# Assignment 5: Camera in Practice

**CS Spring 2019**

*Due Date: Mar. 26th*

Follow the instructions carefully. If you encounter any problems in the setup, please do not hesitate to reach out to TA.

## Requirements

In this lab, we will learn how to define a pinhole model camera in OpenGL. And make yourselves a moving view point camera for the exercise.

## Transformation in Camera

Let's recall what we have learnt in last lecture. There are basically three coordinates in the scene: model, view, and projection. They are handy tools to separate transformations cleanly. You may not use this in real 3D development (there are other more generic defined methods if you understand transformation better), but you should. This is the way everybody does, because it's easier to imagine in this way.

## The Model matrix

All models are defined by a set of vertices. The X,Y,Z coordinates of these vertices are defined relative to the object's centre : that is, if a vertex is at (0,0,0), it is at the centre of the object. We'd like to be able to move this model, maybe because the player controls it with the keyboard and the mouse. In Lab 3, we learn to do with :

```
TransformedVector = TranslationMatrix * RotationMatrix * ScaleMatrix * OriginalVector;
```

You apply this matrix to all your vertices at each frame (in GLSL, not in C++) and everything moves. Something that doesn't move will be at the *centre of the world*.

After this transformation, all the vertices in the model are now in *World Space*. We went from Model Space (all vertices defined relatively to the centre of the model) to World Space (all vertices defined relatively to the centre of the world).

## The View matrix

Initially your camera is at the origin of the World Space. In order to move the world, you simply introduce another matrix. Let's say you want to move your camera of 3 units to the right (+X). This is equivalent to moving your whole world (meshes included) 3 units to the LEFT (-X). we can do this with:

```
// Use #include <glm/gtc/matrix_transform.hpp> and #include <glm/gtx/transform.hpp>
glm::mat4 ViewMatrix = glm::translate(glm::mat4(), glm::vec3(-3.0f, 0.0f ,0.0f));
```

Now we went from World Space (all vertices defined relatively to the centre of the world, as we made so in the previous section) to Camera Space (all vertices defined relatively to the camera).

Thanks to *glm*, we can take the advantage of *glm::lookAt* function.

```
glm::mat4 CameraMatrix = glm::lookAt(
    cameraPosition,
    cameraTarget,
    upVector
);
```

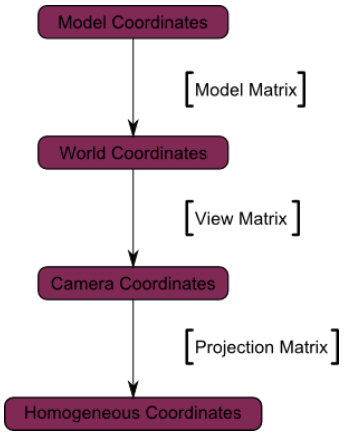
**The Projection matrix**

We're now in Camera Space. This means that after all these transformations, a vertex that happens to have x==0 and y==0 should be rendered at the centre of the screen. But we can't use only the x and y coordinates to determine where an object should be put on the screen: its distance to the camera (z) counts, too. For two vertices with similar x and y coordinates, the vertex with the biggest z coordinate will be more on the centre of the screen than the other. A 4x4 matrix can represent this projection.

```
glm::mat4 projectionMatrix = glm::perspective(
    glm::radians(FoV),
    4.0f / 3.0f,       // Aspect ratio
    0.1f,              // Near clipping plane
    100.0f             // Far clipping plane
);
```

We went from *Camera Space* (all vertices defined relatively to the camera) to *Homogeneous Space* (all vertices defined in a small cube. Everything inside the cube is onscreen).

And the overall transformation diagram should be.



**The interface**

We now learn how to use the mouse and the keyboard to move the camera. If you have defined the camera quite well, we don't need to change much for this part of the code. The major modification is that instead of computing the overall transformation matrix once, we have to do it every frames. So first, we move the code in to a loop

```
do{
    // …..
}
```

We will first recompute position, horizontalAngle, verticalAngle and FoV according to the inputs, and then compute the View and Projection matrices from position, horizontalAngle, verticalAngle and FoV.

**Orientation**

Read and set the mouse position with

```
int xpos, ypos;
glfwGetMousePos(&xpos, &ypos);
```

And computer the updated viewing angles:

```
horizontalAngle += mouseSpeed * deltaTime * float(1024/2 - xpos );
verticalAngle   += mouseSpeed * deltaTime * float( 768/2 - ypos );
```

We can now compute a vector that represents, in World Space, the direction in which we're looking

```
    glm::vec3 direction(
        cos(verticalAngle) * sin(horizontalAngle),
        sin(verticalAngle),
        cos(verticalAngle) * cos(horizontalAngle)
    );
```

Now we want to compute the "up" vector reliably. Notice that "up" isn't always towards +Y : if you look down, for instance, the "up" vector will be in fact horizontal.

In our case, the only constant is that the vector goes to the right of the camera is always horizontal. You can check this by putting your arm horizontal, and looking up, down, in any direction. So let's define the "right" vector : its Y coordinate is 0 since it's horizontal, and its X and Z coordinates are just like in the figure above, but with the angles rotated by 90°.

```
    glm::vec3 right = glm::vec3(
        sin(horizontalAngle - 3.14f/2.0f),
        0,
        cos(horizontalAngle - 3.14f/2.0f)
    );
```

We have a “right” vector and a “direction”, or “front” vector. The “up” vector is a vector that is perpendicular to these two. A useful mathematical tool makes this very easy : the cross product.

```
    glm::vec3 up = glm::cross( right, direction );
```

**Position**

The code is pretty straightforward.

```
    // Move forward
    if (glfwGetKey( GLFW_KEY_UP ) == GLFW_PRESS){
        position += direction * deltaTime * speed;
    }
    // Move backward
    if (glfwGetKey( GLFW_KEY_DOWN ) == GLFW_PRESS){
        position -= direction * deltaTime * speed;
    }
    // Strafe right
    if (glfwGetKey( GLFW_KEY_RIGHT ) == GLFW_PRESS){
        position += right * deltaTime * speed;
    }
    // Strafe left
    if (glfwGetKey( GLFW_KEY_LEFT ) == GLFW_PRESS){
        position -= right * deltaTime * speed;
}
```

The only special thing here is the deltaTime. You don’t want to move from 1 unit each frame for a simple reason :

- If you have a fast computer, and you run at 60 fps, you’d move of 60*speed units in 1 second
- If you have a slow computer, and you run at 20 fps, you’d move of 20*speed units in 1 second

Since having a better computer is not an excuse for going faster, you have to scale the distance by the “time since the last frame”, or “deltaTime”.

```
double currentTime = glfwGetTime();

float deltaTime = float(currentTime - lastTime);
```

**Field of view**

At last, we can also bind the wheel of the mouse to the FOV.

```
float FoV = initialFoV - 5 * glfwGetMouseWheel();
```

Computing the matrices is now straightforward. We use the exact same functions than before, but with new parameters.

**Exercises**

Upload any model into the scene, and define a free view point camera that can move with mouse and keyboard input.