AMD – Introduction to OpenGL 3.0

Introduction

OpenGL continues to evolve, growing alongside the hardware that supports it. With the release of the latest version, OpenGL 3.0, there has been some confusion about what it is, what it is not and how to use it. This paper will help clear up the misconceptions around OpenGL 3.0 as well as provide a guide to getting started. Specifically this paper will cover:

- What OpenGL3.0 is
- Where to find the specifications
- What hardware will support GL3
- What happens to non-GL3 apps
- What deprecation in OpenGL means
- How to setup a GL3 app
- Common Questions

OpenGL 3.0

OpenGL 3.0 is a new open-standard 3D graphics application programming interface specification that gives applications more control over graphics hardware than ever before. OpenGL 3.0 is derived from previous versions of OpenGL, keeping most of the same interfaces and functionality, but adding several new key features. With OpenGL 3.0 comes an updated version of the GL Shading Language, GLSL 1.30. Discussing the individual changes and new features included in OpenGL 3.0 and GLSL 1.30 is beyond the scope of this document.

Specifications

The OpenGL3.0 API specification and GLSL 1.30 language specification are publicly available on the http://www.opengl.org website. They can also be accessed directly here:

OpenGL 3.0:
http://www.opengl.org/registry/doc/glspec30.20080811.pdf

GLSL 1.30.08:
http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.30.08.pdf

Hardware Support and Drivers

AMD currently supports OpenGL 3.0 and GLSL 1.30 natively on the following graphics cards:

Professional Graphics
- Desktop
    - ATI Fire Pro™ V3700, V3750, V5700, V8700 Series
    - ATI FireGL™ V8600, V7600, V5600, V3600, V7700 Series
    - ATI FireMV™ 2200, 2400 Series
- Mobile
    - ATI Mobility FireGL™ V5600

Consumer Graphics
- Desktop
    - ATI Radeon™ HD4800, HD4600, HD 4500, HD4300 Series Graphics
    - ATI Radeon™ HD3800, HD3600, HD3400 Series Graphics
    - ATI Radeon™ HD2900, HD2600, HD2400 Series Graphics
- Mobile
    - ATI Mobility Radeon™ HD4800, HD4600, HD4500, HD4300 Series Graphics
    - ATI Mobility Radeon™ HD3800, HD3600, HD3400 Series Graphics
    - ATI Mobility Radeon™ HD2400, HD2600 Series Graphics

Driver support is available for ATI Radeon products with the release of the ATI Catalyst™ 9.1 driver set. These can be downloaded from http://ati.amd.com/support/driver.html. AMD is planning on making the official ATI FirePro™ and FireGL™ driver set available in the 8.583 driver package.

In addition to supporting OpenGL 3.0 and GLSL 1.30, these drivers also support this partial list of related extensions:

- GL_ARB_color_buffer_float
- GL_ARB_draw_instanced
- GL_ARB_framebuffer_object
- GL_ARB_half_float_pixel
- GL_ARB_half_float_vertex
- GL_ARB_map_buffer_range
- GL_ARB_texture_float
- GL_ARB_texture_buffer_object
- GL_ARB_vertex_array_object
- GL_EXT_gpu_shader4
- GL_EXT_draw_buffers2

These extensions are added for backwards compatibility with applications that may already use them. New applications should use the core versions of these features found in OpenGL 3.0 instead of the extensions where applicable.

Pre-OpenGL 3.0 Applications

To use OpenGL 3.0, an application has to specifically request a context that supports OpenGL 3.0. If an app does not explicitly request an OpenGL 3.0 context, it will get an OpenGL 2.1 context. Any applications that are already shipping will continue to get the OpenGL 2.1 context as they always have. Furthermore, these applications will continue to run, without change on the OpenGL 2.1 context on future hardware, even after releases of future versions of OpenGL.

Any currently shipping applications will continue to function with drivers that support OpenGL 3.0 as they did with previous drivers.

Furthermore, AMD has no intention of removing support for the OpenGL 2.1 API/context in drivers that support OpenGL 3.0.

OpenGL 3.0 and Deprecation

OpenGL 3.0 added a deprecation model that allows future versions of the API to remove older interfaces and/or features. This means any items deprecated in a given version of the specification are marked for removal in a future version. No features are removed from OpenGL 3.0, just marked for future removal.

In almost every case, deprecated functionality exists in a different, more efficient, and more and forward-looking form. In this sense, deprecating older items does not substantially limit what can be done with the OpenGL API. The purpose of deprecation is to streamline the API by removing duplication and features that are not implemented in hardware, providing applications with an API that better represents modern hardware. An additional benefit from a reduced API is the possibility of enhanced performance due to a reduction in required driver state validation and overhead.

OpenGL 3.0 deprecates several large portions of the OpenGL 2.1 specification. This means they may be removed from future versions of the OpenGL specification. All of the deprecated items are listed in the OpenGL and GLSL specifications linked in the above "Specifications" section. Some of these deprecated features are:

- Fixed-function rendering
- Immediate mode
- Display lists
- Imaging subset
- Indexed color mode

To work with a context that removes the deprecated items, developers can create a forward compatible context (demonstrated later). This allows developers to test applications intended to be GL3.0+ compatible against what future versions of OpenGL may look like, at least with respect to items removed from the specification.

AMD encourages developers to consider using OpenGL 3.0 and the streamlined feature set for all of the reasons stated above. However, an extension will be provided by the ARB to allow for the support of all deprecated and removed features from future versions of OpenGL. AMD currently has no plans to remove any of these deprecated and removed items from non-forward compatible OpenGL contexts. AMD plans to support the features and interfaces currently in widespread use among many code-bases.

Getting started using OpenGL 3.0

There are several contexts that can be created with the updated create context interfaces. First, applications are now able to request a specific GL context version. Applications can also request a debug version of these contexts by setting an additional bit. The meaning of a debug context has not been defined by the ARB at this time, but is intended to provide users more feedback via logging and additional runtime checks. A second variation for context creation is to request a forward looking context. This type of context would remove all items deprecated in the version of the API requested. For instance, requesting a forward-compatible 3.0 context would result in a context will all items deprecated in OpenGL 3.0 actually removed.

What does each of these context choices give you?

Creating a standard 2.1 context gives you:

- The same functionality you've been using
- GLSL defaults to version 1.10, version 1.20 is available

Creating a standard 3.0 context gives you:

- The same functionality you've been using
- New extended functionality added into 3.0
- GLSL defaults to version 1.10, but version 1.20 and 1.30 are supported

Creating a standard 3.0 forward-compatible context gives you:

- Reduced set of non-deprecated entry-points and functionality
- New extended functionality added into 3.0
- GLSL must use version 1.30, (use #version 130 in shader source)

Creating an OpenGL 3.0 Context

To create an OpenGL 3.0 context, applications must use the new wglCreateContextAttribs and glXCreateContextAttribs interfaces. A simple example of setting up an OpenGL 3.0 application is shown below using the wgl interface.

First, create and setup a window as usual.

Then create an OpenGL context using the old interface:

```
g_hRC0 = wglCreateContext( g_hDC );
wglMakeCurrent( g_hDC, g_hRC0 );
```

Next, check the list of wgl extensions to see if the new create context method is supported. If it is in the extension string, get the function pointer.

```
const char *extensions = wglGetExtensionsStringARB(g_hDC);

if(strstr(extensions, "WGL_ARB_create_context") == NULL)
{
    // Missing support for wglCreateContextAttribsARB,
    // does not support GL3
    return;
}
wglCreateContextAttribsARB =(PFNWGLCREATECONTEXTATTRIBSARB)
    wglGetProcAddress("wglCreateContextAttribsARB");

if (wglCreateContextAttribsARB == NULL)
{
    printf("wglCreateContextAttribsARB is NOT supported .\n");
    return;
}
```

Now setup attributes to specify the version of OpenGL to be created.

```
int  attribs[5] = {
        WGL_CONTEXT_MAJOR_VERSION_ARB, 3,
        WGL_CONTEXT_MINOR_VERSION_ARB, 0,
        NULL
        };
g_hRC = wglCreateContextAttribsARB(g_hDC, NULL, attribs);
```

Check that a valid context was created. If context creation failed, check the error.

```
if(g_hRC == NULL)
{
    // Context creation failed, find out why
    DWORD error = GetLastError();
    if (error == ERROR_INVALID_VERSION_ARB)
        printf("OpenGL 3.0 contexts are not supported.\n");
    else
        printf("Unknown error creating an OpenGL 3.0 Context.\n");
}
```

If a valid context handle was returned, make it current. Also check the version of the new context.

```
wglMakeCurrent( g_hDC, g_hRC );
```

The old context is no longer needed and can be deleted.

```
wglMakeCurrent(NULL, NULL);
wglDeleteContext(g_hRC0);
```

Now the function pointers for all entry-points can be queried on the correct context. Note that function pointers should only be queried and used on the current context. They are not guaranteed to work on other contexts. Next, the current context version can be queried to validate that an OpenGL 3.0 context was in fact created.

```
// Test GL version
int nMajorVersion = -1;
int nMinorVersion = -1;
glGetIntegerv(GL_MAJOR_VERSION, &nMajorVersion);
glGetIntegerv(GL_MINOR_VERSION, &nMinorVersion);

printf("Reported GL Version %d.%d  [Supported]\n",
        nMajorVersion, nMinorVersion);
// Draw!!!
```

After this, the context can be used as normal with the addition of the new features included in OpenGL 3.0.

To create a forward looking OpenGL 3.0 context, some small additions must be made to the attributes used to create the context.

```
int    attribList[7] = {
                        WGL_CONTEXT_MAJOR_VERSION_ARB,
                        winInfo.majorContextVersion,
                        WGL_CONTEXT_MINOR_VERSION_ARB,
                        winInfo.minorContextVersion,
                        WGL_CONTEXT_FLAGS_ARB,
                        WGL_CONTEXT_FORWARD_COMPATIBLE_BIT_ARB,
                        0,
                        };

g_hRC = wglCreateContextAttribsARB(g_hDC, NULL, attribs);
```

The context created with the above attributes will be an OpenGL 3.0 context with all of the deprecated items removed. Developers can use this in preparation for future versions of OpenGL, such as OpenGL 3.1.

To get more information on the details of using the new wglCreateContextAttribsARB interface, refer to the WGL_ARB_create_context extension at http://opengl.org/registry/ .

The method outlined above is nearly exactly the same for Linux® using GLX, except that GLX has its own interface for creating contexts; glXCreateContextAttribsARB. This interface takes different parameters from the WGL version just as also glXCreateNewContext does. Refer to the GLX_ARB_create_context extension at http://opengl.org/registry/ for more information.

Sharing Contexts

The second attribute of the wglCreateContextAttribsARB is used to specify a second context to share data with. Doing so at context creation is highly preferable to sharing after contexts are created, eliminating the chance of any internal data conflicts. This makes calling wglShareLists unnecessary.

Querying extensions in OpenGL 3.0

Starting with OpenGL 3.0, a new extension query mechanism is available. The new method involves returning each extension name individually when queried by index. The old method returned a single, very long string that was not designed to handle the number of extensions in the modern driver. It is strongly suggested that glGetStringi be used to query for extension existence. The ability to call glGetString with the GL_EXTENSIONS token has been deprecated.

To determine what extensions are exposed by a given GL driver, you must first query how many extensions are in the list:

```
GLint numExts;
glGetIntegerv(GL_NUM_EXTENSIONS, &numExts);
```

Then you can query each extension name individually:

```
// User passes desired extension they are querying
for (int curExt = 0; curExt < numExts; curExt++)
{
    printf("%s\n", (const char*)glGetStringi(GL_EXTENSIONS, curExt));
}
```

To find out if a particular extension name is supported:

```
for (int curExt = 0; curExt < numExts; curExt++)
{
    if (!strcmp(name, (const char*)glGetStringi(GL_EXTENSIONS, curExt)))
    {
        // Extension exists
    }
}
```

Common Questions

1. **Do I have access to all of the new features in OpenGL 3.0 if I don't use a forward compatible context?**
   Yes, all new GL3.0 features are present in the regular OpenGL 3.0 context. It is not necessary to create a forward looking context to gain access. But it is important to note that direct interaction between many new OpenGL 3.0 features and deprecated items is not defined, even though both are available.

2. **How can I use GLSL 1.30?**
   GLSL 1.30 is available in any version of the OpenGL 3.0 contexts. But you must specify the correct version define in shaders. (Adding the line #version 130 first in the shader source)

3. **What happens to past versions of GLSL in OpenGL 3.0?**
   In the regular context, an application can use any of GLSL versions 1.10, 1.20 or 1.30. However, GLSL versions 1.10 and 1.20 are removed in the OpenGL 3.0 forward looking context. This means only GLSL version 1.30 is available in a context created with the WGL_CONTEXT_FORWARD_COMPAATIBLE_BIT_ARB set.

4. **Can I use multiple versions of GLSL in the same program?**
   AMD suggests that shaders using GLSL version 1.30 not be mixed with GLSL 1.20 and 1.10 version shaders.

5. **Why is the list of extensions you mention as part of OpenGL 3.0 different than the OpenGL 3.0 spec?**
   The spec intended to give credit to the extensions which introduced any given set of functionality. Because of this, the list shows extensions which actually aren't a required set of OpenGL 3.0.