C and its Offspring: OpenGL and OpenCL

By Dr Sean Harmer

Part 1

Introduction

The advent of the iPhone-era has ushered in a step change in the paradigms of visual display and user experience from desktop, through mobile and down to embedded applications. This change is being driven by the user's expectations and by what modern hardware is capable of. No longer will users accept mechanical buttons as the primary method of interaction. Even on machine shop and factory floors, the users are demanding visually pleasing, fluid and intuitive user interfaces. At the same time, today's hardware is capable of so much more than that of yesteryear both in terms of graphical output and compute processing.

We will give an overview of an API used to meet the demands of users and engineers to improve the visual quality and computational throughput of systems: OpenGL. We will cover the mental model needed to drive OpenGL, example use cases; explore how to integrate them with the rest of your system; and where the future is heading.

OpenGL for Graphics and Compute

OpenGL has been around in one shape or another for 22 years. Over this time it has always had a dedicated set of followers but the wider technical audience that has dabbled with OpenGL has often been left with a sense of confusion, wild-eyed wonderment or perhaps even fear. To a large extent the level of OpenGL's impenetrability was caused by the mental model of engineers not matching the reality of what OpenGL was executing under the hood. This is no fault of the engineers placed in this situation. Legacy OpenGL was a beast. Lots of global state, an archaic binding-to-edit object model, and cruft gathered as graphics hardware evolved into its current form, which is vastly different to when OpenGL was first conceived.

Silicon Graphics Inc. (SGI) started developing OpenGL in 1991. From 2006, OpenGL has been developed by The Khronos Group, a non-profit consortium of companies. Since Khronos took over, OpenGL has caught up and surpassed competitors. OpenGL is very popular in the fields of CAD, virtual reality, scientific visualization, information visualization, flight simulation, and video games.

Fortunately, in comparison modern OpenGL is much more approachable, flexible and performant, though it is still necessary to have a good mental model of how OpenGL operates. The key to this is understanding the so-called pipeline and how it interacts with the OpenGL C API. The pipeline describes the flow of data through OpenGL and it can be configured in numerous ways to achieve all manner of rendering algorithms such as environment mapped reflections, stylized shading (toon, ink, pencil), shadows, global illumination and many more. Before we can learn about such higher level algorithms, we need to understand the basic pipeline that forms the fundamental building block. So take a deep breath and let's dive in.

The OpenGL Graphics Pipeline

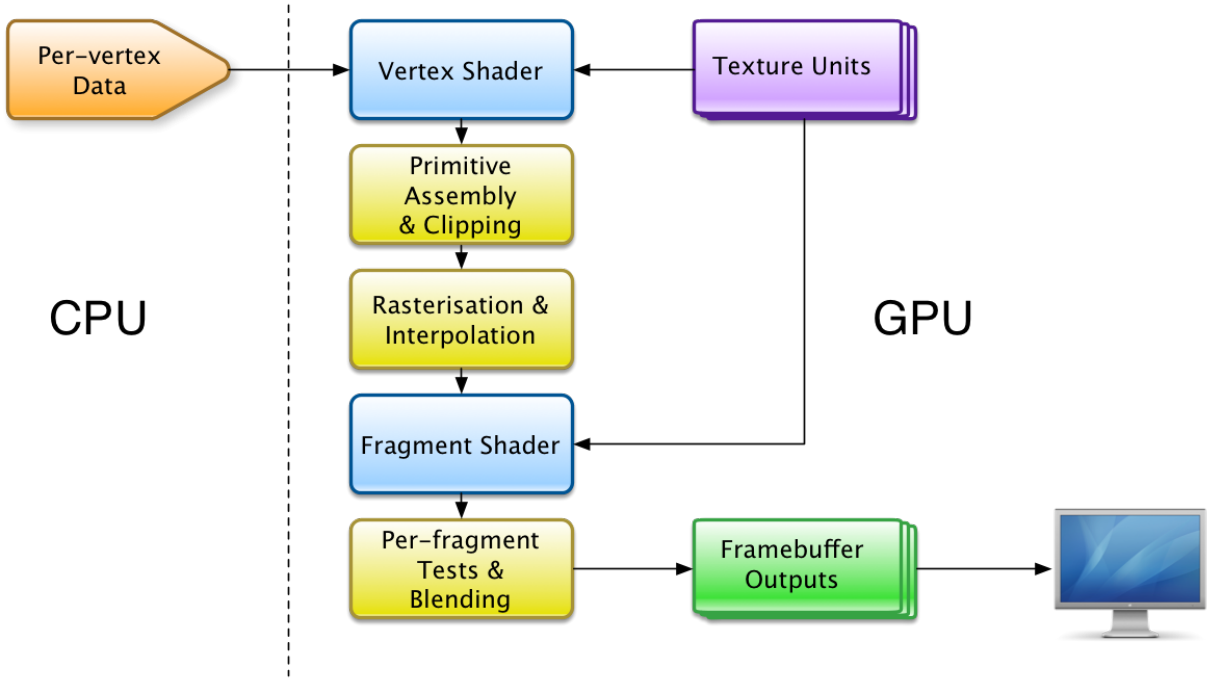

FIGURE 1: SIMPLIFIED OVERVIEW OF THE OPENGL GRAPHICS PIPELINE. DATA FLOWS FROM THE TOP-LEFT TO THE BOTTOM RIGHT. TO BEGIN WITH THE DATA IS GEOMETRIC IN NATURE (WITH ACCOMPANYING DATA). THE VERTEX SHADER STAGE PERFORMS COORDINATE TRANSFORMATIONS. IN THE RASTERIZATION STAGE THE GEOMETRIC PRIMITIVES ARE CONVERTED INTO FRAGMENTS AND ARE LATER GIVEN A COLOR BY THE FRAGMENT SHADER. THOSE FRAGMENTS THAT SUCCESSFULLY PASS A SET OF TESTS EVENTUALLY GET DISPLAYED ON THE RENDER TARGET.

Figure 1 shows a simplified schematic view of the OpenGL pipeline. It begins with data being fed in from the CPU (we will see how shortly). The data usually boils down to a set of vertex positions and their associated attributes (color, normal vector, texture coordinates etc.) but this data can be anything we can encode into a few floats, booleans or integers. Modern OpenGL allows us to be flexible. No longer are we tied to what the designers of the original OpenGL thought we should be using.

Each vertex and its attributes are passed into the vertex shader – a programmable piece of logic. We'll find out later why a shader is thus called. A modern GPU may allocate many cores to processing vertices in parallel, but each instantiation of the vertex shader can only operate on a single vertex at a time. The typical task a vertex shader performs is that of coordinate system transformations. This may be to transform from *model space* to *eye space* for lighting calculations; to *world space* for environment mapping; to *tangent space* for normal or parallax mapping or one of many other possibilities. One thing a vertex shader must do[1] however, is to output the vertex position in *clip-space* as this is used as input to the rasterizer.

Understanding coordinate systems and the transformations between them is key to making effective use of OpenGL. Trying to shortcut this only leads to misery down the line. Be sure you understand the important coordinate systems and when each one is of use, and how to get your data into that coordinate system.

As the transformed vertices pop out of the vertex shader, they are processed by the first[2] piece of fixed functionality in the pipeline – primitive assembly and clipping. This is where the individual vertices that make up a graphical primitive (point, line or triangle usually) get pulled together into a logical entity. This construct is then clipped against the volume that is eventually mapped to the current render target (usually the back buffer of a native window surface or a texture).

The OpenGL graphics pipeline consists of several programmable and fixed function stages. The programmable stages are controlled by writing short programs in the GLSL language that execute directly on the GPU. Where such flexibility is not required the pipeline uses blocks of fixed functionality implemented directly in silicon. Some of these fixed stages can be tweaked to some extent by calling OpenGL API functions from C/C++. Think of these as levers and dials on a machine that change how the machine operates.

Armed with the clipped primitives, the rasterizer is then able to generate fragments for each primitive. Think of a fragment as a pixel in training. Our nascent fragments still have a long journey ahead of them before they may graduate to become a fully-fledged pixel. To help them on their way, each fragment contains data for not only its position but also potentially a host of other data too.

[1] Actually it's the final stage before rasterization that must output the clip-space coordinates. That means the geometry shader if present, otherwise the tessellation evaluation shader if present, otherwise the vertex shader as in the simplified pipeline introduced here.
[2] Again, technically this is not quite correct. The first piece of fixed functionality is the vertex puller that feeds the vertex shader but that is beyond scope for this article.

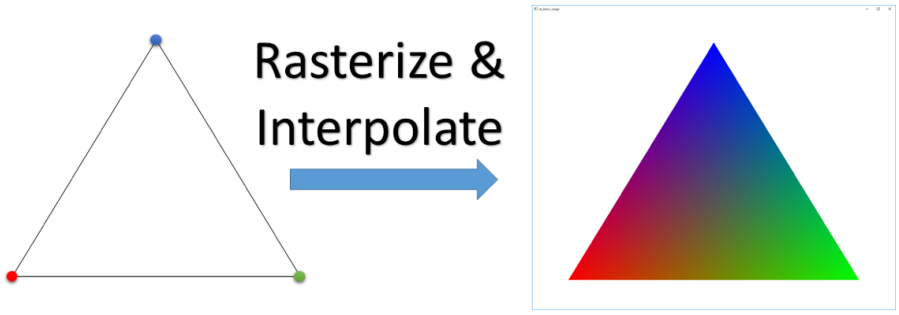

FIGURE 2: THE 3 VERTICES OF A TRIANGLE ARE SUBMITTED TO OPENGL. AFTER TRANSFORMATION THE VERTICES ARE ASSEMBLED INTO A TRIANGLE. THE RASTERIZER PERFORMS A SCAN-LINE CONVERSION OF THE TRIANGLE AND ANY ADDITIONAL ATTRIBUTES ASSOCIATED WITH THE VERTICES ARE INTERPOLATED ACROSS THE SURFACE OF THE TRIANGLE TO CREATE FRAGMENTS. THE FRAGMENTS ARE THEN FED INTO THE FRAGMENT SHADER TO BE PROCESSED FURTHER.

Recall the attributes that we have associated with each of our vertices. Each of these attributes is interpolated across the primitive by the rasterizer. As an example imagine the simple case of the 3 vertices shown in Fig. 2. The 3 vertices have position and a color attribute: red, green and blue respectively. For each fragment generated by the rasterizer, these three colors are interpolated to give a color at the position of that fragment. At the precise center of the resulting triangle (assuming the center is conveniently aligned to the pixel grid) there will be a fragment whose color consists of equal amounts of red green and blue – perfectly grey[3].

Depending upon the detail of the geometry sent into the pipeline, relative to the resulting projected sizes of the rasterized primitives, you will likely find that at this stage of the pipeline there is somewhat of a data explosion. Each of those rasterized fragments must be lovingly crafted by the next programmable stage – the *fragment shader*. It was the fragment shader that gave rise to the general term shader, because the fragment shader's prime responsibility is determining what color, or shade, the fragment should be given on its way to becoming a pixel.

Just as with the vertex shader, each instantiation of the fragment shader executes in isolation from all others[4]. This is to allow many cores on the GPU to process fragments in parallel without

[3] If you can spot it on today's high density displays, you have better eyes than I do.
[4] Sorry, I lied slightly here too. It is possible to get limited amounts of information about neighbouring fragment processing into a fragment shader. This is often achieved via the GLSL functions dFdX and dFdY which allow getting information about gradients between fragments. This is possible because the GPU processes blocks of fragments together and in lock-step. This allows peeking into the registers for neighbouring fragments.

data dependencies between them – remember, there are a lot of fragments to churn through. Given the expressive power and flexibility of the GLSL language, a skilled developer can craft all manner of effects in the fragment shader. For some convincing examples of what can be achieved with a fragment shader and rendering a full-window quad (two triangles, since quads are now relegated to the annals of history), take a look at the impressive examples at https://www.shadertoy.com/.

Although the order in which vertices are fed into the pipeline is well defined it is sometimes useful for a shader stage to be able to sample from a chunk of data in an arbitrary manner. To enable this, data can be exposed to the pipeline in the form of textures, images and special types of buffer object (uniform buffers and shader storage buffers). These can be accessed from any shader stage. Access to textures can also optionally include quite sophisticated sampling and filtering implemented in hardware.

The fragments exiting the fragment shader[5], sporting a (hopefully intended) color now go into another piece of fixed functionality that performs a number of tests that must be passed if our fragment hopes to graduate to pixeldom. Two common examples are the depth test (often referred to as z-testing due to the key role played by the z component in this test) and the stencil test. Both of these tests operate by comparing the data in each fragment to the data in another buffer (the depth buffer or stencil buffer respectively).

Exactly how the data gets into these additional buffers is beyond the scope of this article but suffice it to say that it is very common and very easy to populate the depth buffer. The incoming fragment and the data at the corresponding position in the buffer are compared, using a user-specified comparison operator[6]. If the comparison is true, the fragment passes the test and is allowed to carry on. If the fragment fails it is thrown away.

If blending is disabled, that is the end of the story. The successful fragments get written to the render target and eventually get displayed on the screen (or used as input to a subsequent render pass). If blending is enabled, then the incoming fragments get combined with any fragments that went before it at the same pixel location by way of a user-specified blending operation. At this time blending is still classified as a fixed function, but configurable pipeline stage. Who knows, perhaps in time, blending will also evolve into a full-blown programmable stage.

[5] Not all of them make it out as the GLSL **discard()** function can be used to throw a fragment away and prevent it from undergoing any further processing. This should be used sparingly however as it has performance implications related to early depth testing.

[6] For example, the depth testing comparison function is specified via the glDepthFunc() and passing in an enum value such as GL_LESS, GL_ALWAYS, GL_LEQUAL.

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/