Unity Game Engine

Introduction to Unity – Build a Toy FPS Game

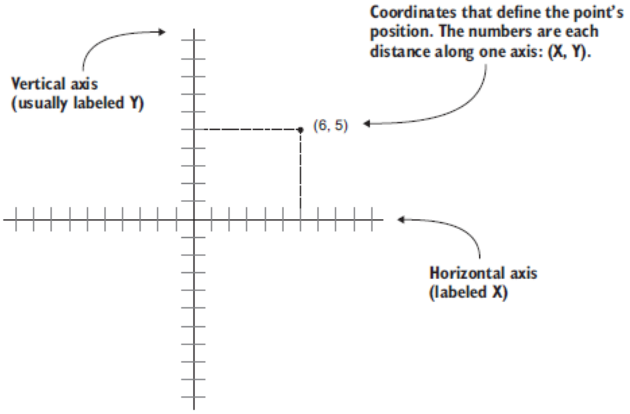
Start with a Toy Game

- Game design is a huge topic unto itself, with many impressively large books focused on how to design a game.
- For now, let's build a simple toy game- basic FPS (first-person shooter)
  - Basic scene with a room to navigate around.
  - The player will see the world from their player character's point of view (First Person View).
  - The player can control the character using the mouse and keyboard.

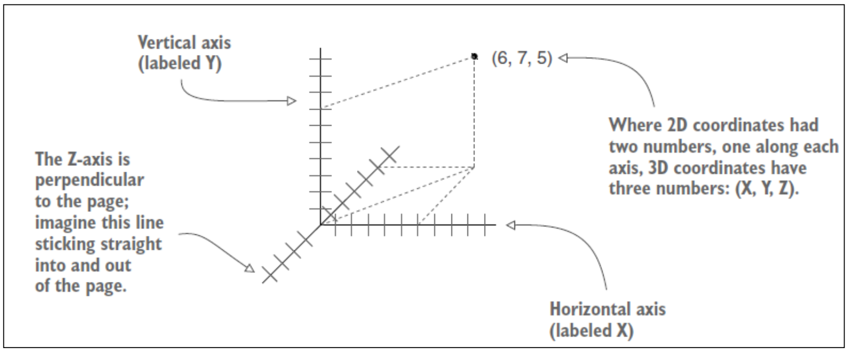
Create A basic FPS (first-person shooter)

- Steps:
  1. Set up the room: create the floor, outer walls, and inner walls.
  2. Place the lights and camera.
  3. Create the player object (including attaching the camera on top).
  4. Write movement scripts: rotate with the mouse and move with the keyboard.

2D Coordinate Space


Coordinates that define the point's position. The numbers are each distance along one axis: (X, Y).
Vertical axis (usually labeled Y)
(6, 5)
Horizontal axis (labeled X)

3D Coordinate Space



Vertical axis (labeled Y)

(6, 7, 5)

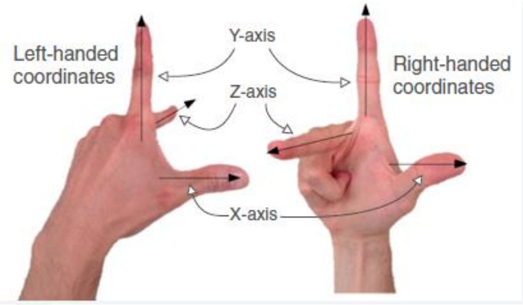
Where 2D coordinates had two numbers, one along each axis, 3D coordinates have three numbers: (X, Y, Z).

The Z-axis is perpendicular to the page; imagine this line sticking straight into and out of the page.

Horizontal axis (labeled X)

3D Coordinate Space

- Unity uses a left-handed coordinate system, as do many 3D art applications. Many other tools use right-handed coordinate systems (OpenGL, for example).

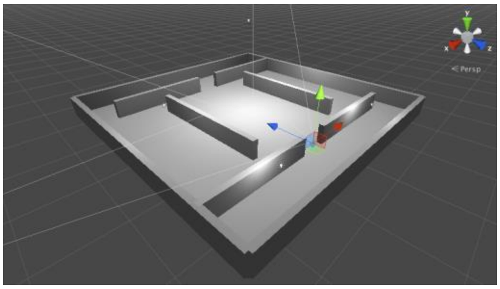


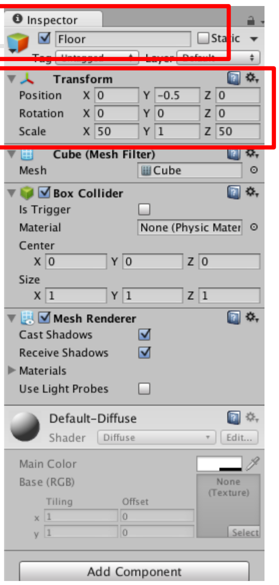
Create A basic FPS (first-person shooter)

- Steps:
  1. Set up the room: create the floor, outer walls, and inner walls.
  2. Place the lights and camera.
  3. Create the player object (including attaching the camera on top).
  4. Write movement scripts: rotate with the mouse and move with the keyboard.

The scene: floor and walls

- GameObject > Cube
- Change its name
- Adjust the position, orientation, and scale





Inspector

☑ Floor ☐ Static

Tag Untagged Layer Default

Transform

Position X 0 Y -0.5 Z 0

Rotation X 0 Y 0 Z 0

Scale X 50 Y 1 Z 50

Cube (Mesh Filter)

Mesh Cube

☑ Box Collider

Is Trigger ☐

Material None (Physic Mater

Center X 0 Y 0 Z 0

Size X 1 Y 1 Z 1

☑ Mesh Renderer

Cast Shadows ☑

Receive Shadows ☑

Materials

Use Light Probes ☐

Default-Diffuse

Shader Diffuse Edit...

Main Color

Base (RGB) None (Texture)

Tiling Offset

x 1 0

y 1 0

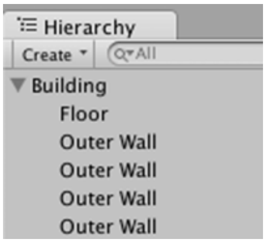
Select

Add Component

Scene Organization

- *Use scene navigations to inspect the scene.*
- *If you ever get lost in the scene, press F to reset the view on the currently selected object.*

- Create <u>empty game objects</u> to organize the scene. Drag objects on top of each other in the Hierarchy view to establish *parent/children* relationship.



Hierarchy

Create ▾ Q▾All

▼ Building
    Floor
    Outer Wall
    Outer Wall
    Outer Wall
    Outer Wall

GameObject

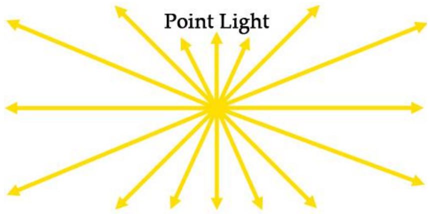
- GameObject is the base class for all entities in Unity scenes.
- All scene objects (empty object, Floor, Camera, etc) are instances of the class GameObject.
- We attach a behavior(s) to a GameObject.
  - Behavior is described in a script (MonoBehaviour).
  - Behavior of the object in the scene depends on what components (eg script) have been added to that GameObject.
  - Cube objects have a Cube component, Sphere objects have a Sphere component, and so on.

Create A basic FPS (first-person shooter)

- Steps:
  1. Set up the room: create the floor, outer walls, and inner walls.
  2. Place the lights and camera.
  3. Create the player object (including attaching the camera on top).
  4. Write movement scripts: rotate with the mouse and move with the keyboard.

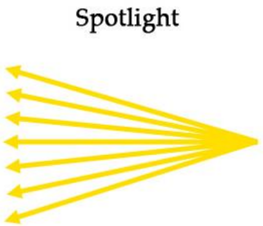
Lights

- Three types of lights: **point**, **spot**, and **directional**.
- ***Point*** *lights* are a kind of light source where all the light rays originate from a single point and project out in all directions, like a lightbulb in the real world.



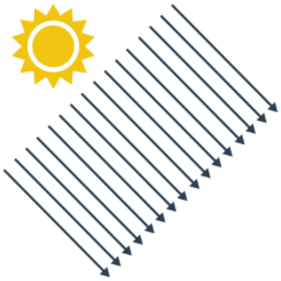
Lights

- ***Spot** lights* are a kind of light source where all the light rays originate from a single point but only project out in a limited cone.



Lights

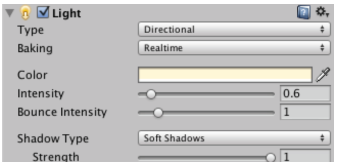
- ***Directional** lights* are a kind of light source where all the light rays are parallel and project evenly, lighting everything in the scene the same way. This is like the sun in the real world.



Directional/Point Light

- The scene probably already has one Directional Light by default.
- Add more lights by choosing GameObject > Light > Directional/Point Light.
  - The position of a directional light doesn't affect the scene.
  - The orientation of a directional light matters.

- Place a directional light high above the room to illuminate the entire room.
- Create a point light and move it around to illuminate a certain portion of the room.
- Select a light and manipulate the orientation and intensity of the light in Inspector view.



☑ Light

Type: Directional

Baking: Realtime

Color

Intensity: 0.6

Bounce Intensity: 1

Shadow Type: Soft Shadows

Strength: 1

Camera

- Let's use the default camera. (Main Camera in Hierarchy View).
- If you want to create a camera, create it through GameObject > Camera.
- For FPS, the camera will be positioned around the top of the player, (0, 0.5, 0), so that the view appears to be the player's eyes.

Create A basic FPS (first-person shooter)

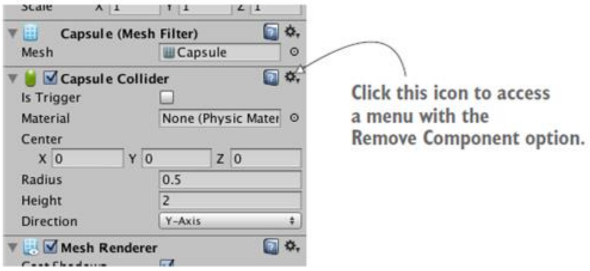
- Steps:
  1. Set up the room: create the floor, outer walls, and inner walls.
  2. Place the lights and camera.
  3. Create the player object (including attaching the camera on top).
  4. Write movement scripts: rotate with the mouse and move with the keyboard.

Create Player Object

- GameObject > Capsule.
- Position this object inside the room while not touching any walls.
- Name the object “Player”.

Modify Player Object

- **Step1:** This object has a capsule collider assigned to it. *(Remove the capsule collider by clicking the gear icon toward the top-right of that component.)*



Scale X 1 Y 1 Z 1

Capsule (Mesh Filter)

Mesh: Capsule

☑ Capsule Collider

Is Trigger: □

Material: None (Physic Mater

Center: X 0 Y 0 Z 0

Radius: 0.5

Height: 2

Direction: Y-Axis

☑ Mesh Renderer

Click this icon to access
a menu with the
Remove Component option.

- **Step2:** Assign a character controller to this object.
  - Inspector > Add Component > Physics > Character Controller

Modify Player Object (2)

- **Step3:** To make it a FPS, attach the camera to the player
  - Drag the camera object onto the player capsule in the Hierarchy view.
  - Position the camera so that it'll look like the player's eyes (e.g. a position of 0, 0.5, 0).
  - If necessary, reset the camera's rotation to 0, 0, 0.

Create A basic FPS (first-person shooter)

- Steps:
  1. Set up the room: create the floor, outer walls, and inner walls.
  2. Place the lights and camera.
  3. Create the player object (including attaching the camera on top).
  4. Write movement scripts: rotate with the mouse and move with the keyboard.

Make the player move

- Write movement scripts attached to the player.
- Eventually, those scripts will respond to keyboard and mouse input,
- Script components have an Update() method that runs every frame.
  - E.g To spin the cube, add code inside Update() that rotates the cube a small amount.

Spin.cs

- Create a C# script named Spin.

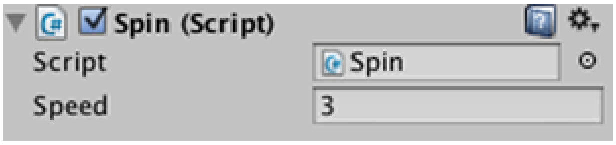
```
using UnityEngine;
using System.Collections;
public class Spin : MonoBehaviour {
  public float speed = 3.0f;
  void Update() {
    transform.Rotate(0, speed, 0);
  }
}
```

Attach Spin.cs to the Player

- Add the script component to the player object (by dragging the script up from the Project view and drop it onto Player in the Hierarchy view.)
- Hit Play and you'll see the view spin around.

Spin.cs Explained

- `public float speed = 3.0f;` declares a variable for speed.
- `public` variables are exposed in the Inspector and be modified
  - In the Inspector, you can type in a new number, and then the script will use that value instead of the default value defined in the code.



☑ Spin (Script)

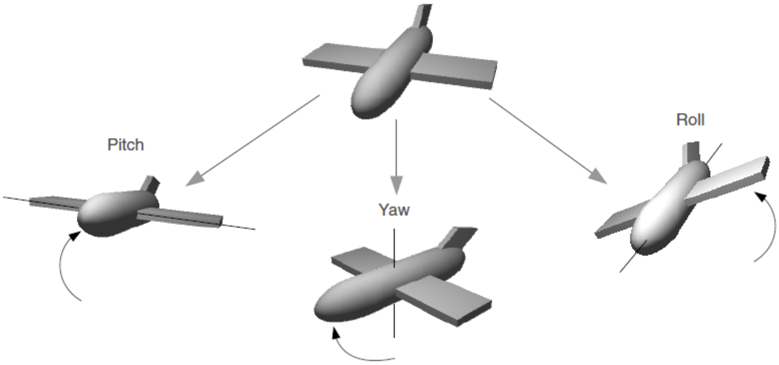
Script Spin

Speed 3

Spin.cs Explained

- `transform` is the Transform (component) attached to this GameObject.
- `transform.Rotate(0, speed, 0);`
  - `Rotate()` is a method of the Transform class.
  - The transform is rotated by speed degrees every frame . It’s for **incrementing the rotation** without limit.
  - By default, the Rotate() method operates on what are called **local coordinates.**
  - Three parameters for Rotate() are X, Y, and Z rotation.

Rotate



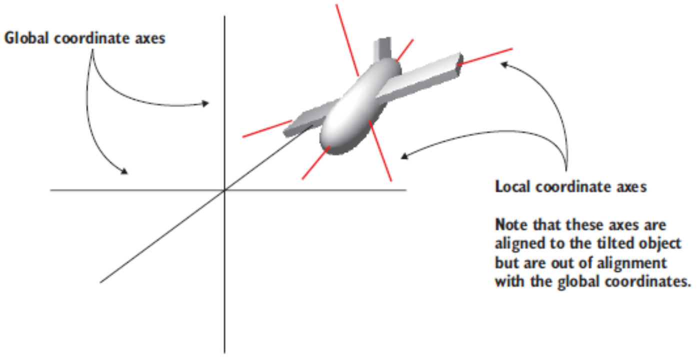
Transform.Rotate

public void **Rotate**(float **xAngle**, float **yAngle**, float **zAngle**, Space **relativeTo** = Space.Self);

- xAngle - Degrees to rotate around the X axis.
- yAngle- Degrees to rotate around the Y axis.
- zAngle- Degrees to rotate around the Z axis.
- relativeTo - Rotation is local to object or World.

- Applies a rotation of zAngle degrees around the z axis, xAngle degrees around the x axis, and yAngle degrees around the y axis (in that order).

Local vs. global coordinate space



Note that these axes are aligned to the tilted object but are out of alignment with the global coordinates.

Local vs. global coordinate space

- By default, the Rotate() method operates on what are called local coordinates.
- Tell the method whether to use local or global coordinates using an optional fourth parameter by writing either Space.Self (default) or Space.World :

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

Space.Self - the rotation is applied around the transform's local axes
Space.World - the rotation is applied around the world x, y, z axes.

Player Control

- Mouse for rotation
- Keyboard for translation

Input from Mouse

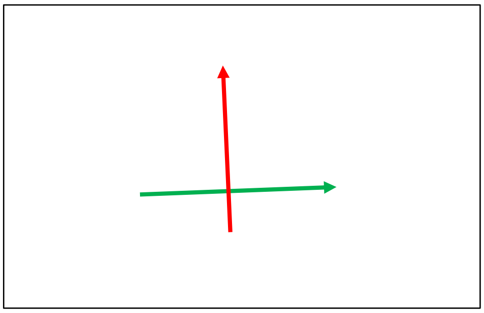
- Let's make player rotation respond to input from the mouse.

- Consider the player's rotation with three different types of rotation behavior (horizontal, vertical, and both).

- Create a new C# script named MouseLook.cs.
- Remove the Spin component and attach this new script to the player object instead.

MouseLook.cs Script

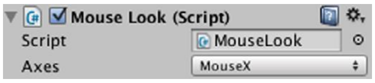
- Make the player (and camera) rotate according to mouse moves.



Horizontal mouse moves for rotation about object's y-axis

Vertical mouse moves for rotation about object's x-axis

```
using UnityEngine;
using System.Collections;
public class MouseLook : MonoBehaviour {
  public enum RotationAxes {
    MouseXAndY = 0, MouseX = 1, MouseY = 2
  } // for the user's choice
  public RotationAxes axes = RotationAxes.MouseXAndY;
  void Update() {
    if (axes == RotationAxes.MouseX) {
      // horizontal rotation here
    }
    else if (axes == RotationAxes.MouseY) {
      // vertical rotation here
    }
    else {
      // both horizontal and vertical rotation here
    }
  }
}
```



☑ Mouse Look (Script)

Script: MouseLook

Axes: MouseX

A public variable typed to that enum will display in the Inspector as a drop-down menu.

```
...
  public float sensitivityHor = 9.0f; // rotation speed
  void Update() {
  if (axes == RotationAxes.MouseX) {
    // horizontal rotation here
    transform.Rotate(0,
          Input.GetAxis("Mouse X") * sensitivityHor,
          0);
    }

...
```

Input.GetAxis().
- **Input** class has a bunch of methods for handling input devices (such as the mouse)
- **GetAxis("Mouse X")** returns positive or negative numbers correlated to the the delta of mouse movement for the horizontal axis.

*** sensitivityHor** makes the speed scale according to mouse movement, scaling down to zero or even reversing direction.

```
...
  public float sensitivityVert = 9.0f;
  public float minimumVert = -45.0f;
  public float maximumVert = 45.0f;
  private float _rotationX = 0;
  void Update() {
    ...
    else if (axes == RotationAxes.MouseY) {
      _rotationX -= Input.GetAxis("Mouse Y")* sensitivityVert;
      _rotationX = Mathf.Clamp(_rotationX,
                         minimumVert, maximumVert);
      float rotationY = transform.localEulerAngles.y;
      transform.localEulerAngles =
         new Vector3(_rotationX, rotationY, 0);
     }
...
```

Provides vertical rotation with limits.
Instead of using Rotate(), this code sets the rotation angle directly.
Mathf.Clamp() to keep the rotation angle between minimum and maximum limits.
localEulerAngles() sets the angles to absolute values.

```
...
  else { // MouseXAndY
    _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
    _rotationX = Mathf.Clamp(_rotationX, minimumVert,
                 maximumVert);
    float delta = Input.GetAxis("Mouse X") * sensitivityHor;
    float rotationY = transform.localEulerAngles.y + delta;
    transform.localEulerAngles =
          new Vector3(_rotationX, rotationY, 0);
  }
...
```

Provides horizontal and vertical rotations.
Instead of using Rotate(), the code is using the delta and rotationY to add small horizontal change to the current orientation.

Disable Phys Rotation

- Add the following line to the script, MouseLook.cs, (before `Update()`) in order to solely control the player by the mouse and not affected by the physics simulation.

```
void Start() {
  Rigidbody body =
     GetComponent<Rigidbody>();
  if (body != null)
    body.freezeRotation = true;
}
```

Keyboard Input

- Let's make player translation respond to input from the keyboard.
- In Unity, both the left/right arrow keys and the letters A/D are mapped to Horizontal, whereas both the up/down arrow keys and the letters W/S are mapped to Vertical.

- Create a new C# script named FPSInput.cs.

FPSInput Script

```
using UnityEngine;
using System.Collections;
public class FPSInput : MonoBehaviour {
  public float speed = 6.0f;
  void Update() {
    transform.Translate(0, speed, 0);
  }
}
```

The code moves the character along its y-axis.

In general, the game moves the player on xz plane.

```
...
  void Update() {
    float deltaX = Input.GetAxis("Horizontal") * speed;
    float deltaZ = Input.GetAxis("Vertical") * speed;
    transform.Translate(deltaX, 0, deltaZ);
 }

...
```

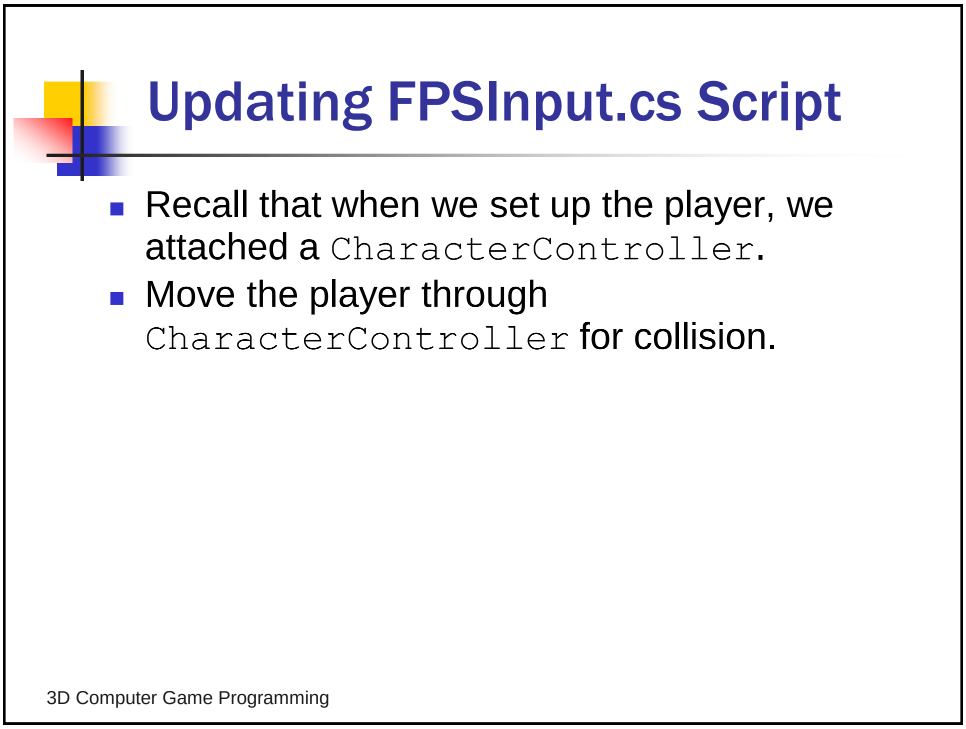
GetAxis() – "Horizontal" or "Vertical" are abstractions for input settings in Unity. GetAxis("Horizontal") returns a value in the range -1..1 when left/right/A/D keys is pressed.

Check the Edit menu under **`Project Settings > Input`** to see a list of abstract input names and the exact controls mapped to those names.

This move is frame rate dependent. For example, if the rate of movement would be 180units/second at 30fps, the rate of movement would be 360units/second at 60fps.

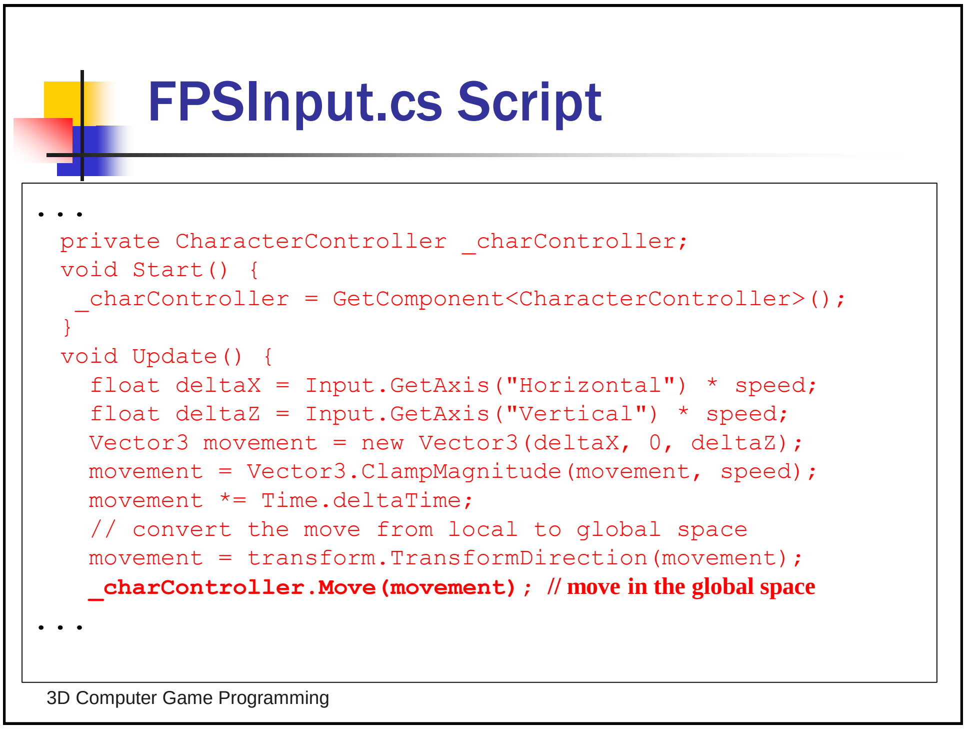
```
...
  void Update() {
    float deltaX = Input.GetAxis("Horizontal") * speed;
    float deltaZ = Input.GetAxis("Vertical") * speed;
    transform.Translate(deltaX * Time.deltaTime,
       0, deltaZ * Time.deltaTime);
 }

...
```

To make the movement speed same on all computers, use `Time.deltaTime`.
`deltaTime` is the amount of time between frames (for example, 30 fps is a deltaTime of 1/30th of a second)
So the code will move the player in a different amount per frame depending on the frame rate resulting in the same amount of move in a second.



Updating FPSInput.cs Script

- Recall that when we set up the player, we attached a `CharacterController`.
- Move the player through `CharacterController` for collision.



FPSInput.cs Script

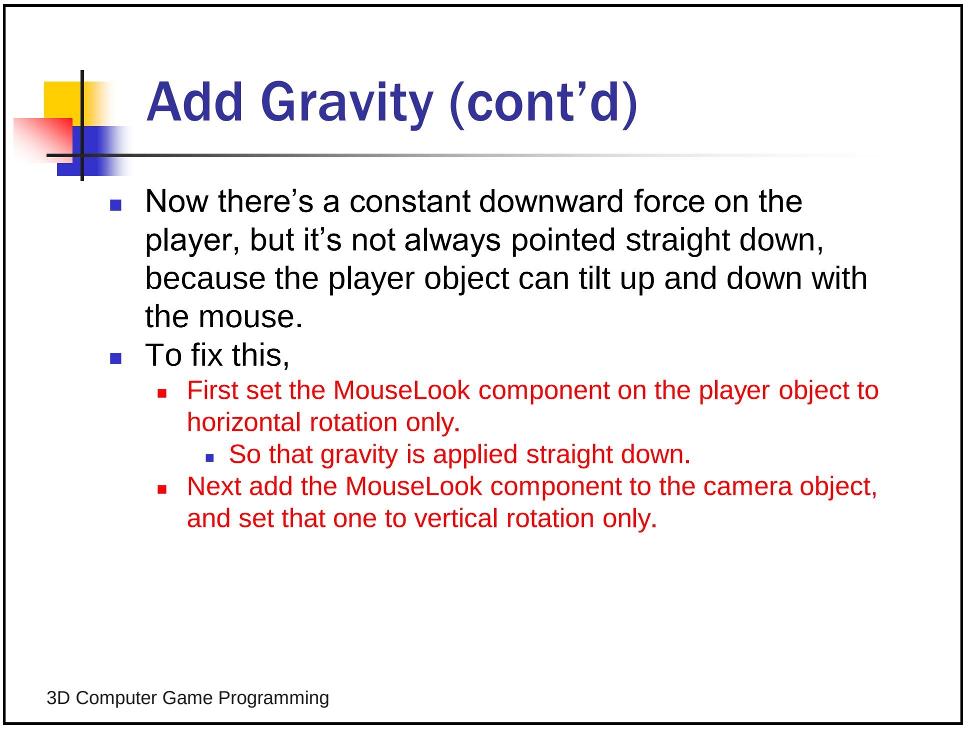
```
. . .
  private CharacterController _charController;
  void Start() {
   _charController = GetComponent<CharacterController>();
  }
  void Update() {
    float deltaX = Input.GetAxis("Horizontal") * speed;
    float deltaZ = Input.GetAxis("Vertical") * speed;
    Vector3 movement = new Vector3(deltaX, 0, deltaZ);
    movement = Vector3.ClampMagnitude(movement, speed);
    movement *= Time.deltaTime;
    // convert the move from local to global space
    movement = transform.TransformDirection(movement);
    _charController.Move(movement); // move in the global space
. . .
```

FPSInput.cs Explained

- Once you have a reference to the characterController, call Move() method on the controller.
  - Pass in a vector to that method
  - Vector3.ClampMagnitude() to limit the vector's magnitude to the movement speed
- We need to pass a movement vector defined in global space to the Move() method. TransformDirection() converts vector from local to global space.
- https://docs.unity3d.com/ScriptReference/CharacterController.Move.html

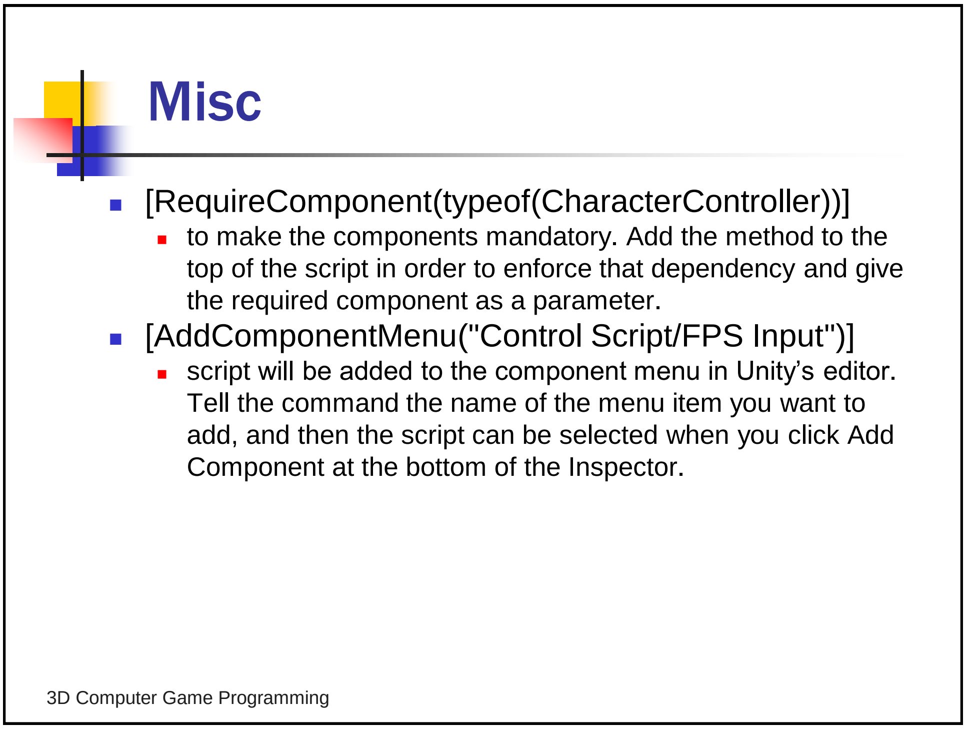
Add Gravity

```
public float gravity = -9.8f;
...
void Update() {
  ...
  movement = Vector3.ClampMagnitude(movement,
                speed);
  movement.y = gravity;

  movement *= Time.deltaTime;
  movement =
      Transform.TransformDirection(movement);
  _charController.Move(movement);
```



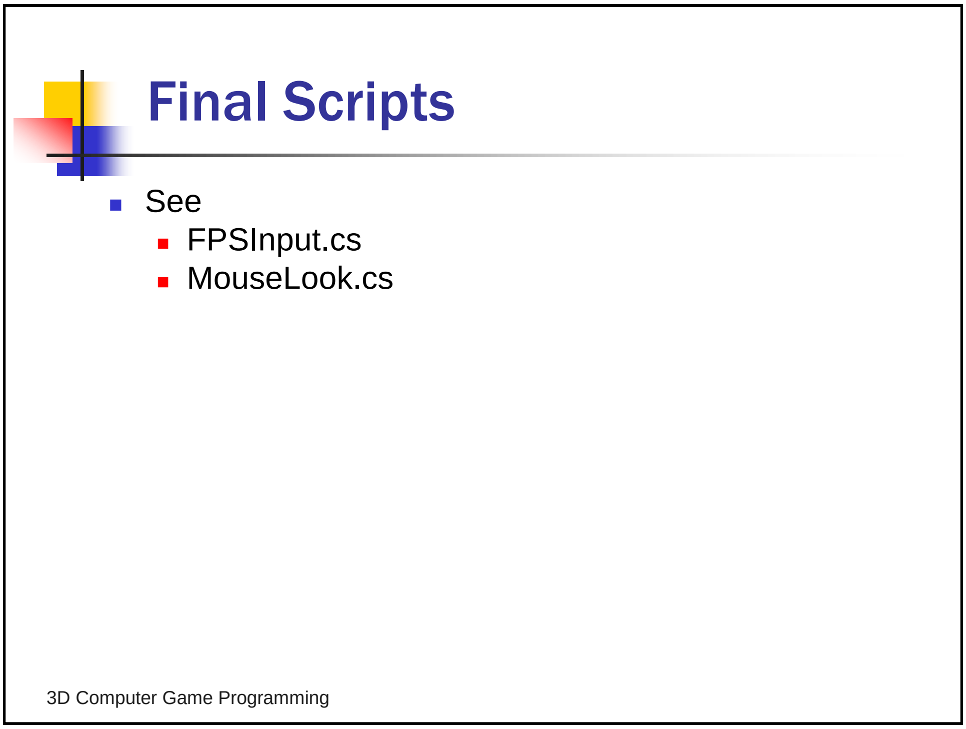
Add Gravity (cont’d)

- Now there’s a constant downward force on the player, but it’s not always pointed straight down, because the player object can tilt up and down with the mouse.
- To fix this,
  - First set the MouseLook component on the player object to horizontal rotation only.
    - So that gravity is applied straight down.
  - Next add the MouseLook component to the camera object, and set that one to vertical rotation only.



Misc

- [RequireComponent(typeof(CharacterController))]
  - to make the components mandatory. Add the method to the top of the script in order to enforce that dependency and give the required component as a parameter.
- [AddComponentMenu("Control Script/FPS Input")]
  - script will be added to the component menu in Unity’s editor. Tell the command the name of the menu item you want to add, and then the script can be selected when you click Add Component at the bottom of the Inspector.



Final Scripts

- See
  - FPSInput.cs
  - MouseLook.cs