About

The **Platformer Project** is a 3D platform game template, made with Unity and written in C#, inspired by classic 3D platform games from our childhood. The Asset was built to help other developers like me to make their 3D platform adventures. The codebase follows the clean code principle and other software engineering principles to deliver a highly scalable and professional toolset.

To keep 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 uses callback patterns and Unity Events, so you can easily trigger your scripts through the inspector. It also takes advantage of Scriptable Objects to control data.

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 this Asset is easy to use and tweak, it's still required for you to have some degree of acknowledgment about programming with C# and general Unity usage, especially if you want to make more profound changes to the project.
> https://learn.unity.com/

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 of the Asset are the following:

General Features

- Mobile Support;
- Humanoid Rig support (share animations);
- Save/Load (Binary, JSON, or Playerprefs);
- Multiple save slots support;
- 3 stars, coins, and best time saving;
- Life system w/ level respawn/restart;
- 3D Dynamic Camera;
- Level Checkpoint system;
- Enemy AI and Hazards;
- Surface Footsteps;
- Cute and animated low poly models.

Gameplay Features

- Full 3D Movement;
- Smooth character leaning;
- Pick Up and Throw objects;
- Spin Attack ability;
- Air Spin Attack;
- Heavy Stomp ability;
- Air Dive ability;
- Multiple jump ability;
- Running ability;
- Coyote time threshold (safer jumps);
- Wall slide and wall jump;
- Ledge Grabbing;
- Ledge Climbing;
- Crouch and Crawl (resize collider);

- Swim and Dive ability for water levels;
- Climb Up/Down poles;
- Moving Platforms;
- Falling Platforms;
- Springs (jump boards);
- Gravity Fields;
- Collectables;
- Hidden Collectables;
- Bouncy Collectables;
- Item Boxes;
- Destroyable Objects;
- Pushable Objects.

> This project is in active development, with more features to come!

Buying the Asset

The Asset is available on Unity Asset Store: https://assetstore.unity.com/packages/slug/206584.

> For security reasons, I will not sell it out of the Asset Store, and I have no control over individual discounts.

Quick Start

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

- Download Unity version **2020.3.20f1** or higher;
- Create a new 3D project and open it;
- Download the asset through `Window > Package Manager > Packages: My Assets` and import it;
- Open the `SampleScene` inside the following directory: `PLAYER TWO/Platformer Project/Examples/Scenes` ;
- Hit Unity's Play button and start to mess around.

> The Platformer Project isn't bundled with Unity packages like Cinemachine or TextMeshPro, but they are compatible.

> When using any scriptable render pipelines, you may need to change the shaders of all materials from `PLAYER TWO/Platformer Project/Examples/Materials` since they all use custom surface shaders.

> If you import the asset into an existing project, your project settings will be overridden, and this happens because the Platformer Project needs specific configurations from the Input System, Physics, Tags, and Layers. These settings are not a requirement for this particular project but general Unity complete project importing.

File Structure

After following the quick start and correctly importing the asset, you'll find a new folder called `PLAYER TWO` , with the `Platformer Project` folder inside it.

You'll find two subfolders in the Platformer Project folder, `Examples` and `Scripts` .

The **Examples** folder contains everything related to this asset demonstration, like audio files, materials, textures, models, etc. Here you'll also find the demo scenes and demonstration prefabs, so you can use those as a base/reference to make your original stuff. If it's the first time you're using the asset, import this folder and closely examine how the demo scenes work. All assets are organized by their respective folder and are self-explanatory, e.g., the Audios folder contains sound effects and music files, and the Prefabs folder contains all the Game Objects that compose the demo scenes.

The **Scripts** folder contains all the C# files that give life to the asset, and they are separated by concern, so there's no UI logic out of the UI folder nor Enemy stuff inside the Player folder. Despite the Misc folder, all others are self-explanatory, but the Misc folder is just a collection of classes that don't need complex logic to run.

> **DO NOT** override anything from these folders, or else you'll be unable to update the asset without losing your changes.

> **DO NOT** delete anything from the Scripts folder unless you know what you're doing, or else the asset will not work correctly.

Making Your Game

Once you've imported the asset into a new project, you're free to do whatever you want with its content. However, there are some best practices you must follow to be still able to download new updates without losing your work. Also, keep in mind that some updates may cause breaking compatibility, so make sure to check the changelog beforehand.

> I highly recommend you to use a versioning control system such as Git. With these kinds of tools, you'll be able to manage your project version and discard undesired changes in case something goes wrong, e.g., reverting and breaking compatibility updates.

As mentioned in the File Structure session, **DO NOT** override or delete anything from the Platformer Project directory unless you know exactly what you're doing. Instead, you must save all game assets in the root Assets folder or a subfolder inside the root Assets folder. If you're familiar with the code base and don't want anything from the demo scenes, you can uncheck the `Examples` folder when importing the asset, and it's not necessary to run the project.

> After tweaking prefabs from the demo scenes, save them as a new original prefab **outside** of the PLAYER TWO folder. https://learn.unity.com/tutorial/prefabs-e#

> If you want to tweak existing scripts, you must create a new script, **outside** of the PLAYER TWO folder, that inherits from the one you want to change. Most methods from the classes were declared as `virtual`, so you can override them on your custom classes. https://learn.unity.com/tutorial/overriding#

Default Inputs

To move the player around, you can use a keyboard or an XBOX controller, the keys are mapped as such:

- Walk: “WASD” or XBOX Left Thumb;
- Run: Hold the “Left Shift” key or XBOX “X”;
- Dive: Hold the “Left Shift” key or XBOX “X”;
- Jump: Press the “Space” key or XBOX “A”;
- PickUp/Throw: Press the “G” key or XBOX “Y”;
- Crouch/Crawl: Hold the “Right Mouse” button key or XBOX Left Trigger;
- Spin Attack: Press the “Left Mouse” button key or XBOX Right Trigger;
- Stomp Attack: Press the “E” key or XBOX “Right Bumper”;
- Air Dive: Press the “Right Mouse” button or XBOX “Left Trigger”;
- Release Ledge: Press the “Left Shift” key or XBOX “X”;
- Pause: Press ESC or XBOX Start.

> You can change the button that correspond to a given action through the Player Input Manager.

> You can change the keyboard mapping through **Edit > Project Settings > Input Manager**.

Getting Help

> Make sure to read the documentation before asking for help!

If you need any support, feel free to contact me at playertwopublisher@gmail.com.

I’m also available on PLAYER TWO’s Discord server https://discord.gg/THjKHVj5DA.

Frequently Asked Questions

The console shows errors when trying to load my new level

When creating a new Level scene, make sure to add its scene to the Build Settings and that its scene’s name on the levels list **matches exactly** its scene’s name from the Build Settings.

Can you add this specific feature?

I consider requests, so feel free to email me or send a message to the discord server about the feature you want and why you need it.

Remember that some requests are too specific and don't fit into a general-purpose asset.

Will you add online multiplayer?

I don't want to add online multiplayer because it relies too much on third-party packages and particular implementations.

In short, I don't want to make this asset too complex and bloated.

When will the next update be out?

Unfortunately, I can't provide an ETA for updates since this asset is not my only occupation.

It depends on my busy schedule, but updates generally take about a month.

Can I contact you to work for me?

No, I'm sorry. I already have a job, so the asset is already being made in my free time.

How can I support you?

If you already bought the asset, feel free to leave a positive review on the Asset Store; it helps a lot!

Entity

The **Entity** is an abstract class that serves as a base for objects that move with Character Controller, like the Player and the Enemy, handling the movement and collision detection. It also provides general-purpose properties and methods, like decomposed velocity properties, raycasting abstractions, collider rescaling, and much more.

The built-in available entities implementations are the Player and the Enemy.

Entity Input Manager

The **Entity** Input Manager abstracts the interaction with the Gamepad class, providing an easy-to-use way of reading inputs from enums, which are more intuitive to read from the inspector instead

of relying on strings. It also provides helper methods to read directional inputs relative to the Camera direction and to convert axis float values into booleans. It's more beneficial for playable characters since AI doesn't need joystick input data as much.

> Camera relative inputs will require a camera tagged as "MainCamera" in your scene.

Entity State

The **Entity State** is the base class for all states. Using it, you have access to a runtime life cycle that helps develop complex and independent behaviors when used together with the Entity State Manager. It also exposes callbacks using Unity Events.

Entity State Manager

The **Entity State Manager** is an approach to the Finite State Machine pattern. It works by executing one Entity State at a time while triggering their entering and exiting events based on a specific transition. Unity's animation system uses a very similar approach. However, this one was made for a more code-based approach, with fast transitions and without relying on animations.

Entity Stats

The **Entity Stats** is a Scriptable Object that holds all the Entity's variables. There are no general variables so far, but it's open to possibilities.

Entity Stats Manager

The **Entity Stats Manager** is responsible for managing Entity Stats. With it, you can keep a collection of different stats and quickly change between them at runtime. It is beneficial when you want to change an entity's behavior quickly by replacing its variables, e.g., creating power-ups.

Entity Volume Effector

The **Entity Volume Effector** is a component used to change how the Entity will handle the velocity. It's helpful to create viscosity volumes that slow down the Entity movement but can also be used to speed it up.

Player

The **Player** is an implementation of the Entity component as an input-driven entity, so its behavior is dictated by the player's joystick commands. Players can stand still, move around, jump one or multiple times, swim, care for objects, climb poles, and so on. You can easily tweak variables using the Player Stats to create different behaviors.

> For more details about Entities, please read the Entity page.

Creating a new Player

To create a new Player, drag and drop the Player script to an empty Game Object or click on the "Add Component" button, and then select `PLAYER TWO/Platformer Project/Player/Player` . All the minimum required components will be added, and they are (you can also add custom or third-party components):

- Player Input Manager;
- Player Stats Manager;
- Player State Manager;
- Health.

The minimum required components are enough to make your Player "exist", but of course, they're not enough to allow it to move around. To be able to move, you'll need to add some Player States to your Game Object. The minimum states you'll need for the basic moves are the following:

- Idle Player State;
- Walk player State;
- Brake Player State;
- Fall Player State.

If you want to have a mesh that plays animations, read the Player Animator section.

There are also other non-required components that add even more details to players, so feel free to add any of them:

- Player Audio;
- Player Particles;
- Player Lean;
- Player Level Pause;
- Player Spin Trail.

The Player component will automatically add a Character Controller to its Game Object. Still, if you need to tweak its settings by hand, like the collider's size or its center, you can manually add the Character Controller, and the Player will use it instead.

Player Input Manager

The **Player Input Manager** inherits from Entity Input Manager.

This component also implements methods to return action-specific input values, so you don't need to change which button corresponds to a given action every time it's read. You can also remap these actions with corresponding buttons through the Inspector.

> The player will need camera relative input directions, so you'll need a camera tagged as "MainCamera" on your scene.

Player Stats Manager

The **Player Stats Manager** inherits from Entity Stats Manager.

> The stats manager must have at least one active stats.

Player Stats

The **Player Stats** inherits from Entity Stats, and contains all of the player's variables; like acceleration amount, top speed, jump height, and so on.

> To tweak the variables, just left-click the asset corresponding to the Player Stats you want to edit, the variables will show up in the Inspector.

Creating new Player Stats

To create a new **Player Stats** asset, right-click the project tab, then click on `Create/PLAYER TWO/Platformer Project/Player/New Player Stats` and finally 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.

Player State Manager

The **Player State Manager** inherits from Entity State Manager.

> The state manager must have at least one active state.

Player States

The Asset comes with many built-in Player States. You can add states to your Player by dragging and dropping the corresponding scripts into the player Game Object, or by clicking on “Add Component” and selecting one state from `PLAYER TWO/Platformer Project/Player/States`. You can also create your own Player State.

Air Dive Player State

The **Air Dive Player State** allows the Player to perform a dive while on air, quickly building up speed.

Brake Player State

The **Brake Player State** is responsible for the quick deceleration when the player turns the joystick in the opposite direction.

Crawling Player State

The **Crawling Player State** moves the Player around while it's also crouched.

Crouch Player State

The **Crouch Player State** keeps the Player crouched and applies friction to stop it if it's moving.

Die Player State

The **Die Player State** is used when the Player dies, applying gravity and friction without any other movement.

Fall Player State

The **Fall Player State** is responsible for providing air controller and gravity. Despite being called "fall", it's also used if the player is moving upwards.

Hurt Player State

The **Hurt Player State** is similar to the Fall Player State but without air control. Used when the player takes damage.

Idle Player State

The **Idle Player State** corresponds to when the Player is grounded and standing still.

Ledge Climbing Player State

The **Ledge Climbing Player State** is responsible for making the Player climb up from a ledge. It works by detaching the Player model from the actual controller, so make sure to reference the model in the "Skin" attribute from the Player component, or else your "climbing up" animation will look off.

Ledge Hanging Player State

The **Ledge Hanging Player State** dictates how the Player moves when hagging a ledge. If you want to climb up the ledge, make sure to also add the Ledge Climbing Player State to your state machine.

Pole Climbing Player State

The **Pole Climbing Player State** controls the Player's movement while attached to a Pole.

Spin Player State

The **Spin Player State** is responsible to handle its spin attack.

Stomp Player State

The **Stomp Player State** allows the Player to perform the Stomp Attack, also known as Butt Stomp.

Swim Player State

The **Swim Player State** controls the Player's movement while underwater.

Walk Player State

The **Walk Player State** handles the player's ground movement using the input direction.

Wall Drag Player State

The **Wall Drag Player State** dictates the Player's behavior when attached to a wall.

Creating new Player States

To create a new Player State you'll need to create a new script that inherits from `PLAYERTWO.PlatformerProject.PlayerState` and then override its methods.

```cs
using PLAYERTWO.PlatformerProject;

public class MyNewPlayerState : PlayerState
{
    protected override void OnEnter(Player player)
    {
        // Executed when this State is called.
    }
```

```
        protected override void OnExit(Player player)
        {
            // Executed when this State exits to another one.
        }

        protected override void OnStep(Player player)
        {
            // Executed at every Update if this State is the current one.
        }
    }
```

Transitions to this component can be made by either editing the state you want to transition from, or by calling the `Change<PlayerState>()` method of the Player State Manager. Please, take a look at the default states, found on `PLAYER TWO/Scripts/Player/States`, before trying to code your own.

The Custom Player States can be added to your Player just like any other state.

Player Animator

To add a mesh that can move around, you'll need a model with a humanoid rig (being humanoid is not required if you want to use your animations). After that, add your mesh as a child of your player game object, add the Animator component to it and create an Animator Controller and assign it to the controller property. Finally, add the Player Animator component to your Player and assign the animator of your mesh to the Animator property.

The Player Animator expects your Animator Controller to have the following parameters:

- State (int);
- Last State (int);
- Lateral Speed (float);
- Vertical Speed (float);
- Lateral Animation Speed (float);
- Is Grounded (bool)
- Is Holding (bool)

Lily's default animation controller works by transitioning from "Any State" to another by the current Player State Manager index. Animation for holding objects is done by using a Layer with an Avatar Mask that corresponds to the upper body. For complex animation, this solution might not be ideal, so feel free to organize your animation controller as you want, it'll not interfere with the player's behavior.

The Player Animator is a very simple component that only reads properties from the Player component and passes them to the Animator. If you want more customization, you can easily create your Animator handler component and read different values.

When adding your own animations, make sure to toggle the “Bake Into Pose” setting for all Root Transforms, or else your Player model will move away from its origin by its animation offset since the Player doesn’t use root motion. Please read more details about it at https://docs.unity3d.com/Manual/RootMotion.html.

> Working with animations in Unity can be very tricky. If you’re a beginner, please take a look at https://learn.unity.com/course/introduction-to-3d-animation-systems.

Player Audio

The **Player Audio** is a component that can be attached to a player to make it able to play some audio clips. Its implementation is straightforward and relies on listening for the Player script’s callbacks. You can, of course, make your component for playing different audios.

> The Player Audio automatically adds an Audio Source component, but you can also manually add one if you need to tweak it.

Player Particles

The **Player Particles** is a component responsible for playing particles. Its implementation is very similar to the Player Audio component, but it also keeps track of the player’s velocity. It won’t instantiate any particles at runtime, will just play or stop then, so make sure to keep your particles as a child of your player.

Player Lean

The **Player Lean** will make your player tilt when making sharp turns. It’s a nice touch to give some sense of weight when moving around.

Player Level Pause

The **Player Level Pause** is a component that allows your player to pause the game, as simple as that.

Player Spin Trail

The **Player Spin Trail** controls whether or not the spin attack trail should appear. It must be attached to a Game Object with the Trail Renderer component, and this Game Object must be a child of your Player. Also, make sure to attach a hand bone to the “Hand” property, or else the trail will be displayed in the wrong position.

Player Camera

The **Player Camera** is a component that you can attach to your camera Game Object to have a very basic camera movement. You can replace the default camera with Cinemachine, or any other camera system you want, just make sure to have a camera in your scene that is tagged as “MainCamera”, or else the inputs won’t work, and the console will display an error message.

Player Footsteps

The **Player Footsteps** is used to add footsteps sounds to the Player. It works by taking a given step interval from the Player movement offset, raycasting downwards, looking for the surface tag, and then playing one sound from the array corresponding to that tag. By default, the package is built with Grass, Wood, and Metal Surfaces. You can add new surface tags using the inspector, so please look at https://docs.unity3d.com/Manual/Tags.html.

Enemy

The **Enemy**, just like the Player, is an implementation of the Entity component, but instead of using inputs, it relies on AI movements. Players and enemies are virtually the same things under the hood, only their behaviors are different. 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.

> For more details about Entities, please read the Entity page.

Creating a new Enemy

To create a new Enemy, drag and drop the Enemy script to an empty Game Object or click on the "Add Component" button, and then select `PLAYER TWO/Platformer Project/Enemy/Enemy`. All the minimum required components will be added, and they are:

- Enemy Stats Manager;
- Enemy State Manager;
- Waypoint Manager;
- Health.

Enemy Stats Manager

The **Enemy Stats Manager** inherits from Entity Stats Manager.

> The stats manager must have at least one active stats.

Creating new Enemy Stats

To create a new Enemy Stats asset, right-click on the project tab, then click on `Create/PLAYER TWO/Platformer Project/Enemy/New Enemy Stats` and finally rename it to whatever you want. All variables will be set to their default values. You can then assign the newly created Enemy Stats to the Enemy Stats Manager slot, and it will be ready to use.

> To tweak the variables, just left-click the asset corresponding to the Enemy Stats you want to edit, the variables will show up in the Inspector.

Enemy State Manager

The **Enemy State Manager** inherits from Entity State Manager.

> The state manager must have at least one active state.

Adding new States

To add new states, drag and drop the script you want to your enemy inspector or click on the “Add Component” button, and then select the state you want from `PLAYER TWO/Platformer Project/Enemy/States` . Do not forget to assign the newly added state to the state manager “States” list.

States

This Asset comes with a few built-in Enemy States. You can program state transitions using the Inspector and the sight Events; e.g. the Enemy starts in idle state, once a Player is spotted (On Player Spotted), change the state to the follow State, but when the Player scape from the sight (On Player Escaped), make a transition back to the idle state.

Follow Enemy State

The **Follow Enemy State** will move the Enemy towards the closest Player position while applying gravity to it.

Idle Enemy State

The **Idle Enemy State** will make the Enemy stop and stand still, also applying gravity.

Waypoint Enemy State

The **Waypoint Enemy State** makes the Enemy move through waypoints, using the Waypoint Manager settings.

Creating new States

To create a new Enemy State you'll need to create a new script that inherits from `PLAYERTWO.PlatformerProject.EnemyState` and then override its methods.

cs

```cs
using PLAYERTWO.PlatformerProject;

public class MyNewEnemyState : EnemyState
{
    protected override void OnEnter(Enemy enemy)
    {
        // Executed when this State is called.
    }

    protected override void OnExit(Enemy enemy)
    {
        // Executed when this State exits to another one.
    }

    protected override void OnStep(Enemy enemy)
    {
        // Executed at every Update if this State is the current one.
    }
}
```

> Please, take a look at the default states, found on `PLAYER TWO/Scripts/Enemy/States`, before trying to code your own.

Enemy Animator

The **Enemy Animator** is a component responsible for controlling the animations of the Enemy. To make it work, you'll need an animated model child of the enemy component with an Animator component attached to it, do not forget to add an Animator Controller to it. After that, add the Enemy Animator to your enemy component and assign the model Game Object to the "Animator" property on it. This component expects the animator controller to have the following parameters:

- Lateral Speed (float);
- Vertical Speed (float);
- Health (float).

Slime's default controller handles transitions from the "Any State" based on its lateral speed and the amount of Health it has. All the variables are read from the Enemy script. If you need to access more variables, I recommend you code your animator.

> Working with animations in Unity can be very tricky. If you're a beginner, please take a look at https://learn.unity.com/course/introduction-to-3d-animation-systems.

Game

The **Game** is responsible for managing levels and the gameplay progress.

Game Data

The **Game Data** represents the game in the data layer, which means a serializable class that can be read and written from the memory when dealing with save and load.

Game Level

The **Game Level** is a class that represents a level with its general characteristics, which is a name, corresponding scene name, a description, an image, etc. It serves as a reference to the game level management and can also be used to create level selection menus. Game Levels at runtime will hold the amount of coins and stars collected, as well as the best completion time.

Game Loader

The **Game Loader** is used to load scenes asynchronously in an easy way, while providing Unity Events for when it starts/finishes loading, and properties for scene management. You can also assign a loading screen prefab to it, so it'll be displayed while your scene is being loaded.

Game Saver

The **Game Saver** is responsible to save the game data to the persistent memory. There are three different save modes available: Binary, JSON, Player Prefs. You can specify the file name, when using JSON and Binary mode, as well as the file extension when using Binary. In most situations, Binary mode should be good enough.

> Save files are stored in the application persistent data path, which changes depending on the build target platform.
> For more details, please visit this page:
> https://docs.unity3d.com/ScriptReference/Application-persistentDataPath.html.

Game Controller

The **Game Controller** is a component that calls public methods from the Game, allowing then to be accessed anywhere, and can be used within Unity Events for some in Inspector programming.

Game Tags

The **Game Tags** is a class that holds all tag names as static fields, so you can read the tags from it without needing to know how their where exactly spelled.

Level

The **Level** is a singleton component that controls the current level state, and should only exist on scenes that correspond to a level (since you can have non-level scenes, like the menus). If you look at the code, it's not doing much by itself and can only track the player reference. I implemented it in this way so it's open to possibilities.

Creating a new level

To create a new level you'll need a scene with the GAME and the LEVEL global prefabs, one Player, and a Camera tagged as "MainCamera" (so you can get relative input data). To make it show up in the level selection, you'll need to register your level in the "Levels" property of the

GAME prefab, it's a very straightforward setup, but don't forget to add its corresponding scene to the Build Settings, or else the game won't be able to load it. After that, your new level will be ready to be played and filled with any other assets you want.

Level Data

The **Level Data** is an object that represents a level in the data layer, which means the data of a given level to be persisted, like the total coins collected on it, collected stars, and its completion time.

Level Finisher

The **Level Finisher** is a singleton component used to safely finish a level, persisting its data in case the level was beaten, or simply changing to another scene in case of a game over or if the player decided to exit it for some reason.

Level Pauser

The **Level Pauser** is responsible to control the pausing state of a level. It also controls if you can or can not save, as well as provides callbacks to pause and unpause actions. In other words, it's a fancier way of setting the `Time.timeScale` to 0 or 1, while keeping some control over it.

Level Respawner

The **Level Respawner** is a component that handles restart, when it resets the entire scene, or respawn when the player is repositioned based on the checkpoint. It communicates with the scoring system to change the number of retries and coins the player has during the level.

Level Score

The **Level Score** corresponds to the score during the level, so it won't be persistent till the level is beaten. It's a singleton component, so you can access its instance from anywhere to collect coins, and stars, or to trigger the consolidation (which tries to save the level data).

Level Starter

The **Level Starter** is a component that handles the begging of a level, controlling the time and input activation. When starting a level from the editor it might look a little off, with a slight delay before being able to control the character, but in-game it's a lot more natural.

Level Controller

The **Level Controller** is a component you can add to any Game Object to have access to some of the level singletons in a single place. It's very useful if you want to change something in the level through Unity Events using the Inspector, providing some degree of in-editor programming.

Gamepad

The **Gamepad** is a static class responsible for managing input data. It's similar to Unity's Cross-Platform Input, an old standard asset. It works by registering a dictionary of inputs, using their names as keys, and custom objects, separated as axis and buttons, for values. The Gamepad works very similarly to the built-in Input class, where you call static methods to retrieve a specific input value, like `Input.GetButton("inputName")` or `Input.GetAxis(axisName)`. The main difference is that you can not only get input values, but you can also set them.

Gamepad Binding

The **Gamepad Binding** is a class containing enums for Axis, Buttons, and Directions, which maps all their names based on the XBOX controller nomenclatures. It's mainly used by the Entity Input Manager to get input values instead of relying on strings. Remember that the names of the buttons on the enums must correspond to their names declared on the project's Input Manager.

Mobile Rig

The **Mobile Rig** is used to enable the virtual gamepad automatically when the target platform is mobile.

Virtual Axis

The **Virtual Axis** is a component used to create mobile axis input. It assigns a float value, which can also be negative, to the corresponding axis by its name if the Game Object its assigned to is being touched. You can find an example of its usage at
`PLAYER TWO/Platformer Project/Examples/Prefabs/UI/Virtual Gamepad` .

Virtual Button

The **Virtual Button** is a component used to create mobile buttons. It works by setting button values in the Gamepad class, depending on the touch events. You can create new mobile buttons by adding this component to any UI button and specifying the button name it corresponds to. Please look at the example prefab that you can find on
`PLAYER TWO/Platformer Project/Examples/Prefabs/UI/Virtual Gamepad` .

Virtual Thumb Stick

The **Virtual Thumb Stick** is a component used to create mobile thumbsticks. Similar to the Virtual Button, it works by setting axis values in the Gamepad class. Creating new thumbsticks for mobile needs a little more challenging setup. Please look at how it works on
`PLAYER TWO/Platformer Project/Examples/Prefabs/UI/Virtual Gamepad` .

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.

Waypoint Manager

The **Waypoint Manager** is a component responsible to control the waypoint flow, which is how the transitions will be made from one point to another, based on the Waypoint Mode. The “Waypoints” list contains an array of transforms, representing the positions in world space of each waypoint of the chain.

Waypoint Mode

The **Waypoint Mode** is an enum declaring all possible sort modes as follows:

- **Loop**: move back to the first waypoint after the chain is finished;
- **Ping Pong**: moves the chain backward after finishing it;
- **Once**: stays in the last waypoint after the chain is finished.

UI

The **UI** is a collection of scripts that reads the data from somewhere and displays them in UI elements on Canvas. There are also some helper components to make your life easier when developing your custom menus or any other UI elements.

HUD

The **HUD** component handles the level HUD, displaying and formatting the total coins collected, the number of retries, the health, the collected stars, and the time count. All the numbers are displayed as UI Text, while starts are images enabled or disabled depending on their “collected” status. The counters are listening to score change events, so it’s not necessary to manually update these values.

UI Animator

The **UI Animator** is a custom component that handles some common animations that can be used by any UI element. With this component, you’ll be able to trigger 2 states, hide and show, and their corresponding animations will automatically be played. This is the component used to animate some screens and menus.

UI Focus Keeper

The **UI Focus Keeper** is a simple component that should be used within the Event System to keep UI elements focused, solving the problem of losing control over menus.

UI Horizontal Auto Scroll

The **UI Horizontal Auto Scroll** is a component used to automatically move horizontal Scroll Views.

UI Level Card

The **UI Level Card** is the component used to read a given level's data and fill up the UI on the Level Select screen.

UI Level List

The **UI Level List** automatically generates a list of UI Level Cards based on the Game data.

UI Save Card

The **UI Save Card** is the component used to read a given save file data and fill up the UI on the File Select screen.

UI Save List

The **UI Save List** automatically generates a list of UI Save Cards based on the Game data.

Misc

The Misc is 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.

Breakable

The **Breakable** component is a way to represent an object that can be broken by a Player's attack. It works by disabling a child Game Object, which represents the object visual when it's not broken, and playing a particle, which represents the object being broken. It can also play an audio clip and provide callbacks.

Buoyancy

The **Buoyancy** component makes rigibodies float on water.

Checkpoint

The **Checkpoint** is a component that saves the Player's position on the level after colliding with it.

Collectable

The **Collectable** is a component that represents Game Objects that can be collected by the Player when its collider is overlapped. It must be used combined with the Level Controller, so you can register what was collected by calling the controller when the event "On Collect" is invoked. You can also inherit from it and create more advanced collectables, like the Stars.

Fader

The **Fader** is a singleton component that can be used with a UI Image to fade in/out the whole screen.

Falling Platform

The **Falling Platform** is a component that moves a Game Object downwards after the player steps on it. It will also make the platform shake before actually falling, so the player has some room to jump before falling with it. The Falling Platform gets back to its original state after a while.

Floater

The **Floater** is a component that makes a Game Object move up and down using a sin curve. It's mostly used for collectables, but you can add it to anything.

Gravity Field

The **Gravity Field** is a component used to move the player upwards when it's overlapping it, allowing the player to reach higher platforms.

Grid Platform

The **Grid Platform** is a platform that automatically rotates when the Player jumps, useful for platforming puzzles.

Hazard

The **Hazard** is a component that applies damage to the Player when it's touched.

Health

The **Health** is a component used to represent a damaging count. It's mainly used by players and enemies.

Hit Flash

The **Hit Flash** component provides a visual feedback for the Health component by changing the material color from an array of skinned mesh renderers.

Item Box

The **Item Box** is a component that spawn collectables when the Player contacts with it from bellow.

Kill Zone

The **Kill Zone** is used to kill the player after its collider is overlapped. Useful to make bottomless pits.

Mover

The **Mover** is used to apply an offset to any Game Object transform.

Moving Platform

The **Moving Platform** is a component that makes Game Objects move around based on a given Waypoint Manager setup.

> Moving Platforms don't necessarily need this component. You can make the Player stick to a given surface by just tagging the surface Game Object as "Platform".

> **NEVER** change the scale of a moving platform transform.
> Separate its mesh as a child Game Object and adjust its collider component properties to fit it.

Panel

The **Panel** is a component that triggers the "On Activate" event after the player steps on it. Currently being used by the Level Finisher Panel.

Pickable

The **Pickable** is an object the Player can hold using the hold action. Each of them has an offset in order to better adjust its position relative to the Player while being held.

Pole

The **Pole** represents an object the Player can climb on. It used a capsule collider, since Unity lacks cylinders, to handle its radius, and the bounds to handle its heigh. You scale the pole Game Object along the Y-axis, but you can't rescale along the X and Z. If you want a thicker pole, separate its mesh Game Object and adjust its collision accordingly by tweaking the Capsule Collider radius property.

Rotator

The **Rotator** is a Game Object that makes the object rotate in place. Used by collectables and some moving platforms.

Sign

The **Sign** component is useful to create signs like the ones used on the sample scene. It basically scales a world space canvas.

Singleton

The **Singleton** component is used as a base class for components that needs to be accessed globally, like some of the Game and Level stuff.

Spring

The **Spring** is a component that applies an upward force to the Player and plays an Audio Clip when it steps on it.

Star

The **Star** is a custom Collectable that register collected stars on the level. It's also going to be disabled if it was already being collected before.

Toggle

The **Toggle** is useful to create interactive On/Off events through the inspector.

Volume

The **Volume** is an abstraction of Unity’s built-in collision trigger events.