

THE OPENGL[®] SHADING LANGUAGE

John Kessenich

Dave Baldwin

Randi Rost

Language Version 1.10

Document Revision 59

30-April-2004

Copyright © 2002-2004 3Dlabs, Inc. Ltd.

This document contains unpublished information of 3Dlabs, Inc. Ltd.

This document is protected by copyright, and contains information proprietary to 3Dlabs, Inc. Ltd. Any copying, adaptation, distribution, public performance, or public display of this document without the express written consent of 3Dlabs, Inc. Ltd. is strictly prohibited. The receipt or possession of this document does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

This document contains intellectual property of 3Dlabs Inc. Ltd., but does not grant any license to any intellectual property from 3Dlabs or any third party. It is 3Dlabs' intent that should an OpenGL 2.0 API specification be ratified by the ARB incorporating all or part of the contents of this document, then 3Dlabs would grant a royalty free license to Silicon Graphics, Inc., according to the ARB bylaws, for only that 3Dlabs intellectual property as is required to produce a conformant implementation.

This specification is provided "AS IS" WITH NO WARRANTIES WHATSOEVER, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE, 3DLABS EXPRESSLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION, OR SAMPLE.

U.S. Government Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions set forth in FAR 52.227.19(c)(2) or subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and/or in similar or successor clauses in the FAR or the DOD or NASA FAR Supplement. Unpublished rights reserved under the copyright laws of the United States. Contractor/manufacturer is 3Dlabs, Inc. Ltd., 9668 Madison Blvd., Madison, Alabama 35758.

OpenGL is a registered trademark of Silicon Graphics Inc.

1 INTRODUCTION

Note: Document revisions for the language specified by this document are being tracked separately from the language version. Changing the revision of the document does not change the version of the language. This document specifies version 1.10 of the OpenGL Shading Language, document revision number 59. It requires `__VERSION__` to be 110, and `#version` to accept 110.

1.1 Changes since version 1.051

- Added issues 101 through 105. Specification changes made from these issues are to make array parameters sized, and add some limitations in constructors. See sections 4.2, 5.4.2, 6.1, 6.1.1.
- Added interactions with `ATI_draw_buffers` and `ARB_color_clamp_control`, particularly, the output variable `gl_FragData[n]`.
- 3.3 Added `#version` and `#extension` to declare version and extensions.
- 7.5 Added built-in state for the inverses and transposes of matrices.
- 8 Added built-in functions **refract**, **exp**, and **log**.
- Added the following clarifications and corrections:
 - 2.1 Remove "Clamping of colors" from the list of what the vertex processor does. This was just out of date.
 - 2.1 Change "Perspective projection" to more clearly call out projective transform and perspective division, which belong in different lists.
 - 3.3 Reserved pre-processor macros that start "GL_".
 - 3.6 Added reserved words **packed**, **this**, **interface**, **sampler2DRectShadow**. Also clarified that the listed keywords and reserved words are the only ones.
 - 4.1.5 Remove "Integer vectors can be used to get multiple integers back from a texture read." This was just out of date.
 - 4.3.5 Clarified that structs can be constants, and what **const** must be initialized with.
 - 5.8 Clarify what `*=`, `+=`, etc. really mean. and that `?:` is not an l-value.
 - 5.9 Clarify that operating between a scalar and a vector is allowed for integers as for floats, and that the list is to list all operators and expressions.
 - 6.1 Correct the examples of dot product prototypes. They were not correct WRT to the list of prototypes, which themselves have been correct for some time.
 - 6.1 Add the clarification "If a built-in function is redeclared in a shader (i.e. a prototype is visible) before a call to it, then the linker will only attempt to resolve that call within the set shaders that are linked with it."
 - 7.2 Remove the out of date text "an implementation will provide invariant results within shaders computing depth with the same source-level expression, but invariance is not provided between shaders and fixed functionality."

Chapter 1 and 2 from "The OpenGL Shading Language" by John Kessenich, Dave Baldwin and Randi Rost
<http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf>

- 7.4 Correct the list of built-in constant names: removed suffixes and brought values up to date.
- 8.2 State the domains for the exponential functions.
- 8.3 Change `step()` to compare $x < edge$ instead of $x \leq edge$.
- 8.7 Clarify the discussion about when shadowing lookups are undefined.
- 8.9 Further specify the range and frequency constraints of noise.
- Grammar: `MOD_ASSIGN` change to reserved (to match the specification text).
- Grammar: Change to require array sizes in function parameters.
- Several typos fixed.

1.2 Overview

This document describes a programming language called *The OpenGL Shading Language*, or *glslang*. The recent trend in graphics hardware has been to replace fixed functionality with programmability in areas that have grown exceedingly complex (e.g., vertex processing and fragment processing). 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.

Independently compilable units that are written in this language are called *shaders*. A *program* is a set of shaders that are compiled and linked together. The aim of this document is to thoroughly specify the programming language. The OpenGL entry points that are used to manipulate and communicate with programs and shaders are defined separately from this language specification.

The OpenGL Shading Language is based on ANSI C and many of the features have been retained except when they conflict with performance or ease of implementation. C has been extended with vector and matrix types (with hardware based qualifiers) to make it more concise for the typical operations carried out in 3D graphics. Some mechanisms from C++ have also been borrowed, such as overloading functions based on argument types, and ability to declare variables where they are first needed instead of at the beginning of blocks.

1.3 Motivation

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 desire to expose the extended capability of the hardware has resulted in a vast number of extensions being written and an unfortunate consequence of this is to reduce, or even eliminate, the portability of applications, thereby undermining one of the key motivating factors for OpenGL.

A natural way of taming this complexity and the proliferation of extensions is to allow parts of the pipeline to be replaced by user programmable stages. This has been done in some recent extensions but the programming is done in assembler, which is a direct expression of today's hardware and not forward looking. Mainstream programmers have progressed from assembler to high-level languages to gain productivity, portability and ease of use. These goals are equally applicable to programming shaders.

The goal of this work is a forward looking hardware independent high-level language that is easy to use and powerful enough to stand the test of time and drastically reduce the need for extensions. These desires must be tempered by the need for fast implementations within a generation or two of hardware.

1.4 Design Considerations

The various programmable processors we are going to introduce replace parts of the OpenGL pipeline and as a starting point they need to be able to do everything they are replacing. This is just the beginning and the examples from the RenderMan community and newer games provide some hints at the exciting possibilities ahead.

To facilitate this, the shading language should be at a high enough level and with the abstractions for the problem domain we are addressing. For graphics this means vector and matrix operations form a fundamental part of the language. This extends from being able to specify scalar/vector/matrix operations directly in expressions to efficient ways to manipulate and group the components of vectors and matrices. The language includes a rich set of built-in functions that operate just as easily on vectors as on scalars.

We are fortunate in having the C language as a base to build on and RenderMan as an existing shading language to learn from. OpenGL is associated with "real-time" graphics (as opposed to off-line graphics) so any aspects of C and RenderMan that hinder efficient compilation or hardware implementation have been dropped, but, for the most part, these are not expected to be noticeable.

The OpenGL Shading Language is designed specifically for use within the OpenGL environment. It is intended to provide programmable alternatives to certain parts of the fixed functionality of OpenGL. By design, it is possible, and quite easy to refer to existing OpenGL state for these parts from within a shader. By design, it is also possible, and quite easy to use fixed functionality in one part of the OpenGL processing pipeline and programmable processing in another. It is the intent that the object code generated for a shader be independent of other OpenGL state, so that recompiles or managing multiple copies of object code are not necessary.

Graphics hardware is developing more and more parallelism at both the vertex and the fragment processing levels. Great care has been taken in the definition of the OpenGL Shading Language to allow for even higher levels of parallel processing.

Finally, it is a goal to use the same high-level programming language for all of the programmable portions of the OpenGL pipeline. Certain types and built-in functions are not permitted on certain programmable processors, but the majority of the language is the same across all programmable processors. This makes it much easier for application developers to embrace the shading language and use it to solve their OpenGL rendering problems.

1.5 Error Handling

Compilers, in general, accept programs that are ill-formed, due to the impossibility of detecting all ill-formed programs. For example, completely accurate detection of use of an uninitialized variable is not possible. Portability is only ensured for well-formed programs, which this specification describes. Compilers are encouraged to detect ill-formed programs and issue diagnostic messages, but are not

INTRODUCTION

Chapter 1 and 2 from "The OpenGL Shading Language" by John Kessenich, Dave Baldwin and Randi Rost
<http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf>

required to do so for all cases. Compilers are required to return messages regarding lexically, grammatically, or semantically incorrect shaders.

1.6 Typographical Conventions

Italic, bold, and font choices have been used in this specification primarily to improve readability. Code fragments use a fixed width font. Identifiers embedded in text are italicized. Keywords embedded in text are bold. Operators are called by their name, followed by their symbol in bold in parentheses. The clarifying grammar fragments in the text use bold for literals and italics for non-terminals. The official grammar in Section 9 "Shading Language Grammar" uses all capitals for terminals and lower case for non-terminals.

2 OVERVIEW OF OPENGL SHADING

The OpenGL Shading Language is actually two closely related languages. These languages are used to create shaders for the programmable processors contained in the OpenGL processing pipeline. The precise definition of these programmable units is left to separate specifications. In this document, we define them only well enough to provide a context for defining these languages.

Unless otherwise noted in this paper, a language feature applies to all languages, and common usage will refer to these languages as a single language. The specific languages will be referred to by the name of the processor they target: vertex or fragment.

2.1 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:

- Vertex transformation (modelview and projection matrices)
- Normal transformation and normalization
- Texture coordinate generation
- Texture coordinate transformation
- Lighting
- Color material application

Programs written in the OpenGL Shading Language 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 list above are responsible for writing the code for all desired functionality from the list above. For instance, it is not possible to use the existing fixed functionality to perform the vertex and normal transformation but 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, such as:

- Perspective division
- viewport mapping
- Primitive assembly
- Frustum and user clipping
- Backface culling
- Two-sided lighting selection
- Polymode processing
- Polygon offset

- Depth Range

Any OpenGL state used by the shader is automatically tracked and made available to the shader. This automatic state tracking mechanism allows the application 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. The design of the vertex processor is focused on the functionality needed to transform and light a single vertex. Vertex shaders must compute the homogeneous position of the coordinate, and they 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. This programmable unit does not have the capability of reading from the frame buffer. However, it does have texture lookup capability. Level of detail is not computed by the implementation for a vertex shader, but can be specified in the shader. The OpenGL parameters for texture maps define the behavior of the filtering operation, borders, and wrapping.

2.2 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:

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

Programs written in the OpenGL Shading Language that are intended to run on this processor are called *fragment shaders*. Fragment shaders can be used to specify a completely general sequence of operations to be applied to each fragment. Fragment shaders that perform some of the computations from the list above must perform all desired functionality from the list above. 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 the fixed functionality graphics operations that occur at the back end of the OpenGL pixel processing pipeline such as:

- Shading model
- Coverage
- Pixel ownership test
- Scissor
- Stipple
- Alpha test
- Depth test
- Stencil test
- Alpha blending

- Logical ops
- Dithering
- Plane masking

Related OpenGL state is also automatically tracked if used by the shader. A fragment shader cannot change a fragment's 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. The values computed by the fragment shader are ultimately used to update frame-buffer memory or texture memory, depending on the current OpenGL state and the OpenGL command that caused the fragments to be generated.

The OpenGL parameters for texture maps 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 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 may compute color and/or depth, 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 1.4. 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. The primary reason for keeping the fixed functionality at the back end of the processing pipeline is that the fixed functionality is cheap and easy to implement in hardware. Making these functions programmable is more complex, since 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.