Using OpenGL Extensions

By design, OpenGL implementors are free to extend OpenGL's basic rendering functionality with new rendering operations. This extensibility was one of OpenGL's original design goals. As a result, scores of OpenGL extensions have been specified and implemented. These extensions provide OpenGL application developers with new rendering features above and beyond the features specified in the official OpenGL standard. OpenGL extensions keep the OpenGL API current with the latest innovations in graphics hardware and rendering algorithms.

This appendix describes the OpenGL extension mechanism. It describes how extensions are used and documented as well as how to use extensions portably in applications. Particular attention is paid to using OpenGL extensions in the Win32 and UNIX environments.

A.1 How OpenGL Extensions are Documented

An OpenGL extension is defined by its specification. These specifications are typically written as standard ASCII text files. OpenGL extension specifications are written by and for OpenGL implementors. A well-written OpenGL specification is documented to the level of detail needed for a hardware designer and/or OpenGL library engineer to implement the extension unambiguously. This means that OpenGL application programmers should not expect an extension's specification to justify fully why the functionality exists or explain how an OpenGL application would go about using it. An OpenGL

extension specification is not a tutorial on how to use the particular extension. Still, being able to read and understand an OpenGL extension specification helps the application programmer fully understand an OpenGL extension's functionality.

A.2 Finding OpenGL Extension Specifications

The latest public OpenGL specifications can be found on the *www.opengl.org* web site. Note that extension specifications are updated from time to time based on reviews and implementation feedback. In the case of certain proprietary OpenGL extensions, it may be necessary to contact the OpenGL vendor that developed the extension for the extension's specification.

A.3 How to Read an OpenGL Extension Specification

When reading an OpenGL extension specification, it helps to be familiar with the original OpenGL specification. The operation of an OpenGL extension is described as additions and changes to the core OpenGL specification. Having a copy of the core OpenGL specification handy is a good idea when reviewing an OpenGL extension.

OpenGL extension specifications consist of multiple sections. There is a common form established by convention that is used by nearly all OpenGL extensions. Often within a specification, the `gl` and `GL` prefixes on routine names and tokens are assumed. The following describes the purpose of the most common sections in the order they normally appear in extension specifications.

Name: Lists the official name of the extension. This name uses underscores instead of spaces between words. The name also begins with a prefix that indicates who developed the extension. This prefix helps to avoid naming conflicts if two independent groups implement a similar extension. It also helps identity who is promoting use of the extension. For example, `SGIS_point_parameters` was an extension proposed by Silicon Graphics. The `SGIS` prefix belongs to Silicon Graphics. SGI uses the `SGIS` prefix to indicate that the extension is specialized and may not be available on all SGI hardware. Other prefixes in use are:

ARB: Extensions officially approved by the OpenGL Architectural Review Board

EXT: Extensions agreed upon by multiple OpenGL vendors

APPLE: Apple Computer

ATI: ATI Technologies

ES: Evans and Sutherland

HP: Hewlett-Packard

IBM: International Business Machines

INTEL: Intel

KTX: Kinetix (maker of 3D Studio Max)

MESA: Brian Paul's freeware portable OpenGL implementation

NV: NVIDIA Corporation

OES: OpenGL ES, OpenGL for Embedded Systems

SGI: Silicon Graphics

SGIS: Silicon Graphics (limited set of machines)

SGIX: Silicon Graphics (experimental)

SUN: Sun Microsystems

WIN: Microsoft

Note that the `SGIS_point_parameters` extension has since been standardized by other OpenGL vendors. Now there is also an `EXT_point_parameters` extension with the same basic functionality as the `SGIS` version. The `EXT` prefix indicates that multiple vendors have agreed to support the extension. Successful OpenGL extensions are often promoted to `EXT` or `ARB` extensions or made an official part of OpenGL in a future revision to the core OpenGL specification. In fact, the point parameters extension was moved into the core in OpenGL 1.4. Almost all of the new functionality in OpenGL 1.1 through 1.5 appeared first as OpenGL extensions.

Name Strings: Name strings are used to indicate that the extension is supported by a given OpenGL implementation. Applications can query the `GL_EXTENSIONS` string with the OpenGL `glGetString` command to determine what extensions are available. OpenGL also supports the idea of window-system-dependent extensions. Core OpenGL extension name strings are generally prefixed with `GL_`, while window-system-dependent extensions are prefixed with `GLX_` for the X Window System or `WGL_` for Win32 based on the platform embedding to which the extension applies. Note that there may be multiple strings if the extension provides both core OpenGL rendering functionality and window-system-dependent functionality.

In the case of the X Window System, support for GLX extensions is indicated by listing the GLX extension name in the string returned by

`glXQueryExtensionsString`. Querying the core OpenGL extension string requires that an OpenGL rendering context be created and made current (calling `glGetString` assumes a current OpenGL context). However, using `glXQueryExtensionString` only requires a connection to an X server. Because the X Window System is client/server based, the OpenGL client library may support different extensions than the OpenGL server. For this reason, it is also possible to query the extensions supported by the client or server individually using `glXQueryClientString` and `glXQueryServerString`, respectively. To actually use most GLX extensions, a GLX extension must be supported by both the OpenGL client and server, but it is possible for an extension to be a pure client-side extension. For this reason, the strings returned by `glXQueryClientString` and `glXQueryServerString` are intended for informational use only. The string returned by `glXQueryExtensionString` is typically an intersection of the extensions supported by both the client and server. This is the string that should be checked before using a GLX extension. There is not a separate mechanism to discover WGL extensions. Instead, WGL extensions are advertised through OpenGL's core extension string, the one returned by `glGetString`.

Version: A source code control revision string to keep track of what version of the specification the given text file represents. It is important to refer to the latest version of the extension specification in case there are any important changes. Normally the version string has the date the extension was last updated.

Number: Each OpenGL extension is assigned a unique number. Silicon Graphics (who owns the OpenGL trademark) allocates these numbers to ensure that OpenGL extensions do not overlap in their usage of enumerants or protocol tokens. This number is only important to extension implementors.

Dependencies: Often an extension specification builds on the functionality of preexisting extensions. This section documents other extensions upon which the specified extension depends. Dependencies indicate that another extension "is required" to support the specified extension or that the specified extension "affects" the specification of another extension. When an extension affects the specification of another extension, the affecting extension is responsible for fully documenting the interactions between the two extensions.

The dependencies section often also indicates which version of the OpenGL core standard the extension specification is based on. Later sections specify the extension based on updates to the relevant section of this particular OpenGL specification. The importance of a given extension to the evolution of OpenGL can be inferred from how many other extensions are listed that depend on or are affected by the given extension.

Overview: The section provides a description, often terse and without justification, for the extension's specified functionality. This section is the closest to describing "what the extension does."

Issues: Often there are issues that need to be resolved in the specification of an extension. This section documents open issues and states the resolution to closed issues. These issues are often things of interest to the extension implementor, but can also help a programmer understand details regarding how the extension really works.

New Procedures and Functions: This section lists the function prototypes for any new procedures and functions the extension adds. The specifications often leave out the `gl` prefix when discussing commands. Also note that the extension's new functions will be suffixed with the same letters used as the prefix for the extension name.

New Tokens: This section lists the tokens (also called enumerants) the extension adds. The commands that accept each set of new enumerants are documented. The integer values of the enumerants are documented here. These values should be added to `<GL/gl.h>`. Keep in mind that specifications often leave out the `GL_` prefix when discussing enumerants. Also note that the extension's new enumerants will be suffixed with the same letters used as the prefix for the extension name.

Additions to Chapter *XX* of the 1.*X* Specification (*XXX*): These sections document how the core OpenGL specification should be amended to add the extension's functionality to the core OpenGL functionality. Note that the exact version of the core OpenGL specification (such as 1.0, 1.1, or 1.2) is documented. The chapters typically amended by an extension specification are:

- Chapter 2, OpenGL Operations
- Chapter 3, Rasterization
- Chapter 4, Per-fragment Operations and the Framebuffer
- Chapter 5, Special Functions
- Chapter 6, State and State Requests
- Appendix A, Invariance

These sections are quite formal. They indicate precisely how the OpenGL specification wording should be amended or changed. Often tables within the specification are amended as well.

Additions to the GLX Specification: If an extension has any window-system-dependent functionality affecting the GLX interface to the X Window System, these issues are documented here.

GLX Protocol: When implementing the extension for the X Window System, if any special X11 extension protocol for the GLX extension is required to support the extension the protocol are documented in this section. This section is only interesting to GLX protocol implementors because the GLX protocol is hidden from application programmers beneath the OpenGL API.

Dependencies on *XXX*: These sections describe how the extension depends on some other extension listed in the *Dependencies* section. Usually the wording says that if the other extension is not supported simply ignore the portion of this extension dealing with the dependent extension's state and functionality.

Errors: If the extension introduces any new error conditions particular to the extension, they are documented here.

New State: Extensions typically add new state variables to OpenGL's state machine. These new variables are documented in this section. The variable's get enumerant, type, get command, initial value, description, section of the specification describing the state variable's function, and attribute group the state belongs to are all documented in tables in this section.

New Implementation-dependent State: Extensions may add implementation-dependent state. These are typically maximum and minimum supported ranges for the extension functionality (such as the widest line size supported by the extension). These values can be queried through OpenGL's `glGet` family of commands.

Backward Compatibility: If the extension supersedes an older extension, issues surrounding backward compatibility with the older extension are documented in this section.

Note that these sections are merely established by convention. While the conventions for OpenGL extension specifications are normally followed, extensions vary in how closely they stick to the conventions. Generally, the more preliminary an extension is the more loosely specified it is. Usually after sufficient review and implementation the specification language and format is improved to provide an unambiguous final specification.

A.3.1 ARB Extensions

The current (as of the OpenGL 1.5 release) set of ARB extensions is as shown here

OpenGL Extensions: `ARB_multitexture`, `ARB_transpose_matrix`, `ARB_multisample`, `ARB_texture_env_add`, `ARB_texture_cube_map`, `ARB_texture_compression`, `ARB_texture_border_clamp`, `ARB_point_parameters`, `ARB_vertex_blend`, `ARB_matrix_palette`, `ARB_texture_env_combine`, `ARB_texture_env_crossbar`, `ARB_texture_env_dot3`, `ARB_texture_mirrored_repeat`, `ARB_depth_texture`, `ARB_shadow`, `ARB_shadow_ambient`, `ARB_window_pos`, `ARB_vertex_program`, `ARB_fragment_program`, `ARB_vertex_buffer_object`, `ARB_occlusion_query`, `ARB_shader_objects`, `ARB_vertex_shader`, `ARB_fragment_shader`, `ARB_shading_language_100`, `ARB_texture_non_power_of_two`, `ARB_point_sprite`

GLX Extensions: ARB_get_proc_address

WGL Extensions: ARB_buffer_region, ARB_extensions_string, ARB_pixel_format, ARB_make_current_read, ARB_pbuffer, ARB_render_texture

A.4 Portable Use of OpenGL Extensions

The advantage of using OpenGL extensions is getting access to cutting-edge rendering functionality so that an application can achieve higher performance and higher quality rendering. OpenGL extensions provide access to the latest features of the newest graphics hardware. The problem with OpenGL extensions is that many OpenGL implementations, particularly older implementations, may not support a given extension. An OpenGL application that uses extensions should be written so that it still works when the extension is not supported. At the very least, the program should report that it requires whatever extension is missing and exit without crashing.

The first step to using OpenGL extensions is to locate the copy of the <GL/gl.h> header file that advertises the API interfaces for the desired extensions. Typically this can be obtained from OpenGL implementation vendor or OpenGL driver vendor as part of a software development kit (SDK). API interface prototypes and macros can also be obtained directly from the extension specifications, but getting the correct <GL/gl.h> from your OpenGL vendor is the preferred way. A version of the <GL/gl.h> header file with all available extensions is also available from www.opengl.org.

Note that the <GL/gl.h> header file sets C preprocessor macros to indicate whether the header advertises the interface of a particular extension or not. For example, the basic <GL/gl.h> supplied with Microsoft Visual C++ 7.0 has a section reading:

```
/* Extensions */
#define GL_EXT_vertex_array             1
#define GL_WIN_swap_hint                1
#define GL_EXT_bgra                     1
#define GL_EXT_paletted_texture         1
#define GL_EXT_clip_disable             1
```

These macros indicate that the header file advertises these five extensions. The EXT_bgra extension makes it possible to read and draw pixels in the Blue, Green, Red, Alpha component order as opposed to OpenGL's standard RGBA color component ordering.[1]

1. The functionality of the EXT_bgra extension was added as part of OpenGL 1.2. The BGRA color component ordering is important because it matches the color component ordering of Win32's GDI 2D API and therefore many PC-based file formats use it.

A program using the EXT_bgra extension should test that the extension is supported at compile time with code like this:

```
#ifdef GL_EXT_bgra
   glDrawPixels(width, height, GL_BGRA_EXT, GL_UNSIGNED_BYTE, pixels);
#endif
```

When GL_EXT_bgra is defined, the GL_BGRA_EXT enumerant will be defined. Note that if the EXT_bgra extension is not supported expect the glDrawPixels line to generate a compiler error because the standard *unextended* OpenGL header does not define the GL_BGRA_EXT enumerant.

Based on the extension name macro definition in <GL/gl.h>, code can be written so that it can optimally compile in the extension functionality if the development environment supports the extension's interfaces. This is not a complete solution, however. Even if the development environment supports the extension's interface at compile time, at runtime the target system where the application executes may not support the extension. In UNIX environments, different systems with different graphics hardware often support different sets of extensions. Likewise, in the Win32 environment different OpenGL-accelerated graphics boards will support different OpenGL extensions because they have different OpenGL drivers. It is not safe to assume that a given extension is supported. Runtime checks should be made to verify that a given extension is supported. Assuming that the application thread is made current to an OpenGL rendering context, the following routine can be used to determine at runtime if the OpenGL implementation really supports a particular extension.

```
#include <GL/gl.h>
#include <string.h>

isExtensionSupported(const char *extension){
   const GLubyte *extensions = NULL;
   const GLubyte *start;
   GLubyte *where, *terminator;

   /* Extension names should not have spaces. */
   where = (GLubyte *) strchr(extension, ' ');
   if (where || *extension == '\0')
      return 0;

   extensions = glGetString(GL_EXTENSIONS);

   /* It takes a bit of care to be fool-proof about parsing the OpenGL
      extensions string. Don't be fooled by substrings, etc. */
```

```
    start = extensions;
    for (;;) {
        where = (GLubyte *) strstr((const char *) start, extension);
        if (!where)
            break;
        terminator = where + strlen(extension);
        if (where == start || *(where - 1) == ' ')
            if (*terminator == ' ' || *terminator == '\0')
                return 1;
        start = terminator;
    }
    return 0;
}
```

The `isExtensionSupported` routine can be used to check if the current OpenGL rendering context supports a given OpenGL extension.[2] To ensure the `EXT_bgra` extension is supported before using it, the application is structured as follows:

```
    /* At context initialization. */
    int hasBGRA = isExtensionSupported("GL_EXT_bgra");

    /* When trying to use EXT_bgra extension. */
#ifdef GL_EXT_bgra
    if (hasBGRA) {
        glDrawPixels(width,height, GL_BGRA_EXT, GL_UNSIGNED_BYTE, pixels);
    } else
#endif
    {
        /* No EXT_bgra so quit (or implement software workaround). */
        fprintf(stderr, "Needs EXT_bgra extension!\n");
        exit(1);
    }
```

Note that if the `EXT_bgra` extension is unavailable at either runtime or compile time this code will detect the lack of `EXT_bgra` support. The code is cumbersome but is necessary for portability. The compile time check can be eliminated if the development environment is well understood and the application will not be compiled with header files that don't support the extensions used by the application. The runtime check is essential in avoiding system or graphics card dependencies in the application.

2. Toolkits such as GLUT include functions to query extension support.

A.5 Using Extension Function Pointers

Many OpenGL implementations support extension commands as if they were core commands. Assuming the OpenGL header file provides the function prototypes and enumerants for the desired extension, the program is simply compiled and linked, presupposing that the extension routines exist. Before calling any extension routines, the program should first check the `GL_EXTENSIONS` string value to verify that the OpenGL extension is supported. If the extension is supported, the code can safely call the extension's routines and use its enumerants. If not supported, the program must avoid using the extension.

This method of using an extension's new routines works because several operating systems today support flexible shared libraries. A shared library delays the binding of a routine name to its executable function until the routine is first called when the application runs. This is known as a *runtime* link instead of a *compile-time* link. A problem occurs when an OpenGL extension routine is called that is not supported by the OpenGL runtime library. The result is a runtime link error that is generally fatal. This is why it is so important to check the `GL_EXTENSIONS` string before using any extension. Once extension support is verified, the program can safely call the extension's routines in full expectation that the system's runtime linker will invoke the extension routine correctly.

Unfortunately, many *open platforms* allow graphics hardware vendors to ship new OpenGL implementations, therefore it is not always possible to provide all of the API interfaces *a priori*. This is often the case for vendor-specific extensions or platforms that support multiple simultaneous graphics adaptors (multiadaptor). One problem is that the platform may include a standard runtime library containing the core entry points. OpenGL implementation vendors may not (and likely should not) replace this library with their own. One reason for not doing this is the multiadaptor scenario where the two graphics accelerators are supplied by two different hardware vendors.

To solve this problem hardware vendors supply additional runtime libraries that provide the hardware-specific functionality. The standard OpenGL library discovers and loads these device-specific libraries as *plugins*. The window system embedding layers also include a command to query a pointer to one or more device-specific extension commands. This mechanism allows an application to retrieve a function pointer for each extension API function. In GLX this function is `glxGetProcAddress` and in WGL `wglGetProcAddress`.[3]

The previous `EXT_bgra` example, showing how to safely detect and use the extension at runtime and compile time, is straightforward. The `EXT_bgra` simply adds two new enumerants (`GL_BGRA_EXT` and `GL_BGR_EXT`) and does not require any new commands. Using an extension that includes new command entry points is more complex on many platforms because the application must first explicitly request the function address from the OpenGL device-specific library before it can call the OpenGL function.

---

3. OpenGL Toolkit libraries such as GLUT include platform-independent wrapper versions of these functions.

We will use the EXT_point_parameters extension to illustrate the process. The EXT_point_parameters extension adds two new OpenGL entry points called glPointParameterfEXT and glPointParameterfvEXT. These routines allow the application to specify the attenuation equation parameters and fade threshold. On the Win32 platform, an OpenGL application cannot simply link with these extension functions. The application must first use the wglGetProcAddress command to find the function address and then call through the returned address to invoke the extension function.

First, declare function prototype *typedefs* that match the extension's entry points. For example:

```
#ifdef _WIN32
typedef void (APIENTRY * PFNGLPOINTPARAMETERFEXTPROC)
             (GLenum pname, GLfloat param);
typedef void (APIENTRY * PFNGLPOINTPARAMETERFVEXTPROC)
             (GLenum pname, const GLfloat *params);
#endif
```

The <GL/gl.h> header file may already have these typedefs declared if the <GL/gl.h> defines the GL_EXT_point_parameters macro. Next declare global variables of the type of these function prototype typedefs like this:

```
PFNGLPOINTPARAMETERFEXTPROC pglPointParameterfEXT;
PFNGLPOINTPARAMETERFVEXTPROC pglPointParameterfvEXT;
```

The names here correspond to the extension's function names. Once wglGetProcAddress is used to assign these function variables to the address of the OpenGL driver's device-specific extension functions, the application can call pglPointParameterfEXT and pglPointParameterfvEXT as if they were normal functions. Pass wglGetProcAddress the name of the extension function as an ASCII string. After verifying that the extension is supported the function pointer variables are initialized as follows:

```
int hasPointParams = isExtensionSupported("GL_EXT_point_parameters");
if (hasPointParams) {
   pglPointParameterfEXT = (PFNGLPOINTPARAMETERFEXTPROC)
      wglGetProcAddress("glPointParameterfEXT");
   pglPointParameterfvEXT = (PFNGLPOINTPARAMETERFVEXTPROC)
      wglGetProcAddress("glPointParameterfvEXT");
}
```

Note that before calling this code there should be a current OpenGL rendering context. With the function variables properly initialized to the extension entry points, the extension

can be used as follows:

```
if (hasPointParams && pglPointParameterfvEXT && pglPointParameterfEXT) {
   static GLfloat quadratic[3] = { 0.25, 0.0, 1/60.0 };
   pglPointParameterfvEXT(GL_DISTANCE_ATTENUATION_EXT, quadratic);
   pglPointParameterfEXT(GL_POINT_FADE_THRESHOLD_SIZE_EXT, 1.0);
}
```

Note that the behavior of `wglGetProcAddress` and `glxGetProcAddress` are subtly different. The function returned by `wglGetProcAddress` is only guaranteed to work for the pixel format type of the OpenGL rendering context that was current when `wglGetProcAddress` was called. If multiple contexts were created for different pixel formats, keeping a single function address in a global variable as shown previously may create problems. The application may need to maintain distinct function addresses on a per-pixel-format basis. The WGL implementation may reference a different function pointer value for each different pixel format. This allows different (heterogenous) device drivers in the multiadaptor scenario to return different device-specific implementations of the function. In contrast, the GLX specification guarantees that the pointer returned will be the same for all contexts. This means that the device-dependent layer and the standard OpenGL library cooperate on routing commands through the correct parts of the device-dependent libraries on a context by context basis. For other window system embedding layers, consult the documentation to determine whether the returned pointers are context-independent.

The requirement of using either `glxGetProcAddress` or `wglGetProcAddress` is cumbersome, but makes applications using extension functionality substantially more portable. Using these coding practices and a portable OpenGL toolkit, such as GLUT, can make the task of developing portable code that works across a wide variety of platforms (UNIX/Linux, Windows, Apple) manageable. The end result is applications that can use state-of-the-art features while maintaining broad compatibility.