Contents

[Scripting for 2D 1](#_Toc14075977)

[Creating our character 2](#_Toc14075978)

[Scripting a 2D Character 3](#_Toc14075979)

[Character Movement 3](#_Toc14075980)

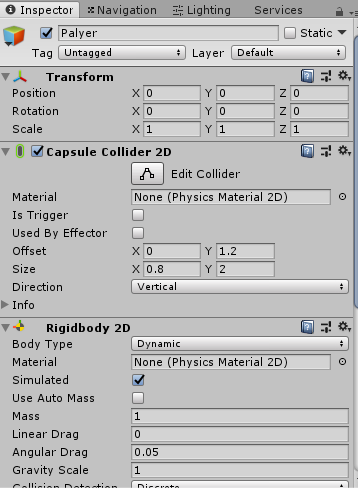
[2D Character Jumping 6](#_Toc14075981)

[Changing Players Direction 12](#_Toc14075982)

# Scripting for 2D

Most of what you know and love about scripting of 3D objects in Unity is the same. The primary difference that you will discover is the name of components has ‘2D’ added to the end. There are, of course, more differences than that, but we will address them as we encounter them.

# Creating our character

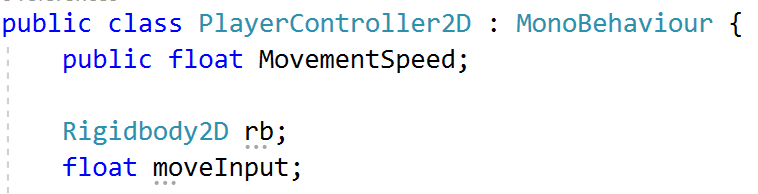
1. Create an empty game object
   1. Call it Player
2. Parent the art assets inside
   1. Make sure the art asset feet are at the base of the player game object
3. Attach a Capsule Collider 2D
   1. Set it to fit the size of the character
4. Attach a Rigidbody 2D

## Scripting a 2D Character

It is now time to write a script to move our character around the level. The two main features our character should have is movement and jumping. We will implement each of these in turn.

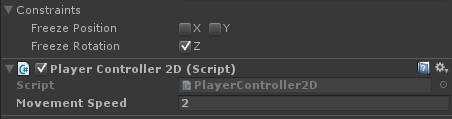
### Character Movement

Our first task is to get our character moving left and right on the screen.

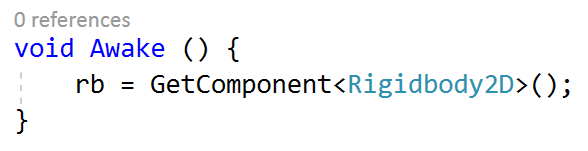
1. Create a script called PlayerController2D
   1. Attach it to the character

* In the Class Block

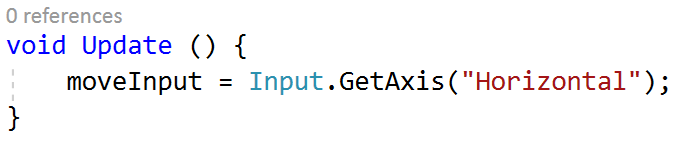
1. Add a public float variable called ‘MovementSpeed’.
2. Add a private Rigidbody2D variable called ‘rb’ to the script.
3. Add a private float variable called ‘moveInput’.

These 3 variables should be familiar to you. Input, speed and a Rigidbody are the 3 things that are common when first setting up most characters to move in Unity.

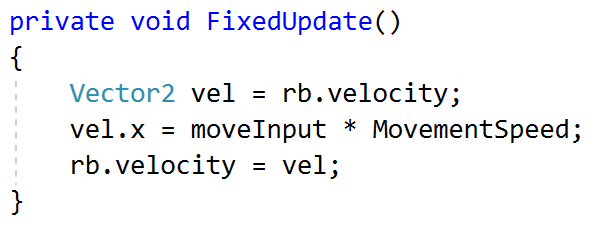
* In the Inspector

1. Set Movement Speed to 2
2. Add an Awake function to the script.
3. Get the Rigidbody2D component from the player and store it in the ‘rb’ variable.

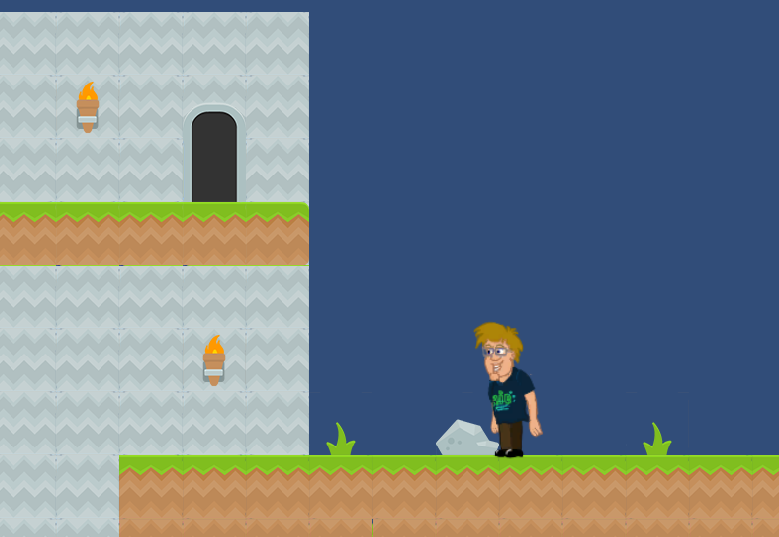
This is yet another familiar step, with the exception of 2D being added to the Rigidbody.

Next, we need to check our inputs.

1. Add an Update function to the script.
2. Get input from the “Horizontal” axis
   1. store it in the ‘moveInput’ variable.

As you will see, we check our inputs inside of Update, but move our character inside of FixedUpdate. Inputs only change each frame. The Update function is run each frame. This is an ideal place to check if we have pressed any buttons or moved any sticks.

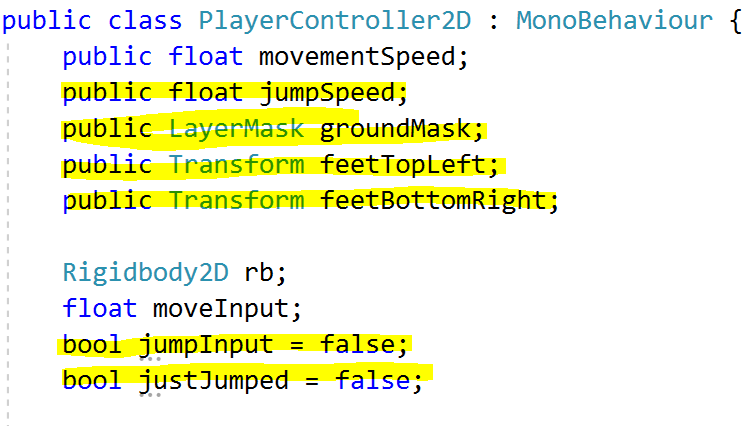
1. Add a FixedUpdate function to the script.
2. Get the current velocity of your player from the Rigidbody2D
   1. store it a Vector2 variable called vel.
3. Multiply ‘moveInput’ by ‘MovementSpeed’ to get how fast the character should be moving on-screen,
   1. store the result in the x-component of ‘vel’.
4. Store the newly updated ‘vel’ back into the velocity of the Rigidbody2D.

This is, of course, not a physically accurate way to move something. For a character we are controlling, however, it’s perfectly fine. We want precise control, not physically accurate control.

Jump back into Unity, select your character and set a movement speed. Now you’re all set to test out your movement! Don’t forget to set a **movement speed** on the character, otherwise your character won’t move.

### 2D Character Jumping

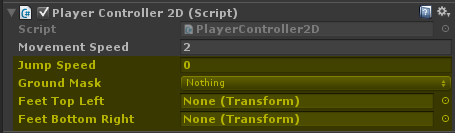
With moving our character complete, we can now try to get jumping working. This is going to require a bit more work.

1. Add some public variables to the top of the script.
   1. Add a float variable called JumpSpeed.
   2. Add a LayerMask variable called GroundMask.
   3. Add a Transform variable called FeetTopLeft.
   4. Add another Transform variable called FeetBottomRight.

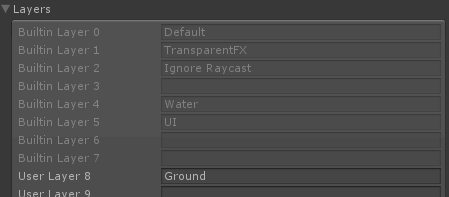
We need quite a few variables to set up correct jumping. We need to provide a speed for our jump, that much is clear. The purpose behind rest of these variables will become clear as we go.

1. Add a private bool variable called jumpInput
2. Add a private bool variable called justJumped

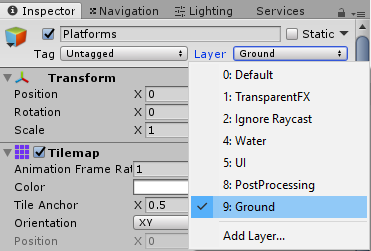
Inputs are also important!

With these variables in place, we will return to Unity and get them appropriately set up.

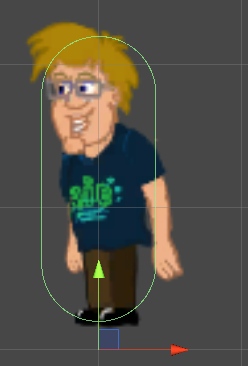
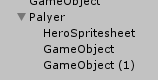
1. Set a speed for your jump. Anything over 10 should work nicely.

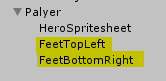
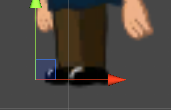
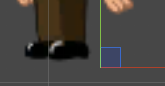
To set up the Ground Mask, we will need to create a new layer for the platforms to be on. We need to know when our character is standing on a platform, so we know that we are standing on something we can jump off of.

1. Open the Tags and Layers inspector. This can be found in Edit->Project Settings->Tags and Layers.
2. Open the Layers section.
3. In the box labelled ‘User Layer 8’, type “Ground” and hit Enter.

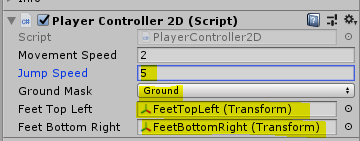
With that set up, we can now set up the Ground Mask on our character and all of our platforms to be on the new layer we set up.

1. Select your character, open the Ground Mask drop-down and click on the option labelled ‘Ground’.
2. For the Platform object Ground

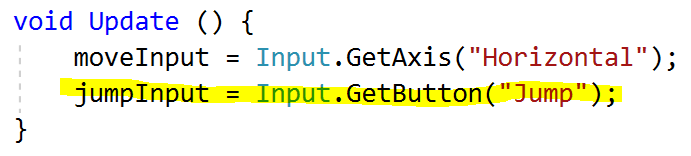
All that is left is too fill in the ‘Feet Top Left/Bottom Right’ boxes. These two transforms are going to form a rectangle at our character’s feet. If this rectangle intersects with one of our platforms, then we will consider our character’s feet planted and we can jump.

1. Right-click on your character in the hierarchy
   1. select ‘Create Empty’.
   2.  Do this twice, so you have two objects attached to your character.
2. Name one of them “FeetTopLeft” and the other “FeetBottomRight”.
3. Position each of these objects appropriately at your player’s feet.
   1. Use the Rect tool to help

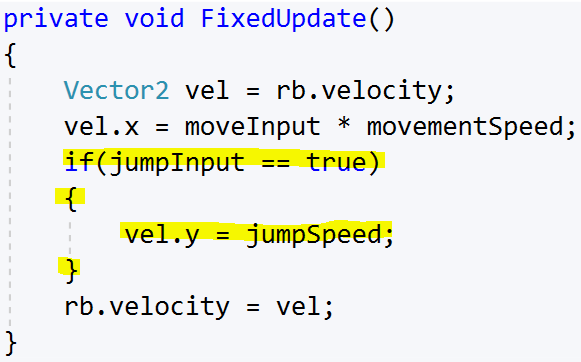
As was mentioned before, these two points will form a rectangle at the player’s feet. If you’d like to see that rectangle, select the two “Feet” objects and switch to the Rect Tool (the button in the top-left that is a square with dots in the corners).

1. Select your character and click-and-drag each “Feet” object into its appropriate slot.

We now have everything in place within Unity, let’s return to our script and put all of that set up to use.



1. In the Update function, check if the “Jump” button is pressed down and store that in the jumpInput variable.



* In the FixedUpdate function,

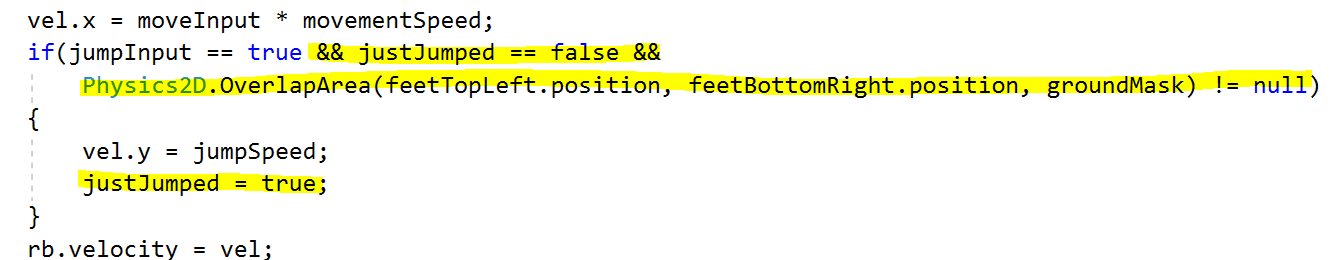
1. add an if statement in-between where you set the x-component of vel and where you set it into the Rigidbody rb.

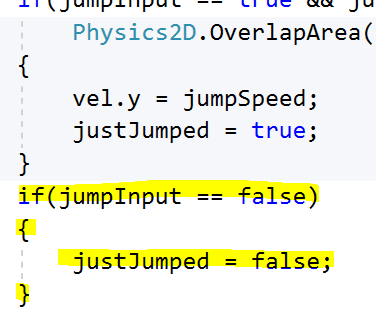
* In the if statement,

1. check if jumpInput is true.
   1. If so, set the y-component of vel to JumpSpeed.

Try this out and you will find that you can fly by holding down the spacebar (depending on the strength of your jump). This is obviously not what we want in a typical platformer, so we will use those “Feet” objects to address this.

1. Add 2 extra conditions to the if statement we wrote previously.
   1. If jumpInput is true,
   2. justJumped is false
   3. and the box created by our two “Feet” objects intersects with something on the ‘Ground’ layer, then allow the character to jump.



1. Add another if statement after the first that checks if jumpInput is false,
   1. If true set justJumped to false.

By checking the area at the character’s feet, the moment they are standing on a platform, you’ll be able to jump again.

You should now have a functional platformer!

If you find you can’t jump sometimes, check the “Feet” objects cover a large enough area beneath your character.

# Changing Players Direction

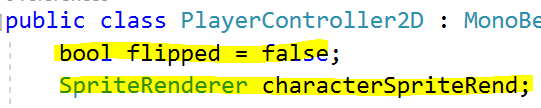
the sprite renderer has a way to flip the spite on the x or y axis

At the moment when we move our character, they are always facing the same direction. Lucky for us the sprite renderer has a flip option to flip our asset.

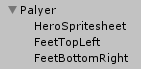
For our code. we need a few things to do this

* A bool to keep track of if we are flipped
* the Sprite renderer of the character
  + It has a bool that draws the sprite flipped

We then need to identify which direction our character needs to face.

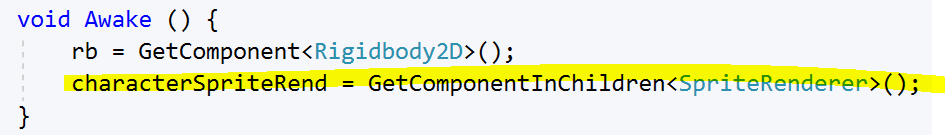
* In the PlayerController2D class block

1. Create a bool variable called flipped
2. Create a SpriteRenderer called charaterSpriteRend.

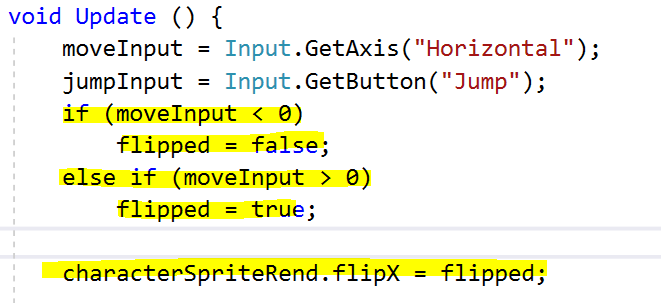
To store the characters sprite renderer, we need to know where it is.

As our spriteRenderer is not on the player game object but is on the art asset. We need a way to grab this and store it. We can use GetComponentInChildren to do this

Our version of the player has the sprite renderer in a child so we need to use getcomponentinchildren to access it

* In the Awake function

1. Set characterSpriteRend to grab from GetComponentInChildren

Now that we have the things that store our information, we need to set our flipped bool at the right time. You will need to think logically about this, or your character will be flipping all over the place.

Things to note

* We want the flip to stay the same direction when not moving
* Flipped is the opposite direction the character is facing
  + Our character is facing left so flipped will turn it right

With that in mind let’s do our logic

* In the Update function

1. Create an **if statement** that checks if **moveInput** is *less* than 0 (facing left)
2. If true set *flipped* to false
3. Create an **if statement** that checks if **moveInput** is *greater* than 0 (facing right)
4. If true set *flipped* to **true**
5. Set **characterSpriteRend** to set its **flipX** *bool* to what **flipped** equals.

Save and test

The player should flip to face where they move to.