C and its Offspring: OpenGL and OpenCL

By Dr Sean Harmer

Part 2

Feeding the Pipeline

The previous section has explained how the flow of data through the pipeline transforms from vertices, through fragments and eventually, for the lucky few, to pixels on the screen. Of course, modern GPUs are very good at doing this but only if we treat them the right way. The programmable shader-based stages of the pipeline give us a huge amount of flexibility but we also need to feed the pipeline with a steady stream of data so as to not allow it to stall.

In addition to flexibility, it is the desire for greater performance that has shaped the OpenGL API in recent times. Legacy OpenGL that you may have seen with its copious calls to **glVertex3f()** and friends are just not a good way of getting data into the pipeline. OpenGL's threading model means that all rendering commands for a particular pipeline must be issued from the same thread. One CPU core is simply not fast enough to feed data one vertex attribute at a time and keep the GPU fully loaded. Laughably far from it.

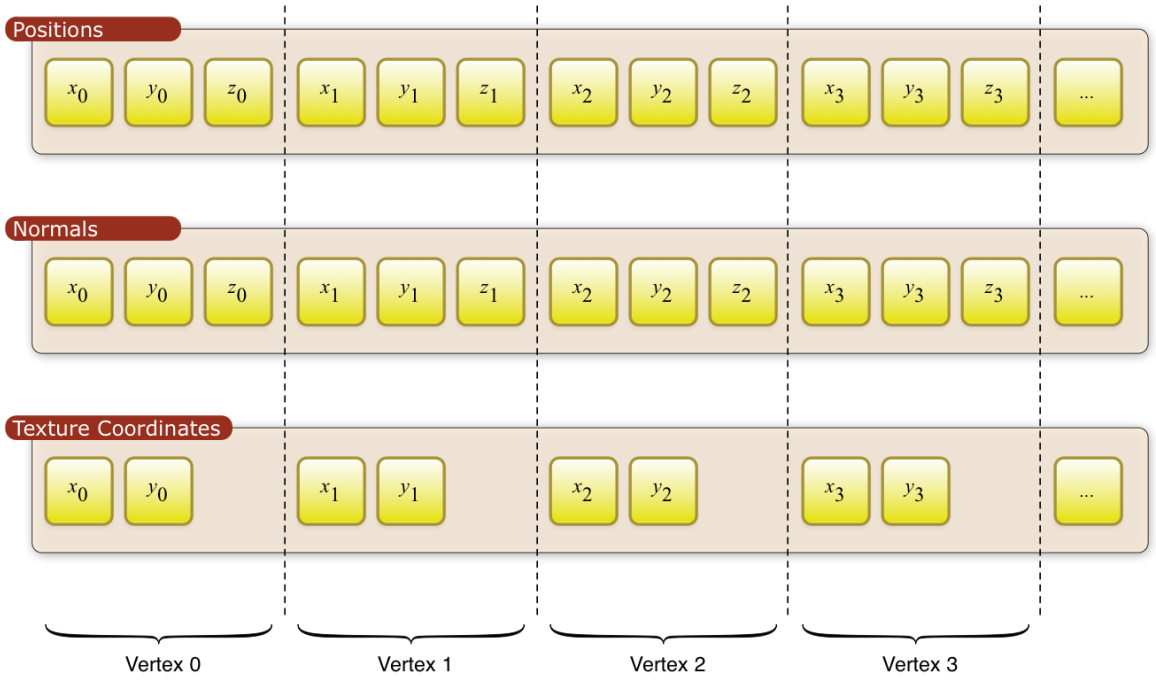

FIGURE 1: OPENGL PERFORMS BEST WHEN WORKING ON LARGE CONTIGUOUS BLOCKS OF MEMORY STORED IN BUFFER OBJECTS. FOR TYPICAL GEOMETRY SUCH DATA CONSISTS OF VERTEX POSITIONS THAT GIVE THE ACTUAL GEOMETRIC SHAPE OF

THE MESH TO BE RENDERED; A NORMAL VECTORAT EACH VERTEX THAT FEEDS INTO THE LIGHTING CALCULATIONS IMPLEMENTED IN THE PROGRAMMABLE SHADER STAGES OF THE PIPELINE; AND TEXTURE COORDINATES USED TO MAP IMAGES ONTO THE SURFACE OF THE 3D MESHES BEING DISPLAYED. THIS DIAGRAM ILLUSTRATES ONE POSSIBLE CONFIGURATION IN WHICH THE DATA CAN BE ARRANGED INTO MULTIPLE BUFFER OBJECTS. OTHER, MORE COMPLEX ARRANGEMENTS ARE POSSIBLE THAT GIVE BETTER PERFORMANCE AS A RESULT OF IMPROVED CACHE COHERENCY. IT IS NECESSARY TO TELL OPENGL ABOUT THE FORMAT OF THE DATA SO THAT IT KNOWS HOW MUCH DATA TO FEED INTO THE PIPELINE FOR EACH VERTEX.

Consequently, OpenGL now requires us to package up the input data (vertex positions and other attributes) into relatively large packages with a well-defined (but user specifiable) format. Figure 1 shows one possible ways of arranging typical vertex data (positions, normal vectors and texture coordinates) into buffer objects. The buffer objects that contain our vertex attribute data can be associated with the inputs to a vertex shader with a few commands on the CPU (**glVertexAttribPointer()** and friends). If we tell OpenGL how the data in our buffers will be used and how often it is likely to be updated, the OpenGL driver *may* be nice and DMA the buffer of data such that it resides in nice and fast GPU-side memory (if your GPU and CPU don't have a shared memory architecture).

The upshot of this is that with minimal CPU overhead[1] we can get everything needed by the GPU lined up and ready to go prior to issuing a draw call. Think of issuing an OpenGL draw call such as **glDrawElements()**, as simply pulling the trigger on a starter's pistol. The draw call returns immediately and the CPU is free to get on with other work (queueing up additional OpenGL work or anything else) whilst the GPU asynchronously gets on with the work described in the previous section.

Maintaining this degree of parallelism between the CPU and GPU is key to maintaining good performance in OpenGL.

OpenGL Without the Triangles

Newer versions of OpenGL (OpenGL 4.3 or OpenGL ES 3.1) introduce a second pipeline in addition to the graphics rasterization one we have been discussing up to now. This second pipeline is very simple and consists of a single programmable shader stage – the *compute shader*.

What is the compute shader and what is it good for, I hear you ask? Well, a compute shader is written in GLSL just like its graphical counterparts, and it is useful when you want to perform general purpose computations that do not directly involve rasterizing primitives. Ideally, to take full advantage of the GPU's parallelism, the algorithms that you code up into compute shaders should be nicely parallelizable in terms of the data they operate on and have no (or very few) dependencies between blocks of work.

[1] Minimal for OpenGL anyway. Vulkan and other explicit APIs offer even better approaches due to newer threading models.

But when should we use compute shaders instead of OpenCL? The functionality exposed by compute shaders in OpenGL is not as full featured as the facilities offered by OpenCL, Cuda and other similar APIs. However, OpenGL's compute feature does have one major advantage: there is no very expensive context switch required when switching between OpenGL graphics and compute pipelines as there is when switching between OpenGL and OpenCL or Cuda and back again.

The upshot of these conditions is that OpenGL's compute shaders often find very good use for processing data that is close to the graphics. For example, performing physics particle simulations by updating the contents of a buffer containing particle positions and velocities or convolving texture data with a kernel ready to be displayed in a subsequent graphics pipeline pass. If your needs exceed that of OpenGL compute shaders, then you will need to look elsewhere and use an interoperability API if you then need to render the results of the calculations performed. We do not have space here to do justice to OpenCL but there are some fantastic online resources available such as https://handsonopencl.github.io/.

Using OpenGL in Practice

OpenGL is very often the go-to API of choice when you want fluidly animated user interfaces, or need to display large quantities of data, or as often found[2], both. OpenGL cannot be used in isolation to write an entire application. OpenGL knows nothing of window surfaces, input devices, and the raft of other tasks an application has to complete. The process of obtaining a window surface on which to draw and an OpenGL context (that holds the current OpenGL state) requires the use of platform-specific APIs such as EGL, WGL, GLX, CGL etc. Alternatively, one can use a cross-platform toolkit such as **Qt**[3] that abstracts these things away nicely allowing you to concentrate more on the task at hand.

Once you have performed the mundane tasks of obtaining a window and context, you can set to with your fancy shader based pipelines and big buffers of data to render your masterpiece at 60 fps. More likely, you will find yourself staring at a black screen because you made a subtle mistake in one[4] of a hundred possible ways that means you don't see what you hoped. Some classic examples of mistakes (all of which I have made at various times) are: transforming the object you want to see so that it is behind the virtual camera; putting the camera inside the object and disabling rasterization of the back faces of triangles; drawing a quad exactly edge on; incorrectly transforming vertices in your vertex shader; using an incomplete texture object (gives black object on a black background) and many more.

[2] At least in the projects KDAB is often asked to complete.
[3] http://www.qt.io/
[4] If you're lucky it's only one problem. In reality there are often multiple issues to resolve.

To avoid repeating many of these mistakes time and again, developers either end up writing their own set of wrappers and utilities to help manage the large amounts of data and concepts or they use an existing framework. There are many such frameworks, often referred to as scene graphs available, both open source and commercial. Take your pick of the one that suits you best. However, you still need a good mental model of what the OpenGL pipeline is doing under the hood and who knows when you'll need to tweak that shader that ships out of the box but doesn't quite do what you need.

Qt once again shines here. The Qt3D module is currently undergoing rapid development and allows both C++ and QML APIs to create and manage your scene graph. Moreover, Qt3D also allows you to specify exactly how the scene graph is processed and rendered by the backend. This allows you, for example, to completely switch the rendering algorithm dynamically at runtime – when transitioning from an outdoor to an indoor scene for example. Qt3D is also extensible to a great degree, and the future will bring features such as collision detection, AI, 3D positional audio and much more.

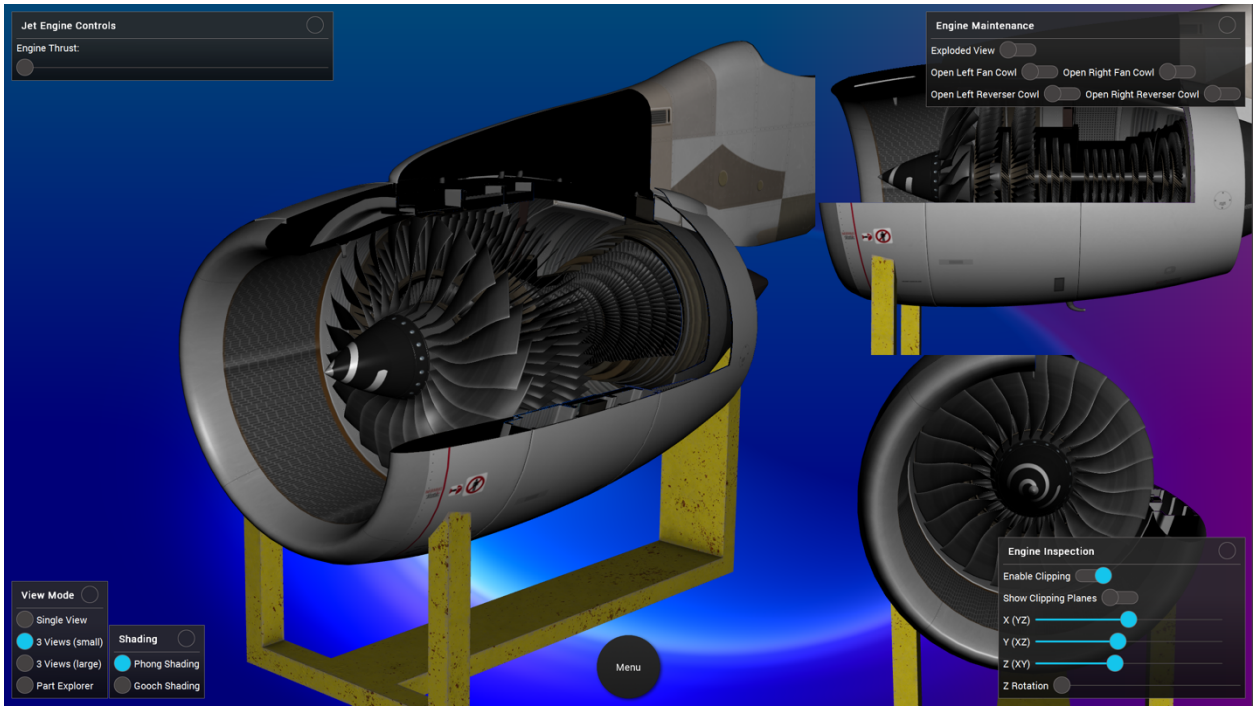

FIGURE 2: EXAMPLE OF AN APPLICATION WRITTEN WITH QT3D SHOWING OFF THE POWER OF OPENGL. THE QT3D FRAMEWORK ALLOWS A MUCH HIGHER LEVEL OF ABSTRACTION THAT USING RAW OPENGL BUT STILL PROVIDES A POWERFUL FEATURE SET. IN THIS SCENE WE CAN SEE A MODEL OF A JET ENGINE BEING RENDERED FROM 3 SEPARATE CAMERA POSITIONS INTO 3 REGIONS OF THE WINDOW. THE ENGINE AND STAND USE A VARIETY OF MATERIALS AND TEXTURES TO GET THE DESIRED SURFACE FINISHES. CUT PLANES ARE IMPLEMENTED WITH SOME CUSTOM GLSL SHADERS TO SELECTIVELY ALLOW SEEING THE INTERNAL STRUCTURE OF THE ENGINE. THE BACKGROUND IS SMOOTHLY ANIMATED TO VARY SLOWLY OVER TIME AND THE 2D USER INTERFACE IS PROVIDED BY QT QUICK AND BLENDED OVER TOP OF THE 3D CONTENT INCLUDING REAL TIME TRANSPARENT PANELS.

Finally, you need to consider how to composite your OpenGL scene with a traditional 2D user interface and any other data such as camera feeds, video streams, 3rd party mapping frameworks etc. As described in the May 2015 issue of RTC[5], Qt provides the Qt Quick UX technology stack. It just so happens[6] that Qt Quick is rendered using OpenGL (by way of a scene graph that specializes in 2D content). Qt Quick also supports camera and video data and is relatively easy to integrate with mapping engines too. Of course Qt Quick and Qt3D work seamlessly together and allow the creation of stunning user interfaces that combine 2D and 3D aspects. Figure 2 shows an example of Qt3D and Qt Quick in action together running at 60 fps.

The Future

Over time, more facilities have been added to OpenGL to get even better performance. Features such as primitive restart, instanced rendering, array textures, and indirect rendering allow a single draw call to trigger far more work than was possible without them – this is often referred to as increasing the batch size.

However, whilst maintaining backwards compatibility there is one problem that cannot be solved in OpenGL. It's a big one[7]. The threading model. OpenGL originates from a time of processors with single cores and this reflects in its architecture. All OpenGL commands for the current context have to be issued from the thread on which the context is current. This means that on modern machines the CPU often has N-1 cores idling whilst they wait for the core driving the OpenGL context to complete its work each frame.

To solve this a new approach is needed. **Vulkan** is the answer to this from Khronos. Vulkan is a new API designed from the ground up to eliminate the problems with OpenGL. It features a threading model that allows multiple CPU cores to build up command buffers that can later be issued to the GPU. Potentially such command buffers can be reused multiple times, even across frames, saving the CPU even more workload in situations that exhibit a high degree of temporal coherency.

Additionally, Vulkan is also an explicit API. The application (you) is responsible for managing every aspect of the resources used by the pipeline. This minimizes surprises that can sometimes be present in OpenGL when certain commands take far longer than expected due to the driver having to do book keeping behind the scenes.

The advent of Vulkan does not spell the end for OpenGL by any means. OpenGL is still perfectly capable of driving the vast majority of use cases encountered today. Vulkan will allow

[5] Embedded's Gone Cute: http://rtcmagazine.com/articles/view/110543, by Rafael Roquetto.
[6] Of course it didn't "just" happen, Qt was designed this way.
[7] If you're really trying to extract every last piece of performance from your GPU or to keep within a thermal envelope.

developers to scale the graphical aspects of their applications horizontally across multiple CPU cores. This is most beneficial at present for high-end game engines on the desktop but also for mobile and embedded devices wishing to get better use of the limited thermal envelopes and/or battery life. Although the API of Vulkan is necessarily different to OpenGL, the concepts are very much the same. So if you have never touched OpenGL before, investing into learning it now will not be lost even if you wish to migrate to Vulkan in the future.

Hopefully this article has shed a little light on some of the concepts involved with using modern OpenGL. Of course there is far more to it than we can hope to cover in a few pages. The good news is that all of the sophisticated lighting models, texturing methods, tessellation and geometry processing all build upon the same core principals.

If all of this still seems a little daunting there are plenty of resources available to help you on your way. The official OpenGL wiki[8] is a great place to start and the OpenGL man pages are very thorough. In addition to consulting services, KDAB also offers professional 3-5 day training courses that can help explain the concepts, demonstrate the techniques that build upon them and arm you with everything needed to be productive with OpenGL. We even include around 100 examples and hands-on exercises to cut your teeth on. See http://www.kdab.com/training/courses-curricula/modern-opengl/ for full details.

About the Author: Dr. Sean Harmer, KDAB

Dr Sean Harmer is a Senior Engineer and Director at KDAB. He has been developing with C++ and Qt since 1998, and in a commercial setting since 2002.

Sean holds a PhD in Astrophysics along with a Masters in Mathematics and Astrophysics. He has a broad range of experience and is keenly interested in scientific visualisation and animation using Qt and OpenGL.

Sean is the maintainer of the Qt3D module and an experienced trainer in OpenGL and Qt. He lives in the north of England and enjoys drinking tea.

http://www.kdab.com/about/contact/

[8] https://www.opengl.org/wiki