



Craft Games

Tutorials to Help You Craft Better Games

≡ Menu



Unity 2D Platformer Movement (Beginner Friendly Tutorial)

June 6, 2019 by admin

Summary



1. Prerequisites
2. Start new project in Unity
3. Setting up basic platformer movement
4. Testing movement
5. Jumping



6. Check if player is grounded
7. Super Mario style jumping
8. Polishing up jumping
9. Adding double/triple jump to our platformer
10. Outro

In this tutorial I'll show you how to make a simple 2D platformer game with Unity. Every section of this tutorial will include all the code you need for that certain functionality to work. This tutorial is beginner friendly and I'll explain most of the things we do here. Although, I expect you to have at least some basic understanding of how Unity works and basics of programming.

Disclosure: Bear in mind that some of the links in this post are affiliate links and if you go through them to make a purchase I will earn a commission. Keep in mind that I link these companies and their products because of their quality and not because of the commission I receive from your purchases. The decision is yours, and whether or not you decide to buy something is completely up to you.

Prerequisites

Here's the list of things I expect you to already know / have in order to go through this tutorial successfully:

- Basic knowledge of how to code (variables, if/else statements, loops, methods)
- Basic level of understanding of Unity (knowing what are scenes, game objects, components, inspector)
- Unity game engine
- Visual Studio or VS Code or any other text editor that works with Unity ^

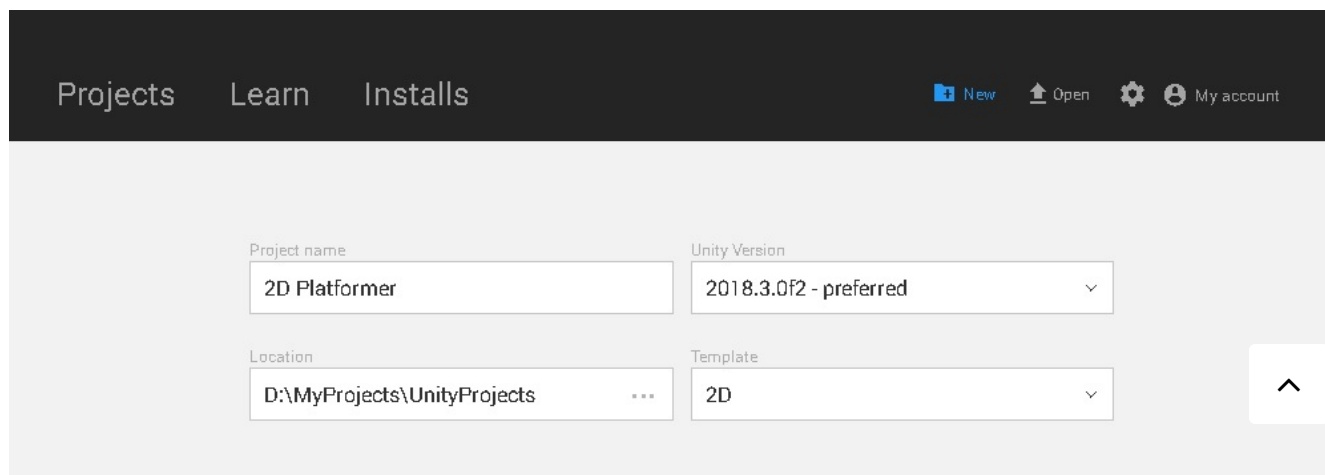
Here are some helpful resources to help you to get started:

- Check out **"Intro to Game Development with Unity"** course on *Zenva Academy* (<https://academy.zenva.com/product/intro-to-unity-game-development/?a=294&campaign=Unity-for-beginners>). In this course you'll master the foundations of game development by exploring Unity engine & the C# programming language.
- To learn basic of C# check out [this video by Code Monkey](#)
- To get basic understanding of Unity check out their [official Roll-a-ball tutorial](#) (it's very quick and easy to follow)
- [Download Unity here](#) (click on "Try Personal" to get started for free)
- Visual Studio will be installed with Unity if you keep the default settings.

Start new project in Unity

To start new project:

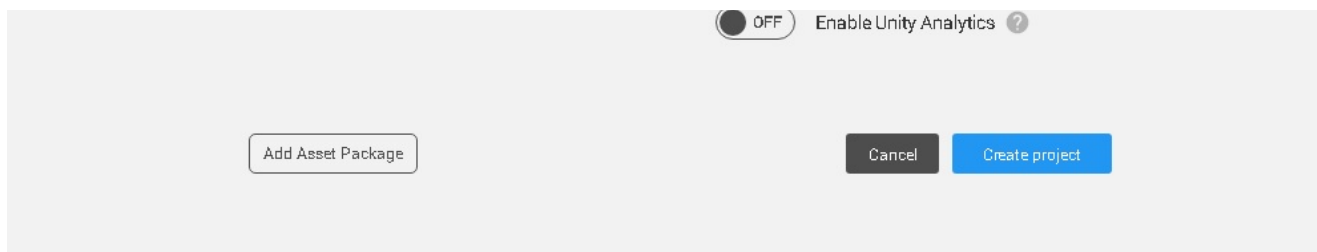
- Open up Unity
- Give your project a name
- Select the destination for your project
- Set template to 2D
- Hit "Create Project"



The screenshot shows the Unity Hub interface with the 'New' button highlighted. Below the navigation bar, the 'New Project' dialog is open, showing the following fields:

Field	Value
Project name	2D Platformer
Unity Version	2018.3.0f2 - preferred
Location	D:\MyProjects\UnityProjects
Template	2D

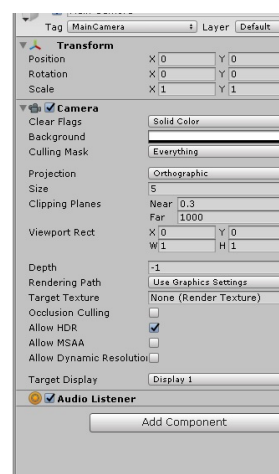
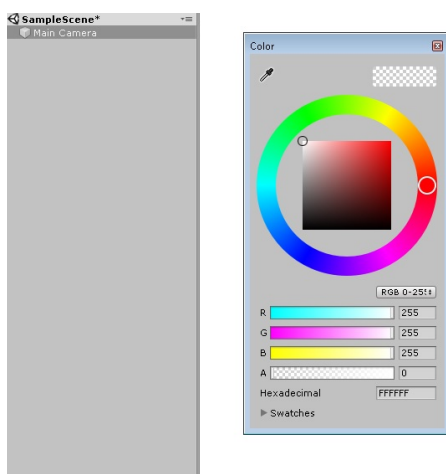
There is a small upward arrow button at the bottom right of the dialog box.



Creating new project in Unity

Next step isn't necessary, but I've changed my camera's color to make the scene look a bit better. If you want to do the same then:

- Select "MainCamera" game object
- Click on Background property of Camera component
- On the color wheel pick white



Setting Background color in Unity

Setting up basic platformer movement

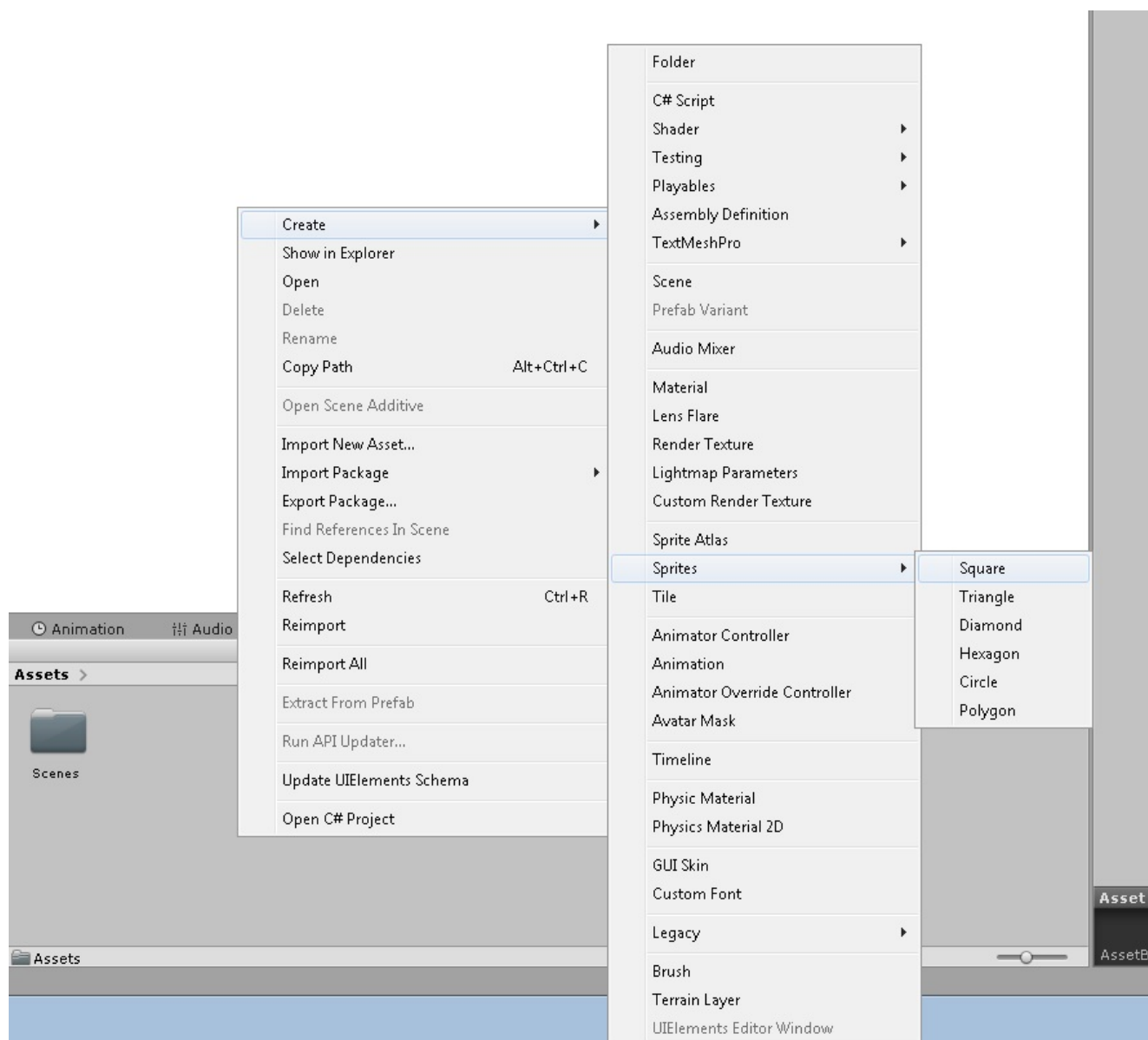
In this section we'll go through the process of creating basic character controller that can move left and right.

To keep it simple, everything in this tutorial is going to be made out of rectangles.

- Go to your Project window
- Right click in the Assets folder



- Navigate to Create -> Sprites -> Square

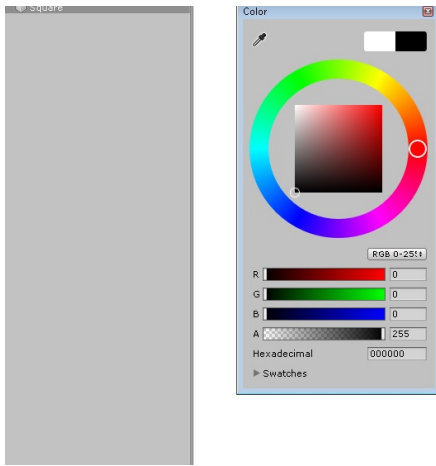


Creating Square asset

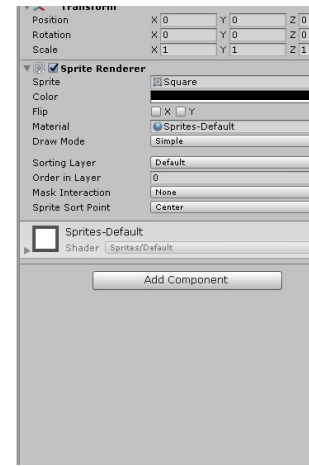
To create our character:

- Drag and drop square sprite you've just created into Hierarchy window
- Select this new game object (it's called "Square" by default)
- Navigate to its Sprite Renderer and click on Color property
- Pick black color





Setting color of the square



To make it dynamic we have to make sure that physics are applied to our Player. To do this:

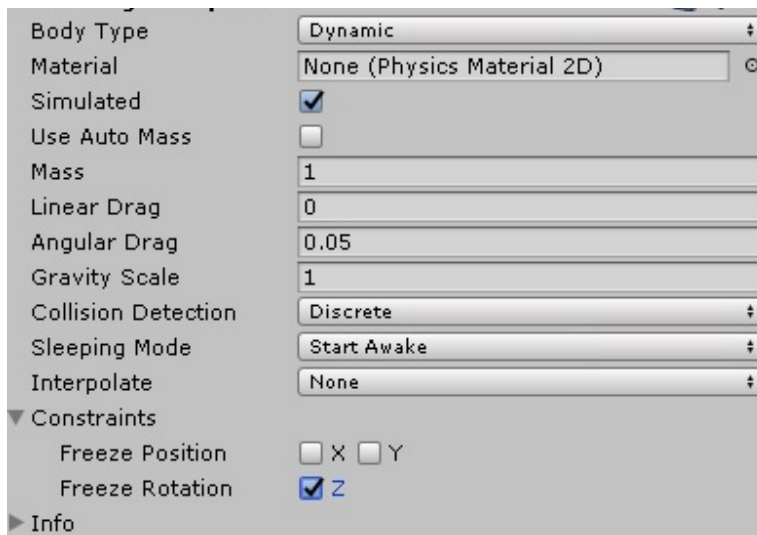
- Select "Square" game object
- In the inspector click "Add Component" button
- Type in "Rigidbody"
- Pick "Rigidbody 2D"

Unity uses Rigidbody component to determine which game objects are physics based. For example, objects that have Rigidbody attached will start moving downwards when you play the game because gravity is applied on them.

To make sure that we have more control over our Player we can set certain constraints. In our case, we would want to constrain player from rotating thus keeping him easier to control. To do this:

- Select "Square" game object
- In the inspector expand "Rigidbody 2D" component so that you can see all of its properties
- Click on "Constraints" to expand its properties
- Check "Freeze Rotation" (Z)

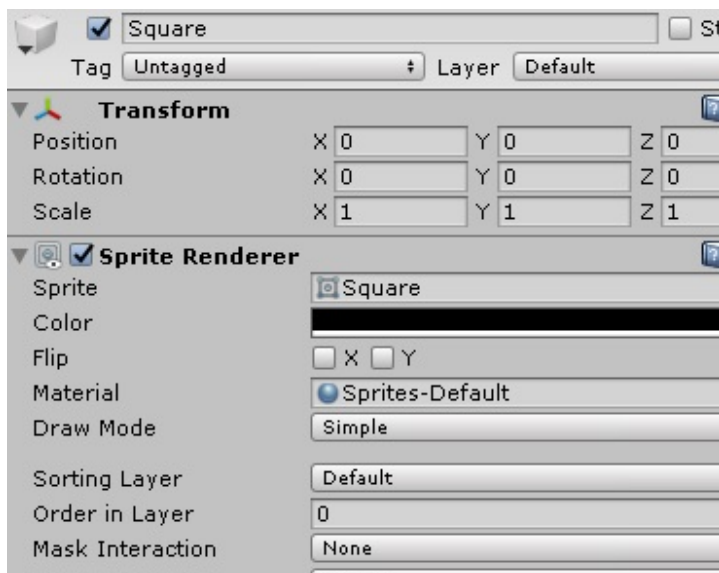
^

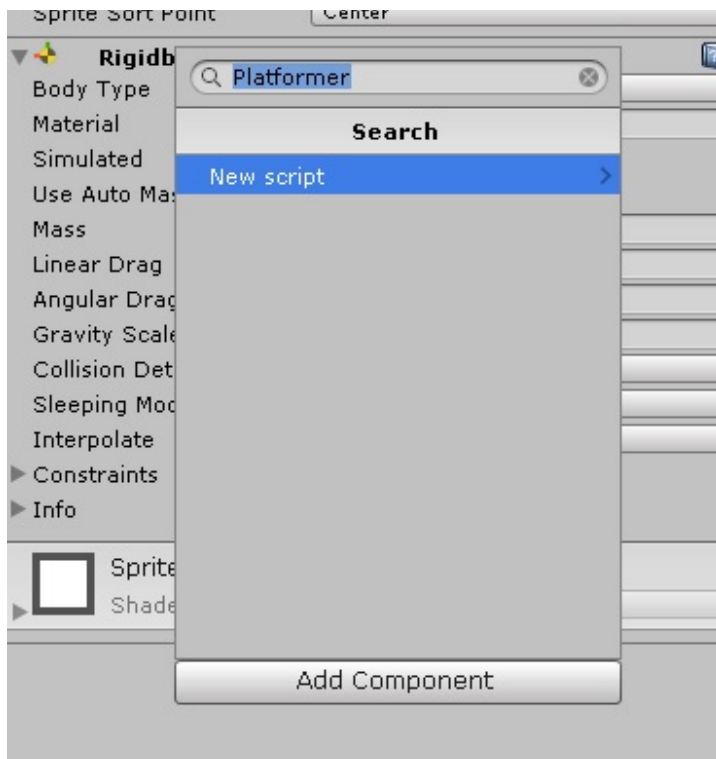


Setting constraint on Rigidbody 2D

In order to have control over this character we need to add a script to it. Process is similar to adding a Rigidbody component because, after all, the script is also a component.

- Select Square
- Click "Add Component" button
- Type in "Platformer"
- Click "New Script"
- Hit "Create and Add"
- This script (called "Platformer") will now be attached to your character and will also appear in your Assets folder





Adding new script called "Platformer"

Double click the script asset to open it up in Visual Studio (or some other editor, [here](#) you can find how to setup Unity for work in some external text editor. It's meant for an older version of Unity but the premise is the same).

Your script will look like this when it loads up:

```
1. using System.Collections;
2. using System.Collections.Generic;
3. using UnityEngine;
4.
5. public class Platformer : MonoBehaviour
6. {
7.     // Start is called before the first frame update
8.     void Start() {
9.
10.    }
11.
12.    // Update is called once per frame
13.    void Update() {
14.
15.    }
16. }
```

We are going to move our character using Unity's physics engine. To do [^]

we have to get a reference to our Rigidbody. Which can be done as follows:

```
7.  Rigidbody2D rb;  
8.  
9.  void Start() {  
10.     rb = GetComponent<Rigidbody2D>();  
11. }
```

Next up, when moving Player we obviously need to know at which speed we want to move it. So to make it as easy to tweak as possible we have to add a new variable. This code needs to be inside Platformer class but outside any of the methods. I'd suggest you put it on top, like we did with Rigidbody.

```
8.  public float speed;
```

Next thing we have to do is to actually take input for keyboard and make out character move. We can do that like this:

```
14. void Update()  
15. {  
16.     Move();  
17. }  
18.  
19. void Move() {  
20.     float x = Input.GetAxisRaw("Horizontal");  
21.     float moveBy = x * speed;  
22.     rb.velocity = new Vector2(moveBy, rb.velocity.y);  
23. }
```

In the move method we first get horizontal axis and set x to its value. This means that if player pressed "D" or right arrow x will have the value of 1. If player presses "A" or left arrow x will be -1. And if player doesn't press any button x will be 0.

Next up, we are calculating for how much we want to move each second and we put this value in "moveBy" variable.

After that, to actually move, we have to access velocity variable that's inside of Rigidbody. This variable, as the name suggest, holds value (of ^

type Vector2) that tells Unity at which velocity this rigidbody moves. So if we set this variable to (2, 0), 2 is the value on the x-axis and 0 is the value on the y-axis. Meaning that our object will move 2 meters per second in the right direction.

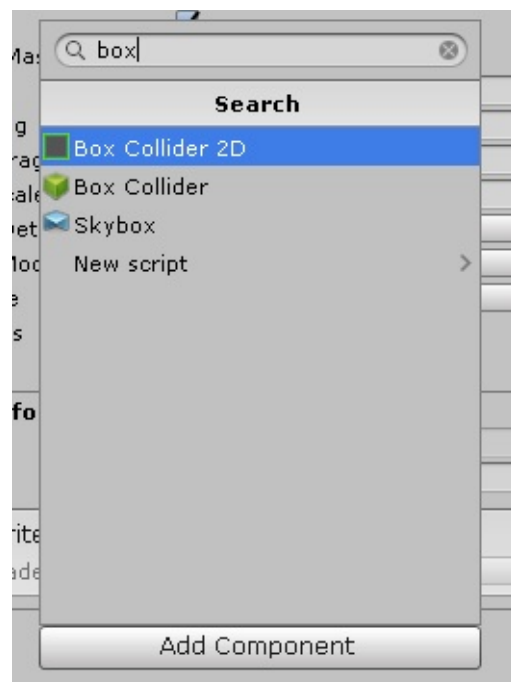
As you can see, when setting the value for velocity we are creating new Vector2, we are setting its x value to "moveBy" and setting its y value to "rb.velocity.y". This means that our Rigidbody will keep the same vertical velocity but its horizontal velocity will change depending on player input.

Note: In order for this method to work you need to call it from Update. That's a must.

Testing movement

To test out movement we need to first do a couple of things.

- Select "Square" game object
- Click "Add Component" and type in "Box"
- Select "Box Collider 2D"

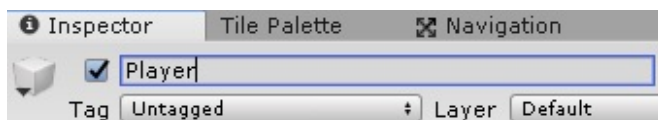


Adding Box Collider

Box collider basically tells the engine where are the bounds of our game object. This might seem obvious to us (we can logically conclude that object ends where its graphics end, but that might not always be the case).

To keep it more organized let's rename the square to "Player". You can do so by:

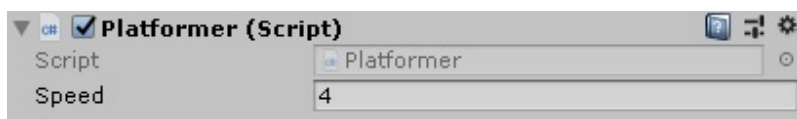
- Selecting "Square"
- Change "Name" in the inspector



Renaming a game object

Remember that we added that "speed" variable to our script let's now set its value.

- Select "Player"
- Find "Platformer" script component
- Set speed to be 4



Setting up speed variable

Also, to be able to move we need something to walk on. So let's add a ground object. Process is very similar to when we added player character.

- Drag and drop square asset (again) to our hierarchy
- This will create new object
- Rename it to "Ground" (same thing we did with the "Player")
- On Sprite Renderer click Color and set it to green (or whatever you prefer)

- Add "Box Collider 2D" to it as well (again, same as with our "Player")
- On "Transform" component of the "Ground" object set position to (0, -1, 0)
- Set its scale to (10, 1, 1)

Now, press "Play" button to play the game. As I said previously you can move using "A" & "D" keys or Left and Right arrows.

Retro Game Development Academy

Learn modern game mechanics while building arcade-style 2D games in the popular Unity game engine – perfect for beginners and portfolio projects.



Retro Game Dev Course on Zenva Academy

<https://academy.zenva.com/product/retro-gamedev-academy/?a=294&campaign=RetroGameDev>

Jumping

What is a platformer without jumping. Our obvious next step is to add jumping functionality to our character.

^

To do this let's go back to our script editor. We (again) need a variable

that's going to give us some more information. In this case, how much we want to jump.

I called this variable "jumpForce".

```
9. public float jumpForce;
```

Create new method inside your script called "Jump". Like this:

```
25. void Jump() {  
26.     if (Input.GetKeyDown(KeyCode.Space)) {  
27.         rb.velocity = new Vector2(rb.velocity.x, jumpForce);  
28.     }  
29. }
```

This method works similarly to how we moved our player. Only difference is that in this case we are checking if "Space" key is pressed. If it is pressed change velocity of our Rigidbody. This time we don't want to change Player's speed on the x-axis but we want to make our velocity positive on the y-axis. This way we actually jump because this force is applied immediately.

Note: Don't forget to call this method from Update like we did with our "Move" method.

```
15. void Update()  
16. {  
17.     Move();  
18.     Jump();  
19. }
```

Again, save script, press "Play" and try pressing "Space".

Check if player is grounded

If you're following this tutorial step-by-step and you tried jumping you might have noticed one thing. We can jump indefinitely. This obviously isn't a behavior that we want when jumping.

To fix this we have to check if player is standing on ground (if he is

^

grounded) and only then jump if "Space" button is pressed.

To do this we have to add a new method to our script. But first up this method is going to need some variables.

```
10. bool isGrounded = false;  
11. public Transform isGroundedChecker;  
12. public float checkGroundRadius;  
13. public LayerMask groundLayer;
```

Variable "isGrounded" is set to "true" when Player is on the ground and set to "false" when Player is in the air.

I'll explain last 3 variables, but I think that now it's more important for you to understand how all this is going to work.

So, you can imagine it like this, we are spawning a small circle of a certain (rather small) radius bellow the player. Let's call this circle "GroundChecker". Basically "GroundChecker" is going to tell us whether player's feet are close to the ground. This way we can know if he is grounded or not. Pretty simple. Right?

So now, let's see what each of those 3 variables mean.

"isGroundedChecker" is going to be a Transform of an empty object that is going to be placed bellow player.

"checkGroundRadius" is going to tell us what's the radius of our "GroundChecker".

"groundLayer" is literally what it says. Unity has its system of layers and each object can be on its own layer. These layers help us to understand if a game object is of a certain type. In this case, if our circle collides with some object we would like to know whether that object is considered to be ground because if it isn't we don't want to set player to be grounded. I hope I explained that well here.

^

Now, let's create the actual method that does all this.

```
35. void CheckIfGrounded() {  
36.     Collider2D collider = Physics2D.OverlapCircle(isGroundedChecker.position,  
    checkGroundRadius, groundLayer);  
37.  
38.     if (collider != null) {  
39.         isGrounded = true;  
40.     } else {  
41.         isGrounded = false;  
42.     }  
43. }
```

I called this method "CheckIfGrounded". First up, inside our method, we initialize an object called "collider". Value of this object is going to be whatever collider "OverlapCircle" returns.

When calling "OverlapCircle" (and this is a method that is spawning that imaginary circle I mentioned before) we need to provide it with a position, radius and layer mask. We already have those variables ready so that part is pretty simple.

Next up, we have to check whether that imaginary circle actually collided with something. If it did collider's value is not going to be "null" so this way we know that we collided with ground and that we are actually grounded.

If collider's value is "null" then we are not grounded and "isGrounded" is set to "false".

To make sure that this method works you have to **call it from Update**.

```
19. void Update()  
20. {  
21.     Move();  
22.     Jump();  
23.     CheckIfGrounded();  
24. }
```

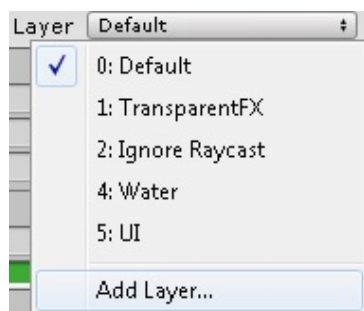
Also, we have to modify our "Jump" method a bit to make sure to account for when the player is grounded. "Jump" method should now look like ^

```
29. void Jump() {  
30.     if (Input.GetKeyDown(KeyCode.Space) && isGrounded) {  
31.         rb.velocity = new Vector2(rb.velocity.x, jumpForce);  
32.     }  
33. }
```

As you can see velocity is now going to be changed only if both "Space" key is pressed and player is grounded. So now we won't be able to fly infinitely.

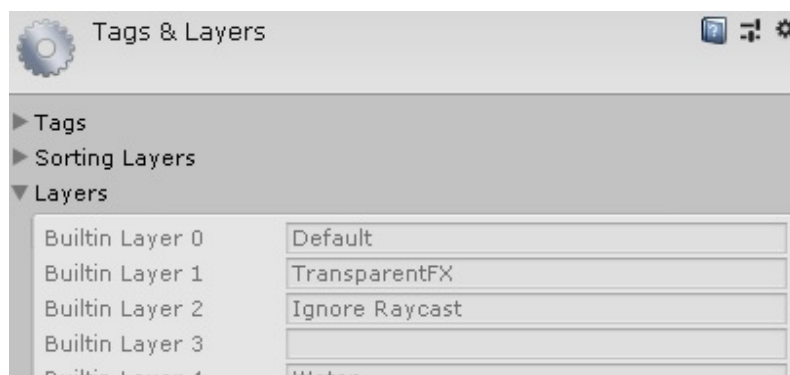
But, to test this out we have to do a couple of things.

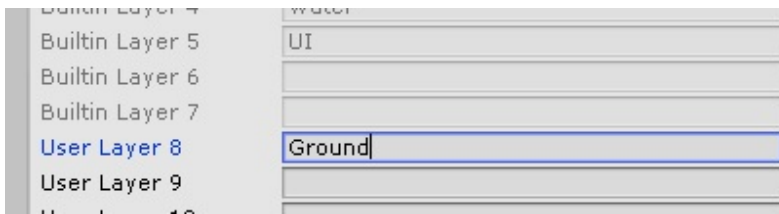
- Go back inside of Unity editor
- Select "Ground" object
- In the inspector (in the upper right corner) click on "Layer"
- Dropdown menu will appear with all the currently available layers
- We want to add a new one so press "Add Layer"



Press "Add Layer" button

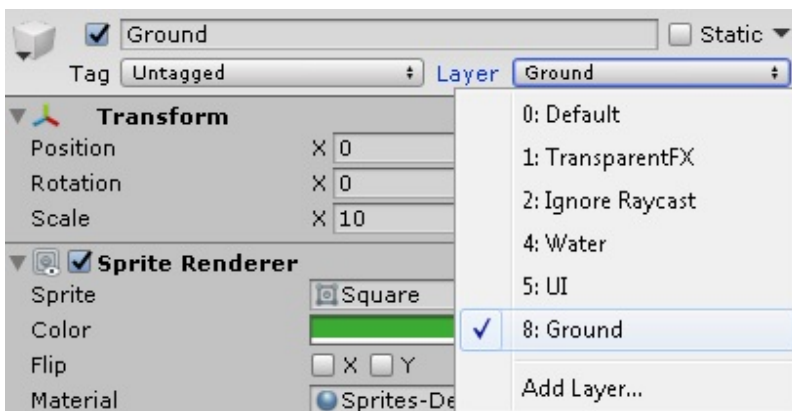
- New menu will appear with the list of all the layers as well as a lot of empty slots
- In one of those empty slots type "Ground"





Adding new layer in Unity

- Again, select “Ground” game object
- Click on its layer property again
- Now this new “Ground” layer will appear in dropdown as well
- Select it



Changing object's layer to “Ground”

Now, to make our “GroundChecker” work properly, we have to:

- Create an empty game object
- Rename it to “GroundChecker”
- Set it as a child of “Player” object

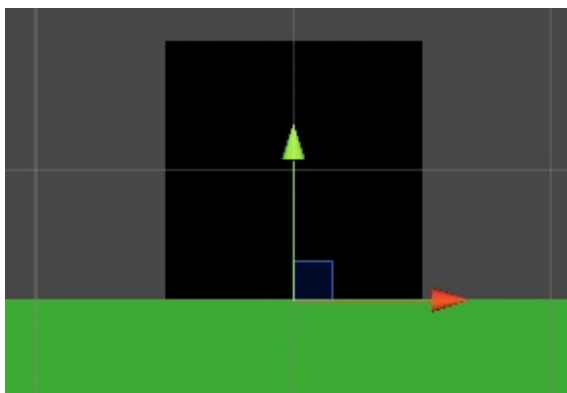


Create “GroundChecker” object

- Place it right under the player

^

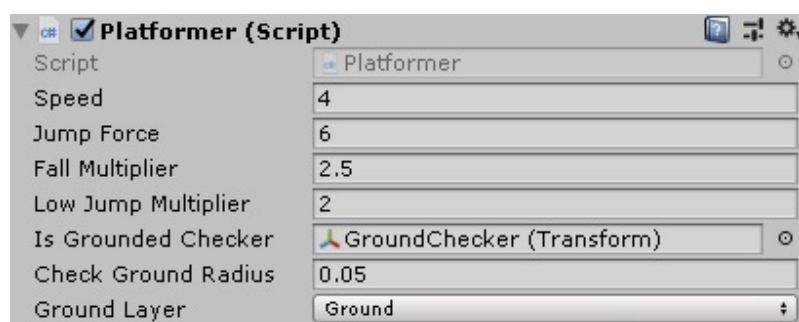




Place "GroundChecker" under player

Now we just have to set everything up on our script component.

- Select "Player"
- Drag and Drop his newly created "GroundChecker" object into "Is Grounded Checker" slot
- Set a value for "Check Ground Radius" to be 0.05
- For "Ground Layer" property select "Ground" layer



Setup variables related to ground checking in the inspector

Super Mario style jumping

If you're following this tutorial step-by-step and you tried jumping you've probably seen that jumps feel a bit floaty. It feels like player is falling down too slow.

There isn't really any problem with our code or with Unity's physics system, this feels bad just because you've played other games.

^

Games usually tweak laws of physics a bit to make them fit the game. So, in many platformers character spends a lot more time getting to the peak of his jump than he spends to get down to the ground. That's why in many platformers jump feels more crisp than jump in our game.



Super Mario style jumping in platformer

This concept is really well explained in a video by “[Board To Bits Games](#)” in [this video](#).

Thankfully, he also gave us a way to fix this issue. I'd strongly suggest you watch his video (it's pretty short) to get better understanding of how this all works. If not you can just copy the code from bellow and paste it into your “Platformer” script.

```
14.     public float fallMultiplier = 2.5f;
15.     public float lowJumpMultiplier = 2f;
```

```
47.     void BetterJump() {
48.         if (rb.velocity.y < 0) {
49.             rb.velocity += Vector2.up * Physics2D.gravity * (fallMultiplier - 1) *
Time.deltaTime;
50.         } else if (rb.velocity.y > 0 && !Input.GetKey(KeyCode.Space)) {
51.             rb.velocity += Vector2.up * Physics2D.gravity * (lowJumpMultiplier - 1) *
Time.deltaTime;
52.         }
53.     }
```

Also, make sure to call “BetterJump” in Update method to make this all work.

```
21.     void Update()
22.     {
23.         Move();
24.         Jump();
25.         BetterJump();
26.         CheckIfGrounded();
27.     }
```



Polishing up jumping

Now that we already started adding polish to our jump let's do one more thing.

To demonstrate what problem we are fixing next let's first:

- Duplicate "Ground" object
- Set its position to (6.5, 0, 0)

It should look like this:



Create new ground object

Now let's play the game and see what happens when we try jumping from this newly created ground object.



Jumping doesn't
feel right at the
moment

As you've might noticed, when we press "Space", right when we're approaching the end of the ground, nothing happens. This somewhat makes sense because at that moment in time we are not grounded so ^

can't jump. That's exactly what we wanted to achieve in one of the previous sections.

The problem in all this is that it just feels stupid when you press a button and nothing happens in a game. It feels bad and unresponsive.

To fix this, we can remember what is the last time we were grounded and then, when we jump, we can check how much time has passed from that moment and depending on that we can decide whether or not we want to let our player jump.

To achieve this we would need two more variables:

```
16. public float rememberGroundedFor;  
17. float lastTimeGrounded;
```

"rememberGroundedFor" will basically help to keep us grounded for a little longer.

"lastTimeGrounded" tells us exactly that, when was the last time we were standing on the ground.

To make this all work we have to modify our "Jump" method.

```
33. void Jump() {  
34.     if (Input.GetKeyDown(KeyCode.Space) && (isGrounded || Time.time -  
lastTimeGrounded <= rememberGroundedFor)) {  
35.         rb.velocity = new Vector2(rb.velocity.x, jumpForce);  
36.     }  
37. }
```

So if we are grounded we want to jump, but if we are not grounded we can check whether the time that has passed from "lastTimeGrounded" is less than or equal to "rememberGroundedFor".

Now, you probably noticed that we haven't set a value for "lastTimeGrounded". Let's do that now:

```
65. void CheckIfGrounded() {
```



```
66.     Collider2D colliders = Physics2D.OverlapCircle(isGroundedChecker.position,
67.     checkGroundRadius, groundLayer);
68.     if (colliders != null) {
69.         isGrounded = true;
70.     } else {
71.         if (isGrounded) {
72.             lastTimeGrounded = Time.time;
73.         }
74.         isGrounded = false;
75.     }
76. }
```

As you can see, when "collider" is equal to "null" we want to check first whether we are grounded. If we are then we want to give a value to "lastTimeGrounded". "Time.time" is a variable that holds how much time has passed since we're running our game.

Let's now go back to Unity.

- Select "Player"
- On "Platformer" script set a value for "rememberGroundedFor" to be something small (0.1 for example)

Now let's run the game and see what has changed.



Jumping is fixed and it now feels a lot better

As you can see, now we can jump even if we are right at the edge of a ground.

Make sure to try out different values for "rememberGroundedFor" to make it feel just right for your game.

^

Adding double/triple jump to our platformer

To complete this course we can add one more thing to our game. An ability for our player to perform double or triple jumps.

To do this let's just add 2 more variables:

```
18. public int defaultAdditionalJumps = 1;
19. int additionalJumps;
```

"defaultAdditionalJumps" will be a variable that tells us how many additional jumps we have. When set to 1 it will allow us to perform double jumps.

"additionalJumps" is doing the same thing but we're going to change its value and that's why I made them separate.

Now let's modify our "Jump" method once again to account for these additional jumps.

```
52. void Jump() {
53.     if (Input.GetKeyDown(KeyCode.Space) && (isGrounded || Time.time -
lastTimeGrounded <= rememberGroundedFor || additionalJumps > 0)) {
54.         rb.velocity = new Vector2(rb.velocity.x, jumpForce);
55.         additionalJumps--;
56.     }
57. }
```

As you can see we want to subtract from "additionalJumps" every time we jump. Also, we need to add one more condition to this method that check whether we have any additional jump left.

To make this work we also need to reset the value of "additionalJumps" every time we stand on the ground. Like this:

```
67. void CheckIfGrounded() {
68.     Collider2D colliders = Physics2D.OverlapCircle(isGroundedChecker.position,
checkGroundRadius, groundLayer);
69.
70.     if (colliders != null) {
71.         isGrounded = true;
72.         additionalJumps = defaultAdditionalJumps;
73.     } else {
```



```
74.         if (isGrounded) {  
75.             lastTimeGrounded = Time.time;  
76.         }  
77.         isGrounded = false;  
78.     }  
79. }
```

You may not understand why this actually works so I'll try to explain it to you. When we jump our character will be pushed up. At this moment "additionalJumps" is going to have a value of 0, because when we jump we subtract 1 from it and its default value is 1.

While going up, player will register that he is still on the ground for a really short time. This is enough to reset "additionalJumps" back to 1. Meaning, while in air we can jump once again.

Outro

That's it. I hope you enjoyed this tutorial and that it was helpful to you. Thank you for reading it!

This script that we made can be downloaded from here: [PlatformerScript](#).

To learn the basics of developing your own **mobile games for Android and iOS** using the Unity game engine visit: <https://academy.zenva.com/product/mobile-games-101-android-and-ios-game-development/?a=294&campaign=Android&iOSGameDev>

To find more tutorials go to <https://craftgames.co/category/tutorial/>.

📁 Tutorials

🔖 game design, game dev, platformer, tutorial

< Improve your Level Design (Tips for Indie Devs)

> Unity 3D FPS Movement (Beginner Friendly Tutorial)

