The Basics of OpenGL

Caden Henrich

March 5, 2020

Contents

1 Opening a Window

Opening a window in OpenGL consists of the following steps:

1. Import GLFW (and optionally GLEW)
2. Initialize the library
3. Create a windowed mode window and its OpenGL context
4. Make the window's context current
   (a) For modern OpenGL, initialize GLEW
5. Loop until the user closes the window
   (a) Clear the screen
   (b) Draw to the screen
   (c) Swap the front and back buffers
   (d) Poll for and process events
6. Once done looping, terminate GLFW and return

1.1 Importing GLFW (and optinally GLEW)

Although GLFW is needed to open a window, GLEW is not necessary until actually using modern OpenGL to draw to the screen. Examples of how to import both are shown below.

GLFW

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

GLFW/GLEW

```
1 #include <GLFW/glfw3.h>
2 #include <GL/glew.h>
```

Only importing the libraries is not enough, however; they must first be initialized before use.

1.2 Intializing the Libraries

Initialization of the libraries requires both calling their initialization functions, and checking their outputs to make sure the libraries initialized successfully.

GLFW can be initialized as follows:

```
1 if (!glfwInit())
2     return -1;
```

The return statement on line 2 can be replaced or preceeded by a message printing to the console describing the error and/or possible fixes. GLEW's initialization must take place after there is a valid and current OpenGL context. See section for more information.

1.3 Creating a Window and Context

First, a `GLFWwindow` object pointer is needed to reference the window itself. This declaration can be just before or after GLFW initialization.

```
GLFWwindow* window;
```

Next, the window pointer needs to be initialized to an instance of a windowed-mode window and its OpenGL context. It is also helpful to check afterwards if the window has been successfully created, and print any debug messages and/or terminate if it hasn't.

```
window = glfwCreateWindow(800, 600, "Title", NULL, NULL);
if (!window)
{
    /* Add any error messages / debug logs, etc. */
    glfwTerminate();
    return -1;
}
```

In this case, no error message was printed, but the `<stdio.h>` or `<iostream>` header files could be used to print such messages to the console.

Note: The glfwCreateWindow function takes parameters (width, height, title, display, share), the final two of which can usually be safely set to NULL.

1.4 Making the Window's Context Current

In order to be able to draw to the window, the context associated with the window needs to be made current. This can be done through one simple function call.

```
glfwMakeContextCurrent(window);
```

If there is no need to use GLEW, then the next step can be skipped. However, if GLEW is required, it should be initialized just after the context is made current. This process is similar to initializing GLFW, except this time its initialization function is checked against a GLEW-specific enum. GLEW also provides the `glewGetErrorString()` function. This can be used to print the specific error if GLEW isn't properly initialized using something like stdio's `fprintf()` function.

```
GLenum err = glewInit();
if (GLEW_OK != err)
{
    /* Problem: glewInit failed, something is seriously wrong. */
    fprintf(stderr, "Error: %s\n", glewGetErrorString(err));
    ...
}
fprintf(stdout, "Status: Using GLEW %s\n", glewGetString(GLEW_VERSION));
```

GLEW's initialization must take place *after and only after* there is a valid, current context. Doing otherwise will result in an error and the application crashing.

1.5 Looping Until the User Closes the Window

This while loop will draw every frame, and will loop until the user presses the 'x' button or closes the window in any other way. The loop will wipe the screen clear, draw anything that needs to be drawn tothe backbuffer, swap the buffers, and then poll the window for any events.

```
while(!glfwWindowShouldClose(window))
{
    glClear(GL_COLOR_BUFFER_BIT);

    /* Draw to the screen here */

    glfwSwapBuffers(window);

    glfwPollEvents();
}
```

1.6 Terminating GLFW and Exiting the Program

This step is run after the user closes the window. The GLFW instance needs to be terminated and cleaned up, and the `main()` method needs to return. *Note: GLEW does not need to be terminated in the same way.*

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

After this step, the program should execute properly and open a blank window, ready to be drawn to using modern OpenGL. The final code should read as follows:

```c
#include <GLFW/glfw3.h>
#include <GL/glew.h>
#include <stdio.h>

int main(void)
{
    /* Declare a window variable */
    GLFWwindow* window;

    /* Initialize GLFW */
    if (!glfwInit())
        return -1;

    /* Create a windowed mode window and its OpenGL context */
    window = glfwCreateWindow(800, 600, "Title", NULL, NULL)
    if (!window)
    {
        glfwTerminate();
        return -1;
    }

    /* Make the window's context current */
    glfwMakeContextCurrent(window);

    /* Initialize GLEW */
    GLenum err = glewInit();
    if (GLEW_OK != err)
    {
        /* Problem: glewInit failed, something is seriously wrong. */
        fprintf(stderr, "Error: %s\n", glewGetErrorString(err));
    }
    fprintf(stdout, "Status: using GLEW %s\n", glewGetString(GLEW_VERSION));

    /* Loop until the user closes the window */
    while (!glfwWindowShouldClose(window))
    {
        /* Clear the buffer */
        glClear(GL_COLOR_BUFFER_BIT);

        /* Draw to the backbuffer here */

        /* Swap front and back buffers */
        glfwSwapBuffers(window);

        /* Poll for and process events */
        glfwPollEvents();
    }

    /* Terminate GLFW and the program */
    glfwTerminate();
    return 0;
}
```

2 Drawing a Basic Triangle

After a window is opened, the next step is to draw to the screen. To do this, the following need to be done:

1. Create an array to store vertex positions
2. Generate a vertex buffer
3. Assign the vertex buffer's attributes
4. Draw the vertex buffer

After all these steps have been completed, a triangle should be successfully drawn to the screen.

2.1 Creating an Array

This step follows basic C++ guidelines on how to create an array. This array should be an array of floats, since the coordinates will be in clip-space. Such an array might look like this:

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

These positions define the vertex positions of a regular, isosceles triangle. Currently, however, they are on the CPU side of the program. In order to get them to the GPU, they need to be put into a buffer.

2.2 Generating a Vertex Buffer

This step and the following steps will make use of modern OpenGL. This means using the GLEW package imported earlier to execute modern OpenGL function calls. One of these functions is `glGenBuffers()`, which takes as parameters a GLsizei n → number of buffers to be generated, and a GLuint *buffers → variable or array in which the buffer ID(s) should be stored. In this case, there is only one buffer needed, and therefore the second argument will be a variable of type unsigned int. This variable can be used to reference the buffer and its data later on. After creation of the variable, the aforementioned `glGenBuffers()` function can be called, and a buffer will be generated.

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

This newly generated buffer only has the data specified in the `positions[]` array, and has no information on how to interpret the data, thus it cannot be drawn to the screen. The next step will cover how to prepare the buffer for drawing.

2.3 Assigning the Vertex Buffer's Attributes

Since OpenGL is a state machine, before the attributes of the newly created buffer can be manipulated, the buffer needs to be bound. This binding will tell OpenGL the type of buffer being bound, and the ID of that buffer.

```
glBindBuffer(GL_ARRAY_BUFFER, buffer);
```

Now that the buffer is bound, OpenGL needs to be told what the data is inside the buffer, and how much.This information should include the buffer ID, the size of the buffer, and the indended usage of the buffer. For now, `GL_STATIC_DRAW` is fine, because the buffer's contents are modified once and then repeatedly drawn to the screen every frame.

```
glBufferData(GL_ARRAY_BUFFER, 6 * sizeof(float), nullptr, GL_STATIC_DRAW);
```

The above code uses the `sizeof()` function to determine the actual size of the data being sent to the GPU. This is recommended practice. The size of a float is also being multiplied by 6 since that is the number of floats in the `positions[]` array.

After telling OpenGL what data is in the buffer, it needs to be told how to interpret that data. This is done through vertex attributes. To set the vertex attributes, first the attribute array needs to be enabled, then the `glVertexAttribPointer()` function can be used to tell OpenGL how to interpret the data.

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), 0);
```

This code enables the array at index 0 (the only array, currently), then tells OpenGL that, at that index, there will be 2 arguments of type `GL_FLOAT`, they are not normalized, and there are two floats worth of data between each vertex. There is also 0 offset for the first vertex from the beginning of the array.

2.4 Drawing the Vertex Buffer

Now that all the required steps have been taken to set up the buffer, the buffer can be drawn within the `while` loop made in the `main()` function to render each frame.

```
while (!glfwWindowShouldClose(window))
{
    /* Clear the buffer */
    glClear(GL_COLOR_BUFFER_BIT);

    /* Draw the array */
    glDrawArrays(GL_TRIANGLES, 0, 3);

    /* Swap front and back buffers */
    glfwSwapBuffers(window);

    /* Poll for and process events */
    glfwPollEvents();
}
```

Notice the buffer to draw is not specified anywhere in the `while` loop. This is because the buffer is already bound, and since OpenGL is a state machine (as mentioned before), the currently bound buffer is the one drawn to the screen. Finally, a white triangle should be drawn to the screen.