Pygame and OpenGL

Windows, Points, and Shapes

Based on the graphics pipeline, the first step to render graphics is to develop an application where the graphics will be displayed.

A **window** is a graphical interface element that displays an application's content. Below are the phases of an interactive and graphics-based windowed application.

1. **Startup**: During this stage, objects are created, values are initialized, and any required external files are loaded.
2. **Main Loop**: This stage repeats continuously while the application is running. It consists of the following three (3) substages:
   - **Process Input** – checks if the user has performed any action that sends data to the computer, such as pressing keys on a keyboard or clicking buttons on a mouse.
   - **Update** – changes values of variables and objects.
   - **Render** – creates graphics that are displayed on the screen.
3. **Shutdown**: This stage typically begins when the user performs an action indicating that the program should stop (for example, by clicking a button to quit the application). This stage may involve tasks such as signaling the application to stop checking for user input and closing any windows created by the application.

Pygame is a set of Python modules designed for creating video games. It is a popular Python game development library.

To set up a window using Pygame, use the `pygame.display.set_mode()` method. It requires two (2) parameters to define the width and height of the window. Below is an example:

```
screen = pygame.display.set_mode((500, 500))
```

A **point** is a location in space. It is the most basic graphical element. To render a single point on the screen, use a vertex shader. A vertex shader will consist of the following code using the OpenGL Shading Language (GLSL).

```
void main()
{
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
}
```

Every shader must contain a **`main()`** function. The `void` keyword indicates that no values are returned, and the main function requires no parameters. Vertex shaders manipulate coordinates in a 3D space and are called once per vertex. The purpose of the vertex shader is to set up the `gl_Position` variable, a special, global, and built-in GLSL variable. It is used to store the position of the current vertex.

GLSL has vector data types, which are often used for storing values that indicate positions, colors, and texture coordinates. Vectors may have two, three, or four components, indicated by vec2, vec3, and vec4 (for vectors consisting of floats). The four (4) float values represent the vertex's x, y, z, and w positions. The fourth parameter (w) is used to manipulate the clipping of the vertex position in the 3D space. Its default value is 1.0.

Every vertex shader that you write will use arrays of data stored in **vertex buffers**. Multiple points are needed to specify the shapes of two-dimensional or three-dimensional objects. Below is a sample code to draw a 3D triangle.

```
glBegin(GL_TRIANGLES);
    glVertex3f(10.0, 90.0, 0.0);
    glVertex3f(10.0, 10.0, 0.0);
    glVertex3f(35.0, 75.0, 0.0);
    glVertex3f(30.0, 20.0, 0.0);
    glVertex3f(90.0, 90.0, 0.0);
    glVertex3f(80.0, 40.0, 0.0);
glEnd();
```

Each of the triangle's vertices is specified by a call to the function **`glVertex3f`**. Vertices must be specified between calls to `glBegin` and `glEnd`. The parameter to `glBegin` tells which type of primitive is being drawn.

Uniform Data

In computer graphics, there are many scenarios where you may want to repeatedly use the same information. For example, you may want to translate all the vertices that define a shape by the same amount. You may also want to draw all the pixels corresponding to a shape with the same color. The most flexible and efficient way to accomplish such tasks is by using uniform variables. **Uniform variables** are global variables that can be accessed by the vertex shader or the fragment shader, and whose values are constant while each shape is being drawn (can be changed between draw function calls).

Uniform shader variables, declared with the type qualifier **`uniform`**, provide a mechanism for directly sending data from variables in a CPU application to variables in a GPU program. You must obtain a reference to the location of the uniform variable before data may be sent to it. This is accomplished with the following OpenGL function:

```
glGetUniformLocation(programRef, variableName)
```

The **`glGetUniformLocation`** function returns a value used to reference a uniform variable (indicated by the type qualifier uniform) with the name indicated by the parameter `variableName` and declared in the program referenced by the parameter `programRef`. If the variable is not declared or not used in the specified program, -1 is returned.

Interactivity

The Pygame library provides support for working with user input events. Pygame detects keydown events and keyup events. Keydown events occur the moment a key is initially pressed, while **keyup events** occur the moment the key is released.

An event is said to be **discrete** if it happens once at an isolated point in time. It is said to be **continuous** if it continues happening during an interval of time. The keydown and keyup events are discrete, but many applications also feature continuous actions (such as moving an object on the screen) that occur for as long as a key is being held down.

The following is a sample code that checks if a keydown event occurs.

```
if event.type == pygame.KEYDOWN:
    if event.key == pygame.K_w:
        print("Player moved up!")
    elif event.key == pygame.K_a:
        print("Player moved left!")
    elif event.key == pygame.K_s:
        print("Player moved down!")
    elif event.key == pygame.K_d:
        print("Player moved right!")
```

References:

Korites, B. (2018). *Python graphics: A reference for creating 2D and 3D images.* Apress.
Marschner, S. & Shirley, P. (2021). *Fundamentals of computer graphics* (5th ed.). CRC Press.
Stemkoski, L. & Pascale, M. (2021). *Developing graphics frameworks with Python and OpenGL.* CRC Press.