OpenGL PIPELINE
THE OFFICIAL ARB NEWSLETTER
Issue 003
Q1 2007
Full PDF print edition

In This Issue:

Climbing OpenGL Longs Peak – An OpenGL ARB Progress Update

Longs Peak – 14,255 feet, 15th highest mountain in Colorado. Mount Evans is the 14th highest mountain in Colorado. (Therefore, we have at least 13 OpenGL revisions to go!)

As you might know, the ARB is planning a lot for 2007. We're hard at work on not one, but two, OpenGL specification revisions code named "OpenGL Longs Peak" and "OpenGL Mount Evans." If you're not familiar with these names, please look at the last edition of the OpenGL Pipeline for an overview. Besides two OpenGL revisions and conformance tests for these, the ARB is also working on an OpenGL SDK, which I am very excited about. This SDK should become a valuable resource for you, our developer community. You can find more about the SDK in the Ecosystem TSG update in this issue.

OpenGL Longs Peak will bring a new object model, which was described in some detail in the last OpenGL Pipeline. Since that last update, we made some important decisions that I would like to mention here:

» **Object creation is asynchronous.** This means that the call you make to create an object can return to the caller before the object is actually created by the OpenGL implementation. When it returns to the caller, it returns a handle to this still to be created object. The cool thing is that this handle is a valid handle; you can use it immediately if needed. What this provides is the ability for the application and the OpenGL implementation to overlap work, increasing parallelism which is a good thing. For example, consider an application that knows it needs to use a new texture object for the next character it will render on the screen. During the rendering of the current character the application issues a create texture object call, stores away the handle to the texture object, and continues issuing rendering commands to finish the current character. By the time the application is ready to render the next character, the OpenGL implementation has created the texture object, and there is no delay in rendering.

» **Multiple program objects can be bound.** In OpenGL 2.1 only one program object can be in use (bound) for rendering. If the application wants to replace both the vertex and fragment stage of the rendering pipeline with its own shaders, it needs to incorporate all shaders in that single program object. This is a fine model when there are only two programmable stages, but it starts to break down when the number of programmable stages increases because the number of possible combinations of stages, and therefore the number of program objects, increases. In OpenGL Longs Peak it will be possible to bind multiple program objects to be used for rendering. Each program object can contain only the shaders that make up a single programmable stage; either the vertex, geometry or fragment stage. However, it is still possible to create a program object that contains the shaders for more than one programmable stage.

» **The distinction between unformatted/transparent buffer objects and formatted/opaque texture objects begins to blur.** OpenGL Longs peak introduces the notion of image buffers. An image buffer holds the data part (texels) of a texture. (A filter object holds the state describing how to operate on the image object, such as filter mode, wrap modes, etc.) An image buffer is nothing more than a buffer object, which we all know from OpenGL 2.1, coupled with a format to describe the data. In other words, an image object is a formatted buffer object and is treated as a subclass of buffer objects.

» **A shader writer can group a set of uniform variables into a common block.** The storage for the uniform variables in a common block is provided by a buffer object. The application will have to bind a buffer object to the program object to provide that storage. This provides several benefits. First, the available uniform storage will be greatly increased. Second, it provides a method to swap sets of uniforms with one API call. Third, it allows for sharing of uniform values among multiple program objects by binding the same buffer object to different program objects, each with the same common block definition. This is also referred to as "environment uniforms," something that in OpenGL 2.1 and GLSL 1.20 is only possible by loading values into built-in state variables such as the gl_ModelViewMatrix.

We are currently debating how we want to provide interoperability between OpenGL 2.1 and OpenGL Longs Peak. Our latest thinking on this is as follows. There will be a new context creation call to indicate if you want an OpenGL 2.1 context or an OpenGL Longs Peak context. An application can create both types of contexts if desired. Both OpenGL 2.1 and Longs Peak contexts can be made current to the same drawable. This is a key feature, and allows an application that has an OpenGL 2.1 (or earlier) rendering pipeline to open a Longs Peak context, use that context to draw an effect only possible with Longs Peak, but render it into the same drawable as its other context(s). To further aid in this, there will be an extension to OpenGL 2.1 that lets an application attach an image object from a Longs Peak context to a texture object created in an OpenGL 2.1 context. This image object becomes the storage for the texture object. The texture object can be attached to a FBO in the OpenGL 2.1 context, which in turn means an OpenGL 2.1 context can render into a Longs Peak image object. We would like your feedback on this. Is this a reasonable path forward for your existing applications?

The work on OpenGL Mount Evans has also started in earnest. The Next Gen TSG is meeting on a weekly basis to define what this API is going to look like. OpenGL Mount Evans will also bring a host of new features to the OpenGL Shading Language, which keeps the Shading Language TSG extremely busy. You can find more in the Next Gen update article in this issue.

Another area the ARB is working on is conformance tests for OpenGL Longs Peak and Mount Evans. We will be updating the existing set of conformance tests to cover the OpenGL Shading Language and the OpenGL Longs Peak API. Conformance tests ensure a certain level of uniformity among OpenGL implementations, which is a real benefit to developers seeking a write-once, run anywhere experience. Apple is driving the definition of the new conformance tests.

Lastly, a few updates on upcoming trade shows. We will be at the Game Developer Conference in San Francisco on Wednesday March 7, presenting in more detail on OpenGL Longs Peak and other topics. Watch opengl.org for more information on this! As usual we will be organizing a BOF (Birds of a Feather) at Siggraph, which is in San Diego this year. We have requested our usual slot on Wednesday August 8th from 6-8pm, but it has not yet been confirmed. Again, watch opengl.org for an announcement. Hopefully I'll meet you at one of these events!

In the remainder of this issue you'll find an introduction to OpenGL ES, updates from various ARB Technical SubGroups, timely information about OpenGL on Microsoft Vista, and an article covering how-to optimize your OpenGL application using gDEBugger. We will continue to provide quarterly updates of what to expect in OpenGL and of our progress so far.

Barthold Lichtenbelt, NVIDIA
Khronos OpenGL ARB Steering Group chair

Polygons In Your Pocket: Introducing OpenGL ES

If you're a regular reader of OpenGL Pipeline, you probably know that you can use OpenGL on Macs, PCs (under Windows or Linux), and many other platforms ranging from workstations to supercomputers. But, did you know that you can also use it on PDAs and cell phones? Yes, really!

Okay, not really, at least not yet; but you *can* use its smaller sibling, OpenGL ES. OpenGL ES is OpenGL for Embedded Systems, including cell phones in particular, but also PDAs, automotive entertainment centers, portable media players, set-top boxes, and -- who knows -- maybe, someday, wrist watches and Coke® machines.

OpenGL ES is defined by the Khronos Group, a consortium of cell phone manufacturers, silicon vendors, content providers, and graphics companies. Work on the standard began in 2002, with support and encouragement from SGI and the OpenGL ARB. The Khronos Group's mission is to enable high-quality graphics and media on both mobile and desktop platforms. In addition to OpenGL ES, Khronos has defined standards for high-quality vector graphics, audio, streaming media, and graphics asset interchange. In 2006, the OpenGL ARB itself became a Khronos working group, and many ARB members are active in OpenGL ES discussions.

Why OpenGL ES?

When the Khronos group began looking for a mobile 3D API, the advantages of OpenGL were obvious: it is powerful, flexible, non-proprietary, and portable to many different OSes and environments. However, just as human evolution left us with tailbones and appendices, OpenGL's evolution has left it with some features whose usefulness for mobile applications is, shall we say, non-obvious: color index mode, stippling, and `GL_POLYGON_SMOOTH` antialiasing, to name a few. In addition, OpenGL often provides several ways to accomplish the same basic goal, and it has some features that have highly specialized uses and/or are expensive to implement. The Khronos OpenGL ES Working Group saw this as an opportunity: by eliminating legacy features, redundant ways of doing things, and features not appropriate for mobile platforms, they could produce an API that provides most of the power of desktop OpenGL in a much smaller package.

Design Guidelines

The Khronos OpenGL ES Working Group based its decisions about which OpenGL features to keep on a few simple guidelines:

If in doubt, leave it out

» Rather than starting with desktop OpenGL and deciding what features to remove, start with a blank piece of paper and decide what features to *include*, and include only those features you really need.

Eliminate redundancy

» If OpenGL provides multiple ways of doing something, include at most one of them. If in doubt, choose the most efficient.

"When was the last time you used this?"

» If in doubt about whether to include a particular feature, look for examples of recent applications that use it. If you can't find one, you probably don't need it.

The Principle of Least Astonishment

» Try not to surprise the experienced OpenGL programmer; when importing features from OpenGL, don't change anything you don't have to. OpenGL ES 1.0 is defined relative to desktop OpenGL 1.3 – the specification is just a list of what is different, and why. Similarly, OpenGL ES 1.1 is defined as a set of differences from OpenGL 1.5.

OpenGL ES is *almost* a pure subset of desktop OpenGL. However, it has a few features that were added to accommodate the limitations of mobile devices. Handhelds have limited memory, so OpenGL ES allows you to specify vertex coordinates using bytes as well as the usual short, int, and float. Many handhelds have little or no support for floating point arithmetic, so OpenGL ES adds support for fixed point. For really light-weight platforms, OpenGL ES 1.0 and 1.1 define a "light" profile that doesn't use floating point at all.

To Market

OpenGL ES has rapidly replaced proprietary 3D APIs on mobile phones, and is making rapid headway on other platforms. The original version 1.0 is supported in Qualcomm's BREW® environment for cell phones, and (with many extensions) on the PLAYSTATION®3. The current version (ES 1.1) is supported on a wide variety of mobile platforms.

Learning More

In future issues of OpenGL Pipeline, I'll go into more detail about the various OpenGL ES versions and how they differ from their OpenGL counterparts. But for the real scoop, you'll need to look at the specs. As I said earlier, the current OpenGL ES specifications refer to a parent desktop OpenGL spec, listing the differences between the ES version and the parent. This is great if you know the desktop spec well, but it's confusing for the casual reader. For those who prefer a self-contained document, the working group is about to release a stand-alone version of the ES 1.1 specification, and (hurray!) man pages. You can find the current specifications and (soon) other documentation at http://www.khronos.org/opengles/1_X/.

Take It for a Test Drive

OpenGL ES development environments aren't quite as easy to find as OpenGL environments, but if you want to experiment with it, you have several no-cost options. The current list is at http://www.khronos.org/developers/resources/opengles. Many of these toolkits and SDKs target multiple mobile programming environments, and many also offer the option of running on Windows, Linux, or both. There's also an excellent open source implementation, Vincent.

Watch This Space

Since releasing OpenGL ES 1.1 in 2004, the Working Group has been busier than ever. Later this year we'll release OpenGL ES 2.0, which (you guessed it) is based on OpenGL 2.0, and brings modern shader-based graphics to handheld devices. You can expect to see ES 2.0 toolkits later this year, and ES2-capable devices in 2008 and 2009. Also under development or discussion are the aforementioned stand-alone specifications and man pages, educational materials, an effects framework, and (eventually) a mobile version of Longs Peak.

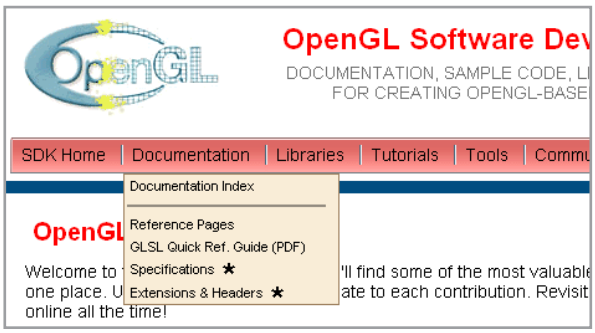
I hope you've enjoyed this quick overview of OpenGL ES. We'd love to have your feedback! Look for us at conferences like SIGGRAPH and GDC, or visit the relevant discussion boards at http://www.khronos.org.

Tom Olson, Texas Instruments
OpenGL ES Working Group Chair

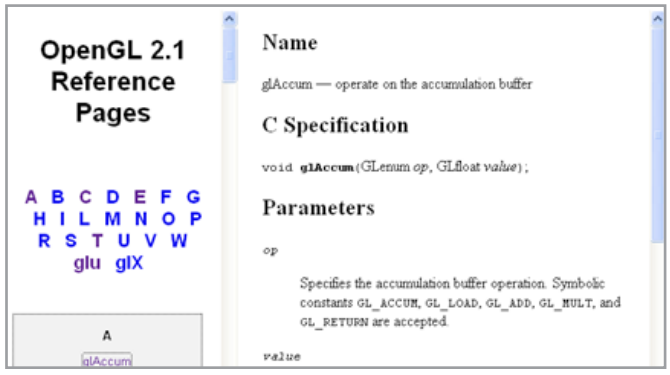
A First Glimpse at the OpenGL SDK

By the time you see this article, the new SDK mentioned in the Autumn edition of OpenGL Pipeline will be public. I will not hide my intentions under layers of pretense; my goal here is to entice you to go check it out. I will try to be subtle.

The SDK is divided into categories. Drop-down menus allow you to navigate directly to individual resources, or you can click on a category heading to visit a page with a more detailed index of what's in there.



The reference pages alone make the SDK a place you'll keep coming back to. If you're like me, reaching for the "blue book" is second nature any time you have a question about an OpenGL command. These same pages can be found in the SDK, only we've taken them beyond OpenGL 1.4. They're now fully updated to reflect the OpenGL 2.1 API! No tree killing, no heavy lifting, just a few clicks to get your answers.



The selection of 3rd-party contributions is slowly growing with a handful of libraries, tools, and tutorials. Surf around, and be sure to check back often. We're just getting started!

BENJ LIPCHAK, AMD
Ecosystem Technical SubGroup Chair

Using the Longs Peak Object Model

In OpenGL Pipeline #002, Barthold Lichtenbelt gave the high-level design goals and structure of the new object model being introduced in OpenGL Longs Peak. In this issue we'll assume you're familiar with that article and proceed to give some examples using the actual API, which has mostly stabilized. (We're not promising the final Longs Peak API will look *exactly* like this, but it should be very close.)

Template and Objects

In traditional OpenGL, objects were created, and their parameters (or "attributes," in our new terminology) set after creation. Calls like `glTexImage2D` set many attributes simultaneously, while object-specific calls like `glTexParameteri` set individual attributes.

In the new object model, many types of objects are immutable (can't be modified), so their attributes must be defined at creation time. We don't know what types of objects will be added to OpenGL in the future, but do know that vendor and ARB extensions are likely to extend the attributes of existing types of objects. Both reasons cause us to want object creation to be flexible and generic, which led to the concept of "attribute objects," or *templates*.

A template is a client-side object which contains exactly the attributes required to define a "real" object in the GL server. Each type of object (buffers, images, programs, shaders, syncs, vertex arrays, and so on) has a corresponding template. When a template is created, it contains default attributes for that type of object. Attribute values in templates can be changed.

Creating a "real" object in the GL server is done by passing a template to a creation function, which returns a *handle* to the new object. In order to avoid pipeline stalls, object creation does not force a round-trip to the server. Instead, the client speculatively returns a handle which may be used in future API calls (if creation fails, future use of that handle will generate a new error, `GL_INVALID_OBJECT`).

Example: Creating an Image Object

In Longs Peak, we are unifying the concepts of "images" -- such as textures, pixel data, and render buffers -- and generic "buffers" such as vertex array buffers. An image object is a type ("subclass," in OOP terminology) of buffer object that adds additional attributes such as format and dimensionality. Here's an example of creating a simple 2D image object. This is not intended to show every aspect of buffer objects, just to show the overall API. For example, the *format* parameter below is assumed to refer to a `GLformat` object describing the internal format of an image.

Once *image* has been created and its contents defined, it can be attached (along with a texture filter object) to a program object, for use by shaders. The contents of the texture, or any rectangular subregion of it, can be changed at any time using `glImageData2D`.

```
// Create an image template
GLtemplate template =    glCreateTemplate(GL_IMAGE_OBJECT);
assert(template != GL_NULL_OBJECT);

// Define image attributes for a 256x256 2D texture image
// with specified internal format
glTemplateAttribt_o(template, GL_FORMAT, format);
glTemplateAttribt_i(template, GL_WIDTH, 256);
glTemplateAttribt_i(template, GL_HEIGHT, 256);
glTemplateAttribt_i(template, GL_TEXTURE, GL_TRUE);

// Create the texture image object
GLbuffer image = glCreateImage(template);

// Define the contents of the texture image
glImageData2D(image,
    0,                  // mipmap level 0
    0, 0,               // copy at offset (0,0) in image
    256, 256,           // copy width & height 256 texels
    GL_RGBA, GL_UNSIGNED_BYTE, // format & type of <data>
    data);              // and the actual texels to use
```

API Design Concerns

You may have some concerns when looking at this example. In particular, it doesn't look very object-oriented, and it appears verbose compared to the classic `glGenTextures() / glBindTexture() / glTexImage2D()` mechanism.

While we do have an object-oriented design underlying Longs Peak, we still have to specify it in a traditional C API. This is mostly because OpenGL is designed as a low-level, cross-platform driver interface, and drivers are rarely written in OO languages; nor do we want to lock ourselves down to any specific OO language. We expect more compact language bindings will quickly emerge for C++, C#, Java, and other object-oriented languages, just as they have for traditional OpenGL. But with the new object model design, it will be easier to create bindings which map naturally between the language semantics and the driver behavior.

Regarding verbosity, the new object model gains us a consistent set of design principles and APIs for manipulating objects. It nails down once and for all issues like object sharing behavior across contexts and object lifetime issues. And it makes defining new object types in the future very easy. But the genericity of the APIs, in particular those required to set the many types of attributes in a template, may look a bit confusing and bloated at first. Consider:

```
glTemplateAttribt_i(template, GL_WIDTH, 256);
```

First, why the naming scheme? What does `t_i` mean? Each attribute in a template has a *name* by which it's identified, and a *value*. Both the name and the value may be of multiple types. This example is among the simplest: the attribute name is `GL_WIDTH` (an enumerated token), and the value is a single integer.

The naming convention we're using is `glTemplateAttrib<name type>_<value type>`. Here, `t` means the type is an enumerated token, and `i` means `GLint`, as always. The attribute name and value are specified consecutively in the parameter list. Using these conventions we can define many other attribute-setting entry points. For example:

`glTemplateAttribt_fv` - name is a token, value is an array of `GLfloat` (`fv`).

`glTemplateAttribti_o` - name is a (token,index) tuple, value is an object handle. This might be used in specifying attachments of buffer objects to new-style vertex array objects, for example, where many different buffers can be attached, each corresponding to an indexed attribute.

Object Creation Wrappers

Great, you say, but all that code just to create a texture? There are two saving graces.

First, templates are *reusable* - you can create an object using a template, then change one or two of the attributes in the template and create another object. So when creating a lot of similar objects, there needn't be lots of templates littering the application.

Second, it's easy to create wrapper functions which look more like traditional OpenGL, so long as you only need to change a particular subset of the attributes using the wrapper. For example, you could have a function which looks a lot like `glTexImage2D`:

```
GLbuffer gluCreateTexBuffer2D(GLint miplevel,
       GLint internalformat, GLsizei width,
       GLsizei height, GLenum format, GLenum type,
       const GLvoid *pixels)
```

This function would create a format object from *internalformat*; create an image template from the format object, *width*, *height*, and *miplevel* (which defaulted to 0 in the original example); create an image object from the template; load the image object using *format*, *type*, and *pixels*; and return a handle to the image object.

We expect to provide some of these wrappers in a new GLU-like utility library - but there is nothing special about such code, and apps can write their own based on their usage patterns.

Any "Object"ions?

We've only had space to do a bare introduction to using the new object model, but have described the basic creation and usage concepts as seen from the developer's point of view. In future issues, we'll delve more deeply into the object hierarchy, semantics of the new object model (including sharing objects across contexts), additional object types, and how everything goes together to assemble the geometry, attributes, buffers, programs, and other objects necessary for drawing.

Jon Leech

ARB Next Gen TSG Update

As noted in the previous edition of OpenGL Pipeline, the OpenGL ARB Working Group has divided up the work for defining the API and feature sets for upcoming versions of OpenGL into two technical sub-groups (TSGs): the "Object Model" TSG and the "Next Gen" TSG. While the Object Model group has the charter to redefine existing OpenGL functionality in terms of the new object model (also described in more detail in the last edition), the Next Gen TSG is responsible for developing the OpenGL APIs for a set of hardware features new to modern GPUs.

The Next Gen TSG began meeting weekly in late November and has begun defining this new feature set, code-named "OpenGL Mount Evans." Several of the features introduced in OpenGL Mount Evans will represent simple extensions to existing functionality such as new texture and render formats, and additions to the OpenGL Shading Language. Other features, however, represent significant new functionality, such as new programmable stages of the traditional OpenGL pipeline and the ability to capture output from the pipeline prior to primitive assembly and rasterization of fragments.

The following section provides a brief summary of the features currently being developed by the Next Gen TSG for inclusion in OpenGL Mount Evans:

» **Geometry Shading** is a powerful, newly added, programmable stage of the OpenGL pipeline that takes place after vertex shading but prior to rasterization. Geometry shaders, which are defined using essentially the same GLSL as vertex and pixel shaders, operate on post-transformed vertices and have access to information about the current primitive, as well as neighboring vertices. In addition, since geometry shaders can generate new vertices and primitives, they can be used to implement higher-order surfaces and other computational techniques that can benefit from this type of "one-input, many-output" processing model.

» **Instanced Rendering** provides a mechanism for the application to efficiently render the same set of vertices multiple times but still differentiate each "instance" of rendering with a unique identifier. The vertex shader can read the instance identifier and perform vertex transformations correlated to this particular instance. Typically, this identifier is used to calculate a "per-instance" model-view transformation.

» **Integer Pipeline Support** has been added to allow full-range integers to flow through the OpenGL pipeline without clamping operations and normalization steps that are based on historical assumptions of normalized floating-point data. New non-normalized integer pixel formats for renderbuffers and textures have also been added, and the GLSL has gained some "integer-aware" operators and built-in functions to allow the shaders to manipulate integer data.

» **Texture "Lookup Table" Samplers** are specialized types of texture samplers that allow a shader to perform index-based, non-filtered lookups into very large one-dimensional arrays of data, considerably larger than the maximum supportable 1D texture.

» **New Uses for Buffer Objects** have been defined to allow the application to use buffer objects to store shader uniforms, textures, and the output from vertex and geometry shaders. Storing uniforms in buffer objects allows for efficient switching between different sets of uniforms without repeatedly sending the changed state from the client to the server. Storing textures in a buffer object, when combined with "lookup table" samplers, provides a very efficient means of sampling large data arrays in a shader. Finally, capturing the output from the vertex or geometry shader in a buffer object offers an incredibly powerful mechanism for processing data with the GPU's programmable execution units without the overhead and complications of rasterization.

» **Texture Arrays** offer an efficient means of texturing from- and rendering to a collection of image buffers, without incurring large amounts of state-changing overhead to select a particular image from that collection.

While the Next Gen TSG has designated the above items as "must have" features for OpenGL Mount Evans, the following list summarizes features that the group has classified as "nice to have":

» **New Pixel and Texture Formats** have been defined to support sRGB color space, shared-exponent and packed floating point, one and two component compression formats, and floating-point depth buffers.

» **Improved Blending Support for DrawBuffers** would allow the application to specify separate blending state and color write masks for each draw buffer.

» **Performance Improvements for glMapBuffer** allow the application to more efficiently control the synchronization between OpenGL and the application when mapping buffer objects for access by the host CPU.

The Next Gen TSG has begun work in earnest developing the specifications for the features listed above, and the group has received a tremendous head start with a much-appreciated initial set of extension proposals from NVIDIA. As mentioned, the Next Gen TSG is developing the OpenGL Mount Evans features to fit into the new object model introduced by OpenGL Longs Peak. Because of these dependencies, the Next Gen TSG has the tentative goal of finishing the Mount Evans feature set specification about 2-3 months after the Object Model TSG completes its work defining Longs Peak.

Please check back in the next edition of OpenGL Pipeline for another status update on the work being done in the Next Gen TSG.

Jeremy Sandmel, Apple
Next Gen Technical SubGroup Chair

OpenGL Shading Language: Center or Centroid? (Or When Shaders Attack!)

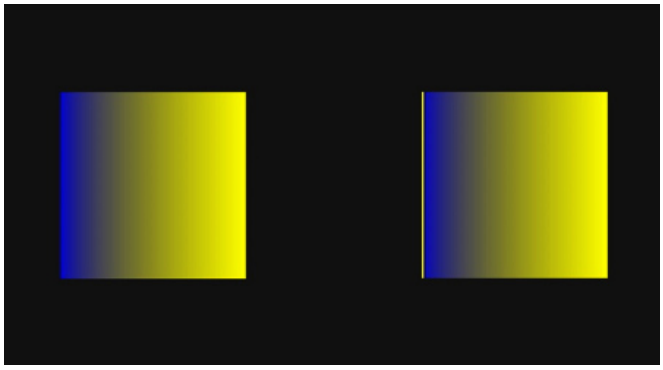


Figure 1 - Correct (on left) versus Incorrect (on right). Note the yellow on the left edge of the "Incorrect" picture. Even though `myMixer` varies between 0.0-1.0, somehow `myMixer` is outside that range on the "Incorrect" picture.

Let's take a look at a simple fragment shader but with a simple non-linear twist:

```
varying float myMixer;
// Interpolate color between blue and yellow.
// Let's do a sqrt for a funkier effect.
void main( void )
{
    const vec3 blue   = vec3( 0.0, 0.0, 1.0 );
    const vec3 yellow = vec3( 1.0, 1.0, 0.0 );
    float a = sqrt( myMixer );
    // udefined when myMixer < 0.0

    vec3 color = mix( blue, yellow, a ); // nonlerp
    gl_FragColor = vec4( color, 1.0 );
}
```

How did the yellow stripe on the "Incorrect" picture get there? To best understand what went wrong, let's first examine the case where it will (almost) always be "Correct." That case is single sample rendering.

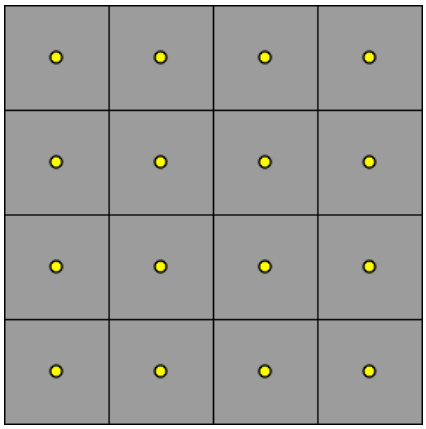


Figure 2 - Squares with yellow dots.

This is classic single sample rasterization. Grey squares represent the pixel square (or a box filter around the pixel center). Yellow dots are the pixel centers at half-integer window coordinate values.

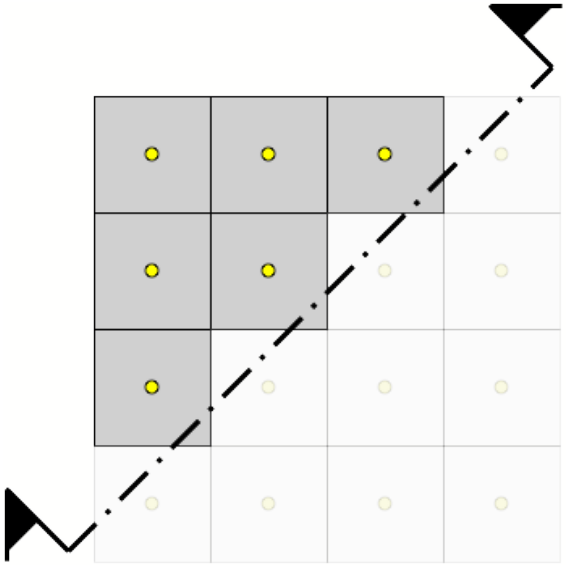


Figure 3 - Squares with yellow dots, halved.

The section line represents a half-space of a primitive. Above and to the left the associated data `myMixer` is positive. Below and to the right it is negative.

In classic single sample rasterization an in/out/on classification at each pixel center will produce a fragment for pixel centers that are "in" the primitive. The six fragments in this example that must be produced are in the upper left. Those pixels that are "out" are dimmed, and will not have fragments generated.

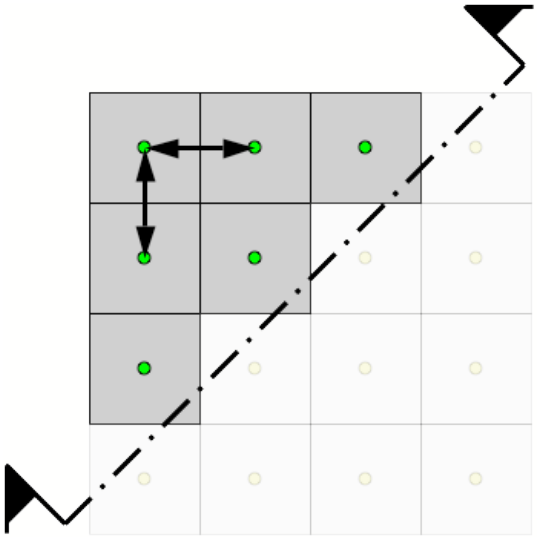


Figure 4 - Squares with green dots halved with arrows.

Green dots show where shading will take place for each of the six fragments. The associated data `myMixer` is evaluated at each pixel center. Note that each of the green dots are above and to the left of the half-space, therefore they are all positive. All of the associated data is interpolated.

While our simple shader uses no derivatives (explicit or implied, such as with mipmapped or anisotropic texture fetches), the arrows represent dFdx (horizontal arrow) and dFdy (vertical arrow). In the interior of the primitive they are quite well defined.

Bottom line: with single sampling, fragments are *only* generated if the pixel center is classified "in," fragment data is evaluated at the pixel center, interpolation only happens within the primitive, and shading only takes place within the primitive. All is good and "Correct." (Almost always. For now we'll ignore some inaccuracies in some of the derivatives on pixels along the edge of the half-space.)

So, all is (almost) well with single sample rasterization. What can go wrong with multi-sample rasterization?

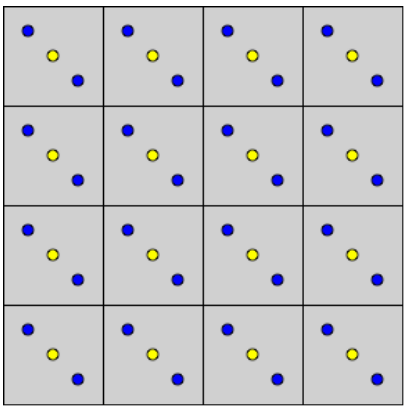


Figure 5 - Squares with yellow and blue dots.

This is classic multi-sample rasterization. Grey squares represent the pixel square (or a box filter around the pixel center). Yellow dots are the pixel centers at half-integer window coordinate values. Blue dots are sample locations. In this example, I'm showing a simple rotated two sample implementation. Everything generalizes to more samples.

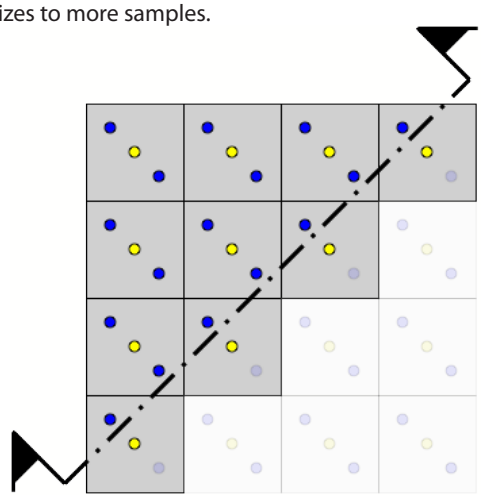


Figure 6 - Squares with yellow and blue dots halved.

The section line again represents a half-space of a primitive. Above and to the left the associated data `myMixer` is positive. Below and to the right it is negative.

In multi-sample rasterization an in/out/on classification at each sample will produce a fragment if *any* sample associated with a pixel is "in" the primitive.

The ten fragments in this example that must be produced are in the upper left. (Note the four additional fragments generated along the half-space. One sample is "in" even though the center is "out.") Those pixels that are "out" are dimmed.

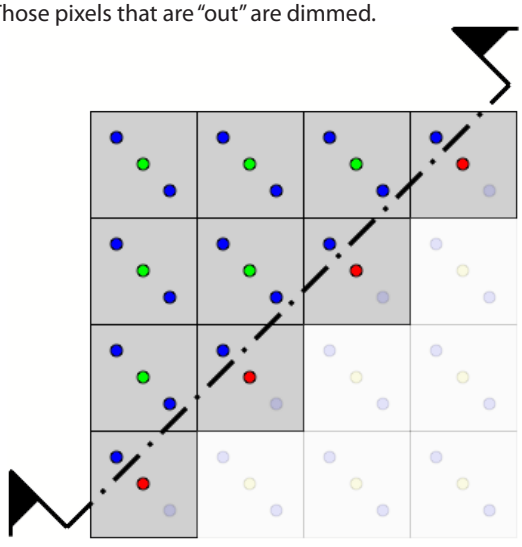


Figure 7 - Squares with yellow blue green and red dots halved.

What if we evaluate at pixel centers?

Green dots *and* red dots show where shading will take place for each of the ten fragments. The associated data `myMixer` is evaluated at each pixel center. Note that each of the green dots are above and to the left of the half-space, therefore they are all positive. But also note that each of the red dots are below and to the right of the half-space, therefore they are negative. The green dots are where the associated data is interpolated, red dots are where they are extrapolated.

In the example shader, `sqrt(myMixer)` is undefined if `myMixer` is negative. Even though the values written by a vertex shader might be in the range 0.0-1.0, due to the extrapolation that might happen, `myMixer` can be outside the range 0.0-1.0. When `myMixer` is negative the result of the fragment shader is undefined!

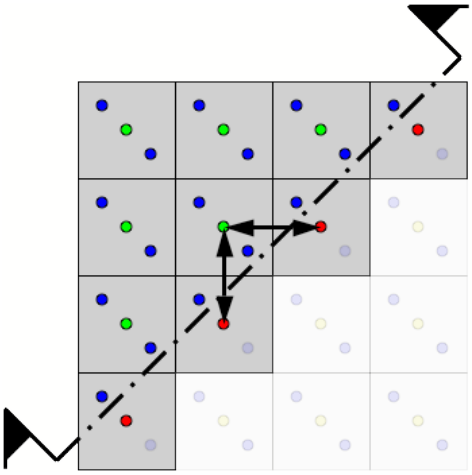


Figure 8 - Squares with yellow blue green and reds dots halved with arrows.

We're still considering the case of evaluation at pixel centers. While our simple shader uses no derivatives, explicit or implied, the arrows represent dFdx (horizontal arrow) and dFdy (vertical arrow). In the interior of the primitive they are quite well defined because all of the evaluation is at the regular pixel centers.

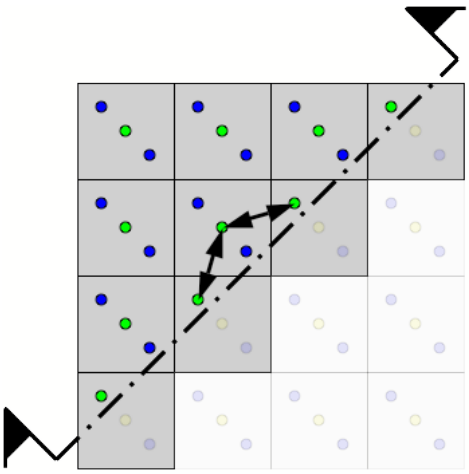


Figure 9 - Squares with yellow blue and green dots halved with arrows.

What if we evaluate other than at pixel centers?

Green dots show where shading will take place for each of the ten fragments. The associated data `myMixer` is evaluated at each pixel "centroid."

The pixel centroid is the center of gravity of the intersection of a pixel square and the interior of the primitive. For a fully covered pixel this is exactly the pixel center. For a partially covered pixel this is a location *other* than the pixel center.

OpenGL allows implementers to choose the ideal centroid, or any location that is inside the intersection of the pixel square and the primitive, such as a sample point or a pixel center.

In this example, if the center is "in," the associated data is evaluated at the center. If the center is "out," the associated data is evaluated at the sample location that is "in." Note that for the four pixels along the half-space the associated data is evaluated at the sample.

Also note that each of the green dots are above and to the left of the half-space. Therefore, they are all positive: always interpolated, never extrapolated!

So why not always evaluate at centroid? In general, it is more expensive than evaluating at center. But that's not the most important factor.

While our simple shader uses no derivatives, the arrows represent dFdx (horizontal arrow) and dFdy (vertical arrow). Note that the spacing between evaluations is not regular. They also do not hold y constant for dFdx, or hold x constant for dFdy. *Derivatives are less accurate when evaluated at centroid!*

Because this is a tradeoff, OpenGL Shading Language Version 1.20 gives the shader writer the choice of when to make the tradeoff with a new qualifier, `centroid`.

```
#version 120
centroid varying float myMixer;
// Interpolate color between blue and yellow.
// Let's do a sqrt for a funkier effect.
void main( void )
{
    const vec3 blue   = vec3( 0.0, 0.0, 1.0 );
    const vec3 yellow = vec3( 1.0, 1.0, 0.0 );
    float a = sqrt( myMixer );
    // undefined when myMixer < 0.0

    vec3 color = mix( blue, yellow, a ); // nonlerp
    gl_FragColor = vec4( color, 1.0 );
}
```

When should you consider using `centroid`?

1. When using an extrapolated value could lead to undefined results. Pay particular attention to the built-in functions that say "results are undefined if!"
2. When using an extrapolated value with a highly non-linear or discontinuous function. This includes for example specular calculations, particularly when the exponent is large, and step functions.

When should you *not* consider using `centroid`?

1. When you need accurate derivatives (explicit or implied, such as with mipmapped or anisotropic texture fetches). The shading language specification considers derivatives derived from centroid varings to be so fraught with inaccuracy that it was resolved they are simply undefined. In such a case, strongly consider at least adding:

```
centroid varying float myMixer;
// beware of derivative
varying float myCenterMixer;
// derivative okay
```

2. With tessellated meshes where most of the quad or triangle boundaries are interior and well defined anyway. The easiest way to think about this case is if you have a triangle strip of 100 triangles, and only the first and last triangle might result in extrapolations, centroid will make those two triangles interpolate but at the tradeoff of making the other 98 triangles a little less regular and accurate.
3. If you know there might be artifacts from undefined, non-linear, or discontinuous functions, but the resulting artifacts are nearly invisible. If the shader is not attacking (much), don't fix it!

Bill Licea-Kane, AMD
Shading Language TSG Chair

OpenGL and Windows Vista™

So Windows Vista is here, but what does it mean for an OpenGL user and developer? In this article we will try to give OpenGL application developers a quick peek at what to expect and the current state of OpenGL on Windows Vista.

Windows Vista supports two primary OpenGL implementations:

1. Hardware manufacturers provide OpenGL ICD (installable client driver) with variable renderer string. The OpenGL version supported depends on the hardware manufacturer.
2. Microsoft's software OpenGL 1.1 implementation (renderer string is GDI Generic), is clustered in higher numbered pixel formats.

Just like Windows XP, Windows Vista does not contain an OpenGL ICD "in the box." End users will need to install drivers from OEMs or video hardware manufacturers in order to access native hardware-accelerated OpenGL. These drivers can be found on the Web sites of most hardware manufacturers.

The two biggest changes that Windows Vista brings to OpenGL are:

1. The new driver model, Windows Display Driver Model (WDDM), formerly known as Longhorn Display Driver Model (LDDM).
2. The new Desktop Window Manager with its Desktop Compositing Engine provides 3D accelerated window composition when Windows Aero is turned on.

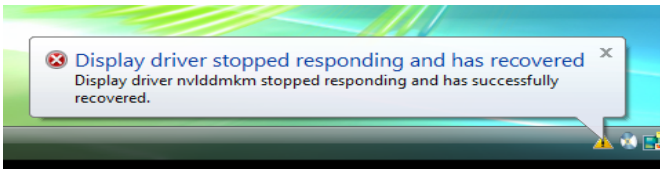
OpenGL and Direct3D are treated the same by Windows Vista, resulting in full integration into the OS for both APIs. For example, both Direct3D and OpenGL will get transparency and dynamic thumbnails when Windows Aero is on, and all the WDDM features (video memory virtualization, etc.) will work in a similar fashion.

Changes Introduced by the New Windows Display Driver Model

Under WDDM, Microsoft takes ownership of the virtualization of video resources at the video memory level, but also at the graphics engine level. In short, this means that multiple simultaneous graphics applications can be running in round robin as scheduled by Windows Vista's Video Scheduler and their working sets (video resources) will be paged in, as needed, by Windows Vista's Video Memory Manager.

Being that the video hardware is virtualized, user-mode components (the OpenGL ICD is one of those) no longer have direct access to that hardware, and need a kernel transition in order to program registers, submit command buffers, or know the real addresses of the video resources in memory.

Because Windows Vista controls the submission of graphic command buffers to the hardware, detecting hangs of the graphics chip due to invalid programming is now possible across the operating system. This is achieved via Windows Vista's Timeout Detection and Recovery (TDR). When a command buffer spends too long in the graphics chip (more than two seconds), the operating system assumes the chip is hung, kills all the graphics contexts, resets the graphics chip and recovers the graphics driver, in order to keep the operating system responsive. The user will then see a popup bubble notifying that the "Display driver stopped responding and has recovered."

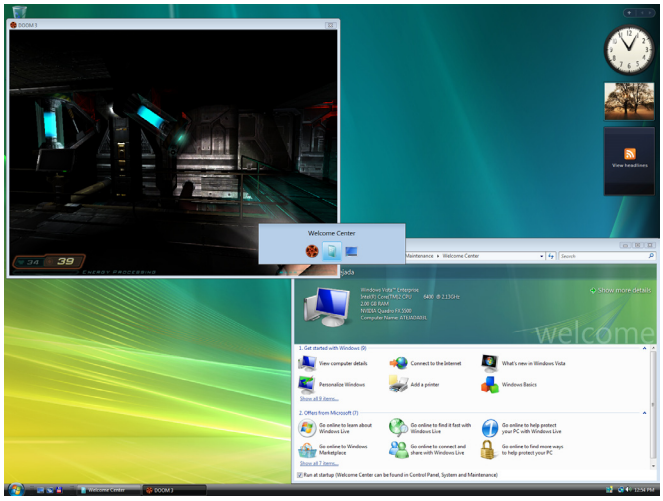


Changes Introduced by the Desktop Window Manager

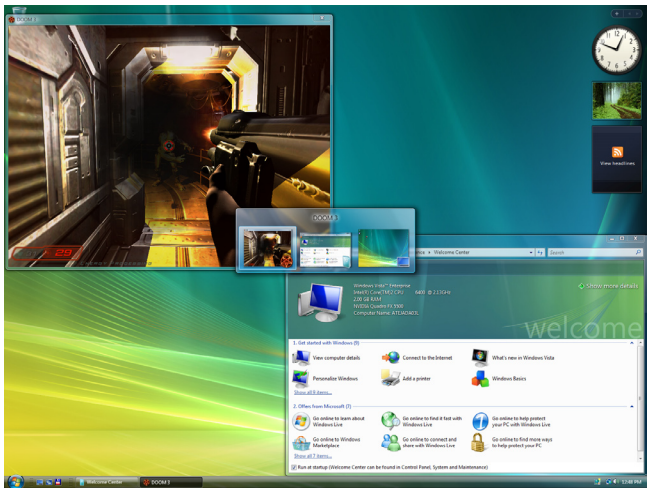
Graphics applications now have to share resources with the 3D-accelerated window manager. Each OpenGL window now requires an offscreen frontbuffer, because there's no longer direct access to the surface being displayed: the desktop. This is also true when the Desktop Windows Manager (DWM) is off.

In order for Windows Vista to perform compositing, DWM allocates an extra window-sized compositing buffer for each top-level window in the system. All these resources add up and increase the video memory footprint.

GDI is no longer hardware-accelerated, but instead rendered to system memory using the CPU. That rendering is later composed on a 3D surface in order to be shown on the desktop. The graphics hardware video driver is no longer involved in GDI rendering, which means that mixing GDI and accelerated 3D rendering in the same window is likely to produce corruption like stale or blanked 3D rendering, trails, etc. Using Microsoft's OpenGL software rendering (the first item in the four OpenGL implementations) will achieve GDI compatibility, but at the expense of rendering speed and lack of modern features.



Windows Vista running with Aero disabled.



Windows Vista running with Aero enabled. Note the semi-transparent windows and the dynamic thumbnails representing the running applications.

What All This Means for the OpenGL ICD User

Software application companies are preparing new versions of their OpenGL applications to take advantage of the new features and fix the possible incompatibilities that Windows Vista may have introduced.

Meanwhile,

» Current Windows XP full screen OpenGL applications are likely to work, although applications that use GDI under the covers (e.g. taking screenshots using Alt+Print Screen, or some enhanced GDI mouse pointers) may not work.

» Carefully written windowed applications should also work. For those which make use of GDI and OpenGL a developer may find that the Desktop Window Manager is disabled when they launch with the message "The color scheme has been changed to Windows Vista Basic." The DWM will be turned on again when the application exits.

» For other windowed applications, if developers observe graphics corruption or lack of rendering refresh, developers may need to disable the DWM manually by switching to the "Windows Vista Basic" theme before starting the application. This also applies to an application's third-party plugins which require GDI interoperability without the application's knowledge. It is possible that some of them will cause corrupted rendering and will require developers to switch off DWM manually.

» Windowed applications that use frontbuffer rendering without ever calling glFlush or glFinish (as they should) are likely to appear completely black, because the rendering will sit forever in the offscreen frontbuffer. Not even switching the DWM off is likely to fix these, given that the offscreen frontbuffer is a requirement of the

driver model itself.

» Windowed stereo rendering will not work.

» Simultaneously using graphics cards from multiple vendors will not work, given that Windows Vista only allows one WDDM driver to be loaded at the same time. Note that multi-card solutions from the same vendor (NVIDIA® SLI™ or AMD™ CrossFire™) should work.

» Memory consumption reduction schemes like Unified Depth/Backbuffer are not possible under the DWM, which increases the memory footprint of the application.

Will My Applications Run Fast?

Performance-wise, developers can expect a decrease of around 10-15% on Windows as compared to Windows XP. Applications that use problematic cases (for example, excessive flushing, or rendering to the frontbuffer, as explained later) can see a larger performance degradation. However, expect this gap to become smaller over time while the graphics hardware vendors work on further optimizing their Windows Vista WDDM drivers.

WDDM's increased memory footprint and new video memory manager approach may worsen resource-hungry scenarios. Applications which were already pushing the limits of memory consumption on Windows XP, just barely fitting, may fall off a performance cliff on Windows Vista. This is due to excessive thrashing because available system and/or video memory is now exhausted.

What All This Means for the OpenGL Developer

GDI compatibility notes

GDI usage over 3D accelerated regions is incompatible with Windows Aero, so developers have two options:

1. Disable Windows Aero
2. Do not use GDI on top of OpenGL rendering.

Windows Vista introduces the new pixelformat flag `FD_SUPPORT_COMPOSITION` (defined in the Driver Development Kit's wingdi.h as `0x00008000`). Creating an OpenGL context for a pixelformat without this flag will disable composition for the duration of the process which created the context. The flag is mutually exclusive with `PFD_SUPPORT_GDI`.

If a developer must use GDI on top of an OpenGL context, use the following rules:

» Create an OpenGL context using a pixelformat with GDI support (`PFD_SUPPORT_GDI` flag set). As this flag is mutually exclusive with `PFD_SUPPORT_COMPOSITION`, this will disable Aero for the lifetime of the current process.

» Don't use `BeginPaint/EndPaint` ouside the `WM_PAINT` message handling.

» As on Windows XP, use the API synchronization calls whenever necessary: `GdiFlush` to synchronize GDI with OpenGL rendering and `glFinish` for the converse.

On the other hand, if a developer wants to have Windows Aero enabled with a windowed OpenGL application, use the following rules to verify that you are not inadvertently trying to mix GDI over OpenGL:

» Create an OpenGL context using a pixelformat with compositing support (`PFD_SUPPORT_COMPOSITION` set).

» Handle the application window's `WM_ERASEBKGND` by returning non-zero in the message handler (this will avoid GDI clearing the OpenGL windows' background).

» Verify that the OpenGL window has the proper clipping styles `WS_CLIPCHILDREN` or `WS_CLIPSIBLINGS`, so GDI rendering of sibling windows in the layout hierarchy is not painted over and vice versa.

» Repaint the application's windows as they are being resized rather than when the final resize happens. This will avoid interacting with GDI's xor drawing of the window border. For example, if the application has splitter bars in a four-viewport application, resize the viewports as the splitter bar is being dragged, otherwise GDI xor rendering over the OpenGL viewport will leave trails.

» Do not use GDI for xor drawing of "rubberbands" or selection highlighting over the OpenGL rendering. Use OpenGL logical operations instead.

» Do not get the desktop's DC and try to paint over it with GDI, as it will corrupt the 3D-accelerated regions.

» Under the DWM's new architecture it is especially important that an application developer verify that the application pairs `GetDC/ReleaseDC` appropriately. The same goes for `LockWindowUpdate` and `LockWindowUpdate(NULL)`.

Performance notes and other recommended practices

If an application renders to the frontbuffer, remember to call `glFinish` or `glFlush` whenever it needs the contents to be made visible on the screen. For the same reason, do not call those two functions too frequently, as they will incur the penalty of copying the contents of the offscreen frontbuffer to the desktop.

Calling `SwapBuffers` on windowed applications incurs two extra copies. One from the backbuffer to the composition surface, and then one from the composition surface to the final desktop.

Calling synchronization routines like `glFlush`, `glFinish`, `SwapBuffers`, or `glReadPixels` (or any command buffer submission in general) now incurs a kernel transition, so use them wisely and sparingly.

Given that under WDDM the OpenGL ICD relinquishes control

over the desktop, fullscreen mode is now achieved by the driver in a similar way to Direct3D's exclusive mode. For that reason do not try to use GDI features on a fullscreen application (e.g. large GDI cursors, doing readbacks via `GetDC/BitBlt`), as they refer to the desktop which resides in a completely different piece of memory than the 3D rendering.

If the application performs extremely GPU intensive and lengthy operations, for example rendering hundreds of fullscreen quads using a complex pixel shader all in a single glDrawElements call, in order to avoid exceeding the 2 second timeout and having an application being killed by Windows Vista's Timeout Detection and Recovery, split the call into chunks and call `glFlush/glFinish` between them. The driver may be able to split long chunks of work for the application, but there will always be corner cases it cannot control, so don't rely solely on the driver to keep rendering from exceeding the two second limit. Instead, anticipate these cases in your application and consider throttling the most intense rendering loads yourself.

Under Windows Vista, the notion of "available video memory" has even less significance than under Windows XP, given that first it is hard for the application to account for the extra footprint needed by the new driver model, and second, the video memory manager may make more memory available to an application on an as-needed-basis.

If your application handles huge datasets, you may find it competing for virtual address space with the video memory manager. In those cases it is recommended that developers move an application to 64-bit or, if not possible, compile them with the `/LARGEADDRESSAWARE` flag and either use a 64-bit OS (which results in 4GB of user address space per process) or boot the 32-bit OS with the `/3GB` flag (which results in 3GB of user address space per process).

Neither of these two solutions is completely trouble-free:

- Compiling for 64-bit has several caveats (e.g. sign extension, extra memory consumption due to larger pointers).
- Compiling `/LARGEADDRESSAWARE` may break applications that assume the high bit of user space addresses will be clear.
- When using `/3GB` a developer may also need to tune `/userva` boot parameter to prevent the kernel from running out of page table entries.

Fun with Windows

Additional References

- DWM interaction with graphics APIs
  http://blogs.msdn.com/greg_schechter/archive/2006/05/02/588934.aspx
- WDDM
  http://msdn2.microsoft.com/en-us/library/aa973510.aspx
- • TDR
  http://www.microsoft.com/whdc/device/display/wddm_timeout.mspx
- `/3GB` and `/userva`
  http://msdn2.microsoft.com/en-us/library/ms791558.aspx
- `/LARGEADDRESSAWARE`
  http://blogs.msdn.com/oldnewthing/archive/2004/08/12/213468.aspx

  http://support.microsoft.com/default.aspx?scid=889654
- 64-bit migration tips
  http://msdn2.microsoft.com/en-us/library/aa384214.aspx

Antonio Tejada, NVIDIA

Optimize Your Application Performance

In the previous article, "Clean your OpenGL usage using gDEBugger," we demonstrated how gDEBugger can help you verify that your application uses OpenGL correctly and calls the OpenGL API commands you expect it to call. This article will discuss the use of ATI and NVIDIA performance counters together with gDEBugger's Performance Views to locate graphics pipeline performance bottlenecks.

Graphics Pipeline Bottlenecks

The graphics system generates images through a pipelined sequence of operations. A pipeline runs only as fast as its slowest stage. The slowest stage is often called the *pipeline bottleneck*. A single graphic primitive (for example, a triangle) has a single graphic pipeline bottleneck. However, the bottleneck may change when rendering a graphic frame that contains multiple primitives. For example, if the application first renders a group of lines and afterwards a group of lit and shaded triangles, we can expect the bottleneck to change.

The OpenGL Pipeline

The OpenGL pipeline is an abstraction of the graphics system pipeline. It contains stages, executed one after the other. Such stages are:

- **Application**: the graphical application, executed on the CPU, calls OpenGL API functions.
- **Driver**: the graphics system driver runs on the CPU and translates OpenGL API calls into actions executed on either the CPU or the GPU.
- **Geometric operations**: the operations required to calculate vertex attributes and position within the rendered 2D image space. This includes: multiplying vertices by the model view and projection matrices, calculating vertex lighting values, executing vertex shaders, etc.
- **Raster operations**: operations operating on fragments / screen pixels: reading and writing color components, reading and writing depth and stencil buffers, performing alpha blending, using textures, executing fragment shaders, etc.
- **Frame buffer**: a memory area holding the rendered 2D image.

Some of the pipeline stages are executed on the CPU; other stages are executed on the GPU. Most operations that are executed on top of the GPU are executed in parallel.

Remove Performance Bottlenecks

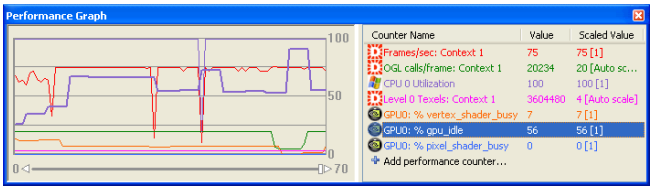
As mentioned in the "Graphics Pipeline Bottlenecks" section, the graphics system runs only as fast as its slowest pipeline stage, which is often called the pipeline bottleneck. The process for removing performance bottlenecks usually involves the following stages:

1. **Identify the bottleneck**: Locate the pipeline stage that is the current graphic pipeline bottleneck.
2. **Optimize**: Reduce the workload done in that pipeline stage until performance stops improving or until you have achieved the desired performance level.
3. **Repeat**: Go back to stage 1.

Notice that after your performance optimizations are done, or after you have reached a bottleneck that you cannot optimize anymore, you can start adding workload to pipeline stages that are not fully utilized without affecting render performance. For example, use more accurate textures, perform more complicated vertex shader operations, etc.

gDEBugger Performance Graph View

gDEBugger Performance Graph view helps you locate your application's graphic pipeline performance bottlenecks; it displays, in real time, graphics system performance metrics. Viewing metrics that measure the workload done in each pipeline stage enables you to estimate the current performance pipeline bottleneck.



Such metrics are: CPU user mode and privilege mode utilizations, graphics driver idle, GPU idle, vertex shader utilization, fragment shader utilization, video memory usage, culled primitives counters, frames per seconds (per render context), number of OpenGL function calls per frame, total size of all loaded textures (in texels) and many other counters.

There is no need to make any changes to your source code or recompile your application. The performance counters will be displayed inside the Performance Graph view.

gDEBugger supports operating system performance counters (Windows and Linux), NVIDIA's performance counters via NVPerfKit, ATI's performance metrics and gDEBugger's internal performance counters. Other IHV's counters will be supported in the future.

gDEBugger Performance Analysis Toolbar

The Performance Analysis toolbar offers commands that enable you to pinpoint application performance bottlenecks by "turning off" graphics pipeline stages. If the performance met-

rics improve while "turning off" a certain stage, you have found a graphics pipeline bottleneck!

These commands include:

 - **Eliminate Draw Commands**: Identify CPU and BUS performance bottlenecks by ignoring all OpenGL command that push vertices or texture data into OpenGL. When ignoring these commands, the CPU and bus workloads remain unchanged, but the GPU workload is almost totally removed, since most GPU activities are triggered by input primitives (triangles, lines, etc).

 - **Eliminate Raster operations:** Identify raster operations bottlenecks by forcing OpenGL to use a 1x1 pixels view port. Raster operations operate per fragment or pixel. By setting a 1x1 pixels view port most raster operations will be eliminated.

 - **Eliminate Fixed Pipeline Lights operations**: Identify "fixed pipeline lights" related calculations bottlenecks. This is done by turning off all OpenGL fixed pipeline lights. Notice that this command does not affect fragment shaders that do not use the fixed pipeline lights.

 - **Eliminate Textures Data Fetch operations**: Identify textures memory performance bottlenecks by forcing OpenGL to use 2x2 pixels stub textures instead of the application defined textures. By using such small stub textures, the texture data fetch operations workload will be almost completely removed.

 - **Eliminate Fragment Shader Operations**: Identify fragment shaders related bottlenecks by forcing OpenGL to use a very simple stub fragment shader instead of the application defined fragment shaders.

The "Combined" Approach

Combining the Performance Analysis toolbar with the Performance Graph view gives an even stronger ability to locate performance bottlenecks. Viewing the way performance metrics vary when disabling graphic pipeline stages can give excellent hints for locating the graphic pipeline performance bottleneck.

For example, an application runs at 20 F/S and has 100% fragment shader utilization and 30% vertex shader utilization. When disabling fragment shader operations, the metrics change to 50 F/S, 2% fragment shader utilization and 90% vertex shader utilization.

The "combined" approach tells us that the current bottleneck is probably the fragment shader operations. It also tells us that if we will optimize and reduce the fragment shader operations workload, the next bottleneck that we will come across will probably be the vertex shader operations.

We hope this article will help you optimize the performance of your OpenGL based applications. In our next article we will talk about OpenGL's debugging model and show how gDEBugger can help you find those "hard to catch" OpenGL-related bugs.

Yaki Tebeka, Graphic Remedy
CTO & Cofounder

Editor's Note: You'll remember from our first edition that Graphic Remedy and the ARB have teamed up to make gDEBugger available free to non-commercial users for a limited time.

OpenGL Pipeline Credits	
Editor	Benj Lipchak, AMD
Web Layout	James Riordon, Khronos Webmaster
Print Layout & Email Distribution	Gold Standard Group
Contributors:	Antonio Tejada, NVIDIA Barthold Lichtenbelt, NVIDIA Benj Lipchak, AMD Bill Licea-Kane, AMD Jeremy Sandmel, Apple Jon Leech, OpenGL Spec Editor Tom Olson, Texas Instruments Yaki Tebeka, Graphic Remedy