WSoft Platformer

Setting Up Your Project

Create a new Unity project (set to 2D). Title it whatever you want, like “WSoft Platformer”.

Which mode to create a project in: 2D or 3D?
When you create a project, selecting 2D or 3D won’t limit you to anything within the editor - it just sets some defaults for you. For this tutorial, we wanted to expose you to some 2D aspects (like sprites). You can read more about the difference here: https://docs.unity3d.com/Manual/2Dor3D.html

In your project tab in the editor, click the “create” button, and in the menu that pops up, go to sprites->square. This will create a simple default square sprite for you to use.

Click the “create” button again to create a scene, and call it “main”. Move this scene into the scene folder. Double click on “main” to open up the scene - you can double check your hierarchy to make sure you’re working in the right scene. Any time you make a change, make sure to save your scene (ctrl + s), and do this often.

How should asset files be organized?
There are two common ways to go about this - one is to create folders for each type of file - a folder for sprites, a folder for prefabs, a folder for scripts, etc. Another is to create a folder for each game object, where for example, a player folder has within it the player prefab, player sprites, player scripts, etc.

Setting Up Your Player

In the navbar of the Unity editor, go to GameObject->2D object->sprite. This will create a Game Object with 2 components: a Transform component (editor manual, scripting API) and a Sprite Renderer component (editor manual, scripting API). Rename this Game Object to have the name “Player”. Select the Player’s tag to be “Player”. Also add a “Player” layer, and select it for Player. In your Transform component, click on the settings icon in the top right, and hit “reset”. (Usually, this shouldn’t change anything - if your transform has position and rotation values of (0, 0, 0), and scale values of (1, 1, 1), you can actually skip this step - this is to get you in the habit of resetting the transform so you know exactly what settings you’re starting with.) In your Sprite Renderer component, click on the sprite field where it says “None (Sprite)” and add your Square sprite. Now you should be able to see your object in the center of the scene.

What are these links to the editor manual and scripting API?
These links are documentation to explain how Unity's editor and the scripting functions work. We're providing some of these links to you to get you in the habit of referencing the documentation to be able to figure out on your own how to use the editor and scripting API. To get to these links yourself, just google "Unity Transform" or "Unity Sprite Renderer", or access the documentation in the editor by going to Help->Unity Manual or Help->Scripting Reference. You can also access the manual for each component by hitting the book icon in the top right corner of each component in the inspector.

To your Player, add the Box Collider component, and the Rigidbody component. In your Rigidbody component, unhide the "Constraints" section, and check the boxes for Freeze Position Z, and for Freeze Rotation X, Y, and Z.

Why are we using the 3D versions of these components, rather than the 2D versions?
The difference between 3D and 2D is in script. We consider it preferable to learn to use 3D physics, because then you can apply it to 3D games and 2D games - for 2D, you just need to restrict the Z axis value.

Now's a good time to try to play your scene. The Player object should simply fall of the screen. Exciting, but not much gameplay.

Setting Up Platforms

Well, since there's not much for the Player to do, let's start building our scene. Click and drag the "Square" asset into your scene. This should create a Game Object, also with a Transform and Sprite Renderer - but this time it will automatically have the Square sprite set. Rename this obejct to "Platform". Add a tag to this object as well - click "add tag" in the tag dropdown menu, and add "Platform". Go back to your game object and add the newly created "Platform" tag. Move this object down a bit - set its Transform position to (0, -2.5, 0). We also want to scale it a bit - set its Transform scale to (10,1, 1).

We have a problem, though, when we play the scene: the Player still just falls through off the screen! To remedy this, add a Box Collider to Platform. This will ensure that the Player will collide with the Platform - and land on it.

Now that we have our Player and Platform objects made, we're going to make a couple prefabs out of them so we can use other instances of them. Click and drag the Player object from your scene hierarchy to your assets folder to create a Player prefab. Do the same with the Platform object. Both should now have a prefab file in the assets folder, and should appear blue in the scene hierarchy. Go ahead and organize these files, along with the Square sprite, as you please.

Player Movement

Now we're going to give the Player some movement. In the Player object, hit "Add Component", and type in "PlayerMovement" in the search bar. Click "New script", make sure the name is correct, and hit "Create and Add". This will create and add a script component to your game object. Wait a second for Unity to compile the script - while Unity is doing so, there will be a timer wheel in the bottom right corner, and the script name will be bold - and then double click the script to open it up to edit. (If you don't wait for the script to compile and get too antsy, Unity won't properly recognize the file for the Intellisense/autocomplete).

Now we'll add some code to the PlayerMovement script:

1. In your PlayerMovement script, above Start, add a public float speed variable. Go back to your Player inspector, wait for the script to compile - and now you should see a Speed field. Set that value to 1 for now.
2. Now scroll up the inspector, to where you see a "Prefab" field just above the Transform component. Hit "apply". This will save the changes you've made to this object to the prefab - any Player you instantiate from the Player prefab will have a script, with a speed value set to 1. (Note: any values that differ from the Prefab will appear bold - once you apply, they won't be bold anymore).
3. Underneath the speed variable, and above start, add a Rigidbody rb variable - do not make this public. This Rigidbody component will be used to set the velocity of the game object.
4. In Start, set rb to GetComponent<Rigidbody>(). This will grab the Rigidbody component attached to the player. Note that GetComponent is a very expensive operation - do not use this in update! It will slow down your game by quite a lot. Always use this in Start, since Start will only run once per game object.
5. In your update loop, add one variable, float xVelocity. Set it equal to speed * Input.GetAxis(). In the parenthesis, we'll add a parameter for xVelocity to ensure we get the right input - if we put in "Horizontal" (with the quotes), we'll get the left and right arrow key input, and if we put in "Vertical" we'll get the up and down arrow key input. Since we want to move left and right, use "Horizontal".
6. Make a variable Vector3 velocity. Set it equal to a new Vector3(xVelocity, rb.velocity.y, 0). The input parameters are x, y, and z. x will be set to xVelocity, y will be set to the current rb.velocity, and z will stay 0 (to keep in line with our 2D physics). This Vector will give the Player a velocity when the player wants to move.
7. Finally, set rb.velocity to our Vector3 velocity.

By the end of these steps, your script should look like this.

Why are you giving the code as an image, rather than letting us copy/paste?
You'll learn better this way, if you have to type it out yourself. Plus, it's helpful to have the explanations for each line of code to look back as a reference. We want to get you in the habit of not copy/pasting code, but understanding what thought process is building this code.

While writing your scripts, we recommend doing so in small increments, switching back to the editor to test your changes as often as possible. ~4 lines or so of code is a perfect time to test what we have, so let's go ahead and do that. Switch back to the editor and hit play. Try moving around. While it works, the movement will feel pretty awful - it's so slow! To remedy this, in your Player inspector, fiddle around with the speed, and see how it plays. For now, set the speed value to 5, and hit "apply" to save this value to the prefab.

Notice how we used a public variable for speed, rather than setting it with the script.
This is because it's much more convenient to change a public variable than to hardcode and change a value within a script - you can easily change the speed within the editor, and even quickly change it as you play! (Note that changes made while in play mode will not be saved.)

Jumping

Now we're going to go ahead and add a jump to our Player using the PlayerMovement script:

1. Underneath the public speed variable you added, add another one: public float jumpPower. Go back to the inspector, and set this value to 1.
2. In your update loop, under your xVelocity variable, add a float yVelocity variable, and set it to rb.velocity.y. Below yVelocity, add an if statement that checks if Input.GetAxis("Vertical") is greater than 0. In between curly braces { }, set yVelocity equal to jumpPower.
3. Now, go to the line where you declare add a new variable Vector3 velocity. Change the y parameter (replace rb.velocity.y with yVelocity).

By the end of these steps, your code should look like this.

Now, save the script, and go back to your editor. Tap the up arrow key. You might notice we have a problem - nothing happens when you tap it! That's because our jumpPower is too low. Change the jumpPower value and experiment with the results. Set jumpPower to 7 for now, and apply that to the prefab. Now you should be able to see the player jump.

However, we still have a problem - the player can now fly indefinitely if the up arrow key is held down. This is not the behavior we want, so we're going to remedy that - by ensuring the player can only jump if it's already on the ground.

In our PlayerMovement script:

1. Below Update, create a new function, bool IsGrounded(). We will use this to check if we are on a platform or not. If we are grounded, it will return true, and if we are not grounded, it will return false.
2. Inside curly braces, add a variable int layerMask, and set it equal to LayerMask.GetMask(). The parameter for GetMask should be the layer that we want to check - which will be the “Platform” layer, so input that with quotes.
3. Finally, add a return statement. This line will call Physics.Raycast(). A raycast will basically send out a ray from a starting position in a direction with a certain max distance. This will return true if it hits something, and false if it doesn’t. We can pass in our layer mask for platform to make sure that the raycast will only return true if it hits a platform. The parameters of our raycast will be (transform.position, Vector3.down, 0.6f, layerMask). If a platform is below the Player, we will return a value of true.
4. Lastly, in Update, in the if statement that checks if the player hits the up arrow key, we’re going to add our IsGrounded() condition. To do so, add a && IsGrounded(): this means only jump if we get an up arrow key input *and* the player is grounded.

By the end of these steps, your code should look like this.

Now, you should be able to move around, and jump only when grounded. You can experiment with expanding the level by dragging in Platform objects from the prefab, and moving them around and resizing them.

Setting Up Enemies

As fun as this platforming is, we could use a challenge now. Let’s start by creating a triangle sprite to be our enemies, like we created a square sprite for our Player. Once you make the sprite, drag it into the scene, and rename the object to “Enemy”. Also create an “Enemy” tag and layer, and add both to the Enemy object. Move the enemy prefab so that it is not touching the player object - set it to position (1, -1.5, 0). Finally, create an Enemy prefab by dragging the object into project files.
We are going to emulate the Goomba enemy from Mario - the enemy will walk in one direction until it hits an obstacle, and then turn around. Our Enemy needs a Rigidbody to move, so add that component to the Enemy object. Like the Player Rigidbody, freeze the rotation in x, y, and z, and freeze the z position.

Click and drag the Platform prefab into the scene twice. Scale both of these to be (1, 10, 0), so that they’re tall. Position one at (-5.5, 2, 0), and the other at (5.5, 2, 0). These will prevent the enemy from running off for now.

Now we're going to script some movement. Add a script object named "EnemyMovement":

1. Since the enemy can turn around, we are going to create a representation of the direction. We can use any number of things to represent this; enum, int, char, bool, etc. For readability, we recommend using enums. So, before Start, create a public enum Direction with two values: left and right.
2. Underneath the enum, create a public Direction direction. In the inspector, set this value to right for now.
3. Underneath your public direction variable, create a public float speed variable. In your inspector, set the value to 1 for now. Make sure to apply this changes to the prefab.
4. In Start, add an if statement. If the direction is Direction.right, do a Transform.Rotate(). Since we only want to rotate -90 degrees around the z axis, the parameters will be (0, 0, -90). If the direction is not Direction.right, it's Direction.left, rotate around z by 90. Also if it is left, multiply the speed value by -1 to make it negative. Add this in an else statement below the if statement. This will rotate the triangle enemy so that its "nose" points in the direction it will be moving.
5. Before Start, create a non-public variable Rigidbody rb. In Start, do a GetComponent<Rigidbody> to get the enemy rb to move it around.
6. In Start, set rb.velocity to be equal to a new Vector3 with values (speed, rb.velocity.y, 0).

By the end of these steps, your script should look like this.

Save, and go back to the editor and hit play. You might notice we have a problem - the enemy isn't moving! That's because it's being stopped by friction. There are two ways to fix this - one is to set rb.velocity every frame, like we do for the player. We're going to do it the second way - making the Enemy and Platform frictionless.

To make these objects frictionless, go to your project files, and hit "create". Go to Physic Material. Name this material "Frictionless", and in the inspector for the material, set both friction parameters to 0. Now, in the Project Files tab, go to the Enemy, Platform, and Player prefabs, and for each Box Collider, set the material to "Frictionless". By editing the prefab this way, you won't need to hit "apply" to apply the change to all the instantiated Platform objects. Now that you've set the Physics Material, the Enemy should move.
However, there's another problem. You might have noticed that if the enemy runs into a wall, it gets stuck indefinitely - we want our enemy to turn around. So let's implement that using Raycasts.

In your EnemyMovement script:

1. In Update, add an if statement. This if statement will be a Physics.Raycast(). Our starting position will be transform.position, our direction will be transform.up, and our max distance will be 0.6f.
2. Add curly braces after the if statement, and inside the if block, call Transform.Rotate(). The parameters for this should be 0 for x and y, and 180 for z - we want our enemy to flip around.

3. Multiply the speed by -1, and set it to that value.
4. Using that speed as the x parameter, rb.velocity.y for y, and 0 for the z parameter, set rb.velocity to a new Vector3.

After these steps, your code should look like this.

Save the file, go back to the editor, and hit play. Now, the enemy should go right, hit a wall, and turn around. However, there is a problem with this - when the enemy approaches the Player, the enemy just turns around, too! We don't want this behavior, since we *want* enemies to touch the player, and later, kill the Player to add danger. This is a simple fix, we'll add a layer mask so that the Player only turns around from running into a Platform.

In your EnemyMovement script:

1. Underneath your non-public Rigidbody declaration, add a non-public int layerMask.
2. After setting the Rigidbody velocity in Start, add a line to set layerMask to be equal to LayerMask.GetMask() with a parameter of "Platform".
3. In Update, modify your Physics.Raycast() to have a 4th parameter, layerMask.

Finally, by the end of these steps, your code should look like this.

Now, the Enemy will properly turn around, unless it hits the Player, and it'll stop. This is fine, since the Player will die as soon as it hits the Enemy, and the scene will be reset.

Note that if we were not to kill the Player upon collision, but instead give the Player some damage per hit, the Enemy would need to keep moving.
In this case, consider setting the velocity every frame in update. This also would eliminate the need to have your colliders be frictionless, in case you want friction. However, this still would have some wonky behavior - the Enemy would start pushing the Player around, in which case, you should add a knockback and iframes to the Player when it gets hit by an enemy.

Adding Player Death

Now, we want to add some danger to the Player, by scripting the interaction when a Player and Enemy collide.
Create a script called "EnemyInteraction" and add it to the Player:

1. Make a void OnCollisionEnter function - this will take a parameter of Collision collision.
2. Add an if statement, and within it, we're going to check the tag of the object we've collided with. To get the game object from a collision object, we can do collision.gameObject. To get the tag of a gameObject, we can do gameObject.tag. To check if the tag is "Enemy", we can do tag.Equals("Enemy"). So, this becomes collision.gameObject.tag.Equals("Enemy"), which will go in the if statement.
3. Now, we're going to do a very simple implementation of death - resetting the scene. Up at the very very top of the script file, are several statements - "using …;" To manage the

scene, we're going to need to add one of those "using …;" statements. Add "using UnityEngine.SceneManagement" to be able to access the SceneManager.

4. Now, we're going to want to reload the current scene. First, to get current scene we're playing, we can use SceneManager.GetActiveScene(). To get the name of the scene, we can do scene.name. And to load this scene, we just need to do SceneManager.LoadScene() and pass in the scene name as a parameter. To do this, we will do SceneManager.LoadScene() with SceneManager.GetActiveScene().name as a parameter.

When you are done, the script should look like this.

Now, save your script, return to the editor, and play your game. The game should now restart once you hit the enemy. This is our first step to creating a Game Loop.

Note that Game Loops are very important moving forward!
A Game Loop is important, because it allows you to test your game from start to finish, and see how it plays. We ***highly*** *recommend creating a game loop as early as possible, as it allows for testing, not only for devs to play as they create, but for others (playtesters) to come along, play your game, and provide feedback. Plus, a proper, standalone game loop is a "complete" game - it might not be the best game, but it is complete, which means there's only improving from there. Start with a small game loop, and gradually iterate until you add more to your gameplay.*

To finish your game loop, we recommend you add two more things. The first is a second type of death - the Player falling off the platforms. The second is a way for Players to reset the game on their own, at minimum, pressing R to reset. To get a little more advanced for this tutorial (and this is something you should aim for in Game Jams, or in any other game you make on your own), you can add UI elements to reset your game with an on screen button, or in a menu. This allows people to play your game again, or for others to get a chance to play your game!

Setting Up The Camera

When you play most platformers you don't expect to stay on one screen - you expect the level to be larger. In Unity, the camera is also a game object, which means we can add script and components to customize and control it. Create a new script called CameraController. This will control the positioning of our camera.

If you look at the uploaded script you can see it has a function called GameObject.Find("Player"). This does just as it says, and gets a reference to our single Player object. From here we can get the Transform information of the Player and set our camera's position directly to it. Attach this script to the Camera game object in your scene and try it out! Make sure to make the platform longer so you have room to move around.

This script shows how you could add more functionality to the camera. We have two public variables, the first which lets us set the time it takes to lerp or linear transform from the current camera position to the current player position. This can be used to add a smoothing follow to the camera as opposed to snapping to the player's transform. Additionally, the camera can lerp to a different position such as a waypoint. This could be used to scroll over a level. The second variable is whether or not we will use lerping.

We use a coroutine as a timer which lets us move the camera over the lerp. Since lerps sets the position at a fraction of the position between the given points if we are at 5 seconds into our 10 second timer, then the position will be 50% of the way between the start and end point. The lerping variable makes sure we finish the lerp before starting another coroutine which would become a mess really fast otherwise.

In update we check with movement option the camera is set to using. Additionally we add the offset between the camera and player from the start of the game scene which allows us to apply the difference in the z direction so the camera can be far enough back to see the player.

The quick and dirty way of doing this would be to make the camera a child of our player object and thus be locked to it's transform in the same way. This is fine for a simple project, however, it is not a modular approach and by using a script you are able to feature fill your camera with the ability to move smoothly as the player traverses the level or move if you want to add cutscene capabilities. Another good example would be in the game Spelunky where you can use the camera to look around you and move it a bit off the position of the player to see if you're jumping into spikes or falling just far enough to take damage.

Next Steps

We highly recommend you create incremental builds of your game: https://docs.unity3d.com/Manual/PublishingBuilds.html. This is how you can share your game with others - as long as you share the right folders/files, anyone can play on their machine.

There are also several ways you can improve the game! Expand the level to be more interesting gameplay wise - drag in more Platform objects into the scene and change the scale and position. You can also tweak the public variable values to be more fun. You can even think of your own mechanics to add to the game. Lastly, we encourage you to add your own sprites, sound, and music to the game. Experiment with it! Change around colors and effects. Add animations. See what you can do to spruce up the game!