Platformer Project

by PLAYER TWO

Thanks for your support!!!

playertwopublisher@gmail.com

https://discord.gg/THjKHVj5DA

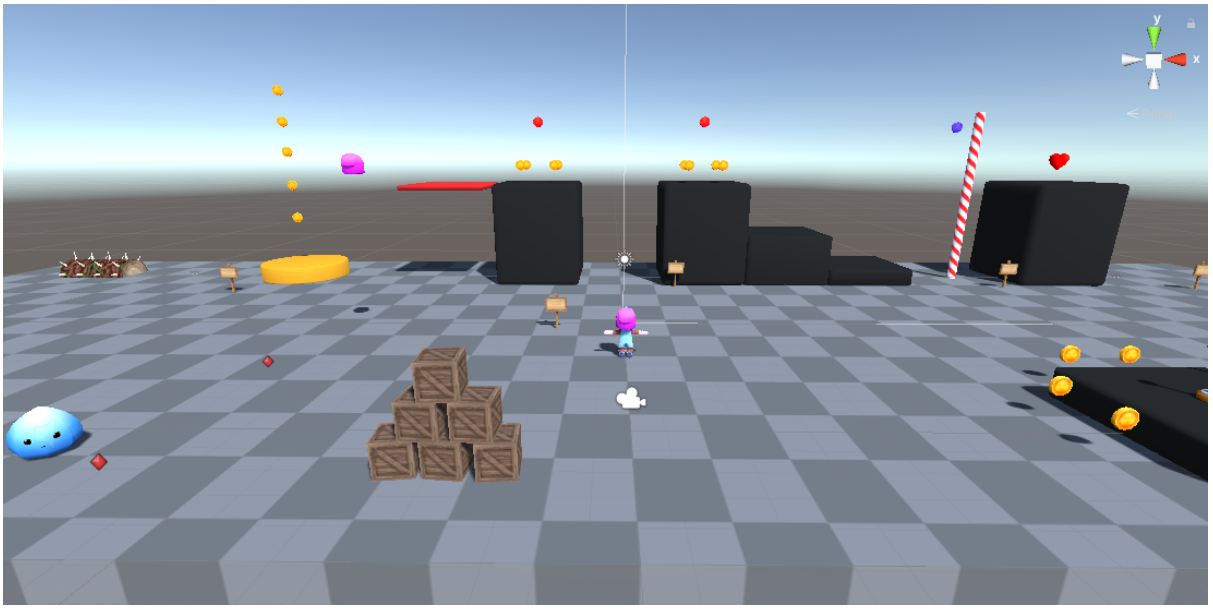
Summary

Hi, there! Welcome to the Platformer Project documentation!!!

Quick Start

To open and start using the asset, make sure to follow these steps:

- Download Unity version **2019.4.0f1** or higher;
- Create a fresh 3D project and open it;
- Download the asset through the Package Manager and Import it;
- Open the SamplaScene inside the following folder: "PLAYER TWO/Platformer Project/Examples/Scenes/SampleScene";
- Hit the Play button and start to mess around.



The Platformer Project doesn't make use of any third-party package, but it's fully compatible with Cinemachine, Post Processing, TextMeshPro, and other packages that you may like to use. It's also compatible with HDRP and URP.

If you need any support, email me at **playertwopublisher@gmail.com** and make sure to include the invoice number on the email's body. If you prefer, you can also join my Discord server **https://discord.gg/THjKHVj5DA**. I'll be glad to help you!

Overview

The Platformer Project is a 3D platform game template inspired by classic 3D platform games from our childhood. The asset was built to help other developers like me to make their 3D adventures. The codebase follows clean code and other engineering principles to deliver a highly scalable and professional toolset.

For the sake of keeping the asset easy to use and lightweight, I've decided to implement only core features of general platformer games. To make your life easier, the asset makes use of callback patterns and Unity Events, so you can easily trigger your scripts through the inspector. It also takes advantage of Scriptable Objects to control variables.

With the asset in hand, you'll be able to quickly develop and distribute your 3D platform games for all different platforms with minimal effort. Even though it's easy to use and tweak, it's still required for you to have some degree of acknowledgment about programming and general Unity usage, especially if you want to make changes to the asset.

Features

The Platformer Project contains many features, from complete Player movement to Enemy AI, and it's also bundled with general-purpose scripts. The main features from the asset are the following:

- 3D Movement:
  - Mobile Support;
  - Multi-jump;
  - Coyote Jump threshold;
  - Ground Snapping;
  - Slope Sliding.
- Wall Drag:
  - Slide down a wall;
  - Jump away from the wall.
- Swimming:
  - Dive;
  - Jump from water.
- Climbing:
  - Move up and down poles.
- Springs (jump boards);
- Floating Fields;
- Collectables;
- Checkpoints;
- Enemy AI (killed by jumping);
- Tweakable Hazards;
- Pushable Objects.

Content

After installing and importing the asset, all the content from the asset will be placed in a folder called “PLAYERTWO/Platformer Project”. Inside of this folder, you'll have two more folders called Examples and Scripts.

The Examples folder contains all assets that made the Sample Scene work, like models, textures, materials, etc. This folder also includes all sample prefabs, like the Player, Collectables, etc.

The Scripts folder contains the brain of the asset. Here you'll find all the movement scripts, states, managers, and miscellaneous. Each subfolder represents an object, and each of them has object-specific scripts.

When importing the assets, do not try cherry-picking stuff you want, especially if you're not sure about how the asset works as a whole. Either import both Examples and Scripts or only the Scripts. Incorrectly importing the assets may end up breaking functionalities.

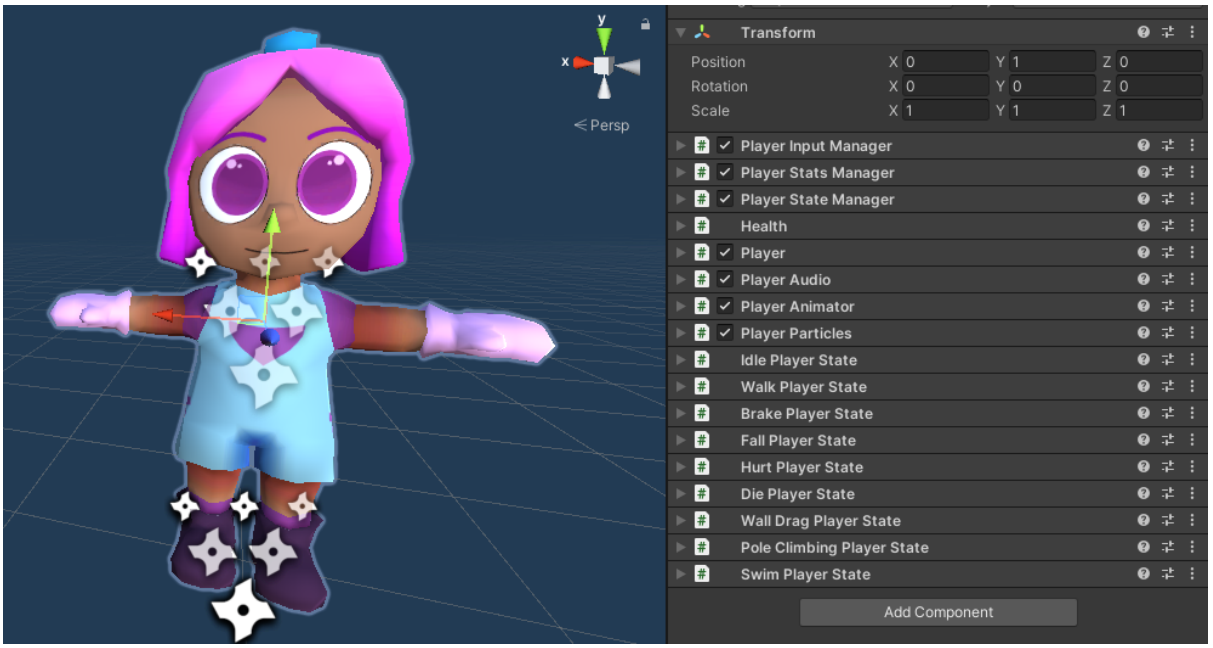
Objects

Objects are how I decided to name anything that takes a relative amount of complexity to work, most times needing more than one script file. Everything else that takes just a single simple file to work I classify as Miscellaneous. The Objects from the asset are the following:

Entity

The Entity is the foundation of Players and Enemies, where both extend from the Entity base class. The Entity is nothing more than an abstraction to handle the Character Controller's movement and collision detection. Each Entity has its Stats, States, and State Manager, so the behavior is driven by an implementation of a Finite State Machine.

Player



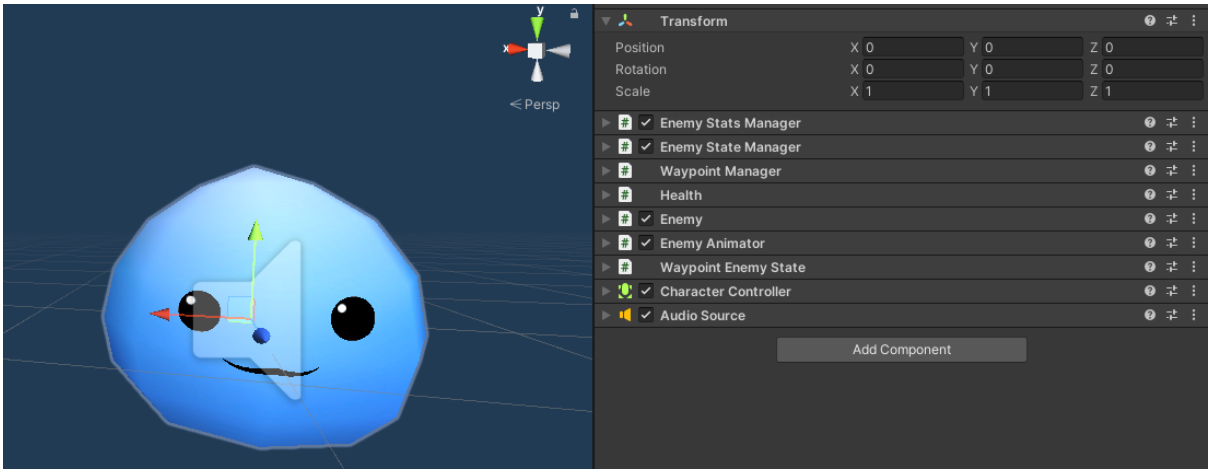
The Player is an implementation of the Entity where its movement is driven by user input. It also implements the Entity States for platforming movements. You can find all States inside the "States" folder.

You may have noticed that the Player script itself has no variables. Its variables are provided by the Player Stats Manager, which is a list of Player Stats (Scriptable Object). This way, you can easily replace all the values with an entire new Scriptable, making it possible to create power ups, for example.

If you want to change the state's behavior, you can either edit them directly in the "PLAYER TWO/Platformer Project/Scripts/Player/States" folder or create new ones inheriting from the Player State abstract class.

The Player also contains scripts for audio, particles, and animations. They all just read data from the Player instance or listen to Unity Events to do something else, like playing an audio clip when the Player jumps.

Enemy



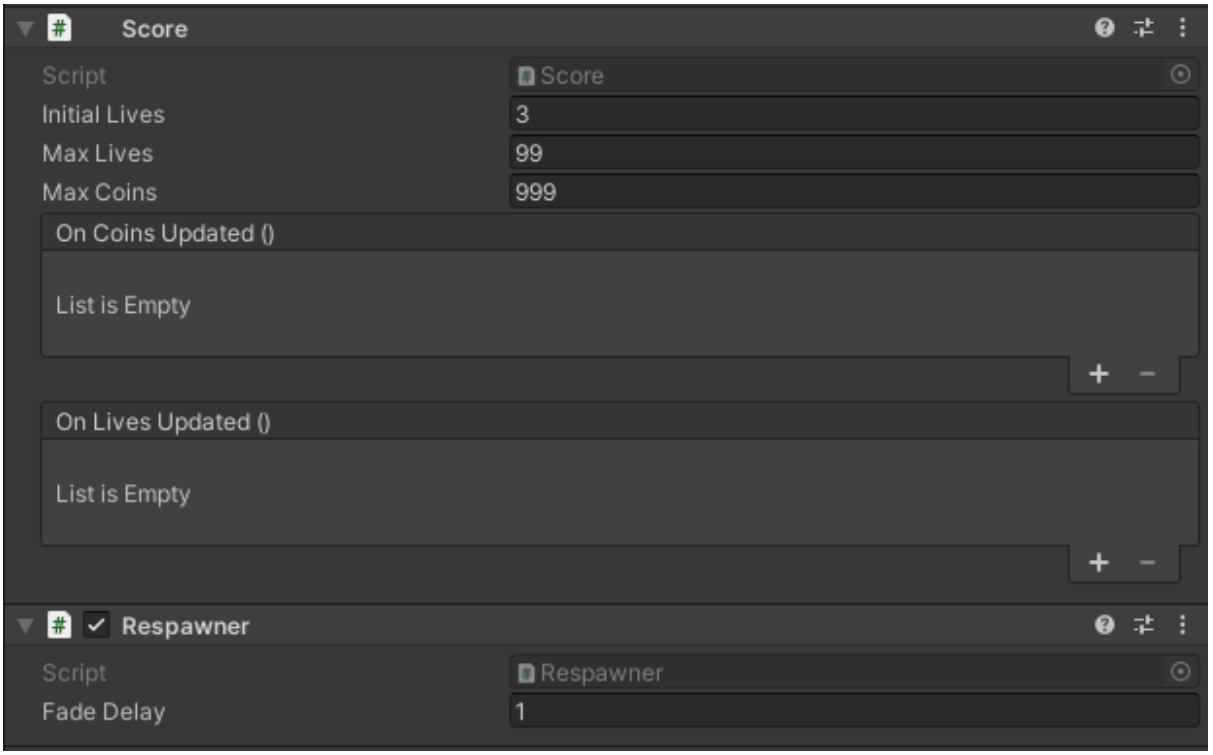
The Enemy is also an implementation of the Entity, but its movement is driven by AI. Like the Player, its variables are controlled by a manager with the same purpose, the Enemy Stats Manager. Everything that applies to the Player also applies to the Enemies. The only difference is that it contains more callbacks provided by its Player detection mechanisms.

In the folder "PLAYER TWO/Platformer Project/Scripts/Enemy/Enemies" you can find examples about how to implement simple AI behaviors. So far, enemies can stand still, walk for waypoints, and seek the Player. You can also code behaviors using the

inspector to invoke state changes through Unity Events. Take a look at how the Slime enemy plays an audio clip at “PLAYER TWO/Platformer Project/Examples/Prefabs/Entities/Slime” prefab, you can also trigger state transitions or any other behaviors with the Unity Event technique.

To create new States it's virtually the same thing from the Player, but you must inherit from the Enemy State abstraction instead.

Game



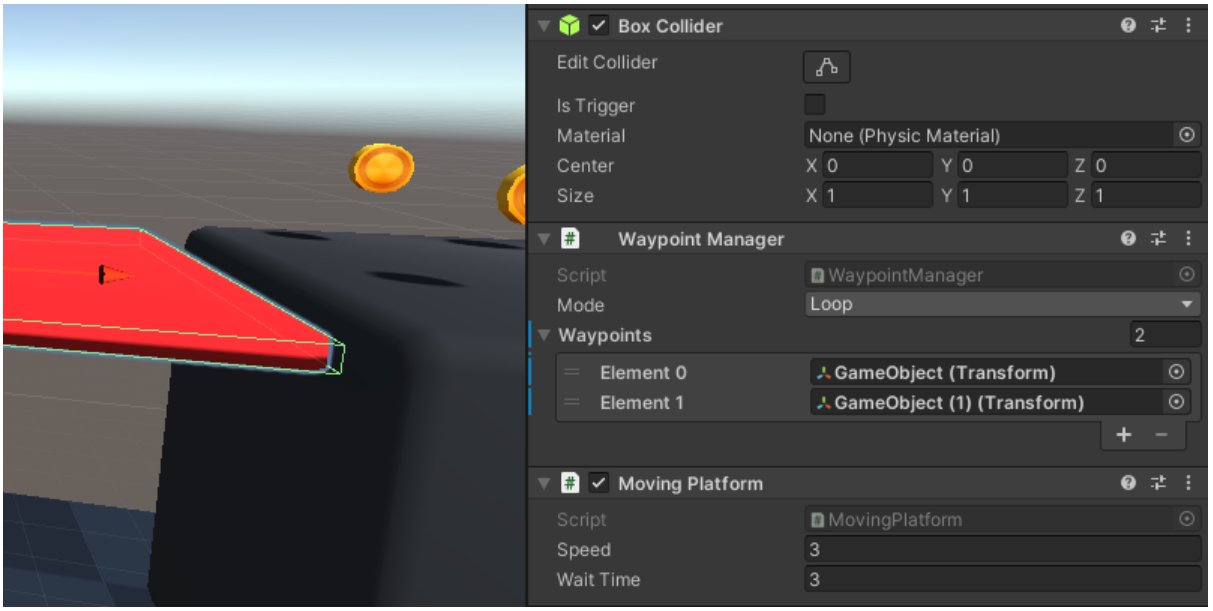
The Game controls the score and scene behavior, like counting the number of collected coins, and respawning the Player when it dies. It also contains a list of tags, so you can easily reference them without remembering how they're spelled.

You can use the Score Controller to change the Score data from other game objects, without needing to implement a "Get Score" every time. There's an example of how it works on collectibles prefabs.

Gamepad

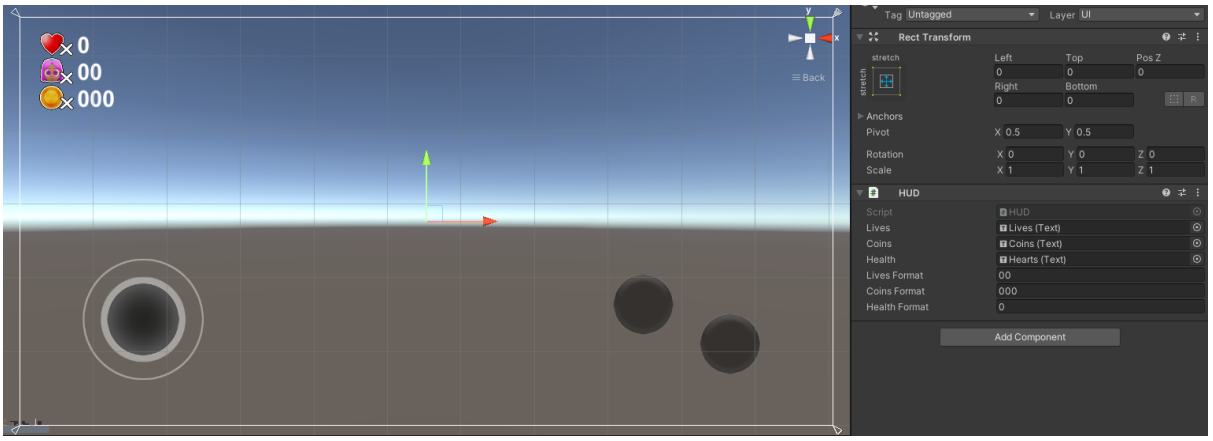
The Gamepad is an Input Manager, simple as that. The Gamepad is based on Unity's Cross-Platform Input, and it's an easy way of getting input data from a mobile device through interactive UI. Except if you want to replace the Input System, there's nothing to change here.

Waypoint



The Waypoint is what is used to move platforms and Enemies. It contains a Waypoint Manager, which dictates how the waypoint sort will behave. The loop mode can be "Loop", to move back to start when the waypoint finishes, "Ping Pong", to reverse the order of the sorting or "Once", to stay in the last waypoint after it finishes.

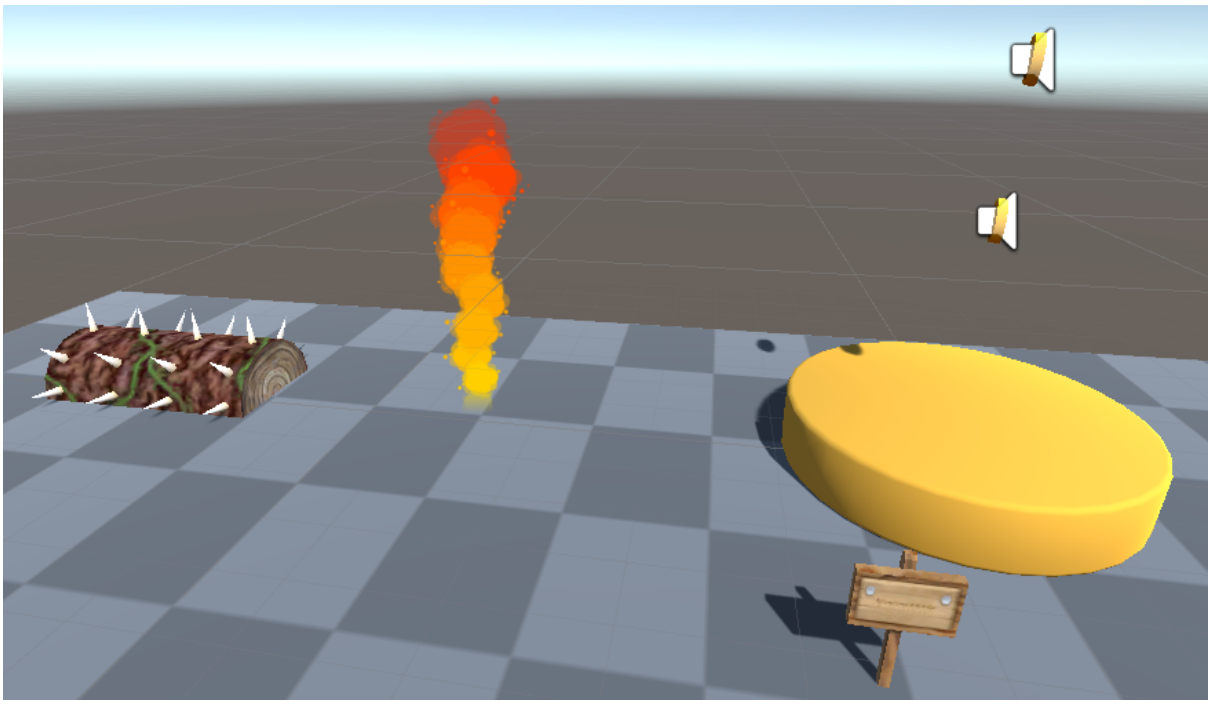
UI



The UI so far only implements the HUD, which displays the Score data for the Player. It also keeps track of Player's Health, represented by the heart counter.

To enable the mobile virtual gamepad you just need to activate its Game Object in the Canvas, to do that, go to the Hierarchy panel, click on “Canvas/Virtual Gamepad”, then toggle the Game Object on the inspector panel.

Misc



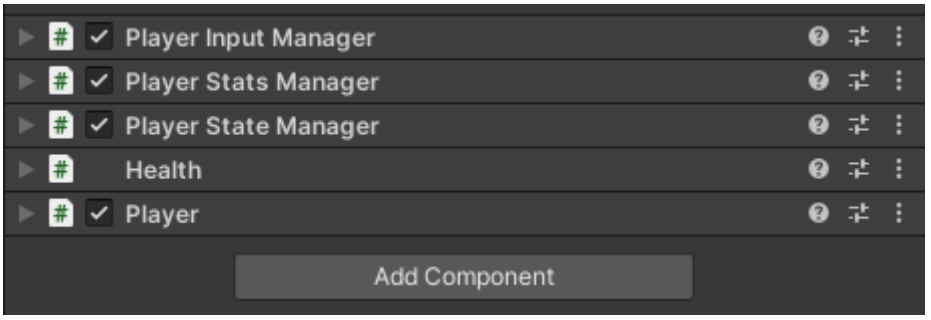
The Misc are just a collection of random objects. The functionality of most of them is very self-explanatory. Some of them detect the Player's contact and react in some way, others are just for the Player to contact with, and there are also general-purpose movement scripts. Here you can find the Checkpoint, Health, Collectables, etc.

Tutorials

Creating a new Player

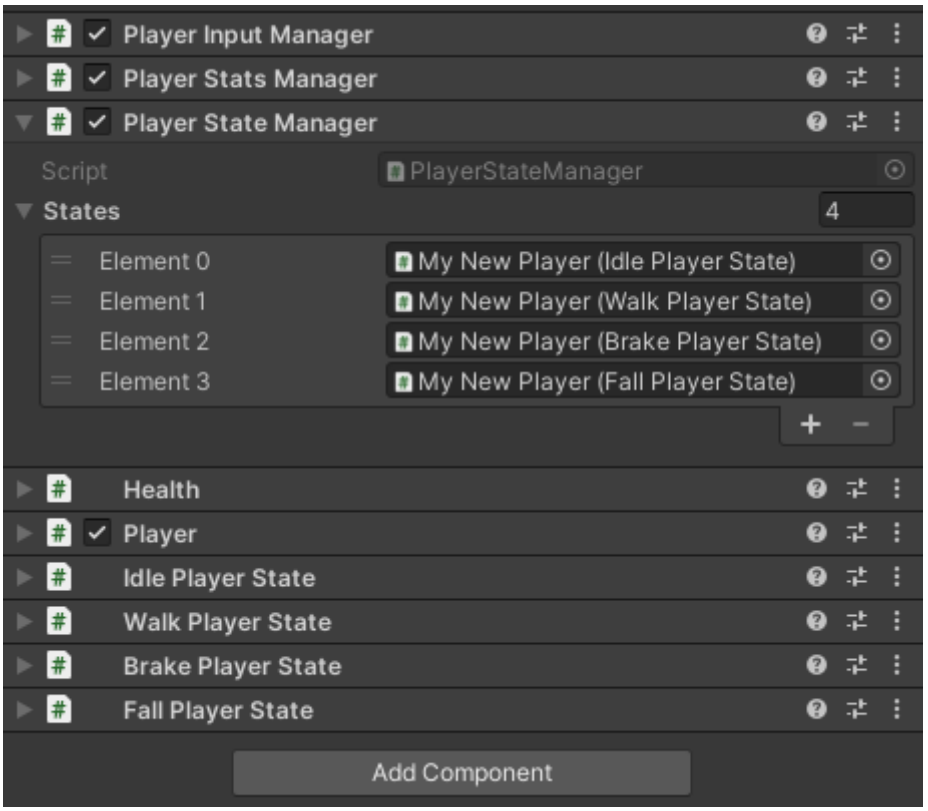
To create a new Player, first of all, you'll need to add a new Empty Game Object to your scene, which will be used as the root of all your Player's components. To do that, you'll need to right-click on the Hierarchy tab and then click on "Create Empty". Rename this new Game Object to whatever you want, like "My New Player".

The main component of any Player is the "Player" script. To add it, select your new Player Game Object, click on "Add Component" on the Inspector tab, and then search for "PLAYER TWO/Platformer Project/Player/Player". After adding the Player script. After that, other components will be automatically added to your new Player Game Object, and they are Player Input Manager, Player Stats Manager, Player State Manager, and Health. Without those, the Player will not work properly.



The Player Input Manager will be the source of every input used within any of the Player's actions, like moving around, jumping, etc. This component works by creating an interface with Unity's default Input System. If you want to communicate with another Input Manager that is not the default one, all you need to do is create a new Player Input Manager script extending from it and override its methods. The names for the inputs provided on the inspector must match the ones on Unity's Input Manager. You can make a second Player by assigning names of inputs that are provided by other Joystick numbers. You can also stop the Player from moving by toggling the Active button.

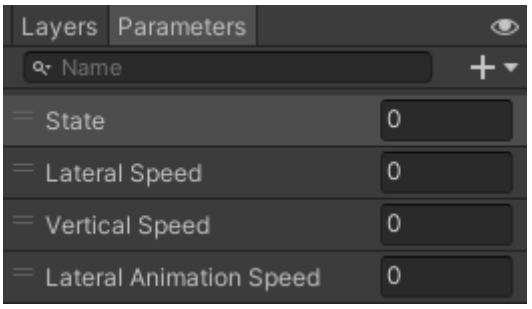
The Player Stats Manager is responsible for controlling all the Player Stats. Player Stats is a Scriptable Object which declares all the variables used by the Player, like how fast it can accelerate and how high it can jump. The Player Stats Manager can carry many Player Stats, but only one will be in use. To create a new Player Stats object, right-click the project tab, click on "Create/PLAYER TWO/Platformer Project/Player/New Player Stats" and then rename it to whatever you want. All variables will be set to their default values. You can then assign the newly created Player Stats to the Player Stats Manager slot, and it will be ready to use.



The Player State Manager is where all the magic happens. The Platformer Project implements a Finite State Machine approach to control the Player's actions. It also provides an API to handle state transitions. Each Player State is a component that represents a specific action, like idling, walking, swimming, and so on. The Platformer Project already provides a bunch of Player States for you, so there's no need to put your hands on any code if you're happy with the current behaviors available for the Player. To make the Player do something, you will need to add a Player State to it and assign it to the Player States Manager. You can see a list of available Player States on "Add Component/PLAYER TWO/Platformer

Project/Player/States", you can pick any Player States you want, but the basic ones are Idle Player State, Walk player State, Brake Player State, and Fall Player State. You can create a new Player State by declaring a class that inherits from "Player State" base class. As mentioned before, do not forget to assign the Player States to the Player State Manager component.

After following the steps above, you'll have a fully configured player that can move around and jump. However, there's no graphical representation so far. If you want to display something and start playing, you can add a Capsule Game Object as a child of your Player, make sure its position and rotation are set to zero, and remove its capsule collider. To add an actual model, all you need to do is import it to your Assets folder, and then drag and drop it to your new Player Game Object. To play its animation, just add the component "PLAYER TWO/Platformer Project/Player/Player Animator" and assign your model Game Object to the "Animator" slot. Make sure to assign an Animator Controller to your model's Animator component. The Player Animator will expect your Animator Controller to have the parameters "State" (Int), "Lateral Speed" (Float), "Vertical Speed" (Float), and "Lateral Animation Speed" (Float). It's up to you how to organize your animation transitions. Lily's animations work by transitioning from "Any State" to the specific animation related to its current Player State, using the "State" parameter, which corresponds to the current Player State index from the Player State Manager.



It's done!!! Now you have a basic functional Player in your game, just turn it into a prefab by dragging and dropping it into your Assets folder. You can improve it even more by adding the components Player Audio, Player Particle, and Player Lean. Also, make sure to check how the built-in character, Lily, works. You can find its prefab in the folder "PLAYER TWO/Platformer Project/Examples/Prefabs/Entities".