CS3388B: Lecture 14

March 21, 2023

14 Manipulating the Camera: Part 2

Manipulating the camera continues to be a fundamental problem in graphics applications. In particular, how to manipulate the camera as if it were an object in the world. This is crucial to nearly every graphical application.

Recall from Lecture 9 that we saw how to manipulate the camera using a "look at" function. This allowed the camera to be placed somewhere in world coordinates and then look in the direction of a particular target. This is a very fundamental operation, and we can use it as part of the solution to manipulate a camera in a variety of other dynamic ways.

14.1 First Person

You **are** the camera.

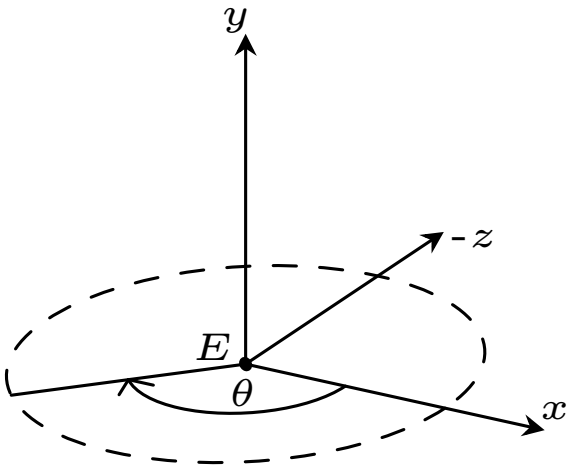
A **first-person camera** is commonly employed in video games, applications which require some immersive experience, and applications without a character or **avatar** in the rendered scene to represent the human user. The goal of first person is to control the camera and move around as if your eyes were the camera, a *first person perspective*.

First person cameras are the most simple to implement (but usually very hard to get right). Pick a height for the camera relative to the ground/floor of your scene. Place the camera there. Then, have it look "forward" parallel to the floor.

Unlike humans which have necks that can rotate independent of the body, first person cameras (outside of VR applications) usually keep movement *stiff*. That is, any camera movement/rotation also affects the movement/rotation of the player's body/avatar and vice versa. To implement this is not so hard. We can use the basic `look_at` function to help.

Start with a world position we want for the camera, say $E = (x_e, y_e, z_e)$ (E for eye). Since that target position to look at should be parallel to the floor, we should have the Target T have the same y coordinate at E. Moreover, since this is first person, we expect that the camera to be level. That is, it's "up" vector should point directly upwards in world space.

Now, we have one last degree of freedom: rotation. Since we have a fixed height y_e from the ground, and we want "up" to be directly upwards, the last parameter to the rotation about E. Viewing E as the origin of the camera's coordinate system, we have only θ left to define; its rotation around the y axis.



Relative to E, we can come up with a look direction using cos, sin, and θ. Relative to E, the look direction is $D = (\cos(\theta), 0, \sin(\theta))$ So, our view matrix V is fully determined by E and θ.

We could come up with a matrix directly which describes this **camera pose** (position and orientation). Or, we could use the `look_at` function. In this case we want the camera to be positioned at E and look *at* $E + D$.

14.1.1 Moving in First Person

Our camera pose is defined by two parameters: E and θ. Classic **tank controls** only move the camera forward or backward based on its current rotation. It always moves parallel to its look direction using up/down arrow keys. Then, the direction is changed with left/right arrow keys. Therefore, left/right should change θ meanwhile up/down should move E by a proportion of the current direction.

```
double currentTime = glfwGetTime();
static double lastTime = glfwGetTime();
float deltaTime = (currentTime - lastTime);
// Rotate counterclockwise
if (glfwGetKey( window, GLFW_KEY_LEFT ) == GLFW_PRESS) {
    theta -= deltaTime * speed;
}
// rotate clockwise
if (glfwGetKey( window, GLFW_KEY_RIGHT ) == GLFW_PRESS) {
    theta += deltaTime * speed;
}
glm::vector3 dir(cos(theta), 0, sin(theta));

if (glfwGetKey( window, GLFW_KEY_UP ) == GLFW_PRESS) {
    eye += dir * (deltaTime * speed);
}
// Move backward
if (glfwGetKey( window, GLFW_KEY_DOWN ) == GLFW_PRESS) {
    eye -= dir * (deltaTime * speed);
}
V = glm::lookAt(position, eye, {0, 1, 0});
lastTime = currentTime;
```

Of course, there are many ways to move around in a first-person perspective. Where multiple **analog inputs** are available (video game controllers) it is typical for the left analog stick to motion and the right analog stick to control rotation.

Yet another option is to use WASD (or up/down/left/right) to control motion and "click and drag" mouse interaction to change the rotation.

14.2 Third Person

In third person, the camera is situated behind the player's avatar and shows the avatar themselves and a larger field of view of the environment. This is the most common viewpoint for video games, particularly action/adventure games.

There are three main types of third-person perspective cameras.

- **Fixed**: the camera is always a constant position and orientation relative to the in-game avatar. Any movement applied to the avatar also applies to the camera.
- **Tracking**: the camera follows the avatar, but is not fixed relative to the avatar. The player has no control of camera at all, in fact. This is common in paradigms with relatively linear progression. Think *Mario Kart, Crash Bandicoot, Subway Surfers,* etc.
- **Interactive**: the player controls both the in-game avatar and the camera independently, but the program enforces particular constraints on the camera's pose like distance from player and orientation. This is the standard and modern implementation of third person cameras in most video games.

Tracking Third-Person Camera

Let's consider the easiest case of a fixed third-person camera. let the pose of the player's avatar have position $P = (x, y, z)$ and orientation in the x-z plane be θ (this is just as in the first-person case, since most 3D games are actually limited to "flat" movement). Thus, the player is looking in the direction $D = (\cos(\theta), 0, \sin(\theta))$ relative to P. Now, the camera's pose is defined as a fixed offset of the player's pose.

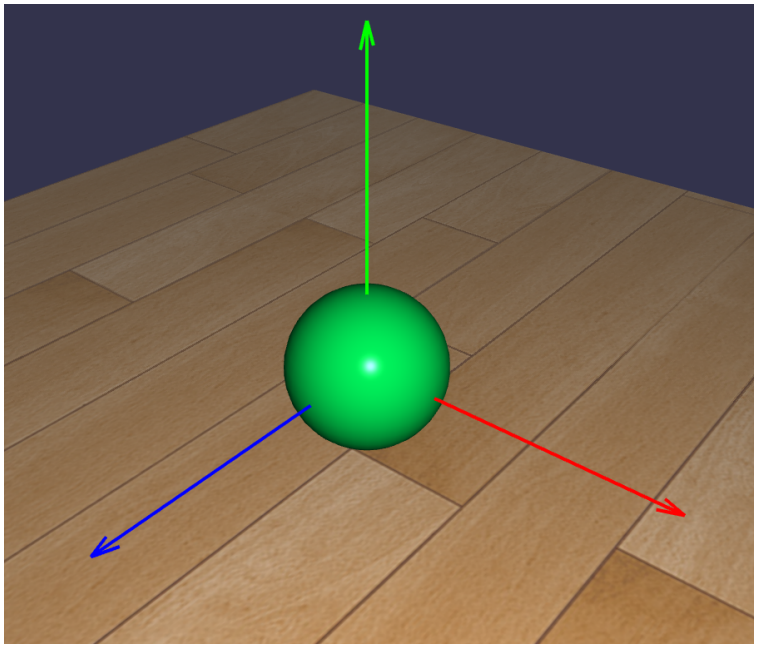
It is often the case that the camera's offset is only offset in terms of position, while the orientation is left to be exactly the same as the avatar. In the other main case, the camera's orientation is a fixed rotation relative to the avatar's orientation. Typically, this fixed rotation is such that the camera is looking directly at at the avatar from it's fixed offset. position.

Let the camera's offset in position be $O = (x_o, y_o, z_o)$ with no offset in orientation. We define the view matrix now using the `look_at` function. The camera is at world position $P_p + O$ and is looking at $P_p + D$, the same direction as the player.

14.3 Movement in Third-Person

In contrast to first person perspective, movement in third person must move both the camera and the avatar. Albeit with the same movement. The main difference this time is that the input from the player moves the avatar rather than the camera. Then, the program computes the motion of the camera in order to maintain the predetermined offset.

The main difficulty to remember is that the motion applied to the avatar (using the Model matrix) cannot be defined using the `look_at` function. (Recall that `look_at` actually applied an inverse transformation to make `V` transform from world coordinates to camera coordinates)). Thus, we have to apply transformations manually to build up the model matrix. Not so bad, really, once we understand the **local coordinates** of the model.



x is red, y is green, z is blue.

As we have always discussed, local coordinates and world coordinates are right-handed, with positive x being right and positive y being up. This makes z point "out of the screen", since the camera looks down $-z$ by default.

It is convention that the positive z direction in local coordinates is considered "forward" for an object/avatar. Therefore, in a third-person perspective, we want forward motion to be applied in the direction of the avatar's positive z axis.

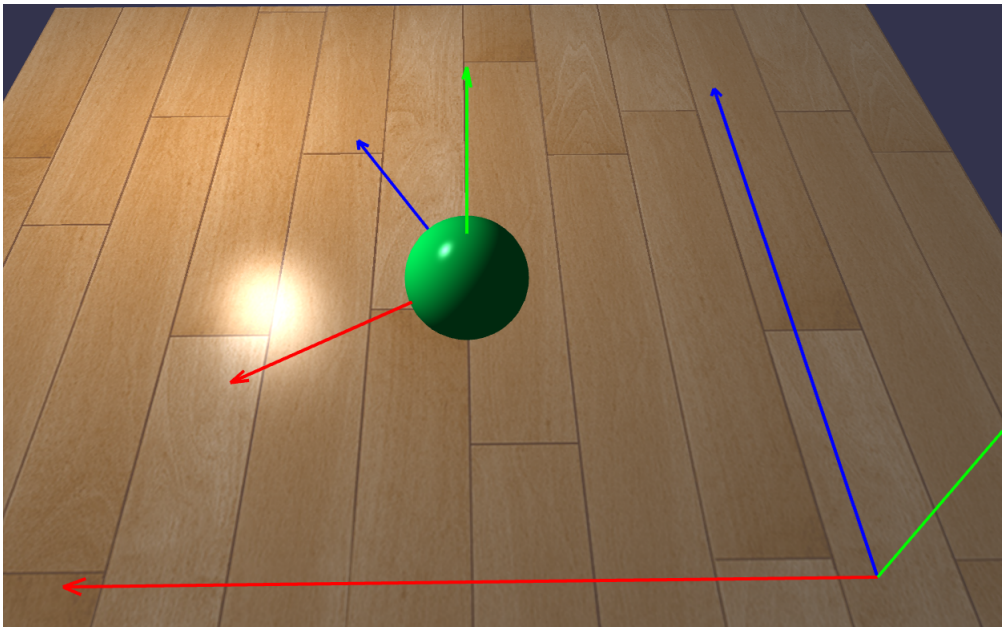
However, it is not so easy to do something like:

```
glm::vec3 pos(x,y,z);

// Move forward
if (glfwGetKey( window, GLFW_KEY_UP ) == GLFW_PRESS) {
    pos.z += deltaTime * speed;
}
// Move backward
if (glfwGetKey( window, GLFW_KEY_DOWN ) == GLFW_PRESS) {
    pos.z -= deltaTime * speed;
}
```

But, why?

In a third person perspective, the player is viewing the avatar in the context of the world coordinates. Thus, the player expects the model to move relative to the world. If we only change `pos.z` with the up/down arrows, then the model will not necessarily move forward or backward relative to its *current orientation in the world*. Rather, one must modify the avatar's position in the direction of it's local z axis *relative to world coordinates*.



We want to define the direction of the model's positive z axis with respect the world's z axis. Let's consider the rotation between the two as θ. If θ equals 0, then the model's z axis is the same as the world's z: (0, 0, 1). If the rotation is 90°, then the model's z is perpendicular and has direction in the world (1, 0, 0).

Notice that what's actually happening is that we are definition rotations relative to the world's z axis using θ. This is really no different from the unit circle. **Except**, that the what is normally the x axis is now the z axis, and what is normally the y axis is now the x axis. Therefore, the direction of the model's z axis in world coordinates can be given as $(\sin(\theta), 0, \cos(\theta))$.

So, given θ, we can adjust the model's position using:

```
glm::vec3 pos(x,y,z);
glm::vec3 dir(sin(theta), 0, cos(theta));

// Move forward
if (glfwGetKey( window, GLFW_KEY_UP ) == GLFW_PRESS) {
    pos += dir * (deltaTime * speed);
}
// Move backward
if (glfwGetKey( window, GLFW_KEY_DOWN ) == GLFW_PRESS) {
    pos.z -= dir * (deltaTime * speed);
}
```

Okay, but now how do we adjust θ? Again, we have to realize that we are viewing the avatar using a third person perspective. Thus, the camera is **behind** the avatar. With such a camera pose, the avatar's local z axis appears to point directly forward **into the screen**. The avatar's local x axis points **to the left**. Therefore, rotating to the left (that is, counterclockwise) is actually increasing θ, as just defined. We are *increasing* the angle between the local z axis the world's z axis.

Therefore, rotation in the third person perspective would be given as:

```
// Rotate counterclockwise
if (glfwGetKey( window, GLFW_KEY_LEFT ) == GLFW_PRESS) {
    theta += deltaTime * speed;
}
// rotate clockwise
if (glfwGetKey( window, GLFW_KEY_RIGHT ) == GLFW_PRESS) {
    theta -= deltaTime * speed;
}
```

Putting it all together, we can define a model matrix using the avatar's world position and θ:

```
M = glm::mat4(1.0f);
M = glm::translate(M, pos);
M = glm::rotate(M, theta, up);
```

Notice that we translate first because we want the model to first move to that world position and *then* rotate in-place to have a particular orientation.

Finally, one last step. We need to place the camera. Since the camera has a fixed offset relative to the model, this is quite easy to compute. Say the camera's offset is such that it is behind the avatar. Therefore, the camera should be placed in the $-z$ direction relative to the avatar's local positive z direction. If the camera should be d units *behind* the avatar, the camera's world position and View matrix would be given by:

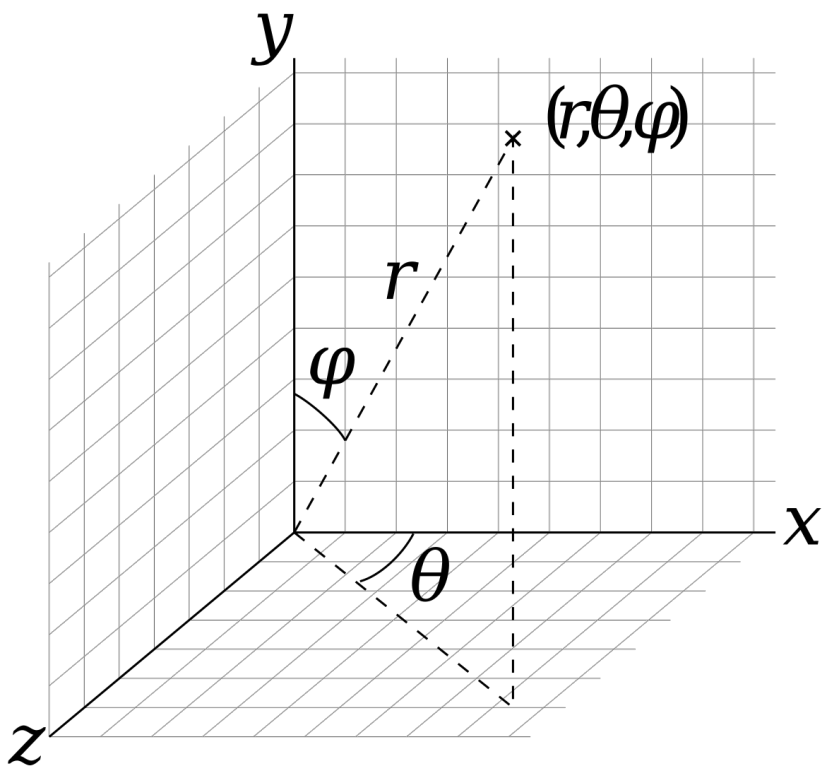
```
glm::vec3 camPos = pos;
camPos.x -= sin(theta) * d;
camPos.z -= cos(theta) * d;
V = glm::lookAt(camPos, pos, up);
```

We place the camera at the same position as the avatar. Then, move the camera d units backward, along the model's local z axis. This is `camPos` in world coordinates. Finally, define the View matrix by using `lookAt` to place the camera at `camPos` and look at the avatar.

14.4 Spherical Coordinates

In normal Cartesian coordinates, each point in space is specified by their x, y, z, the distances from the origin along each of the coordinate axes. In Spherical coordinates, each point in space is specified by three different parameters:

1. r: the distance of the point to the origin.
2. θ: the rotation of the point in the x-z plane, from 0°to 360°, measured from the positive x axis toward the positive z axis.
3. ϕ: the rotation of the point off the y-axis, from 0°to 180°, measured from the positive y axis toward the negative y axis.



Note that this definition is specialized to the case of OpenGL and is different from what you might get when Google-ing spherical coordinates. Each discipline (math, physics, graphics, etc.) has a different definition which is convenient for that.

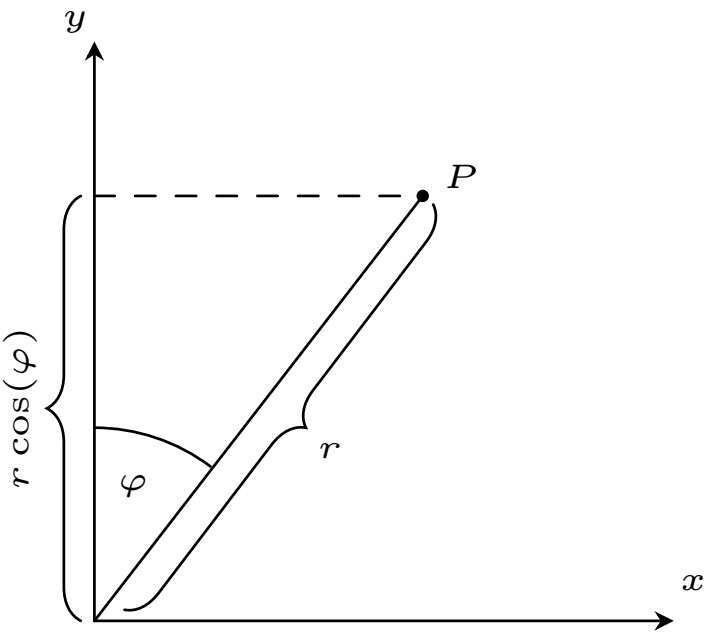
In our definition, r is the **radial distance5**, θ is the **azimuth angle**, and φ is the **polar angle** or **zenith angle**.

It is relatively easy to convert a Cartesian point (x, y, z) to is equivalent description in spherical coordinates. $r = \sqrt{x^2 + y^2 + z^2}$ regardless of the definition of which coordinate axis points in which direction. The computation of θ and ϕ is left as an exercise to the reader.

In the opposite direction, from spherical coordinates to Cartesian coordinates, we have:

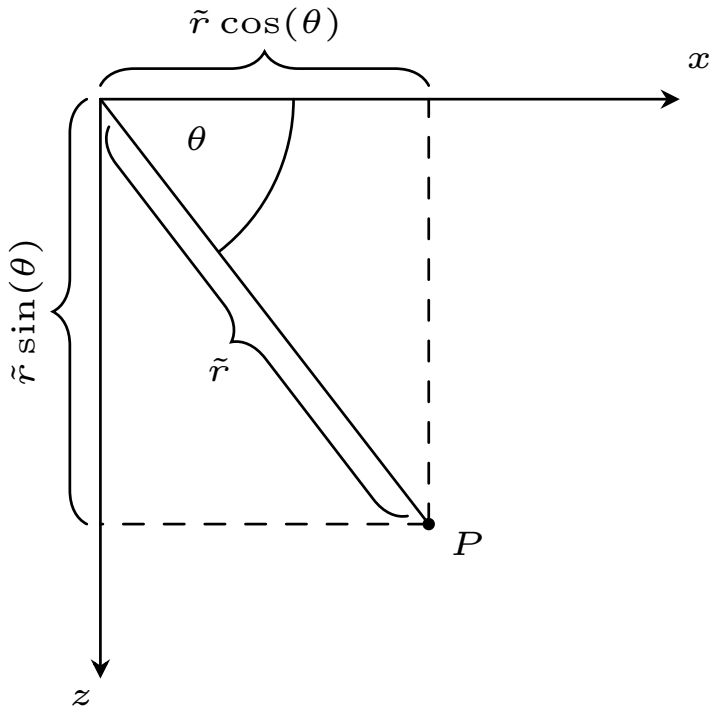
$$\begin{aligned} x &= r\cos(\theta)\sin(\varphi) \\ y &= r\cos(\varphi) \\ z &= r\sin(\theta)\sin(\varphi) \end{aligned}$$

Where do these formulas come from? Think about the line connecting the point P to the origin. This creates a right triangle when projected onto the x-y plane or the x-z plane.

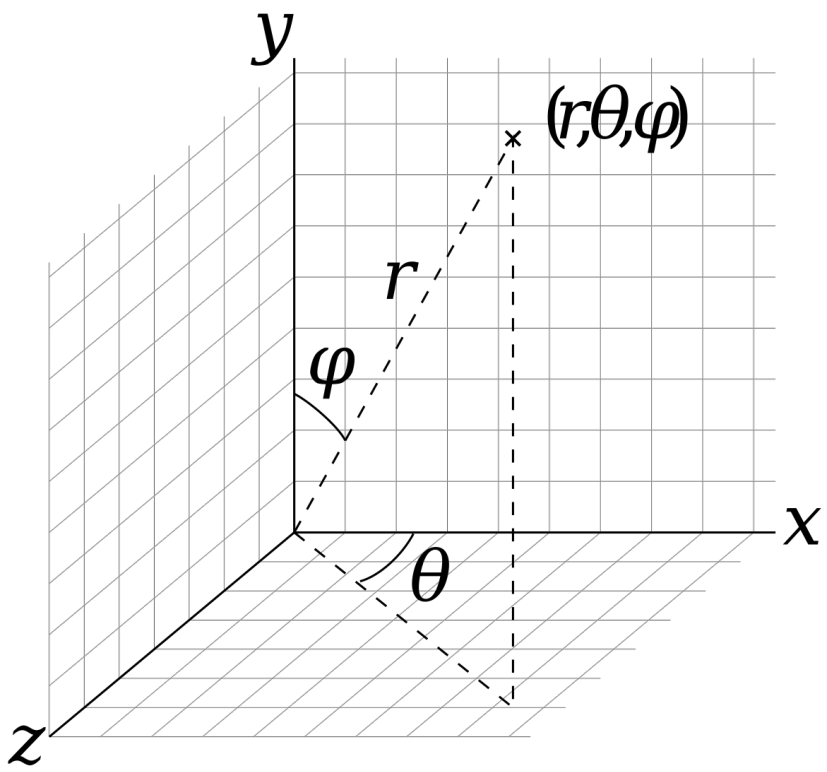


So, to get y in Cartesian coordinates we need to find the distance of P along the y axis. With our right triangle and φ, this is easily given as $r\cos(\varphi)$.

Now let's say we have P projected onto the $x-z$ plane. We can similarly find the Cartesian coordinates for x and z as $\tilde{r}\cos(\theta)$ and $\tilde{r}\sin(\theta)$, respectively.



But what's $\tilde{r}$? Consider again the 3D figure.



Notice that the "full length" of the line connecting the point to the origin exists in the x-y plane. That is, when $\theta = 0$. On the other hand, to project this line down to the x-z plane, some of the length of that line is lost. That is to say, for a fixed θ, different values of φ will produce different projections of the point down into the x-z plane.

The length after projection to the x-z plane is given by the complementary case to when we computed $y = r\cos(\varphi)$. We have $\tilde{r} = r\sin(\varphi)$. Putting it all together, we have:

$$\begin{aligned} x &= r\cos(\theta)\sin(\varphi) \\ y &= r\cos(\varphi) \\ z &= r\sin(\theta)\sin(\varphi) \end{aligned}$$

14.5 Moving the camera in Spherical Coordinates

Of course, spherical coordinates are introduced to make our lives easier, not just to look at fun formulas. Working in Spherical coordinates allows the camera to very easily move around in a sort of "globe" motion.

- The camera can maintain it's orientation and move closer or further from the origin by modifying r but keeping θ and φ the same.
- Given a particular distance r from the origin and a particular zenith φ (latitude), the cam-

era can "spin around" an object at a fixed elevation by simply varying θ. That is, the camera can vary though all the degrees of longitude at a fixed latitude.

- Given a particular azimuth θ (longitude), the camera can move along a line of longitude from the north pole to the south pole by only varying φ.

Using the arrow keys we only have two degrees of motion: up/down and left/right. Not enough to specify all three of our parameters. To add in an additional method of input, consider the mouse. We can "click and drag" with the mouse to affect θ and φ.

On a first frame that the mouse is clicked, cache it's current position. On all subsequent frames, compare the current mouse's position to the previous frame's position. If there is movement, say Δx and Δy, change the values of θ and φ, respectively.

```
static double mouseDownX;
static double mouseDownY;
static bool firstPress = true;

double dx = 0.0, dy = 0.0;
int state = glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT);
if (state == GLFW_PRESS)
{
    if (firstPress) {
        glfwGetCursorPos(window, &mouseDownX, &mouseDownY);
    }

    double xpos, ypos;
    glfwGetCursorPos(window, &xpos, &ypos);

    dx = xpos - mouseDownX;
    dy = ypos - mouseDownY;

    mouseDownX = xpos;
    mouseDownY = ypos;
}
if (state == GLFW_RELEASE) {
    firstPress = true;
}
```