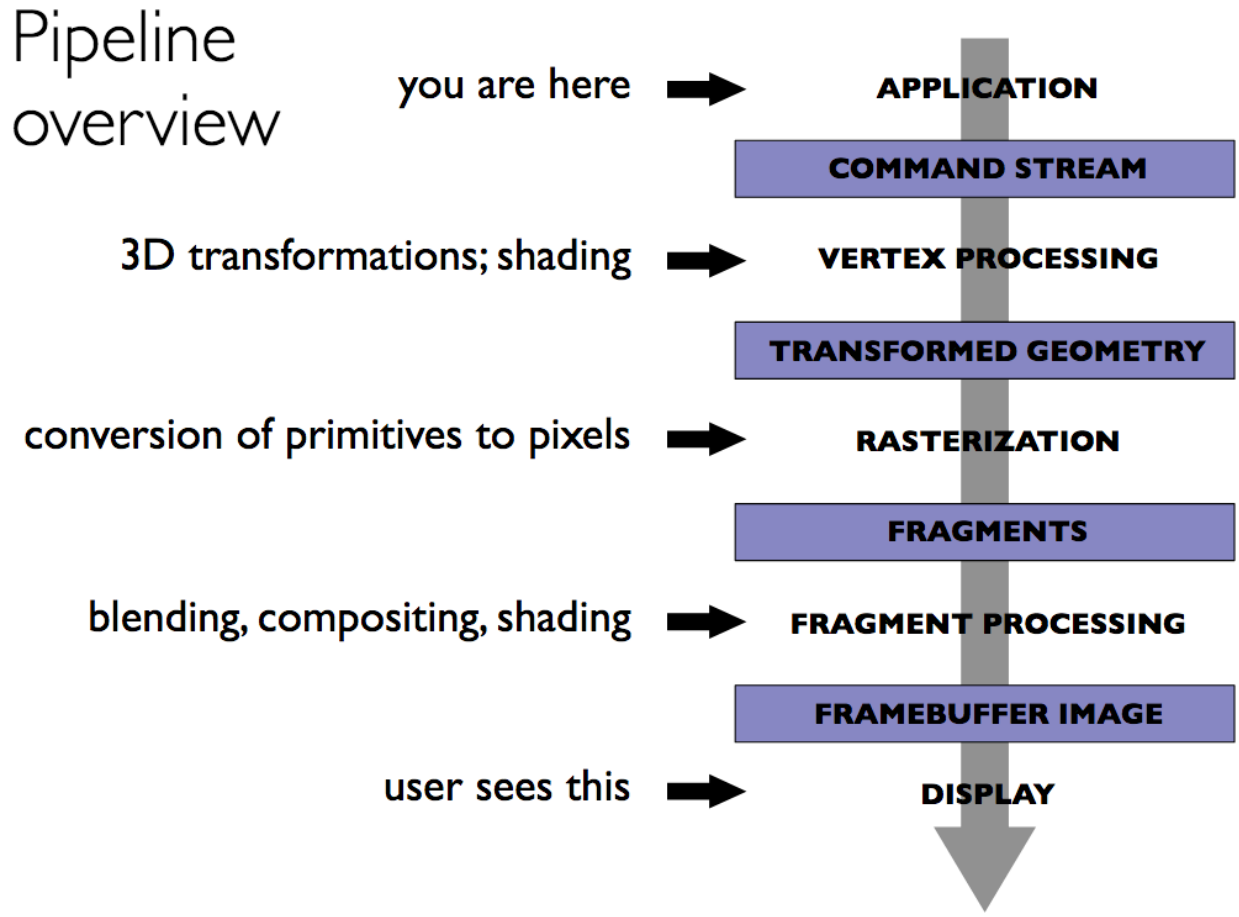
Lecture/Lab 20 - WebGL and GLSL - Notes

Using OpenGL amounts to programming a second computer. It's probably unlike any computer you've programmed before. Learning to program a new computer can be overwhelming and involve a lot of mutually-dependent information that must be tackled more or less at once.

Review: The Graphics Pipeline

It's important to keep in mind the conceptual framework inside which we're operating - **The Graphics Pipeline**:



OpenGL is one implementation of the graphics pipeline.

OpenGL/WebGL is one example of a hardware-accelerated implementation of this pipeline. What that means is that the steps listed above are performed by

- you, in the application code (Javascript, in our case)
- the graphics library on the GPU
- you, via custom-written mini-programs (shaders) that run on the GPU

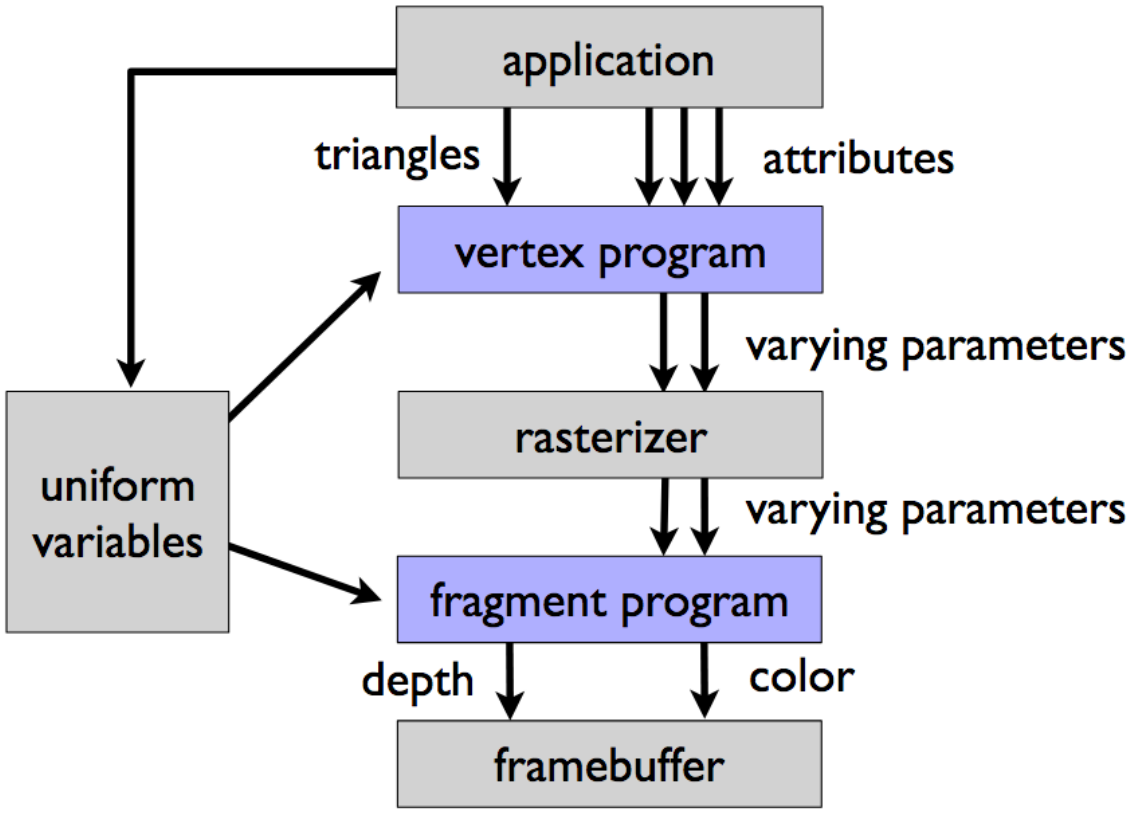
Here's a basic overview of how OpenGL accomplishes each step of the graphics pipeline:

The basic implementation is as follows:

- You supply the command stream to send geometry to GPU RAM
- You supply a **vertex shader** to do the vertex processing. This is a miniature program that runs for each *vertex* of geometry you provided. The job of this program is to position the vertices in normalized device coordinates (*"clip space"*, in GL parliance). It should also compute any additional information about the vertex that will be needed further down the pipeline.
- GL performs the **rasterization** step for you: this includes **clipping** geometry that's outside the view volume and generating pixel-shaped **fragments** that correspond to each pixel that a given **primitive** overlaps. Values known at each vertex are **interpolated** to determine their value at each fragment, which is passed on to the next stage.
- You supply a **fragment shader** that is run for each *fragment* generated by the rasterizer. This determines the color of the fragment based on its position and whatever other data you've passed along to the fragment shader.

The **vertex shader** (or vertex program) and **fragment shader** (fragment program) are written in a domain-specific language called **GLSL** - the GL Shader Language. This is a C/C++-like language with a bunch of nice built-in features and types that make for concise code that deals with vectors and matrices up to 4 dimensions. We'll see some of these features in action as we get into writing shaders.

In the above, I've bolded terminology that we've already discussed. Now, I'll introduce some more terminology to talk about some of the specific moving parts in the above process. Here's a schematic that shows how these things interact:



attributes are vertex-specific values that are passed into the vertex shader. This includes things like the vertex's position, normal, color, and texture coordinates.

buffers (buffer objects) are the typical way to store attribute data. These are basically just arrays that you carefully tell GL how to interpret as vertex data. These can take many forms, but one approach is to have a **vertex buffer object** (VBO) that stores each vertex's position and an **index buffer object** (IBO) that stores triangle indices into the VBO. It's also possible to pack multiple vertex attributes into a single array.

uniforms (uniform variables) are specified in the application program and are provided as input to both the vertex and fragment shader; these values are *uniform* across the entirety of the geometry being rendered, hence the name. Perhaps the most common example of a uniform variable is the 4x4 model-view-projection matrix that transforms vertices from model to clip space.

varyings (varying parameters) are values computed per-vertex by the vertex shader which are then automatically **interpolated** by the rasterizer to determine their value at each fragment, so that their value *varies* across the primitive. The interpolated values are then made available to the fragment shader. Often these will be derived from attributes, such as normal, color, or texture coordinates.

The devil is in the details

This is, conceptually, what I want you to know about how to work with OpenGL. When you dive into the code, you'll see that GL code involves a lot of obtuse-looking boilerplate. Each conceptual step takes several calls to GL functions. For example, the shader programs are stored in your program as strings, which then have to be compiled, linked, and combined into a program that is then enabled

on the GPU. Creating a VBO and sending it to the GPU requires creating the buffer, binding it, pointing it at the right data, and telling OpenGL to interpret as vertex data.

Today's lab activity will help you dive in and get a feel for how to use this machinery, but rather than get into the details of every single sub-step described above, we'll hide some of that away in helper functions so you can focus on linking the code to the concepts.