INTRODUCTION TO COMPUTER GRAPHICS

FEB. 13, 2019 NOTES

HOMEWORK ASSIGNMENT 1

- The first homework assignment is due on 2/24/19 Sunday at **MIDNIGHT**.
- The homework will be submitted on Sakai.

The homework will consist of two parts-

1) Software portion
   a. **Description -** Write an OpenGL program to display a rotating pinwheel.
   b. **Minimum Requirement –**
      i. Create a rotating 2D model of a pinwheel
      ii. Have simple colors for each wing
   c. **Extra Credit –**
      i. Create a rotating 3D model of a pinwheel using a texture map.
      ii. You are not required to create the texture map for the pinwheel, as you are allowed to download one from online (Although creating your own texture map would be considered additional extra credit and should be specified in your submission).
      iii. Instead of creating a pinwheel, you may create a 3D model of a Ferris wheel. The seats on the Ferris wheel can remain in a FIXED position during the rotation animation.
   d. **Resources –** Easiest modeling software to use for this assignment is **Blender**.

2) Hardware portion
   a. **Description –** Implement a traffic light circuit using the Arduino Board.
   b. **Minimum Requirement –**
      i. Emulate two lanes that are perpendicular to each other.
      ii. Program should enforce that only ONE lane could be green at a time while the other is red, effectively simulating a real-life traffic light system.

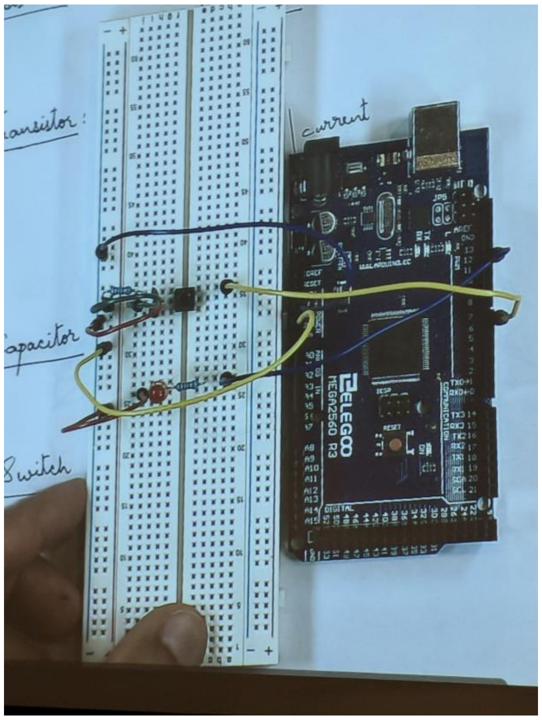
iii. i.e. If lane 1 is currently green, if a car stops and waits at lane 2, then the lights in lane 1 will change from green to yellow, and then from yellow to red. Finally, the lights in lane 2 will change from red to green. The same logic will work vice versa.

iv. Two button switches will be used to simulate a car waiting at each respective light.

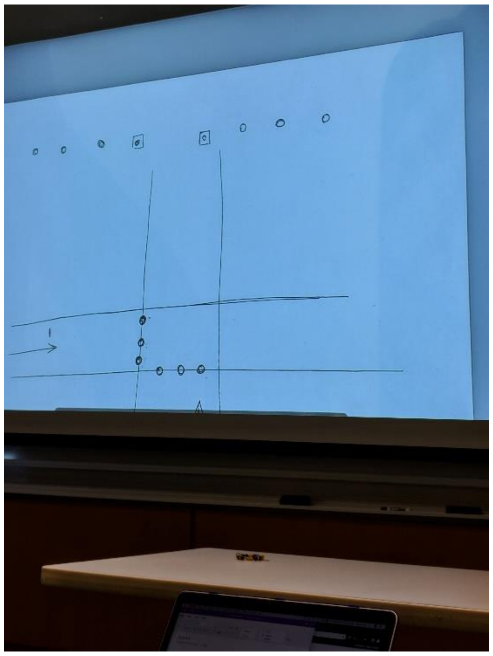
v. If lane 1 is green, then only the button from lane 2 will work if pressed. (The button switch in lane 1 should NOT work).

vi. In the submission file, provide a reporting containing as much detail as possible in regards to the circuit. Also, you must provide a video demo with a step- by-step walkthrough of the finished circuit and an explanation of all the components.

c. **Example of the button circuit-**



d. **Example of the circuit layout for the two traffic lights-**



The submission file will consist of a single .zip file containing-

1) A PDF file containing a written report explaining, in great detail, both parts of the assignment.
2) A video file containing a demo and explanation of the circuit from the hardware portion of the assignment.

5.1 TRANSFORMATIONS

- Transformations allow for us to rotate or change the positions of objects rendered in OpenGL. (This information is important for the software portion of the homework assignment).
- There are 2D and 3D transformations
- You can also implement camera movement
- The c classes that will be using to accomplish this will remain the same as the ones used in previous lectures.
- For example, the shader.h class will be used, alongside the vector and fragment shaders.
- A third-party library called **glm** will also be used to perform matrix operations on the GPU.

Key Changes to the Vertex Shader

- A transform is denoted by a 4x4 matrix, and is represented by the following code:

```
uniform mat4 transform;
```

Key Changes to the Fragment Shader

- None

In the main.cpp file

- In order to generate a 4x4 matrix, use the following code:

```
// generate transformation matrix
glm::mat4 trans(1.0f);
```

- The **(1.0f)** portion initializes the matrix to the identity transform.
- The variable **trans** and its parameters holds an offset of the object's current position:

```
trans=glm::translate(trans,glm::vec3(0.5f,-0.5f,0.0f));
trans=glm::rotate(trans,glm::radians((GLfloat)glfwGetTime()*50.0f),glm::vec3(0.0f,0.0f,1.0f))
```

- In the second trans variable has a function call to **vec3(x, y, z)** , which specifies the axis at which the object is rotating
- An important note is that all these transformations are appended to the initial transform of the object.
- More clearly, we are using the initial value of the transform, adding it to a new transform, and we are then storing the result in a transform variable.
- Once we initialized a 4x4 matrix, we set the transform matrix to a specific location:

```
GLuint transform_location=glGetUniformLocation(our_shader.program,"transform");
glUniformMatrix4fv(transform_location,1,GL_FALSE,glm::value_ptr(trans));
```

- With this one change, we then specify in the **vertex shader** that the position changed:

```
gl_Position=transform*vec4(position,1.0f);
```

Errors to be expected when writing transformations:

1) In the old version of glm, you didn't need to specify a **1.0f** when generating a transformation matrix. But the new version requires this value- **glm::mat4 trans(1.0f);**
    a. If **1.0f** is not included, then an empty window is displayed.

Homework

- For the homework, 4 separate objects (the wings) are rotating.
- In this example however, we are rotating two triangles that are smushed together.

5.2 COORDINATES

- When generating 3d rotating cubes, a matrix is needed to hold the positions of all the cubes:

```
glm::vec3 cube_positions[]={
    glm::vec3( 0.0f, 0.0f, 0.0f),
    glm::vec3( 2.0f, 5.0f,-15.0f),
    glm::vec3(-1.5f,-2.2f,-2.5f),
    glm::vec3(-3.8f,-2.0f,-12.3f),
    glm::vec3( 2.4f,-0.4f,-3.5f),
    glm::vec3(-1.7f, 3.0f,-7.5f),
    glm::vec3( 1.3f,-2.0f,-2.5f),
    glm::vec3( 1.5f, 2.0f,-2.5f),
    glm::vec3( 1.5f, 0.2f,-1.5f),
    glm::vec3(-1.3f, 1.0f,-1.5f)
};
```

- The following loop describes how the 3d cubes are transformed in respect to the world space:

```
for(GLuint i=0;i<10;++i)
{
    // world space transform
    glm::mat4 model(1.0f);
    model=glm::translate(model,cube_positions[i]);
    model=glm::rotate(model,glm::radians((GLfloat)glfwGetTime()*50.0f),glm::vec3(0.5f,1.0f,0.0f));
    GLfloat angle=glm::radians(20.0f*i);
    model=glm::rotate(model,angle,glm::vec3(1.0f,0.3f,0.5f));
    glUniformMatrix4fv(model_location,1,GL_FALSE,glm::value_ptr(model));

    glDrawArrays(GL_TRIANGLES,0,36);
}
```

- The following three matrices are used to handle 3d model transformations-

```
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
```

- Inside the gl loop, we are getting the locations of all the uniform primitives in the vertex shader and are assigning them based on the matrices defined above.

5.3 CAMERA CONTROL

- The concept of camera movement involves not changing the scene itself, but instead adding more functionality with the use of **callbacks.**
- The following bool variable keeps track of which key is pressed:

```
bool keys[1024];
```

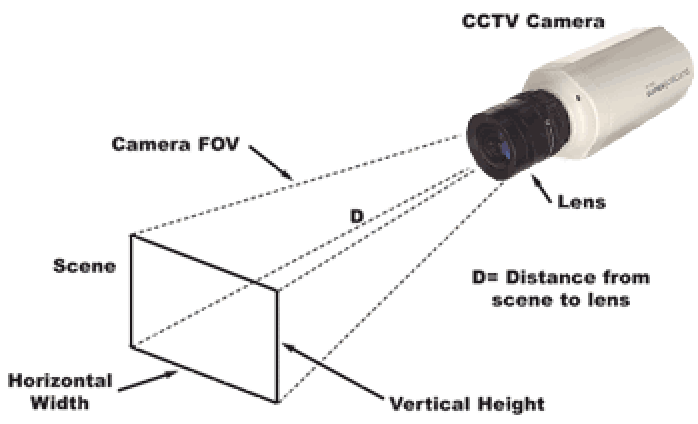
- The following variables keep track of the current and last positions of the cursor:

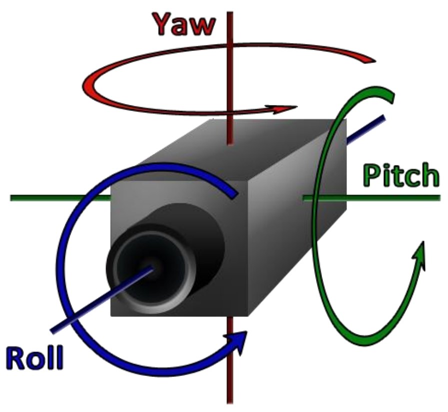
```
GLfloat delta_time=0.0f;
GLfloat last_frame=0.0f;
GLfloat last_X=0.0f,last_Y=0.0f;
```

- The **FOV** and **Yaw** values of the camera are defined with the following statements:

```
GLfloat fov=45.0f;
GLfloat yaw=-90.0f;
```

- To get a better understanding of what FOV and YAW are, here are a couple of visuals:





- The following callback method handles camera movement:

```
void do_movement()
{
    // camera controls
    GLfloat camera_speed=5.0f*delta_time;
    if(keys[GLFW_KEY_W]) camera_position+=camera_speed*camera_front;
    if(keys[GLFW_KEY_S]) camera_position-=camera_speed*camera_front;
    if(keys[GLFW_KEY_A]) camera_position-=glm::normalize(glm::cross(camera_front,camera_up))*camera_speed;
    if(keys[GLFW_KEY_D]) camera_position+=glm::normalize(glm::cross(camera_front,camera_up))*camera_speed;
}
```

- The following variable defines the camera speed:

```
GLfloat camera_speed=5.0f*delta_time;
```

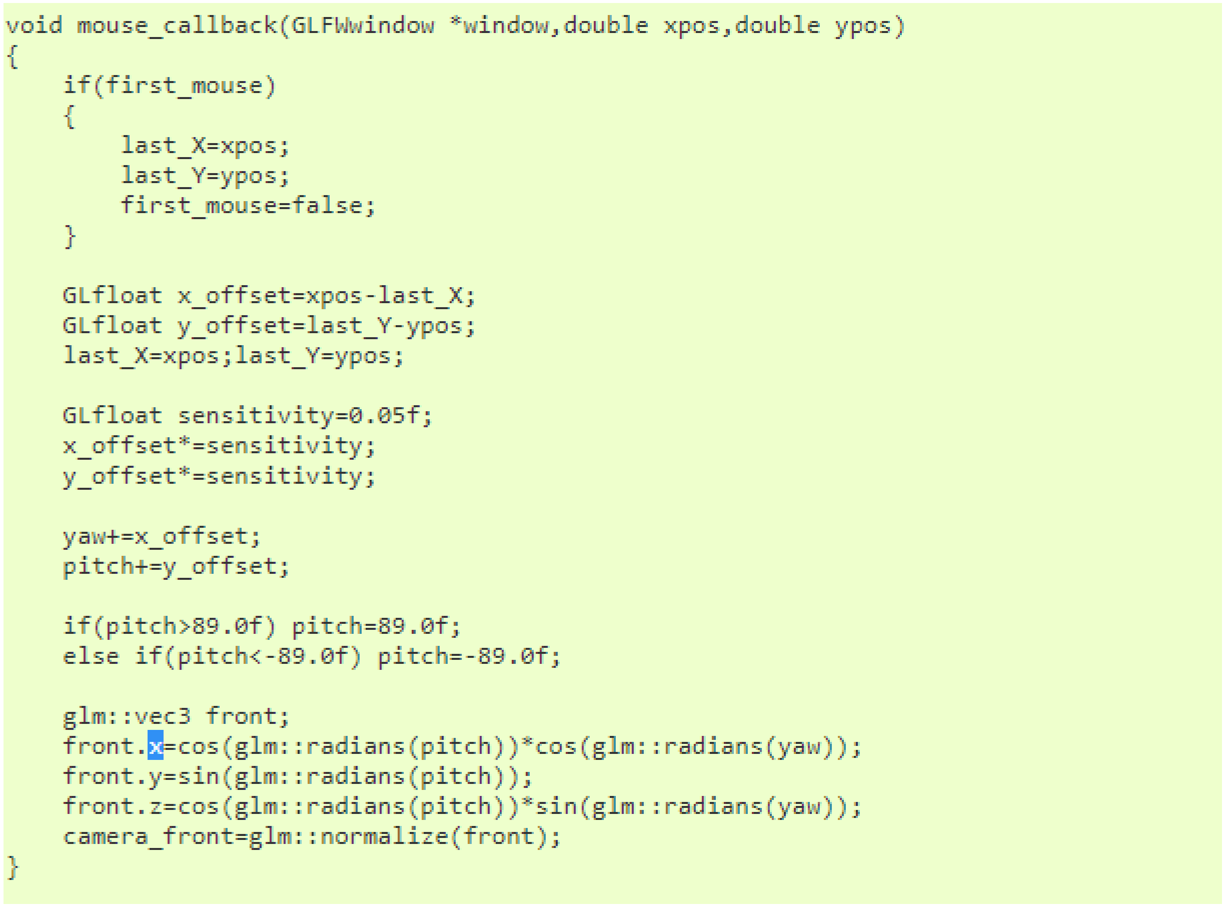
- There are 4 keys that control the 4 directions in which a camera can move:
    1. **W Key –** camera goes in positive y direction
    2. **S Key –** camera goes in negative y direction
    3. **A Key –** camera goes in negative x direction
    4. **D Key –** camera goes into positive x direction
- The following callback uses the mouse cursor to change the angle of the camera:

```
void scroll_callback(GLFWwindow *window,double xoffset,double yoffset)
{
    if(fov>=1.0f && fov<=45.0f)  fov-=yoffset;
    if(fov<=1.0f) fov=1.0f;
    if(fov>=45.0f) fov=45.0f;
}
```

- The **Key_Callback()** functions the same as **do_movement()** except the key difference is that key_callback() keeps track of an action:

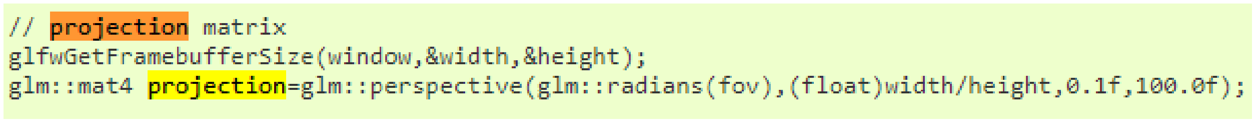
```
void key_callback(GLFWwindow *window,int key,int scancode,int action,int mode)
{
    if(key==GLFW_KEY_ESCAPE && action==GLFW_PRESS)
        glfwSetWindowShouldClose(window,GL_TRUE);

    if(key==GLFW_KEY_UP && action==GLFW_PRESS)
    {
        mix_value+=0.1f;
        if(mix_value>1.0f) mix_value=1.0f;
    }

    if(key==GLFW_KEY_DOWN && action==GLFW_PRESS)
    {
        mix_value-=0.1f;
        if(mix_value<0.0f) mix_value=0.0f;
    }

    if(action==GLFW_PRESS) keys[key]=true;
    else if(action==GLFW_RELEASE) keys[key]=false;
}
```

- The **Mouse_Callback()** keeps track of where the cursor is and the corresponding orientation of the FOV and how you are moving it.



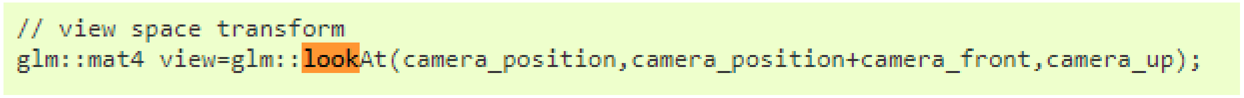
```
void mouse_callback(GLFWwindow *window,double xpos,double ypos)
{
    if(first_mouse)
    {
        last_X=xpos;
        last_Y=ypos;
        first_mouse=false;
    }

    GLfloat x_offset=xpos-last_X;
    GLfloat y_offset=last_Y-ypos;
    last_X=xpos;last_Y=ypos;

    GLfloat sensitivity=0.05f;
    x_offset*=sensitivity;
    y_offset*=sensitivity;

    yaw+=x_offset;
    pitch+=y_offset;

    if(pitch>89.0f) pitch=89.0f;
    else if(pitch<-89.0f) pitch=-89.0f;

    glm::vec3 front;
    front.x=cos(glm::radians(pitch))*cos(glm::radians(yaw));
    front.y=sin(glm::radians(pitch));
    front.z=cos(glm::radians(pitch))*sin(glm::radians(yaw));
    camera_front=glm::normalize(front);
}
```

- The **Projection Matrix** is directly taken from the field of view, width, and height of the current window:



```
// projection matrix
glfwGetFramebufferSize(window,&width,&height);
glm::mat4 projection=glm::perspective(glm::radians(fov),(float)width/height,0.1f,100.0f);
```

- The **View Matrix** is not initialized as the identity, instead it is initialized with a function called **lookAt()**, which refers to the camera position and the direction in which the camera faces.



```
// view space transform
glm::mat4 view=glm::lookAt(camera_position,camera_position+camera_front,camera_up);
```

- Once again, the biggest takeaway with camera movement is the use of callbacks, which enables key controls and mouse controls.
- However, keyboard control is not the preferred practice in real-life application, as mouse control with an orbit camera is better in all-around usability.