

OpenGL 2.0 Overview

Copyright © 2002 3Dlabs, Inc. Ltd.

This paper is distributed for the sole purpose of soliciting feedback on a possible OpenGL 2.0 API specification proposal to the OpenGL Architecture Review Board (ARB). Permission to reproduce and distribute it is granted for this purpose only, and any such reproduction must include the complete document including this disclaimer. Reproduction or distribution for any other reason requires the prior written consent of 3Dlabs.

This paper 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 paper, then 3Dlabs would grant a royalty free license to SGI, according to the ARB bylaws, for only that 3Dlabs intellectual property as is required to produce a conformant implementation.

This paper 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.

If you wish to provide feedback to 3Dlabs on this material, please send email to OGL2@3dlabs.com to request a Review Agreement.

Change History

Issue	Date	Change
1.0	12 Oct 01	First issue.
1.1	07 Dec 01	Added diagram and text to explain application transition path Updated to reflect changes to OpenGL 2.0 object management Updated to reflect changes to Asynchronous OpenGL Added classification of existing OpenGL extensions
1.2	25 Feb 02	Added program objects Added clarification to PINNED memory policy Updated the proposal for a pure OpenGL

Contents

1	MOTIVATION	4
2	GOALS	4
3	OVERVIEW OF OPENGL 2.0	6
3.1	THE VERTEX PROCESSOR	8
3.2	THE FRAGMENT PROCESSOR	9
3.3	THE UNPACK PROCESSOR	11
3.4	THE PACK PROCESSOR	12
3.5	OPENGL 2.0 OBJECTS	12
3.6	AUX DATA BUFFERS	14
3.7	PERFORMANCE ISSUES WITH OPENGL-MANAGED OBJECTS	15
3.8	ASYNCHRONOUS BEHAVIOR AND SYNCHRONIZATION	16
4	OPENML REQUIREMENTS	17
5	STANDARDIZATION OF NON-CORE FEATURES	18
6	A DEFINITION FOR PURE OPENGL 2.0	18
6.1	FUNCTIONS AVAILABLE IN PURE OPENGL 2.0	18
6.2	LEGACY MODE FUNCTIONS	22
7	CLASSIFICATION OF EXISTING EXTENSIONS	25
7.1	ARB EXTENSIONS BY NUMBER	25
7.2	NON-ARB EXTENSIONS BY NUMBER	25
7.3	UNNUMBERED EXTENSIONS	29

OpenGL 2.0 Overview

Version 1.2
February 25, 2002

Randi J. Rost
3Dlabs, Inc.

Abstract

This document presents an overview of the changes being proposed by 3Dlabs to create an OpenGL 2.0 specification. The first two sections discuss the motivation and goals of the effort. The next section contains an overview of the proposed changes, including a high-level block diagram. Subsequent sections provide more details of the proposed changes.

1 Motivation

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. A paradigm shift is occurring today, as 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.

This paradigm shift brings with it the opportunity to address some of the major issues facing OpenGL today, namely:

- System and graphics architectures have changed significantly since 1992, but OpenGL has not changed to keep pace.
- The growing perception that OpenGL is lagging behind the advanced functionality and performance of current graphics hardware.
- The complexity of implementing OpenGL or writing an OpenGL application given the sheer number of OpenGL extensions that exist today.
- The desire to develop OpenGL products in markets other than those that have been traditional OpenGL strongholds.

2 Goals

To address the issues described above, we have defined the following goals for the OpenGL 2.0 project:

Bring OpenGL to a level that reflects the capabilities and performance of current and near-future graphics hardware.

Since its introduction, the Direct3D API has made great strides in both functionality and performance. The rate at which it has been changed has led to consternation by some who have used it, but certain markets are willing to live with the pace of change in order to reap the advantages of ever-increasing functionality and performance. OpenGL has not changed nearly as rapidly. While the stability of OpenGL was once one of its strengths, it is becoming more of a liability. We are in the situation now where applications written to OpenGL cannot use many of the new features available in today's graphics hardware. Furthermore, OpenGL does not present an interface that allows the highest possible immediate mode performance on today's hardware. These issues must be addressed quickly so that OpenGL will remain a viable cross-platform option for application developers.

Provide a vision for the development of future generations of programmable graphics hardware.

For some time now, hardware vendors have been able to implement OpenGL on a single chip. Recent progress in evolving the API has been evolutionary, as hardware vendors have exposed minor new features through extensions, some of which eventually get standardized as ARB extensions or rolled into a minor revision of the OpenGL spec. This approach is not viable for dealing with the revolutionary change that is occurring as a result of the paradigm shift to programmable hardware. Our view is that there should be one high level language for writing both vertex shaders and fragment shaders, and it should be a high level language rather than an assembler or a microcode language. We think this is a healthy situation, since it causes the discussion of issues to rise above squabbling over who currently has the better feature set. It provides a consistent framework and a vision for the direction in which graphics hardware should evolve. Hardware vendors will have the ability to innovate underneath a high level language, and application developers will have a standard high level language with which to access programmable hardware capabilities.

Reduce the need for existing and future extensions to OpenGL by replacing complexity with programmability.

Over 230 extensions to OpenGL have been defined. Some companies have implemented as many as 70 extensions and have extension documentation that is twice the length of the OpenGL specification itself. A great number of these extensions can be eliminated if the right kind of programmability can be defined in the right places in OpenGL. Part of the OpenGL 2.0 effort is to make a conscious effort to survey the existing extensions and eliminate the need for as many as possible by replacing OpenGL's fixed functionality with programmability.

Define OpenGL 2.0 in such a way that there is a compelling compatibility story for existing applications.

OpenGL has proven to be a well-designed and long-lasting graphics API. The stability of OpenGL has been one of its key strengths since its introduction. While certain parts of the API need to change to reflect today's realities, much of the API remains as valid today as it did 10 years ago. Some applications will want to change very little as OpenGL moves forward. For this reason, we believe it is necessary to provide a method that allows OpenGL 1.3 applications to run unmodified and at high performance on an OpenGL 2.0 implementation. Furthermore, we

believe it is also necessary to allow OpenGL 1.3 applications to gradually begin using OpenGL 2.0 features (e.g., easily mixing fixed functionality state management with the programmability of OpenGL 2.0).

Define OpenGL 2.0 in a way that allows easier deployment in markets without backwards compatibility requirements.

The programmability of OpenGL 2.0 obviates the need for some of the fixed functionality defined in OpenGL 1.3. For markets that do not require a compelling compatibility story, it should be possible to define a much simpler, cleaner core of OpenGL 2.0 that eliminates redundant and out-of-date features. This would permit more rapid development of OpenGL-based hardware, drivers, and applications. It could also spur the growth and acceptance of OpenGL in markets other than the traditional workstation markets.

Address the needs of dynamic media authoring and playback applications as elucidated by the Khronos Group.

Several OpenGL extensions have been defined as requirements for OpenML 1.0. Moving forward, there is a large overlap between the goals listed above and the high priority graphics requirements for the next version of OpenML (e.g., vertex and fragment programmability). Other requirements for OpenML 1.1 include texture and memory management improvements and improved synchronization and time control mechanisms. There is also a great deal of interest in the Khronos Group for defining small footprint versions of OpenGL for deployment in a variety of embedded markets.

3 Overview of OpenGL 2.0

In order to describe the changes more effectively, we say that something is considered “legacy mode” if it is redundant with a new feature being introduced and is needed only for an OpenGL 2.0 implementation that supports backward compatibility with OpenGL 1.3. Both the legacy mode feature and the new feature are available to applications, and semantics are defined for how the two interact. Our assumption is that “legacy mode” features will not be available in some implementations, and we refer to such implementations as “Pure OpenGL 2.0”.

Figure 1 shows the transition path for OpenGL applications running today on OpenGL 1.3 (or an earlier version of OpenGL). Since OpenGL 2.0 is 100% backwards compatible with OpenGL 1.3, existing applications can run without modification on OpenGL 2.0. As development schedules allow, applications can gradually begin using OpenGL 2.0 features while continuing to use OpenGL 1.3 features. Pure OpenGL 2.0 will be the architectural basis for future versions of OpenGL, so we expect applications to eventually migrate to the functionality of Pure OpenGL 2.0 or use a compatibility library that sits on top of it. New applications or applications without legacy requirements can begin using Pure OpenGL 2.0 immediately and be well-positioned for the future evolution of OpenGL.

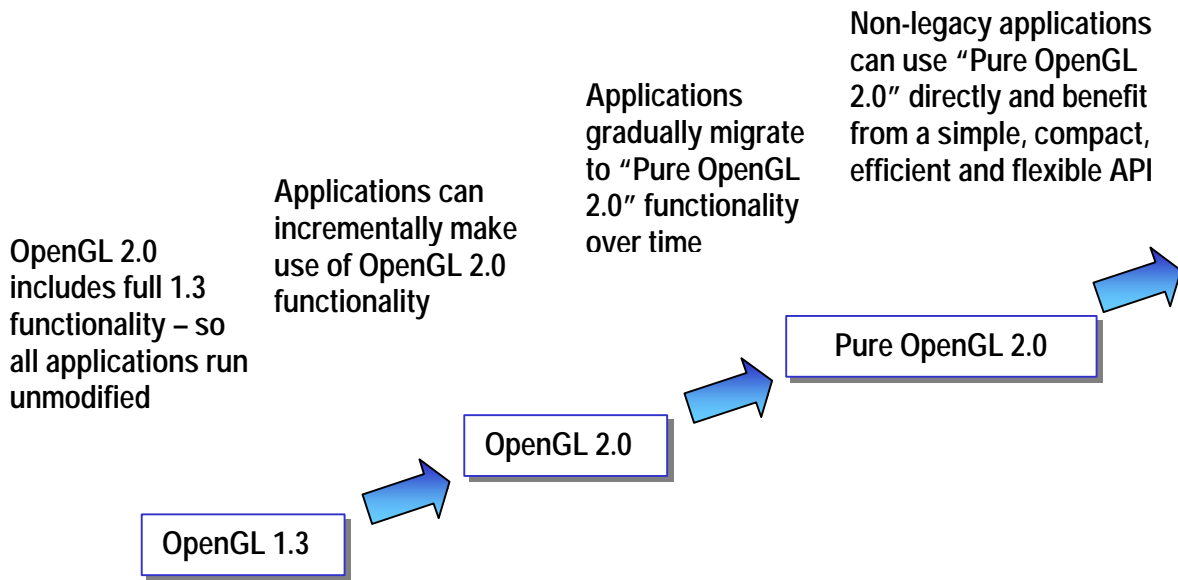


Figure 1: OpenGL Transition Path

Figure 2 is a simplified version of the logical diagram for OpenGL 2.0. It is based on the logical diagram called *The OpenGL Machine* published with the OpenGL Reference Manual (blue book). It illustrates some of the primary differences between OpenGL 2.0 and OpenGL 1.3. It should not be interpreted as an implementation diagram, it is a logical diagram that illustrates the state blocks, processing modules, and data paths in OpenGL 2.0.

The OpenGL 2.0 state machine is very similar to that of OpenGL 1.3, but several areas of fixed functionality have been replaced with programmable processors. More specifically, Figure 2 can be considered a logical diagram for Pure OpenGL 2.0, since it only shows the programmable units and not the fixed functionality that would also be supported in an implementation of OpenGL 2.0 that supports legacy mode.

The newly defined programmable units are shown as blue (shaded) stars. State blocks that remain the same as in OpenGL 1.3 are shown in white rectangles. To simplify the diagram, some of the OpenGL 1.3 functionality that remains the same in OpenGL 2.0 is grouped together in a single white rectangle (e.g., clip, project, viewport, cull). Memory that is under control of the application is shown on the left, and memory that is controlled by OpenGL is shown in orange (shaded) boxes. The arrows represent the primary flow of data. For immediate mode geometry commands, vertex data starts out in application memory and is sent down what is referred to as the geometry pipeline, generating pixels that are ultimately written into frame buffer memory. The application sends pixel data to the unpack processor and after rasterization, it is written to either the frame buffer or texture memory. The fragment processor can read texture memory during subsequent rendering operations. The application can read back pixels from the frame

buffer, optionally passing them through the fragment processor for processing, and then packing them in application memory under the control of the pack processor.

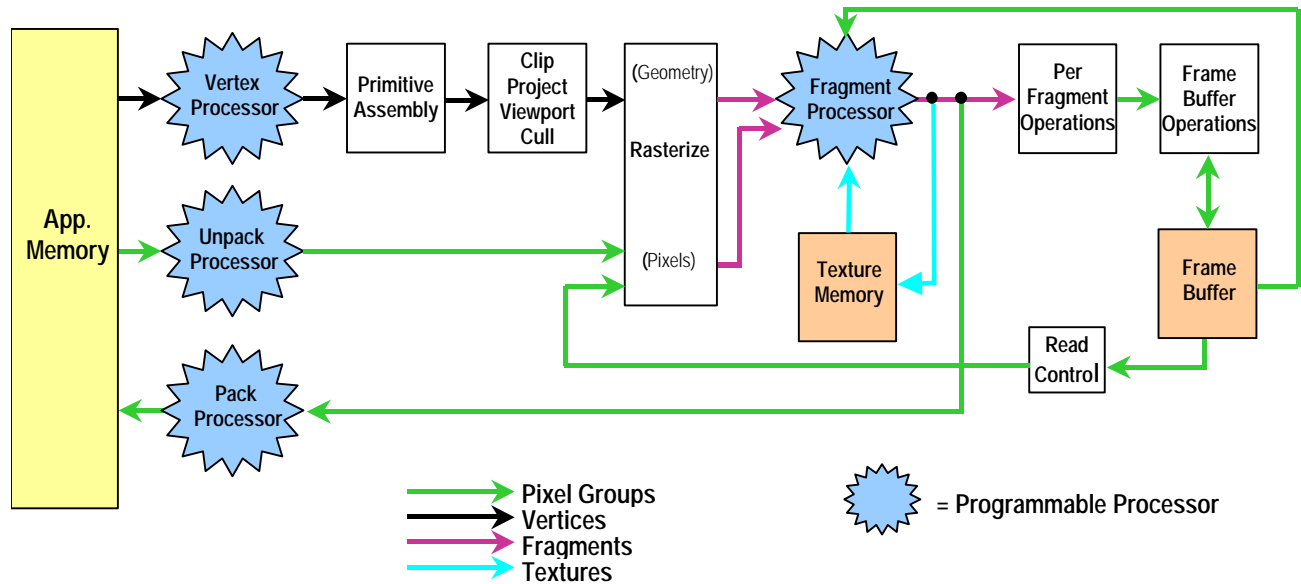


Figure 2: OpenGL 2.0 Logical Diagram

3.1 The Vertex Processor

The vertex processor is a programmable unit that is described in the *OpenGL 2.0 Shading Language* white paper. The programming language for this unit and the new OpenGL functions for creating and using programs (also called *shaders*) are defined in that paper.

When the vertex processor is active, the following OpenGL functions are disabled:

- Vertex transformation
- Normal transformation and normalization
- Texture coordinate generation
- Texture coordinate transformation
- Lighting
- Color material application
- Clamping of colors

The default state of the vertex processor is that it uses the shader object with an ID (or *handle*) of 0. This program mimics the default behavior of the OpenGL 1.3 fixed functionality state in the list above. In legacy mode, the fixed functionality state is active whenever the shader being used by the vertex processor has a handle of 0. Whenever a shader with a handle other than 0 is loaded into the vertex processor, it becomes active, and the fixed functionality is disabled

Vertex shaders that intend to perform computations similar to the fixed functionality of OpenGL 1.3 are responsible for writing the code for all of the functionality in the 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 program must be written to perform all three functions. In legacy mode, 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 will allow 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 language that is used to write programs for the vertex processor 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.

The vertex processor operates on one vertex at a time. This programmable unit does not have the capability of reading from texture memory or the frame buffer. The design of this unit is focused on the functionality needed to transform and light a single vertex. The vertex shader must compute the homogeneous position of the coordinate, and it may also compute color, texture coordinates, and other arbitrary values to be passed to the fragment processor. The output of the vertex processor is sent through subsequent stages of processing that are defined exactly the same as they are for OpenGL 1.3: primitive assembly, user clipping, frustum clipping, perspective projection, viewport mapping, polygon offset, polygon mode, shade mode, and culling.

After all this processing, vertex data arrives at the rasterization stage. OpenGL defines rasterization as consisting of two parts. The first part of rasterization is to determine which squares of an integer grid in window coordinates the primitive occupies. This portion of the rasterization process remains unchanged for OpenGL 2.0. The second part of rasterization, assigning color, depth, and stencil values to each square, is almost completely replaced in OpenGL 2.0 as described in the next section.

3.2 The Fragment Processor

The fragment processor (defined in the *OpenGL 2.0 Shading Language* white paper) is designed to operate on the fragments produced by the rasterization stage. In OpenGL, a fragment is defined as a grid square (i.e., an x/y position) and its assigned color, depth, and texture coordinates. In OpenGL 1.3, there are very strict rules for how the fragment's associated data can be modified. Additional rules have been added through extensions, exposing the greater flexibility of current hardware. In OpenGL 2.0, a programmable unit is defined that allows 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 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

and the following fixed functionality relating to pixel processing is also replaced:

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

Related OpenGL state is also automatically tracked if used by the shader. With the exception of a few built-in functions, the language used to program the fragment processor is nearly identical to the language used to program the vertex processor. A fragment shader can change a fragment's depth, color, and stencil value, but not its x/y position. To support parallelism at the fragment processing level, fragment shaders are written in a way that expresses the computation required for a single fragment, and access to neighboring fragments is not allowed. A fragment shader is free to read multiple values from a single texture, or multiple values from multiple textures. It is not allowed to directly write either texture memory or frame buffer memory, but it can directly read frame buffer memory at the current pixel location.

The OpenGL 1.3 parameters for texture maps are carried forward to OpenGL 2.0 and continue to define the behavior of the filtering operation, borders, and wrapping. These operations are applied when a texture is accessed. The fragment shader is free to use the resulting texel however it chooses. It is possible for a fragment shader to read multiple values from a texture and perform a custom filtering operation. It is also possible to use a 1D texture to perform a lookup table operation. In both cases the texture should have its texture parameters set so that nearest neighbor filtering is applied on the texture access operations.

One of the major areas of change in OpenGL 2.0 is in the support for pixel processing operations. The fragment processor defines almost all of the capabilities necessary to implement the pixel transfer operations defined in OpenGL 1.3. Rather than have an additional programmable unit that does nearly the same thing as the fragment processor, OpenGL 2.0 supports pixel processing with the fragment processor. Dedicated lookup tables are replaced 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 is promoted to full citizenship in OpenGL 2.0 (it was part of the optional imaging subset in 1.3) and can be accessed easily when matrix state tracking is enabled. Convolution and pixel

zoom are supported by writing an image to texture memory, and then accessing it as a texture while rendering a quadrilateral primitive. Histogram and minmax operations are left to be defined as extensions, since these prove to be quite difficult to support at the fragment level with high degrees of parallelism.

For each fragment, the fragment shader can compute color, depth, stencil, or one or more arbitrary data values. A fragment shader must compute at least one of these values. The color, depth, and stencil values will remain as computed by the previous stages in the OpenGL pipeline if the fragment shader does not modify them.

The results of the fragment shader are then sent on for further processing. The remainder of the OpenGL pipeline remains as defined in OpenGL 1.3. Fragments are submitted to coverage application, pixel ownership testing, scissor testing, alpha testing, stencil testing, depth testing, blending, dithering, logical operations, and masking before ultimately being written into the frame buffer. 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 the application can perform alternate operations within a fragment shader.

3.3 The Unpack Processor

Another part of OpenGL that has been prone to numerous extensions is pixel unpacking. This part of OpenGL is responsible for extracting pixel values (color, depth, or stencil) in the application's memory space and transferring them to OpenGL. The original design for OpenGL specified quite a few combinations of format (element order and meaning) and type (GL data type, e.g., byte, short, float, etc.). The list of combinations was lengthened considerably when a number of new packed pixel formats were added in OpenGL 1.2, and there is continuing pressure to support even more combinations (e.g., OML_interlace, OML_subsample, OML_resample).

To address the growing complexity of this area, we are proposing a programmable unit called the unpack processor. The purpose of this unit is to provide a very simple but flexible way for an application to define the layout of pixel data in application memory and describe the operations necessary to produce a coherent stream of pixels for consumption by the fragment processor. The unpack processor and the C-like language used to program it are defined in the *OpenGL 2.0 Pixel Pipeline* white paper. Although much more limited, this language is similar to the language used by the vertex and fragment processors. The language supports several data types, simple integer operations, conversion to and from floats according to OpenGL rules, and looping constructs.

One of the main elements of this language is the input buffer, which the application describes in an unpack program as a C-like structure. The fields in this structure are defined by type and size (in bits), so it is possible for an application to describe any of the existing type and format combinations in a clear and straightforward fashion. The statements in the unpack program then

describe how pixel data is extracted from the input buffer and converted into a coherent stream of pixel groups that are sent on to the fragment processor.

Mechanisms for creating, compiling, linking, using, and deleting programs for the unpack processor are the same as for the vertex and fragment processors. This programmable unit is deliberately designed with fewer capabilities than the vertex and fragment processors. It cannot do floating-point math, limited integer math, and there are only a few built-in functions. The emphasis is on making this unit as simple to implement in hardware as it can be, while still providing the full generality needed for unpacking pixel data. Operations are limited to those that can be applied to a data stream. Temporary storage is also extremely limited. It is reasonable to write a program that buffers up two or three pixels, but not hundreds of pixels.

3.4 The Pack Processor

Pixel packing in OpenGL refers to the process of taking values read from the frame buffer or texture memory and formatting them to be written back into application memory. The process is similar in many ways to pixel unpacking, but the two operations are not symmetrical. The programmable unit that is introduced to handle this process for OpenGL 2.0 is called the pack processor. The operation of this unit and the language used to program it are defined in the *OpenGL 2.0 Pixel Pipeline* white paper. Mechanisms for creating, compiling, linking, using, and deleting programs for the pack processor are the same as for the vertex, fragment, and unpack processors.

The language that is used to program the pack processor is very similar to that used to program the unpack processor. One of the key differences is that the pack processor defines an output buffer, rather than an input buffer. The purpose of the pack program is to read pixels from the frame buffer or from texture memory, process them as needed, and format them into the output buffer from whence they will be written into the application's memory space.

Another difference between the pack and unpack processors is that the coherent stream of pixels created by the unpack processor is defined to flow to the fragment processor and the operations that follow (see Figure 1). In the case of the pack processor, it is the fragment processor that actually reads the frame buffer or texture memory, and the output of the fragment program is then made available to the pack processor. Some applications will want to use the power of the fragment processor to perform color matrix, scale and bias, and color lookup operations on the data as it is being read from the frame buffer or texture memory.

3.5 OpenGL 2.0 Objects

OpenGL 1.1 introduced texture objects, which were designed to provide better performance for texturing operations and to communicate more information from the application to the OpenGL implementation about how and when to use them. To retain compatibility with OpenGL 1.0, texture objects were added in a way that was compatible with the OpenGL 1.0 notion of texture targets. Although somewhat clumsy, OpenGL 1.1 texture objects have proved their worth, and OpenGL 2.0 builds on this experience to add new object types and harmonize the framework for dealing with objects.

In OpenGL 2.0, an *object* is an OpenGL-managed opaque data structure that consists of state and data. The data portion of an object may be quite large, so applications are provided more control over how objects are managed. All objects are given names (or handles) by OpenGL at object creation time. Applications can specify the data that is to be stored in objects and can modify their state through the OpenGL 2.0 API. Objects can be created, deleted, modified, and used as part of the current rendering state.

OpenGL 2.0 introduces 9 types of objects: texture objects, cube map objects, vertex array objects, image objects, display list objects, shader objects, program objects, frame buffer objects, and buffer objects. Some objects (e.g., cube map, texture, and frame buffer objects) are simply containers to which other objects can be attached and don't contain data. Other objects (e.g., image, vertex array, display list, shader, and buffer objects) are designed specifically to serve as storage for a (potentially large) data structure. Program objects are somewhat unique, in that they serve as containers for shader objects and they contain the data for an executable program once linking has occurred.

Image Objects – Image objects are new to OpenGL 2.0. These objects are the primary storage for pixel data in OpenGL memory. Pixel data that is to be used as part of a texture is stored in an image object, and then the image object is attached to a texture object to define the texture. Image objects can also be used as the source or destination of copy operations. OpenGL 2.0 provides new entry points that allow image objects to have pixel data loaded into them or to be rendered (see the *Minimizing Data Movement and Memory Management* white paper).

Texture Objects – Texture objects in OpenGL 2.0 are replacements for OpenGL 1.1-style texture objects. The main difference is that OpenGL 2.0 assigns handles with the 2.0-style texture objects. Certain new functionality in OpenGL 2.0 is defined to work only with the 2.0-style texture objects. Texture objects can be thought of as container objects. They do not directly contain pixel data, but do contain handles of one or more image objects.

Cube Map Objects – Cube map objects are new in OpenGL 2.0 and are designed to duplicate the functionality of cube maps in OpenGL 1.3. Cube map objects are also container objects. They do not directly contain pixel data, but they do contain handles for six texture objects that represent the faces of the cube map. These texture objects in turn contain handles of one or more image objects that contain the pixel data to be used in texturing operations involving the cube map.

Vertex Array Objects – Several OpenGL extensions have been defined that allow vertex arrays to be stored directly on the graphics card. OpenGL 2.0 eliminates the need for these extensions by introducing vertex array objects, allowing vertex array data to be managed in OpenGL memory (including memory on the graphics card) just like all other objects.

Display List Objects – Display list objects are new in OpenGL 2.0, but they are very nearly the same as display lists in OpenGL 1.0-1.3. One difference is that with OpenGL 2.0, display list objects have names that are assigned by OpenGL 2.0 rather than by the application. Display list objects are subject to the same memory management policies as other objects (see the white

paper *Minimizing Data Movement and Memory Management*). This eliminates the need for OpenGL extensions whose purpose is to define a mechanism that allows applications to indicate that display lists should be stored in memory on the graphics card.

Shader Objects – Shader objects are used to store the source code for programs (shaders) that are to be executed on one of the programmable units in OpenGL 2.0. OpenGL 2.0 provides new functions that allow them to have source code loaded into them and to be compiled (see the white paper *OpenGL 2.0 Shading Language*).

Program Objects – Program objects are container objects that are used to define a set of shaders that are linked together to form an executable program. Shader objects that are attached to a program object are checked for compatibility, unresolved references, and so on. A program object that is valid (i.e., has no link errors) can be used to set the current program for one or more of the programmable processors in OpenGL 2.0.

Frame Buffer Objects – OpenGL 2.0 introduces the notion of a configurable frame buffer. In order to give applications more flexibility in how the frame buffer is allocated, frame buffer objects are defined. Frame buffer objects are container objects that have buffer objects (e.g., depth buffer, stencil buffer, accumulation buffer, etc.) attached to them. Frame buffer objects can be visible (i.e., part of a window that is visible on the display device) or offscreen (i.e., part of offscreen video memory). All rendering operations in OpenGL 2.0 are directed toward a frame buffer object.

Buffer Objects – A variety of buffers have been defined in OpenGL, and the object framework in OpenGL 2.0 provides a way for an application to manage these buffers in a flexible way – allocating them when needed and deallocating them when they are no longer needed. Buffer objects come in several different flavors (color, depth, stencil, accumulation, etc.). These objects are the ones that actually store the values associated with rendered pixels.

For more information about OpenGL 2.0 objects and the basic operations that can be performed on them see the *OpenGL 2.0 Objects* white paper. Additional information about managing objects can be found in the white paper *Minimizing Data Movement and Memory Management*.

3.6 Aux Data Buffers

Fragment shaders can easily produce results other than those intended for display. OpenGL 2.0 defines a new type of buffer called the aux data buffer. This buffer is specifically designed to hold floating-point data in order to provide a simple, generalized mechanism that enhances programmability. It is the primary mechanism in OpenGL 2.0 for allowing applications to directly implement multipass algorithms. It can be used to temporarily store multiple outputs from the fragment processor, store intermediate results from a multipass algorithm, store results from multi-spectral image processing operations, or as a destination for computing a floating-point image that can then be read back to the host for archival or further processing.

The aux data buffer is a type of buffer object, and as such it can be attached to a frame buffer object and used as the target for rendering operations. More than one aux data buffer object may

be attached to a frame buffer object, and each aux data buffer can hold 1, 2, 3, or 4 floating-point values. New functions are defined to set the value used to clear aux data buffers and to process the aux buffer (i.e., send all the aux data buffer values through the fragment processor for further processing).

For more information on the aux data buffer, see the *OpenGL 2.0 Objects* white paper.

3.7 Performance Issues with OpenGL-Managed Objects

In certain areas, OpenGL has defined “black box” behavior, meaning that the knowledge of underlying mechanism is hidden from applications. This is fine in many cases, as applications do not need to be bothered with many of the details of how OpenGL is implemented. However, system architecture has changed over the years. Some of the main performance issues faced by applications today are caused by the fact that there is no way for an application to provide OpenGL with enough information to allow maximum use of available memory bandwidth or bandwidth to the graphics subsystem. A number of extensions have been proposed to deal with certain aspects of this problem. The white paper *Minimizing Data Movement and Memory Management* looks at this problem across all of OpenGL and proposes a framework for better communication of performance critical information between applications and OpenGL.

The issue with data movement centers on memory that is managed by OpenGL. In OpenGL 1.3, textures are allowed to be stored in high performance graphics memory near the graphics hardware. With the introduction of vertex array objects in OpenGL 2.0, vertex data can also be stored in high performance graphics memory near the graphics hardware. This proposal provides an architectural framework that allows incorporation of several similar ideas that were previously implemented as vendor-specific extensions to OpenGL.

New commands are introduced in order to query and make decisions about memory operations for any of the defined OpenGL objects. Objects can be given a specific memory management policy as well as hints that describe how the application intends to utilize the object. Currently defined memory management policies are POLICY_DEFAULT (just use the current OpenGL “black box” management policy), POLICY_PINNED_ACTIVE (the object will stay in high performance memory as long as the object’s context is active), and POLICY_PINNED_ALWAYS (the object will stay in high performance memory regardless of the status of its context). These policies refer to OpenGL’s treatment of memory, not to the underlying operating system’s treatment. An object with a policy of POLICY_PINNED_ALWAYS will not be moved by OpenGL, but it still could be paged out by the operating system in some situations. Usage hints are also defined that allow the application to communicate to OpenGL whether the object will be used only once, used many times, or is a type of object that will never be read, only written. These capabilities allow applications to communicate enough information to the OpenGL implementation to allow more optimal performance.

Applications have often struggled attempting to figure out whether a set of objects would fit into available memory on a graphics card. New query mechanisms are introduced that allow an application to query whether an object or a set of objects will fit when a specific memory

management policy is used. This gives applications the ability to make more intelligent decisions about how to use precious resources in the graphics subsystem. Mechanisms are also defined that allow applications to directly read and write memory that is accessible to the OpenGL implementation. This can improve performance by eliminating the need to store an extra copy of the data since the application and the OpenGL implementation share one and the same piece of memory for the data. Rendering speed can be improved since a copy operation is avoided.

These mechanisms allow applications to communicate more information to OpenGL about OpenGL-managed objects. This allows the OpenGL implementation to do a better job of optimizing the performance of such objects by minimizing copies, minimizing CPU usage, preserving system memory bandwidth, and preserving bandwidth to the graphics subsystem.

3.8 Asynchronous Behavior and Synchronization

A number of attempts have been made to define OpenGL extensions that give applications better control over when things happen in OpenGL. Some applications have real-time rendering requirements that are difficult or impossible to meet with the existing capabilities of OpenGL. Some of these capabilities have been provided through extensions. The OML_sync_control and async extensions were recent efforts that provided additional insight into the limitations of OpenGL in this area.

One of the current issues is that OpenGL has no way to synchronize with the completion of a set of commands. Applications can call Finish(), but this is a heavyweight operation that synchronizes with all previously issued commands. Because it blocks, Finish prevents the application from issuing additional commands while waiting for the previous commands to complete. Thus the CPU sits idle when it could be doing productive work for the application.

Another issue that arises is with commands that are lengthy to complete. OpenGL says that commands are bound synchronously, that is, all command data is copied before control is returned to the application. Large image downloads are one example of where this semantic hurts parallelism as well.

The *Asynchronous OpenGL* white paper addresses these and other, related issues. To provide finer-grained synchronization control, the GLsync data type is introduced. A GLsync can be supplied to a command that will use it to notify the application when the command completes. API entry points are defined for allocating, freeing, and waiting on a GLsync as well as for waiting on multiple GLsyncs. The GLsync data type gives applications a simple, but powerful synchronization tool. Driver developers can also implement it easily on top of existing operating system functionality.

A new command, Fence(), is defined to provide applications with an efficient mechanism for synchronizing a thread with an arbitrary point in the rendering command stream. The Fence() command takes a GLsync as a parameter, and it causes the specified GLsync to be set once all the commands issued prior to the fence have been completed.

The GLsync data type is also used as a parameter in several new commands that are bound asynchronously (i.e., control is returned to the application before the OpenGL implementation has completed copying the data to OpenGL memory). Asynchronous versions of various lengthy rendering commands are defined, and for each of these commands, the associated GLsync is set when the data binding is complete. A Fence() command can be used to determine when the operation is complete.

To provide notification of periodic events such as vertical refresh (i.e., the beginning of the vblank period), we introduce new commands that provide a GLsync that is momentarily set when such an event occurs.

Better control over buffer swapping is also provided. New swap commands are introduced that let the application control whether the swap is inserted into the queue of currently pending rendering commands, or whether the swap should happen immediately. For the latter, the application can use GLsyncs to wait until particular rendering is complete and/or wait for a particular vertical blank to occur before requesting a swap. In the meantime, neither the host nor the graphics subsystem need be stalled. The new swap commands are never required to do finish or flush operations.

4 OpenML Requirements

OpenML 1.0 requires the presence of the imaging subset, which is an optional part of OpenGL 1.3. This functionality, with a couple of exceptions, is promoted to full citizenship in OpenGL 2.0. The functionality of the imaging subset is replaced primarily by the programmable capabilities of the fragment processor. The color matrix, constant blend color, blend minmax, and blend subtract are made part of standard 2.0. Convolution and scale and bias can be implemented in a fragment shader. Lookup tables can be implemented in a fragment shader as 1D texture references. Histogram and min/max functions are dropped, since they are of questionable value at this point in the processing pipeline and since they are very difficult to implement in a high performance fashion on parallel graphics hardware.

The OML_interlace, OML_subsample and OML_resample extensions are requirements for OpenML 1.0. The *Pixel Pack/Unpack Processor* white paper shows how these extensions can be supported through the combination of pack, unpack, and fragment shaders.

The OML_sync_control extension is another requirement of OpenML 1.0 and it can be implemented on top of the mechanisms defined in the *Asynchronous OpenGL* white paper. The lone exception is in support for Unadjusted System Time (UST). This capability is not defined as part of OpenGL 2.0 since it has a dependency on UST mechanisms defined by OpenML.

The SGIS_texture_color_mask and the EXT_texture_lod_bias extensions are required for OpenML 1.0. We are proposing that these be promoted to full citizenship in OpenGL 2.0.

Moving forward, the list of high priority items for OpenML 1.1 includes:

- Extended precision and greater dynamic range for colors (much of what's needed is provided by fragment processor and aux data buffer capabilities)

- Render to texture (supported by the OpenGL 2.0 object framework)
- Memory and texture management improvements (addressed through the mechanisms defined in the *Minimizing Data Movement and Memory Management in OpenGL* white paper)
- Support for non-power of 2 textures (supported with the new texture and image objects)
- Fragment shaders and vertex shaders (standard in OpenGL 2.0)
- Support for asynchronous binding of lengthy operations (addressed through the mechanisms defined in the *Asynchronous OpenGL* white paper)
- Texture filtering for interlaced video (supported with customized filters implemented in a fragment program)

5 Standardization of Non-Core Features

Certain operations that are fundamental to OpenGL were left to be defined as part of the window-specific “glue” libraries for different operating environments. This has made it harder to specify these operations and has hampered application portability.

We propose that certain key operations be defined as part of the OpenGL 2.0 specification, even if they have window-specific bindings. These operations include:

- CreateContext
- GetCurrentContext
- GetCurrentDC/Drawable
- MakeCurrent
- StreamSwapBuffers
- AsyncSwapBuffers

Certain requirements should also be defined in the OpenGL specification for creating a drawing surface (window) and choosing the format of attached buffers. These changes would give architectural control of key features back to the OpenGL ARB.

6 A Definition for Pure OpenGL 2.0

Although it is bound to be a controversial topic, this section makes an initial proposal about how Pure OpenGL 2.0 might be defined. Pure OpenGL 2.0 can be thought of as containing all of the functionality of full OpenGL 2.0 with the minimum set of OpenGL 2.0 entry points. Where there is redundant functionality, we pick that which we believe is the most useful going forward. The goal is to define a clean, consistent API for new applications, and to make it easier for driver implementers focused on markets where backwards compatibility is less important.

6.1 Functions available in Pure OpenGL 2.0

Primitives

The following commands are retained in Pure OpenGL 2.0 for their general usefulness. However, application writers should be aware that the vertex API entry points should not be used if the goal is achieving the highest possible performance.

Vertex, Normal, Color, Index, TexCoord, MultiTexCoord, EdgeFlag, Begin, End, ArrayElement

Vertex Arrays

Commands pertaining to vertex arrays are considered to present the highest performance API for drawing geometry. A number of OpenGL 1.3 commands are superseded by the capabilities defined in the *Minimizing Data Movement and Memory Management* white paper.

DrawArrays, DrawElements, DrawRangeElements, InterleavedArrays, VertexArrayPointer, DrawIndexedArrays, DrawRangeIndexedArrays, DrawArraysAsync, DrawIndexedArraysAsync, DrawRangeIndexedArraysAsync, BindArrayToIndex, LoadVertexArrayData, LoadVertexArrayFormattedData, StartVertexArrayFormat, AddElement, EndVertexArrayFormat

Coordinate Transform

Matrix commands in OpenGL 1.3, including pushing and popping, are an integral part of Pure OpenGL 2.0. The color matrix that was optional in the OpenGL Imaging Subset is made a first class citizen.

DepthRange, Viewport, MatrixMode, LoadMatrix, MultMatrix, LoadTransposeMatrix, MultTransposeMatrix, LoadIdentity, Rotate, Translate, Scale, Frustum, Ortho, ActiveTexture, PushMatrix, PopMatrix

Coloring and Lighting

Most of the functionality of this section is superseded by the capabilities of the vertex processor in OpenGL 2.0. The remaining commands are:

FrontFace, ShadeModel

Clipping

User clipping is a valuable capability for certain classes of applications.

ClipPlane, GetClipPlane

Rasterization

RasterPos is superseded by ImagePos and Bitmap is superseded by DrawPixelRect.

ImagePos PointSize, LineWidth, LineStipple, CullFace, PolygonStipple, PolygonMode, GetPolygonStipple

Pixel Operations

Pixel operations are partially superseded by the programmable pack and unpack processors of OpenGL 2.0. Pixel zoom, pixel map, and pixel transfer operations are subsumed by the capabilities of the programmable fragment processor.

PixelStore, DrawPixelRect, DrawPixelStream, ReadPixelRect,
ReadPixelStream, ReadImageData

Textures

The following commands affecting textures are defined as part of Pure OpenGL 2.0:

LoadImageData1D/2D/3D, LoadSubImageData1D/2D/3D,
DrawImageObject, CopyImage1D/2D/3D

Frame Buffer Operations

The blending modes defined as optional in the OpenGL Imaging Subset are required in Pure OpenGL 2.0. The image mask (a.k.a., texture color mask) is promoted to a first class citizen as well.

Scissor, SampleCoverage, AlphaFunc, StencilFunc, StencilOp,
DepthFunc, BlendColor, BlendEquation, BlendFunc, LogicOp,
DrawBuffer, ReadBuffer, IndexMask, ColorMask, DepthMask,
StencilMask, ImageMask, Clear, ClearColor, ClearIndex,
ClearDepth, ClearStencil, ClearAccum, Accum, ClearAuxData,
ProcessAuxData

Selection and Feedback

We currently believe that feedback mode should be deprecated. There is also a temptation to deprecate selection mode, since it invariably needs to be implemented in software. However, the functionality is useful and it remains in OpenGL 2.0 at this point.

InitNames, PopName, PushName, LoadName, RenderMode, SelectBuffer

Display Lists

The sole reason for the existence of display lists is to achieve better performance than through immediate mode. Some believe that with the addition of the memory management and direct access features of OpenGL 2.0, there is no longer a compelling need for display lists. For now, we retain them as part of OpenGL 2.0.

EndList, BeginList, DrawList, DrawLists

Modes and Execution

Although Enable and Disable are supported in Pure OpenGL 2.0, some of the modes they control apply to legacy mode functionality and are not part of Pure OpenGL 2.0. A complete list is TBD.

Enable, Disable, Hint

State Queries

Some thought has been given to eliminating most queries in OpenGL. But these commands may be useful for application debugging, so they are left in for now. Pushing and popping attributes are also of somewhat questionable value for sophisticated applications. They are useful for display list mode, however, so they are left in for now.

GetError, GetBoolean, GetInteger, GetFloat, GetDouble,
IsEnabled, GetPointerv, GetString, PushClientAttrib, PopAttrib,
PushAttrib, PopClientAttrib, GetFormats, GetHandle

Programmability

A number of new functions to support programmability are defined in the white paper *OpenGL 2.0 Shading Language*. These functions are required as part of Pure OpenGL 2.0. Functions prefixed with Standard will return only values defined for Pure OpenGL 2.0, and legacy mode state will not be included.

GetInfoLog, LoadShader, AppendShader, CompileShader,
LinkProgram, VertexAttribute, VertexAttributes,
BindAttributeLocation, LoadUniform, LoadUniformInt,
LoadUniformBool, LoadUniformArray, LoadUniformMatrix,
LoadUniformMatrixArray, GetUniformLocation

Synchronization

A number of new functions to support synchronization and increased parallelism are defined in the white paper *Asynchronous OpenGL*. These functions are required as part of Pure OpenGL 2.0.

SyncAlloc, SyncFree, SyncWait, SyncWaitMultiple,
wglConvertSyncToEvent/wglConvertEventToSync (and something
equivalent for GLX), FlushStream, Fence, QueryFence,
StreamSelect, StreamCopyContext, QueryFenceBackground,
wgl/glxGetVerticalBlankPulse, FenceWait,
wgl/glxGetVerticalBlankRate, wgl/glxGetVerticalBlankCount,
StreamSwapBuffers, wgl/glxAsyncSwapBuffers AllocateStreamFence,
InsertStreamFence, StreamFenceWait, DeleteStreamFence,
TriggerFence

Objects

The object framework described in the OpenGL 2.0 Objects white paper defines the following new entry points:

CreateTextureObject, CreateDisplayListObject,
CreateCubeMapObject, CreateImageObject1D/2D/3D,
CreateVertexArrayObject, CreateShaderObject,
CreateProgramObject, CreateFramebufferObject,
CreateBufferObject, GetFramebufferHandle, GetFormats,
DeleteObject, DeleteObjects, IsObject, ObjectParameterf/I,
GetObjectParameterfv/iv, UseImageObject, DetachObject,
AttachImageObject, UseTextureObject, AttachTextureObject,
UseVertexArrayObject, AttachShaderObject, UseProgramObject,
AttachBufferObject, UseFramebufferObject

Memory Management

A number of new functions to support object management are defined in the white paper *Minimizing Data Movement and Memory Management in OpenGL*. These functions are required as part of Pure OpenGL 2.0.

ObjectPolicy, ClearObjectPolicy, GetObjectPolicy,
ObjectPriority, GetObjectPriority, DefaultObjectPolicy,
DefaultObjectPriority, MakeSpaceForObjects, IsSpaceForObjects,
EnableDirectAccess, DisableDirectAccess, AcquireDirectPointer,
ReleaseDirectPointerRange, ReleaseDirectPointer1D/2D/3D

Asynchronous Commands

The mechanisms defined in the Asynchronous OpenGL white paper allow the introduction of commands that operate asynchronously, that is, control is returned to the application before the command has completed. The following asynchronous commands are defined as part of PureOpenGL 2.0.

LoadVertexArrayDataAsync, LoadVertexArrayFormattedDataAsync,

6.2 Legacy mode functions

Primitives

No need for this special purpose function, it can easily be implemented with other primitives.

Rect, PolygonOffset

Vertex Arrays

The following vertex array functions are rendered obsolete as a result of new capabilities introduced in the white paper *Minimizing Data Movement and Memory Management in OpenGL*.

EdgeFlagPointer, TexCoordPointer, ColorPointer, IndexPointer,
NormalPointer, VertexPointer, EnableClientState,
DisableClientState, ClientActiveTexture

Coloring and Lighting

These functions are superseded by the capabilities of the vertex processor in OpenGL 2.0:

Material, Light, LightModel, ColorMaterial, GetLight,
GetMaterial

Rasterization

Most of the rasterization functions remain in OpenGL 2.0. The Bitmap function is superseded by DrawPixelRect.

Bitmap

Pixel Operations

Pixel zoom, pixel transfer, and pixel map functionality is subsumed by the capabilities of the fragment processor. Histogram and Minmax are left as future extensions.

PixelTransfer, PixelMap, ColorTable, ColorTableParameter, CopyColorTable, ColorSubTable, CopyColorSubTable, ConvolutionFilter2D, ConvolutionParameter, ConvolutionFilter1D, SeparableFilter2D, CopyConvolutionFilter2D, CopyConvolutionFilter1D, ConvolutionParameter, Histogram, Minmax, PixelZoom, DrawPixels, ReadPixels, CopyPixels, GetPixelMap, GetColorTable, GetColorTableParameter, GetConvolutionFilter, GetSeparableFilter, GetConvolutionParameter, GetHistogram, ResetHistogram, GetHistogramParameter, GetMinmax, ResetMinmax, GetMinmaxParameter

Textures

The following functions are superseded by a combination of new functionality in OpenGL 2.0:

TexGen, TexImage1D/2D/3D, TexSubImage1D/2D/3D, GenTextures, TexEnv, GetTexEnv, GetTexGen, GetTexImage, BindTexture, DeleteTexture, AreTexturesResident, PrioritizeTextures, IsTexture, CompressedTexImage1D/2D/3D, CompressedTexSubImage1D/2D/3D, GetCompressedTexImage, TexParameter, GetTexParameter, GetTexLevelParameter

Fog

The fog functionality is superseded by the capabilities of the fragment processor in OpenGL 2.0

Fog

Evaluators

We anticipate supporting evaluators through (as yet undefined) programmable means in a version of OpenGL beyond 2.0. That functionality would be part of Pure OpenGL, and the functions below could be implemented on top of it.

Map1/2, EvalCoord1/2, MapGrid1/2, EvalMesh1/2, EvalPoint1/2, GetMap

Selection and Feedback

It is not clear how to effectively do feedback with the vertex processor as defined in OpenGL 2.0. We recommend deprecating this functionality.

FeedbackBuffer, PassThrough

Display Lists

The following display list functions are rendered obsolete as a result of new capabilities introduced in the white paper *Minimizing Data Movement and Memory Management in OpenGL*.

NewList, CallList, CallLists, ListBase, GenLists, IsList,
DeleteLists

Modes and Execution

The following commands are superceded by the capabilities defined in the Asynchronous OpenGL white paper:

Flush, Finish

7 Classification of Existing Extensions

As part of the effort of writing white papers to express our ideas for OpenGL 2.0, 3Dlabs surveyed all of the extensions listed in the OpenGL extension registry, and classified them according to their disposition with the OpenGL 2.0 functionality. It is interesting to note the vast number of extensions that are subsumed by the capabilities of the 3Dlabs OpenGL 2.0 proposal.

add - Suggested to add to OpenGL 2.0

core - Extension is now part of OpenGL 1.3 core

program - Extension is replaced by the shading language or pack/unpack language

async - Extension is replaced by the Asynchronous paper

mem - Extension is replaced by the memory management paper

objects - Extension is replaced by OpenGL 2.0 objects

NO - Lets not add it to OpenGL 2.0

? - Not sure what to do with this extension

7.1 ARB extensions by number

1. [GL_ARB_multitexture](#) - core
2. [GLX_ARB_get_proc_address](#) - core
3. [GL_ARB_transpose_matrix](#) - core
4. [WGL_ARB_buffer_region](#) - objects
5. [GL_ARB_multisample](#) - core
6. [GL_ARB_texture_env_add](#) - program
7. [GL_ARB_texture_cube_map](#) - program
8. [WGL_ARB_extensions_string](#) - add
9. [WGL_ARB_pixel_format](#) - add
10. [WGL_ARB_make_current_read](#) - add
11. [WGL_ARB_pbuffer](#) - objects
12. [GL_ARB_texture_compression](#) - core
13. [GL_ARB_texture_border_clamp](#) - core
14. [GL_ARB_point_parameters](#) - program
15. [GL_ARB_vertex_blend](#) - program
16. [GL_ARB_matrix_palette](#) - program
17. [GL_ARB_texture_env_combine](#) - program
18. [GL_ARB_texture_env_crossbar](#) - program
19. [GL_ARB_texture_env_dot3](#) - program
20. [WGL_ARB_render_texture](#) - objects

7.2 Non-ARB extensions by number

1. [GL_EXT_abgr](#) - program
2. [GL_EXT_blend_color](#) - core (imaging subset)
3. [GL_EXT_polygon_offset](#) - core
4. [GL_EXT_texture](#) - core
6. [GL_EXT_texture3D](#) - core
7. [GL_SGIS_texture_filter4](#) - program
9. [GL_EXT_subtexture](#) - core
10. [GL_EXT_copy_texture](#) - core
11. [GL_EXT_histogram](#) - NO
12. [GL_EXT_convolution](#) - program
13. [GL_SGI_color_matrix](#) - program

14. [GL_SGI_color_table](#) - program
15. [GL_SGIS_pixel_texture](#) - program
16. [GL_SGIX_pixel_texture](#) - rogram
16. [GL_SGIS_texture4D](#) - NO
17. [GL_SGI_texture_color_table](#) - program
18. [GL_EXT_cmyka](#) - program
20. [GL_EXT_texture_object](#) - core
21. [GL_SGIS_detail_texture](#) - program
22. [GL_SGIS_sharpen_texture](#) - program
23. [GL_EXT_packed_pixels](#) - core
24. [GL_SGIS_texture_lod](#) - core
25. [GL_SGIS_multisample](#) - core
27. [GL_EXT_rescale_normal](#) - core / program
28. [GLX_EXT_visual_info](#) - core
30. [GL_EXT_vertex_array](#) - core / mem
31. [GL_EXT_misc_attribute](#) - NO
32. [GL_SGIS_generate_mipmap](#) - ?
33. [GL_SGIX_clipmap](#) - program / mem / implementation detail
34. [GL_SGIX_shadow](#) - program
35. [GL_SGIS_texture_edge_clamp](#) - core
36. [GL_SGIS_texture_border_clamp](#) - core
37. [GL_EXT_blend_minmax](#) - core
38. [GL_EXT_blend_subtract](#) - core
39. [GL_EXT_blend_logic_op](#) - core
40. [GLX_SGI_swap_control](#) - async
41. [GLX_SGI_video_sync](#) - async
42. [GLX_SGI_make_current_read](#) - object
43. [GLX_SGIX_video_source](#) - ?
44. [GLX_EXT_visual_rating](#) - ?
45. [GL_SGIX_interlace](#) - program
47. [GLX_EXT_import_context](#) - ?
49. [GLX_SGIX_fbconfig](#) - core / image objects
50. [GLX_SGIX_pbuffer](#) - core / image objects
51. [GL_SGIS_texture_select](#) - program
52. [GL_SGIX_sprite](#) - program
53. [GL_SGIX_texture_multi_buffer](#) - NO
54. [GL_EXT_point_parameters](#) - program
55. [GL_SGIX_instruments](#) - NO
56. [GL_SGIX_texture_scale_bias](#) - program
57. [GL_SGIX_framezoom](#) - NO
58. [GL_SGIX_tag_sample_buffer](#) - NO
60. [GL_SGIX_reference_plane](#) - NO
61. [GL_SGIX_flush_raster](#) - NO
62. [GLX_SGI_cushion](#) - objects?
63. [GL_SGIX_depth_texture](#) - program
64. [GL_SGIS_fog_function](#) - program
65. [GL_SGIX_fog_offset](#) - program
66. [GL_HP_image_transform](#) - program
67. [GL_HP_convolution_border_modes](#) - program
69. [GL_SGIX_texture_add_env](#) - program
74. [GL_EXT_color_subtable](#) - program
75. [GLU_EXT_object_space_tess](#) - NO
76. [GL_PGI_vertex_hints](#) - NO
77. [GL_PGI_misc_hints](#) - NO
78. [GL_EXT_paletted_texture](#) - program
79. [GL_EXT_clip_volume_hint](#) - ?

- 80. [GL SGIX_list_priority](#) - mem
- 81. [GL SGIX_ir_instrument1](#) - NO
- 83. [GLX SGIX_video_resize](#) - ?
- 84. [GL SGIX_texture_lod_bias](#) - ?
- 85. [GLU SGI_filter4_parameters](#) - NO / program ?
- 86. [GLX SGIX_dm_buffer](#) - objects
- 90. [GL SGIX_shadow_ambient](#) - program
- 91. [GLX SGIX_swap_group](#) - async ?
- 92. [GLX SGIX_swap_barrier](#) - async ?
- 93. [GL_EXT_index_texture](#) - program
- 94. [GL_EXT_index_material](#) - program
- 95. [GL_EXT_index_func](#) - program
- 96. [GL_EXT_index_array_formats](#) - program
- 97. [GL_EXT_compiled_vertex_array](#) - mem
- 98. [GL_EXT_cull_vertex](#) - program ?
- 100. [GLU_EXT_nurbs_tessellator](#) - NO
- 101. [GL SGIX_ycrcb](#) - program
- 102. [GL_EXT_fragment_lighting](#) - program
- 110. [GL IBM_rasterpos_clip](#) - NO / program ?
- 111. [GL_HP_texture_lighting](#) - program
- 112. [GL_EXT_draw_range_elements](#) - core
- 113. [GL_WIN_phong_shading](#) - program
- 114. [GL_WIN_specular_fog](#) - program
- 115. [GLX_SGIS_color_range](#) - objects
- 117. [GL_EXT_light_texture](#) - program
- 119. [GL SGIX_blend_alpha_minmax](#) - program
- 120. [GL_EXT_scene_marker](#) - NO
- 127. [GL SGIX_pixel_texture_bits](#) - program
- 129. [GL_EXT_bgra](#) - program
- 132. [GL SGIX_async](#) - async
- 133. [GL SGIX_async_pixel](#) - async
- 134. [GL SGIX_async_histogram](#) - NO
- 135. [GL_INTEL_texture_scissor](#) - program
- 136. [GL_INTEL_parallel_arrays](#) - NO
- 137. [GL_HP_occlusion_test](#) - ?
- 138. [GL_EXT_pixel_transform](#) - program
- 139. [GL_EXT_pixel_transform_color_table](#) - program
- 141. [GL_EXT_shared_texture_palette](#) - program
- 142. [GLX_SGIS_blended_overlay](#) - ?
- 144. [GL_EXT_separate_specular_color](#) - program
- 145. [GL_EXT_secondary_color](#) - program
- 146. [GL_EXT_texture_env](#) - program
- 147. [GL_EXT_texture_perturb_normal](#) - program
- 148. [GL_EXT_multi_draw_arrays](#) - add
- 149. [GL_EXT_fog_coord](#) - program
- 155. [GL_REND_screen_coordinates](#) - program ?
- 156. [GL_EXT_coordinate_frame](#) - program
- 158. [GL_EXT_texture_env_combine](#) - program
- 159. [GL_APPLE_specular_vector](#) - program
- 160. [GL_APPLE_transform_hint](#) - NO
- 163. [GL_SUNX_constant_data](#) - mem / async
- 164. [GL_SUN_global_alpha](#) - program
- 165. [GL_SUN_triangle_list](#) - NO
- 166. [GL_SUN_vertex](#) - program
- 167. [WGL_EXT_display_color_table](#) - NO ?
- 168. [WGL_EXT_extensions_string](#) - ARB version

- 169. [WGL_EXT_make_current_read](#) - ARB
- 170. [WGL_EXT_pixel_format](#) - ARB
- 171. [WGL_EXT_pbuffer](#) - ARB
- 172. [WGL_EXT_swap_control](#) - async
- 173. [GL_EXT_blend_func_separate](#) - program
- 174. [GL_INGR_color_clamp](#) - program
- 175. [GL_INGR_interlace_read](#) - program
- 176. [GL_EXT_stencil_wrap](#) - add
- 177. [WGL_EXT_depth_float](#) - NO
- 178. [GL_EXT_422_pixels](#) - program
- 179. [GL_NV_texgen_reflection](#) - program
- 181. [GL_SGIX_texture_range](#) - program
- 182. [GL_SUN_convolution_border_modes](#) - program
- 183. [GLX_SUN_get_transparent_index](#)
- 185. [GL_EXT_texture_env_add](#) - program
- 186. [GL_EXT_texture_lod_bias](#) - add / program
- 187. [GL_EXT_texture_filter_anisotropic](#) - program
- 188. [GL_EXT_vertex_weighting](#) - program
- 189. [GL_NV_light_max_exponent](#) - program
- 190. [GL_NV_vertex_array_range](#) - mem
- 191. [GL_NV_register_combiners](#) - program
- 192. [GL_NV_fog_distance](#) - program
- 193. [GL_NV_texgen_emboss](#) - program
- 194. [GL_NV_blend_square](#) - program
- 195. [GL_NV_texture_env_combine4](#) - program
- 196. [GL_MESA_resize_buffers](#) - NO
- 197. [GL_MESA_window_pos](#) - add
- 198. [GL_EXT_texture_compression_s3tc](#) - NO
- 199. [GL_IBM_cull_vertex](#) - program ?
- 200. [GL_IBM_multimode_draw_arrays](#) - add ?
- 201. [GL_IBM_vertex_array_lists](#) - add ?
- 206. [GL_3DFX_texture_compression_FXT1](#) - NO
- 207. [GL_3DFX_multisample](#) - NO
- 208. [GL_3DFX_tbuffer](#) - NO
- 209. [WGL_EXT_multisample](#) - core except for multipass
- 210. [GL_SGIX_vertex_preclip](#) - NO
- 212. [GL_SGIX_resample](#) - program
- 214. [GL_SGIS_texture_color_mask](#) - add
- 215. [GLX_MESA_copy_sub_buffer](#) - ?
- 216. [GLX_MESA_pixmap_colormap](#) - ?
- 217. [GLX_MESA_release_buffers](#) - ?
- 218. [GLX_MESA_set_3dfx_mode](#) - ?
- 220. [GL_EXT_texture_env_dot3](#) - program
- 221. [GL_ATI_texture_mirror_once](#) - add ?
- 222. [GL_NV_fence](#) - async
- 223. [GL_IBM_static_data](#) - mem / async
- 224. [GL_IBM_texture_mirrored_repeat](#) - add ?
- 225. [GL_NV_evaluators](#) - ?
- 226. [GL_NV_packed_depth_stencil](#) - program
- 227. [GL_NV_register_combiners2](#) - program
- 228. [GL_NV_texture_compression_vtc](#) - NO
- 229. [GL_NV_texture_rectangle](#) - ?
- 230. [GL_NV_texture_shader](#) - program
- 231. [GL_NV_texture_shader2](#) - program
- 232. [GL_NV_vertex_array_range2](#) - mem
- 233. [GL_NV_vertex_program](#) - program

- 234. [GLX_SGIX_visual_select_group](#) - image objects?
- 235. [GL_SGIX_texture_coordinate_clamp](#) - program
- 237. [GLX_OML_swap_method](#) - add
- 238. [GLX_OML_sync_control](#) - async
- 239. [GL_OML_interlace](#) - program
- 240. [GL_OML_subsample](#) - program
- 241. [GL_OML_resample](#) - program
- 242. [WGL_OML_sync_control](#) - async
- 243. [GL_NV_copy_depth_to_color](#) - image objects?
- 244. [GL_ATI_envmap_bumpmap](#) - program
- 245. [GL_ATI_fragment_shader](#) - program
- 246. [GL_ATI_pn_triangles](#) - NO (tessellation)
- 247. [GL_ATI_vertex_array_object](#) - mem
- 248. [GL_EXT_vertex_shader](#) - program
- 249. [GL_ATI_vertex_streams](#) - program

7.3 Unnumbered extensions

These extensions have not yet been assigned numbers, are still under development, or were abandoned (but are kept in the extension registry for reference purposes).

- [GL_EXT_static_vertex_array](#) - mem
- [GL_EXT_vertex_array_set](#) - mem
- [GL_EXT_vertex_array_setXXX](#) - mem
- [GL_SGIX_fog_texture](#) - program
- [GL_SGIX_fragment_specular_lighting](#) - program
- [NV_multisample_filter_hint](#) (Quincunx) - NO