CAMOSUN COLLEGE

COMPUTER SCIENCE DEPARTMENT

ICS 123 - GAMING AND GRAPHICS CONCEPTS

LAB 3 – ADDING A PLAYER TO A 3D SPACE

DUE: DEMO BY END OF LAB PERIOD MON FEB 6
You must work individually for this lab.
Submission Instructions are on the last page of the lab.

Objective:

To have the students add a player into the 3D space they created in Lab 2 and introduce first-person controls to move within the 3D space.

Preparation:

Supplemental Video Tutorials:

https://unity3d.com/learn/tutorials/topics/interface-essentials
https://unity3d.com/learn/tutorials/topics/scripting

Background:

Unity's Component System:

In Object-Oriented programming, it is often required to define relationships between classes. This can be accomplished in one of two ways: Inheritance or Composition. Inheritance is a relationship like in figure 1 below:

```
class Banana(Fruit):
      ...
```

Figure 1a: Inheritance in Python

```
class Banana extends Fruit {
      ...
}
```

Figure 1b: Inheritance in Java

```
class Banana : Fruit {
    ...
}
```

Figure 1c: Inheritance in C#

In inheritance, the child or subclass (class Banana) would inherit all the attributes and methods of class Fruit. That is, an object of class Banana is an object of Class Fruit with some extras (in the form of additional attributes and methods).

Composition, on the other hand, establishes a relationship like so:

```
class Fruit:
       ...

class Banana:
    def __init__(self):
       self.fruit = Fruit()
```

Figure 2a: Composition in Python

```
class Fruit { ... }

class Banana {
    private Fruit fruit;

    public Banana() {
       this.fruit = new Fruit();
    }
}
```

Figure 2b: Composition in C#/Java

In this case, class Banana contains a reference to an instance of the Fruit class. Instead of extending Fruit's attributes and behaviours, it *contains* Fruit's attributes and behaviours. Therefore, you could say that Fruit is a *component* of Banana.

If you do some research, you'll find there is much debate about which approach is better for defining relationships between classes. In Unity, there is no debate - Unity's game objects use composition (but as you saw in Lab 1, components themselves can use inheritance). The Unity component system is integrated in the Visual Editor. For example, you have already seen the following components:

- Transform component – defines a game object's position, rotation, and scale in the scene. Every game object must have this component.
- Script component – attaches a script to a game object. Script components are often used to define a game object's behaviours and interactions.
- Collider component – define the shape of an object for the purpose of physical collisions.
- Mesh renderer component – draws objects in the scene.

In this lab, you will add and remove components using the Visual Editor!!

Local vs. Global Coordinate Space:

In Tasks Part 3 of this lab, you are going to start moving the Player by first Rotating the player object. By default, when you transform an object in code, you are transforming it using the object's local coordinates. What does that mean?

Recall in Lab 2 when you built your 3D Space. When you positioned all the wall and floor objects relative to each other, you were using global coordinates. Global coordinates mean that the direction of the X, Y, and Z axis never changes.

Each object that you place in your 3D space, however, has its own local coordinates. That means that each object has its own X, Y, and Z axis that can change direction based on how you transform the object.

If this sounds confusing, don't worry! Hopefully Tasks Part 3 Step 6 will clear it up for you!

Understanding Object Movement:

Recall from Lab 1 that a Script component comes with a method you can override called `Update()`. Recall also that *Update()* is called once every game frame. If you move an object slightly by changing either its position, rotation, or scale in the *Update()* method, then that change will get recorded every frame. When you play the game, and assuming a high enough frame rate, it will appear that the object is moving! This is the exact same principle they use in movies! One big difference is that in games, the frame rate can vary (in movies, it's constant) because different hardware process code and graphics at different speeds. If you have code that doesn't take this into account, it is termed *frame rate dependent*. This can be a serious problem especially with older games! Ideally, you want any movement code to be *frame rate independent*. Unity has a solution for this! A static variable called `deltaTime` that is a property of the `Time` class stores the amount of time it took to complete the last frame, in seconds. For example, if a game were running at 30 fps, then each frame takes 1/30 of a second and that value will be stored in `deltaTime`. Likewise, if a game were running at 60 fps, then each frame takes 1/60 of a second. As you can see, a faster frame rate means `deltaTime` will contain a smaller value. We can therefore make our movement code *frame rate independent* by simply multiplying it with `deltaTime`.

Unity API References for this Lab:
https://docs.unity3d.com/ScriptReference/Transform.Rotate.html
https://docs.unity3d.com/ScriptReference/Input.GetAxis.html
https://docs.unity3d.com/ScriptReference/Mathf.Clamp.html
https://docs.unity3d.com/ScriptReference/Transform-localEulerAngles.html
https://docs.unity3d.com/ScriptReference/Vector3.html
https://docs.unity3d.com/ScriptReference/Transform.Translate.html

Tasks:

Tasks Part 1 – Importing Your Scene from Lab 2 into a New Project:

1. You will start a new Project for Lab 3 but you will want to import the Scene you created from Lab 2. First, open your Lab 2 project in Unity. Go up to the menu bar and select *Assets -> Export Package*. This should open the Exporting Package window:

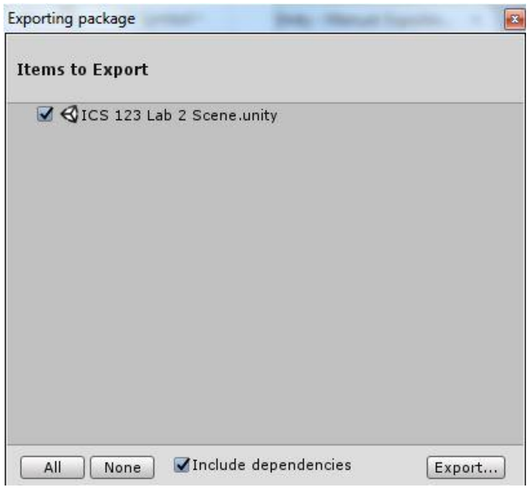


Figure 3: The Exporting package window

   Ensure that your scene is checked and that *Include dependencies* is checked. This ensures that all Assets associated with the scene will also be exported. Hit the *Export…* button.

2. Save this package under your Lab 3 folder (create one if you haven't yet!). Save this new package as 'ICS123Lab2Scene'. A *.unitypackage* extension will automatically be added.

3. Now go up the menu bar and select *File -> New Project*. Name this new project 'ICS 123 Lab 3'. Hit the 'Create project' button. This will close your current Lab 2 Project and relaunch Unity with a new empty Lab 3 Project.

4. Once the new empty project is loaded in Unity, go up to *Assets* and select *Import Package -> Custom Package*. Browse for the package you created in Step 2. An Import Unity Package window will pop-up. Ensure your Lab 2 Scene is checked and hit the Import

button.

5. In the Project Tab under Assets, you can double click on your Lab 2 Scene to load it up into the Scene Tab. You're done! You have successfully transferred the Scene you created in Lab 2 into a new Unity Project.

Tasks Part 2 – Adding a 3D Object to the Scene to Represent the Player:

1. Now that you have finished the Scene, it is time to add the player into your 3D world. For this project, you'll use a simple primitive shape. Go up to the *GameObject* menu at the top and select *3D Object -> Capsule*. This will create a new Capsule object to place in the Scene. Under the Inspector Tab and in the Transform Component, set the Position of this object as X = 0, Y = 1.1, and Z = 0. Also, rename this to Object to Player (Right-click on the Object in the Hierarchy Tab and select Rename). You should see the Player in your 3D world!

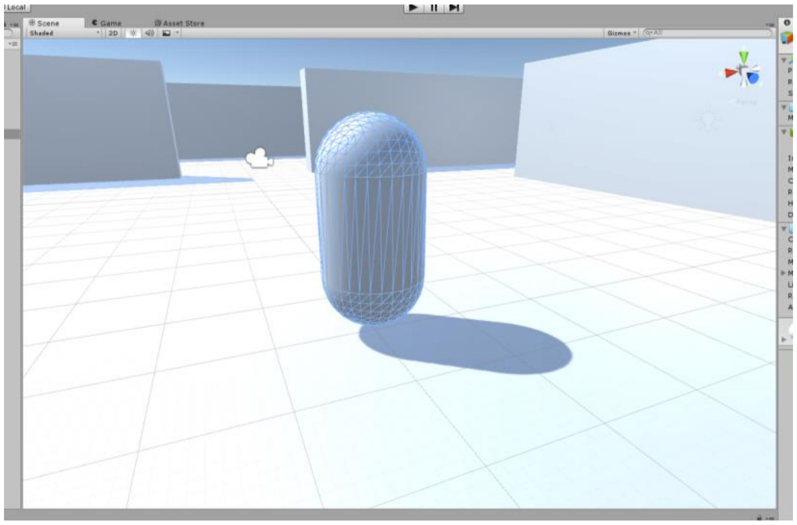
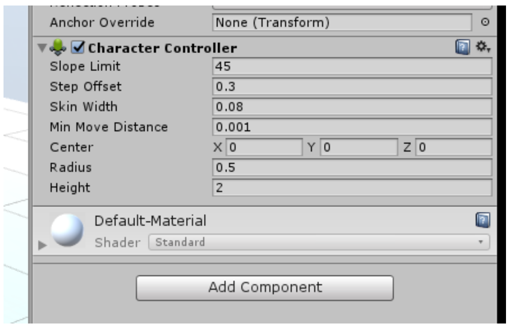


Figure 4: Yes, that's the Player! A Giant Pill. Very Retro.

2. If you look in the Inspector Tab, you'll notice that the Capsule Object has a Capsule Collider component. Collider components allow objects in a Scene to interact with each other. But this Capsule Object is special, it's the Player! You need a slightly different component than a collider. Click on the gear icon on the far right of the Collider component and select *Remove Component*. This should remove the Collider component from the Object.

3. You want to add what's called a Character Controller component. To do so, click on the ***Add Component*** button at the bottom of the Inspector Tab. This will bring up a menu. Select *Physic -> Character Controller*. This should add the Character Controller to the Player Object.



Anchor Override: None (Transform)

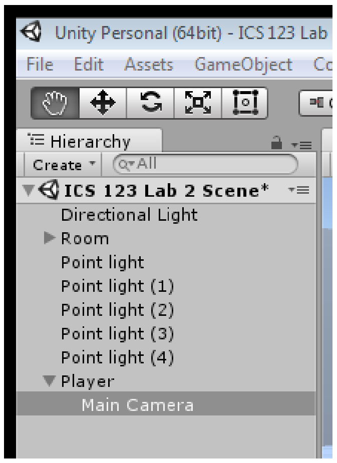
Character Controller

Slope Limit	45
Step Offset	0.3
Skin Width	0.08
Min Move Distance	0.001
Center	X 0 Y 0 Z 0
Radius	0.5
Height	2

Default-Material
Shader: Standard

Figure 5: The new Character Controller Component
(Also showing the Add Component button at the bottom)

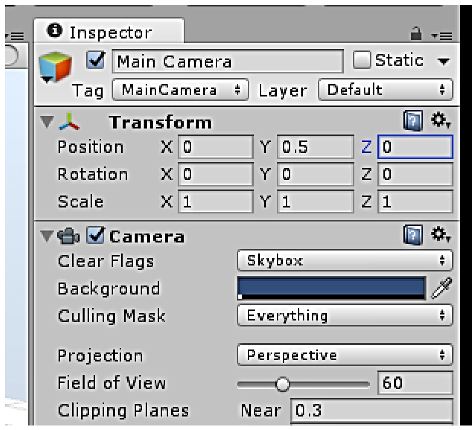
4. The last thing you need to do is to add the 'eyes' for the Player. The eyes are a camera. A camera will have already been added to the scene by default (Main Camera that you can see in the Hierarchy Tab). This is a first-person shooter so you want to attach this camera to the Player object. This is easy - drag the Main Camera onto the Player object in the Hierarchy Tab. Once you've done this, position the camera so that X = 0, Y = 0.5, and Z = 0.



Unity Personal (64bit) - ICS 123 Lab
File Edit Assets GameObject Co
Hierarchy
Create
All

ICS 123 Lab 2 Scene*
- Directional Light
- Room
- Point light
- Point light (1)
- Point light (2)
- Point light (3)
- Point light (4)
- Player
  - Main Camera

Figure 6a: The Main Camera attached to the Player Object



Main Camera Static
Tag: MainCamera Layer: Default

Transform

	X	Y	Z
Position	0	0.5	0
Rotation	0	0	0
Scale	1	1	1

Camera

Clear Flags	Skybox
Background	
Culling Mask	Everything
Projection	Perspective
Field of View	60
Clipping Planes	Near 0.3

Figure 6b: Correct Position for the Camera

Now the Player can see your 3D world. But how do you make the Player move and look around??

Tasks Part 3 – Rotating the Player Object:

1. For starters, get the Player object to spin. Go up to the Assets menu item and select *Create -> C# Script*. Name this script Spin.

2. Double click on this Script in the Project Tab. This should open it up in **MonoDevelop**. This is the same procedure you did in Lab 1.

3. In Lab 1, you modified the *Start()* method. The *Start()* method as you might recall is called only once when the object is first loaded in the Scene. This time, you are going to modify

the *Update()* method. Add the following statement in *Update()*:

```
transform.Rotate(0, speed, 0);
```

Also, add the following global variable (should be declared and initialized right below the class declaration):

```
public float speed = 3.0f;
```

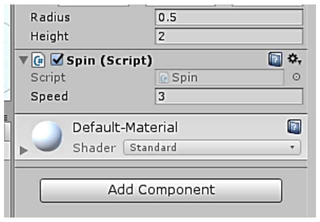
Now you could have declared this variable right in the *Update()* method but Unity has a neat trick up its sleeve. Variables declared with the public access modifier show up in Unity's Visual Editor in the Inspector Tab. What that means is that you can easily modify this value without having to go back into **MonoDevelop**.

The code for your Spin class should look like Figure 7 below:

```
public class Spin : MonoBehaviour {
    public float speed = 3.0f;

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {
        transform.Rotate (0, speed, 0);
    }
}
```

Figure 7: Spin Class with Modified Code

Close **MonoDevelop**. When it asks you to *Save Files* make sure to press the ***Save and Quit*** button. Back in Unity, you need to attach your Script to the Player object. To do so, simply drag your script in the Assets window in the Project Tab onto the Player object listed in the Hierarchy Tab. Once completed, click on the Player object in the Hierarchy Tab to confirm your Script is now a component of this object by looking at its components in the Inspector Tab.



Radius 0.5
Height 2
Spin (Script)
Script Spin
Speed 3
Default-Material
Shader Standard

Figure 8: Spin Component shown in the Inspector Tab (notice `speed` shows up!)

4. As you can see in figure 8, the `speed` variable shows up in the Visual Editor. Now you have a nice easy way to change the speed of rotation right inside the Editor! Hit the Play button above the Scene Tab to see what happens. To stop the Scene, just hit the button again. Try changing the value of the Speed variable and hitting Play again. What do you notice when you enter a negative value?

   Discussion of Code:

   1. `public float speed = 3.0f;`
      This is exactly the same as in Java. An 'f' has to be used at the end of the floating-point number to indicate it is a float value and not a double (which C# assumes by default)

   2. `transform.Rotate(0, speed, 0);`
      transform is the name given to the Transform object that is a component of the Player object. *transform* is an instance of the Transform class. You didn't have to instantiate this object yourself, Unity does it for you.

      Rotate() is a method you can call on the transform object. In this case, you gave it three arguments: 0, speed (which initially was 3.0f), and another 0. Each of these represent the X, Y, and Z axes respectively. Any value given represents the number of degrees to rotate per. frame. Since X and Z were set to 0, your player doesn't rotate on either of those axes. Instead, the player will only rotate on the Y axis, three degrees per. frame. If you forget where each axes are, refer back to Lab 2 and the ***<u>left-hand coordinate system</u>***!! Try moving the speed variable around. What happens when you replace the X or Z axis with this variable?

5. *Rotate()* by default uses local coordinates. When you rotated the Player object, you were actually rotating it around its own X, Y, or Z axes. However, initially the direction of the local X, Y and Z axes matches that of the global ones. To demonstrate the difference, do the following:

   a. Before playing the Scene again, first tilt the Player object. With it selected in the Hierarchy Tab, change the X Rotation to 30 in the Transform component of the Inspector Tab. The direction of the local X, Y, and Z axes (the local coordinates) will now also be tilted 30 degrees*. Note:* you will see a green outline after you have tilted the Player. This has to do with the Character Controller component and detecting collisions. Don't worry about it for this exercise!

   b. Now play the Scene again. You will notice that the view the camera sees is different! This is due to the direction of the local axes being tilted.

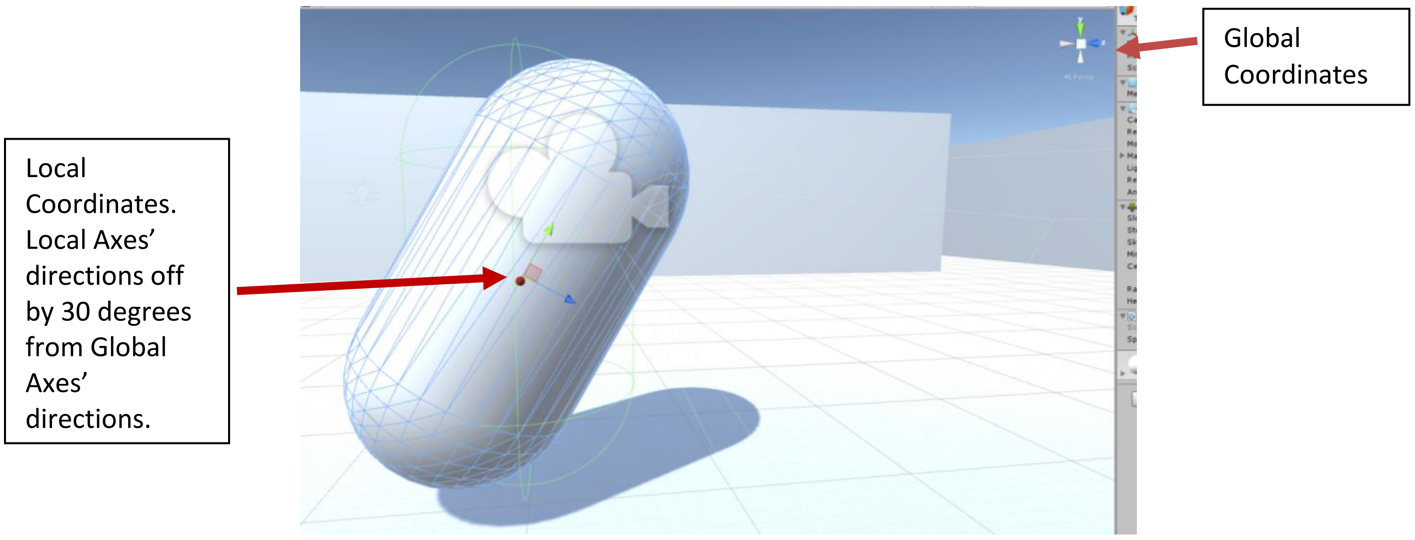

Local Coordinates. Local Axes' directions off by 30 degrees from Global Axes' directions.
Global Coordinates

Figure 9: Player Object tilted by 30 degrees

c. Is there a way to rotate the Player Object using the Global direction of an axis? The answer is yes. Double click on your Spin Script in the Assets window of the Project Tab to open it in **MonoDevelop**. Add an optional fourth parameter to your call to *Rotate()*:

```
transform.Rotate(0, speed, 0, Space.World);
```

d. Save your updated Script and close **MonoDevelop**. Run your Scene again. The view in the Game Tab should look different. This time, it will be rotating using one of the global axes. You can remove that optional fourth argument again to compare! When you are done, change the X Rotation value back to 0 in the Transform component to remove the 'tilt' on the Player object.

Tasks Part 4 – Adding Mouse Input to the Player Object to Adjust the Player's View:

1. Create a new C# script (in Unity, Assets -> Create -> C# Script) and call it *MouseLook*. Double click on it to open it in **MonoDevelop**.

2. Modify it to look like the figure on the next page.

```
using UnityEngine;
using System.Collections;

public class MouseLook : MonoBehaviour {
    // enum to set values by name instead of number.
    // makes code more readable!
    public enum RotationAxes
    {
        MouseXAndY = 0,
        MouseX = 1,
        MouseY = 2
    }

    // public class-scope variable so it shows up in Inspector
    public RotationAxes axes = RotationAxes.MouseXAndY;

    // Update is called once per frame
    void Update () {
        if (axes == RotationAxes.MouseX) {
            // horizontal rotation here
        } else if (axes == RotationAxes.MouseY) {
            // vertical rotation here
        } else {
            // both horizontal and vertical rotation here
        }
    }
}
```

Figure 10: Initial *MouseLook* Script

The conditional statement inside the *Update()* method allows us to isolate on which axis the player's view will rotate. For starters, you will write the code to only allow rotation horizontally (around the Y axis). Then you will isolate the rotation to be only vertical (along the X axis). Finally, you will combine the two to allow full horizontal and vertical rotation.

3. Back in Unity, click on your Player object in the Hierarchy Tab. Over in the Inspector Tab, remove the Spin Script Component (click on the little gear on the right and select *Remove Component*). Now drag your *MouseLook* Script from the Assets window of the Project Tab onto the Player object in the Hierarchy Tab to attach the Script to the Player.

4. You won't be needing the Spin Script anymore so you can delete it and remove it from your project by right clicking on it in the Assets window and selecting Delete (remember always do this inside Unity!!!).

Tasks Part 4a – Responding to the Mouse for Horizontal Rotation:

5. After the axes variable declaration, declare the following public variable:

```
public float sensitivityHorz = 9.0f;
```

In the first *if* block of the conditional statement after the '*horizontal rotation here*' comment, add the following statement:

```
transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHorz, 0);
```

You'll notice that we are now using a method from a new class called *Input*. This class handles mouse, keyboard, and joystick inputs. The "Mouse X" *GetAxis()* argument tells this method to return the change in location of the mouse horizontally. Our rotation speed (this time called `sensitivityHorz`) is then multiplied by this value. The result is the player's view will spin around according to mouse input!

6. Save your script and close **MonoDevelop**. Click on the Player object in the Hierarchy Tab. In the Mouse Look Component of the Inspector Tab, change the Axes dropdown to *Mouse X*. Now play your Scene and move the mouse!

Tasks Part 4b – Responding to the Mouse for Vertical Rotation:

7. Now you'll add vertical rotation. Unlike horizontal, you will want to put limits on how much the player can tilt (assuming you don't want your Player to be standing on her head in the scene!). We also won't be able to use *Rotate()* anymore because it calculates angle cumulatively. Double click on your *MouseLook* Script in the Assets window of the Project Tab to open it in MonoDevelop. Add the following declarations just below where you declared `sensitivityHorz`:

```
public float sensitivityVert = 9.0f;

public float minVert = -45.0f;
public float maxVert = 45.0f;

private float _rotationX = 0.0f;
```

The _*rotationX* variable is private so we use an underscore as a convention to indicate it has class only scope. This variable will store the rotation angle (which we need this time to ensure it doesn't go out of bounds). The variables *minVert* and *maxVert* define the minimum and maximum angles the Player's view can tilt (in this case 45 to -45 degrees).

Now add the following code inside the *else if* block just after the *'vertical rotation here'* comment:

```
_rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
_rotationX = Mathf.Clamp (_rotationX, minVert, maxVert);

transform.localEulerAngles = new Vector3 (_rotationX, 0, 0);
```

The first statement will get the change in mouse movement vertically. Since we need to ensure that this doesn't go out of our defined limits, it must be stored in a variable. You'll notice the -= operator. We use this so that when you mouse forward (away from you) your view will point up (+ve X rotation). Likewise, when you move the mouse back (towards you) your view will point down (-ve X rotation). If you change the operator to +=, then it will be reversed.

The second statement does the limits check. *Clamp()* is a static method in the floating point Math library (Mathf) which will clamp `_rotationX` to `minVert` and `maxVert` (-45 to 45).

The third statement actually sets the view angle of the Player object. There are a few things going on here. First, *localEulerAngles* is a property of transform (Euler angles are angles made against the X, Y, and Z axes). It defines the x, y, and z rotation of the object. The type of this property is *Vector3*. This is a structure defined in Unity. It represents a 3D Vector (a point that has magnitude and direction in the X, Y, and Z directions). Transform vectors are non-mutable. That means we can't simply modify the existing vector of the Player object; instead, a new Vector has to be instantiated. This new Vector has Y and Z set to 0 since we are only rotating in the vertical direction (X axis).

8. Save your script and close **MonoDevelop**. Click on the Player object in the Hierarchy Tab. In the Mouse Look Component of the Inspector Tab, change the Axes dropdown to *Mouse Y*. Now play your Scene and move the mouse!!!

Tasks Part 4c – Responding to the Mouse for both Horizontal and Vertical Rotation:

9. Stop your Scene and double click on *MouseLook* in the Assets window of the Project Tab to once again open your Script in **MonoDevelop**. Now you need to fill out the *else* block that will be both X and Y rotation. Start by copying and pasting all your code from the *else if* block you just did. This takes care of the vertical rotation.

10. For horizontal rotation, add the following two statements after the `_rotationX` assignment statements but before the `transform.localEulerAngles` statement:

```
float delta = Input.GetAxis("Mouse X") * sensitivityHorz;
float rotationY = transform.localEulerAngles.y + delta;
```

The first statement is the exact same thing we fed to `transform.Rotate()` when you did the horizontal rotation in Tasks Part 4a. The second statement takes the current rotation angle (given by `transform.localEulerAngles.y`) and adds the delta to it. The reason we don't use `transform.Rotate()` here is that we want to combine both the horizontal and vertical rotation and we already know, from Task 4b, that vertical can't use Rotate(). To finish this off, you only need to replace the 0 with `rotationY` in your `transform.localEulerAngles` statement:

```
transform.localEulerAngles = new Vector3 (_rotationX, rotationY, 0);
```

11. Save your script and close **MonoDevelop**. Click on the Player object in the Hierarchy Tab.

In the Mouse Look Component of the Inspector Tab, change the Axes dropdown to *Mouse X and Y*. Now play your Scene and move the mouse!!!

Tasks Part 5 – Making the Player Move:

1. At this point your Player can look around but still can't move!! We need to add that capability!! Create a new C# script and name it FPSInput and make it a component of the Player object (you should know how to both these steps now!!! If not, refer to Task 4 Steps 1 to 3).

2. Declare and initialize the following public variable:

```
public float speed = 9.0f;
```

   Add the following three statements to `Update()`:

```
float deltaX = Input.GetAxis("Horizontal") * speed;
float deltaZ = Input.GetAxis("Vertical") * speed;
transform.Translate (deltaX * Time.deltaTime, 0,
            deltaZ * Time.deltaTime);
```

   We use `Input.GetAxis()` again except now we use "Horizontal" and "Vertical" as arguments. Horizontal is the side to side movement. It is mapped to the left and right arrow keys and the letters A and D (this can be customized). Vertical is the forward and backward movement and thus the Z axis. It is mapped to the up and down arrow keys and the letters W and A.

   `transform.Translate()` is the change in coordinates to physically move the object (as opposed to Rotate() which was just rotating the object). The Y axis is set to 0 since, in this case, we don't want the Player leaving the ground.

3. Save your script and close **MonoDevelop**. Click on the Player object in the Hierarchy Tab. In the Mouse Look Component of the Inspector Tab, change the Axes dropdown to *Mouse X* only (no vertical rotation or your Player will be able to fly). Now play your Scene and wander around (You may notice a problem, don't worry, it will be handled in Lab 4).

Submission:

Demo to the Lab Instructor:

- **show your *unitypackage* from Task 1 in File Explorer to the Instructor**
- **that a capsule, representing a player, has been placed in your Scene**
- **change the Axes dropdown to Mouse X and Y and show that the Player's view changes in both the X and Y directions (Task 4c)**
- **change the Axes dropdown to Mouse X only and show that the Player can move around the Scene (Task 5)**

- A successful demo scores you 10 marks.