The GPU

The acronym *GPU* stands for *graphics processing unit.* Similar to how the CPU is in charge of all computations required by the computer, the GPU is in charge of all graphics computations. We will not dive too much into the architecture of the GPU itself, but we will be discussing certain aspects of it and how it relates to the graphics pipeline.

A Very Brief History of the GPU

Graphics acceleration began in software by implementing simple things such as blending neighbouring pixels and then moving on to more complex things. Eventually, it was decided that since these operations need to be performed *very* frequently (as fast as the screen can refresh itself in most cases) it would be better if we had specific hardware to perform these computations. This is the dawn of the graphics specific hardware. The term GPU was coined by Sony in 1994 referring to the original Playstation, whereas NVidia popularized it in 1999 with the GeForce 256. Since then, GPUs have become faster and massively more powerful, capable of performing a large number of computations per second, and able of handling far more advanced algorithms.

The Graphics Pipeline and the GPU

The question now is: if we have a card that is built *specifically* for doing graphics computations, then which parts of the graphics pipeline are performed on it? The answer is simple: *everything* with the exception of the application stage is performed on the GPU. In fact, several stages of the pipeline are completely optimized at the hardware level, while others can be configured (that is to say the programmer can change the behaviour within a certain set, but it cannot be programmed) and finally others are fully programmable. Before we dive into this however, we need to understand the following: if the majority of the pipeline resides in the GPU, how does the application stage, which resided in the CPU, have control over it?

Graphics APIs and OpenGL

In order to understand how we interact with the GPU, we first need to understand how our code interacts with the computer. Note that this will be a very superficial and over-simplified explanation of what happens when we execute our code.

Suppose that I take a C++ program. In order to execute it, I must first ask the operating system to run it, which will cause the OS to invoke the `main` function. From here, any operation such as dynamic memory allocations, printing to the console, saving (or opening) a file, etc will require a request to the OS to perform a certain task. In other words, the OS is a layer of software that sits between us and the hardware. This is mostly transparent to us as both the OS and our code are executed by the same processing unit: the CPU.

The situation becomes more complex when we add the GPU. The GPU is a processing unit that is outside the CPU. Since the hardware of the GPU cannot be guaranteed (in other words it could have anything), we need a layer that abstracts that and allows us to interact with it. This layer is called the *driver*. But the driver merely abstracts the hardware of the GPU, but it doesn't give us a direct way of interacting with it. What we ultimately require is an API such as the one that C++, C, etc give us. This is where OpenGL comes in. OpenGL is a *graphics API* that stands for Open Graphics Library and is controlled by a group called Khronos. Khronos itself is comprised of several companies such as Microsoft, NVidia, AMD, amongst others.

So, whenever we want to perform graphics operations, we use the API that OpenGL provides us. From here, the command is forwarded by the OS to the driver, which then decides what needs to happen from here. It is important to keep in mind that we are always dealing *through* the driver. In fact, the layer that the driver controls for an API like OpenGL is so thick that it can do certain operations (such as cleaning up resources) on its own! This is not a problem for us, but it is a problem for applications such as game engines that require a higher level of performance.

Now that we have a graphics API, we can turn our attention to trying to render something. In our case, we will be rendering a triangle. In order to do this, we will walk through the rendering pipeline setting things up as we go so we can finally have a triangle on the screen.

Rendering a Triangle

The OpenGL Context and Window

The first step in the pipeline is the application stage. In here, we will need to create the data for the triangle as well as set up the required OpenGL state in order to drive the pipeline. Our first task is to get a window and a *context*. The window will be used to display the triangle that we render. The context on the other hand is our way of talking with the driver. Due to the way that the functions in the OpenGL API are implemented within the driver, we can't simply include a header and start using them. What we need is to load the addresses of the functions themselves along with all the necessary state related to OpenGL. All of this is encapsulated (at least for our purposes) in the *OpenGL context*. For the sake of simplicity we will simply assume that we can only have one context at any one time, along with only one window. This isn't necessarily true, but it does satisfy our needs for now.

> **Note:** The code that will be shown here is for demonstration purposes only and may not always follow best design practices. Once a concept has been explained it will be encapsulated in an object for ease of use as we move forward.

To start, we first need a window. Since the context is ultimately tied to the window we are drawing to, it makes sense to make that first. Atlas provides functions to simplify this process. We will begin our `main` as follows:

```
int main()
{
    glx::WindowSettings settings;
```

`WindowSettings` is a struct that contains all the data necessary to create our window, such as size, title, etc. For the time being we will only change the title of the window as follows:

```
settings.title = "Hello Triangle";
```

We will accept the defaults that Atlas has and use this struct to create the window. Atlas utilizes a library called GLFW to handle window creation, so we first must initialize GLFW. In order to capture errors, GLFW takes in a function that it can invoke to log them, so we will implement the function first.

```
void errorCallback(int code, char const* message)
{
    fmt::print("error: ({}): {}\n", code, message);
}
```

And now we can initialize GLFW:

```
if (!glx::initializeGLFW(errorCallback))
{
    return 1;
}
```

The check to ensure that GLFW initializes correctly isn't strictly speaking necessary since (for the most part) GLFW never fails, but it is good practice to have it. For the sake of brevity we will be omitting such checks in the future. Now that GLFW is initialized, we can create our window:

```
GLFWWindow* window = glx::createGLFWWindow(settings);
if (window == nullptr)
{
    return 1;
}
```

The `GLFWWindow` pointer will hold our window, and we will be using it for any functions that require it. Next, we need to tell GLFW that the window that we just created should be the window that the OpenGL context that we will create later on is bound to. We do this as follows:

```
glfwMakeContextCurret(window);
```

Now we are ready to create the OpenGL context. The `settings` struct that we created earlier also contains the OpenGL version that we require for the context, so we will pass it along:

```
if (!glx::createGLContext(window, settings.version))
{
    return 1;
}
```

And now we have the window and the context created. Next stop: the main loop.

The Main Loop

In order to have a window remain open until we decide to close it, we need to write an infinite loop that will run until a condition is met (or until we break out of it). Every application has a loop like this somewhere in its code, and therefore it is called the *main loop*. In our case, all rendering code is going to be handled in the main loop. The general structure of the loop is the following:

```
while (!glfwWindowShouldClose(window))
{
    // Do something.
}
```

Let's go back to ray tracing for a second. At the beginning of our code we defined the size of the image and created the array of pixels that we would later fill in. With OpenGL we have to do something similar. The only difference is that instead of calling it the viewing plane, we will call it the *viewport*. We will assume that our view port is the same size as the window itself. To do this, we can ask GLFW to give us back the size of the window. We will query this through a function called `glfwGetFramebufferSize`. The reason why we don't use a function that returns the size of the window itself is to avoid having problems where the pixel density of the monitor can change the size. So, at the top of our main loop we do:

```
while (!glfwWindowShouldClose(window))
{
    int width, height;
    glfwGetFramebufferSize(window, &width, &height);
    glViewport(0, 0, width, height);
```

`glViewport` is the first OpenGL function that we will see. All OpenGL functions are prefaced with `gl` so they are easy to identify. So, for our case the viewport spans the whole screen. What OpenGL expects us to provide is the *rectangle* that will hold our viewport. We can specify this rectangle by giving the coordinates to the **lower-left** corner and the dimensions. It is important to keep in mind that for OpenGL, the viewport coordinates go from $(0, 0)$ on the lower-left and (w, h) on the upper-right. Hence, we pass the values `0, 0, width, height` to specify the dimensions of the viewport.

The next thing we are going to do is to clear out the screen in order to ensure that we are always starting with a blank space to work with. This is similar to how we needed to initialize memory in C, since we could not guarantee that the memory we just received didn't have garbage in it. To do that, we do the following:

```
glClearColor(0, 0, 0, 1);
```

This will clear the pixels in the screen with the provided RGBA value. Note that the values need to be in the range $[0, 1]$. In essence, what this function is doing is specifying the *value* that the buffers will be cleared to when they are cleared. Now recall that the final output of the pipeline is the image. The way this is internally represented is through an array of colours, which we will call the *colour buffer*. In addition to the colour buffer, we also produce a depth buffer. What we will do is clear both of them to ensure that our next rendering commands are executed on a clean slate.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

The last two things our main loop will handle are: telling GLFW to swap the buffers (more on this later) and to check if any inputs have come in. We do this as follows:

```
glfwSwapBuffers(window);
glfwPollEvents();
```

If the window closes, we just need to clean up the window and terminate GLFW. We do that as follows:

```
glx::destroyGLFWWindow(window);
glx::terminateGLFW();

return 0;
```

Once we run our code, we will see a black window and we are ready to start setting up the triangle data.