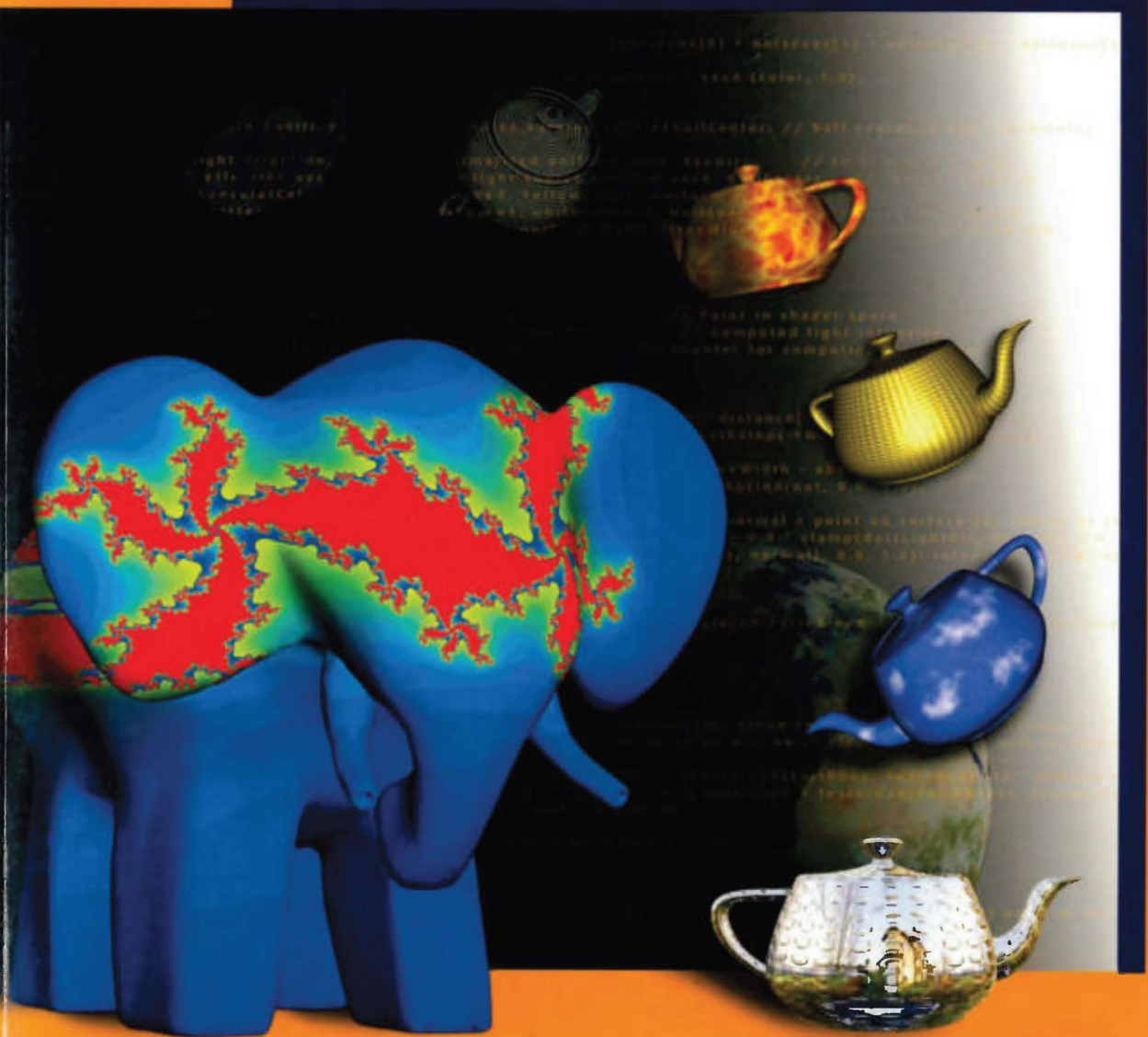


OpenGL[®]

Shading Language



Randi J. Rost

With contributions by John M. Kessenich and Barthold Lichtenbelt
Foreword by Marc Olano

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Hewlett-Packard Company makes no warranty as to the accuracy or completeness of the material included in this text and hereby disclaims any responsibility therefore.

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:

International Sales
(317) 581-3793
international@pearsontechgroup.com

Visit Addison-Wesley on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Rost, Randi J., 1960-

Open GL shading language / Randi Rost ; with contributions by John M. Kessenich and Barthold Lichtenbelt.

p. cm.

ISBN 0-321-19789-5 (acid-free paper)

1. Computer graphics. 2. OpenGL. I. Kessenich, John M. II. Lichtenbelt, Barthold. III. Title.

T385.R665 2004

006.6'86—dc22

2003022926

Copyright © 2004 by Pearson Education, Inc.

Chapter 3 © 2003 by John M. Kessenich

Parts of Chapter 4 © 2003 by Barthold Lichtenbelt

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116
Fax: (617) 848-7047

ISBN: 0-321-19789-5

Text printed on recycled paper

1 2 3 4 5 6 7 8 9 10—CRS—0807060504

First printing, February 2004

been augmented with programmable stages that can do everything the fixed functionality stages can do—and a whole lot more. The OpenGL Shading Language has been designed to allow application programmers to express the processing that occurs at those programmable points of the OpenGL pipeline.

The OpenGL Shading Language code that is intended for execution on one of the OpenGL programmable processors is called a `SHADER`. The term `OPENGL_SHADER` is sometimes used to differentiate a shader written in the OpenGL Shading Language from a shader written in another shading language such as RenderMan. Because two programmable processors are defined in OpenGL, there are two types of shaders: `VERTEX_SHADERS` and `FRAGMENT_SHADERS`. OpenGL provides mechanisms for compiling shaders and linking them together to form executable code called a program. A program contains one or more `EXECUTABLES` that can run on the programmable processing units.

The OpenGL Shading Language has its roots in C and has features similar to RenderMan and other shading languages. The language has a rich set of types, including vector and matrix types to make code more concise for typical 3D graphics operations. A special set of type qualifiers manages the unique forms of input and output needed by shaders. Some mechanisms from C++, such as function overloading based on argument types, and the capability to declare variables where they are first needed instead of at the beginning of blocks, have also been borrowed. The language includes support for loops, subroutine calls, and conditional expressions. An extensive set of built-in functions provides many of the capabilities needed for implementing shading algorithms. In brief

- The OpenGL Shading Language is a high-level procedural language.
- The same language, with a small set of differences, is used for both vertex and fragment shaders.
- It is based on C and C++ syntax and flow control.
- It natively supports vector and matrix operations, as these are inherent to many graphics algorithms.
- It is stricter with types than C and C++, and functions are called by value-return.
- It uses type qualifiers rather than reads and writes to manage input and output.
- There are no practical limits to a shader's length, nor is there a need to query it.

The following sections contain some of the key concepts that you will need to understand in order to use the OpenGL Shading Language effectively. The concepts will be covered in much more detail later in the book, but this introductory chapter should help you understand the big picture.

2.2 Why Write Shaders?

Until recently, OpenGL has presented application programmers with a flexible but static interface for putting graphics on the display device. As described in Chapter 1, you could think of OpenGL as a sequence of operations that occurred on geometry or image data as it was sent through the graphics hardware to be displayed on the screen. Various parameters of these pipeline stages could be altered in order to select variations on the processing that occurred for that pipeline stage. But neither the fundamental operation of the OpenGL graphics pipeline nor the order of operations could be changed through the OpenGL API.

By exposing support for traditional rendering mechanisms, OpenGL has evolved to serve the needs of a fairly broad set of applications. If your particular application was well served by the traditional rendering model presented by OpenGL, you may never have the need to write shaders. But if you have ever been frustrated by the fact that OpenGL does not allow you to define area lights or the fact that lighting calculations are performed per-vertex rather than per-fragment or if you have run into any of the many limitations of the traditional OpenGL rendering model, you may need to write your own OpenGL shader.

The purpose of the OpenGL Shading Language and its supporting OpenGL API entry points is to allow *applications* to define the processing that occurs at key points in the OpenGL processing pipeline using a high-level programming language specifically designed for this purpose. These key points in the pipeline are defined to be programmable in order to give applications complete freedom to define the processing that occurs. This gives applications the capability to utilize the underlying graphics hardware to achieve a much wider range of rendering effects.

To get an idea of the range of effects possible with OpenGL shaders, take a minute now and browse through the color images that are included in this book. This book presents a variety of shaders that only begin to scratch the surface of what is possible. With each new generation of graphics hardware, more complex rendering techniques can be implemented as OpenGL

shaders and can be used in real-time rendering applications. Here's a brief list of what's possible using OpenGL shaders:

- Increasingly realistic materials—metals, stone, wood, paints, and so on
- Increasingly realistic lighting effects—area lights, soft shadows, and so on
- Natural phenomena—fire, smoke, water, clouds, and so on
- Non-photorealistic materials—painterly effects, pen-and-ink drawings, simulation of illustration techniques, and so on
- New uses for texture memory—textures can be used to store normals, gloss values, polynomial coefficients, and so on
- Fewer texture accesses—textures can be created procedurally instead of accessing texture maps stored in texture memory
- Image processing—convolution, unsharp masking, complex blending, and so on
- Animation effects—key frame interpolation, particle systems, procedurally defined motion
- User programmable antialiasing methods

Many of these techniques have been available before now only through software implementations. If at all possible through OpenGL, they were possible only in a limited way. The fact that these techniques can now be implemented with hardware acceleration provided by dedicated graphics hardware means that rendering performance can be increased dramatically and at the same time the CPU can be off-loaded so that it can perform other tasks.

2.3 OpenGL Programmable Processors

The biggest change to OpenGL since its inception and the reason a high-level shading language is needed are the introduction of programmable vertex and fragment processors. In Chapter 1, we discussed the OpenGL pipeline and the fixed functionality that implements vertex processing and fragment processing. With the introduction of programmability, the fixed functionality vertex processing and fixed functionality fragment processing are disabled when an OpenGL Shading Language program is made current (i.e., made part of the current rendering state).

Figure 2.1 shows the OpenGL processing pipeline when the programmable processors are active. In this case, the fixed functionality vertex and the fragment processing shown in Figure 1.1 are replaced by programmable vertex and fragment processors as shown in Figure 2.1. All other parts of the OpenGL processing remain the same.

This diagram illustrates the stream processing nature of OpenGL via the programmable processors that are defined as part of the OpenGL Shading Language. Data flows from the application to the vertex processor, on to the fragment processor, and ultimately to the frame buffer.

2.3.1 Vertex Processor

The VERTEX PROCESSOR is a programmable unit that operates on incoming vertex values and their associated data. The vertex processor is intended to perform traditional graphics operations such as the following:

- Vertex transformation
- Normal transformation and normalization
- Texture coordinate generation
- Texture coordinate transformation

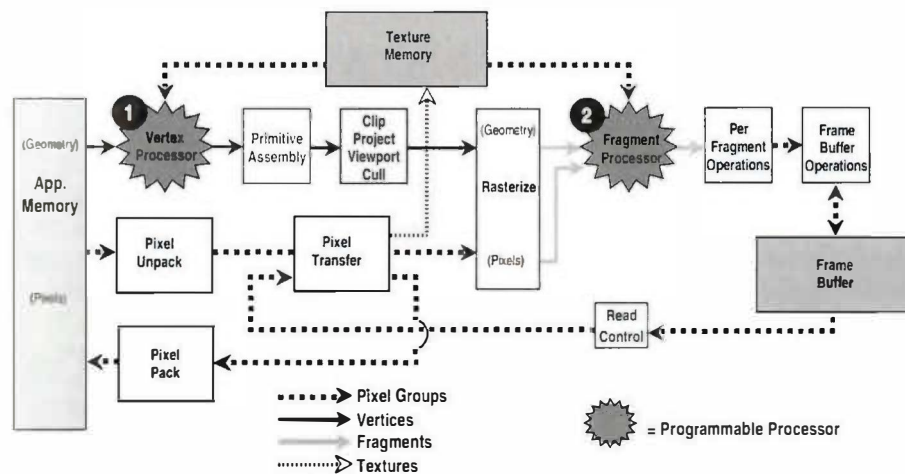


Figure 2.1 OpenGL logical diagram showing programmable processors for vertex and fragment shaders rather than fixed functionality

- Lighting
- Color material application

Because of its general purpose programmability, this processor can also be used to perform a variety of other computations. Shaders that are intended to run on this processor are called vertex shaders. Vertex shaders can be used to specify a completely general sequence of operations to be applied to each vertex and its associated data. Vertex shaders that perform some of the computations in the preceding list are responsible for writing the code for all desired functionality from the preceding list. For instance, it is not possible to use the existing fixed functionality to perform the vertex and normal transformation but to have a vertex shader perform a specialized lighting function. The vertex shader must be written to perform all three functions.

The vertex processor does not replace graphics operations that require knowledge of several vertices at a time or that require topological knowledge. OpenGL operations that remain as fixed functionality in between the vertex processor and the fragment processor include perspective divide and viewport mapping, primitive assembly, frustum and user clipping, backface culling, two-sided lighting selection, polygon mode, polygon offset, selection of flat or smooth shading, and depth range.

Figure 2.2 shows the data values that are used as inputs to the vertex processor and the data values that are produced by the vertex processor. Vertex shaders are used to express the algorithm that executes on the vertex processor to produce output values based on the provided input values. Type qualifiers that are defined as part of the OpenGL Shading Language are used to manage the input to the vertex processor and the output from it.

Variables defined in a vertex shader may be qualified as `ATTRIBUTE VARIABLES`. These represent values that are passed from the application to the vertex processor on a very frequent basis. Because this type of variable is used only for data from the application that defines vertices, it is permitted only as part of a vertex shader. Applications can provide attribute values between calls to `glBegin` and `glEnd` or via vertex array calls, so they can change as often as every vertex.

There are two types of attribute variables: built-in and user-defined. Standard attribute variables in OpenGL include things like color, surface normal, texture coordinates, and vertex position. The OpenGL calls `glColor`, `glNormal`, `glVertex`, and so on, and the OpenGL vertex array drawing commands can be used to send standard OpenGL vertex attributes to the vertex processor.

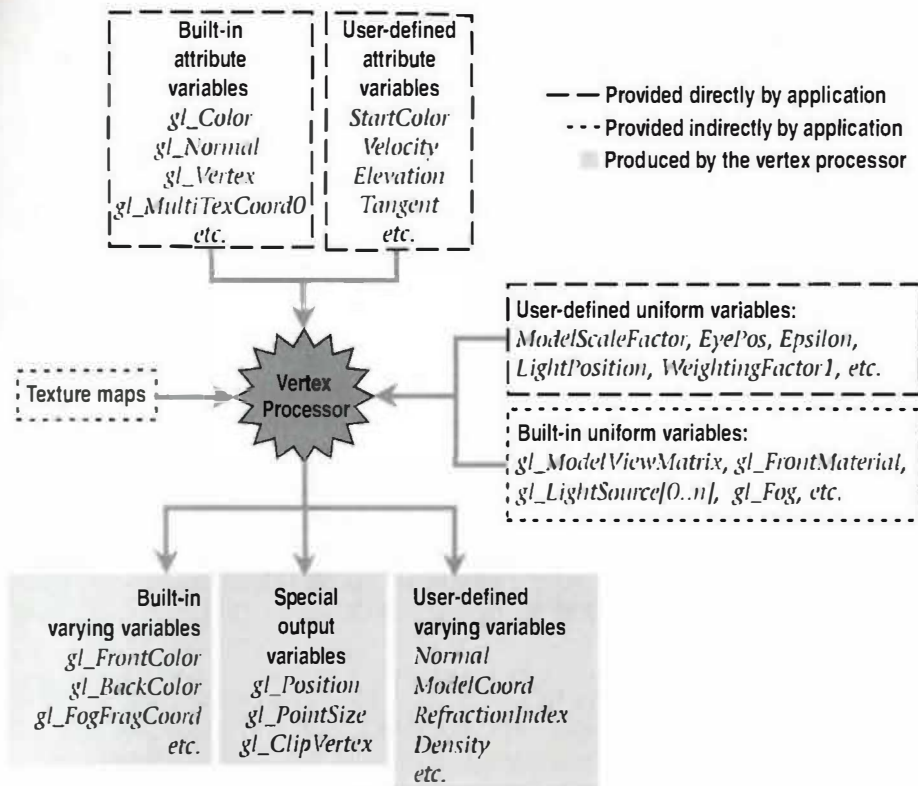


Figure 2.2 Vertex processor inputs and outputs

When a vertex shader is executing, it can access these data values through built-in attribute variables named *gl_Color*, *gl_Normal*, *gl_Vertex*, and so on.

Because this method restricts vertex attributes to the set that is already defined by OpenGL, a new interface has been added to allow applications to pass arbitrary per-vertex data. Within the OpenGL API, generic vertex attributes are defined and referenced by numbers from 0 up to some implementation-dependent maximum value. The command **glVertexAttribARB** sends generic vertex attributes to OpenGL by specifying the index of the generic attribute to be modified and the value for that generic attribute.

Vertex shaders are allowed to access these generic vertex attributes through user-defined attribute variables. Another new command, **glBindAttribLocationARB**, has been added to OpenGL to allow an application to tie together the index of a generic vertex attribute and the name with which to associate that attribute in a vertex shader.

UNIFORM VARIABLES are used for passing data values from the application to either the vertex processor or the fragment processor. Uniform variables are typically used to provide values that change relatively infrequently. A shader can be written so that it is parameterized using uniform variables. The application can provide initial values for these uniform variables, and the end user can be allowed to manipulate them through a graphical user interface to achieve a variety of effects with a single shader. But they cannot be specified between calls to `glBegin` and `glEnd`, so they can change at most once per primitive.

The OpenGL Shading Language supports both built-in and user-defined uniform variables. Vertex shaders and fragment shaders can access current OpenGL state through built-in uniform variables containing the reserved prefix “`gl_`”. User-defined uniform variables can be used by the application to make arbitrary data values available directly to a shader. `glGetUniformLocationARB` can be used to obtain the location of a user-defined uniform variable that has been defined as part of a shader. Data can be loaded into this location using another new OpenGL command, `glUniformARB`. There are a number of variations of this command in order to facilitate loading of floating-point, integer, Boolean, and matrix values, as well as arrays of these.

Another new feature is that vertex processors have also been given the capability to read from texture memory. This allows vertex shaders to implement displacement mapping algorithms, among other things. (However, the minimum number of vertex texture image units required by an implementation is 0, so texture map access from the vertex processor still may not be possible on all implementations that support the OpenGL Shading Language.) For accessing mipmap textures, level of detail can be specified directly in the shader. Existing OpenGL parameters for texture maps define the behavior of the filtering operation, borders, and wrapping.

Conceptually, the vertex processor operates on one vertex at a time (but an implementation may have multiple vertex processors that operate in parallel). The vertex shader is executed once for each vertex passed to OpenGL. The design of the vertex processor is focused on the functionality needed to transform and light a single vertex. Output from the vertex shader is accomplished partly by using special output variables. Vertex shaders must compute the homogeneous position of the coordinate in clip space and store the result in the special output variable `gl_Position`. Values to be used during user clipping and point rasterization can be stored in the special output variables `gl_ClipVertex` and `gl_PointSize`.

Variables that define data that is passed from the vertex processor to the fragment processor are called `VARYING VARIABLES`. Both built-in and user-defined varying variables are supported. They are called varying variables because the values will be potentially different at each vertex and perspective-correct interpolation will be performed to provide a value at each fragment for use by the fragment shader. Built-in varying variables include those defined for the standard OpenGL color and texture coordinate values. A vertex shader can use a user-defined varying variable to pass anything along that needs to be interpolated: colors, normals (useful for per-fragment lighting computations), texture coordinates, model coordinates, and other arbitrary values.

There is actually no harm (other than a possible loss of performance) in having a vertex shader calculate more varying variables than are needed by the fragment shader. A warning may be generated if the fragment shader consumes fewer varying variables than the vertex shader produces. But you may have good reasons to use a somewhat generic vertex shader with a variety of fragment shaders. The fragment shaders can be written to use a subset of the varying variables produced by the vertex shader. Applications that manage a large number of shaders may find that reducing the costs of shader development and maintenance is more important than squeezing out an additional percent of performance.

The vertex processor output (special output variables and user-defined and built-in varying variables) is sent on to subsequent stages of processing that are defined exactly the same as they are for OpenGL 1.5: primitive assembly, user clipping, frustum clipping, perspective divide, viewport mapping, polygon offset, polygon mode, shade mode, and culling.

2.3.2 Fragment Processor

The `FRAGMENT PROCESSOR` is a programmable unit that operates on fragment values and their associated data. The fragment processor is intended to perform traditional graphics operations such as the following:

- Operations on interpolated values
- Texture access
- Texture application
- Fog
- Color sum

A wide variety of other computations can be performed on this processor. Shaders that are intended to run on this processor are called fragment shaders. Fragment shaders are used to express the algorithm that executes on the fragment processor and produces output values based on the input values that are provided. A fragment shader cannot change a fragment's x/y position. Fragment shaders that perform some of the computations from the preceding list must perform all desired functionality from the preceding list. For instance, it is not possible to use the existing fixed functionality to compute fog but have a fragment shader perform specialized texture access and texture application. The fragment shader must be written to perform all three functions.

The fragment processor does not replace graphics operations that require knowledge of several fragments at a time. 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. An implementation may have multiple fragment processors that operate in parallel.

The fragment processor can be used to perform operations on each fragment that is generated by the rasterization of points, lines, polygons, pixel rectangles, and bitmaps. If images are first downloaded into texture memory, the fragment processor can also be used for pixel processing that requires access to a pixel and its neighbors. A rectangle can be drawn with texturing enabled, and the fragment processor can read the image from texture memory and apply it to the rectangle while performing traditional operations such as the following:

- Pixel zoom
- Scale and bias
- Color table lookup
- Convolution
- Color matrix

The fragment processor does not replace the fixed functionality graphics operations that occur at the back end of the OpenGL pixel processing pipeline such as coverage, pixel ownership test, scissor, stipple, alpha test, depth test, stencil test, alpha blending, logical operations, dithering, and plane masking.

Figure 2.3 shows the values that are used to provide input to the fragment processor and the data values that are produced by the fragment processor.

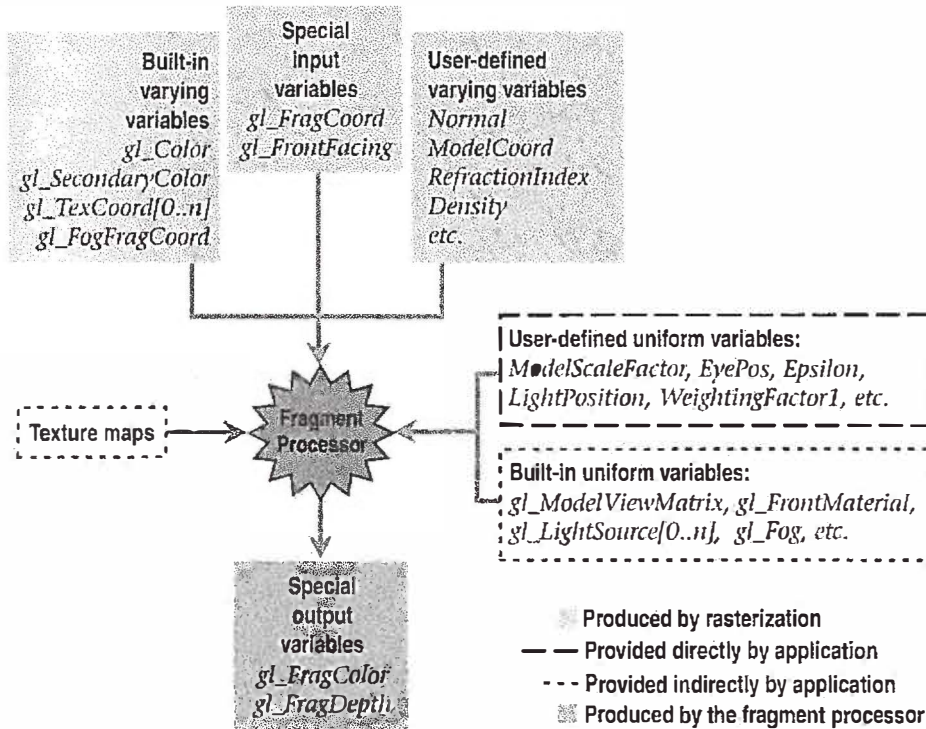


Figure 2.3 Fragment processor inputs and outputs

The primary inputs to the fragment processor are the interpolated varying variables (both built-in and user-defined) that are the results of rasterization. User-defined varying variables must be defined in a fragment shader, and their types must match those defined in the vertex shader.

Values computed by fixed functionality between the vertex processor and the fragment processor are made available through special input variables. The window coordinate position of the fragment and whether the fragment was generated by rasterizing a front-facing primitive are communicated through the special input variables *gl_FragCoord* and *gl_FrontFacing*.

Just as in the vertex shader, existing OpenGL state is accessible to a fragment shader through built-in uniform variables. All of the OpenGL state that is available through built-in uniform variables is available to both vertex and fragment shaders. This makes it possible to implement traditional vertex

operations such as lighting in the fragment shader using standard OpenGL state management.

User-defined uniform variables are used to allow the application to pass relatively infrequently changing values to a fragment shader. The same uniform variable can be accessed by both a vertex shader and a fragment shader if it is declared by both shaders using the same data type.

One of the biggest advantages of the fragment processor is that it can access texture memory an arbitrary number of times and combine the values that it reads in arbitrary ways. A fragment shader is free to read multiple values from a single texture or multiple values from multiple textures. The result of one texture access can be used as the basis for performing another texture access (a `DEPENDENT TEXTURE READ`). There is no inherent limitation on the number of such dependent reads that are possible, so ray-casting algorithms can be implemented in a fragment shader.

The OpenGL parameters for texture maps continue to define the behavior of the filtering operation, borders, wrapping, and texture comparison modes. These operations are applied when a texture is accessed from within a shader. The shader is free to use the resulting value however it chooses. It is possible for a shader to read multiple values from a texture and perform a custom filtering operation. It is also possible to use a texture to perform a lookup table operation.

The fragment processor defines almost all of the capabilities necessary to implement the pixel transfer operations defined in OpenGL 1.5, including those in the imaging subset. This means that advanced pixel processing is supported with the fragment processor. Lookup table operations can be done with 1D texture accesses, allowing applications to have full control over their size and format. Scale and bias operations are easily expressed through the programming language. The color matrix can be accessed through a built-in uniform variable. Convolution and pixel zoom are supported by accessing a texture multiple times to compute the proper result. Histogram and minimum/maximum operations are left to be defined as extensions because these prove to be quite difficult to support at the fragment level with high degrees of parallelism.

For each fragment, the fragment shader may compute color and depth (writing these values into the special output variables `gl_FragColor` and `gl_FragDepth`) or completely discard the fragment. The results of the fragment shader are then sent on for further processing. The remainder of the OpenGL pipeline remains as defined in OpenGL 1.5. 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. The back end of the processing pipeline remains as fixed functionality because it is easy to implement in nonprogrammable hardware. Making these functions programmable is more complex because read/modify/write operations can introduce significant instruction scheduling issues and pipeline stalls. Most of these fixed functionality operations can be disabled, and alternate operations can be performed within a fragment shader if desired (albeit with possibly lower performance).

2.4 Language Overview

Because of its success as a standard, OpenGL has been the target of our efforts to define an industry standard high-level shading language. The shading language that has been defined as a result of the efforts of OpenGL ARB members is called the OpenGL Shading Language. This language has been designed to be forward looking and eventually to support programmability in other areas as well.

This section provides a brief overview of the OpenGL Shading Language. For a complete discussion of the language, see Chapter 3, Chapter 4, and Chapter 5.

2.4.1 Language Design Considerations

In the past few years, semiconductor technology has progressed to the point where the levels of computation that can be done per vertex or per fragment have gone beyond what is feasible to describe by the traditional OpenGL mechanisms of setting state to influence the action of fixed pipeline stages. A natural way of taming this complexity and the proliferation of OpenGL extensions is to allow parts of the pipeline to be replaced by user programmable stages. This has been done in some recent OpenGL extensions but the programming is done in assembly language. By definition, assembly languages are hardware-specific, so going down this path would lead software developers to create code that is hardware- or vendor-specific and that might not even run on future generations of graphics hardware.

The ideal solution to these issues was to define a forward looking hardware-independent high-level language that would be easy to use, powerful enough to stand the test of time, and drastically reduce the need for

2.5 System Overview

We have already described briefly some of the pieces that provide applications with access to the programmability of underlying graphics hardware. This section gives an overview of how these pieces go together in a working system.

2.5.1 Driver Model

A piece of software that controls a piece of hardware and manages shared access to that piece of hardware is commonly called a `DRIVER`. No matter what the environment in which it is implemented, OpenGL will fall into this category because OpenGL manages shared access to the underlying graphics hardware. Some of its tasks must also be coordinated with, or supervised by, facilities in the operating system.

Figure 2.4 illustrates how OpenGL shaders are handled in the execution environment of OpenGL. Applications communicate with OpenGL by calling functions that are part of the OpenGL API. A new OpenGL function, `glCreateShaderObjectARB`, has been created to allow applications to allocate within the OpenGL driver, the data structures that are necessary for storing an OpenGL shader. These data structures are called `SHADER OBJECTS`. After a shader object has been created, the application can provide the source code for the shader by calling `glShaderSourceARB`. This command is used to provide to OpenGL the character strings containing the shader source code.

As you can see from Figure 2.4, the compiler for the OpenGL Shading Language is actually part of the OpenGL driver environment. This is one of the key differences between the OpenGL Shading Language and other shading language designs, such as the Stanford Shading Language, High-Level Shader Language (HLSL) from Microsoft, or Cg from NVIDIA. In these other languages, the high-level shading language compiler sits above the graphics API and translates the high-level shading language into something that can be consumed by the underlying graphics API. (See Chapter 17 for more details.) With the OpenGL Shading Language, the source code for shaders is passed to the OpenGL driver, and in that environment, the shaders will be compiled to the native machine code as efficiently as possible.

After source code for a shader has been loaded into a shader object in the OpenGL driver environment, it can be compiled by calling `glCompileShaderARB`. A `PROGRAM OBJECT` is an OpenGL-managed data structure that acts as a container for shader objects. Applications are required to attach shader objects to a program object by using the command `glAttachObjectARB`.

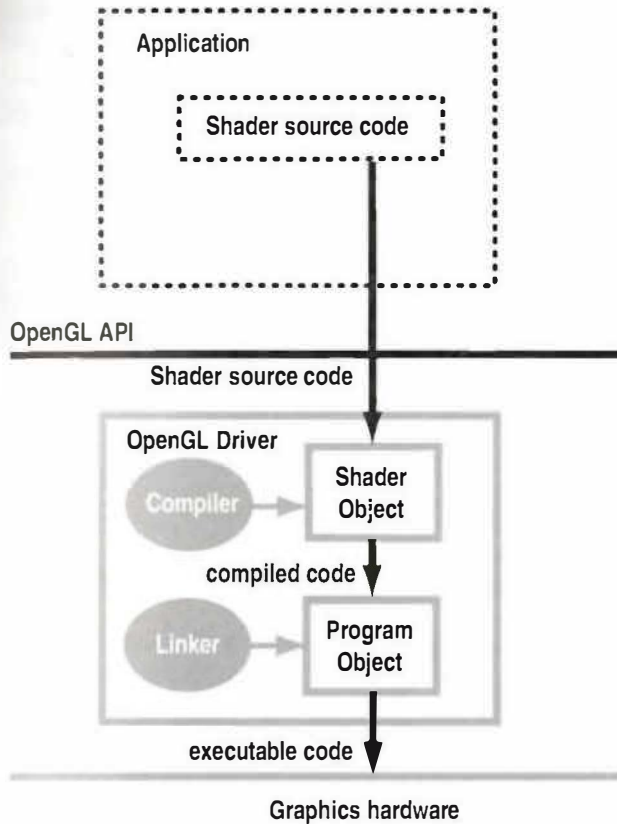


Figure 2.4 Execution model for OpenGL shaders

When attached to a program object, the compiled shader objects can be linked together by calling `glLinkProgramARB`. Support for multiple shader objects (and the subsequent need for a linker built into OpenGL) is a key difference between the OpenGL Shading Language and assembly-level APIs such as those provided by the OpenGL extensions *ARB_vertex_program* and *ARB_fragment_program*. For more complex shading tasks, separately compiled shader objects are a much more attractive alternative than a single, monolithic block of assembly-level code.

The link step will resolve external references between the shaders, check the compatibility between the vertex shader and the fragment shader, assign memory locations to uniform variables, and so on. The result is one or more executables that can be installed as part of OpenGL's current state by calling `glUseProgramObjectARB`. This command installs the executables on the vertex processor and/or the fragment processor where they will be used to render all subsequent graphics primitives.

2.5.2 OpenGL Shading Language Compiler/Linker

The source for a single shader is an array of strings of characters, and a single shader is made from the concatenation of these strings. Each string can contain multiple lines, separated by new-lines. No new-lines need be present in a string; a single line can be formed from multiple strings. No new-lines or other characters are inserted by the OpenGL implementation when it concatenates the strings to form a single shader. It is entirely up to the application programmer to provide shader source code to OpenGL with new-lines between each line of source code.

Diagnostic messages returned from compiling a shader must identify both the line number within a string and the source string to which the diagnostic message applies. Source strings are counted sequentially with the first string counted as string 0. When parsing source code, the current line number is one more than the number of new-lines that have been processed.

The front end of the OpenGL Shading Language compiler has been released as open source by 3Dlabs and can be used by anyone interested in writing his or her own compiler. This publicly available front end is capable of doing lexical analysis of OpenGL Shading Language source code to produce a token stream and then performing syntactic and semantic analysis of this token stream to produce a binary, high-level representation of the language. This front end is intended to act as a reference implementation of the OpenGL Shading Language, and therefore it goes hand-in-hand with the language specification in order to define the language clearly. Another advantage to using this publicly available front end in an OpenGL Shading Language compiler implementation is that the syntax and semantics for shaders will be checked consistently by all implementations that use this front end. More consistency among compiler implementations makes it easier for developers to write shaders that work as intended across a variety of implementations.

It is assumed that the back end of the OpenGL Shading Language compiler will be implemented differently on different platforms. Each implementation will need to take the high-level representation produced by the publicly available front end and produce optimized machine code for a particular hardware target. This is an area where individual hardware vendors will add value to their shading language implementation by figuring out ways to map the high-level representation onto the actual machine instructions found in their hardware. Likewise, the linking stage will also be highly hardware dependent because it involves operations like assigning variables to actual memory locations in the hardware.

The net result of this is that graphics hardware vendors will be implementing the majority of the OpenGL Shading Language compiler and linker. Along with the OpenGL driver itself, this software will typically be included as part of the graphics driver installation package that is provided by a graphics hardware vendor.

2.5.3 OpenGL API Extensions

Currently, support for the OpenGL Shading Language is not available as part of standard OpenGL, but it is available as a set of extensions that are supported by a number of graphics hardware vendors. It is expected that these extensions will fairly quickly be adopted as part of the OpenGL standard. When this happens, there is expected to be no change, except for dropping the ARB suffix on the function names, constants, and data types that are defined by these extensions. For now, the way to get to this functionality is through these commonly supported extensions.

The extension that provides the framework for the shader and program objects and the functionality that is common to all the programmable processors is called *ARB_shader_objects*. This extension provides the following basic capabilities:

- **glCreateShaderObjectARB**—Create a shader object
- **glCreateProgramObjectARB**—Create a program object
- **glDeleteObjectARB**—Delete a shader object or a program object
- **glShaderSourceARB**—Load source code strings into a shader object
- **glCompileShaderARB**—Compile a shader
- **glAttachObjectARB**—Attach a shader object to a program object
- **glDetachObjectARB**—Detach a shader object from a program object

- **glLinkProgramARB**—Link a program object to create executable code
- **glUseProgramObjectARB**—Install a program object's executable code as part of current state
- **glValidateProgramARB**—Return validation information for a program object
- **glUniformARB**—Set the value of a uniform variable
- **glGetActiveUniformARB**—Obtain the name, size, and type of an active uniform variable for a program object
- **glGetAttachedObjectsARB**—Get the list of shader objects attached to a program object
- **glGetHandleARB**—Obtain the handle for the program object currently in use
- **glGetObjectParameterARB**—Query one of the parameters of an object
- **glGetShaderSourceARB**—Get the source code for a specific shader object
- **glGetUniformARB**—Query the current value of a uniform variable
- **glGetUniformLocationARB**—Query the location assigned to a uniform variable by the linker
- **glGetInfoLogARB**—Obtain the information log for a shader object or a program object

The extension *ARB_vertex_shader* addresses the capabilities of the newly defined programmable vertex processor. This extension defines how the programmable vertex processor fits into the OpenGL processing pipeline, and it provides API entry points for features that are unique to the vertex processor. Functionality defined by this extension includes

- How vertex shaders are created
- How vertex shaders are enabled/disabled
- Which OpenGL fixed functionality is disabled when a vertex shader is active
- How values passed to standard OpenGL vertex entry points are passed into a vertex shader
- How generic vertex attributes are handled
- How a vertex shader interfaces with OpenGL fixed functionality that follows vertex processing, including primitive assembly, clipping, and rasterization

New entry points defined by the *ARB_vertex_shader* extension are

- **glVertexAttribARB**—Send generic vertex attributes to OpenGL a vertex at a time
- **glVertexAttribPointerARB**—Specify location and organization of generic vertex attributes to be sent to OpenGL using vertex arrays
- **glBindAttribLocationARB**—Specify the generic vertex attribute index to be used for a particular user-defined attribute variable in a vertex shader
- **glEnableVertexAttribArrayARB**—Enable a generic vertex attribute to be sent to OpenGL using vertex arrays
- **glDisableVertexAttribArrayARB**—Disable a generic vertex attribute from being sent to OpenGL using vertex arrays
- **glGetVertexAttribARB**—Returns current state for the specified generic vertex attribute
- **glGetVertexAttribLocationARB**—Returns the generic vertex attribute index that is bound to a specified user-defined attribute variable
- **glGetVertexAttribPointerARB**—Returns the vertex array pointer value for the specified generic vertex attribute
- **glGetActiveAttribARB**—Obtains the name, size, and type of an active attribute for a program object

The third and final extension that provides support for the OpenGL Shading Language is called *ARB_fragment_shader*. This extension is similar to the *ARB_vertex_shader* extension, except that it defines the capabilities of the newly defined programmable fragment processor and how the programmable fragment processor fits into the OpenGL processing pipeline. There are actually no new API entry points defined by this extension because it builds on the generic capabilities defined in the *ARB_shader_objects* extension. It does provide new functionality, however, including

- How fragment shaders are created
- How fragment shaders are enabled/disabled
- Which OpenGL fixed functionality is disabled when a fragment shader is active
- How values produced by OpenGL rasterization are passed into a fragment shader
- How values produced by a fragment shader are provided to OpenGL's fixed functionality back-end processing

These new entry points are all discussed in more detail in Chapter 7. Reference pages for all of the API entry points defined by these extensions are included in Appendix B at the back of this book.

2.6 Key Benefits

The following key benefits are derived from the choices that were made during the design of the OpenGL Shading Language.

Tight integration with OpenGL—The OpenGL Shading Language was designed for use in OpenGL. It is designed in such a way that an existing, working OpenGL application can be easily modified to take advantage of the capabilities of programmable graphics hardware. Built-in access to existing OpenGL state, reuse of API entry points that are already familiar to application developers, and a close coupling with the existing architecture of OpenGL are all key benefits to using the OpenGL Shading Language for shader development.

Compilation occurs at runtime—Source code stays as source code, in its easiest-to-maintain form, for as long as possible. An application passes source code to any conforming OpenGL implementation that supports the OpenGL Shading Language, and it will be compiled and executed properly. There is no need for a multitude of binaries for a multitude of different platforms.¹

No reliance on cross-vendor assembly language—Both DirectX and OpenGL have widespread support for assembly language interfaces to graphics programmability. High-level shading languages could be (and have been) built on top of these assembly language interfaces, and such high-level languages can be translated into these assembly language interfaces completely outside the environment of OpenGL or DirectX. This does have some advantages, but relying on an assembly language interface as the primary interface to hardware programmability restricts innovation by graphics hardware designers. Hardware designers have many more choices

¹ At the time of this writing, the OpenGL ARB is still considering the need for an API that allows shaders to be specified in a form other than source code. The primary issues are the protection of intellectual property that may be embedded in string-based shader source code and the performance that would be gained by allowing shaders to be at least partially precompiled. When such an API is defined, shader portability may be reduced, but application developers will have the option of getting better code security and/or better performance.

for acceleration of an expressive high-level language than they do for a restrictive assembly language. It is much too early in the development of programmable graphics hardware technology to establish an assembly language standard for graphics programmability. C, on the other hand, was developed long before any CPU assembly languages that are in existence today, and it is still a viable choice for application development.

Unconstrained opportunities for compiler optimization will lead to optimal performance on a wider range of hardware—As we've learned through experience with CPUs, compilers are much better at quickly generating efficient code than humans are. By allowing high-level source code to be compiled within OpenGL, rather than outside of OpenGL, individual hardware vendors have the best possible opportunity to deliver optimal performance on their graphics hardware. In fact, compiler improvements can be made with each and every OpenGL driver release, and the applications won't need to change any application source code, recompile the application, or even relink it. Furthermore, the current crop of assembly language interfaces is string based, so using these as an interface requires that string-based high-level source be translated into string-based assembly language, and then that string-based assembly language must be passed to OpenGL and translated from string-based assembly language to machine code.

A truly open, cross-platform standard—No other high-level graphics shading language has been approved as part of an open, multivendor standard. Like OpenGL itself, the OpenGL Shading Language will be implemented by a variety of different vendors for a variety of different environments.

One high-level language for all programmable graphics processing—The OpenGL Shading Language is used to write shaders for both the vertex processor and the fragment processor in OpenGL, with very small differences in the language for the two types of shaders. In the future, it is intended that the OpenGL Shading Language will be used to bring programmability to other areas of OpenGL as well. Areas that have already received some discussion include programmability for packing/unpacking arbitrary image formats and support for programmable tessellation of higher order surfaces in the graphics hardware.

Support for modular programming—By defining compilation and linking as two separate steps, shader writers have a lot more flexibility in how they choose to implement complex shading algorithms. Rather than implement a complex algorithm as a single, monolithic shader, developers are free to implement it as a collection of shaders that can be independently compiled and attached to a program object. Shaders can be designed with

common interfaces so that they are interchangeable, and a link operation is used to join them to create a program.

No additional libraries or executables—The OpenGL Shading Language and the compiler and linker that support it are defined as part of OpenGL. Applications need not worry about linking against any additional runtime libraries. Compiler improvements are delivered as part of OpenGL driver updates.

2.7 Summary

Here are the key points to understand about how all the pieces fit together at execution time.

- When installed as part of current state, the executable created for the vertex processor will be executed once for every vertex provided to OpenGL.
- When installed as part of current state, the executable created for the fragment processor will be executed once for every fragment that is produced by rasterization.
- There are two ways for an application to communicate directly with a vertex shader: by using attribute variables and by using uniform variables.
- Attribute variables are expected to change very frequently and may be supplied by the application as often as every vertex.
- Applications can pass arbitrary vertex data to a vertex shader using user-defined attribute variables.
- Applications can pass standard vertex attributes (color, normal, texture coordinates, position, etc.) to a vertex shader using built-in attribute variables.
- An application communicates directly with a fragment shader using uniform variables.
- Uniform variables are expected to change relatively infrequently (at a minimum, they are constant for an entire graphics primitive).
- The compiler and linker for the language are contained within OpenGL (but tools for compiling, linking, and debugging shaders can exist outside of OpenGL as well).

To summarize, the following are the most important points about the language.

- The language is based on the syntax of C.
- Basic structure and many keywords are the same as in C.
- Vectors and matrices are included in the language as basic types.
- Type qualifiers **attribute**, **uniform**, and **varying** are added to describe variables that manage shader I/O.
 - Variables of type **attribute** allow the communication of frequently changing values from the application to the vertex shader.
 - Variables of type **varying** are the output from a vertex shader and the input to a fragment shader.
 - Variables of type **uniform** allow the application to provide relatively infrequently changing values to both vertex shaders and fragment shaders.
- The data type **sampler** is added for accessing textures.
- Built-in variable names can be used to access standard OpenGL state and to communicate with OpenGL fixed functionality.
- A variety of built-in functions are included for performing common graphics operations.
- Function declarations are required, and overloading based on number and type of arguments is supported as in C++.
- Variables can be declared when needed.

To install and use OpenGL shaders, you need to do the following:

1. Create one or more (empty) shader objects using **glCreateShaderObjectARB**.
2. Provide source code for these shaders by calling **glShaderSourceARB**.
3. Compile each of the shaders by calling **glCompileShaderARB**.
4. Create a program object by calling **glCreateProgramObjectARB**.
5. Attach all the shader objects to the program object by calling **glAttachObjectARB**.

6. Link the program object by calling `glLinkProgramARB`.
7. Install the executable program as part of OpenGL's current state by calling `glUseProgramObjectARB`.

After these steps, subsequent graphics primitives will be drawn using the shaders you've provided rather than with OpenGL's defined fixed functionality pipeline.

2.8 Further Information

Just keep reading this book, and you'll get to all of the really good stuff! If you really must go and get more technical details, here are pointers to the official specification documents. The 3DLabs Web site also has additional material, including slide presentations, demos, example shaders, and source code.

- [1] *3DLabs developer Web site*. <http://www.3dlabs.com/support/developer>
- [2] Kessenich, John, Dave Baldwin, and Randi Rost, *The OpenGL Shading Language, Version 1.051*, 3DLabs, February 2003.
<http://www.3dlabs.com/support/developer/ogl2>
- [3] OpenGL Architecture Review Board, *ARB_vertex_shader Extension Specification*, OpenGL Extension Registry.
<http://oss.sgi.com/projects/ogl-sample/registry>
- [4] OpenGL Architecture Review Board, *ARB_fragment_shader Extension Specification*, OpenGL Extension Registry.
<http://oss.sgi.com/projects/ogl-sample/registry>
- [5] OpenGL Architecture Review Board, *ARB_shader_objects Extension Specification*, OpenGL Extension Registry.
<http://oss.sgi.com/projects/ogl-sample/registry>

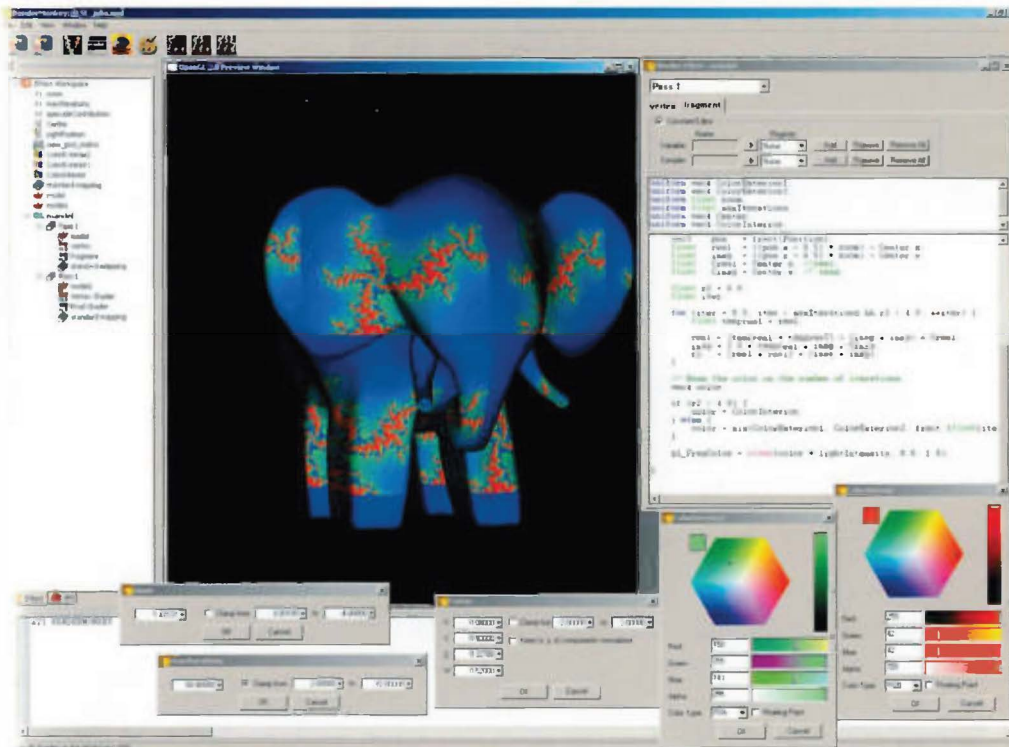
In Color Plate 15, the threshold values were both set to 0.13. This means that more than three-quarters of the fragments were being discarded! And that's what I call a "holy cow!"

11.4 Bump Mapping

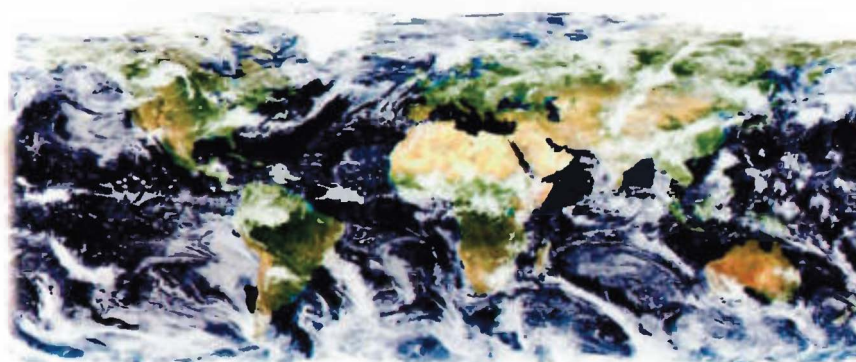
We have already seen procedural shaders that modified color (*brick, stripes*) and opacity (*lattice*). Another whole class of interesting effects can be applied to a surface using a technique called BUMP MAPPING. Bump mapping involves modulating the surface normal before lighting is applied. The modulation 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. Bump mapping has proven to be an effective way of increasing the apparent realism of an object without increasing the geometric complexity. It can be used to simulate surface detail or surface irregularities.

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 a planet as a sphere and shading it with a bump map so that it appears to have mountains that are quite large relative to the diameter of the planet. Because nothing has been done to change the underlying geometry, which is perfectly round, the silhouette of the sphere will always appear perfectly round, even if the mountains (bumps) go right up to the silhouette edge. In real life, you would expect the mountains 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.

Bump mapping adds apparent geometric complexity during fragment processing, so once again the key to the process will be our fragment shader. This implies that the lighting operation will need to be performed by our fragment shader, instead of by the vertex shader where it is often handled. Again, this points out one of the advantages of the programmability that is available through the OpenGL Shading Language. We are free to perform whatever operations are necessary, in either the vertex shader or the fragment shader. We don't need to be bound to the fixed functionality ideas of where things like lighting are performed.



Color Plate 1 Screen shot of the RenderMonkey IDE user interface. The shader that is procedurally generating a Julia set on the surface of the elephant is shown in the source code window. Color selection tools and user interface elements for manipulating user-defined uniform variables are also shown.



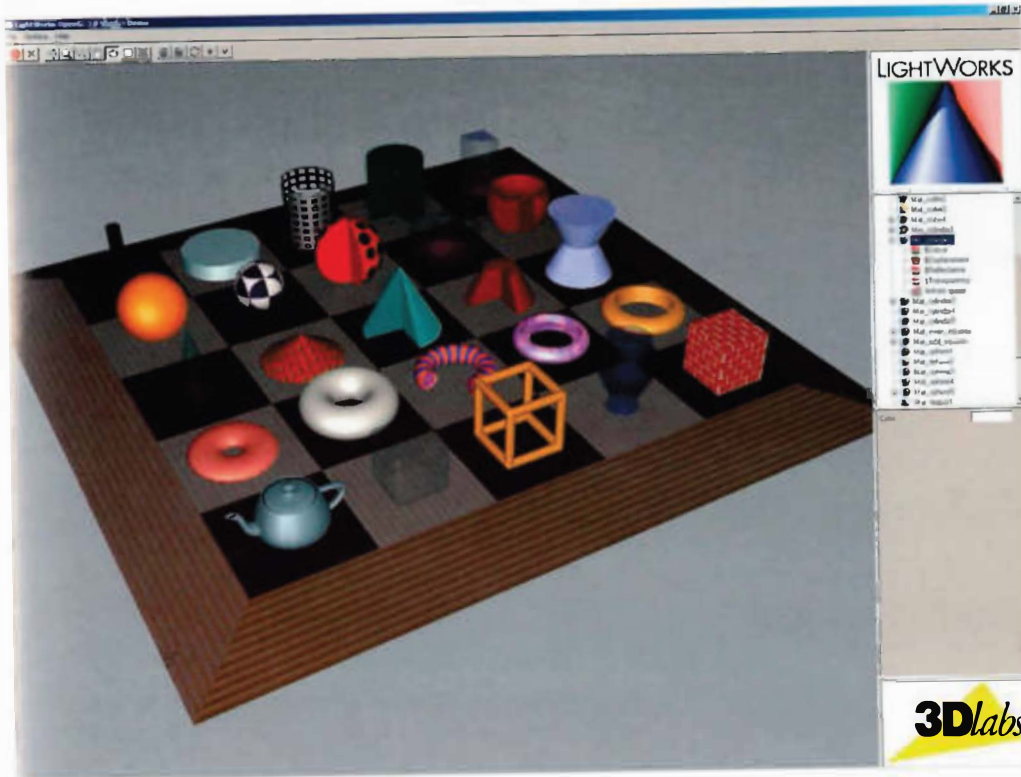
Color Plate 2 A full-color image of Earth that is used as a texture map for the shader discussed in Section 10.2. (Blue Marble image by Reto Stockli, NASA Goddard Space Flight Center)



Color Plate 10 Image showing the difference between a conventional texture map (lower left) and a polynomial texture map (PTM) (upper right). The conventional texture looks flat and unrealistic as the light source is moved, while the PTM faithfully reproduces changing specular highlights and self-shadowing. (© Copyright 2003 Hewlett-Packard Development Company, L.P., Reproduced with Permission)



Color Plate 11 A torus rendered using the BRDF PTM shaders described in Section 10.5. Although the paint is basically black, note the change in the highlight color (from bluish-purple in the back to reddish-brown in the front) as the reflection angle changes. (© Copyright 2003 Hewlett-Packard Development Company, L.P., Reproduced with Permission)



Color Plate 12 A variety of objects rendered with OpenGL procedural shaders in a demo application by LightWork Design. This application shows a graphics user interface designed for interacting with procedural shaders. (Courtesy of LightWorks Design)