OpenGL PIPELINE
THE OFFICIAL ARB NEWSLETTER
Issue 004
Spring 2007
Full PDF print edition

In This Issue:

Climbing OpenGL Longs Peak, Camp 3 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!)

Since the last edition of *OpenGL Pipeline* we've increased our efforts even more. We held a face-to-face meeting in March and another face-to-face meeting at the end of May. Currently we're on track to meet face-to-face six times this year, instead of the usual four! The ARB recognizes it is extremely important to get OpenGL Longs Peak and Mount Evans done. We also still meet by phone five times per week. This is a big commitment from our members, and I'm very happy and proud to see the graphics industry working together to make OpenGL the best graphics platform!

A lot has happened since the last edition of *Pipeline*. Below follows a brief summary of the most important advances. Other articles in this edition will go into more detail on some of the topics. Happy reading!

Maximize vertex throughput using buffer objects. Just like in OpenGL 2.1, an application can map a buffer object in OpenGL Longs Peak. Mapping a buffer object returns a pointer to the application which can be used to write (or read) data to (or from) the buffer object. In OpenGL Longs Peak the mapping is made more sophisticated, with the end result that maximum parallelism can be achieved between the application writing data into the buffer object and the GL implementation reading data out of it. Read more about this cool feature in an article later in this newsletter.

More context creation options are available. In the previous edition of OpenGL Pipeline I described how we are planning on handling interoperability of OpenGL 2.1 and Longs Peak code. As a result, the application needs to explicitly create an OpenGL Longs Peak or OpenGL 2.x context. To aid debugging, it is also possible to request the GL to create a debug context. A debug context is only intended for use during application development. It provides additional validation, logging and error checking, but possibly at the cost of performance.

The object handle model is fleshed out. We finalized all the nitty-gritty details of the object model that have to do with object and handle creation and deletion, attachment of an object to a container object, and the behavior of these operations across contexts. Here is a brief summary:

1. The GL creates handles, not the application, as can be the case in OpenGL 2.1. This is done in the name of efficiency.
2. Object creation can be asynchronous. This means that it is possible that the creation of an object happens later in time than the creation of the handle to the object. A call to an object creation routine will return the handle to the caller immediately. The GL server might not get to the creation of the actual object until later. This is again done for performance reasons. The rule that all commands are executed in the order issued still applies (within a given context). Thus, asynchronous object creation might mean that a later request to operate on an object will have to block until the object is created. Fences and queries can help determine if this will be the case.
3. Object use by the GL is reference counted. Once the "refcount" of an object goes to zero, the GL implementation is free to delete the storage of the object. Object creation sets the refcount to 1.
4. The application does not delete an object, but instead invalidates the object handle. The invalidation decrements the object's refcount.
5. An object's refcount is incremented whenever it is "in use." Examples of "in use" include attaching an object to a container object, or binding an object into the context.
6. Once a handle is invalidated, it cannot be used to refer to its underlying object anymore, even if the object still exists.

Most context state will be moved into an object. We are currently pondering which state stays in the context, and which context state is moved into an object. One interesting set of state I want to highlight is the state for the per-fragment operations, described in Chapter 4 of the OpenGL 2.1 specification. This state actually applies per sample, not per fragment. Think of state such as alpha test, stencil test, depth test, etc. We expect that some time in the future hardware will be available that makes all these operations programmable. Once that happens, we'll define another program object type, and would like to be able to just "drop it in" to the framework defined in OpenGL Longs Peak. Therefore, we are working on defining a sample operation state object that contains all this state.

We're also working on fleshing out the draw commands as well as display lists. Good progress was made defining what the draw calls will look like. We decided to keep it simple, and largely mirror what is done in OpenGL 2.1. There will be DrawArrays, DrawElements, etc. commands that take vertex indices. In order to actually render, at least a program object, a vertex array object, and an FBO need to be bound to the context. Possibly a sample operation state object, as describe above, will also need to be bound.

You can meet the designers behind OpenGL Longs Peak and Mount Evans at Siggraph 2007 in August. The traditional OpenGL BOF (Birds of a Feather) will likely be on Wednesday evening, August 8th, from 6:00pm – 8:00pm. I hope to see you there!

In the remainder of this issue you'll find an update from the OpenGL ES Working Group, a discussion of Longs Peak buffer object improvements, a look at the Longs Peak object model with source code samples, and an article showing how to use gDEBugger as a window exposing what's happening within the GL.

Barthold Lichtenbelt, NVIDIA
Khronos OpenGL ARB Steering Group chair

Shaders Go Mobile: Announcing OpenGL ES 2.0

It's here at last! At the Game Developers Conference in March, the OpenGL ES Working Group announced the release of OpenGL ES 2.0, the newest version of OpenGL for mobile devices. OpenGL ES 2.0 brings shader-based rendering to cell phones, set-top boxes, and other embedded platforms. The new specification has been three years in the making – work actually started *before* the release of our last major release, OpenGL ES 1.1. What took so long? When we created the ES 1.x specifications, we were using mature technology, following paths that the OpenGL ARB had thoroughly explored in older versions of the desktop API. With OpenGL ES 2.0, we moved closer to the cutting edge, so we had less experience to guide us. But the work is done now. We're very pleased with what we came up with, and excited to have the specification released and silicon on the way. We think you'll agree that it was worth the wait.

A Lean, Mean, Shadin' Machine…

Like its predecessors, OpenGL ES 2.0 is based on a version of desktop OpenGL – in this case, OpenGL 2.0. That means, of course, that it supports vertex and fragment shaders written in a high-level programming language. But almost as interesting as what ES 2.0 has, is what it doesn't have. As I said in the OpenGL ES article in OpenGL Pipeline #3, one of the fundamental design principles of OpenGL ES is to avoid providing multiple ways of achieving the same goal. In OpenGL 2.0 on the desktop, you can do your vertex and fragment processing in shaders *or* you can use traditional fixed-functionality transformation, lighting, and texturing controlled by state-setting commands. You can even mix and match, using the fixed-functionality vertex pipeline with a fragment shader, or vice versa. It's powerful, flexible, and backward compatible; but isn't it, perhaps, a little bit… redundant?

One of the first (and toughest) decisions we made for OpenGL ES 2.0 was to break backward compatibility with ES 1.0 and 1.1. We decided to interpret the "avoid redundancy" rule to mean that *anything that can be done in a shader should be removed from the fixed-functionality pipeline*. That means that transformation, lighting, texturing, and fog calculation have been removed from the API. We even removed alpha test, since you can perform it in a fragment shader using `discard`. Depth test, stencil test, and blending are still there, because you can't perform them in a shader; even if you could read the frame buffer, these operations must be executed per sample, whereas fragment shaders work on fragments.

Living without the fixed-functionality pipeline may seem a little scary, but the advantages are enormous. The API becomes very simple and easy to learn – a handful of state-setting calls, plus a few functions to load and compile shaders. At the same time, the driver gets a lot smaller. An OpenGL 2.0 driver has to do a lot of work to let you switch back and forth smoothly between fixed-functionality and programmable mode, access fixed-functionality state inside your shaders, and so on. Since OpenGL ES 2.0 has no fixed-functionality mode, all of that complexity goes away.

…with Leather Seats, AC, and Cruise Control

OpenGL ES 2.0 lacks the fixed-functionality capability of OpenGL ES 1.x, but don't get the impression that it is a stripped-down, bare-bones API. Along with the shader capability, we've added many other new features that weren't available in ES 1.0 or 1.1. Among them are:

More Complex Vertices

ES 2.0 vertex shaders can declare at least eight general-purpose vec4 attributes, versus the five dedicated vertex arrays of ES 1.1. On the output side, the vertex shader can send at least eight vec4 varyings to the fragment shader.

Texture Features Galore

OpenGL ES 2.0 implementations are guaranteed to provide at least eight texture units, up from two in ES 1.1. Dependent

texturing is supported, as are non-power-of-two texture sizes (with certain limitations). Cube map textures are added as well, because what fun would fragment shaders be without support for environment mapping, global illumination maps, directional lookup tables, and other cool hacks?

Stencil Buffer

All ES 2.0 implementations provide at least one configuration with simultaneous support for stencil and depth buffers.

Frame Buffer Objects

OpenGL ES 2.0 supports a version of the EXT_framebuffer_object extension as a mandatory core feature. This provides (among other things) an elegant way to achieve render-to-texture capabilities.

Blending

OpenGL ES 2.0 extends the options available in the fixed-functionality blending unit, adding support for most of BlendEquation and BlendEquationSeparate.

Options

Along with the ES 2.0 specification, the working group defined a set of options and extensions that are intended to work well with the API. These include ETC1 texture compression (contributed by Ericsson), 3D textures, NPOT mip-maps, and more.

The Shader Language

OpenGL ES 2.0 shaders are written in GLSL ES, a high-level shading language. GLSL ES is very similar to desktop GLSL, and it is possible (with some care, and a few well-placed #ifdefs) to write shader code that will compile under either. We'll go over the differences in detail in a future issue of *OpenGL Pipeline*, and talk about how to write portable code.

Learning More

The ES 2.0 and GLSL ES 1.0 specifications are available for download at http://www.khronos.org/registry/gles/. The API document is a 'difference specification', and should be read in parallel with the desktop OpenGL 2.0 specification, available at http://www.opengl.org/documentation/specs/. The shading language specification is a stand-alone document.

Take it for a test drive

OpenGL ES 2.0 silicon for mobile devices won't be available for a while yet, but you can get a development environment and example programs at http://www.imgtec.com/PowerVR/insider/toolsSDKs/KhronosOpenGLES2xSGX/. This package runs on the desktop under Windows or Linux, using an OpenGL 2.0 capable graphics card to render ES 2.0 content. Other desktop SDKs may well be available by the time you read this, so keep an eye on the Khronos home page and the resource list at http://www.khronos.org/developers/resources/opengles/. If you just want to experiment with the shading language, AMD has announced that GLSL ES will be supported in RenderMonkey 1.7, coming soon.

Tom Olson, Texas Instruments
OpenGL ES Working Group Chair

Longs Peak Update: Buffer Object Improvements

Longs Peak offers a number of enhancements to the buffer object API to help streamline application execution. Applications that are able to leverage these new features may derive a considerable performance benefit. In particular they can boost the performance of applications that have a lot of dynamic data flow in the form of write-once/draw-once streamed batches, procedurally generated geometry, or frequent intra-frame edits to buffer object contents.

Under OpenGL 2.1, there are two ways to transfer data from the application to a buffer object: the `glBufferData`/`glBufferSubData` calls, and the `glMapBuffer`/`glUnmapBuffer` calls. The latter themselves do not transfer any data but instead allow the application temporary access to read and write the contents of a buffer object directly. The Longs Peak enhancements described here are focused on the latter style of usage.

The behavior of `glMapBuffer` is not very complicated under OpenGL 2.1: it will wait until all pending drawing activity using the buffer in question has completed, and it will then return a pointer representing the beginning of the buffer, implicitly granting access to the entire buffer. Once the application has finished reading or writing data in the buffer, `glUnmapBuffer` must be called to return control of the storage to GL. This model is straightforward and easy to code to, but can hold back performance during some usage patterns. The usage patterns of interest are strongly centered on write-only traffic from the application, and the enhancements to the Longs Peak API reflect that.

Longs Peak will allow the application to exercise tighter control over the behavior of `glMapBuffer` (tentatively referred to as `lpMapBuffer`) by offering these new requests:

- mapping only a specified range of a buffer
- strict write-only access
- explicit flushing of altered/written regions
- whole-buffer invalidation
- partial-buffer invalidation
- non-serialized access

An application may benefit from using some or all of the above techniques. They're listed above in roughly increasing order of challenge for the developer to utilize correctly; getting the maximum performance may take more developer work and testing, depending on how application code is structured. Let's look at each of the options in more detail. Each is exposed via an individual bit flag in the access parameter to the `lpMapBuffer` call.

Sub-range mapping of a buffer: Under OpenGL 2.1 it was not possible to request access to a limited section of a buffer object; mapping was an "all or nothing" operation. One side effect of this is that GL has no way to know how much data was changed before unmapping, whether it involves a single range of data or potentially multiple ranges of data. In Longs Peak, by explicitly mapping sub-ranges of a buffer, the application can provide use-

ful information to help accelerate the delivery of those edits to the buffer contents.

For example, if the application maintains a multi-megabyte vertex buffer and wishes to change a few kilobytes of localized data, it can map just the area of interest, write any changes to it, and then unmap. On implementations where altered data ranges must be copied or mirrored to GPU storage, the work at unmap time is thereby reduced significantly.

While in some cases an application may be able to achieve the same partial edit to a large buffer by using `glBufferSubData`, that technique assumes the original data exists in a readily copyable form. This enhancement to the `lpMapBuffer` path allows more efficient partial edits to a buffer object even when the CPU is sourcing the data directly via some algorithm, such as a decompression technique or procedural animation system (particles, physics, etc.). The application can map the range of interest, use the pointer as the target address for the code actually writing the finished data, and then unmap.

Write-only access: While a request of write-only access was possible in GL2, reading from those mappings was discouraged in the spec as likely to be slow or capable of causing a crash. Under Longs Peak this is even more strongly forbidden; reading from a write-only mapping may either crash or return garbage data even if the read succeeds. If there is any need to read from a mapped buffer in a Longs Peak program, you absolutely must request read access in the access parameter to `lpMapBuffer`.

By defining this behavior more strictly we can enhance the notion of one-way data flow from CPU to memory to GPU and free up the driver to do some interesting optimizations, the net effect being that `lpMapBuffer` can return more quickly with a usable pointer for writing when needed. Write-only access is especially powerful in conjunction with one or more of the options described below.

Explicit flushing: In some use cases it can be beneficial for the application to map a range of a buffer representing the "worst case" size needs for the next drawing operation, then write some number of vertices up to that amount, and then unmap. Normally this would imply to GL that all of the data in the mapped range had been changed. But by requesting explicit flushing, the application can undertake the responsibility of informing GL which regions were actually written.

Use of this option requires the application to track precisely which bytes it has written to, and to tell GL where those bytes are prior to unmap through use of the `lpFlushMappedData` API.

For some types of client code where vertices are being generated procedurally, it can be difficult to predict the number of vertices generated precisely in advance. With explicit flush, the application can "reserve" a worst-case-sized region at map time, and then "commit" the portion actually generated through the `lpFlushMappedData` call, prior to unmap.

This ability to convey precisely how much data was written (and where) has a number of positive implications for the driver with respect to any temporary memory management it may need to do in response to the request. While an application can and should use the map-time range information to constrain the amount of storage being manipulated, explicit flushing allows for additional control if that amount cannot be precisely predicted at map time.

This is another case where the same net effect could be accomplished by using a separate temp buffer for the initial data generation, followed by a call to `glBufferSubData`. However, being able to write the finished data directly into the mapped region can eliminate a copying step for the application and also potentially reduce processor cache pollution depending on the implementation.

Whole-buffer invalidation: This is analogous to the `glBufferData(NULL)` idiom from OpenGL 2.1, whereby a new block of uninitialized storage is atomically swapped into the buffer object, but the old storage is detached for the driver to release at a later time after pending drawing operations have completed -- also known as "buffer orphaning." Since Longs Peak no longer allows the `glBufferData(NULL)` idiom, this functionality is now provided as an option to the `lpMapBuffer` call. This is especially useful for implementing efficient streaming of variable sized batches; an application can set up a fixed size buffer object, then repeatedly fill and draw at ascending offsets -- packing as many batches as possible into the buffer -- then perform a full buffer invalidation and start over at offset zero.

Partial-buffer invalidation: This option can and should be invoked when the application knows that none of the data currently stored within the mapped range of a buffer needs to be preserved. That is, the application's intent is to overwrite all or part of that range, and only the newly written data is expected to have any validity upon completion. This option is only usable in conjunction with write-only access mode. It has a number of positive implications for performance, as it releases the driver from the requirement of providing any valid view of the existing storage at map time. Instead it is free to provide scratch memory in order to return a usable pointer to the application more quickly.

Generally speaking, a program can and should make use of both partial and whole buffer invalidation, but the usage frequency of the former is expected to be much higher. Restated, partial invalidation is useful for efficiently accumulating individual batches of CPU-sourced data into a common buffer, whereas whole buffer invalidation should be invoked when one buffer fills up and a fresh batch of storage is needed. Whole buffer invalidation, like `glBufferData(NULL)` in OpenGL 2.1, enables the application to perform these hand-offs without any need for sync objects, fences, or blocking.

Non-serialized access: This option allows an application to assume complete responsibility for scheduling buffer accesses. When this option is engaged, `lpMapBuffer` may not block if there is pending drawing activity on the buffer of interest. Access may be granted without consideration for any such concurrent activity. Another term for this behavior is "non-blocking mapping." If you have written code for OpenGL 2.1 and run into stalls in `glMapBuffer`, this option may be of interest.

When used in conjunction with write-only access and partial invalidation, this option can enable the application to efficiently accumulate any number of edits to a common buffer interleaved with draw calls using those regions, keeping the drawing thread largely unblocked and effectively decoupling CPU progress from GPU progress. On contemporary multi-core-aware implementations where multiple frames' worth of drawing commands may be enqueued at any given moment, the impact of being able to interleave mapped buffer access with drawing requests (without blocking the application) can be quite significant.

An application can only safely use this option if it has taken the necessary steps to ensure that regions of the buffer being used by drawing operations are not altered by the application before those operations complete. This can be accomplished using proper use of sync objects, or by enforcing a write-once policy per region of the buffer. A developer must not set this bit and expect everything to keep working as-is; careful thought must go into analysis of existing access/drawing patterns before proceeding with the use of this technique. The caution level on the part of the developer must be very high, but the potential rewards are also significant.

As the Longs Peak spec is still evolving and minor naming or API changes may yet be made, some of the terminology above could change before the final spec is drafted and released. This article is intended to offer a "sneak peek" at the types of improvements under consideration. Please share your questions and feedback with us on the OpenGL forums at http://www.opengl.org/message_boards.

T. HUNTER
Object Model Technical SubGroup Contributor

Another Object Lessson

"The object of the superior man is truth" -- Confucius

The OpenGL Longs Peak object model is substantially defined now, and we have a good notion of what a Longs Peak program will look like at a high level. Many smaller details are still being filled in, but after reading this article you should understand Longs Peak in considerable detail. For a background refresher, refer to "The New Object Model" in *OpenGL Pipeline* Volume 002, and "Using the Longs Peak Object Model" in *OpenGL Pipeline* Volume 003.

What's In A Namespace?

Or, a GL by any other prefix would smell as sweet.

An important decision is that the OpenGL Longs Peak API will exist in a new namespace. Originally we thought Longs Peak could continue to use "`gl`" prefixed functions, "`GL`" prefixed types, and "`GL_`" prefixed tokens, but as we wrote up object specifications, we realized there were too many collisions. For example, both OpenGL 2.1 and Longs Peak have a `MapBuffer` entry point, but they take different parameters. We haven't chosen the namespace prefix yet; it's a marketing and branding issue, not a technical issue. As a placeholder until that's decided, we're using "`lp`" as the prefix.

The Object Hierarchy

"A mental act is cognitive only in the sense that it takes place in reference to some object, which is said to be known" -- Samuel Alexander

The objects defined in Longs Peak fall into several different categories depending on their behavior and semantics. In a true object-oriented language binding of the API, these categories would be abstract classes from which the concrete classes inherit behavior. Since our C API doesn't support inheritance, the categories are useful primarily as a conceptual tool for understanding the API. In any event, the categories are as follows:

- *Templates* are *client state*, meaning they exist in the client (application) address space. All the other categories are *server state*, existing in the Longs Peak driver address space. Templates are fully *mutable*, meaning that any of their properties can be changed at any time; this makes it easier to reuse them for generating multiple objects. Templates, and the APIs to create and use them, are described more fully in OpenGL Pipeline 003.
- *State Objects* contain a group of closely related attributes defining the behavior of some part of the graphics pipeline. They are fully immutable once created, which allows the driver to pre-cache derived state and otherwise optimize use of these objects, and they may be shared by multiple contexts. State objects are typically small. State object classes described below include *format* objects, *shader* objects, and *texture filter* objects.
- *Data Objects* have an immutable *structure* (organization) defined when they are created, and a fully mutable *data store* filling out that structure. They may be shared by multiple contexts, although there are some remaining issues regarding when changes made in one context to the data store of an object will be visible to another context using the same object. Data object classes described below include *buffer* objects, *image* objects, and several types of *sync* objects (*fences* and *queries*).
- *Container Objects* have one or more mutable *attachments*, which are references to other data, state, or container objects. They also have immutable *attachment properties,* which describe how to interpret their attachments. Container objects may not be shared by multiple contexts, mostly because the side effects of changing their attachments may be costly. For example, changing a shader attachment of a program object in use by another context could invalidate the state of that context at worst, and force time-consuming and unexpected relinking and validation at best. Container object classes described below include *framebuffer* objects, *program* objects, and *vertex array* objects.

Concrete Object Descriptions

"An object is not first imagined or thought about and then expected or willed, but in being actively expected it is imagined as future and in being willed it is thought" -- Samuel Alexander

Each of the concrete object classes mentioned above is explained in somewhat more detail here. The descriptions are organized according to the dependencies of the object graph, to avoid backwards references.

Format Objects fully resolve data formats that will be used in creating other types of objects. Such an object's defined usage must either match or be a subset of the usage supported by its format object. Format objects are a powerful generalization of the *internalformat* parameter used in specifying texture and pixel images in OpenGL 2.1. In addition to the raw data format, format objects include:

- intended usage: pixel, texture, and/or sample image, and which texture dimensionalities (1D, 2D, 3D, cubemap, and array), vertex, and/or uniform buffer
- minimum and maximum allowed texture or pixel image size
- mipmap pyramid depth and array size
- and whether data can be mipmapped, can be mapped to client address space, or is shareable.

Buffer Objects replace vertex arrays and pixel buffers, texture images, and renderbuffers from OpenGL 2.1. There are two types of buffer objects. *Unformatted* buffers are used to contain vertex data (whose format and interpretation may change depending on the vertex buffer object they're bound to) or uniform blocks used by shaders. *Images* are formatted buffers with a size, shape (dimensionality), and format object attachment. Changing buffer contents is done with APIs to load data (`lpBufferData` and `lpImageData[123]D`) and to map buffers in and out of client memory with several options allowing considerable flexibility in usage. See the article "Longs Peak Update: Buffer Object Improvements" earlier in this issue for more details.

Texture Filter Objects replace the state set with `glTexParameter` in OpenGL 2.1 controlling how sampling of textures is performed, such as minification and magnification filters, wrap modes, LOD clamps and biases, border colors, and so on. In Longs Peak, texture images and texture filters have been completely decoupled; a texture filter can be used with many different image objects, and an image can be used with many different texture filter objects.

Shader Objects are a (typically compiled) representation of part or all of a shader program, defined using a program string. A shader object may represent part or all of a stage, such as vertex or fragment, of the graphics pipeline.

Program Objects are container objects which link together one or more shader objects and associate them with a set of images, texture filters, and uniform buffers to fully define one or more stages in the programmable graphics pipeline. There is no incremental relinking; if a shader needs to be changed, simply create a new program object.

Framebuffer Objects are containers which combine one or more images to represent a complete rendering target. Like FBOs in OpenGL 2.1, they contain multiple color attachments, as well as depth and stencil attachments. When image objects are attached to an FBO, a single 2D image must be selected for attachment. For example, a 3D mipmap could have a particular mipmap level and Z offset slice selected, and the resulting 2D image attached as a color attachment. Similarly, a specific cubemap face could be selected and attached as a combined depth/stencil attachment. Each attachment point has an associated format object for determining image compatibility. When an image is bound to an FBO attachment, the format object used to create the image and the format object associated with the attachment point must be the *same* format object or validation fails. This somewhat draconian constraint greatly simplifies and speeds validation.

Vertex Array Objects are containers which encapsulate a complete set of vertex buffers together with the interpretation (stride, type, etc.) placed on each of those buffers. Geometry is represented in Longs Peak with VAOs, and unlike OpenGL 2.1, VAOs are entirely server state. That means no separate client arrays or enables! It also becomes very efficient to switch sets of vertex buffers in and out, since only a single VAO need be bound -- in contrast to the many independent arrays, and their interpretation, that have to be set in OpenGL 2.1 when switching VAOs. (The vendor extension `GL_APPLE_vertex_array_object` provides similar efficiency today, but is only available in Apple's implementation of OpenGL.)

Sync Objects are semaphores which may be set, polled, or waited upon by the client, and are used to coordinate operations between the Longs Peak server and all of the application threads associated with Longs Peak contexts in the same share group. Two subclasses of sync objects exist to date. *Fence Syncs* associate their semaphore with completion of a particular command (set with `lpFence`) by the graphics hardware, and are used to indicate completion of rendering to a texture, completion of object creation, and other such events. *Query Syncs* start a region with `lpBeginQuery`, and keep count of fragments rendered within that region. After `lpEndQuery` is called to end the query region, the semaphore is signaled once the final fragment count is available within the query object. In the future we will probably define other types of syncs associated with specific hardware events -- an example would be a sync associated with monitor vertical retrace -- as well as ways to convert syncs into equivalent platform-specific synchronization primitives, such as Windows events or pthreads semaphores.

The remaining objects making up Longs Peak are still being precisely defined. They are likely to include: *display list* objects, which capture the vertex data resulting from a draw call for later reuse; *per-sample operation* objects, which capture the remaining fixed-functionality state used for scissor test, stencil test, depth test, blending, and so on; and perhaps a "*miscellaneous state*" object containing remaining bits of state that don't have an obvious better home, such as edge flag enables, point and line smooth enables, polygon offset parameters, and point size.

Context is Important

"One context to rule them all, one context to bind them" -- with apologies to J.R.R. Tolkien

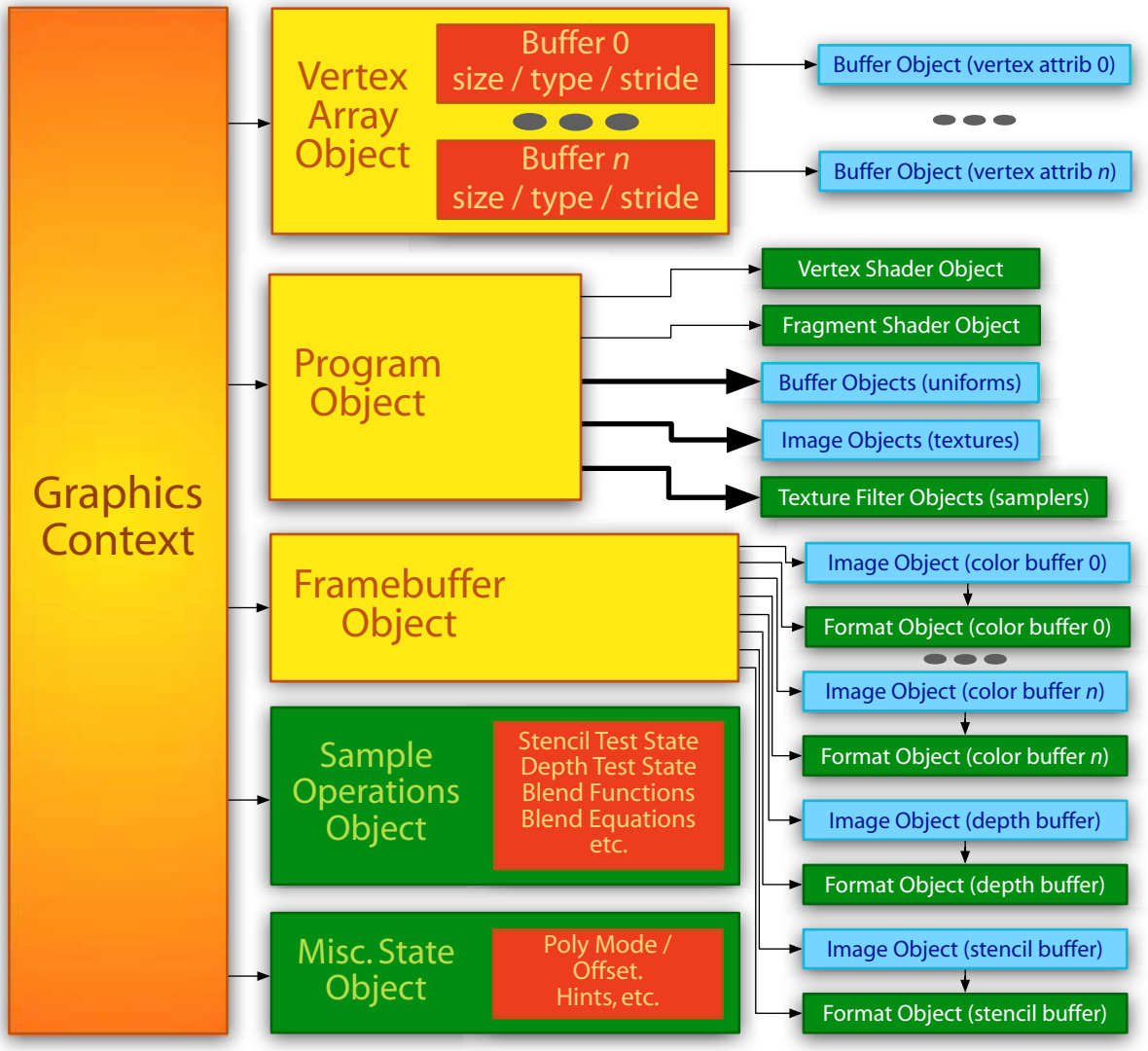
Just as in OpenGL 2.1, the Longs Peak graphics context encapsulates the current state of the graphics pipeline. Unlike OpenGL 2.1, most context state is encapsulated in attributes of server objects. A small number of objects are required to define the pipeline state. These objects are *bound* to the context (see figure 1); changing a binding to refer to another object updates the graphics hardware state to be consistent with that object's attributes.

Changing state by binding objects can be very efficient compared to the OpenGL 2.1 model, since we are changing large groups of state in one operation, and much of that state may have already been pre-validated while constructing the object being bound. This approach will also be useful for applications and middleware layers performing complex state management. It is both more general and more powerful than either the `glPushAttrib/glPopAttrib` commands or encapsulating state changes in GL display lists, which are the only ways to change large groups of state in one operation today.

Figure 1: Graphics Context Bindings. The Longs Peak context contains bindings for geometry (a vertex array object), programs (a program object), a rendering target (framebuffer object), sample operations state, and remaining fixed-functionality state affecting rasterization, hints and other miscellaneous state. In this diagram, yellow objects are containers, green objects are state objects, blue objects are data objects, red blocks represent attributes of container and state objects, and arrows represent attachments to objects or bindings to the context. The context itself, while not strictly speaking an object, is shown in yellow-red to indicate that it takes on aspects of a container object.

Drawing Conclusions

Once all required objects are bound to the context, we can draw geometry. The drawing call looks very much like the OpenGL 2.1 `glDrawArrays`, but combines multiple draw array and primitive instancing parameters into a single call:



```
void lpDrawArrays(LPenum mode, LPint *first,
                  LPint *count, LPsizei primCount,
                  LPsizei instanceCount)
```

mode is the primitive type, just as in OpenGL 2.1. *first* and *count* define the range of indices to draw. *primCount* ranges are specified, so *count[0]* vertices starting at index *first[0]* will be drawn from the currently bound vertex array object and passed to the vertex program. Then *count[1]* vertices starting at index *first[1]*, ending with *count[primCount-1]* vertices starting at index *first[primCount-1]*. Finally, *instanceCount* is used for geometry instancing; the entire set of ranges will be drawn *instanceCount* times, each time specifying an instance ID available to the vertex shader, starting at 0 and ending at *instanceCount-1*.

A similar variation of `glDrawElements` is also provided:

```
void lpDrawElements(LPenum mode, LPsizei *count,
                    LPsizeiptr *indices,
                    LPsizei primCount,
                    LPsizei instanceCount
```

The drawing calls are among the small number of Longs Peak entry points that do not take an object as an argument, since all the objects they use are already bound to the graphics context.

Outline for Success

"If somebody hits you with an object you should beat the hell out of them" -- Charles Barkley

Finally, we've reached the point of outlining a Longs Peak sample program. The outline is not intended to be detailed source code, just to give a sense of the steps that will need to be taken to fully define the objects required for rendering. While this initialization looks complex, most of it is simple "boilerplate" code that can readily be encapsulated in utility libraries or middleware such as GLUT. It is also likely that at least some of the required objects can be predefined by the driver; for example, if the application is rendering to a window-system provided drawable, then a "default framebuffer object" will be provided.

```
// Create a framebuffer object to render to
// This is the fully general form for offscreen
// rendering, but there will be a way to bind a window-
// system provided drawable as a framebuffer object, or
// as the color image of an FBO, as well.
LPformat cformat, dformat, sformat = { create format
objects for color, depth, and stencil buffers
respectively }

LPframebuffer fbo = { create a framebuffer object,
specifying cformat, dformat, and sformat as the
required formats of color buffer 0, the depth buffer,
and the stencil buffer respectively }

LPbuffer cimage, dimage, simage = { create image
objects, specifying cformat, dformat, and sformat as
the formats of the color image, depth image, and
stencil image respectively }

Attach cimage, dimage, and simage to fbo at its color
buffer 0, depth buffer, and stencil buffer attachment
points respectively
// Create a program object to render with

LPshader vertshader, fragshader = { create shader
objects for the vertex and fragment shader stages,
specifying the shader program text for each stage as
an attribute of the respective shader object}

LPprogram program = { create program object,
specifying vertshader and fragshader as attributes of
the program object}

LPbuffer vertbuffer, fragbuffer = { create unformatted
buffer objects for the uniform storage used by the
vertex and fragment shaders, respectively }

Attach vertbuffer and fragbuffer to program as the
backing store for the uniform partitions of the vertex
and fragment shaders, respectively

// Create vertex attribute arrays to render with
LPbuffer attribs = { create an unformatted buffer
object containing all the attribute data required by
the bound programs }

LPvertexArray vao = { create a vertex array object
with specified size/type/stride/offset attributes for
each required attribute array }

Attach attribs to vao at each attachment point for a
required attributes

// Create miscellaneous required state objects
LPsampleops sampleops = { create sample operations
object with specified fixed-function depth test,
stencil test, blending, etc. attributes }

LPmiscstate misc = { create "miscellaneous state"
object with specified rasterization settings, hints,
etc. }

// Bind everything to the context
lpBindVertexArray(vao);
lpBindProgram(program);
lpBindFramebuffer(fbo);
lpBindSampleops(sampleops);
lpBindMiscState(misc);

// Finally, all required objects are defined and we
// can draw a single triangle (or lots of them)
LPint first = 0, count = 3;
lpDrawArrays(LP_TRIANGLES, &first, &count, 1, 1);
```

While we still have a lot of work to do, and the final details may differ slightly, the ARB has now defined the overall structure of the Longs Peak API and the organization and definition of the object classes in the API. We'll continue to show you details of Longs Peak in future issues of OpenGL Pipeline, and when Longs Peak is released, we'll expand these articles into a tutorial and sample code in the ARB's online SDK.

Jon Leech
OpenGL Spec Editor / ARB Ecosystem TSG Chair

(Subtitles in this article are thanks to the late-night availability of Google and www.brainyquote.com)

Transforming OpenGL Debugging to a "White Box" Model

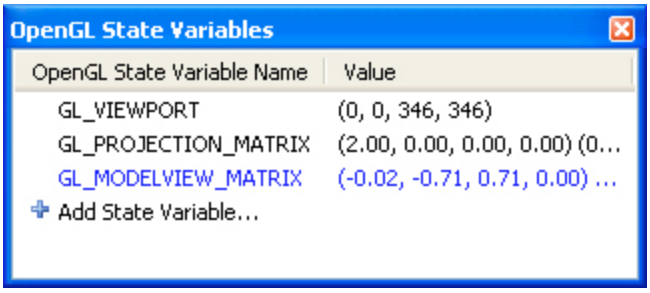
The OpenGL API is designed to maximize graphics performance. It is not designed for ease of debugging. When a developer works on top of OpenGL, he sees the graphics system as a "black box;" the program issues thousands of API calls into it and "magically" an image comes out of the system. But, what happens when something goes wrong? How does the developer locate the OpenGL calls that caused the problem?

In this article we will demonstrate how gDEBugger transforms OpenGL application debugging tasks from a black box model to a white box model, letting the developer peer into OpenGL to see how individual OpenGL commands affect the graphics system.

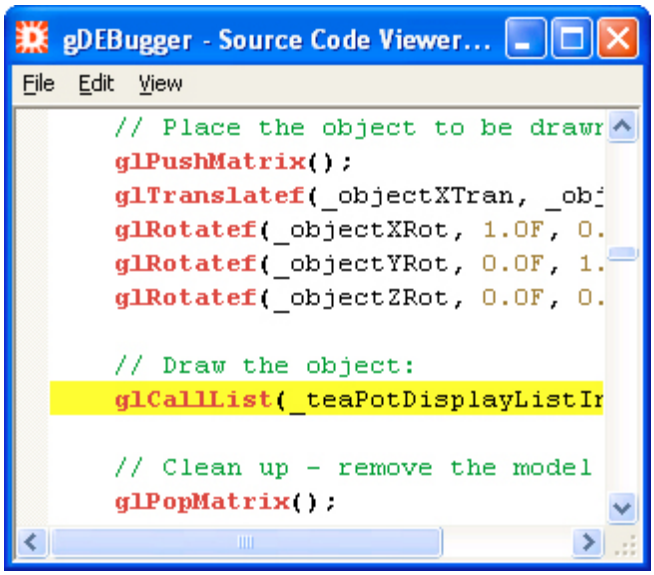
State variable related problems

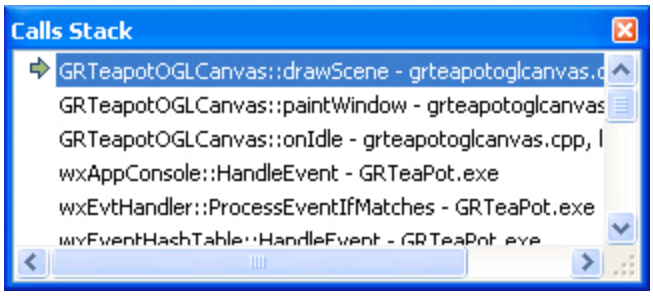
An OpenGL render context is a huge state variable container. These state variables, located inside the graphics system, are treated as "global variables" that are repeatedly queried and changed by numerous OpenGL API functions and mechanisms. However, when using a general purpose debugger, a developer cannot view state variable values, cannot put data breakpoints on state variables, and, at least in Microsoft Visual Studio®, cannot put breakpoints on OpenGL API functions that serve as their high-level access functions. This black box model makes it hard to locate state variable related problems.

Using gDEBugger's OpenGL State Variables view, a developer can select OpenGL state variables and watch their values interactively.



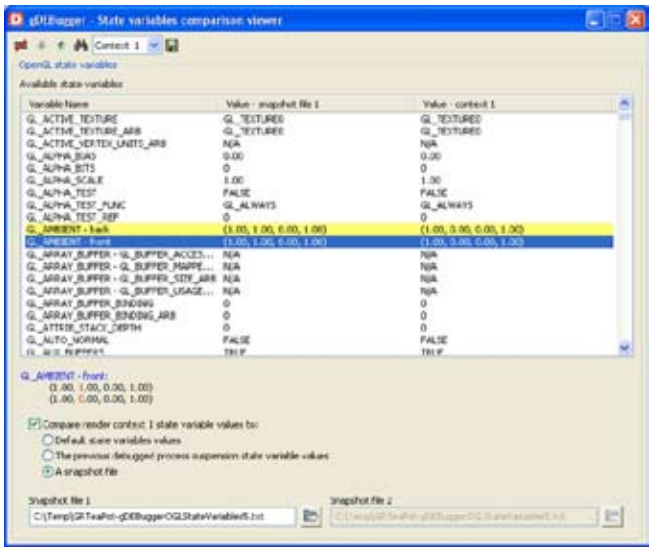
For example, if a program renders an object, but it does not appear in the rendered image, the developer can break the debugged application run when the relevant object is being rendered and watch the related OpenGL state variable values (`GL_MODELVIEW_MATRIX`, `GL_PROJECTION_MATRIX`, `GL_VIEWPORT`, etc.). After locating the state variable values that appear to cause the problem, the developer can put API breakpoints on their access functions (`glRotatef`, `glTranslatef`, `glMultMatrixf`, etc.) and use the Call Stack and Source Code views to locate the scenario that led to the wrong state variable value assignment.





Some OpenGL mechanisms use more than just a few OpenGL state variables. For debugging these mechanisms, gDEBugger offers a State Variables Comparison Viewer. This viewer allows a developer to compare the current state variable values to either:

a. The OpenGL default state variable values.
b. The previous debugger suspension values.
c. A stored state variable value snapshot.



For example, if a game has a mode in which a certain character's shading looks fine, and another mode in which the character's shading looks wrong, the developer can:

a. Break the game application run when the character is rendered fine.
b. Export all state variables and their values into a "state variable snapshot" file.
c. Break the application run again when the character is rendered incorrectly.
d. gDEBugger's Comparison Viewer will automatically compare the OpenGL's state variable values to the exported state variable snapshot file values.

If, for example, the game does not have a mode in which the character is rendered fine, the developer can:

a. Break the game application run when the character is being rendered.
b. gDEBugger's Comparison Viewer will automatically compare the OpenGL's state variable values to the default OpenGL values.

Displaying only the state variable values that were changed by the game application helps the developer track the cause of the problem.

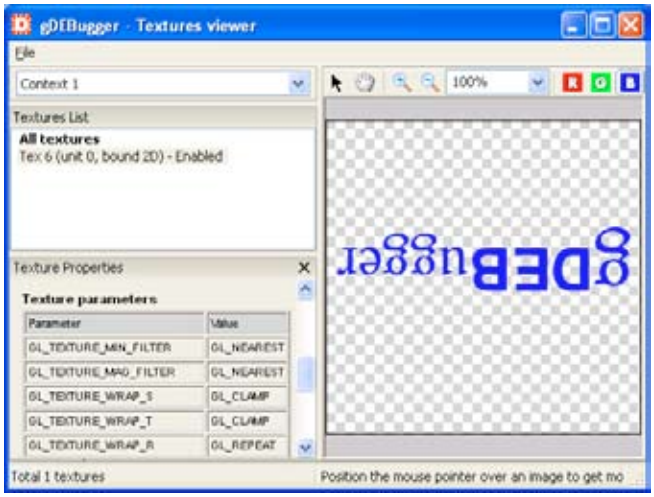
Breaking the debugged application run

In the previous section, we asked the developer to "Break the game application run when the character is being rendered." This allows the developer to view state variable values, texture data, etc. when a certain object is being rendered. gDEBugger offers a few mechanisms to do that:

a. API function breakpoints: The Breakpoint dialog lets a developer choose OpenGL / ES, WGL, GLX, EGL and extension functions breakpoints.
b. The Draw Step command allows a developer to advance the debugged application process to the next OpenGL function call that has "visible impact" on the rendered image.
c. The Interactive Mode Toolbar enables viewing of the graphics scene as it is being rendered, in full speed or in slow motion mode. This is done by forcing OpenGL to draw into the front color buffer, flushing the graphics pipeline after each OpenGL API function call and adding the desired slow motion delay.

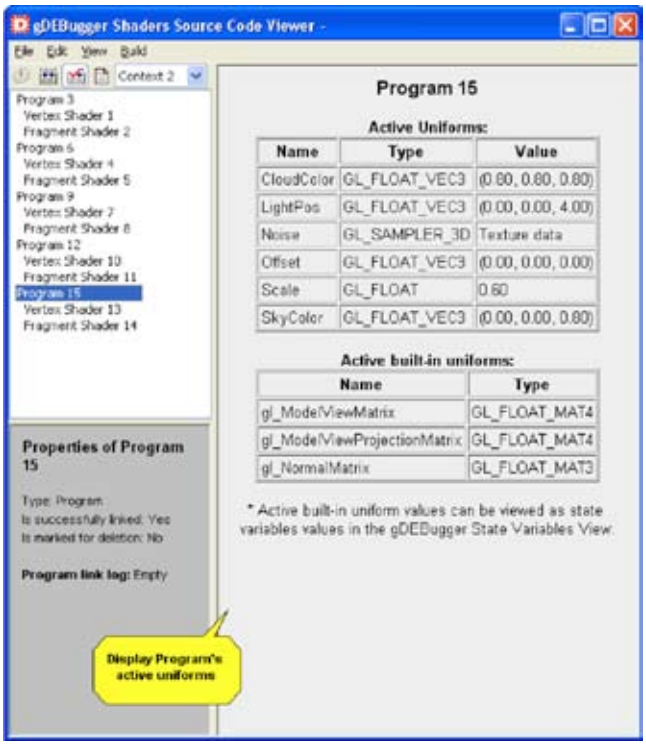
Texture related problems

gDEBugger's Textures Viewer allows viewing a rendering contexts' texture objects, their parameters and the texture's loaded data as an image. Bound textures and active textures (those whose bind targets are enabled) are marked. This helps the developer to pinpoint texture related problems quickly and easily.



Program and shader related problems

gDEBugger's Shaders Source Code Editor displays a list of programs and shaders allocated in each rendering context. The editor view displays a shader's source code and parameters, a program's parameters, a program's attached shaders, and its active uniform values. The editor also allows editing shader source code, recompiling shaders, and linking and validating programs "on the fly." These powerful features save development time required for developing and debugging GLSL program and shader related problems.



We hope this article demonstrated how gDEBugger transforms the OpenGL debugging task to a white box model, minimizing the time required for finding those "hard to catch" OpenGL-related bugs and improving your program's quality and robustness.

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:	Barthold Lichtenbelt, NVIDIA Tom Olson, Texas Instruments T. Hunter, Object Model TSG Contributor Jon Leech, OpenGL Spec Editor Yaki Tebeka, Graphic Remedy

Khronos BOFS & BOF Socials

If you develop multimedia content for games, DCC, CAD or mobile devices, join these BOFs to learn about the new industry standards for royalty-free multimedia development:

- Applications driving next-generation handset requirements
- Opportunities opened up by innovation and standardization in graphics and mobile gaming
- Technological advances in multimedia handset technology

What's a "BOF?"

They are "Birds of a Feather" events that consist of presentations, discussions, and demonstrations for people who share interests, goals, technologies, environments, or backgrounds and are free of charge, and open to all SIGGRAPH 2007 attendees, and non-commercial in nature. You can find the complete BOF schedule on the Siggraph 2007 website.

WED AUG 8th
All events are held in Conference room #2

» **OpenGL BOF:**
Most widely-adopted 2D & 3D graphics API in the industry
Barthold Lichtenbelt, NVIDIA
5:15pm - 7:00pm

» **OpenGL Party:**
Make the "Ascent to the Top" for Games,
Giveaways & Demos
Andrew Riegel, Khronos Group
7:00pm - 8:00pm