Climbing Longs Peak: The Steep Road to the Future of OpenGL

One of the bedrock components of interactive graphics applications is getting an overhaul—the first one in 14 years. And it’s not just a small face-lift; it’s a fundamental overhaul that might impact the way interactive graphics applications are written for a long time to come.

That component is OpenGL. From the beginning, one of OpenGL’s most defining characteristics has been an emphasis on backward compatibility. Code written for OpenGL 1.0 will still compile and run today, exactly as it did 14 years ago, although hopefully a lot faster. The OpenGL Architecture Review Board (ARB), the custodian of OpenGL’s state and development, has been rigorous about keeping the backward compatibility intact. Now, for the first time, that’s changing. And by Siggraph 2007, the first version of the new OpenGL specifications will be made public. The changes are significant, and people who haven’t been closely following OpenGL’s development won’t recognize much. To understand what motivated this drastic change after more than a decade of stability, let’s look at the history and motivations behind OpenGL.

OpenGL: The past

In 1992, a team at Silicon Graphics Inc. (SGI) designed OpenGL to replace the aging IrisGL API that was used to drive SGI’s high-end graphics hardware systems. IrisGL had gone through numerous hardware generations and multiple window systems (this was before X11 dominated the Unix environment). In the process, IrisGL accumulated many aspects that went far beyond graphics (window management, keyboard and mouse input, and so on), which proved redundant in the current environment. In addition, there were no provisions for clear naming rules. Its developers used many basic names like arc, circ, and draw for API functions, often interfering with applications that tried to use those names. OpenGL was to make a break, alleviate those issues, and provide a cleaner interface for new graphics hardware.

OpenGL’s main design goal is to reconcile two opposing goals: provide an abstraction layer that is usable on various hardware architectures, which would work better with a high-level abstraction, and applicable to a wide variety of applications, which would work better with a very low-level abstraction.

Geometric primitives (such as points, lines, and polygons) are at a very low level, and the lighting model is limited to what was state of the art for hardware implementations in 1993—Phong1 lighting on a vertex level and Gouraud2 interpolation of colors between vertices. The conceptual model is that of a pipeline designed to mimic the organization of early ‘90s graphics hardware (see Figure 1). The application feeds commands and data into this pipeline one by one in an immediate mode command stream. The specification also mandates the existence of a z-buffer, limiting the hardware implementation options.

But the specification also had some significant forward-looking strengths. The first was a complete and accurate specification, which served well to guide both implementers and developers. It was also an open standard that companies other than SGI could support by becoming ARB members. This big step away from the then-custom approach of locking in customers and developers to one manufacturer through proprietary APIs helped make OpenGL the most widely available 3D graphics API across many different platforms.

The specification had a clear focus on separation of functionalities and orthogonality. Unless features competed with each other logically, they could be combined freely and worked well with each other. The specification enabled a great amount of flexibility and power especially in the lighting and material field. Developers could specify arbitrary combinations of directional, point, and spot lights and change them independently, allowing flexible applications as well as development of
higher-level libraries like scene graphs.

All parameters were kept in a state used to display any specified primitives. The state was global and not specified as a parameter to the drawing commands in order to reduce their overhead.

The specification provided support for an array of data types for an application to specify graphics data. Developers could use all data types, from unsigned bytes to double-precision floating point numbers, to specify colors, normals, vertices, and texture coordinates. In addition to the types, OpenGL supported all sensible dimensionalities, allowing developers to use it seamlessly for 2D and 3D operations. One major difference between OpenGL and most other APIs at the time was the approach of specifying data in the smallest possible units: each attribute for each vertex was specified with an individual function call. This approach gave application developers the highest flexibility and enabled them to use data in whatever format it was stored in their application, so they could feed OpenGL without having to rearrange application data structures. Target applications were CAD, simulation, or visualization applications, where graphics is important but not the main point. Furthermore, interactive 3D graphics was not as pervasive, and one of OpenGL's goals was to reduce the entry barrier to this field.

Specialized graphics applications, such as flight simulators, from very early on struggled with OpenGL's setup, due to performance issues. The extremely fine level of API structure puts large emphasis on efficient procedure calls and meticulous, low-level optimizations of inner loops. OpenGL's designers were aware of that, but they expected that the time spent inside the procedures would be great enough to offset the procedure call overhead.

The operative phrase here is “great enough,” which needs to be seen in the time domain: as long as the time spent on the procedure call is negligible compared to the time spent inside the procedure, the overhead is irrelevant. For the graphics systems at the time, this was an achievable goal. It required careful programming and some unusual tricks (for example, calling methods through a function pointer instead of directly was one cycle faster), but it could be done for developers who needed the performance and were willing to spend the time optimizing their program.

For everybody else, there were display lists. Display lists were the only way to give the driver a way to actively optimize the data: they record all executed commands and play them back using a single call at a later time. This would let the drivers analyze the commands in the display lists and recognize certain patterns, like a display list that sets several states for lighting, a display list that specifies a texture or a display list that specifies a geometric object, and optimize those cases accordingly. As they were immutable, the driver was free to rearrange and optimize the data's internal representation as necessary. Display lists would also allow the reduction of the amount of data to be sent over a low-bandwidth channel like a network between the client application and the display server.

One of OpenGL's most far-sighted components was the built-in support for dynamic extensions. The API contains provisions to test the existence of named extensions and to access pointers to new functions as necessary. This lets the hardware vendors implement new functionality totally independently and without needing to change or update the core API. All the vendors must do is write an extension to the specification and allow an application to access the functionality if it’s supported on the platform that it’s currently executing on.

Overall OpenGL was a smart design for a low-level layer abstracting different graphics hardware, powerful for existing hardware generations and flexible for future ones.

**OpenGl: The present**

OpenGL has grown significantly over the past 14 years. Numerous extensions were developed by different hardware vendors (at last count, there were 384 extensions registered in the OpenGL extension registry at [http://www.opengl.org/registry](http://www.opengl.org/registry)). Many of them started out as an experimental feature for one hardware system and were subsequently adopted by other vendors, then by the ARB, and finally incorporated as part...
of a new version of the specification. Over the years, the specifications have been revised eight times (the latest one being OpenGL 2.1 in December 2006). All of these primarily added former extensions to core status without removing any existing functionality. Backward compatibility was seen as one of OpenGL's core strengths, compared to other low-level 3D graphics APIs.

All of these extensions were driven by developments in the graphics hardware arena, which changed significantly over the years. In the early and mid-1990s, it was the graphics workstation companies, focusing on CAD and visual simulation applications. In the late ’90s, the PC and game-oriented graphics card manufacturers took the lead. This competitive business forces such companies to innovate quickly, leading to new chip generations every 12 to 18 months with new architectures and new features. This motivation has led to the explosion of OpenGL extensions even through today. Newer developments in graphics hardware have made many of these extensions obsolete, but for backward compatibility they’re still supported by the drivers and listed in the extension registry. Since OpenGL was invented, the development of programmable graphics hardware in particular has revolutionized the way graphics hardware operates and is used.

When OpenGL was designed, programmability wasn’t an option, as the hardware at the time couldn’t execute a low-level program. Instead, the main control of the pipeline was enabling or disabling certain blocks to ensure that OpenGL implementations stayed close to the hardware and allowed high-performance implementations. However, shortly after OpenGL was designed, this limitation had partially eroded due to the use of programmable (or at least microprogrammable) processors for vertex transformations, even though that programmability was not exposed to the user.

That changed in the early 2000s with the introduction of vertex and pixel shader programmability, which let users fully replace the vertex transformation stage (see the transform and lighting in Figure 1) with their own programs (see Figure 2). Prior to that, achieving higher levels than the built-in shading effects required multiple passes over the geometry, as in Diefenbach’s work. Simulating calculations on a per-fragment level was also done using this approach, or with very creative use of the available texture unit arithmetic hardware.

Programmable shaders really opened the floodgates for new features and approaches for interactive graphics and have been instrumental in almost any publication in this area since then. The striking results are evident in many games, demos, and other applications. In the gaming world, the built-in OpenGL transform and lighting pipeline has seen hardly any use in the past six years. However, other areas like CAD and many scientific visualization systems are very satisfied with the standard models.

The main hurdle in using shaders is that they are an all-or-nothing proposition; there is no way to replace only pieces of the default pipeline with a shader. As soon as a shader comes into the picture, the application developer must cover all features of the standard fixed-function pipeline, like different kinds of lights and numbers of textures, and orthogonality in general.

Given that both approaches are still supported in the latest versions of OpenGL, everybody has access to the most appropriate programming model for their domains.

The price that must be paid for this wide array of functionality is driver complexity. OpenGL has rather strict requirements for feature support. They must be supported either fully or not at all. If the hardware is able to accelerate only parts of the specification, the rest must be emulated in software. At the beginning of OpenGL, this was an important feature that let application developers run their software on all OpenGL-compliant systems, with confidence that it would work the same everywhere (albeit at reduced performance levels on smaller systems). In today’s practice, this often translates into having an implementation of a fairly large part of the OpenGL pipeline in software inside the driver, in addition to the compiler for the shading language(s) that it supports. But that’s not all the drivers must do.

As we’ve mentioned, OpenGL was designed with a lot of flexibility in mind, and therefore the API is at a very low level with a large number of procedure calls to specify the object’s geometry instead of building complex data structures internal to the graphics library and drawing from those. This was a good approximation to the graphics hardware structure at the time it was designed. Workstation systems like those of Akeley and Jermoluk were strongly pipeline oriented, as the bandwidth between the main system and the graphics subsystem was large enough to transfer the graphics data as needed, and high-speed memory was too expensive to have a lot of it on the graphics system. Only in Montrym’s work was the first buffer for geometry data added to the graphics system.

In the PC space, the connection between the main memory and the graphics card is more of a bottleneck, and memory has gotten cheap enough that large amounts of it can be dedicated to the graphics card itself. Therefore, nowadays the problem of getting good performance out of the graphics system revolves around how to make sure that the data needed is stored on the
graphics card when it is needed. Modern high-end graphics cards have more than 512 Mbytes of memory of board, in large part for this purpose.

This evolution significantly changes the way geometry needs to be handled, as it is now necessary to keep as much of the graphics data directly on the card as possible (in strong contrast to the OpenGL pipeline model). OpenGL's original answer to this problem was display lists.

But display lists turned out to be too generic to be efficiently implementable. As they store arbitrary sequences of OpenGL commands, they can contain partial primitives (such as the first two vertices of a triangle), while the rest of the primitive is either given explicitly in immediate mode or even in another display list. This level of flexibility makes it hard to optimize them beyond a simple record and replay functionality. Given that they were the only way to optimize on a larger scale in the original specification, this creates a significant conflict. Besides implementation problems, there are also conceptual problems with display lists. The idea of replaying recorded commands does not take into account situations where the data's format and memory layout stays constant, but the content changes (for example, video-based textures or dynamically changing vertex positions). Given that memory management is an expensive operation, not being able to separate the two aspects creates another conflict.

Both of these problems became apparent soon after the introduction of OpenGL in the form of handling textures. One of the first extensions to the specification was the addition of texture objects (GL_EXT_texture_object) as a way to manipulate and manage textures explicitly and more specifically than a special case of display lists. Other extensions that followed quickly and were integrated into version 1.1 of the specification were concerned with reducing the number of procedure calls necessary to specify geometry, at the cost of limiting the flexibility of data layout (GL_EXT_vertex_array). This trend of adding extensions to give more information to the driver in fewer commands continues today, and by now there are many different ways of specifying and manipulating data for OpenGL consumption. As a consequence, applications that want to stay compatible with a wide variety of hardware platforms have to support many different software paths to support all the different extensions, and drivers that want to support a wide variety of applications must support all the ways of doing it. At the same time, the support from the driver side is not equally well developed for all of them. Typically only one best way will deliver the highest performance, but finding out which one it is for each driver/board combination is left to the application developer.

Another large area of extensions addresses ways of approximating per-pixel lighting calculations through applying multiple textures and simple, hard-coded/configurable (as opposed to programmable) arithmetic. A lot of these extensions have been made effectively obsolete by the introduction of programmable fragment shaders. But, again for backward compatibility, driver writers must support them in whatever way they can make the current hardware do it.

This list of existing but effectively obsolete extensions has detrimental effects on many levels. Developers have a wide variety of ways to solve problems, but only a few of them are actually native to the hardware and deliver the best possible performance. This forces developers to experiment quite heavily and use a lot of extension handling to try to determine the best way, depending on which hardware is currently used to run the application. Driver writers have to come up with ideas on how to support old features on new hardware and spend time developing, verifying, and debugging large amounts of code that few (if any) applications care about any longer. For both of them, the specification gets longer and more intricate to describe the interactions between many different extensions that are hardly ever used.

This situation consumes a lot of resources on all sides of the table that could be much better used for developing new algorithms and applications as well as advancing and optimizing drivers to better support what the hardware really looks like. This is where the next generation of OpenGL comes in.

**OpenGL: The future**

In September 2005, the ARB began efforts to write a new specification to clean up obsolete functionality and adapt OpenGL to become what it was designed to be 14 years ago: a thin layer on top of the hardware that hides differences between different implementations and provides the most efficient access to the hardware’s functionalities. In December 2005, ATI and Nvidia presented a first proposal, and different working groups have been working on the details since then. At the time of this writing, some details are still undefined and not published, but the main direction is clear, and a first full version is expected to be presented at Siggraph 2007.

The ARB will define two versions of the new API. The first one, which is the version expected to be ready for Siggraph, is code-named OpenGL Longs Peak (LP) and targets current- and last-generation hardware and will encompass the functionality described in the current OpenGL specification 2.1 from August 2006. The second version, Mount Evans (ME), will extend LP to include new features, like Geometry Shaders and others, to cover all the capabilities available on the latest hardware. The following discussion applies to both versions.
In these new versions of OpenGL, three major areas of change are expected:

- **Vertex Buffer Objects.** Geometry specification will be done using only VBOs—no immediate mode, no vertex arrays, no display lists. The last point is still under discussion; the API might include support for display lists that contain only geometry, but most likely layered on top of the VBO solution.

- **Shaders.** The old fixed function lighting, shading, and texturing pipeline is completely removed; instead everything is based on shaders. This also means that all API calls and state variables that dealt with shading parameters and their equivalent inside the shading language will not be available.

- **Global state.** The global state is much more consolidated in the form of larger blocks (objects). An object’s structure is immutable; only the data can be changed once an object is created.

### Advantages
These changes offer some significant benefits, both for a driver and an application developer. “One goal of this cleanup is to eliminate gratuitous flexibility,” Nvidia’s Michael Gold writes in the discussions threads on the OpenGL Web site (http://www.opengl.org). “When we provide too many mechanisms for accomplishing the same task, it’s less clear to developers which path to choose, and it’s more difficult for implementers to optimize all the paths.”

Having only one way of specifying geometry simplifies both their jobs, as they have to support only one data path. As a result, the development effort can be more focused, and the one path will be better optimized than the variety of paths available right now. “Fixed function hardware disappeared three generations ago. The driver is generating programs on your behalf to emulate the legacy interface,” Gold writes.

Removing the fixed function pipeline removes a large part of the API, reducing the amount of code that must be maintained in the driver quite significantly. It also removes the need to keep the code that generates the shader programs for the fixed function pipeline in the driver.

Changing the object model to one that is immutable simplifies the job of verifying the state’s consistency and lets the driver switch between states respectively state objects (that have been verified to be complete and consistent) much more efficiently. This is especially relevant for shader parameters, which can incur noticeable verification and activation costs in current drivers.

These advantages mainly benefit the driver developers, but where is the impact on application developers?

### Disadvantages
The changes we’ve discussed will significantly impact backward compatibility to older versions of OpenGL. To keep old applications running, the regular functionality will be kept in the driver during a transition phase, but new developments will not be ported back into it. There are also ideas for a hybrid approach where some parts of an application use the old model and others use the new one. However, the switch between old and new will probably involve an expensive context switch and incur performance losses in addition to possible interaction limitations. Applications under active development will have to migrate to the new model sooner or later—the sooner the better.

For applications that already use shaders exclusively, the change is not all that large. For applications that use the fixed function pipeline, however, this can mean a significant effort. One of the attractive features of OpenGL was always the low entry barrier: getting the first OpenGL program up and running can be done in less than 20 lines of code (plus window creation), and the used commands are easy to explain. If OpenGL LP requires the use of shaders for even elementary operations, this entry barrier will rise significantly, noticeably impacting the use of OpenGL as a development and teaching tool.

In addition to raising the initial entry barrier, focusing on shaders also removes the orthogonality that characterized large parts of OpenGL. Shaders are all or nothing; the application has to take full responsibility and provide the correct shader for each combination of light sources, materials, and textures.

Applications with a fixed environment (such as only one directional light source) will not have a major problem with this, but applications that support dynamically changing environments (for example, based on user input) will need to spend noticeable effort on a shader framework.

Similar arguments apply to the geometry and object changes. Immediate mode made it easy to just specify the data that was immediately needed. Having to create a VBO and a geometry object just to draw a single triangle seems like a lot of overhead.

### Benefits and chances
So, it seems like the transition has mainly disadvantages for the application developers. Admittedly that is a negative way of looking at the situation. Experienced developers will do most of the necessary things already anyway: using VBOs for geometry has been the high-performance way since they were introduced, and many research applications (fewer, but some, commercial ones) are using shaders for all their lighting calculations anyway. The group that will suffer most under the new rules are inexperienced users and learners.

But for those application areas, performance is generally a secondary issue. In that case, the simple interfaces of the old OpenGL can be layered on top of the
new system. It will incur some overhead, but that will be more than offset by the increased learning speed.

Naturally the OpenGL ARB is aware of these issues, and they are considering designing these wrapper layers to provide an interface at a level similar to the old OpenGL one.

Alternatively, other people in the scientific and software community can do this job. This would lift part of the burden from the ARB and give them more time to perfect (and actually implement) the specification. The same approach has worked well for extension handling libraries, where the open source community has pretty much converged on two libraries (the OpenGL Extension Wrangler Library and the OpenGL Easy Extension library) to handle this task.

But the main potential benefit lies in the incredible flexibility that shaders offer, flexibility that allows much better quality and variability than the fairly fixed OpenGL lighting and shading models could ever support.

At this point, all of this is done at the cost of orthogonality; shaders are very much separated from each other. For a general lighting and shading environment, it would be desirable to be able to mix and match all the areas that shaders can be and are used, from dynamically creating geometry, such as for character skins, past arbitrary light source shapes and shadows, to creating visible surface structures like bumps and wrinkles. Finding a way to independently design and specify these components and efficiently combine them in a unified shader remains an open research problem, if the new way of doing OpenGL is not to just replace but also enhance the old way.

Conclusion

The road up Longs Peak can be fairly long and rocky, depending on where you must start. But the view is terrific, and it goes a long way into the future. So, it’s definitely worth it. We’re currently working on a generic shader composition framework, but the problem area is big enough that, just like the layered libraries we’ve mentioned, there’s room for multiple implementations. OpenGL Longs Peak will come, and the need for supporting infrastructure will be there. Let’s go make it happen!

References


Contact author Dirk Reiners at dirk@louisiana.edu.
Contact department editor Miguel Encarnação at me@imedia-labs.com.