Visual Computing

Lab Exercises

Samuel Silva

October 26, 2021

Contents

Hello Triangle! 1

For today's class, the goal is to provide you with a first contact with OpenGL in python. You should not aim to do it as fast as you can, just copy-pasting the code in this guide, but understanding what you are doing throughout. There is a lot going on in the code and you need to understand why it needs to happen. Do not mind if you cannot memorize the instructions. Just remember their purpose. Additionally, if you are new to python, take you time to understand the program structure and ask for assistance, whenever necessary. In this class you will:

- configure the environment for OpenGL development
- create and customize a window to support OpenGL rendering
- add support to detect and react to events
- create a basic **graphics pipeline**
- render a triangle
- have a first contact with shaders
- experiment with different rendering parameters

1.1 Environment Specificity

[*] The first step required is to set the environment that will be used to program using OpenGL. You need to install the Python interpreter, an editor (we advise pyCharm) and the packages `pyOpenGL` and `pygame`. If everything goes well, the packages will be installed automatically, for you, the first time you try to use them in PyCharm.

1.2 Viewport Formulation

For rendering, OpenGL first needs a rendering context. In our case, we will resort to the package `pygame` to create a window for that purpose. So, let's get started. Create a new Python file and add the following code:

There are several options that can be used, such as glFW. However, this also serves for you to have a first contact with a package that is actually a 2D game engine.

```
import pygame as pg
from pygame.locals import *

pg.init()
windowsize = (640, 480)
pg.display.set_mode(windowsize,  DOUBLEBUF | OPENGL | RESIZABLE)
```

Basically, the code imports the `pygame` package, initializes `pygame` and creates a window. The parameters for `set_mode` are self-explanatory and tell `pygame` to create a window for OpenGL that can be resizeable and supports double buffering.

We will get into **double buffering**, later, and talk about why it is important when rendering computer graphics.

If you run your code, as is, you will probably see a window for a very short time and, them the program will end. Not much use for it this way,

[*] I may have named these sections adopting a "The Big Bang Theory" style...

right? This is because there is nothing that would keep the application going and waiting for things to be drawn in the window: an event loop.

1.3 Eventful Celerity

To create an event loop, you can type the code below just after your code:

```
while True:
    for event in pg.event.get():
        if event.type == pg.QUIT:
            pg.quit()
```

This code is using the event manager from `pygame` to listen to events inside an infinite loop. Every time the cycle runs, the program checks if there is any event available. If it does, in this particular case it checks if it is `pg.QUIT`, which means that the application received a quit signal (for instance, the user closed the window). If that is the case, the application wraps things up by closing `pygame` and leaving the application. Now, run your program, again. This time, the window will stay open.

1.4 Windowsill Observation

Now that we have a window supporting OpenGL, it is time to make our first (simple) lines of OpenGL code. First, we need to add support to OpenGL by importing the pyOpenGL package:

Then, after the line creating the window, add the following line

```
glClearColor(0.0, 1.0, 0.0, 1.0)
```

Notice two things, in this command: it starts with **`gl`** meaning it is part of the OpenGL package; and it takes four parameters corresponding to the red, green, blue, and alpha components of the color to use when clearing the window.

All these vary between 0.0 and 1.0, with the **alpha** component concerning the level of opacity of the color with 0.0 standing for fully transparent and 1.0 for fully opaque

- ▶ Can you say what color is being set?

Now run the program. Yes, the window is still black. We just defined what color to use to clear the window, but now we need to tell OpenGL to clear it. Inside the `while` loop, add the following line:

```
    glClear(GL_COLOR_BUFFER_BIT)
```

This tells OpenGL to clear the color buffer and the color that will be used is the one defined earlier. Now, if you run the program you will see... a black window. Now is the time to talk about double buffers.

The color buffer is where the color for each pixel is stored

When an OpenGL scene is drawn, it requires the execution of multiple commands, e.g., to draw lines or surfaces, and this is not instantaneous, which means that, eventually, we could perceive the scene being drawn or some artefacts related with it. To avoid this, OpenGL always draws to a hidden canvas (buffer) and, the scene is only shown after it is complete by swapping it with the current (visible) buffer. So, double buffer refers to having two buffers that are used for rendering and alternate in visibility.

Therefore, when we perform a `glClear`, we are actually clearing the hidden buffer. Now, we need to say that it is ready for showing. To this effect, add this line after the `glClear` line:

```
pg.display.flip()
```

It basically tells `pygame` to flip to the hidden buffer. Run the program. You have your first OpenGL code running!

Now experiment with changing some details in your program:

- ▶ change the size of the window
- ▶ change the background color
- ▶ move the `glClear` instruction outside the `while` loop, just after the `glClearColor`. What happened with the window? Can you explain why it happens?

If you had any issue in completing the preceding steps – and only as a last resort – you can check a basic working example in **`OpenGL_Ex1.py`**

1.5 Faraway Triangularity

Before you see your first OpenGL triangle being properly rendered on the window you just created, we still have some steps to go. Remember, we need to implement the different stages of the **graphics pipeline**.

- ▶ define the vertex data and get it into the GPU
- ▶ define and compile a vertex shader
- ▶ define and compile a fragment shader
- ▶ load the shaders into the GPU
- ▶ execute the rendering of the vertex data

Open the file **`OpenGL_Ex2.py`**. It already has some code including code to create a window and a place where to put code for rendering, in the future. It is packed inside the class `VCHelper`, the Visual Computing Helper, where we will put some of the core code for working with OpenGL, over time.

Take a look at the code and identify the parts your are already familiar with. and check where the functions are called.

Look for:

```
def init_window(self, w, h
):
def render(self):
```

Data Specificity

There is an empty function called `init_vertex_buffers`. Inside, start by setting the data required to define a triangle by using the code below. Can you understand what these numbers represent?

```
        self.triangle_vertices = array([
                [-0.5, 1, 0],
                [-1, -1, 0],
                [0, -1, 0]
            ], 'f')
```

This defines the data to draw a triangle, but we need to move it into the GPU, first. To this end, we need to create a **vertex buffer object (VBO)**. Just below the definition of the triangle, type the following:

```
        self.vbo_triangle = glGenBuffers(1)
        glBindBuffer(GL_ARRAY_BUFFER, self.vbo_triangle)
        glBufferData(GL_ARRAY_BUFFER, 4 * self.triangle_vertices.
    size, self.triangle_vertices, GL_STATIC_DRAW)
```

Let us go through it, to understand what is being done. First, we ask to generate a buffer and place the ID on `self.vbo_triangle`. In the second line, we bind the buffer we just created to be a buffer to contain an array.

Binding a buffer is basically stating that in what follows, whatever is done is considering this **vertex buffer object**. So, when we run the `glBufferData` instruction, we know that data will be available whenever I bind into the `self.vbo_triangle` VBO.

Finally, on the third line, we copy the triangle data into the buffer. Note that it requires stating the size of the array in bytes (equals the number of items times 4 bytes per item, because they are floats), and the variable where you placed the data, `self.triangle_vertices`. Finally, the last argument, `GL_STATIC_DRAW`, just tells the GPU that we will not change the triangle data (i.e., its shape will not change, over time). And, that's it, the triangle data is on the GPU.

Telling that we will not change the data, over time, helps the GPU optimize where to store it. If that is not the case, we can state `GL_DYNAMIC_DRAW`.

Triangular Apparition

Now, we can try and check if we are able to render the triangle. So, find the function where the render code is to be placed and, just after the `glClear` instruction, type the following:

```
        glBindBuffer(GL_ARRAY_BUFFER, self.vbo_triangle)
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * 4,
    None)
        glEnableVertexAttribArray(0)
        glDrawArrays(GL_TRIANGLES, 0, 3)
```

Here, we are binding to the VBO created earlier, where the vertex data has been stored and, in the second line, we are stating that the vertex attribute in **position 0**, has an item size of 3 (i.e., three values per vertex), their type is float (`GL_FLOAT`), it is not normalized (`GL_FALSE`), the stride is 3 items at 4 bytes each, and no offset is required, i.e., the first vertex starts at the beginning of the buffer.

You will understand the purpose and meaning of this position when we get into shaders

The **stride** tells the GPU how many bytes to jump to get to the next element. In this case, we have a list of vertex coordinates. So, each three values is a set of coordinates. So, to jump to the next coordinate we just add 3 * 4 bytes. This may seem unnecessary work, as it is rather obvious, but it is very important when an array has multiple attributes in sequence. For instance, one could have [x y z **r g b** x y z...], with color just after each set of coordinates. In that case, the stride would be different from the size of a vertex.

The third line is just telling the GPU that the attribute defined at position 0 (the vertex coordinates) is to be enabled, i.e., considered for rendering. And, the last line just tells the GPU to draw the data considering it is a list of triangles (GL_TRIANGLES), starting from the first vertex in the data (0) and use 3 vertices (actually the total number of the list we provided).

Now, if you try to run your code it will give you an error. This is because we are actually not running the init_vertex_buffers code, yet, meaning that there is not data in the buffer. To do so, just before the `while` loop and after initializing the window write:

```
cv.init_vertex_buffers()
```

Now, run the program and, hopefully, you should see a skinny white triangle. The file `OpenGL_Ex2_Final.py` has the full implementation of what is described above. *It is not intended for you to go there and copy-paste contents* to reach the solution, but to serve as a reference, in case you run into trouble and need some hints.

1.6 Shader Shaping

While we already have a triangle being drawn on screen, we still have not implemented the basic aspects of the graphics pipeline. This only worked due to the fact that the OpenGL driver, when none is defined, adopts a (very) basic configuration for the vertex and fragment shaders. The GPU has no default shaders installed. However, as it is, we would not be able to do much more with our scene. So, we need to understand and define the shaders. The code in class VCHelper already has basic vertex and fragment shaders defined in `init_shaders`.

For now, just explore the code and try to understand the overall logic of defining and compiling the shaders into a program. In the next class we will get a bit more into the function of shaders and how they are programmed.

Then, to use them, you still need to perform two actions. First, add a call to init_shaders just before the while loop. This is to actually run the code to create the shaders. And, finally, just before asking the GPU to draw the array, in the `render` function, you need to tell it what **shader program** to use:

```
glUseProgram(self.shader)
```

Now, if you run the program, you will notice that the triangle has a different color. You now have a basic graphics pipeline from vertex data to

1.7 The Chaotic Experimentation

Now that you already have a basic rendering example working, you can experiment with some of the parameters and code.

- ▶ Can you tell where the triangle color is being defined? Can you change it to a color of your choice?
- ▶ We used GL_TRIANGLES in glDrawArray, but it supports many more modes: `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_-LOOP`, GL_TRIANGLE_STRIP, etc.
- ▶ Experiment with changing the triangle coordinates. Note that, at this time, the visible area is limited to -1 to 1 in all axes. We will change this in a future class;
- ▶ Can you draw two triangles? What about a square? Explore a `drawArray` mode that allows the least amount of data to do it.
- ▶ Try to explore the key events and modify the code so that you can move the top vertex of the triangle with the arrow keys, thus modifying the triangle's shape in realtime. For this to happen, please note that if the vertices move, then you need to copy the data into the GPU buffer before rendering. What changes are needed?

1.8 Shader Mingling

In the experimentations above, when you used `GL_POINTS` as the drawing mode you probably noticed how small the points were. This property can actually be customized using the vertex shader. To this effect, you need to add the following line to the **vertex shader**, inside **main()**:

```
gl_PointSize = 5;
```

And, then, you need to tell OpenGL that you want to be able to change point size in the vertex shader. To do it, add the following to the init_-window function, just after defining the clearing color:

```
        glEnable(GL_VERTEX_PROGRAM_POINT_SIZE)
```