**INFO-6028 Day 5, VAOs, VBOs, and Index Buffers, oh My!**



Yeah, yeah, old man, you’re not the boss of ME!!!

But here’s The Deal: We want to draw a bunch of objects on the screen. Different objects. Cows, bunnies, TIE Fighters (yes, that’s the *correct* spelling – it’s an acronym – look it up, people!), and so on.

**Vertex Array Objects (VAOs) are your friend. Really.**

We could do it the short term, painful way, which will get more painful later, and will be replaced with VAOs, causing a giant re-write of the code, lots of annoyance, and likely great confusion.

OR, we could look at VAOs right away, structuring the code to use them right “off the hop”, as they say.

But here’s the thing. Can I be completely honest with you? They won’t make any sense. At least not at first. You will be muttering to yourself “why didn’t I take COBOL?” and “what the heck is going on?” and “this has *got* to be the *stupidest* way to do this...”.

But my dear, sweet, innocent students, that isn’t the case. Sure, there will be short term confusion, but it will lead to sun-lit field of sunshine, rainbows, and unicorns passing out beer (or wine, or whatever nice food/drink you like... skittles? I don’t know... something good).

The issue is that VOAs are tied closely with shader objects, specifically shader programs, but they aren’t actually “connected” to them. Instead, they are connected to the *state* that OpenGL is in while a shader is in use. They also, like so many things in OpenGL, have a name that is confusing for newcomers. The name *does* make sense, but perhaps not at the moment.

We will be looking at VAO (Vertex Array Objects). They are SUPER handy for all sorts of things, but to use them – or to make sense of what they are and what they are for – we need to know a bunch of other stuff, first.



We are going to use “Indexed Primitive” models, which are super common (PLY are one type). This means that the vertices are all listed in an array, by index, and the “primitives” (aka “triangles” for us) are listed separately, referring to the vertices by index.

Easy peezy.

This is so common that OpenGL, DirectX (and other APIs) support these directly.

In OpenGL, the vertices are copied into a Vertex Array (or “Vertex Buffer Object” or “VBO”) and the indices are copied into a separate buffer called an “Element Array” buffer (but everyone calls them an “Index Buffer” so that’s what I’ll call them – also, DirectX calls them an “index buffer”).

The shader is “fed” these vertices through the “Vertex Attributes”, which consist of an “array” (referring to the variables in the shader – we have two variables now (vCol and vPos), but could have many more – and the “pointer” which allows the shader to “point to” the parts of the vertex in the “Vertex Array” (VBO).

The only thing new here is the “Element Array”/”Index Buffer”, but interestingly, we don’t have to worry to much about that – the index buffer simply refers to a particular vertex buffer, and doesn’t actually connect to the shader at all; it’s really that OpenGL *indirectly* feeds vertices into the shader through the index buffer rather than right from the VBO.

Here’s the snag with all that....

What if you have more than one VBO?

Why would you want that? Well, perhaps you’d like to draw more than one thing?

If you did, then you could do one of two things:

* Place all the models into the *same* VBO (and possibly index buffer), but keep track of the offsets into the buffer,
  + This would mean you only have to set one vertex array, ever.
  + BUT you would have to keep track of where the models are in the buffer (and how big they were – how many vertices)
  + And this would assume all the models have identical vertex attributes
* Or, place each model into its own VBO (and corresponding index buffer)
  + You wouldn’t have to worry about where the models start in the buffers (always at the start)
  + BUT, you would have to deal with setting the vertex attributes for each buffer...
  + AND switching them at run-time

In older cards, switching buffers was expensive, so you’d often do the 1st way (all models into one buffer), but while there *is* a cost to switching, it’s almost nothing now.

Note that you could also do a combination of these two options: Multiple buffers with multiple models inside each, say if you have models with a different vertex layout (like 3D portions of your application, and 2D GUI portions).

There’s also another “catch” that this vertex layout is specific to *each shader that you use* AND that the shader *doesn’t* keep track of any of this.

What does this mean?

It means that if you want draw from more than one buffer, you have to set the vertex attributes *every single time you draw from a different buffer* for *every shader you use*. And this happens at run-time, in your drawing loop.

**There must be a better way.**

There is. It’s called a “Vertex Array Object” or “VAO”. This is *so* helpful, that it’s not incorporated into OpenGL ES 3.0 (the mobile version of OpenGL).

The picture attempts to show what a VAO “looks after”, which is:

* The particular VBO(s)
* The particular Index Buffers (aka Element Arrays)
* The vertex attributes
* For an particular shader

How do you use it?

That part is pretty simple, actually. You simply create and enable (“bind”) the VAO, then set up all the things in the list above. The VOA “remembers” the state you set all these things to. When you set bind the VAO later, it restores all those settings for you!

Catches: Whatever “things” are bound *last* are the ones it’s going to “remember”, so be careful.

You need a new one for each shader (and for each “context”, which is what the shader is tied to – we will only use one context, ever, so you don’t need to worry about that, but *technically* it’s tied to context as well).

The best of all? You only need one value, the VAO “name” (or number).

**So what’s the Big Hairy Deal, Feeney?**

Well, we are going to be someone pre-emptive and start making a simply “shader manager”, which will look after “all things shader related”, including loading the shader source, handling errors and “uniforms” (we’ll look at those in a moment), and this will be tied to the VAO.

Wait, *why* are the *shaders* tied to the VOA? Well, because the “vertex array” (the connection between the vertex buffer and the shader) requires a working and loaded shader to “bind” to.

Put another way: it doesn’t make sense to enable a particular VAO if there isn’t a shader there to be feed the vertices. BUT, the VAO doesn’t “pay attention” to this, only enabling the connection to a particular shader – not necessarily the one you are currently using.

Why do this? Well, because it’s very likely that the reason you are using another shader is that you want to the vertices to be handled in a different way, **so each “vertex attribute” layout would be specific for each, individual instance of a particular shader**.

What does *that* mean? Well, it means that if you have one VAO (VBO+Index+Vertex Attribute) for each model (i.e. your buffers only hold *one* model), then you have to remember that each VAO is tied to an individual shader. This is fine if you only use one shader (we are likely only using one), but if you have multiple shaders, or multiple shaders in use per model, then you have a vast combination of VAO to shaders.

To avoid this, and still use a VAO, you could load all your models into the same VAO, but “back” them one after another. The “catch” here is that you have to:

1. Keep track of where they start and end *and* make sure they don’t overlap
2. Have to have them all the same vertex layout (which is likely the case, anyway)
3. Know how big ALL of them are BEFORE you create your buffers:  
   You *can’t* change the *size* of a buffer after creation. You can change the contents, but not the size, so if you want a bigger buffer, you have to create a new one (this is how the std::vector works BTW).

The sequence for setting up a VAO:

1. The Vertex Array Object:
   1. glGenVertexArrays()
   2. ***glBindVertexArray()***
2. The Vertex Buffer Object (VBO):
   1. glGenBuffers() (for the vertex buffer)
   2. glBindBuffer(GL\_ARRAY\_BUFFER...)
   3. glBufferData(GL\_ARRAY\_BUFFER...) (copying the vertex information into buffer)
3. The Index Buffer (“Element Array Buffer” in OpenGL speak):
   1. glGenBuffers() (for the index buffer)
   2. glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER)
   3. glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER...) (copying index info. into buffer)
4. The Vertex attributes. For EACH vertex attribute:
   1. glGetAttribLocation()
   2. glEnableVertexAttribArray()
   3. glVertexAttribPointer()
5. Then disable everything (if you want):
   1. ***glBindVertexArray(0);***
   2. **glBindBuffer(GL\_ARRAY\_BUFFER, 0);**
   3. **glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);**
   4. You can also call glDisableVertexAttribArrays() on all the vertex attributes, too

**Note:** The order that you do the glBindVertexArray() calls is ***CRITICAL***, as OpenGL is a giant state machine. So *WHATEVER BUFFERS, etc., ARE SET UP WHILE THE VAO IS BOUND IS WHAT THE VAO WILL REMEMEBER!!*

**Huh?** If you 5b) and/or 5c) and/or 5d) (in the above list), so UNBINDING the vertex/index buffers OR you disable the vertex array information BEFORE you unbind the VAO, it will REMEMBER THAT (that the buffers, etc. ***AREN’T*** bound, which is NOT what you want, likely).

So you HAVE to disable the VAO ***BEFORE*** you unbind all that stuff.

**Sequence for drawing, using a VAO:**

Trivial:

1. **glBindVertexArray( VAO\_ID );** 🡨 enable VAO (and everything else)
2. glDrawElements( GL\_TRIANGLES, ...);
3. **glBindVertexArray(0);** 🡨 disable VAO (and everything else)

Bam! That’s why you use VAOs. They *seem* like a big pain to set up, but in fact you ***have to do all that set up anyway each time you do a draw call*** no matter what (i.e. if you *don’t* use VAOs, then and you have multiple vertex+index buffers and/or vertex attributes then you have to do this *anyway*.

The “deal” with VAOs is that OpenGL will *remember* the entre complicated set-up you used, allowing only a single call to glBindVertexArray() at draw time.

Note that you can avoid do all these calls if you use only a *single* set of buffers, *single* shader, *single* set of vertex attribute arrays, and never change it. But where’s the fun in that?!?