In This Issue:

OpenGL Transitions

As of September 21st, 2006, the OpenGL Architecture Review Board (the ARB) has ceased to exist. As described in the previous issue of OpenGL Pipeline, OpenGL API standardization has moved to Khronos, and will take place in the new OpenGL ARB Working Group (WG). We've retained the ARB label as a historical nod, so it makes sense to continue using the "ARB" suffix in extensions the group approves.

After holding the job for nine years, Jon Leech decided to step down as the ARB secretary. Barthold Lichtenbelt has been elected unanimously to lead the ARB. A big thank you to Jon for a job well done!

In general, the ARB WG will operate very much like the old, independent ARB. One great advantage of joining Khronos is closer collaboration with our sister working group, the OpenGL ES working group. This working group develops an API based on OpenGL and the OpenGL Shading Language for the handheld and embedded markets. Before joining Khronos, swapping ideas with the OpenGL ES group was difficult, mainly due to IP concerns. That barrier is now gone. Another huge benefit to joining Khronos is the extra support the ARB gets in the form of professional marketing and clout that comes with being associated with the Khronos name. Procedurally, not much changes. The Khronos bylaws and procedures differ in minor ways; the only major change is a final review and signoff performed by the Khronos Board of Promoters on any specifications the ARB WG develops.

The main task for the ARB is to deliver two new OpenGL releases in 2007. The first one, code named OpenGL "Longs Peak" (the actual releases will have version numbers), is slated to be released in summer 2007. The second one, code named OpenGL "Mt. Evans," is targeted for an October 2007 release. Why code names? We want to give the ARB's marketing group a chance to think through what the right names would be for these releases. Too many suggestions have already been made, ranking from OpenGL 2.2, OpenGL 3.0, OpenGL 3.1 and even OpenGL 4.0. This is not the time yet to pin down the version number, and therefore we'll be using code names.

OpenGL Longs Peak will be a significant departure for us. While there will still be backwards API compatibility, the new "Lean and Mean" profile, and a substantial refactoring in terms of the new object model, make it in many ways an entirely new API design. This is an ambitious task and requires a high degree of commitment from the ARB members. We are already seeing some welcome participation from Khronos members who were not members of the old ARB, and hope to see much more.

While OpenGL Longs Peak will be implementable on current and last generation hardware, OpenGL Mt. Evans will only be implementable on the newest hardware available. The OpenGL Mt. Evans release will be a continuation of OpenGL Longs Peak, with a lot of new functionality added. Some of the highlights are: geometry shading, a more central role for buffer objects, and a full integer pipeline accessible via the OpenGL Shading Language.

Why *two* new OpenGL releases do you ask? This split in two makes it easy for ISVs to develop a new title spanning a wide range of graphics hardware. By coding their core rendering engine to OpenGL Longs Peak, both older and the newest hardware will be covered. By then coding incrementally to the OpenGL Mt. Evans API, the newest hardware can be exploited to its maximum potential.

Lastly, here is more detail about the ARB Working Group organizational structure. The ARB WG contains a top-level Steering Group (SG) and a number of "Technical Sub-Groups" or TSGs. Each TSG focuses on a specific area and has its own chairman. At present, the responsibilities and structure of the ARB WG include:

**OpenGL ARB Steering Group
(Chair: Barthold Lichtenbelt, NVIDIA)**

The top-level SG will define the overall OpenGL strategy and timeline; liaise with the OpenGL ES Working Group; develop conformance tests; and perform any other major functions not assigned to one of the TSGs.

Ecosystem TSG (Chair: Benj Lipchak, AMD)

The Ecosystem TSG will develop the OpenGL SDK, in cooperation with many external projects and developers who are contributing to this effort; write developer documentation; develop naming conventions; define the partitioning between the core OpenGL Longs Peak "Lean and Mean" profile and the compatibility layer for OpenGL 1.x/2.x; support

outside efforts such as the subsidy program for academic users of Graphic Remedy's gDEBugger; and perform some marketing functions, such as this OpenGL Pipeline newsletter.

Object Model TSG (Chair: Barthold Lichtenbelt, NVIDIA)

The Object Model TSG will define the new object model; figure out how existing OpenGL functionality such as framebuffer objects, program objects, texture objects, etc. will be expressed in the new model; and define small bits of entirely new functionality such as sync objects. This is where most of the work for OpenGL Longs Peak will take place.

Platform TSG (Chair: Jon Leech)

The Platform TSG will define the GLX and WGL APIs and GLX stream protocol; liaise with the Khronos OpenKODE Steering Group to provide requirements for the EGL API (a platform-neutral analog of GLX and WGL); and handle topics related to OS integration such as Application Binary Interfaces, Device Driver Interfaces, or reference source code for link libraries and shim layers.

Shading Language TSG (Chair: Bill Licea-Kane, AMD)

The Shading Language TSG will be responsible for all aspects of the OpenGL Shading Language, including new functionality needed in both the Longs Peak and Mt. Evans releases. This TSG will also liaise with OpenGL ES Shading Language work taking place in the Khronos OpenGL ES Working Group.

Next Gen TSG (Chair: Jeremy Sandmel, Apple)

The Next Gen TSG will be responsible for all aspects of the Mt. Evans API design, keeping the new functionality aligned with the new object model being designed by the Object Model TSG.

In the remainder of this issue, there's more information about OpenGL Longs Peak and Mt. Evans, and about activity happening in some of the TSGs. We will continue to provide quarterly updates of what to expect in OpenGL and of our progress so far. But for now it's back to the teleconference salt mines. We have a very aggressive schedule to meet and don't want to disappoint!

JON LEECH
OpenGL ARB Secretary, outgoing

BARTHOLD LICHTENBELT, NVIDIA
Khronos OpenGL ARB Steering Group Chair, incoming

One SDK to Rule Them All

The Ecosystem TSG has been focused on one goal this past quarter: delivering an OpenGL SDK. We intend the SDK to serve as one-stop-shopping for the tools and reference materials that developers desire, nay, *demand* in order to effectively and efficiently target the OpenGL API. In an opengl.org poll conducted earlier this year, you let us know in no uncertain terms that above all else you'd like to see the ARB publish a full-blown SDK. Herein I will describe the various components we're planning to include.

Documentation

We will provide reference documentation for all OpenGL 2.1 core entrypoints. We aim to make this available in a variety of convenient formats, including HTML and PDF. Over time we may also back-fill this reference material to cover ARB extensions.

Aside from the relatively dry reference docs, we'll also feature a number of tutorials which will walk you through the use of OpenGL API features, from your very first basic OpenGL program to the latest and greatest advanced rendering techniques. These tutorials will be contributed by the most respected OpenGL tutorial sites on the web.

Samples

What better way to start your own OpenGL project than by re-using someone else's sample code? And what better way to spark your imagination and see what's possible with OpenGL than by running cutting-edge demos on your own system? This category will contain both samples (with source code), and demos (without source).

Libraries

No need to reinvent the wheel. Others have already abstracted away window system specific issues, initialization of extension entrypoints, loading of images, etc. Make use of the libraries in this category and spend your development time on the exciting new things you're bringing to life with OpenGL.

Tools

There are some impressive tools and utilities out there to make your life easier while developing for OpenGL. Whether you need a debugger, a performance profiler, or just an image compressor, we'll have you covered in this category of the SDK. Some of these tools are commercially available, while others are free.

Conformance tests

The success of OpenGL depends on all implementations behaving the same way. When you write an application on one piece of hardware, you want it to run smoothly on others. This requires the graphics drivers from various vendors to all conform to a single set of rules: the OpenGL specification (which of course is also available as part of the SDK). A suite of tests will be developed to help ensure that compatibility can be maintained and developers won't spend the majority of their precious time writ-

ing vendor-specific code paths to work around incompatibilities.

Does it seem like the ARB is biting off more than it can chew? Absolutely. That's why we're teaming with leading OpenGL companies, open source projects, and individual professionals who will do all the heavy lifting. The SDK will be a clearinghouse for contributions in the above categories which the Ecosystem TSG has deemed worthy of your attention. The ARB will still be on the hook for reference documentation, conformance tests, and other components that we're in a unique position to supply. But wherever possible we'll be leaning on the rest of the community to step up to the plate and help flesh out our SDK – your SDK.

BENJ LIPCHAK, AMD
Ecosystem TSG Chair

The New Object Model

The Object Model TSG has been working diligently on the new object model. Object management in OpenGL is changing considerably. There will soon be a consistent way to create, edit, share and use objects.

The existing object model evolved over time, and did not have a consistent design. It started with display lists in OpenGL 1.0, and then added texture objects in OpenGL 1.1. Unfortunately texture objects were added in a less than optimal way. The combination of `glBindTexture` and `glActiveTexture` makes it hard to follow and debug code, for example. Texture objects can be incomplete, and worse, their format and size can be redefined with a call to `glTexImage1D/2D/3D`. After texture objects, new object types were added with sometimes inconsistent semantics. Furthermore, the existing object model is optimized for object creation, not for runtime performance. Most applications create objects once, but use them often. It makes more sense to ensure that using an object for rendering is lightweight. This is not the case right now.

The goals we are shooting for with the new object model are to:

» achieve maximum runtime performance
» eliminate incomplete objects
» allow for sharing across contexts on a per-object basis
» (partially) eliminate existing bind semantics

I'll explain each of these goals in the following paragraphs.

Maximum runtime performance is influenced by many factors, including the application's ability to use the underlying hardware as efficiently as possible. In the context of the new object model it means that maximum runtime performance is achieved only when overhead in the OpenGL driver is minimized. The new object model reduces driver overhead in several ways:

» The amount of validation the OpenGL implementation needs to perform at draw time will be reduced. In OpenGL today, quite a lot of validation can and will happen at draw time, slowing rendering down.

» Fewer state changes will be needed in the new object model, again improving runtime performance. For example, the number of bind operations, or copies of uniform values, will be reduced.

» The new object model will reduce time spent in the OpenGL driver looking up object handles.

The fact that incomplete objects can exist adds flexibility to OpenGL, but without a clear benefit. They make it harder for the OpenGL implementation to intelligently allocate memory for objects, can result in more validation at rendering time, and can leave developers confused when rendering does not produce expected results. For example, a texture object is built up one mipmap level at a time. The OpenGL implementation has no idea in advance what the total memory requirements will be when an application hands it the first mipmap level. And if the application fails to provide a complete set of mipmap levels when mipmapping is enabled, that incomplete texture will not be used at all while rendering, a pitfall that can be difficult to debug.

Sharing of objects across contexts is an "all or nothing" switch today. Dangerous race conditions can occur when one context is using an object while another context is changing the object's size, for example. Not to mention what happens when deleting an object in one context while it is still in use in another context. This case actually leads to different behavior on different implementations due to ambiguities in the existing spec language.

The existing object model's "bind to edit" and "bind to use" semantics can have complex side effects and in general are confusing. There is no good reason to bind an object into the current rendering state if it just needs to be edited. Binding should only occur when the object is required for rendering.

The new object model is designed to overcome the flaws of the old one. Here are some of the highlights:

» Object creation is atomic. All attributes needed to create an object are passed at creation time. Once created, the OpenGL implementation returns a handle to the new object.

» An object handle is passed as a parameter to an OpenGL command in order to operate on the object. Gone is the bind semantic just to edit an object. Binding is only required when using an object for rendering.

» An attribute can be set at object creation time to indicate if the object is eligible to be shared across contexts. Creation will fail if that type of object cannot be shared. Container objects, such as FBOs, cannot be shared across contexts in order to eliminate associated race-conditions. Container objects are light-weight, so duplicating one in another context is not a great burden.

» All attributes passed at object creation time are immutable. This means that those properties of an object cannot be subsequently changed. Instead, a new object would be created by the application with a different set of attributes. For example, an image object needs to have its size and dimensionality set at creation time.

Once created, it is no longer possible to change the size of an image object. The data stored in the image object is, however, mutable. Compare this to the old object model where both the size and data of a texture object are mutable. A call to `glTexImage` will both resize the texture object and replace the texel data in the object. Separating the object's shape and size from its data in this way has some nice side effects. It removes guesswork by the OpenGL implementation about what the object will look like, so it can make intelligent choices up front about memory allocation. It makes for more efficient sharing across contexts, since it removes dangerous race-conditions with respect to the shape and size.

» It is more efficient for the OpenGL implementation to validate an object at render time, which in turn means higher rendering performance.

» There will be new data types in the form of a per-object class typedef, which will enforce strong type-checking at compile time. This should be a nice aid, and if a compiler warning or error occurs, an early indicator of coding issues.

» An object handle will be the size of the pointer type on the system the OpenGL implementation is running. This allows, but does not mandate, the OpenGL implementation to store a pointer to an internal data structure in the handle. This in turn means that the OpenGL implementation can very quickly resolve a handle being passed, resulting in higher performance. Of course, this can also lead to possible crashes of the application. A debug layer can help find these kind of bugs during development of a title.

» The new object model is intended to be easier for OpenGL application developers to use.

Stay tuned to opengl.org for more information! We have most object types worked out now, and are in the process of writing all this up in spec form. Once all object types have been worked out, we'll run it by you, our developer community.

BARTHOLD LICHTENBELT, NVIDIA
Object Model TSG Chair

Platform TSG Update

The Platform TSG is one of the more specialized subgroups within the new ARB. Our charter is, broadly, to deal with issues specific to the various platforms on which OpenGL is implemented. In practice this means we're responsible for the GLX and WGL APIs that support OpenGL drivers running within the platform native window system, as well as being the liason to EGL, which is specified by the OpenKODE Graphics TSG within Khronos.

We also will look at general OS and driver integration issues such as platform ABIs and Device Driver Interfaces, reference libraries, and so on, although much of this work may be performed primarily outside Khronos. For example, the Linux ABI is being updated within the Linux Standards Base project, although Khronos members are significant contributors to that work.

Currently, we are finishing up GLX protocol to support vertex buffer objects, based on work Ian Romanick of IBM has been doing within Mesa. This will lead to an updated GLX Protocol specification in the relatively near future.

JON LEECH
Platform TSG Interim Chair

"Clean" your OpenGL usage using gDEBugger

Cleaning up your application's OpenGL usage is the first step to optimize your application and gain better performance. In this article we will demonstrate how gDEBugger helps you verify that your application uses OpenGL correctly and calls the OpenGL API commands you expect it to call.

Remove OpenGL Errors

Removing OpenGL errors before starting to optimize your graphics application's performance is an important task. It is important because in most cases, when an OpenGL error occurs, OpenGL ignores the API call that generated the error. If OpenGL ignores actions that you want it to perform, it may reduce your application's robustness and dramatically affect rendering performance.

The OpenGL error mechanism does not tell you the location of the error and therefore it is hard to track GL errors. gDEBugger points you to the exact location of the OpenGL error. It offers two mechanisms for tracking errors:
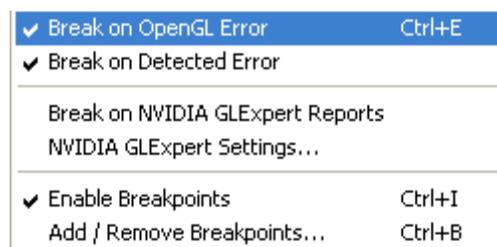
1. Break on OpenGL Errors: Breaks the application run whenever an OpenGL error occurs.

2. Break on Detected Errors: Tells gDEBugger's OpenGL implementation to perform additional error tests that OpenGL drivers do not perform. gDEBugger will also break the application run whenever a detected error occurs.

After the application run is suspended, the gDEBugger System Events view will display the error description. The Call Stack and Source Code viewers will show you the exact location of the error.

Remove Redundant OpenGL Calls

Most OpenGL-based applications generate a lot of redundant OpenGL API calls. Some of these redundant calls may have a significant impact on rendering performance. We offer a two-step solution for locating and removing these redundant calls:

1. Use gDEBugger's OpenGL Function Calls Statistics view to get an overview of the last fully rendered frame. This view displays the number of times each API function call was executed in the previously fully rendered frame. You should look for:

» functions that have a high percentage of the total API function executions.

» functions that are known to reduce rendering performance: `glGet` / `glIs` functions, immediate mode rendering, `glFinish`, etc.

» redundant OpenGL state changes: changing the same state over and over.

» repeatedly turning on and off the same OpenGL mechanisms.

2. Use the Breakpoints dialog to break on each redundant API function call. When the application run breaks, use the OpenGL Function Calls History, Call Stack and Source Code views to view the call stack and source code that led to the redundant API call.

This process should be repeated until the redundant calls that seem to have impact on rendering performance are removed.

Locate Software Fallbacks

When the application calls an OpenGL function or establishes a combination of OpenGL state that is not accelerated by the graphics hardware (GPU), the driver runs these functions on the CPU in "software mode." This causes a significant decrease in rendering performance.

gDEBugger and NVIDIA GLExpert driver integration offers a mechanism that traces software fallbacks. Simply check the "Report Software Fallback Messages" and "Break on GLExpert Reports" check boxes in gDEBugger's NVIDIA GLExpert Settings dialog. In this mode, whenever a "Software Fallback" occurs, gDEBugger will break the debugged application run, letting you view the call stack and source code that led to the software fallback.

We hope this article will help you deliver "cleaner" and faster OpenGL based applications. In the next article we will discuss the use of ATI and NVIDIA Performance Counters together with gDEBugger's Performance Views for finding graphics pipeline performance bottlenecks.

Function Calls Statistics

OpenGL Function Na...	%	# of Calls in Previous Frame
glNormal3fv	33.29	166395
glTexCoord2f	33.29	166395
glVertex3fv	33.29	166395
glTexEnvi	0.04	195
glMaterialfv	0.01	52
glMatrixMode	0.01	35
glMultMatrixd	0.01	26
glPopMatrix	0.00	20
glPushMatrix	0.00	20
glLoadIdentity	0.00	17
glBegin - GL_TRIANGLES	0.00	13
glBindTexture - GL_T...	0.00	13
glColorMaterial	0.00	13
glDisable - GL_COLOR...	0.00	13
glDisable - GL_TEXTU...	0.00	13
glEnable - GL_TEXTU...	0.00	13
glEnd	0.00	13
glMaterialf	0.00	13
glTexParameteri - GL...	0.00	13
glTexParameteri - GL...	0.00	13
glLightfv	0.00	12
glColor3fv	0.00	6
glLightf	0.00	3
glRotatef	0.00	3
glBlendFunc	0.00	2
glDepthFunc	0.00	2
glDisable - GL_LIGHT0	0.00	2
glDisable - GL_BLEND	0.00	2
glEnable - GL_BLEND	0.00	2
glGetBooleanv	0.00	2
glOrtho	0.00	2
glTranslatef	0.00	2
glClear	0.00	1
glDepthMask	0.00	1
glDisable - GL_LIGHT1	0.00	1
glDisable - GL_LIGHT2	0.00	1
glDisable - GL_LIGHT3	0.00	1
glDisable - GL_LIGHTING	0.00	1
glEnable - GL_LIGHTING	0.00	1
glEnable - GL_LIGHT1	0.00	1
glEnable - GL_LIGHT2	0.00	1
glEnable - GL_LIGHT3	0.00	1
glEnable - GL_COLOR...	0.00	1
glFinish	0.00	1
glTranslated	0.00	1
wglMakeCurrent	0.00	1
wglSwapBuffers	0.00	1
Total (47 items)	100	499761

Editor's Note: You'll remember from our last edition (Q3 2006) that Graphic Remedy and the ARB have teamed up to make gDEBugger available free to non-commercial users for a limited time. Here's a link to that article:

http://opengl.org/pipeline/article/vol001_3/

YAKI TEBEKA
CTO & Cofounder, Graphic Remedy
www.gremedy.com