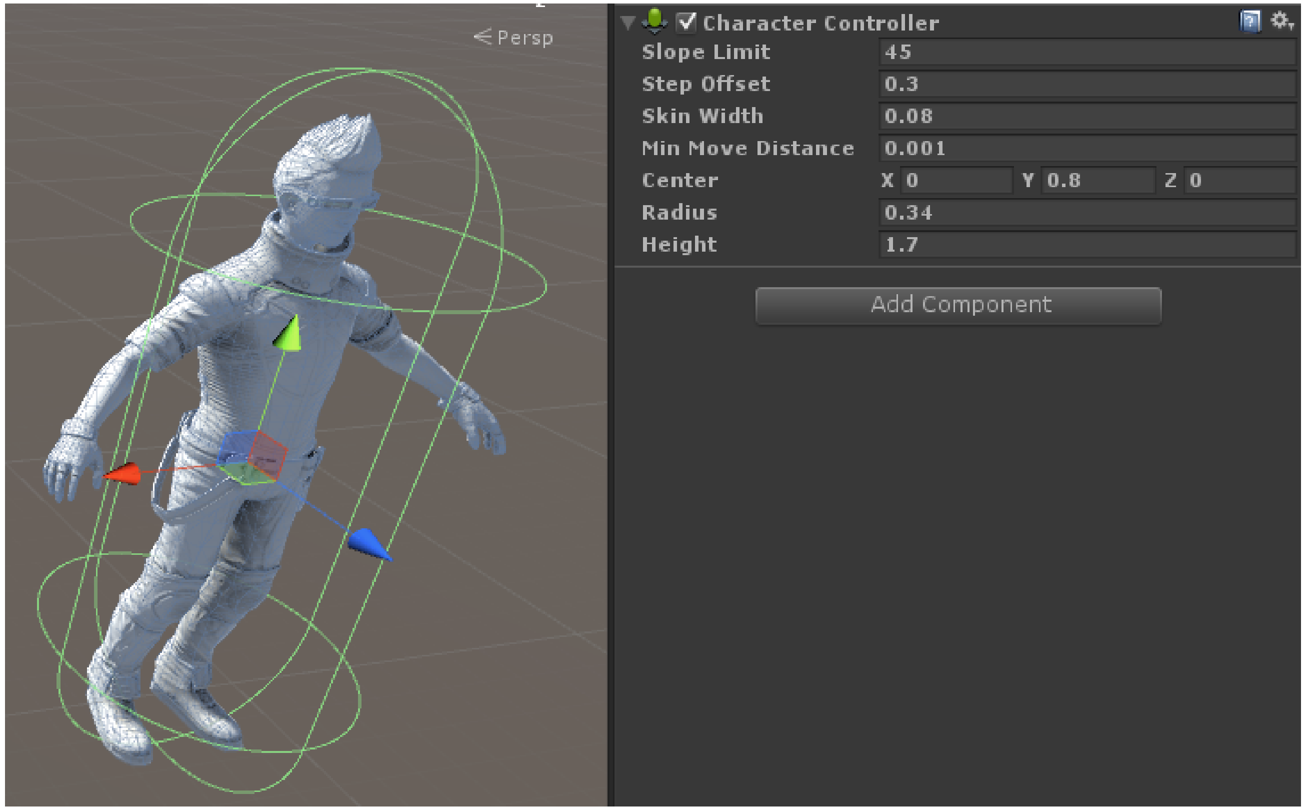
Character Controllers Taking control of the situation

Contents

What is a Character Controller?

A Character Controller is a physics component that is used when you want to control a character with physics that isn't perfectly accurate, i.e. instant stopping and starting movement, double-jumps, etc. Basic physics are still used, but your character won't tip over (unless you program it to).

This is often used for a 3rd or 1st person player control.



Character Controller

Slope Limit	45
Step Offset	0.3
Skin Width	0.08
Min Move Distance	0.001
Center	X 0 Y 0.8 Z 0
Radius	0.34
Height	1.7

Creating the Character Controller

For this example, we are going to create a character controller that does not contain any assets. Depending on what we want we can add a character, or even a camera for 1st person, later

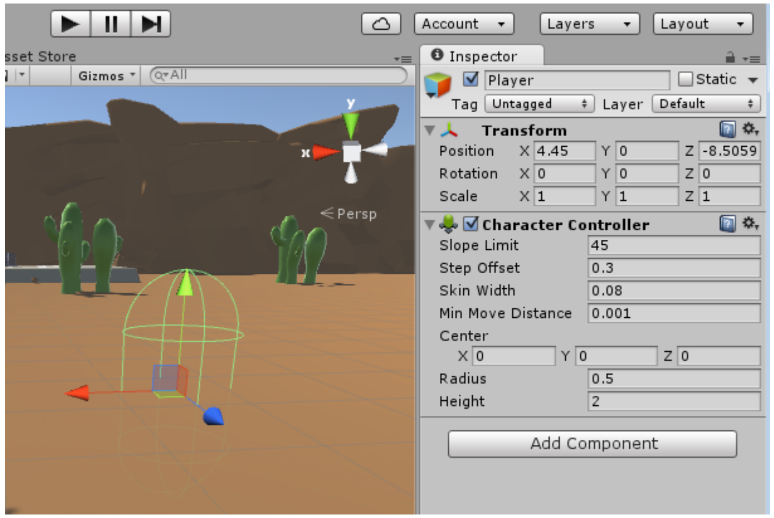
1. Create an **empty Object**
2. Rename it to something that identifies it
    a. In my case Player
3. Add a **Character Controller Component**
    a. Add Component -> Physics -> Character Controller
    Note: Not Physics 2D

You will also notice that if you position it such that it is embedded in the ground, it will fall through when you press play. This can be fixed in a couple of ways:

4. Position the character above the ground
    a. This means that you need to take that into account when placing player spawners.
5. Change the position of the Character Controllers centre so that it is above the pivot point
    b. Making it slightly easier to position the character in the world

If you want to see the position of the Player, create a Capsule and make it a child of the Player object. Make sure it is positioned properly.

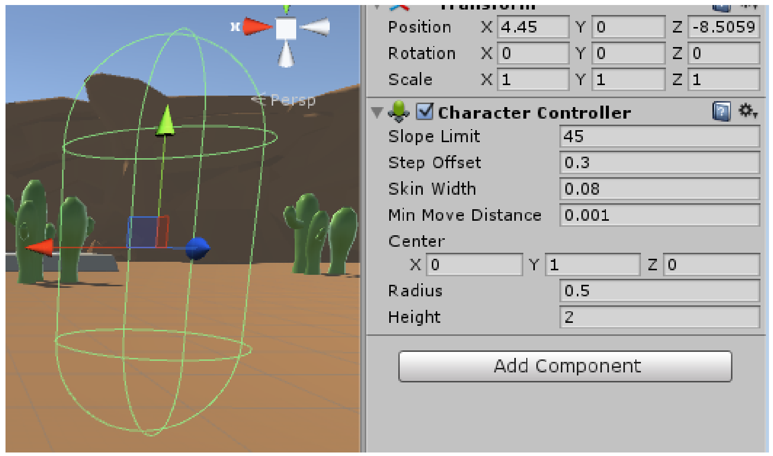
6. Optional: Add a capsule, parent and position for testing purposes



Inspector
Player — Static
Tag: Untagged — Layer: Default

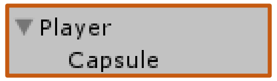
Transform
Position X 4.45 Y 0 Z -8.5059
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

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



Position X 4.45 Y 0 Z -8.5059
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Character Controller
Slope Limit: 45
Step Offset: 0.3
Skin Width: 0.08
Min Move Distance: 0.001
Center: X 0 Y 1 Z 0
Radius: 0.5
Height: 2



Player
    Capsule

Character Controllers Taking control of the situation

Coding the Character Controller

Now we have the character controller, we can make a script to control it, much like we did with the Rigidbody in the last game

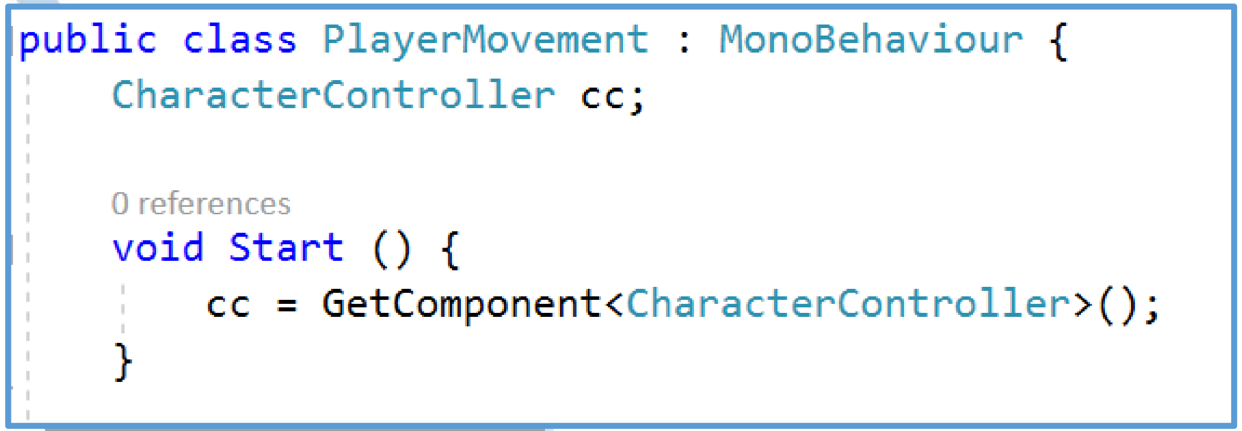
1. Create a **new script** called **PlayerMovement**
    a. Attach it to the player

- Inside the class block

2. Make a **variable** for the **CharacterController**, we called cc,

- In the Start function

3. and assign the component to it.



```
public class PlayerMovement : MonoBehaviour {
    CharacterController cc;

    0 references
    void Start () {
        cc = GetComponent<CharacterController>();
    }
```

Basic Movement

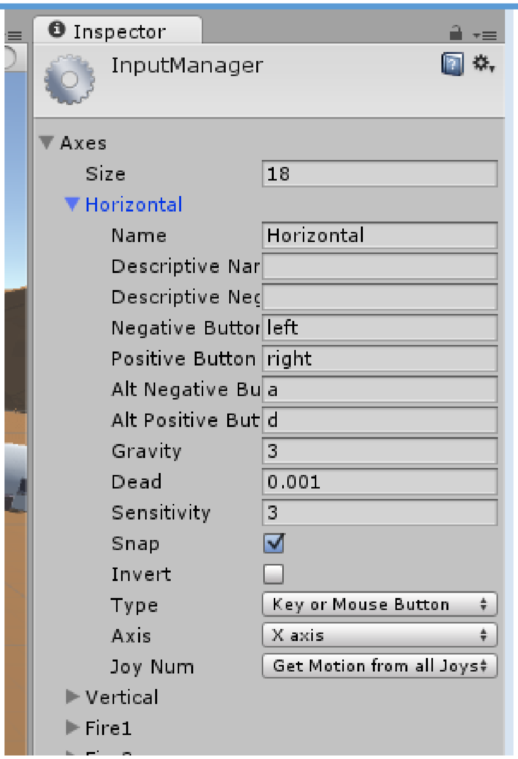
To move our character we need to get some input commands from us. For this we can use the Input commands that you can view in the inspector by going:

Edit -> Project Settings -> Input

From here you can see the types of inputs and what keys are required to use them.

- The input gives a value from 1 to -1
- For key input this is either 1 or -1
- For a controller it gives a value between 1 and -1 and the option of a dead zone to compensate for worn-out thumb-sticks

With that in mind we should be able to code in some movement using the information from the input manager.



Size	18
Name	Horizontal
Descriptive Nar	
Descriptive Neg	
Negative Buttor	left
Positive Button	right
Alt Negative Bu	a
Alt Positive But	d
Gravity	3
Dead	0.001
Sensitivity	3
Snap	✓
Invert	
Type	Key or Mouse Button
Axis	X axis
Joy Num	Get Motion from all Joys

Setting Up our Functions

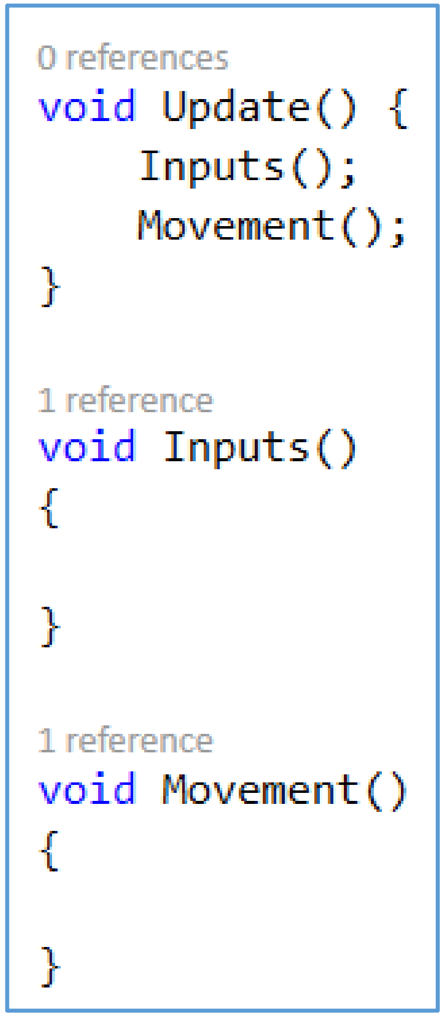
When coding up our character, we want to split the functionality into easy to manage parts. Functions are good for this as it allows us to sort and manage different tasks.

In our character code we want to include

- An Inputs function to deal with all the inputs our character will have
- A Movement Function that deals with our characters Movements

Later we can have different functions for Attacking, shooting, pickups and other tasks.

1. **Create** two void functions
   a. Inputs and Movement
2. Call the functions in the **Update Function**



```
0 references
void Update() {
    Inputs();
    Movement();
}

1 reference
void Inputs()
{

}

1 reference
void Movement()
{

}
```

Setting the Inputs

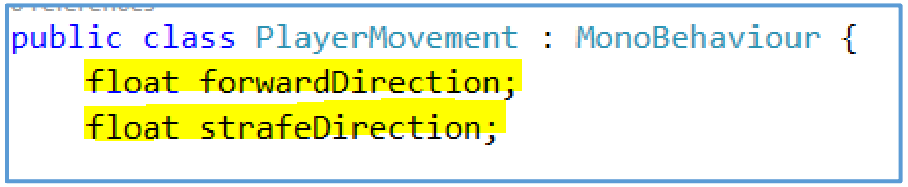
For our character to move we need to some inputs and store them for use.

In this example we are going to have the player move forwards and backwards with the Vertical direction (W,S, Up Down and forwards and backwards on the joystick) as well as strafeing left and right (not turning) with the Horizontal Direction (A,D, Left, Right and side directions on the joystick)

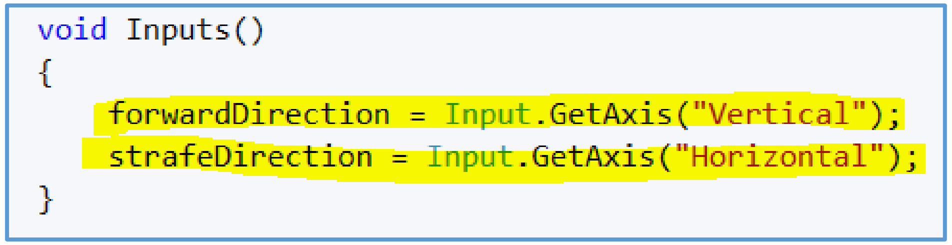
- In The Class Block
  1. Create a **float** called **forwardDirection**
  2. Create a **float** called **strafeDirection**

- In the Inputs() functions
  1. Set **forwardDirection** to equal **Inputs GetAxis** for **"Vertical"**
  2. Set **strafeDirection** to equal **Inputs GetAxis** for **"Horizontal"**

Note the Capital letters



```
public class PlayerMovement : MonoBehaviour {
    float forwardDirection;
    float strafeDirection;
```



```
void Inputs()
{
    forwardDirection = Input.GetAxis("Vertical");
    strafeDirection = Input.GetAxis("Horizontal");
}
```

SimpleMove

SimpleMove is used to make basic movement. It has simple gravity and all it asks for is a Vector3 that tracks how fast it should move. We can use this like we used Rigidbody for movement, SimpleMove's gravity systems however, are simpler.

We can't just pass this into the SimpleMove, we need to pass in a Vector3 stating the direction we wish to move.

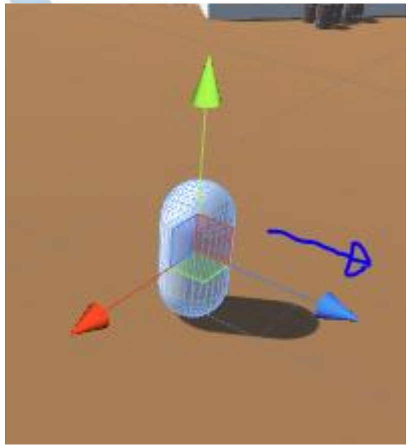
3. Create a **Vector3** called **direction**

We need to identify what the forward and side Axis will be for your character.

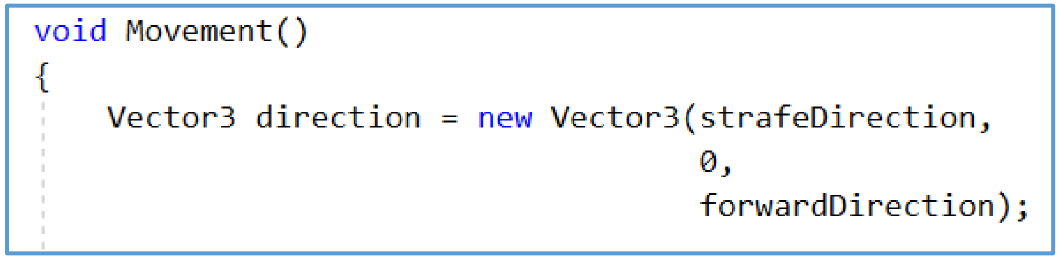
4. Set the **vector3** so that x, and y are 0 and z is equal to the **forwardSpeed**.
   - **a.** **x axis** is strafeDirection
   - **b.** **Y axis** to 0
   - **c.** **z axis** is forwardDirection

Now we have to normalise our Vector so that the direction we are moving in is equal to one. This helps stop speed up error's latter.

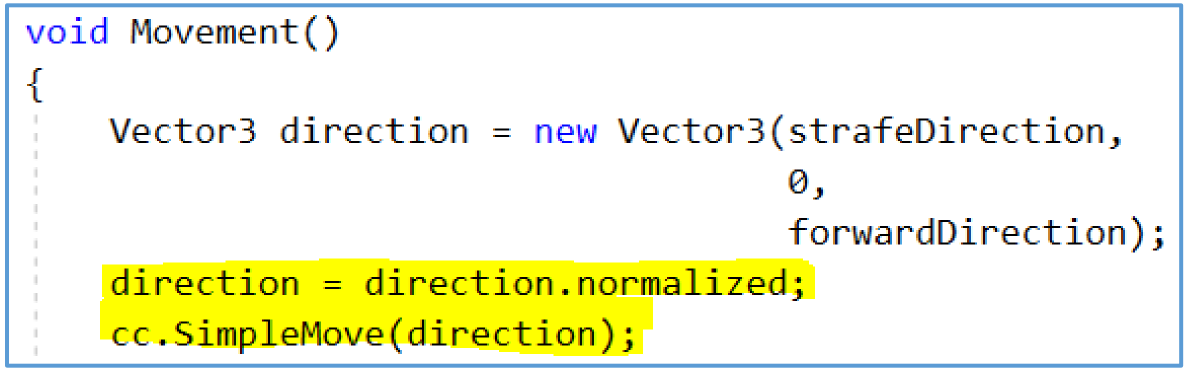
5. Set the direction to be to be a normalised direction
6. Get your **CharacterController** (cc) and call the function SimpleMove.
   - a. Pass in **speed**.



THE X AXIS IS IN RED AND THE Y AXIS IS BLUE



```
void Movement()
{
    Vector3 direction = new Vector3(strafeDirection,
                                    0,
                                    forwardDirection);
```



```
void Movement()
{
    Vector3 direction = new Vector3(strafeDirection,
                                    0,
                                    forwardDirection);
    direction = direction.normalized;
    cc.SimpleMove(direction);
```

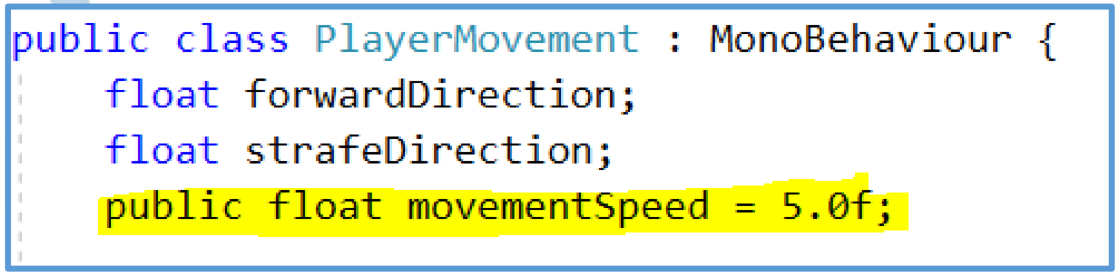
Now **test the game.**

The character is moving, hopefully in the right directions. If not, look at which axis it is actually moving on and adjust.

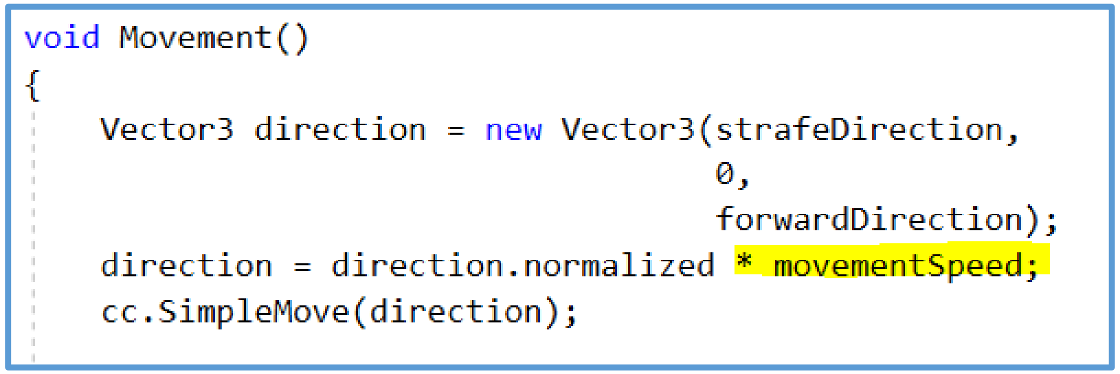
You will also note that it is going really slowly. This is because of the input value being between 1 and -1. We need to increase that to make it work better

1. Create a **public float variable**, in the class block, called **movementSpeed**
    a. make it equal five
- In the Movement() function
2. change the value of **direction** by multiplying the normalised direction by **movementSpeed**

Now when we test the game the character is going faster. You can adjust the movementSpeed value to a desired speed.



```
public class PlayerMovement : MonoBehaviour {
    float forwardDirection;
    float strafeDirection;
    public float movementSpeed = 5.0f;
```



```
void Movement()
{
    Vector3 direction = new Vector3(strafeDirection,
                                    0,
                                    forwardDirection);
    direction = direction.normalized * movementSpeed;
    cc.SimpleMove(direction);
```

Move with Gravity

Simple move is great. It allows us to input minimal instructions, our speed and direction, to give us movement with some gravity and wall collisions.

However, if you want to jump and be able to change our upwards movement SimpleMove is not good as it ignores the y coordinates. This means you cannot fly or jump.

Switching from SimpleMove to Move will help fix this. However, it expects some different inputs to make it work.

1. Replace **SimpleMove** with **Move**

```
void Movement()
{
    Vector3 direction = new Vector3(strafeDirection,
                                    0,
                                    forwardDirection);
    direction = direction.normalized * movementSpeed;
    //cc.SimpleMove(direction); -- REPLACE
    cc.Move(direction);
}
```

Movement per Frame

While SimpleMove expects a speed and direction, Move wants to know how many units to move per frame (each time update is called).

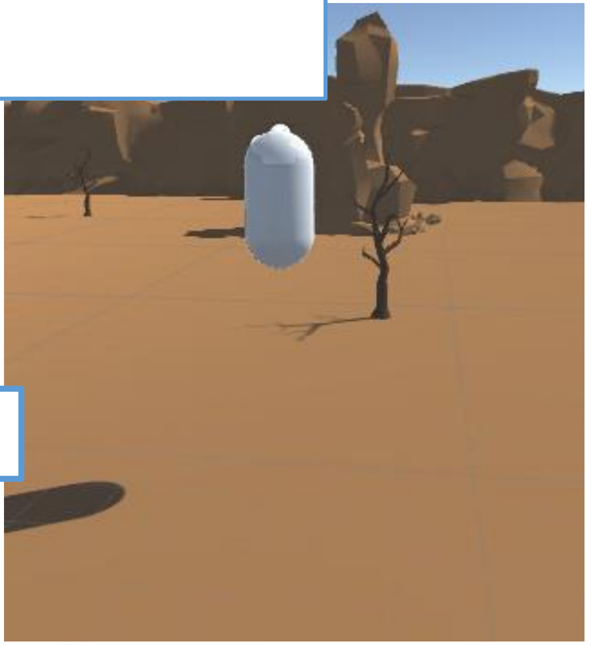
This is easy to fix.

- In the **Move** function call
  2. Multiply **direction** by **Time.DeltaTime.**

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

This should not change much in our game at the moment. But if you lifted your character up above the terrain, you would notice that **it does not fall.**

So we now have to pass in some gravity.


LOOK MA! I'M FLYIN'!

INSERTING BASIC GRAVITY

Our game has a default gravity set that can be used. You can find this in the Physics Manager:

Edit-> Project Settings -> Physics

You will be able to see some information about the default physics in the game.

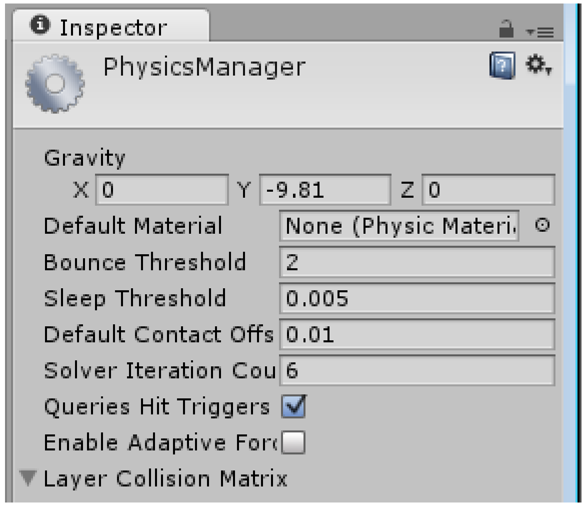
We want to focus on the Gravity Vector shown at the top. This is what we can use for default gravity which, if we wanted to, we could change and it would affect the gravity for all objects using it.

Using this, we can pass Physics.gravity's y value to push down on the player.

Now when your character is in game they will fall.

However this is not real gravity. For this we are falling at a constant speed of -9.81. While real gravity would be an acceleration starting at 0.

This will also override the jump gravity too. Which can cause issues.



Gravity X 0 Y -9.81 Z 0
Default Material: None (Physic Materi
Bounce Threshold: 2
Sleep Threshold: 0.005
Default Contact Offs: 0.01
Solver Iteration Cou: 6
Queries Hit Triggers: ✓
Enable Adaptive Forc: ☐
Layer Collision Matrix

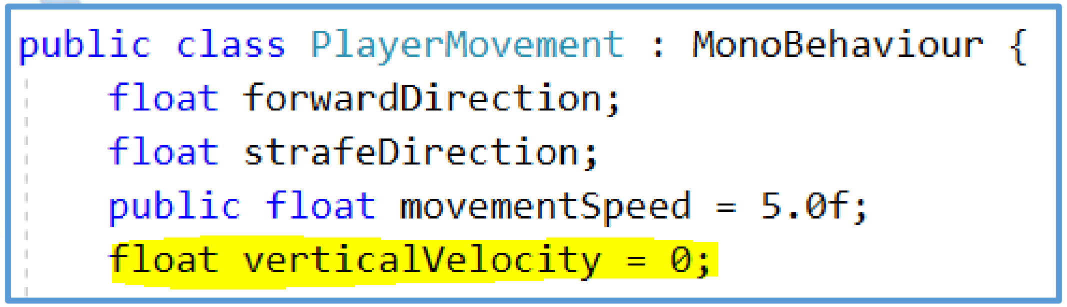
```
//Set the y coordinate to take from the Physics.gravity
Vector3 speed = new Vector3(sideSpeed, Physics.gravity.y, forwardSpeed) * movementSpeed;
```

NON-USED CODE - WE CAN PASS IN THE GRAVITY CONSTANT, HOWEVER THIS DOES NOT HELP US HAVE PROPPER GRAVITY

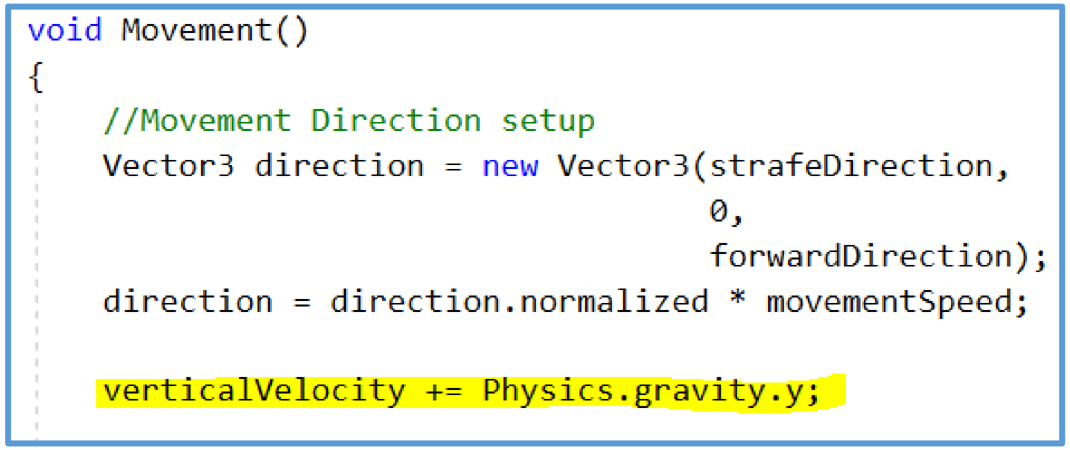
Keeping track of our fall

We want to keep track of our vertical velocity so we know how fast we are falling.

- In the class block
  3. Create a float variable called verticalVelocity
     a. Set it to equal 0
- In the Movement() function
  4. Set verticalVelocity to equal
     a. Itself plus Physics.gravity.y



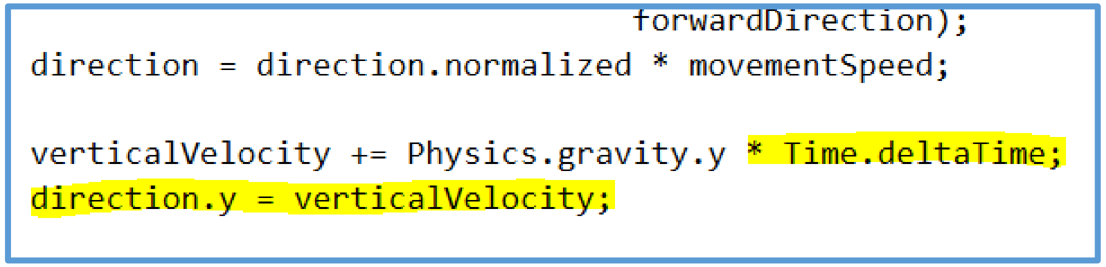
```
public class PlayerMovement : MonoBehaviour {
    float forwardDirection;
    float strafeDirection;
    public float movementSpeed = 5.0f;
    float verticalVelocity = 0;
```



```
void Movement()
{
    //Movement Direction setup
    Vector3 direction = new Vector3(strafeDirection,
                                    0,
                                    forwardDirection);
    direction = direction.normalized * movementSpeed;

    verticalVelocity += Physics.gravity.y;
```

Because it is acceleration, we can't just add the gravity every frame, as that would make us go to warp speed in seconds. To fix that we multiply the gravity by deltaTime.

5. Multiply verticalVelocity by Time.deltaTime
6. Set direction's y coordinate to equal verticalVelocity



```
                                forwardDirection);
direction = direction.normalized * movementSpeed;

verticalVelocity += Physics.gravity.y * Time.deltaTime;
direction.y = verticalVelocity;
```

Now when we start the game with our character high up and we will slowly start to fall. Accelerating by 9.81 every second.

JUMPING

Now that we have our gravity working. Let's start jumping.

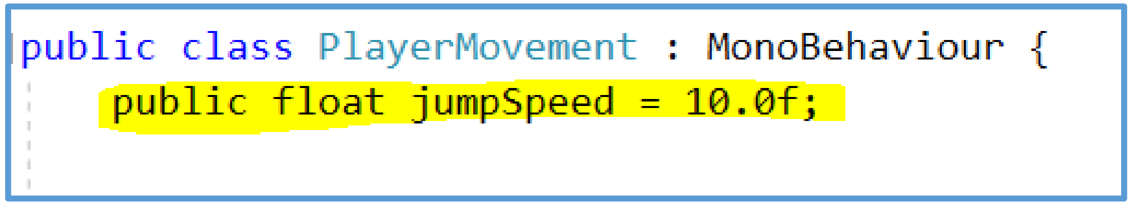
We want it so that when we hit the spacebar, we jump up a certain height. We can alter the upwards momentum by adding to the verticalVelocity

- In the class block

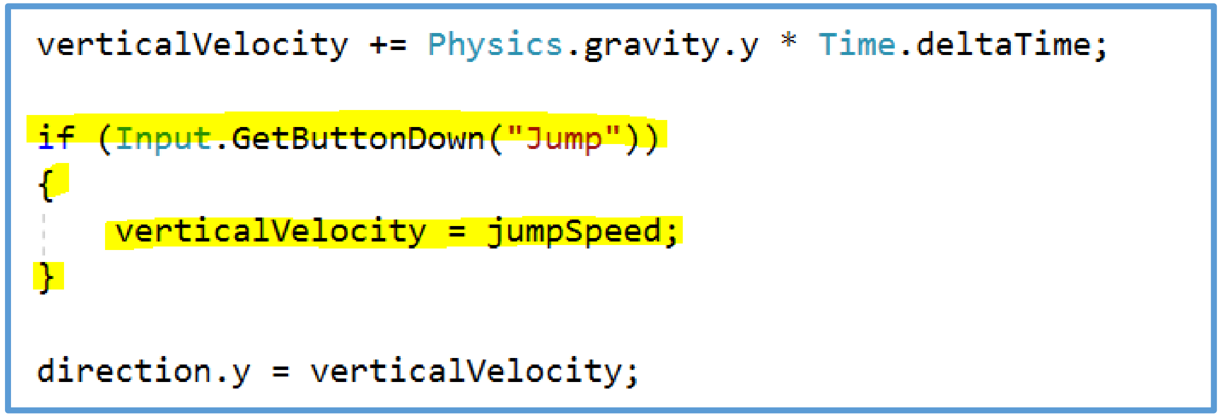
7. Create a **public float** called **jumpSpeed**
    a. Set its value to 10.0f

- In the Update() function
    - After verticalVelocity has been set

8. Create an if statement
    a. Have it check to see if **Input.GetButtonDown** is pressing the "**Jump**" Input
    b. If it is
        i. Set **verticalVelocity** to equal **jumpSpeed**



```
public class PlayerMovement : MonoBehaviour {
    public float jumpSpeed = 10.0f;
```



```
verticalVelocity += Physics.gravity.y * Time.deltaTime;

if (Input.GetButtonDown("Jump"))
{
    verticalVelocity = jumpSpeed;
}

direction.y = verticalVelocity;
```

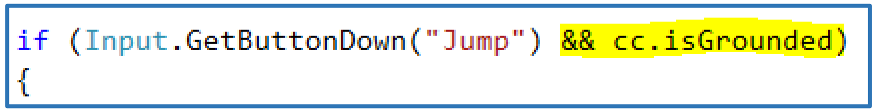
This creates a jump. A very high jump, but a jump. We can adjust the jump height depending on what you need for your game.

One issue however is that when you press jump again in the air. You will be able to jump again, and again, and again.

To make sure we can only jump off the ground, we need to know if we are on the ground or not.

Well luckily for us the character controller can tell us whether we are on the ground or not. All we have to do is put an if statement around the part you want to only work when grounded. In this example we will do it with the jump button pressed down. However, we could lock out movement or even have different movement depending on whether we are grounded or not.

- In the jump button if statement
  9. Add an **AND** check to see that **cc.isGrounded** is true



```
if (Input.GetButtonDown("Jump") && cc.isGrounded)
{
```

With that we have the player jumping and moving.

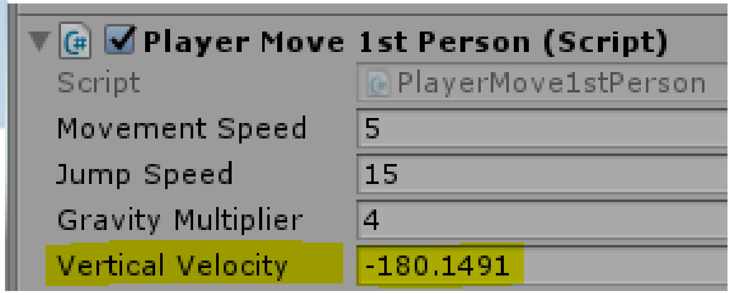
- Play around with the values to get the controls how you want them to be like.
  - Movement speed
  - Jump speed
  - You might want to increase the downward forces a little

Tweaking the Code

Once the core of the movement is working. There are a few things that we might want to tweak

Fall Code

If we leave the fall code as it is there will be some small problems.



```
Player Move 1st Person (Script)
Script                PlayerMove1stPerson
Movement Speed        5
Jump Speed            15
Gravity Multiplier    4
Vertical Velocity     -180.1491
```

The value just keeps getting lower and lower, to when we eventually fall we seem to teleport to the floor.

We don't notice this when jumping because we change the value back to a positive value and it resets our fall value.

We need a way to set our vertucal Velocity to 0 if we are on the ground

We can with the cc.isGrounded

- Above the verticalVelocity adition to gravity
    1. Create an if statement that checks if we are on the ground
        a. If so, set verticalVelocity to equal 0

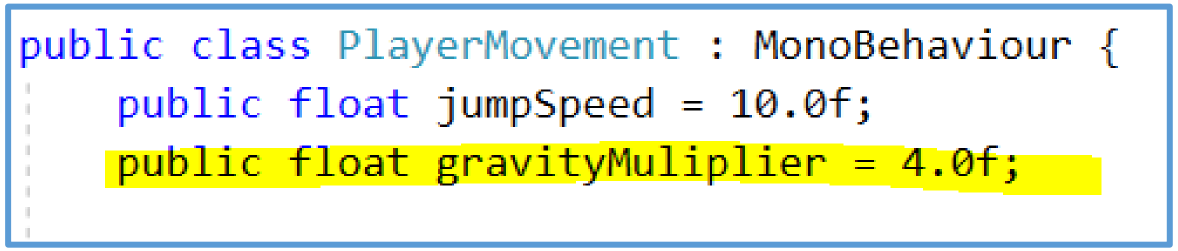
```
if(cc.isGrounded)
{
    verticalVelocity = 0;
}

verticalVelocity += Physics.gravity.y * gravityMultiplier * Time.deltaTime;
```

More Gravity

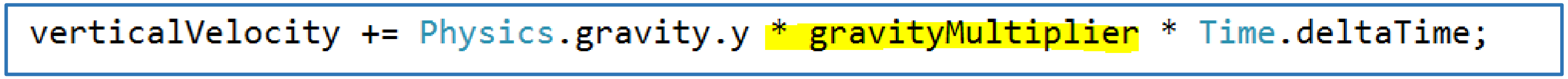
In the game at the moment the jump movement can be a bit… floaty. This is because we are jumping abnormally high but still falling at a regular gravity.

To fix this games often increase the gravity by up to 7X the weight of normal gravity.

1. We want to store our gravity Multiplier
2. And then multiply our physics by the gravity multiplier



```
public class PlayerMovement : MonoBehaviour {
    public float jumpSpeed = 10.0f;
    public float gravityMuliplier = 4.0f;
```



```
verticalVelocity += Physics.gravity.y * gravityMultiplier * Time.deltaTime;
```