Computer Graphics
Theory and Applications

Konstantinos Zouridakis

Contents

1 OpenGL

1.1 Basics

An **object** in OpenGL is a collection of options that represents a subset of OpenGL's state. For example, we could have an object that represents the settings of the drawing window; we could then set its size, how many colors it supports and so on. One could visualize an object as a C-like struct:

```
struct object_name {
float option1;
int option2;
char[] name;
};
```

Whenever we want to use objects it generally looks something like this (with OpenGL's context visualized as a large struct):

```
// The State of OpenGL
struct OpenGL_Context {
    ...
    object_name* object_Window_Target;
    ...
};
```

```
// create object
unsigned int objectId = 0;
glGenObject(1, &objectId);
// bind/assign object to context
```

```
glBindObject(GL_WINDOW_TARGET, objectId);
// set options of object currently bound to GL_WINDOW_TARGET
glSetObjectOption(GL_WINDOW_TARGET, GL_OPTION_WINDOW_WIDTH, 800);
glSetObjectOption(GL_WINDOW_TARGET, GL_OPTION_WINDOW_HEIGHT, 600);
// set context target back to default
glBindObject(GL_WINDOW_TARGET, 0);
```

This little piece of code is a workflow you'll frequently see when working with OpenGL. We first create an object and store a reference to it as an id (the real object's data is stored behind the scenes). Then we bind the object (using its id) to the target location of the context (the location of the example window object target is defined as `GL_WINDOW_TARGET`). Next we set the window options and finally we un-bind the object by setting the current object id of the window target to 0. The options we set are stored in the object referenced by objectId and restored as soon as we bind the object back to `GL_WINDOW_TARGET`.

The great thing about using these objects is that we can define more than one object in our application, set their options and whenever we start an operation that uses OpenGL's state, we bind the object with our preferred settings. There are objects for example that act as container objects for 3D model data (a house or a character) and whenever we want to draw one of them, we bind the object containing the model data that we want to draw (we first created and set options for these objects). Having several objects allows us to specify many models and whenever we want to draw a specific model, we simply bind the corresponding object before drawing without setting all their options again.

1.2 Hello Window

Let's see if we can get GLFW up and running. First, create a `.cpp` file and add the following includes to the top of your newly created file.

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
```

Make sure to include `glad.h` before `glfw3.h`. Instead of `glad.h` we can include `GL/glew.h`. While for beginners `glew` is sufficient, in more advanced use cases `glad.h` might be more suitable.

Next, we create the `main` function where we will instantiate the GLFW window:

```
int main()
{
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    //glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);

    return 0;
}
```

`glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);`
and `glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);` translates to setting the OpenGL version to 3.3. The next line sets the OpenGL profile to the core profile. The core profile means that

deprecated functionality will be removed. Note that on Mac OS X you need to add `glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);` to your initialization code for it to work.

Next we're required to create a window object. This window object holds all the windowing data and is required by most of GLFW's other functions.

```
GLFWwindow* window = glfwCreateWindow(800, 600, "LearnOpenGL", NULL, NULL);
if (window == NULL)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window);
```

The `glfwCreateWindow` function requires the window width and height as its first two arguments respectively. The third argument allows us to create a name for the window; for now we call it "LearnOpenGL" but you're allowed to name it however you like. We can ignore the last 2 parameters. The function returns a `GLFWwindow` object that we'll later need for other GLFW operations. After that we tell GLFW to make the context of our window the main context on the current thread.

We will also need to initialize GLAD as seen below:

```
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
}
```

We pass GLAD the function to load the address of the OpenGL function pointers which is OS-specific. GLFW gives us `glfwGetProcAddress` that defines the correct function based on which OS we're compiling for.

Before we can start rendering we have to do one last thing. We have to tell OpenGL the size of the rendering window so OpenGL knows how we want to display the data and coordinates with respect to the window. We can set those dimensions via the glViewport function:

```
glViewport(0, 0, 800, 600);
```

The first two parameters of `glViewport` set the location of the lower left corner of the window. The third and fourth parameter set the width and height of the rendering window in pixels, which we set equal to GLFW's window size.

We could actually set the viewport dimensions at values smaller than GLFW's dimensions; then all the OpenGL rendering would be displayed in a smaller window and we could for example display other elements outside the OpenGL viewport.

Behind the scenes OpenGL uses the data specified via `glViewport` to transform the 2D coordinates it processed to coordinates on your screen. For example, a processed point of location (-0.5,0.5) would (as its final transformation) be mapped to (200,450) in screen coordinates. Note that processed coordinates in OpenGL are between -1 and 1 so we effectively map from the range (-1 to 1) to (0, 800) and (0, 600).

However, the moment a user resizes the window the viewport should be adjusted as well. We can register a callback function on the window that gets called each time the window is resized. This resize callback function has the following prototype:

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height);
```

The framebuffer size function takes a `GLFWwindow` as its first argument and two integers indicating the new window dimensions. Whenever the window changes in size, GLFW calls this function and fills in the proper arguments for you to process.

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}
```

We do have to tell GLFW we want to call this function on every window resize by registering it:

```
glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
```

When the window is first displayed `framebuffer_size_callback` gets called as well with the resulting window dimensions. For retina displays width and height will end up significantly higher than the original input values.

There are many callbacks functions we can set to register our own functions. For example, we can make a callback function to process joystick input changes, process error messages etc. We register the callback functions after we've created the window and before the render loop is initiated.

We don't want the application to draw a single image and then immediately quit and close the window. We want the application to keep drawing images and handling user input until the program has been explicitly told to stop. For this reason we have to create a while loop, that we now call the render loop, that keeps on running until we tell GLFW to stop. The following code shows a very simple render loop:

```
while(!glfwWindowShouldClose(window))
{
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

The `glfwWindowShouldClose` function checks at the start of each loop iteration if GLFW has been instructed to close. If so, the function returns true and the render loop stops running, after which we can close the application. The `glfwPollEvents` function checks if any events are triggered (like keyboard input or mouse movement events), updates the window state, and calls the corresponding functions (which we can register via callback methods). The `glfwSwapBuffers` will swap the color buffer (a large 2D buffer that contains color values for each pixel in GLFW's window) that is used to render to during this render iteration and show it as output to the screen.

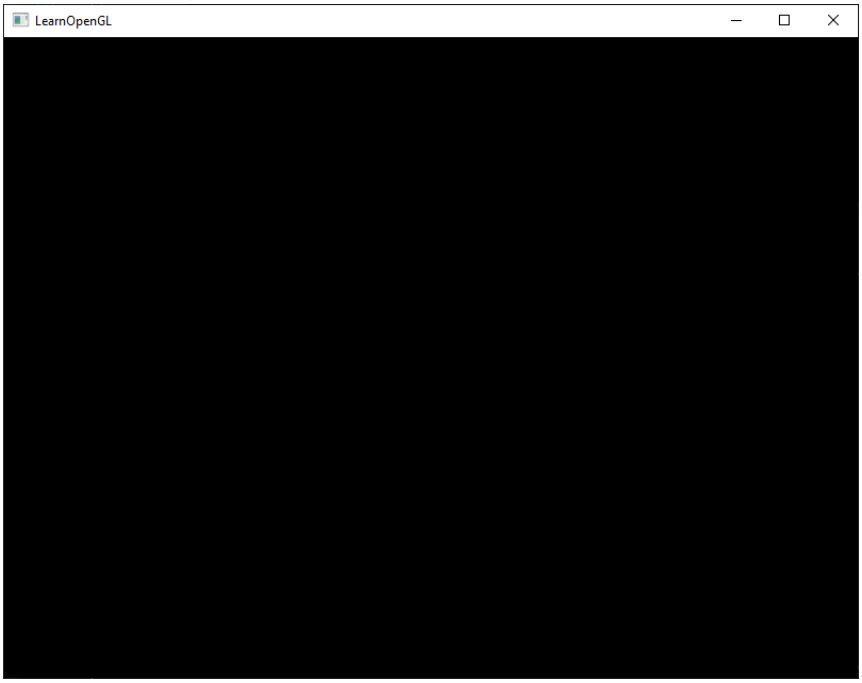
Double Buffer

When an application draws in a single buffer the resulting image may display flickering issues. This is because the resulting output image is not drawn in an instant, but drawn pixel by pixel and usually from left to right and top to bottom. Because this image is not displayed at an instant to the user while still being rendered to, the result may contain artifacts. To circumvent these issues, windowing applications apply a double buffer for rendering. The **front** buffer contains the final output image that is shown at the screen, while all the rendering commands draw to the **back** buffer. As soon as all the rendering commands are finished we swap the back buffer to the front buffer so the image can be displayed without still being rendered to, removing all the aforementioned artifacts.

As soon as we exit the render loop we would like to properly clean/delete all of GLFW's resources that were allocated. We can do this via the `glfwTerminate` function that we call at the end of the main function.

```
glfwTerminate();
return 0;
```

This will clean up all the resources and properly exit the application. Now try to compile your application and if everything went well you should see the following output:



We also want to have some form of **input** control in GLFW and we can achieve this with several of GLFW's input functions. We'll be using GLFW's `glfwGetKey` function that takes the window as input together with a key. The function returns whether this key is currently being pressed. We're creating a processInput function to keep all input code organized:

```
void processInput(GLFWwindow *window)
{
    if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
}
```

Here we check whether the user has pressed the escape key (if it's not pressed, `glfwGetKey` returns `GLFW_RELEASE`). If the user did press the escape key, we close GLFW by setting its WindowShouldClose property to true using `glfwSetwindowShouldClose`. The next condition check of the main while loop will then fail and the application closes.

We then call processInput every iteration of the render loop:

```
while (!glfwWindowShouldClose(window))
{
    processInput(window);

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

This gives us an easy way to check for specific key presses and react accordingly every frame. An iteration of the render loop is more commonly called a frame.

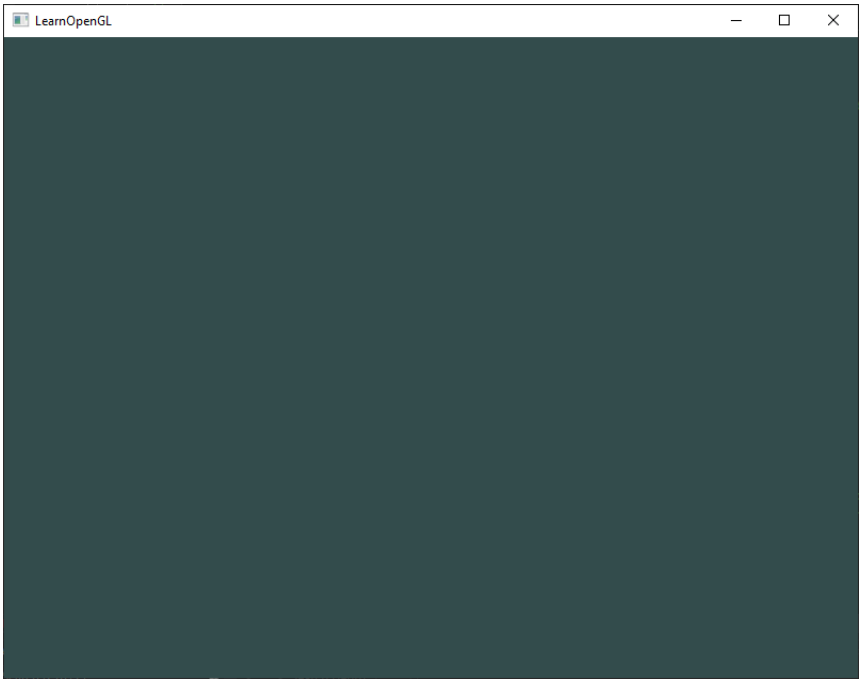
We want to place all the rendering commands in the render loop, since we want to execute all the rendering commands each iteration or frame of the loop. This would look a bit like this:

```
// render loop
while(!glfwWindowShouldClose(window))
{
    // input
    processInput(window);

    // rendering commands here
    ...

    // check and call events and swap the buffers
    glfwPollEvents();
    glfwSwapBuffers(window);
}
```

Just to test if things actually work we want to clear the screen with a color of our choice. At the start of frame we want to clear the screen. Otherwise we would still see the results from the previous frame (this could be the effect you're looking for, but usually you don't). We can clear the screen's color buffer using glClear where we pass in buffer bits to specify which buffer we would like to clear. The possible bits we can set are `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT` and `GL_STENCIL_BUFFER_BIT`. Right now we only care about the color values so we only clear the color buffer.

```
glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
```

Note that we also specify the color to clear the screen with using `glClearColor`. Whenever we call glClear and clear the color buffer, the entire color buffer will be filled with the color as configured by `glClearColor`. This will result in a dark green-blueish color.



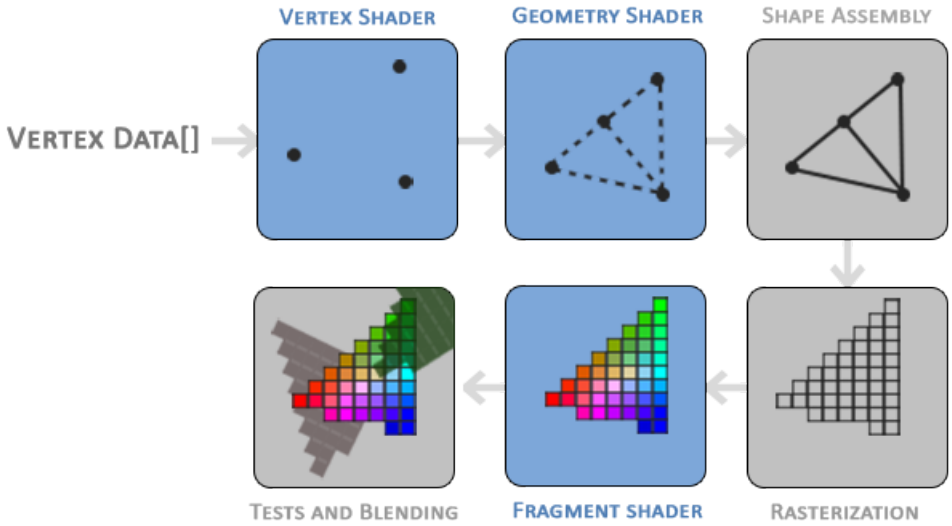
1.3 Hello Triangle

In OpenGL everything is in 3D space, but the screen or window is a 2D array of pixels so a large part of OpenGL's work is about transforming all 3D coordinates to 2D pixels that fit on your screen. The process of transforming 3D coordinates to 2D pixels is managed by the **graphics pipeline** of OpenGL. The graphics pipeline can be divided into two large parts: the first transforms your 3D coordinates into 2D coordinates and the second part transforms the 2D coordinates into actual colored pixels. In this chapter we'll briefly discuss the graphics pipeline and how we can use it to our advantage to create fancy pixels.

The graphics pipeline takes as input a set of 3D coordinates and transforms these to colored 2D pixels on your screen. The graphics pipeline can be divided into several steps where each step requires the output of the previous step as its input. All of these steps are highly specialized (they have one specific function) and can easily be executed in parallel. Because of their parallel nature, graphics cards of today have thousands of small processing cores to quickly process your data within the graphics pipeline. The processing cores run small programs on the GPU for each step of the pipeline. These small programs are called shaders.

Some of these shaders are configurable by the developer which allows us to write our own shaders to replace the existing default shaders. This gives us much more fine-grained control over specific parts of the pipeline and because they run on the GPU, they can also save us valuable CPU time. Shaders are written in the **OpenGL Shading Language (GLSL)** and we'll delve more into that in the next chapter.

Below you'll find an abstract representation of all the stages of the graphics pipeline. Note that the blue sections represent sections where we can inject our own shaders.



As you can see, the graphics pipeline contains a large number of sections that each handle one specific part of converting your vertex data to a fully rendered pixel. We will briefly explain each part of the pipeline in a simplified way to give you a good overview of how the pipeline operates.

As input to the graphics pipeline we pass in a list of three 3D coordinates that should form a triangle in an array here called `Vertex Data`; this vertex data is a collection of vertices. A vertex is a collection of data per 3D coordinate. This vertex's data is represented using vertex attributes that can contain any data we'd like, but for simplicity's sake let's assume that each vertex consists of just a 3D position and some color value.

In order for OpenGL to know what to make of your collection of coordinates and color values OpenGL requires you to hint what kind of render types you want to form with the data. Do we want the data rendered as a collection of points, a collection of triangles or perhaps just one long line? Those hints are called primitives and are given to OpenGL while calling any of the drawing commands. Some of these hints are `GL_POINTS`, `GL_TRIANGLES` and `GL_LINE_STRIP`.

Vertex Shader

The first part of the pipeline is the vertex shader that takes as input a single vertex. The main purpose of the vertex shader is to transform 3D coordinates into different 3D coordinates (more on that later) and the vertex shader allows us to do some basic processing on the vertex attributes.

Geometry Shader

The output of the vertex shader stage is optionally passed to the geometry shader. The geometry shader takes as input a collection of vertices that form a primitive and has the ability to generate other shapes by emitting new vertices to form new (or other) primitive(s). In this example case, it generates a second triangle out of the given shape.

Primitive Assembly

The primitive assembly stage takes as input all the vertices (or vertex if `GL_POINTS` is chosen) from the vertex (or geometry) shader that form one or more primitives and assembles all the point(s) in the primitive shape given; in this case two triangles.

Rasterization Stage

The output of the primitive assembly stage is then passed on to the rasterization stage where it maps the resulting primitive(s) to the corresponding pixels on the final screen, resulting in fragments for the fragment shader to use. Before the fragment shaders run, clipping is performed. **Clipping** discards all fragments that are outside your view, increasing performance.

Fragment Shader

The main purpose of the fragment shader is to calculate the final color of a pixel and this is usually the stage where all the advanced OpenGL effects occur. Usually the fragment shader contains data about the 3D scene that it can use to calculate the final pixel color (like lights, shadows, color of the light and so on).

Alpha Tests and Blending Stage

After all the corresponding color values have been determined, the final object will then pass through one more stage that we call the **alpha test** and **blending** stage. This stage checks the corresponding depth (and stencil) value (we'll get to those later) of the fragment and uses those to check if the resulting fragment is in front or behind other objects and should be discarded accordingly. The stage also checks for alpha values (alpha values define the opacity of an object) and blends the objects accordingly. So even if a pixel output color is calculated in the fragment shader, the final pixel color could still be something entirely different when rendering multiple triangles.

As you can see, the graphics pipeline is quite a complex whole and contains many configurable parts. However, for almost all the cases we only have to work with the **vertex** and **fragment** shader. The geometry shader is optional and usually left to its default shader.

In modern OpenGL we are **required** to define at least a vertex and fragment shader of our own (there are no default vertex/fragment shaders on the GPU). For this reason it is often quite difficult to start learning modern OpenGL since a great deal of knowledge is required before being able to render your first triangle. Once you do get to finally render your triangle at the end of this chapter you will end up knowing a lot more about graphics programming.

To start drawing something we have to first give OpenGL some input vertex data. OpenGL is a 3D graphics library so all coordinates that we specify in OpenGL are in 3D (x, y and z coordinate). OpenGL doesn't simply transform all your 3D coordinates to 2D pixels on your screen; OpenGL only processes 3D coordinates when they're in a specific range between -1.0 and 1.0 on all 3 axes (x, y and z). All coordinates within this so called **normalized device coordinates** range will end up visible on your screen (and all coordinates outside this region won't).

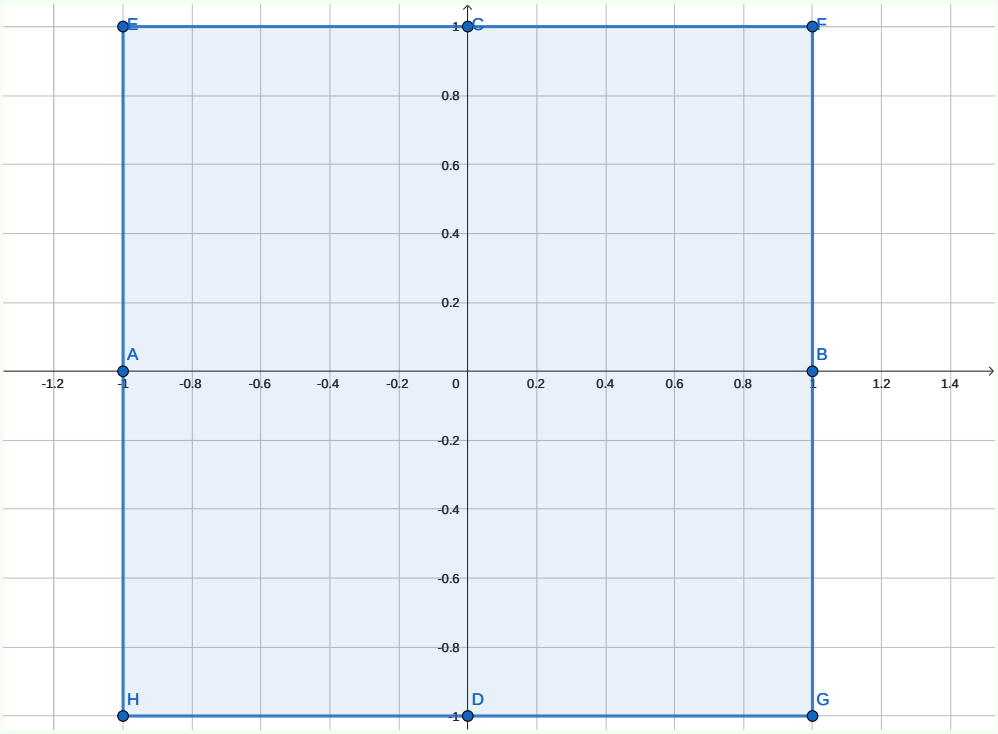
Because we want to render a single triangle we want to specify a total of three vertices with each vertex having a 3D position. We define them in normalized device coordinates (the visible region of OpenGL) in a float array:

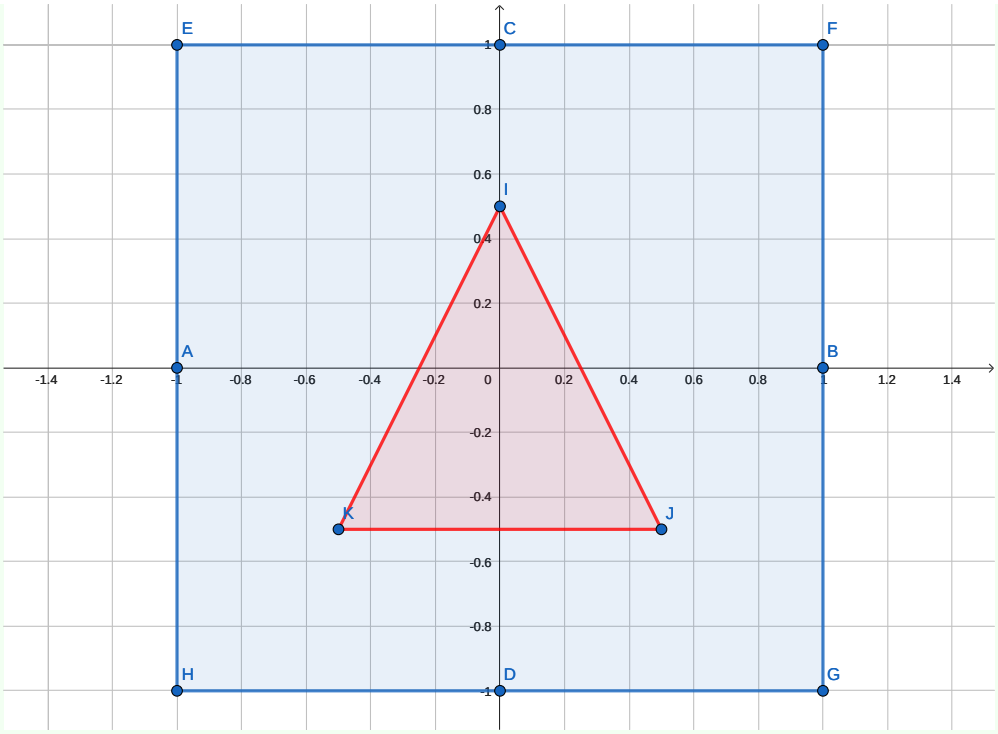
```
float vertices[] = {
    -0.5f, -0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
    0.0f, 0.5f, 0.0f
};
```

Because OpenGL works in 3D space we render a 2D triangle with each vertex having a z coordinate of 0.0. This way the depth of the triangle remains the same making it look like it's 2D.

Normalized Device Coordinates (NDC)

Once your vertex coordinates have been processed in the vertex shader, they should be in normalized device coordinates which is a small space where the x, y and z values vary from -1.0 to 1.0. Any coordinates that fall outside this range will be discarded/clipped and won't be visible on your screen. Below you can see the triangle we specified within normalized device coordinates (ignoring the z axis):





Unlike usual screen coordinates the positive y-axis points in the up-direction and the (0,0) coordinates are at the center of the graph, instead of top-left. Eventually you want all the (transformed) coordinates to end up in this coordinate space, otherwise they won't be visible.

Your NDC coordinates will then be transformed to screen-space coordinates via the viewport transform using the data you provided with `glViewport`. The resulting screen-space coordinates are then transformed to fragments as inputs to your fragment shader.

With the vertex data defined we'd like to send it as input to the first process of the graphics pipeline: the vertex shader. This is done by creating memory on the GPU where we store the vertex data, configure how OpenGL should interpret the memory and specify how to send the data to the graphics card. The vertex shader then processes as much vertices as we tell it to from its memory.

We manage this memory via so called **vertex buffer objects (VBO)** that can store a large number of vertices in the GPU's memory. The advantage of using those buffer objects is that we can send large batches of data all at once to the graphics card, and keep it there if there's enough memory left, without having to send data one vertex at a time. Sending data to the graphics card from the CPU is relatively slow, so wherever we can we try to send as much data as possible at once. Once the data is in the graphics card's memory the vertex shader has almost instant access to the vertices making it extremely fast.

Just like any object in OpenGL, this buffer has a unique ID corresponding to that buffer, so we can generate one with a buffer ID using the `glGenBuffers` function:

```
unsigned int VBO;
glGenBuffers(1, &VBO);
```

OpenGL has many types of buffer objects and the buffer type of a vertex buffer object is `GL_ARRAY_BUFFER`. OpenGL allows us to bind to several buffers at once as long as they have a different buffer type. We can bind the newly created buffer to the `GL_ARRAY_BUFFER` target with the `glBindBuffer` function:

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

From that point on any buffer calls we make (on the `GL_ARRAY_BUFFER` target) will be used to configure the currently bound buffer, which is VBO. Then we can make a call to the `glBufferData` function that copies the previously defined vertex data into the buffer's memory:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

`glBufferData` is a function specifically targeted to copy user-defined data into the currently bound buffer. Its first argument is the type of the buffer we want to copy data into: the vertex buffer object currently bound to the `GL_ARRAY_BUFFER` target. The second argument specifies the size of the data (in bytes) we want to pass to the buffer; a simple `sizeof` of the vertex data suffices. The third parameter is the actual data we want to send. The fourth parameter specifies **how we want the graphics card to manage the given data**. This can take 3 forms:

- `GL_STREAM_DRAW`: the data is set only once and used by the GPU at most a few times.
- `GL_STATIC_DRAW`: the data is set only once and used many times.
- `GL_DYNAMIC_DRAW`: the data is changed a lot and used many times.

The position data of the triangle does not change, is used a lot, and stays the same for every render call so its usage type should best be `GL_STATIC_DRAW`. If, for instance, one would have a buffer with data that is likely to change frequently, a usage type of `GL_DYNAMIC_DRAW` ensures the graphics card will place the data in memory that allows for faster writes.

As of now we stored the vertex data within memory on the graphics card as managed by a vertex buffer object named VBO. Next we want to create a vertex and fragment shader that actually processes this data, so let's start building those.

The first thing we need to do is write the vertex shader in the shader language GLSL (OpenGL Shading Language) and then compile this shader so we can use it in our application. Below you'll find the source code of a very basic vertex shader in GLSL:

```
#version 330 core
layout (location = 0) in vec3 aPos;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```