Implements

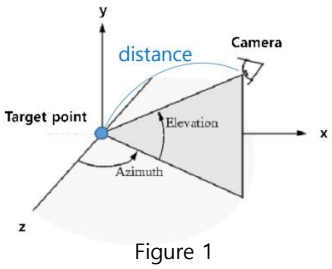


Figure 1

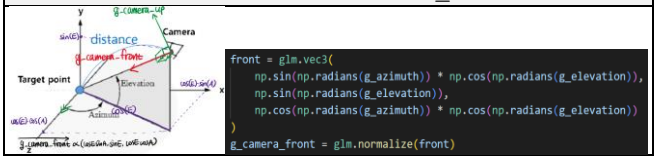
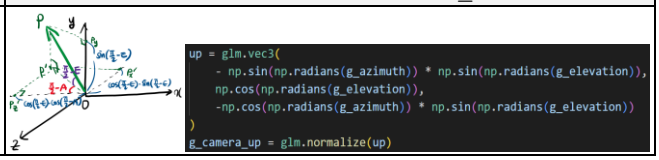
The description of the implementation is based on '2. Requirements - A' in the task specification document.

ii. When the program is first executed, the camera looks at the target point (0,0,0) from the position (0, 0, -1).

* Explanation of the global variables used:

1. g_azimuth, g_elevation: Angles shown in Figure 1.
2. g_projection_is_ortho: A flag that distinguishes whether it is an orthographic projection or not (since it starts with a perspective projection, the initial value is False).
3. g_screen_width, g_screen_height: Screen size.
4. last_mouse_x_pos, last_mouse_y_pos: Stores the most recent clicked point on the screen.
5. mouse_pressed: A flag that stores whether the left mouse or the right mouse is clicked.
6. g_P: Projection matrix.
7. g_camera_pos, g_camera_front, g_camera_up: Camera position, vector indicating the direction in which the camera is facing(-w vector in camera frame), and the vector indicating the up direction of the camera(v vector in camera frame).

iii-1. **Orbit**: As shown in Figure 1, the angle rotated in the direction from the +z-axis to the +x-axis is called azimuth, and the angle rotated in the direction from the +y-axis to the xz plane's first quadrant is called elevation. In the `mouse_button_callback` function, it checks whether the left mouse is pressed and sets the `mouse_pressed['left']` flag accordingly. When the left mouse is pressed and the cursor is moved, the logic related to rotation is processed inside the if statement where the flag is true in the `cursor_position_callback`. Assuming that the window we see is the xy plane, we add the x_offset moved by the cursor in the x-direction to `g_azimuth` and the difference moved in the y-direction, which is y_offset, to `g_elevation`.

Rotation vector for `camera_front`	Rotation vector for `camera_up`
	

iii-2. **Pan**: Until the right mouse button is clicked and released, the mouse_button_callback sets the flag for the right mouse to True, which allows it to enter the right if statement in the cursor_callback function. At this point, we multiply y_offset by the camera_up vector direction and add it, and we multiply x_offset by the direction vector of the cross product between camera_up and camera_front and add it, allowing us to move left and right in the u direction and up and down in the v direction with respect to the camera frame.

```
elif mouse_pressed.get('right'):
    # panning
    moving_speed = 0.05
    g_camera_pos += g_camera_up * y_offset * moving_speed + glm.normalize(glm.cross(g_camera_up, g_camera_front)) * x_offset * moving_speed
```

iii-3. **Zoom**: When a scroll event occurs, the scroll_callback function is called. By multiplying y_scroll by the vector in the direction of camera_front and adding it to the existing camera_pos, we can move in the direction of camera_front vector(camera_front vector is along w axis of the camera frame mentioned in assignment document).

```
g_camera_pos += g_camera_front * move_speed * y_scroll
```

However, in perspective projection, depth is represented, so zooming in and out can be visually observed when scrolling. In contrast, in orthogonal projection, depth is not represented, so zooming cannot be observed.

iv. **Toggle projection**: By pressing v key, the key_callback is called and it goes into the logic in if statement according to the flag that shows the state of projection(True when orthogonal projection). By using flag, we can toggle projection. After changing the flag, the corresponding projection is calculated. In orthogonal projection, the ortho_height is fixed and the ortho_width is calculated according to the screen aspect ratio. Then, the values corresponding to the 6

arguments (left, right, bottom, top, near, far) of the ortho() function are filled in. In perspective projection, near and far are fixed, as well as fov, and aspect_ratio is calculated based on the screen size and then filled in. Additionally, a framebuffer_size_callback is added to adjust the window size when the screen size changes(This is optional feature).

```
if key == GLFW_KEY_V and action == GLFW_PRESS:
    g_projection_is_ortho = not g_projection_is_ortho

    if g_projection_is_ortho:
        ortho_height = 1.
        ortho_width = ortho_height * g_screen_width/g_screen_height
        g_P = glm.ortho(-ortho_width*.5,ortho_width*.5, -ortho_height*.5,ortho_height*.5, -10,10)
    else:
        near = 0.5
        far = 20.0
        aspect_ratio = g_screen_width/g_screen_height
        g_P = glm.perspective(glm.radians(45.0), aspect_ratio, near, far)
```

Figure 2 toggle projection matrix

```
def framebuffer_size_callback(window, width, height):
    global g_P, g_projection_is_ortho, g_screen_width, g_screen_height

    glViewport(0, 0, width, height)

    g_screen_width, g_screen_height = width, height

    if g_projection_is_ortho:
        ortho_height = 10.
        ortho_width = ortho_height * width/height
        g_P = glm.ortho(-ortho_width*.5,ortho_width*.5, -ortho_height*.5,ortho_height*.5, -10,10)
    else:
        near = 0.5
        far = 20.0
        aspect_ratio = width/height
        g_P = glm.perspective(glm.radians(45.0), aspect_ratio, near, far)
```

Figure 3 change size of screen keeping ratio

v. **grid on xz plane**: In the prepare_vao_grid() function, the vertex position and color information required for the vao is stored and then returned as the vao. In draw_grid(), the information from the vao is drawn using glDrawArrays(GL_LINES, 0, 84).

```
def prepare_vao_grid():
    # prepare vertex data (in main memory)

    vertices = glm.array(glm.float32,
    -1.0, 0, -1, 1, 1, 1, -1.0, 0, 1, 1, 1, 1, -0.9, 0, -1,
    )

    # create and activate VAO (vertex array object)
    VAO = glGenVertexArrays(1)  # create a vertex array obje
    glBindVertexArray(VAO)      # activate VAO

    # create and activate VBO (vertex buffer object)
    VBO = glGenBuffers(1)  # create a buffer object ID and
    glBindBuffer(GL_ARRAY_BUFFER, VBO)  # activate VBO as a

    # copy vertex data to VBO
    glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices.

    # configure vertex positions
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * glm.
    glEnableVertexAttribArray(0)

    # configure vertex colors
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * glm.
    glEnableVertexAttribArray(1)

    return VAO
```

Figure 4 grid VAO

```
def draw_grid(vao, MVP, MVP_loc):
    glBindVertexArray(vao)
    glUniformMatrix4fv(MVP_loc, 1, GL_FALSE, glm.value_ptr(MVP))
    glDrawArrays(GL_LINES, 0, 84)
```

Figure 5 draw grid function(called in main while loop)

Optional Implement. **World frame**: x, y, z axis for red, green, blue line to help where is the direction of camera is watching.

Screenshots