader demo (jerome@photographica.co.uk, http://www.photographica.co.uk).

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