Advanced Computer Graphics

2 - Introduction to OpenGL

Yoonsang Lee
Fall 2018

Today's Topics

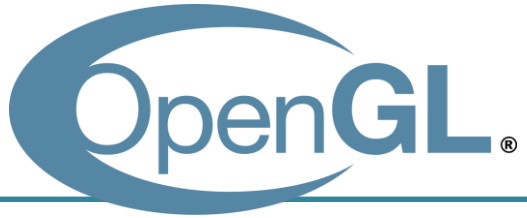
- What is OpenGL?
- OpenGL basics
- GLFW input handling
- Legacy OpenGL & Modern OpenGL
- OpenGL as a Learning Tool

What is OpenGL?

- **Op**en **G**raphics **L**ibrary

- OpenGL is an **API** (Application Programming Interface) for graphics programming
  - Unlike its name, OpenGL is not a library.

What is OpenGL?



- **API** is a **specification**
  - API describes **interfaces** and **expected behavior**

- As for OpenGL API,
  - OS vendors provide OpenGL interface (e.g. opengl32.dll on windows)
  - GPU vendors provide OpenGL implementation, the graphic card driver (e.g. Nvidia drivers)

Characteristics of OpenGL

- Cross platform
  - You can use OpenGL on Windows, OS X, Linux, iOS, Android, ...
- Language independent
  - OpenGL has many language bindings (C, Python, Java, Javascript, ...)
  - We'll use its Python binding in this class - PyOpenGL

So, what can we do with OpenGL?

- **Just only drawing things**
  - Provides small, but powerful set of low-level drawing operations
  - No functions for creating windows & OpenGL contexts, handling events (we'll discuss the "context" later)

- Thus, additional utility libraries are required to use OpenGL
  - GLFW, FreeGLUT : Simple utility libraries for OpenGL
  - Fltk, wxWigets, Qt, Gtk : General purpose GUI framework

Utility Libraries for Learning OpenGL

- General GUI frameworks(e.g. Qt) are powerful, but too heavy for just learning OpenGL.

- GLUT “was” most popular for this purpose
  - But it’s outdated and unmaintained.
  - Its open-source clone FreeGLUT is mostly concerned with providing a stable clone of GLUT.

- Now, GLFW is getting more popular.
  - Provides much fine control for managing windows and events.
  - So GLFW is our choice for this class.

Python/OpenGL environment for this class

- Python 3.5+

- + additional python modules

Install Additional Modules

- We'll use a few python modules in this class
  - NumPy, PyOpenGL, glfw
- My recommendation for installing python modules is using **pip** (Python Package Index)
- NumPy
  - Windows

```
> py -3 -m pip install numpy
```

  - Ubuntu

```
# if you don't have pip, install it first.
$ sudo apt-get install python3-pip

$ python3 -m pip install numpy
```

Install Additional Modules

- PyOpenGL
  - Windows
    - Download proper *PyOpenGL-3.1.2-cp**3x**-cp**3x**m_xxx.조/* for your system from https://www.lfd.uci.edu/~gohlke/pythonlibs/#pyopengl

```
> py -3 -m pip install PyOpenGL-<version in your file>.whl
```

  - Ubuntu

```
$ python3 -m pip install PyOpenGL
```

Install Additional Modules

- GLFW
  - Windows

```
> py -3 -m pip install glfw
```

  - Ubuntu

```
$ sudo apt-get install libglfw3
$ python3 -m pip install glfw
```

Numpy

- NumPy is de-facto standard for numerical computing in Python.

- References
  - https://docs.scipy.org/doc/numpy/user/quickstart.html
  - There are numerous resources on the web. Google it!

[Practice] First OpenGL Program

```
import glfw
from OpenGL.GL import *

def render():
    pass

def main():
    # Initialize the library
    if not glfw.init():
        return
    # Create a windowed mode window and its OpenGL context
    window = glfw.create_window(640,480,"Hello World", None,None)
    if not window:
        glfw.terminate()
        return

    # Make the window's context current
    glfw.make_context_current(window)

    # Loop until the user closes the window
    while not glfw.window_should_close(window):
        # Poll events
        glfw.poll_events()

        # Render here, e.g. using pyOpenGL
        render()

        # Swap front and back buffers
        glfw.swap_buffers(window)

    glfw.terminate()

if __name__ == "__main__":
    main()
```

import X
: access X's attribute or method using X.attribute, X.method()

from X import *
: access X's attribute or method just using attribute, method()

If the python interpreter is **running this source file as the main program**, it sets the special **__name__** variable to have a value **"__main__"**.

If this file **is being imported from another module**, **__name__** will be set to **the module's name**.

[Practice] Draw a Triangle

```
def render():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glBegin(GL_TRIANGLES)
    glVertex2f(0.0, 1.0)
    glVertex2f(-1.0,-1.0)
    glVertex2f(1.0,-1.0)
    glEnd()
```

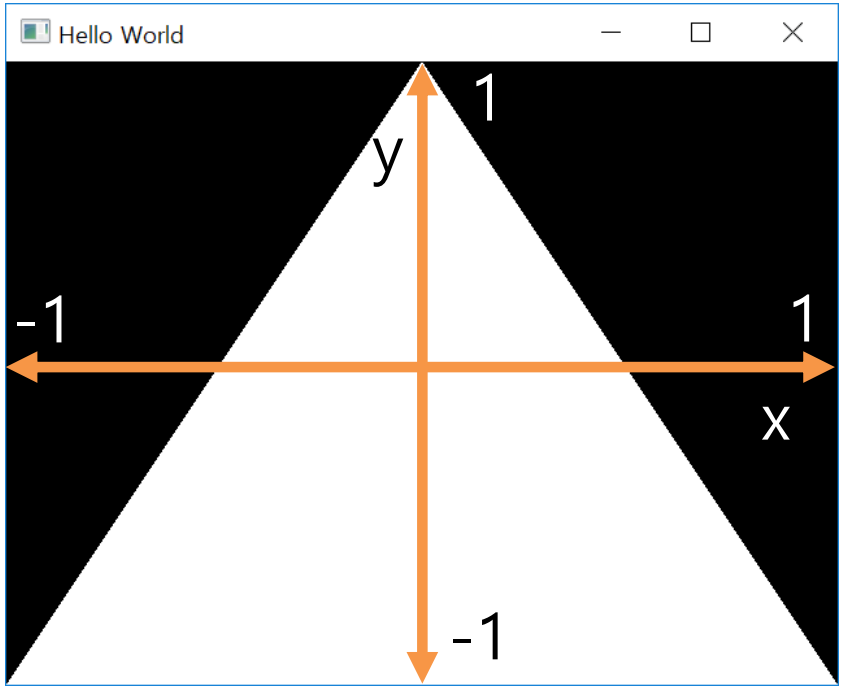
Vertex

- In OpenGL, geometry is specified by **vertices**
- To draw something, vertices have to be listed between *glBegin(primitive_type)* and *glEnd()* calls.
- *glVertex*()* specifies the coordinate values of a vertex.

```
glBegin(GL_TRIANGLES)
glVertex2f(0.0, 1.0)
glVertex2f(-1.0,-1.0)
glVertex2f(1.0,-1.0)
glEnd()
```

Coordinate System

- You can draw the triangle anywhere in a 2D square ranging from (-1, -1) to (1, 1).
- Called “Normalized Device Coordinate” (NDC)
- We’ll see how objects are transformed to NDC in later classes.

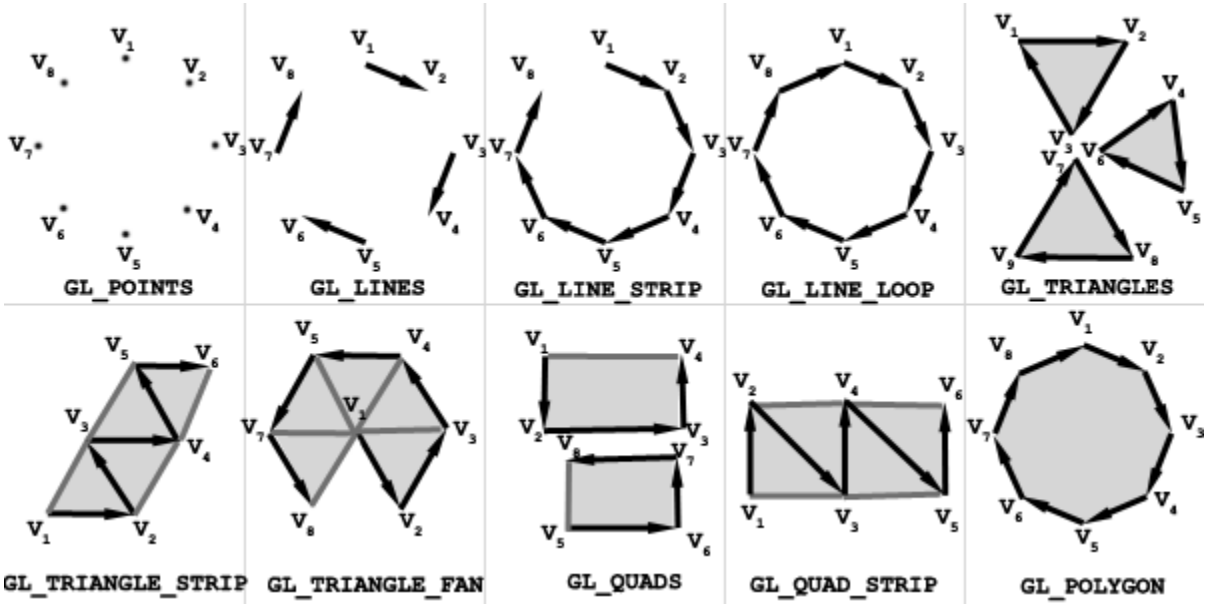


[Practice] Resize the Triangle

```
def render():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glBegin(GL_TRIANGLES)
    glVertex2f(0.0, 0.5)
    glVertex2f(-0.5,-0.5)
    glVertex2f(0.5,-0.5)
    glEnd()
```

Primitive Types

- Primitive types in *glBegin(primitive_type)* :



- They represents how vertices are to be connected.

[Practice] Change the Primitive Type

```
def render():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glBegin(GL_POINTS)
    # glBegin(GL_LINES)
    # glBegin(GL_LINE_STRIP)
    # glBegin(GL_LINE_LOOP)
    # ...
    glVertex2f(0.0, 0.5)
    glVertex2f(-0.5,-0.5)
    glVertex2f(0.5,-0.5)
    glEnd()
```

Vertex Attributes

- In OpenGL, a vertex has these attributes:
  - **Vertex coordinate** : specified by glVertex*()
  - **Vertex color** : specified by glColor*()
  - **Normal vector** : specified by glNormal*()
  - **Texture coordinate** : specified by glTexCoord*()

- (We'll see normal vector & texture coord. in later classes)

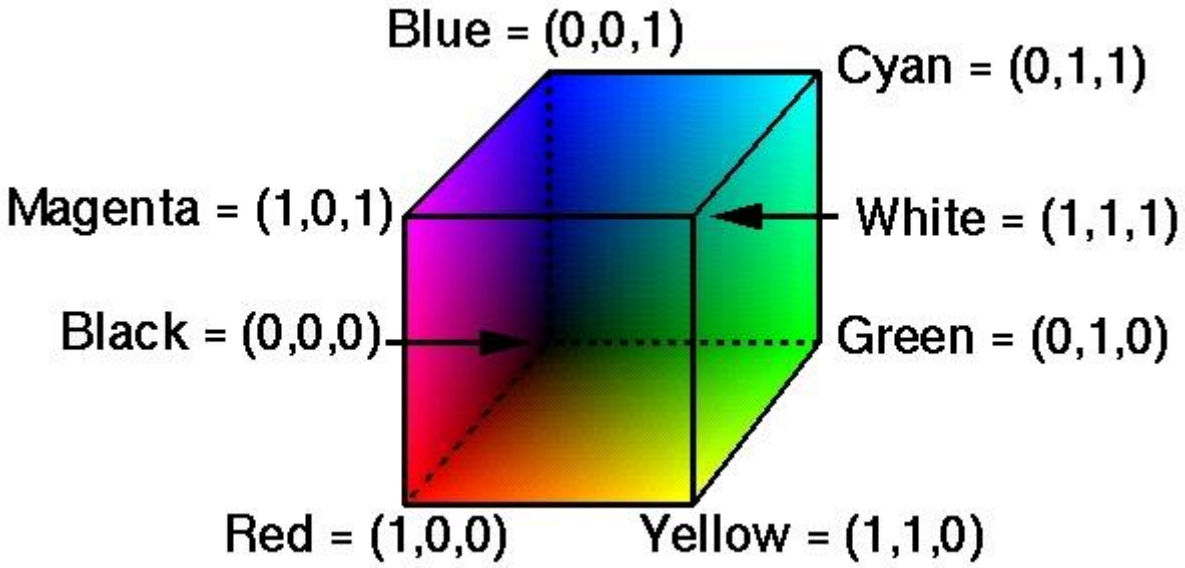
- Now, let's have a look at the vertex color.

[Practice] Colored Triangle

```
def render():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glBegin(GL_TRIANGLES)
    glColor3f(1.0, 0.0, 0.0)
    glVertex2f(0.0, 1.0)
    glColor3f(0.0, 1.0, 0.0)
    glVertex2f(-1.0,-1.0)
    glColor3f(0.0, 0.0, 1.0)
    glVertex2f(1.0,-1.0)
    glEnd()
```

Color

- OpenGL uses the RGB color model.



- Colors in interior are interpolated.

Then, how to draw a just “red” triangle?

- Set red color for each vertex?

- You can do it just by:

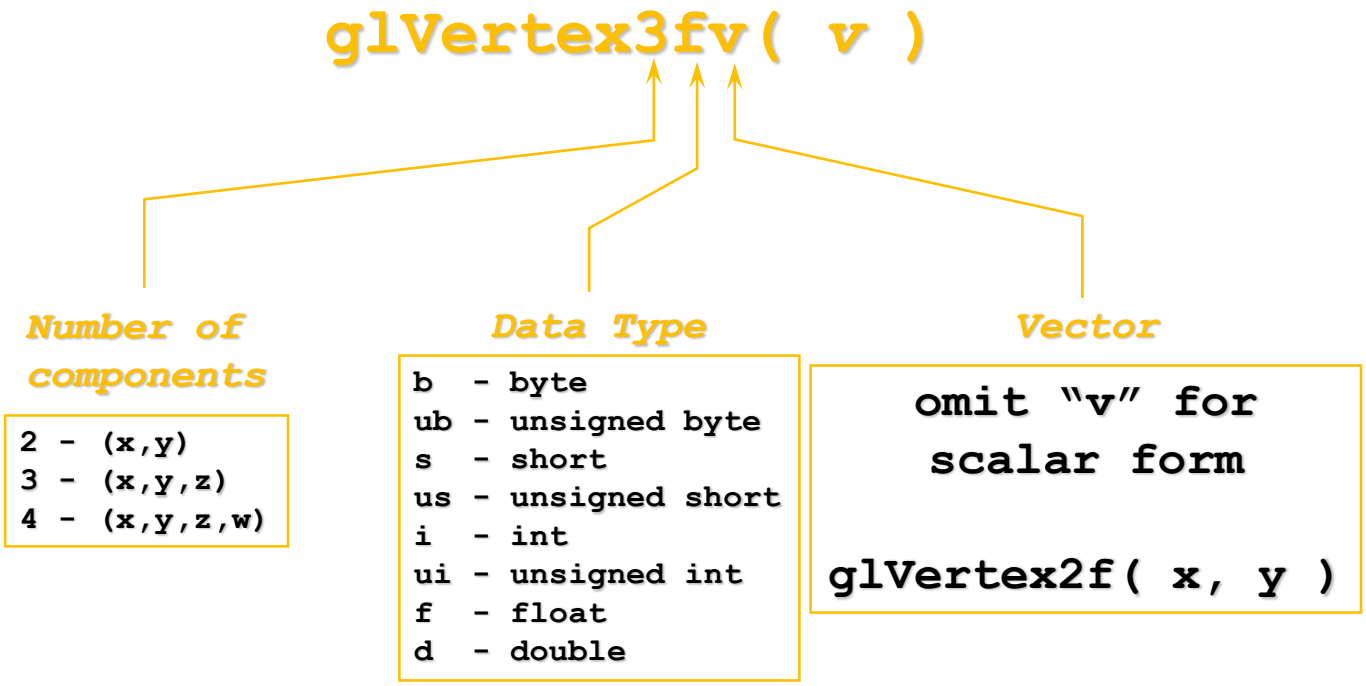
```
def render():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glBegin(GL_TRIANGLES)
    glColor3f(1.0, 0.0, 0.0)
    glVertex2f(0.0, 1.0)
    glVertex2f(-1.0,-1.0)
    glVertex2f(1.0,-1.0)
    glEnd()
```

OpenGL is a State Machine

- If you set a value for a state (or mode), **it remains in effect until you change it.**
  - E.g. the “current” color
  - Others states: the “current” viewing and projection transformations, “current” polygon drawing modes, “current” positions and characteristics of lights, and “current” material properties of the objects, ...
  - Many state variables refer to modes that are enabled or disabled with the command glEnable() or glDisable().

- **OpenGL context** stores all of the state associated with this instance of OpenGL.

OpenGL Functions



[Practice] Using other forms of OpenGL Functions

```
import numpy as np

def render():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glBegin(GL_TRIANGLES)
    glColor3ub(255, 0, 0)
    glVertex2fv((0.0, 1.0))
    glVertex2fv([-1.0,-1.0])
    glVertex2fv(np.array([1.0,-1.0]))
    glEnd()
```

GLFW Input Handling

- *glfw.poll_events()*
  - Processes events that have already been received and then returns immediately.
  - Calls a user-registered callback function for each type of events.

Event type	Set a callback using…
Key input	*glfw.set_key_callback()*
Mouse cursor position	*glfw.set_cursor_pos_callback()* or just poll the position using *glfw.get_cursor_pos()*
Mouse button	*glfw.set_mouse_button_callback()*
Mouse scroll	*glfw.set_scroll_callback()*

```
import glfw
from OpenGL.GL import *

def render():
    pass

def key_callback(window, key, scancode, action, mods):
    if key==glfw.KEY_A:
        if action==glfw.PRESS:
            print('press a')
        elif action==glfw.RELEASE:
            print('release a')
        elif action==glfw.REPEAT:
            print('repeat a')
    elif key==glfw.KEY_SPACE and action==glfw.PRESS:
        print ('press space: (%d, %d)'%glfw.get_cursor_pos(window))

def cursor_callback(window, xpos, ypos):
    print('mouse cursor moving: (%d, %d)'%(xpos, ypos))

def button_callback(window, button, action, mod):
    if button==glfw.MOUSE_BUTTON_LEFT:
        if action==glfw.PRESS:
            print('press left btn: (%d, %d)'%glfw.get_cursor_pos(window))
        elif action==glfw.RELEASE:
            print('release left btn: (%d, %d)'%glfw.get_cursor_pos(window))

def scroll_callback(window, xoffset, yoffset):
    print('mouse wheel scroll: %d, %d'%(xoffset, yoffset))
```

```python
def main():
    # Initialize the library
    if not glfw.init():
        return
    # Create a windowed mode window and its OpenGL context
    window = glfw.create_window(640, 480, "Hello World", None, None)
    if not window:
        glfw.terminate()
        return

    glfw.set_key_callback(window, key_callback)
    glfw.set_cursor_pos_callback(window, cursor_callback)
    glfw.set_mouse_button_callback(window, button_callback)
    glfw.set_scroll_callback(window, scroll_callback)

    # Make the window's context current
    glfw.make_context_current(window)

    # Loop until the user closes the window
    while not glfw.window_should_close(window):
        # Poll for and process events
        glfw.poll_events()
        # Render here, e.g. using pyOpenGL
        render()
        # Swap front and back buffers
        glfw.swap_buffers(window)

    glfw.terminate()
if __name__ == "__main__":
    main()
```

Documentation for glfw

- http://www.glfw.org/documentation.html

- Note there are changes in the python binding:
  - function names use the pythonic **words_with_underscores** notation instead of camelCase
  - **GLFW_ and glfw prefixes have been removed**, as their function is replaced by the module namespace
  - functions like glfwGetMonitors **return a list instead of a pointer and an object count**
  - see https://pypi.python.org/pypi/glfw for more information

Legacy OpenGL & Modern OpenGL

- Legacy OpenGL (OpenGL 1.x)
  - Invented when “fixed-function” hardware was standard
  - No shaders
  - Easier to use & good for rapid prototyping
  - Deprecated since OpenGL 3.0

- Modern OpenGL (OpenGL 2.x~)
  - Now programmable hardware is the common industry practice
  - Use of programmable shaders
  - More difficult to program but far more flexible & powerful

OpenGL as a Learning Tool

- My focus is on fundamental computer graphics concepts, not on concrete implementation.
- So I choose the legacy OpenGL as a basic learning tool, thanks to its simplicity.
- Legacy OpenGL is **just one implementation** of fundamental concepts we'll learn.
- Other implemetations:
  - Graphics libraries: Modern OpenGL, DirectX, Vulkan, Nvidia Optix, ...
  - Game engines: Unreal, Unity, ...
  - Authoring tools: Maya / Blender, ...

Next Time

- 2D Affine Transformations, Frame Buffer
- Homogeneous Coordinates, 3D Affine Transformations

- Assignment 1 (Due date: Sep 11, 23:59)