First Person Camera

Head banging in our game

CONTENTS

FIRST PERSON CAMERAS

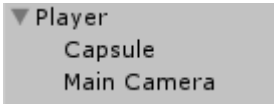
First person refers to the camera being from the viewpoint of the character in the game. You can use the mouse or controller to look around, while you move in the environment

Even through it is just a view, to actually make the game work nicely, you can set it so that the direction you face is the direction that your whole body moves so you walk in that direction.

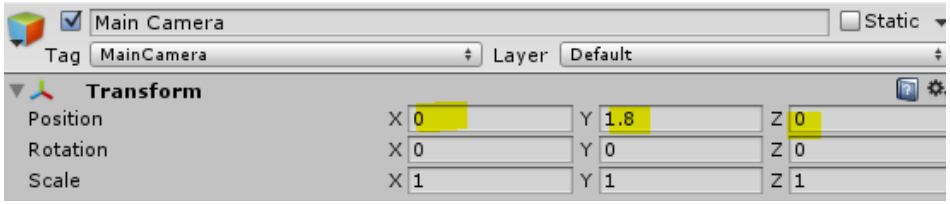
Before working on this first person rotation you will want to go through the movement handout.

POSITIONING THE CAMERA

For a first-person camera, because the camera is the eyes of the player, the camera should be a part of the player game Object.

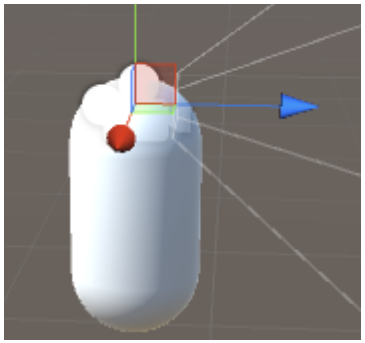


1. Make the camera a child of the player
2. Set the cameras position so that it is inside the player and raised to eye level.
    a. X =0
    b. Y= 1.8 - 1.8 Meters tall is often used
    c. Z = 0



Standard height for player characters in games is around 1.8 meters so, we emulate this by positioning the camera around where the eye is.

3. Test it

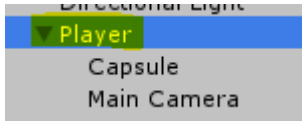


We will be able to move, but our view is locked to one position.

Let's change that

Left and Right rotation

First, we shall get our camera view to be able to move left and right. To do this, we want to not only move the camera, but move the player. This will have it so we are always facing where we look and if we move we will move in that direction. We will move the Player Game objects rotation to do this



To modify the left and right we modify the "yaw rotation"

- In the **PlayerMovement** Script
  1. Create a new **void** function called **CameraMovment**
- Inside CameraMovment()
  2. Create a **float** called **rotateYaw**
     a. Set its value to equal Input.GetAxis("Mouse X")
     b. Remember case sensitivity
  3. Pass in rotateYaw into the transforms Rotate function
     c. Pass it into the y axis

```
void CameraMovement()
{
    float rotateYaw = Input.GetAxis("Mouse X");
    transform.Rotate(0, rotateYaw, 0);
```

Next, we need to call this function so it is used.

- In the Update() function
  4. Call the CameraMovment() function

```
void Update()
{
    CameraMovement();
    Inputs();
    Movement();
}
```

MOVING IN THE RIGHT DIRECTION

This rotation Works, however the rotation does not affect the movement position. If we look one way, the players forward direction is not changed.

We always move in the same directions, no mater how.

We can fix this by multiplying the speed we are traveling in by the rotation transform Vector.

This works because we have a vector pointing in the direction we want to move in, set by our rotation code. The direction vector is just the speed we want to move towards, so multiplying them gives us our speed and direction.

- In the **Movement()** function
  - Just before **cc.Move** is called
    3. Set the speed variable so that it is equal transform.rotation times the speed
       a. Has to be in the order shown

```
    direction.y = verticalVelocity;
    //Camera Add
    direction = transform.rotation * direction;

    cc.Move(direction * Time.deltaTime);
}
```

First Person Camera

Head banging in our game

CHANGE THE MOUSE SENSITIVITY

The mouse movement is not necessarily fast enough for the player, we might want to have a way to change the mouse sensitivity.

- In the **class block**
  4. Create a public variable called mouseSensitivity
     b. Set it to equal 2.0f
- In the CameraMovment() function
  5. Change it so that rotateYaw has mouseSensitivity multiplied to it

```
public class CharacterMovement : MonoBehaviour {
    public float mouseSensitivity = 2.0f;
```

```
void CameraMovement()
{
    float rotateYaw = Input.GetAxis("Mouse X") * mouseSensititivity;
    transform.Rotate(0, rotateYaw, 0);
```

First Person Camera

Head banging in our game

PITCH ROTATION (UP AND DOWN)

If we wanted our camera to move up and down (pitch rotation), we can't affect our character controller as it does not allow to be rotated that way. We would not want to anyway, as it would make our character dip up and down. So, to change our pitch view, we need to affect the camera itself.

- In the **class** block
  1. Create a public camera called firstPersonCam

```
public class PlayerMovement : MonoBehaviour {
    public float mouseSensitivity = 2.0f;
    Camera firstPersonCam;
```

We want to know which camera to use and, because we have the camera attached to the character, we can easily search for it in the script

- In the **Update** function
  2. **Set** firstPersonCamera to equal GetComponentniChilderen<Camera>().

This is like GetComponent, but searches through all the child objects the script is attached to

```
void Start()
{
    cc = GetComponent<CharacterController>();
    firstPersonCam = GetComponentInChildren<Camera>();
}
```

First Person Camera

Head banging in our game

Now we have the camera, we can set the the cameras rotation for our pitch

- In the **CameraMovement**() function
  3. Create a **float** called **rotatePitch**
     a. Set it to equal **Input.GetAxis** for the “Mouse Y” input
     b. **Multiply** it by **mouse sensitivity**
  4. Pass the **float** into the **x axis** of the **playerCams** transform Rotate function

You will find this works, however the rotation is inverted. To change it so that you rotate in the opposite direction, we have to set the Input to a negative value.

5. Change it so that the input is a negative value

```
void CameraMovement()
{
    float rotateYaw = Input.GetAxis("Mouse X") * mouseSensititivity;
    transform.Rotate(0, rotateYaw, 0);

    float rotatePitch = Input.GetAxis("Mouse Y") * mouseSensititivity;
    firstPersonCam.transform.Rotate(rotatePitch,0,0);
```

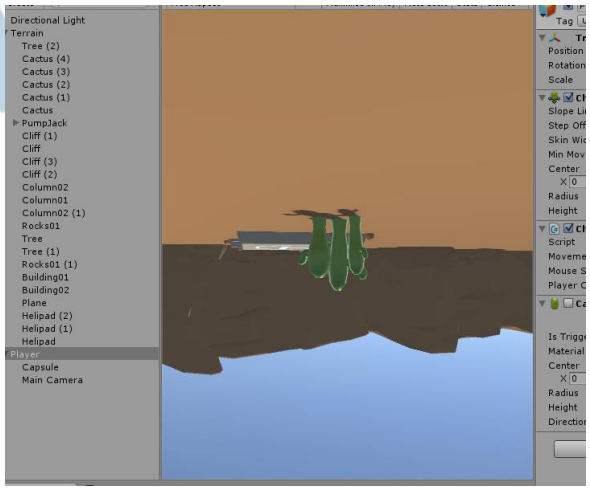
```
void CameraMovement()
{
    float rotateYaw = Input.GetAxis("Mouse X") * mouseSensititivity;
    transform.Rotate(0, rotateYaw, 0);

    float rotatePitch = -Input.GetAxis("Mouse Y") * mouseSensititivity;
    firstPersonCam.transform.Rotate(rotatePitch,0,0);
```

All seems to be going well. But what happens if I go up, and up, and up, and up? You end up with a weird view point.

We want to limit how much we far you can tilt your head back.

We can do this by using a clamp.



CLAMP

The Mathf.Clamp function is a function that allows us to clamp a value between two values. If it goes over the max or under the minimum numbers, it will instead stay at the max or min values until the number goes back between the minimum and maximum numbers

First Person Camera

Head banging in our game

We want to clamp it down so it can only rotate, let's say 60 degrees. First we need a variable to hold our current rotation

- In the class block

1. Create a float variable called rotatePitch

This will set hold our current rotation which will be grabbed for checking

2. Add a PitchRange float
    a. Set its value to 60.0f;

The current way we are rotating (the Rotation function) will not work how we need it to work. This is because the Rotate function adds the amount given to it to the previous rotation angle. Instead we will set the full rotation manually using the localRotation variable.

- In update

3. Set it so rotatePitch is not initiated
    a. Remove the float at the front
4. Set it so rotatePitch is equal itself, plus the Input
5. Remove the float at the start
6. Use Mathf.Clamp to clamp rotatePitch
    a. Minimum to –pitchRange
    b. Maximum to pitchRange
7. Change playerCam to be localRotation
    a. Set to equal Quaternion.Euler
        i. RotatePitch, 0, 0

This should allow us to look up and down nicely.

```
public class FirstPersonCamera : MonoBehaviour {

    public float mouseSensitivity = 2.0f;

    public Camera playerCam;

    float rotatePitch;
    float pitchRange = 60.0f;
```

```
void CameraMovement()
{
    float rotateYaw = Input.GetAxis("Mouse X") * mouseSensititivity;
    transform.Rotate(0, rotateYaw, 0);

    rotatePitch += -Input.GetAxis("Mouse Y") * mouseSensititivity;
    rotatePitch = Mathf.Clamp(rotatePitch, -pitchRange, pitchRange);
    //firstPersonCam.transform.Rotate(rotatePitch,0,0); - REMOVE
    firstPersonCam.transform.localRotation = Quaternion.Euler(rotatePitch, 0, 0);
}
```

LOCKING THE CURSOR (LOCKSTATE)

Often, in first person games, we want to lock it so that the pointer is locked to the centre of the game screen.

We can set the locked state using a variable in the Class Cursor called LockState

Cursor keeps track of all thing's cursor including the lockState variable. We can set the lockState by using the CursorLockMode enum which we can set the lock in 3 different ways

- Confine – Confines the mouse cursor to the game screen
- Locked – Confines the mouse cursor to the centre of the screen
- None – Unlocks the mouse cursor

- In the Start function
  4. Access Cursors lockState variable
     a. Set it to equal CursorLockMode.Locked

```
0 references
void Start()
{
    cc = GetComponent<CharacterController>();
    firstPersonCam = GetComponentInChildren<Camera>();

    Cursor.lockState = CursorLockMode.Locked;
}
```

Your mouse should now be locked to the centre of the screen.

To escape the game in the Engine you will need to click the **esc button**, or **alt tab** to another program.

To escape from a build, use **alt tab**