
# 159.709 Computer Graphics

## Cameras

Daniel Playne
d.p.playne@massey.ac.nz

# Cameras

Most of the examples in this course have used view matrices that implement some simple functionality such as rotating around a central object.

This is a rather limited and manual way of controlling the view of the scene.

A more general way to control the view within a scene is to use a *camera*.

# Cameras

A camera is really just an object in the application that can be used to generate a view matrix.

There are many different types of camera that can be constructed and respond to user input in different ways.

We will start by looking at a ***Free Look*** camera.

# Free-Look Camera

A ***Free Look*** camera allows the user to freely explore the scene and look in any direction they wish.

The camera stores a *position* and an *orientation* that represents where the camera is and which direction it is pointing in.

The user can change the position and orientation of the camera (usually by using the keyboard and mouse).

# Free-Look Camera

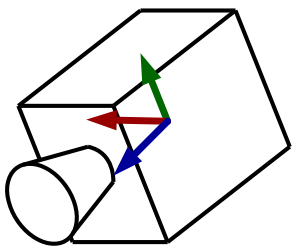
We will use a vector to store the position of the camera and a quaternion to store the orientation.

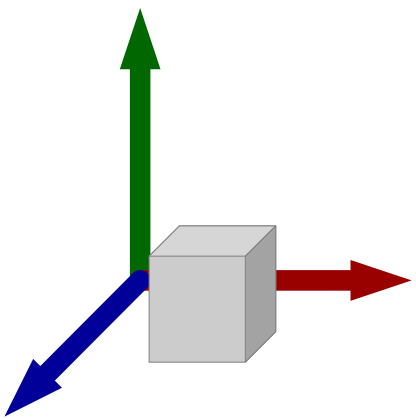
```
 1 class FreeLookCamera : public Camera {
 2 public:
 3     // Constructor
 4     FreeLookCamera(GLFWwindow *window);
 5     ...
 6 protected:
 7     // Data Members
 8     glm::vec3 mPosition;
 9     glm::quat mOrientation;
10     ...
11 };
```

# Free-Look Camera

The orientation of the camera can be used to calculate the ***forward***, ***up*** and ***right*** vectors of the camera.

Combined with the position, this is all that is necessary to determine the view from the camera.





# Free-Look Camera

The view matrix can be easily generated from these two data members as follows:

```
1 // Camera View Matrix
2 glm::mat4 FreeLookCamera::getViewMatrix() {
3     // Generate View Vectors
4     glm::vec3 forward = glm::vec3(mOrientation * glm::vec4( 0,  0, -1,  0));
5     glm::vec3 up      = glm::vec3(mOrientation * glm::vec4( 0,  1,  0,  0));
6
7
8     return glm::lookAt(mPosition, mPosition + forward, up);
9 }
```

# Free-Look Camera

For a **Free Look** camera that can rotate in any direction, we can accumulate rotations in the *orientation* of the camera.

New rotations are usually applied in the ***local*** frame of reference based on the current position and orientation of the camera.

One way of implementing this is to provide methods to change the *pitch*, *yaw* and *roll* of the camera.

# Free-Look Camera

These can be implemented as follows:

```
 1 void FreeLookCamera::pitch(float angle) {
 2     // Pitch
 3     mOrientation *= glm::quat(cos(angle/2.0f), glm::sin(angle/2.0f), 0.0f, 0.0f);
 4 }
 5 void FreeLookCamera::yaw(float angle) {
 6     // Yaw
 7     mOrientation *= glm::quat(cos(angle/2.0f), 0.0f, glm::sin(angle/2.0f), 0.0f);
 8 }
 9 void FreeLookCamera::roll(float angle) {
10     // Roll
11     mOrientation *= glm::quat(cos(angle/2.0f), 0.0f, 0.0f, -glm::sin(angle/2.0f));
12 }
```

# Free-Look Camera

Likewise moving the camera *forwards* and *backwards* (or *left* and *right*) is also usually based on the orientation of the camera.

To move the *forwards*, we calculate the vector representing the direction the camera is facing and add some multiple of this vector onto the position.

The camera can be moved *up/down* and *left/right* in a similar fashion.

# Free-Look Camera

This can be implemented as follows:

```
 1 void FreeLookCamera::moveForward(float x) {
 2     // Generate Forward Vector
 3     glm::vec3 forward = glm::vec3(mOrientation * glm::vec4( 0,  0, -1,  0));
 4 
 5     // Move Forward
 6     mPosition += forward * x;
 7 }
 8 void FreeLookCamera::moveRight(float x) {
 9     // Generate Right Vector
10     glm::vec3 right = glm::vec3(mOrientation * glm::vec4( 1,  0,  0,  0));
11 
12     // Move Right
13     mPosition += right * x;
14 }
```

# Gimbal Camera

The ***Free-Look Camera*** we have built can actually be a bit hard to use.

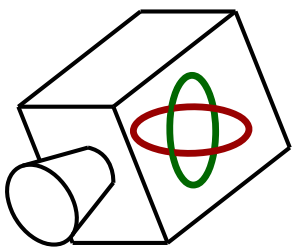
Because the rotations are applied in a local frame of reference, the camera can be rotated around the z-axis (roll) by a combination of rotations around the x-axis (pitch) and y-axis (yaw).

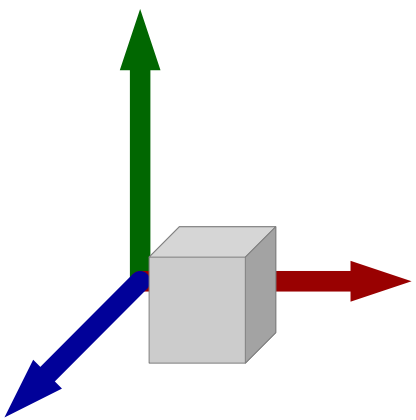
While this would be useful for an application such as an aeroplane, it can be disorientating and confusing when we are used to a gimbal system.

# Gimbal Camera

We can still implement and use a gimbal camera if we prefer, the correct choice will depend on our application.

For this camera, a separate rotation for the x- and y- axes will be stored and applied in the same order.





# Gimbal Camera

The Gimbal camera inherits the position and orientation from the FreeLookCamera class and adds an x and y view angles.

```
1 class GimbalFreeLookCamera : public FreeLookCamera {
2 public:
3     ...
4 private:
5     // Data Members
6     float mAngleX, mAngleY;
7 };
```

# Gimbal Camera

The `pitch`, `yaw` and `roll` functions can be overridden to limit the movement to a gimbal.

```
 1 void GimbalFreeLookCamera::pitch(float angle) {
 2    mAngleX += angle;
 3
 4    if(mAngleX < -glm::pi<float>() / 2.0f) {
 5       mAngleX = -glm::pi<float>() / 2.0f;
 6    } else if(mAngleX > glm::pi<float>() / 2.0f) {
 7       mAngleX = glm::pi<float>() / 2.0f;
 8    }
 9    mOrientation =
10     glm::quat(cos(mAngleY/2.0f), glm::vec3(0.0f, glm::sin(mAngleY/2.0f), 0.0f)) *
11     glm::quat(cos(mAngleX/2.0f), glm::vec3(glm::sin(mAngleX/2.0f), 0.0f, 0.0f));
12 }
13 void GimbalFreeLookCamera::yaw(float angle) {
14    mViewAngleY += angle;
15    mOrientation =
16     glm::quat(cos(mAngleY/2.0f), glm::vec3(0.0f, glm::sin(mAngleY/2.0f), 0.0f)) *
17     glm::quat(cos(mAngleX/2.0f), glm::vec3(glm::sin(mAngleX/2.0f), 0.0f, 0.0f));
18 }
19 void GimbalFreeLookCamera::roll(float angle) {
20    // No Roll for Gimbal Free Look Camera
21 }
```

# Gimbal Camera

Note that in the `pitch` function the rotation around the x axis is limited to a range $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$.

Bounds such as these are often used in applications to prevent the user looking upside down.

```
 1 void GimbalFreeLookCamera::pitch(float angle) {
 2    mAngleX += angle;
 3
 4    if(mAngleX < -glm::pi<float>() / 2.0f) {
 5       mAngleX = -glm::pi<float>() / 2.0f;
 6    } else if(mAngleX > glm::pi<float>() / 2.0f) {
 7       mAngleX = glm::pi<float>() / 2.0f;
 8    }
 9    mOrientation =
10     glm::quat(cos(mAngleY/2.0f), glm::vec3(0.0f, glm::sin(mAngleY/2.0f), 0.0f)) *
11     glm::quat(cos(mAngleX/2.0f), glm::vec3(glm::sin(mAngleX/2.0f), 0.0f, 0.0f));
12 }
```

# Types of Cameras

There are many different types of cameras that can be implemented for computer graphics.

The two we have looked are both ***First-Person*** cameras, the view of the scene is from the *player's* point-of-view.

Another common type of camera is a ***Third-Person*** camera where the user controls some character or object within the scene and the camera shows a view of the scene that includes this character.

# Third-Person Cameras

***Third-person cameras*** may be connected to the character they view by a *boom* which can be rotated or zoomed in and out by the user.

They may also implement some kind of spring that allows the boom to respond smoothly to the position of the character.

Cameras may also interact with the objects in the scene to prevent them accidentally moving 'inside' a solid object.