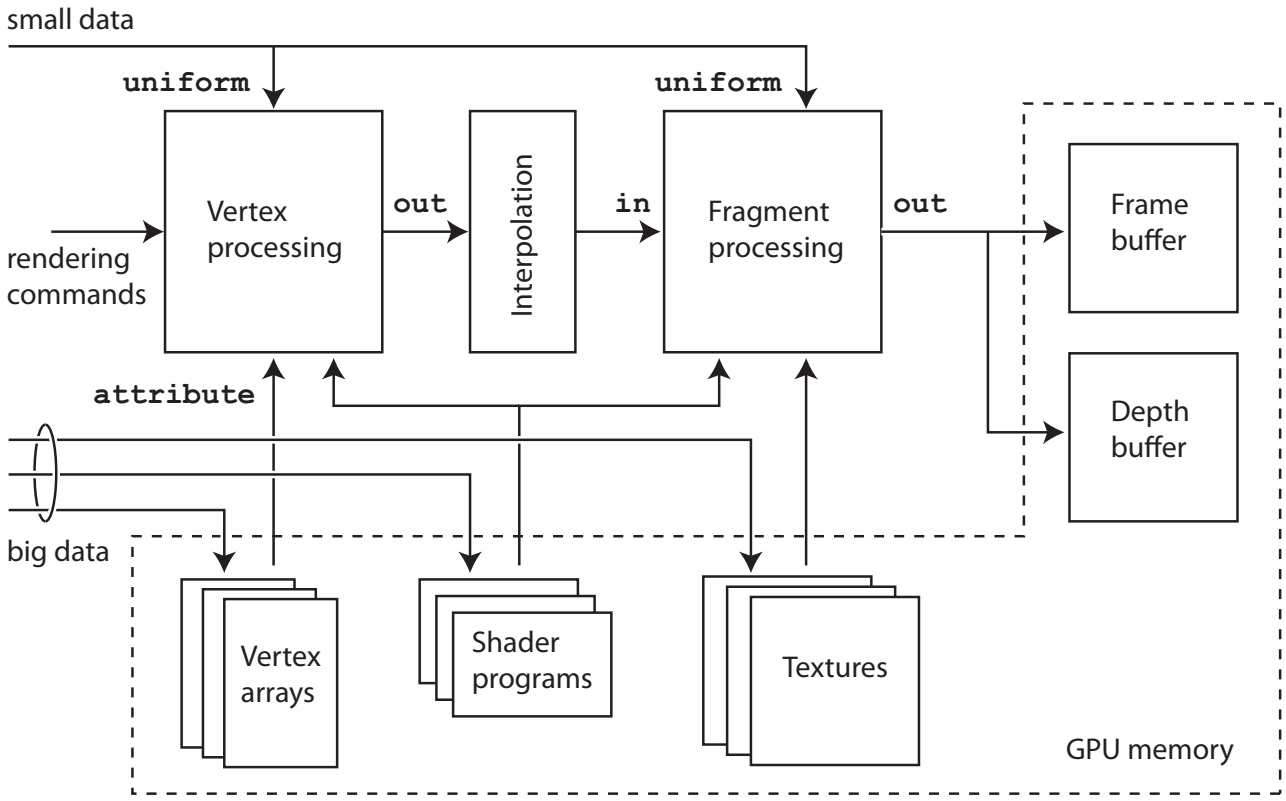
OpenGL and GPU programming

– a mental model for programmers

Stefan Gustavson (stegu@itn.liu.se) 2015-04-27

It could be argued that a good graphics API should hide the low level details of how a GPU works, to relieve programmers from the complexity of the implementation and allow them to focus on higher level tasks. Many APIs for 3D graphics try to do exactly that, with varying degrees of success, but there are still popular and very useful APIs that leave the low level details fully exposed. One such API is OpenGL. Programming in OpenGL requires considerable insight into how 3D graphics is rendered on a modern GPU, and programmers need to know quite a lot about hardware issues to make good use of OpenGL. The advantage is that OpenGL allows you to do anything you want that is within the capabilities of the hardware, and to do it in the most efficient manner. More user friendly APIs make it easier to do simple stuff, but they can also make it unnecessarily hard, even impossible, to do exactly what you want. We think programming at the "raw" OpenGL level provides a good understanding of what a modern GPU can do, and looking back, the single 3D graphics API that has remained relevant over the past decades is OpenGL. It may not be perfect, but it's good, stable, useful, fast and available on many different platforms.

OpenGL can be confusing, and textbooks often fail to make a clear enough connection between code and the GPU, between software and hardware. This document is an attempt to do just that. Using a mental model of the GPU, we hope to shed some light on how OpenGL commands and data structures connect and interact with the hardware. The model of the GPU presented here takes a few shortcuts and hides some details, but it is reasonably simple and easy to understand, and it's a close enough approximation to be a useful mental model for a programmer.



A mental model of a modern GPU. The rest of this document will explain the details.

Big data is pre-loaded to GPU memory

OpenGL is considered a “direct rendering” framework, meaning that each triangle is rendered by direct request from the programmer, in the order that the rendering commands are issued. However, this does not mean that all data is sent to the GPU for every frame. The most common and most efficient way of rendering with OpenGL is to pre-load bulky data to the GPU memory and try to keep it there between frames. Such bulky data are vertex arrays, texture images and shader programs, all of which are uploaded to GPU memory by separate commands, identified by an integer ID number that is assigned at the time of the data upload, and later used for rendering by referring to those ID numbers. ID numbers are optionally assigned also for render buffers. With proper setup, rendering becomes a matter of changing only small things like transformation matrices between frames, and sending rendering commands with ID numbers to the GPU to render data that is already stored in GPU memory. This makes such rendering fast and efficient, relieving the CPU from sending lots of data to the GPU each frame and leaving it mostly free to perform other tasks.

The rendering pipeline is programmable

Since a few years back, all processing of vertex and pixel data in a GPU is programmable by the use of shader programs. Shader programs are written in a special shader language and compiled for the GPU at hand. OpenGL accepts shaders written as text in the reasonably high level language GLSL, “OpenGL Shading Language”. The OpenGL graphics driver (the CPU portion of OpenGL) compiles the shader from GLSL source text to a binary shader program that can be run by the particular GPU that is installed in the computer. This on-the-fly compilation makes OpenGL shaders very portable and scalable, and also forward compatible in the sense that even old shader programs will be able take advantage of new and improved GPUs when they are run on such systems. Compilation of a GLSL shader is quick, but it's performed by the CPU, and it can't be done for every frame. Shaders are pre-compiled and uploaded to the GPU before rendering, and activated during rendering by referring to their ID numbers.

Small data is sent to the shaders

Large chunks of data should be pre-loaded in GPU memory whenever possible, but small data can easily be sent to the GPU for each frame during rendering. One way of doing this is by declaring *uniform variables* in shaders. Such variables can be set from the CPU, and they can be of any type supported by GLSL: transformation matrices, vectors and scalar values. Things that are typically specified as uniform variables are transformation matrices, information on light sources for shaders that compute lighting, color values and other material parameters, and time dependent information for shader-based animation.

Output is sent to render buffers

All rendered output is sent to render buffers in GPU memory. The traditional and most common format for these render buffers is one four-channel RGBA with 8 bits per channel, and one Z buffer with 24 or 32 bits per channel, but there are many other useful choices for the format, and output can actually be sent to several buffers at a time. We will not go into details on how this is done, but we will mention one important implication: because the rendered output resides in GPU memory, a rendered image can be saved and used as a texture in a second rendering pass. This makes it possible to perform shadow mapping and to perform 2D image effects by post-processing. Modern OpenGL applications make heavy use of such multi-pass rendering.

A brief introduction to GLSL

The shader programming language in OpenGL, GLSL, was created for the single purpose of writing short programs to process vertex and pixel data for graphics. Because of this, it is a fairly small and simple language that is reasonably easy to learn. Its syntax is similar to that of C, C++ and Java, but GLSL has data types, operators and functions that are tailored to the task at hand. Here, we will not present an overview of the entire language. There are proper books and more ambitious tutorials for that, and lots of online documentation. Instead, we will present the basics by showing a few examples of simple shaders and commenting on them.

A complete shader program must contain at least two shaders: one *vertex shader* and one *fragment shader*. The vertex shader is responsible for transformation of the vertex data to make the triangles appear in the right place on the screen, and the fragment shader is responsible for setting the color of each pixel. The smallest useful vertex shader and the smallest useful fragment shader would be the following pair:

Vertex shader

```
#version 330
layout(location = 0) in vec3 Position;
void main() {
  gl_Position = vec4(Position, 1.0);
}
```

Fragment shader

```
#version 330
out vec4 Color;
void main() {
  Color = vec4(1.0, 1.0, 1.0, 1.0);
}
```

This shader pair results in triangles being painted at exactly the (x,y,z) coordinates specified in the vertex array without any transformations, and every pixel is painted in a solid white color. While not terribly interesting, it's still a perfectly valid mode of rendering.

The first line, `#version 330`, tells the GLSL compiler that we are using version 3.30 of the language. This line is a required part of any GLSL shader. Different versions of OpenGL treat the input and output a little differently, and there are also some differences in what data types and functions you can use in each version. Between versions 3.30 and above the differences are fairly small, but version 3.30 made some significant changes compared to previous versions. We do not recommend using earlier versions if you have a choice, but sometimes you don't have that choice. The most notable example is WebGL, which as of this writing (April 2015) supports only an older and less capable version of GLSL. You will also find plenty of examples using the older syntax. Books and online resources have not always kept up with the recent rapid development.

Note that the input to the vertex shader (`in vec3 Position`) is a *vertex attribute*, data contained in the vertex array. The vertex shader needs to know the layout of the data in the vertex array, and conversely, the vertex array needs to be specified in the format that the vertex shader assumes. This is a tight coupling between the data format and the shader program that is somewhat undesirable, but OpenGL is focused on efficiency, not flexibility, and allowing any leeway in terms of data format or data layout would require the GPU to spend considerable time reshuffling data or converting between data types during rendering.

The only required output from the vertex shader is that you should set the pre-declared variable `gl_Position` (a 4-element `vec4`) to the desired position of the processed vertex. Because we have a 3-element input vector `vec3 Position` from our vertex array, we add a fourth component with the constant value `1.0` to make it a homogeneous coordinate vector, but other than that, the coordinate is passed through unmodified.

The output from the fragment shader is a 4-element vector: `vec4 Color`. For convenience, OpenGL associates the first output variable (in this case the only one) with the RGBA output that is sent to the render buffer. This is the most common way of storing the output. You need to connect your output variables explicitly to render buffers only if you want to render to multiple buffers, and we won't go into those details here.

A more interesting vertex shader would need to perform some kind of transformation, and that can be accomplished by a 4x4 matrix:

Vertex shader

```
#version 330
layout(location = 0) in vec3 Position;
uniform mat4 M;
void main() {
  gl_Position = M * vec4(Position, 1.0);
}
```

The fragment shader can remain the same for now. This vertex shader uses a single 4x4 matrix `mat4 M`, but one matrix is enough to perform any linear transformation of the vertex positions, and also a perspective transformation by using homogeneous coordinates. If the fourth component of the output vector (`gl_Position`) is anything else than `1.0`, the vector is normalized after the vertex shader to perform the perspective division. This division is done in a fixed function step between the vertex shader and the fragment shader. Other fixed functions include the splitting of each triangle into pixels (samples, *fragments*), and the interpolation of output variables across the surface of each triangle.

The matrix is specified as a `uniform` variable, and it is set before the shader is used by calling OpenGL functions in the CPU program. The CPU needs to know the name of the variable in the GLSL shader code to find its address and set it to a value. This is an undesirably strong connection between the CPU code and the GPU shader code, but once again, GLSL was designed for speed, which caused some compromises in its design and some convenience to be sacrificed.

A more interesting fragment shader requires at least some data to be sent to it from the outside. The color can be specified by a uniform variable, but that would still paint a constant color for all pixels. To have varying values across a triangle, you specify vertex attributes and send them along from the vertex shader to the fragment shader. Values are interpolated across the surface of the triangle. Let's add vertex colors to our vertex array, and specify them as a second input to our vertex shader. We also need to pass data along to the fragment shader to have them interpolated. This is performed by setting an `out` variable in the vertex shader and reading a corresponding `in` variable in the fragment shader.

Vertex shader

```
#version 330
layout(location = 0) in vec3 Position;
layout(location = 1) in vec3 vertexColor;
uniform mat4 M;
out vec3 interpolatedColor;
void main() {
  interpolatedColor = vertexColor;
  gl_Position = M * vec4(Position, 1.0);
}
```

Fragment shader

```
#version 330
in vec3 interpolatedColor;
out vec4 Color;
void main() {
  Color = vec4(interpolatedColor, 1.0);
}
```

Variables specified as `out` in the vertex shader must be exactly matched by `in` variables in the fragment shader: they must have the same type and the same name. Any mismatch between `out` and `in` variables in the vertex and fragment shaders results in a compilation error.

Some more details of GLSL deserve to be mentioned. Some matrix and vector data types were introduced above without much explanation. As you can see from the example, a matrix multiplication can be performed by the multiplication operator, which is very convenient. This requires that the dimensions of the matrix and the vector match: a 4x4 `mat4` can be multiplied by a 4x1 `vec4`. A `mat4` can also be multiplied with a `mat4` to create composite transformations.

The most important and most widely supported numeric data types in GLSL are the following. For a full list, refer to the documentation for the version of GLSL you are using.

```
float int          // scalar values
vec2 vec3 vec4     // floating point vectors of 2, 3 and 4 values
ivec2 ivec3 ivec4  // integer vectors
mat2 mat3 mat4     // floating point matrices: 2x2, 3x3 and 4x4
```

Vector types and scalar types can be mixed in mathematical operations. This has the effect of performing the same operation on each component of the vector. Performing operations between vectors of the same length results in operations being performed between corresponding elements.

Other than matrix multiplications, most familiar operations from 3D vector math are supported. A scalar product (“dot product”) between two vectors of the same length is performed by the function `dot()`, and a vector product (“cross product”) between two 3D vectors is performed by the function `cross()`. The length of a vector (the square root of the sum of the squares of its components) is computed by `length()`, and a vector can be normalized to unit length by the function `normalize()`.

Individual components of a vector can be set and read by using a “dot notation”. The first two components of a 3-element vector `vec3 Position` can be written `Position.xy`, and if you want to swap the order of x and y you can write `Position.yx`. Lots of swapping and duplication operations between components can be written like this, and the dot notation is often somewhat abused by programmers, making some GLSL programs hard to read. The dot notation is often called “swizzling”. It's a nonsense name, but it's useful to know. The components can be referred to by the letters `xyzw`, but also by `rgba` to make it easier to read operations on color values. A third option is `stpq`, which is provided to make it possible to more clearly separate texture coordinates (often denoted s,t) from vertex coordinates.

To make a longer vector from one or more shorter vectors, you use a constructor and supply enough data to fill all elements of the new vector. To make a shorter vector from a longer vector, you select the components you want by a dot notation:

```
vec3 Position = vec3(1.0, 2.0, 3.0);
vec4 hPosition = vec4(Position, 1.0);
vec2 xyPosition = Position.xy;
```

And, finally, the dot notation can be used to the left of the assignment operator as well:

```
Position.z = 0.0;
Position.xy = vec2(0.0, 1.0);
```

For a more thorough introduction to GLSL, there are other sources than this short presentation, but hopefully this is enough to get you started. You will now be able to read and learn from examples, and you are better prepared to read the official documentation to learn more.