A Hitchhiker's Guide to OpenGL

Bart van Blokland

Department of Computer and Information Science, NTNU

Revision 3

Contents

Chapter 1

A bird's eye view of OpenGL

1.1 Introduction

OpenGL is an API for rendering images, accelerated by specialised graphics processing hardware.

The API was initially released in January 1992, and has since been updated regularly. Over the years, the overall design has seen major changes. Additionally, a significant number of obselete or inefficient methods have been deprecated or removed entirely.

OpenGL is intended as a relatively thin layer on top of the hardware its execution depends upon. As such the support for a revision of the OpenGL specification depends on the capabilities of individual graphics card models.

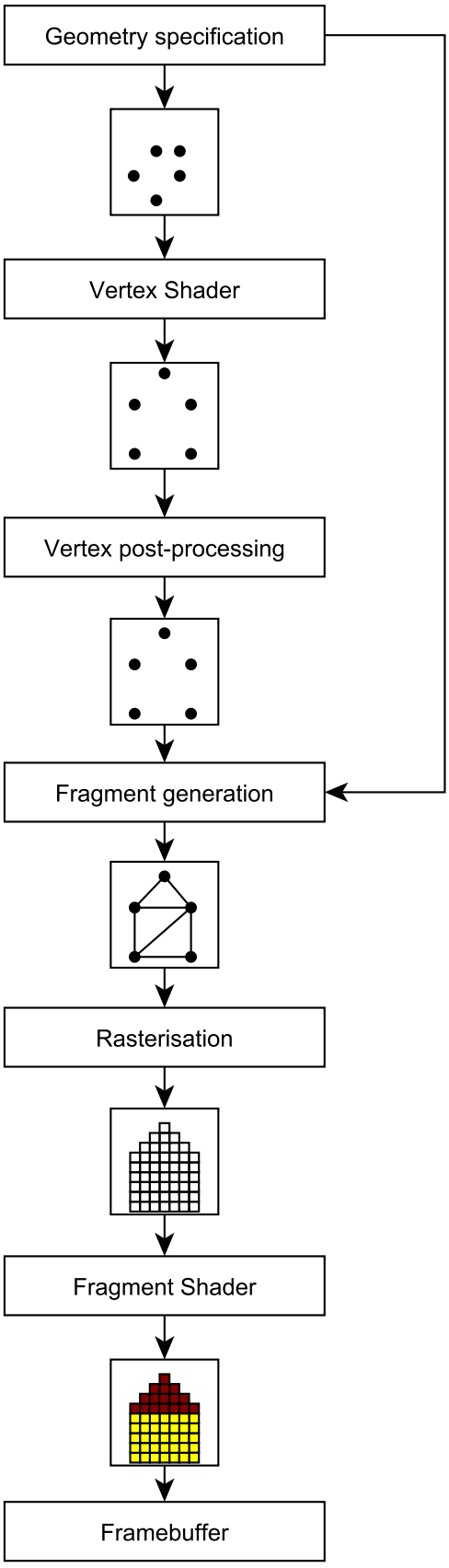
The overall design of the API is in principle a massive state machine. You change its state by calling functions on it. As state transitions are intended to be atomic operations, it is by design meant to only be used from a single thread.

I would love to show a fancy diagram here with all possible states OpenGL can reside in, but unfortunately there are only so many carbon atoms on this sheet of paper (or terabytes on your hard drive, depending on where you are viewing this guide).

Fortunately, unless you call OpenGL functions in a very strange order there aren't really any notable restrictions as to which functions you can call at any point in time. For instance, it's entirely fine to load in some 3D models while in the process of rendering a frame (useful for loading screen in games).

1.2 The OpenGL Pipeline

The process of converting a 3D model into pixels on a screen is not a simple one. OpenGL therefore uses the concept of a *pipeline* to model its operation. A diagram of this pipeline showing each stage's output is shown below:


Geometry specification
Vertex Shader
Vertex post-processing
Fragment generation
Rasterisation
Fragment Shader
Framebuffer

Let's go over each of these stages in detail:

Geometry Specification Drawing in OpenGL starts with the specification of a scene. You define a set of geometric primitives (most commonly triangles), and tell OpenGL to draw these. Every primitive will then enter the pipeline and is processed in various ways along each stage.

During the initial stages, however, OpenGL only operates on the vertices of the geometry specification.

Vertex Shader A Shader (as we will see later in this guide), is a little program that runs on the GPU. The Vertex Shader is a Shader which is responsible for applying transformations and a projection.

Transformations include moving, scaling or rotating vertices around the scene so that they are in desirable positions. This for instance allows objects to be animated by incrementally moving them around.

You can consider the projection stage to be the "camera distortion operation". It warps the world as if the world is seen through a camera lens. You define the operation of this lens yourself.

Vertex Post-Processing Most notably performs clipping so that nothing outside the bounds of the scene is rendered.

Additional operations can be performed on the input geometry to make rendered objects look more detailed or more aesthetically pleasing. These are not in the scope of this guide, however.

Fragment Generation From the processed vertices, using the geometry specification, we can deduce which vertices are combined into which geometric primitives (again, mostly triangles).

Rasterisation We can now convert the primitives located during the previous phase into pixels on screen using rasterisation algorithms. These are discussed during the lectures.

Fragment Shader In this stage, each pixel is given its colour by running another Shader, called the Fragment Shader. This can include lighting effects to indicate roundness, as well as textures (images projected on to the primitives).

Framebuffer The final step is writing the rendered pixels to the Framebuffer. The Framebuffer is sent to the monitor for displaying the results upon the completion of the frame.

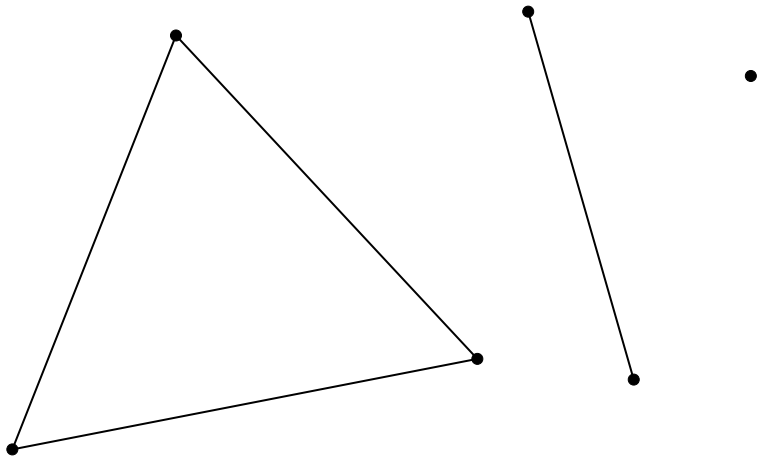
Don't worry if you don't really understand many of these things; we will be seeing each of these stages in more detail as we go along throughout this guide. What is important to take away here is that OpenGL performs its rendering in multiple stages, and it is your task to provide input and configure this pipeline.

Chapter 2

Introduction to Drawing

2.1 Defining Geometry

OpenGL can in principle only draw 3 kinds of shapes, known as "primitives"; triangles, lines and points.



Of these three, the triangle is the one that's most commonly used in practice.

Why only three basic shapes? What about rectangles, circles, ellipses, spheres, and so on?

There are two reasons for this. First, as you will see in the lectures rasterising and rendering triangles is very easy and can therefore be performed very cheaply on the graphics card. Second, you can approximate all the shapes I listed using triangles anyway.

Looking at the primitives, you may notice all of them consist of at least one vertex. In OpenGL, vertices are used to define the *geometry* of a scene. That is, the collection primitives that are drawn during a single frame. As you'll see, drawing in OpenGL essentially consists of handing a list of vertices to OpenGL and issuing a draw command

in which you specify which primitive type you'd like to draw. OpenGL will then run through your list, taking the number of vertices for each primitive at a time, and draws them (3 for vertices, 2 for lines, etc.).

2.1.1 Specifying vertices

In OpenGL, defining the aforementioned "vertex list" starts with creating something known as a *Vertex Array Object* (VAO).

The simplest way to use (and think of) a VAO is to consider it to represent a single "object" in your scene. What the object *is* depends on what you define it to be. For instance, if you want to render a car, the car's body could be stored in a single VAO, and the tires in a separate one. It's also possible to store the entire model in a single VAO.

In practice, a VAO is little more than a metadata object. It does not actually store any vertex definitions. The vertex definitions themselves are commonly stored in dedicated buffers, known as *Vertex Buffer Objects* (VBO's).

The contents of a VBO are generally stored on the memory bank (VRAM) of your graphics card. You therefore have to explicitly transfer data to it before you can use it to draw things.

The sole purpose of VBO's is to hold data. They do not know how their contents are formatted, what their contents represent, nor where their contents are supposed to be used. All of that information is held by the Vertex Array Object.

We'll now look at how to define one of these Vertex Array Objects, followed by how to define and fill a Vertex Buffer Object, and set both up for rendering.

One note on the functions listed in this chapter; the OpenGL documentation commonly lists rather cryptic data types for each of the parameters. For the sake of being able to understand them, I changed them to those you will be most commonly calling these functions with. If you want the official specifications, I recommend using the official online documentation pages.

Creating and setting up a Vertex Array Object

The first step is to create a new Vertex Array Object using:

```
void glGenVertexArrays(int count, unsigned int* arrayIDs);
```

The arrayIDs parameter requires a pointer to a location where the generated array ID(s) can be stored. You are responsible for allocating enough space for these. If not you can cause data corruption or even crashes. The function is relatively easy to use when only allocating a single array; just create an empty unsigned int on the previous line (`uint array = 0;`), and pass a reference to it into the function using the & operator (`&array`). The ID of the VAO will be stored in `array` afterwards.

The first thing to note here is that you don't actually get a Vertex Array *Object*, but instead the *ID* of the generated VAO. As such you don't actually get to modify the object directly. Instead you use the ID to refer to the array whenever you want to do anything with it. You modify it through OpenGL function calls.

This is common to the design of all data structures in OpenGL: you only get references instead of complete data structures.

You may also have noticed that this function allows you to create multiple arrays at once. To avoid confusion I'd recommend just creating one at a time (setting count to 1).

If we want to add Vertex Buffer Objects to the Vertex Array Object, we have to ensure that the any configuration values are set *while the VAO is active*. You can imagine this process to be like opening a file; in order to add some lines of text (the VBO's) to a text file (the VAO), you first have to open it. As OpenGL is a state machine, this "file" will remain open until another one is opened in its place. This process is referred to as *binding*.

The function for binding a vertex array object is defined as follows:

```
void glBindVertexArray(unsigned int vertexArrayID);
```

Creating buffers

Now that we have created and bound a VAO, we can proceed with defining a Vertex Buffer Object to hold the vertices of our triangles. In order to set up a VBO and fill it with data, there is a particular sequence of functions you need to call. We'll be taking a look at these functions below.

The first step is to create the Vertex Buffer Object. This function call works identically to the one used for creating VAO's:

```
void glGenBuffers(int count, unsigned int* bufferIDs);
```

Note that even though the buffer is meant to hold data, you don't need to specify the size of your buffer at this point. OpenGL will perform these allocations behind the scenes as you fill or append data to your buffer.

Binding buffers

Like Vertex Array Objects, Vertex Buffer Objects are also required to be *bound* before they can be modified, and only one can be bound at a time.

The function for binding a buffer is:

```
void glBindBuffer(enum target, unsigned int bufferID);
```

The target parameter specifies which kind of buffer you would like to bind your buffer as. Most of these types are advanced uses that are not relevant for this course. You'll therefore usually want to pass in `GL_ARRAY_BUFFER` here.

The bufferID parameter is the ID of the buffer we would like to work with. This should be the buffer you generated using `glGenBuffers()`.

It is possible to "unbind" a VBO by binding a buffer with an ID of 0. However, it is not something that ought to be done in practice for two reasons. First, any subsequent time a buffer is going to be modified, it will have to be bound first anyway. Second, binding and unbinding buffers may incur additional overhead on the OpenGL driver implementation side.

Filling buffers

Now that we have created and bound our buffer, we can fill it with data. This of course requires that you already have some kind of data to fill it with. A general method for doing so is to allocate an array of floats (although other data types will work too):

```
                //   x    y    z    x    y    z    x    y    z  .. and so on
float vertices[] = {1.0, 3.0, 2.0, 5.0, 4.0, 3.0, 2.0, 6.0, 3.0};
```

In practice, this data usually originates from a separate file.

Note that by default, all OpenGL coordinates are mapped between -1 and 1 on each axis.

Once you have your float array ready to go, you can use the `glBufferData()` function to transfer the data to the GPU:

```
void glBufferData(enum target, size_t size, void* data, enum usage);
```

The target parameter has to match the target parameter you supplied in the `glBindBuffer()` call.

The size parameter is the size of your data array in bytes. You'll probably need to use the `sizeof(/* data type */)` function here, which returns the number of bytes a particular datatype occupies in memory. Note that `sizeof()` takes in a datatype as its parameter, such as `float` or `int`.

It is in some cases possible to pass the array into the `sizeof()` function directly to get the number of bytes occupied by the array's contents in memory.

For instance:

```
int someArray[] = {1, 2, 3};
printf("The size of someArray in bytes is \%i.\n", sizeof(someArray));
// Prints: "The size of someArray in bytes is 12."
```

However, whenever an array is passed as a parameter into another function, it is quietly converted into a pointer. This even happens when the function parameter has the array type. As a pointer only represents a memory address, the array dimensions information is lost. As such calling sizeof() on the array variable will give you the size of the pointer (usually 8 bytes on a 64 bit system), rather than the total size of the array.

It may therefore be beneficial to pass in the length of the array into the function as a separate parameter, so this confusing behaviour can be avoided altogether. In that case, multiply the length of the array with the size of the datatype of that array (e.g. `arrayLength * sizeof(float)`) to get the size of the array in bytes.

Next is the data parameter, which (surprise surprise) contains the data that should be copied to the GPU. Note that the type of this parameter is a `void` pointer. A `void` pointer in C means that you can supply a pointer to any data type you want. So for instance `float*` or `int*` are both valid parameter types here.

The usage parameter provides a hint what the purpose of the data is you're supplying. Based on this parameter, the OpenGL implementation may perform optimisations to a greater or lesser degree, if at all. It does not restrict how you can use the buffer.

For the purpose of this lab it's probably best to use `GL_STATIC_DRAW` here. The `STATIC` component indicates that the contents of the buffer are not expected to change often, if at all. The alternatives for `STATIC` are `DYNAMIC` and `STREAM`, which are intended for increasing modification rates to the buffer.

The `DRAW` component indicates that the contents of the buffer are intended for rendering geometry. The alternatives are more advanced uses outside the scope of this course.

Format Specification

You may have noticed from the glBufferData function specification that there was no requirement for defining the format of your buffer. OpenGL does not know whether you passed in floats or integers, nor does it know whether you specified x, y and z coordinates, or only x and y coordinates.

For this reason we have to set a *Vertex Attribute Pointer*. The Vertex Attribute refers to the buffer we have just created in this Vertex Array Object. In a way OpenGL considers every buffer to contain a list of vertex attributes. For instance, one buffer could contain the coordinates of the vertex, while another might have information on how well the vertex reflects light.

Here's the function for setting the Vertex Attribute Pointer:

```
void glVertexAttribPointer(
    unsigned int index,
    int size,
    enum type,
    bool normalised,
    size_t stride,
    void* pointer
);
```

Even more parameters than `glBufferData()`! Let's go over them.

The index parameter specifies which index the buffer should have as input to the Vertex Shader. Don't worry if that doesn't make sense right now, we'll come back to it in the section about Shaders. In a nutshell it tells OpenGL which ID you will use to refer to the particular buffer (or Shader input in general).

This is a number you can make up yourself, as long as it is between 0 and the OpenGL constant `GL_MAX_VERTEX_ATTRIBS`, and you use the same IDs for the same Vertex Attribute both in OpenGL function calls as well as your Shader source code.

Note that the values of OpenGL constants need to be queried. You can find out the value of the constant by using:

```
int maxVertexAttribs;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &maxVertexAttribs);
printf("GL_MAX_VERTEX_ATTRIBS: %i\n", maxVertexAttribs);
```

On the two machines I tested this on, this returned a maximum of 16 attributes.

The size parameter can either be 1, 2, 3 or 4. It defines the number of values per *entry* in the buffer. For instance, only specifying x and y coordinates per vertex means that `size` should be 2. If you specify x, y and z coordinates, `size` is 3, and so on.

`Type` defines the data type of the values in the buffer. This should match the type of values you passed in with the `glBufferData()` call. Here's some of the possible data types you can pass in here:

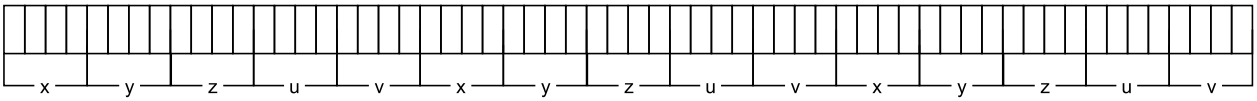
- `GL_BYTE`
- `GL_UNSIGNED_BYTE`
- `GL_SHORT`
- `GL_UNSIGNED_SHORT`
- `GL_INT`

- `GL_UNSIGNED_INT`
- `GL_FLOAT`

Performance tip: you should always choose the smallest possible data type possible that is acceptable for your data. For instance, if you have an index buffer which never exceeds 255 elements, you can make indices of the type `GL_UNSIGNED_BYTE`. Doing this can significantly reduce the memory bandwidth requirements of the graphics card, increasing performance.

Next up is the `normalised` parameter. It defines whether OpenGL should normalise the values in your buffer. In most cases you'll want to pass `GL_FALSE` here.

The `stride` parameter defines the number of bytes between each *entry* in the buffer. This may sound strange at first; if a buffer only contains coordinates, can't you just calculate these from the other parameters? The thing is that if you want, a single Vertex Buffer Object can contain multiple Vertex Attributes. For instance, you can pack both vertex coordinates and texture coordinates [1] in the same buffer:



In this case, the buffer contains 3 coordinate components (x, y, and z) of 4 bytes each, as well as 2 texture coordinates of 4 bytes each. The number of bytes from the first byte of an x coordinate to the next is 20 bytes. This is known as the *stride*.

If your buffer only contains a single entry type, you can pass in 0 here, and OpenGL will deduce the stride for you. However, like in the case of the buffer shown above, when multiple entry types are present you'll need to determine the proper stride yourself.

Finally, the `pointer` parameter defines the number of bytes until the first value in the buffer. Using the example from the buffer shown above, the x, y and z coordinates start at index 0, while the texture coordinates start at byte 12. If you only have a single entry type in your buffer, this parameter is usually 0.

Enabling the VBO's

Finally, we need to enable the Vertex Buffer Objects that should serve as input to the rendering pipeline. Inputs can be enabled with:

```
void glEnableVertexAttribArray(unsigned int index);
```

[1] In a nutshell, textures are images that can be mapped on to triangles. Texture Coordinates tell OpenGL what part of such an image to project on a given triangle. Since textures are (usually) two-dimensional objects, you can use two coordinates to define locations on the texture itself. The "u" and "v" names by which each axis is often referred to is the origin of the other common name for textures; UV maps.

The index parameter should correspond to the index you passed into the `glVertexAttribPointer()` while setting up the VAO. You need to call `glEnableVertexAttribArray()` once for every VBO you would like to use as input for rendering.

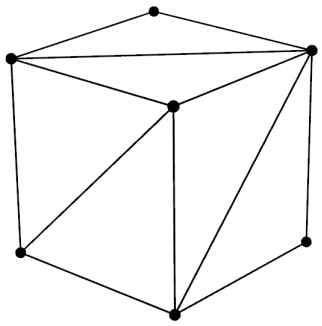
Calling `glVertexAttribPointer()` and `glEnableVertexAttribArray()` ensures the corresponding VBO's are enabled and set up correctly when you want to draw your VAO.

And that's it! Your VBO is now part of the VAO, ready to be used. However, there's one more step before we can draw its contents.

2.1.2 The index buffer

We now have a buffer which defines the coordinates of a number of vertices. However, we did not specify any information about how these are supposed to be combined into primitives. We can do this through the use of a special buffer called an "index buffer".

You might wonder why we have to specify an additional buffer to combine vertices into primitives instead of just using the vertices in the order they are defined in your VBO? To answer this, let's take a look at a cube made up of triangles:



Notice how most vertices are part of multiple triangles. The center one is even part of 4 of them. 3D surfaces tend to be "watertight" in order to appear convincing on the rendered image. As this requires using the same vertices several times over for different triangles, it makes sense to only define the vertices once, and combine them together by referring to their *index* in the VBO. If executed well, you can save quite a bit of memory usage.

Fortunately, the mechanism for creating and filling an index buffer is very similar to the way we set up VBO's. I'll therefore only outline the differences here.

First, you generate another buffer using `glGenBuffers()`.

Next, you bind the generated buffer, with as target `GL_ELEMENT_ARRAY_BUFFER`, as opposed to `GL_ARRAY_BUFFER`. The index buffer has a special "status" within the Vertex Array Object and thus has a separate buffer type.

Now we can fill the buffer with indices. They should be unsigned integers, unsigned shorts or unsigned chars (bytes). Just like with the VBO's you should create an array of these. The index of the vertices you defined in your VBO start counting at 0.

Finally, we call `glBufferData()` to copy the integer array into the index buffer. Note that the target should in this case also be `GL_ELEMENT_ARRAY_BUFFER`.

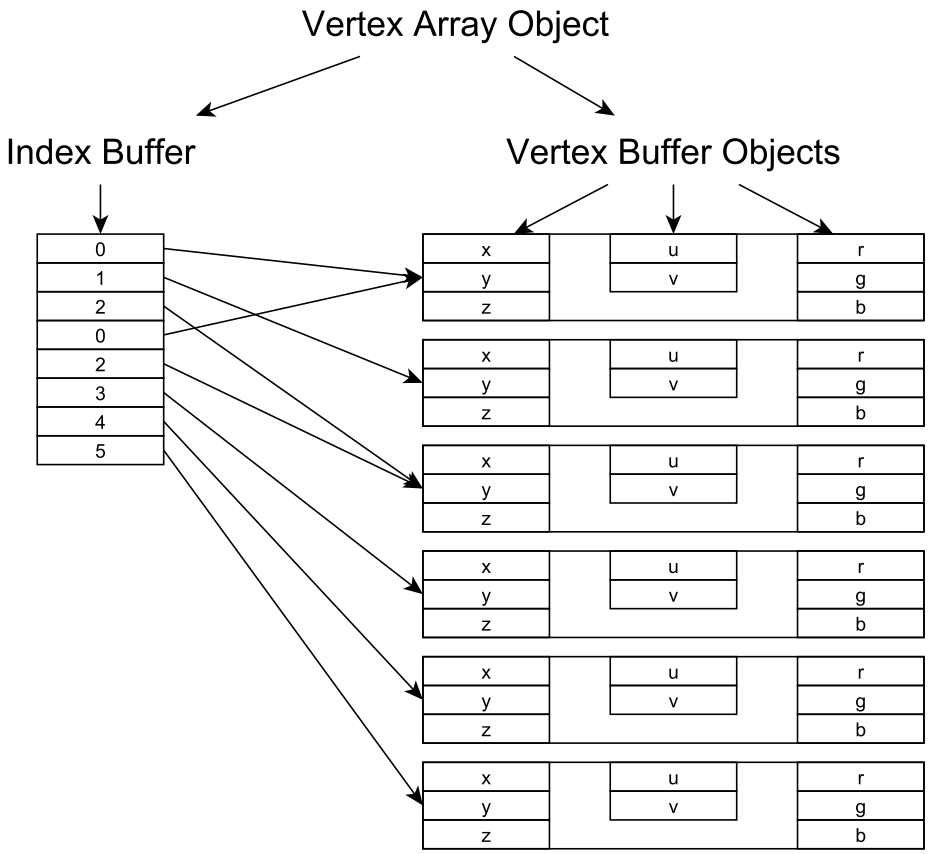
And we're done! We're now ready to draw our Vertex Array Object :)

Unlike the VBO's, you don't need to call `glVertexAttribPointer()` to set up your index buffer.

2.2 Drawing Vertex Array Objects

2.2.1 Overview

So where have we gotten up to this point? Here's a diagram showing the buffers we set up:



We have created a Vertex Array Object which consists of an Index Buffer and one or more Vertex Buffer Objects. By setting up Vertex Attribute Pointers, OpenGL knows where to look for vertex attributes, and groups them together by their index in their respective buffers (as shown in the figure above).

In turn, the index buffer refers to these individual vertex attribute groups, and defines which of them are combined into primitives. Each vertex attribute is used as input to

any active shaders, where the location of each input is matched against the ID (or index) of the vertex attribute.

We will now see how to initiate this rendering process. Don't worry, it's a lot simpler than setting up the VAO!

2.2.2 Drawing the Vertex Array Object

As the frame buffer has to be cleared every frame (the handout code already contains a call to `glClear()`), you have to redraw the scene every frame as well. Drawing a scene usually happens in a loop known as the "main loop". The handout code already has one set up for you. You can insert your drawing calls in there.

The first step of drawing a VAO is to bind it. This works exactly the same way as when we set it up; a single call to `glBindVertexArray()`. After this, any drawing command will use this VAO as input. You can only draw from a single VAO at a time.

Issuing the draw command

As we have already set up the VAO in its entirety previously, OpenGL already knows how to interpret the various Vertex Buffer Objects. As such we just have to call a single function to draw the VAO:

```
void glDrawElements(enum mode, int count, enum type, void* indices);
```

`glDrawElements` will use the data specified in the VAO as input to the rendering pipeline.

The `mode` parameter specifies the type of primitive you'd like to draw. The basic primitive modes are `GL_POINTS`, `GL_LINES` and `GL_TRIANGLES`.

Here are some other handy drawing modes that combine the basic primitives:

`GL_LINE_STRIP`
: Start with one vertex. For every vertex that follows, draw a line from the previous vertex to the current one.

`GL_LINE_LOOP`
: Same as `GL_LINE_STRIP`, but adds an additional line from the last to the first vertex.

`GL_TRIANGLE_STRIP`
: Works similar to a line strip, but uses the previous 2 vertices and the current vertex as coordinates of the triangle.

`GL_TRIANGLE_FAN`
: Handy for drawing circles. Start with a center vertex and a vertex on the edge of the circle. Every vertex you add draws a triangle through the center vertex, the previous and the current one.

The `count` paramter specifies how many elements from the index buffer should be drawn. Note that this is the *number of elements*, not the *number of triangles, lines or points*. For example, in the case of `GL_TRIANGLES`, `count` should always be a multiple of 3.

`Type` specifies the data type of the values in your index buffer. This is usually `GL_UNSIGNED_INT`, although `GL_UNSIGNED_SHORT` and `GL_UNSIGNED_BYTE` are also accepted if you happened to specify your indices with those data types.

Finally, `indices` specifies the start index in your index buffer to start drawing from. Just pass in 0 here.

Try running the program now and see if you can see the fruits of your labour :)

Chapter 3

Introduction to GLSL

3.1 Shaders in OpenGL

We've seen in the previous section how you can draw some basic geometry in OpenGL. Now we address two other important issues in rendering images: where does the object appear in the scene, and what colour(s) does it have?

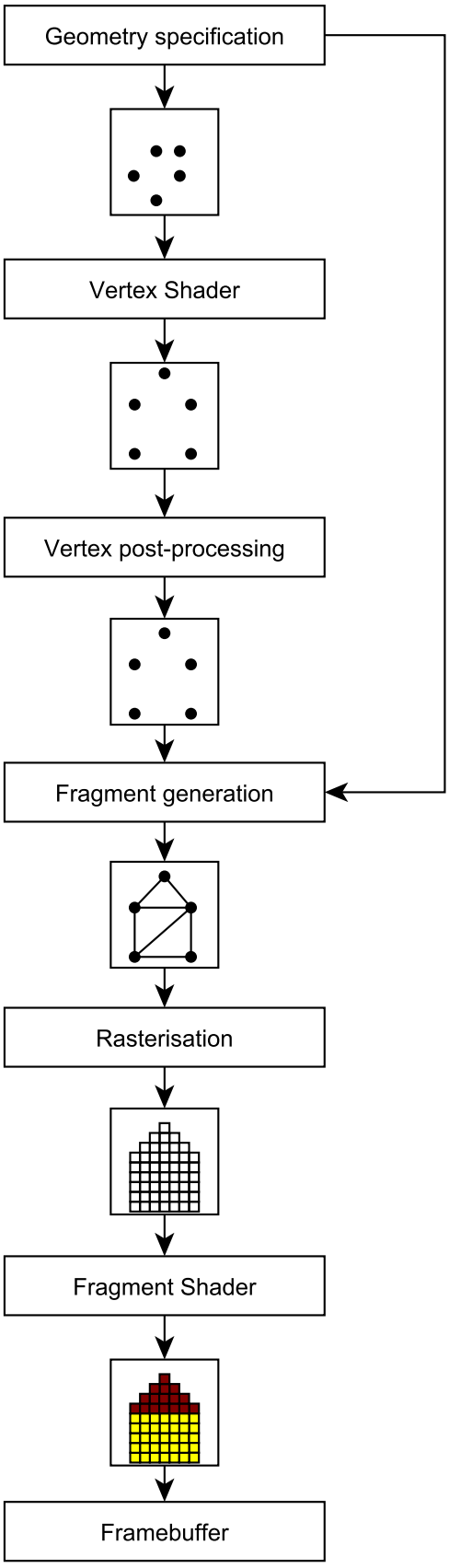
Both of these are accomplished through something known as a *Shader*. In OpenGL, a Shader is a small program that runs on the GPU. There exist a number of types of Shaders for different purposes.

Shaders are commonly invoked many times (in the order of millions) to render even a single frame. There's a range of different kinds of Shaders, but most of them are irrelevant unless you're trying to do something very advanced.

In practice, you'll commonly only be using two particular types of Shaders:

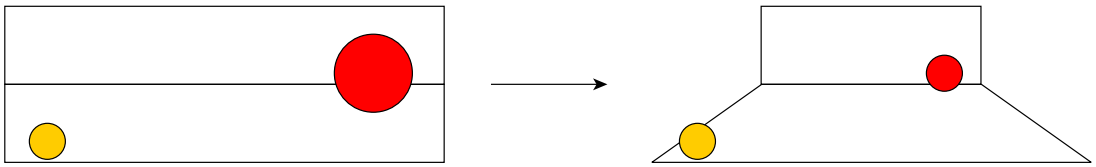
- The Vertex Shader
- The Fragment Shader

These two particular types of Shaders are an integral part of the OpenGL rendering pipeline. For reference, here's the diagram showing the pipeline we've seen in chapter 1. Note where the Vertex and Fragment shaders are located in the pipeline.


Geometry specification
Vertex Shader
Vertex post-processing
Fragment generation
Rasterisation
Fragment Shader
Framebuffer

The Vertex Shader runs once for every single vertex that is drawn. It is responsible for transforming (translating, scaling, rotating, etc.) individual vertices around the scene. This is for instance used to move objects around the scene, or to place them in desired locations.

Additionally, the Vertex Shader is responsible for projecting the scene on to the camera. Projections are used to define how the world looks through the *lens* of the camera. For example, a "perspective projection" causes objects that are further away from the camera to shrink in size, mimicking the way the human eye perceives a scene. Here is the effect of a perspective projection on a scene with two spheres:



Original Geometry

Perspective Projection

After the Vertex Shader has finished processing all vertices, OpenGL uses the geometry specification to figure out which geometric primitives it should draw. These can either be points, lines or triangles. It then rasterises these primitives into pixels.

This is where the Fragment Shader comes in, which is responsible for determining the colour of each fragment. Fragments are pixels OpenGL attempts to draw. These may end up behind other geometry, thus not being visible in the final frame! They are thus not exactly the same thing as a pixel on screen.

The Fragment Shader is executed once for **every single** fragment. For complicated scenes this may imply rendering significantly more fragments than there are pixels on the screen.

3.2 The GLSL Language

So how exactly does one write a Shader? The OpenGL standard describes a language, imaginatively named the "OpenGL Shading Language", or GLSL for short.

GLSL is designed as a dialect of C, with a number of additions and limitations. We will therefore mainly look at the differences between these two languages. However, due to the extensive number of features in GLSL not all language components will be mentioned here for the sake of simplicity. As such you should not consider this overview complete or exhaustive, but it should include everything you need to complete the labs.

3.2.1 General Layout of a Shader

A minimal Shader source file commonly follows a specific layout. This layout has been outlined below:

```
#version xxx
// Line 1: GLSL version declaration (this *HAS* to be line 1!)

// Input/output variable declaration
// For example:
in layout(location=2) vec3 vertex;

// A main function
void main() {
        // Do things
}
```

As you can see, there's a version declaration of which version of GLSL you are using, as well as a listing of what values are required and returned by the Shader. Finally, the main function represents the entry point of the Shader. Note that unlike some versions of C, the main function in GLSL always returns `void`.

We'll take a look at each of these parts in the sections below.

3.2.2 The #version Statement

The first line in every Shader **must** be a declaration of which version of GLSL the Shader is written in. There have been a number of iterations of GLSL over the years, some of which have significantly changed the language syntax.

We are using OpenGL 4.0 or higher in these labs, so the version statement should reflect that. For example, putting `#version 400` at the top of a shader file will cause OpenGL to use the revision of GLSL that accompanies OpenGL 4.0.

3.2.3 The Input/Output Declaration

Because the Vertex and Fragment Shaders are programmable stages of the OpenGL pipeline, they require input for their operation and are expected to produce a certain kinds of output depending on their place in the pipeline.

OpenGL requires these inputs and outputs to be defined as global variables in the Shader. To mark a variable as an input or output, you have to use the `in` and `out` keywords respectively, like this:

```
in vec4 vertex;
out vec4 transformedVertex;
```

Because OpenGL tries to be both helpful and versatile, the Vertex and Fragment Shader both require you to define different inputs and outputs. Here's an overview of what you have to define in each case:

Vertex Shader input:
You need to explicitly define the vertex the Vertex Shader should process as an input. This is usually: `in vec4 vertex`. Note that if you want to pass any additional attributes to the Fragment Shader, you also have to define those as inputs here, and assign them to the output variables in the Shader code. This will cause them to be passed on to the Fragment Shader.

Vertex Shader output:
You are required to somewhere in your Shader code set the predefined output variable `out vec4 gl_Position` containing the transformed location of the vertex being processed by the Shader. In addition, you can define additional outputs that will be passed on to the Fragment Shader in case additional values are required for calculating the colours of pixels.

Fragment Shader input:
Any values that have been explicitly passed on from the Vertex Shader. Note that because the Vertex Shader is only executed for every vertex instead of every pixel, values from different vertices are interpolated automatically by OpenGL before they are passed into the Fragment Shader.

This can be turned off if desired (see the OpenGL documentation for details), but is in the vast majority of cases a useful feature.

Fragment Shader output:
Usually only a single `vec4` with RGBA format containing the colour of the fragment (pixel). Note that in GLSL all colour channel values range between 0 and 1.

There is one additional note to make on the Vertex Shader; if you require more than one input value (for instance normals or texture coordinate), OpenGL has to know where these values are originating from.

If you recall, when creating the Vertex Array Object (VAO), you explained to OpenGL how your buffer(s) were formatted. You used the `glVertexAttribPointer()` function for this. Unfortunately, OpenGL still has no idea what buffer contains what data, so it has no idea what Shader input each buffer corresponds to.

OpenGL therefore contains a "layout" mechanism to identify these correspondences. If you look at the `glVertexAttribPointer()` function specification, the first parameter requires you to specify an index. As mentioned in the explanation for setting up the VAO, this number can mostly be anything you specify.

To ensure that the correct Vertex Attribute (roughly a VBO) is connected to the correct shader input, you just have to ensure that the index specified in `glVertexAttribPointer()` matches with the index of the shader input variable.

The easiest way for accomplishing this is to use the `layout(location=[index])` construct before your input declaration. For example, if we would like the input variable to correspond to a buffer with index 6, we can define the input variable as follows:

```
layout(location=6) in vec4 vertex;
```

If you want to pass values on to the next Shader, such as the Fragment Shader, you will also need to specify layout indices, in the same way as you did for inputs. You then have to make sure that the output index of the Vertex Shader matches the input index of the Fragment Shader to ensure OpenGL understands this output and input correspond to each other.

For illustration, the following output of the Vertex Shader:

```
layout(location=2) out vec4 colour;
```

Would correspond to this input of the Fragment Shader:

```
layout(location=2) in vec4 thisNameCanBeDifferent;
```

Note, however, that this feature is only available from OpenGL 4.3 or higher. If your graphics card and driver does not support this version, or you have configured Gloom to use an earlier version (it currently uses version 4.0 by default), you need to use a different method.

In this case there are two options.

First, it is possible to let OpenGL automatically determine the indices at which Vertex Attributes should be assigned. You can accomplish this by leaving out the location specification in the Shader source.

Next, ensure that your Shader is loaded before you construct your VAO(s). You can now use the `glGetAttribLocation()` function to obtain the indexes which OpenGL has assigned to your inputs. You should set the indexes passed in to `glVertexAttribPointer()` and `glEnableVertexAttribArray()` while setting up the VAO.

Alternatively, after compiling the shaders, but before linking them, it is possible to specify the input locations by calling the function `glBindAttribLocation()`. This function replaces the effect of layout specification syntax.

Of the two alternatives, this method is superior. However, it requires modification to the shader loading function supplied in Gloom. You can decide yourself which solution works best for you. All three methods are considered correct in the context of grading of these labs.

Uniforms

There is another "special" kind of input to a Shader; the uniform variable. You can consider uniforms to be *parameters* that are the same for every instance of your Shader that's being executed. Hence the name "uniform".

You can use them to pass in a value of any type supported by GLSL into a Shader. They act as constants while the Shader is executing. You can thus only read from them from within the Shader itself. One application of uniforms is to define values that are the same throughout a single draw call, but still change from time to time.

The value of a Uniform is kept between executions of the shader. It retains its value until it is changed. Also note that GLSL considers uniforms inputs to your Shader. You may therefore want to specify positions for them explicitly, like you did with the inputs and outputs before. Here's an example:

```
uniform layout(location = 3) vec4 lightPosition;
```

You can then use the glUniform[number of elements][parameter datatype]() function to set the value(s) of the uniform. For the uniform defined above, this is how you can set its value in OpenGL:

```
// Function used:
// glUniform4f(int index,
//             float value1, float value2, float value3, float value4);
// Note that the position of the uniform was defined to be 3
glUniform4f(3, 1f, 2f, 3f, 4f);
```

If the layout specification syntax is not supported on your machine, you can use the `glGetUniformLocation()` function to obtain the index assigned to the input by OpenGL.

3.2.4 Data Types

There exist five basic datatypes in GLSL: `bool`, `int`, `uint`, `float` and `double`. Note that unlike C, integers are specified to be 32-bit values. Also, most modern graphics cards do not natively support double precision values, or are extremely slow at processing them. You'll therefore want to default to single precision whenever you need to work with floating point values.

GLSL also includes vector and matrix types for conveniently grouping data together.

The most commonly used vector types are `vec2`, `vec3` and `vec4`, containing 2, 3 and 4 single precision floats respectively. You can create a new vector like this:

```
vec4 vector = vec4(1.0, 0.0, 0.0, 1.0);
```

And access individual elements like this:

```
// method 1: use the built-in rgba components
float element0 = vector.r;
float element1 = vector.g;
float element2 = vector.b;
float element3 = vector.a;

// method 2: use the built-in xyzw components:
float element0 = vector.x;
float element1 = vector.y;
float element2 = vector.z;
float element3 = vector.w;

// method 3: use array indices:
float element0 = vector[0];
float element1 = vector[1];
float element2 = vector[2];
float element3 = vector[3];
```

If you only need certain elements of the vector, you can also combine different properties like this:

```
vec2 location2D = vector.xy;
vec3 colour = vector.rgb;
```

Besides vectors, GLSL also contains built-in matrix types. These types are formatted as "mat[number of columns]x[number of rows]". For instance, the type of a matrix with 3 rows and 2 columns would be `mat2x3`. Unlike vector types, there is only one method for accessing matrices.

Also note that matrices in OpenGL are column major. If you're not familiar with the term, column major means that you first address the column, then the row when dealing with multidimensional arrays, such as matrices.

Here's a snippet of code showing how to use them:

```
// defines the matrix: [1, 2]
//                     [3, 4]
mat2x2 matrix = {{1, 3}, {2, 4}};
// changes it to: [1, 2]
//                [5, 4]
matrix[0][1] = 5.0;
```

3.2.5 Operators

Even though most operators such as addition, subtraction or multiplication work exactly the same as in C, there are some notable differences with GLSL.

First of all, you can't typecast in GLSL in the way you can in C. Instead, you can in most cases call the type you want to convert to as a "function". Here's an example showing how to convert from a float to an int:

```
int intValue = int(floatValue);
```

Secondly, you can use the basic arithmetic operators (+, -, *, /) on matrices and vectors, or a combination of the two. Do note that this follows the rules of matrix multiplication. So addition performs element wise addition, given that the matrices or vectors are of equal dimensions.

Here's some examples:

```
// Creates a 4x4 matrix with 1's in the leading diagonal;
// the identity matrix.
mat4x4 matrix = mat4(1);

vec3 positionXYZ = vec3(1, 2, 3);
vec4 positionXYZW = vec4(1, 2, 3, 1);

// Compilation error: a 4x4 matrix can't be multiplied with a 3x1 matrix
vec4 error = matrix * positionXYZ;

// returns vec4(1, 2, 3, 1) as anything multiplied with the
// identity matrix results in the same matrix.
vec4 newPosition = matrix * positionXYZW;

// returns vec3(2, 4, 6)
vec3 doublePosition = 2 * positionXYZ;
```

3.2.6 Language Constructs

The remainder of GLSL should pretty much be downhill from here on out, because it's practically the same as C :)

But let's go over them for the sake of reference.

First of all, the if statement. You really want to avoid these as much as possible, as the branch instructions generated by if statements are incredibly expensive on GPUs. Especially considering that the Vertex and Fragment Shaders are executed many times per frame.

Moreover, modern GPUs tend to cluster cores together in order to allow them to share part of the processor logic. This for instance includes instruction decoding. It allows the size of each individual core to be shrunk, and as a result more cores can fit on the die.

This also affects the way shaders are executed. Specifically, if a branch instruction causes some cores to choose the *if* clause and some the *else* clause, *both* clauses are executed. Cores which chose the *else* clause will need to wait until the *if* clause is finished, and vice versa. As such you don't get a speedup in the execution of your shader the way you would if it were executed on a CPU. It may even cause a slowdown.

```
if(someVariable > 3) {
        // Do something
} else {
        // Do something else.
}
```

The same is true for for loops. If you need the performance, it's usually better to look for alternate strategies or redundancies to avoid having to use branching.

```
for(int i = 0; i < 10; i++) {
        // Do something
}
```

That also counts for the while loop:

```
while(someCondition != false) {
        // Do something
}
```

And finally, you can define additional functions if you need them:

```
bool isGreaterThan(float a, float b) {
        return a > b;
}
```

3.3 Loading and using Shaders

You've hopefully gotten a sense at this point on how to write a Shader. Now we should take a look at how to set them up and use them.

In terms of the lab work, doing this yourself is not required, though if you want to implement the procedure yourself you're welcome to do so. An implementation of the shader loading procedure has been implemented for you in the file Shader.hpp.

As opposed to Direct3D's shader model, OpenGL has opted for a local compilation approach. This means that whenever you want to use a shader on the user's machine, you have to compile it locally. Major graphics card vendors therefore include GLSL compilers in their graphics drivers.

This may sound like an odd decision at first. After all, why do you not compile it once on your development machine, and save everyone else the effort?

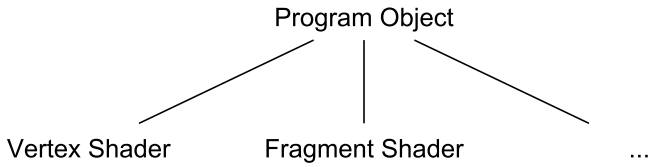
The problem is that modern graphics hardware differs from vendor to vendor. The binary code generated by each vendor's compiler may differ significantly. Local compilation avoids tricky situations where you have to compile explicitly for each vendor's card, possibly even different models. Local compilation is therefore a more fitting approach.

3.3.1 The Shader Program Object

In OpenGL shaders are grouped together into so-called *Program Objects*. The idea is that each shader is compiled individually, attached to a Program Object and finally linked together. Once set up, Program Objects can be activated at will.

The linking stage is responsible for checking that shader combinations are compatible. That is, that one shader's output specifications match with another shader's input specification.

The resulting structure looks something like this:



First off, we create a new Program Object using:

```
unsigned int glCreateProgram();
```

Once again, the function does not return a structure or object, but an ID that can be used to refer to the object in OpenGL calls.

3.3.2 Loading and Compiling a Shader

Next, we need to load and compile each of the shaders we would like to use. OpenGL does not provide any means for loading shader source code from a file, so loading a text file into memory is your responsibility.

Hint: it may be useful to drop the loading of a single shader in its own function as the process is nearly identical for each shader type.

Compiling a shader starts with the creation of a Shader object:

```
unsigned int glCreateShader(enum shaderType);
```

The shaderType parameter specifies the type of shader you are creating. Some relevant options are:

- `GL_VERTEX_SHADER`
- `GL_FRAGMENT_SHADER`

Next we pass in the source code of the shader Program using:

```
void glShaderSource(
        unsigned int shaderID,
        int count,
        char** strings,
        int** lengths
);
```

The `shaderID` parameter is the ID returned by `glCreateShader()`.

The source string itself can either be one large string, or chopped up into smaller bits. Appending them all together should give the complete source code of the Shader.

`Count` represents the number of parts you have divided the Shader source code into. `Strings` and `lengths` are arrays of strings and integers, respectively. `Strings` contains the actual source code, and `lengths` the length of each string in `strings`.

Now that the source code is in place, we can compile it using:

```
void glCompileShader(unsigned int shaderID);
```

Although you're strictly not required to, it's very good practice to check for compilation errors.

You can check whether compiler errors occurred using:

```
int shaderCompilationStatus = 0;

// glGetShaderiv() allows requesting information about a shader.
// In this case the compilation status (GL_TRUE if success, GL_FALSE if not)
glGetShaderiv(shaderID, GL_COMPILE_STATUS, &shaderCompilationStatus);

bool compilationErrorsOccurred = shaderCompilationStatus == GL_FALSE;
```

If an error occurred, you can obtain an information log using:

```
int logLength = 0;

// Again using glGetShaderiv() for obtaining the log length
glGetShaderiv(shaderID, GL_INFO_LOG_LENGTH, &logLength);

// Allocating enough memory to store the info log
char* infoLog = (char*) malloc(logLength);

glGetShaderInfoLog(shaderID, logLength, NULL, infoLog);

// Print the info log:
printf("A Shader compilation error occurred. Info log:\n");
printf("%s\n", infoLog);

free(infoLog);
```

You can delete a shader to free up memory if the shader is not going to be used for a significant stretch of time. You can do so using:

```
void glDeleteShader(unsigned int shaderID);
```

And we're done! Run this process for both the Vertex and Fragment shaders, and we can move on to the linking stage of compilation.

3.3.3 Attaching and Linking Shaders

After compiling both shaders, you have to attach each shader to the Program object you created. The `glAttachShader()` function does just that:

```
void glAttachShader(unsigned int programID, unsigned int shaderID);
```

This will make Shaders a part of the Program Object.

Now that all shaders have been attached to the Program Object, we can perform the linking stage of compilation:

```
void glLinkShader(unsigned int programID);
```

Again, it is strictly not necessary to check for linking errors, but doing so is considered good practice. The process is almost identical compared to Shader compilation error checking. You can obtain the linking status of your Program by calling:

```
int programLinkStatus = 0;

glGetProgramiv(programID, GL_LINK_STATUS, &programLinkStatus);
bool linkingFailed = programLinkStatus == GL_FALSE;
```

Obtaining the error log is also the same compared to Shaders, apart from the OpenGL function calls:

```
int logLength = 0;

// Using glGetProgramiv() instead of glGetSDhaderiv().
// The functions have the same intent; obtaining information
// about the Program and Shader, respectively.
glGetProgramiv(programID, GL_INFO_LOG_LENGTH, &logLength);

// Allocating enough memory to store the info log
char* infoLog = (char*) malloc(logLength);

glGetProgramInfoLog(shaderID, logLength, NULL, infoLog);

// Print the info log:
printf("Program linking failed. Info log:\n");
printf("%s\n", infoLog);

free(infoLog);
```

And that's it. We can now use the Program object at will. When enabled, the Shaders you defined will take their respective places in the OpenGL pipeline, and do whatever you have instructed them to do.

3.3.4 Enabling and Disabling the Program Object

The only thing that now remains is the question of how to activate your Program Object. This can be done by calling:

```
void glUseProgram(unsigned int programID);
```

Passing in your programID will activate it. Passing in 0 will restore OpenGL's default behaviour.

3.3.5 Debugging Shaders

Because shaders are executed in such large quantities and communication between the CPU and GPU is complicated, it is very difficult to debug one. The Fragment Shader is

in my own experience the most complicated one to get right, so I tend to use the fragment colour as a debug value.

For instance, if a value becomes inexplicably large, you can insert an if statement that checks for such large values. It can then set the pixel colour to red, which makes it visible to you.

Another tip is to read your shader's source code. Shader code tends to (and should!) be very short. The problem space is therefore usually much smaller than typical CPU code. Careful reading can therefore get you quite far.

Don't Panic