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;
- Slope Physics (Speed Up or Slow Down);
- Rail Grinding (slopes & loop-the-loop);
- Pick Up and Throw objects;
- Spin Attack ability;
- Air Spin Attack;
- Gliding ability;
- 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;
- Portals travel.

Supported Packages

- Cinemachine;
- New Input System;
- Post Processing;
- Splines.

Supported Integrations

- Emerald AI.

Assets with integrations must be purchased separately.

> 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:

1. Download Unity version **2021.3.25f1** or higher;
2. Create a new 3D project and open it;
3. Download the asset through `Window > Package Manager > Packages: My Assets` and import it;
4. Click on “Import” on the complete project pop-up;
5. Click on “Install/Upgrade” to install all Package Manager dependencies;
6. Click on “Yes” to disable the old Input System;
7. After your editor restarts, import the asset again (repeat steps 3 to 5);
8. Click on “Import” to import all the asset data and close the Package Manager window;
9. Open the `SampleScene` inside the following directory: `PLAYER TWO/Platformer Project/Examples/Scenes` ;
10. Hit Unity’s Play button and start to mess around.

> 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 Physics, Tags, and Layers settings. It’s a requirement to override those when you’re importing any complete project on Unity.

URP and HDRP

If you don’t plan to use any of the built-in materials and shaders in your game, there’s no need to follow this guide. All the code from this asset works regardless of the render pipeline you want to use. These instructions are only necessary to fix the pink materials from the sample scene.

To use the URP or HDRP scriptable render pipelines, import the asset into a new project using either the Universal RP or High Definition RP. When loading the template scene, you’ll notice that all materials are pink. URP and HDRP packages provide tools to automatically convert the materials from the built-in rendering pipeline, and they are under the `Edit > Rendering` tab. Please, visit the following pages for more details about migrating the materials:

- Upgrading your Shaders (URP)
- Upgrading to HDRP

> Unity provides tools to convert from the built-in render pipeline to URP or HDRP, but you can't convert from URP to HDRP, and vice-versa, or back to the built-in. Please, be careful when changing the render pipeline and always make backups.

After converting the materials, you'll notice that some are still pink. This happens because this asset uses custom shaders to enhance some of its visuals. To fix the rest of the pink materials, replace the shaders from the `Examples > Shaders` folder with the ones from the following zip files that match the render pipeline you're using:

- Universal RP Shaders
- High Definition RP Shaders

> URP shaders can achieve very similar results to the built-in ones, but HDRP looks very different, even though the shaders do the same thing, because of how lighting is handled differently.

Please note that URP and HDRP don't support projectors so the character will lose its blob shadow effect. The abscess of a perpendicular blob shadow makes platforming harder for the Player. Please, consider replacing it with decals when not using the built-in render pipeline. You can read more about decals on the following pages:

- Universal RP Decals
- High Definition RP Decals

After that, you'll need to adjust the post-processing to fix the sample scene. Both URP and HDRP handle post-processing differently. On URP, you'll need to enable the post-processing in the Main Camera first and then replace the Post Processing Volume component with the Volume component. Create a new Profile for the Volume component and assign the desired effects. On HDRP, you'll need to tweak the effects under the HDRP Global Settings from `Edit > Project Settings > Graphics`. To match the visuals from the built-in one, enable Bloom, use color grading ACES, increase saturation to 30, and decrease the contrast by 10.

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 your project from a 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 of your game assets in the root Assets folder or in a subfolder inside of it. 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 since 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 a joystick. The joystick buttons where named with the XBOX controller in mind, but any controller will work with the asset using the corresponding buttons positions. The keys are mapped as such:

- Walk: "WASD" or Gamepad "Left Thumb";
- Look: "Mouse" or Gamepad "Right Thumb";
- Run: Hold the "Left Shift" key or Gamepad "X";
- Dive: Hold the "Right Mouse" button or Gamepad "X";
- Jump: Press the "Space" key or Gamepad "A";
- Glide: Hold the "Right Mouse" button or Gamepad "Right Shoulder";
- Dash: Press the "F" key or Gamepad "Left Shoulder";
- PickUp/Throw: Press the "G" key or Gamepad "Y";
- Crouch/Crawl: Hold the "Q" key or Gamepad "Left Trigger";
- Spin Attack: Press the "Left Mouse" button or Gamepad "Right Trigger";
- Stomp Attack: Press the "E" key or Gamepad "Right Shoulder";
- Air Dive: Press the "Q" key or Gamepad "Left Trigger";
- Release Ledge: Press the "Right Mouse" button or Gamepad "Left Trigger";
- Backflip: Crouch and Jump or Skid and Jump;
- Rail Dash: "F" key or Gamepad "Left Shoulder";
- Rail Brake: "Q" key or Gamepad "Left Trigger";
- Pause: Press ESC or Gamepad Start.

> You can change the button that correspond to a given action through the Input Actions located on `Assets/PLAYER TWO/Platformer Project/Examples/Input Actions/Player Input Ac` asset.

Getting Help

> Make sure to read the documentation before asking for help. Also, look at the **FAQs** to see if your question wasn't already answered.

If you need any support, feel free to contact me at playertwopublisher@gmail.com. Please, don't forget to add your **Invoice Number** to the email body so I can validate your purchase.

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

Frequently Asked Questions

Why is the console showing errors when I load my new Level?

When creating new Levels, add their respective scenes to your project's Build Settings. Also, ensure the scene's file name **matches exactly** the Level's scene name on the Game "Levels" list.

Why is the console showing errors related to inputs?

There seems to be a problem with the new Input System that happens randomly. If you just opened the project and have input errors in the console related to null references in the Player Input Manager, or if the inputs are not working, try to close Unity and open it again. If the problems persist, try to reinstall the new Input System from the Package Manager.

Can I use non-humanoid characters?

Yes, you can use any character you want. It can be a dragon, a dog, or a floating capsule. The "humanoid rig support" refers to a feature in Unity that allows you to reuse animations easily. So if you have a character configured to use the humanoid rig, you can easily replace the template character while transferring all of its animations to yours. If your character is not a humanoid, you'll

need to create your animations, but all of the movement code and underlying systems work regardless of what characters or animations you're using.

How can I replace the template character's model?

If the model you want to replace with supports the humanoid rig, follow the replacing the template character's model instructions. If it doesn't, you'll have to build another animator controller for it, based on the built-in one, using your own custom animations. In any case, see the Player Animator section for more details.

Can you add "this specific feature" to the asset?

I consider requests, but sometimes they don't fit into a general-purpose asset. So, before making requests, ask yourself if the feature you want is not too specific to only your needs.

Do you have plans on adding online multiplayer?

Multiplayer on Unity can be achieved by either using third-party packages, which I don't want to bloat into the asset, or by the newly released MLAPI, which is still new on the market and I don't have much experience with. So, in short, do not create expectations of having online multiplayer support any soon.

When the next update will be released?

There is no ETA for upcoming updates since working on the asset is not my only occupation. Updates generally take a month but can take longer if I don't have enough time to work on the project.

Can I hire you to work on my project?

Sorry, but no. I already have a full-time job, and I'm busy working on the asset in my free time.

How can I support you?

If you already bought the asset, feel free to leave a 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 Events

The **Entity Events** class defines all the events triggered by the Entity component.

Entity Hitbox

The **Entity Hitbox** is a component that allows a given Entity to apply damage to others or break objects using Unity's built-in colliders and trigger collision detection system. It also allows the Entity that is applying the damage to react to the hit event, either by rebounding; which causes the Entity to move upward or downward based on the hit direction; or by pushback, which applies a backward force to the Entity.

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 State Manager Events

The **Entity State Manager Events** class defines all the events triggered by the Entity State Manager component.

Entity State Manager Listener

The **Entity State Manager Listener** is a component that allows you to have an inspector interface for the “On Enter” and “On Exit” events triggered by the Entity State Manager component. To use it, you need to add the name of the state you’ll want to listen to to the “states” list, and then whatever action you added to the events will be fired when either the State Manager enters or exit the states from the list.

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 a component that inherits its base functionalities from the Entity component. The main difference between the Player from other Entity implementations is its input-driven behavior, which means this Entity is designed to move based on the Player Input. Players also use the

Player State Manager, which controls the Player States and allows it to perform actions, such as standing still, walking around, swimming, climbing ledges, etc.

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

Creating a new Player

If you look at the Player Component, you'll notice that it provides many Unity Events for its actions. Unity Events are a handy way of making reactive programming inside of Unity. Use it to your advantage. You'll also notice that it expects the reference for two transforms, the "Pickable Slot," which will be the transform of a Game Object to attach objects while the Player is holding them, and the "Skin," which must be a Game Object parent of any of your Player's models and will be used to control the character's model offset when performing actions such as Ledge Climbing.

> If crates disappear when your Player tries to grab them, you didn't set a Pickable Slot transform. It can be, for example, a Game Object child of the Player's model hand.

As you may have noticed, the Player component alone can't do much, but it's the main component you'll need when creating a new Player. To create a new Player prefab from scratch, you'll need to create an empty Game Object and drag and drop the Player component to its inspector. You can also add the component using the "Add Component" button from the inspector under the `PLAYER TWO/Platformer Project/Player` section, which lists all components relative to the Player's domain. After that, all the minimum necessary components will be automatically added to your new Player Game Object, and they are:

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

These components are the base for any Player prefab. Note that since the Player Input Manager is based on Unity's New Input System, you'll need to provide an Input Action Asset to the "Actions" field. You can create a new Input Action Asset by right-clicking on the Project window and selecting `Create > Input Actions`. You can also use the default Input Actions from the default character, located in the `PLAYER TWO/Platformer Project/Input Action` folder.

The Player Stats Manager component also expects a reference to at least one Stats asset from the “Stats” list. The stats used by the default character can be found in the `PLAYER TWO/Platformer Project/Stats` folder. You can also create a new Stats asset containing all the default values by right-clicking on the Project window and selecting `Create > PLAYER TWO > Platformer Project > Player > New Player Stats`.

The Player States will drive the actual motion of the Player. To make the Player react to the world and perform basic actions like moving around based on inputs, click on the plus button on the “states” list from the Player State Manager and select the state you want to add from the dropdown list. For a simple platforming game, these are the minimum states you must have:

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

You can also create your own original Player States.

With the Player Game Object setup complete, you’ll be able to hit the play button and have a floating capsule moving around, as long as the Game Object is selected and the Gizmos are enabled on the Game window. Please note that the Player component automatically generates a Character Controller when starting the game. So, if you want to adjust properties like the Player’s collider height, center, radius, etc., you can do it by manually adding a Character Component to your Player’s Game Object.

If you want to add a character model with animations to your Player Game Object, either by using the default ones or any custom animations, please follow the instruction under the Player Animator section.

> Please, do not skip the instructions under the Player Animator section. Suppose you are trying to configure the animations yourself without knowing how animations are being handled. In that case, you’ll either end up with a character playing no animations or playing the wrong ones.

You can add other non-required components to your Player to make it more reactive to its surroundings and make it more alive:

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

You can also create custom components and add them to the Player like any other Unity Game Object.

After completing the setup of your newly created Player, don't forget to save it as a prefab by dragging and dropping it to a folder under the Project window. This way, you'll be able to quickly add it to any scene without needing to re-do the entire setup. You'll also be able to make changes to it without needing to update it across all the scenes it's being used.

> Please, consider looking at the default character's prefab to understand better how Players are assembled. You can find it at `PLAYER TWO/Platformer Project/Prefabs/Entities/Lily` .

Player Input Manager

This component is responsible for reading input action data from the input actions asset. It works as an interface between the game's contextual actions - like jump, run, or crawl - and raw input data, so you can easily change the input logic without needing to update multiple files every time.

> The Player will need camera relative input directions, so you must have a camera tagged as "MainCamera" on your scene.

Adding New Input Actions

The Platformer Project makes use of Unity's new Input System. Since I don't recommend you to modify any of the asset's data directly, it's a good practice to copy and past the default Player Input Actions, located on `Assets/PLAYER TWO/Platformer Project/Examples/Input Actions` , to your project folder. After that, you can access the Actions list by double clicking on the asset file and from there, you can create new actions or edit the current ones. After making your changes, don't forget to save.

To read the actions from your asset, you'll need to change Player Input Manager. Since I don't recommend changing files directly, create your input manager script and inherit it from the Player Input Manager class. For example, you can make your class with the following structure:

```cs
using UnityEngine.InputSystem;

// Make your new Player Input Manager component inheriting from the built-i
public class MyPlayerInputManager : PLAYERTWO.PlatformerProject.PlayerInput
{
    // This is a reference to the input action.
    private InputAction m_myAction;

    // This method will be called once on Start and it's useful to cache
    // the actions reference and avoids fetching actions by string all the
    protected override void CacheActions()
    {
        // Call the base class to keep the default caching.
        base.CacheActions();

        // Cache your new action into a variable using it's name which
        // corresponds to the name defined on the input actions asset
        m_myAction = actions["MyAction"];
    }

    // Define a new method to access the value you want from the Input Acti
    public bool GetMyActionDown()
    {
        // This method will return true if the button corresponding
        // to the action was pressed in the current frame.
        return m_myAction.WasPerformedThisFrame();
    }
}
```

Note that I defined the getter to return a boolean, the value type for regular buttons. Suppose you specified your action as an Action Type that is not Button. In that case, you'd need to write the method to return a value type that matches with your action's Control Type and read the value from `m_myAction` with the `ReadValue<T>()` method ("T" corresponds to the action Control Type type). Example:

```cs
// If your action has an Action Type "Value" and Control Type "Vector2"
// you must define the method to expect a Vector2 value type as well.
public Vector2 GetMyActionDirection()
{
    return m_myAction.ReadValue<Vector2>();
}
```

With your new Input Manager created, you can create as many actions as you need by following the same principles of “defining > caching > reading.” Feel free to take a look on the default implementation found on `Assets/PLAYER TWO/Platformer Project/Scripts/Player/PlayerInputManager.cs`.

Also, don’t forget to assign your new Input Manager component to your Player prefab, replacing the built-in Player Input Manager previously in use, and set the new input actions asset to the Actions attribute of the manager, or else the changes won’t take any effect.

Finally, to do something with the newly created input action, you’ll need a reference to your new `MyPlayerInputManager` component to call the corresponding method of your action and get its value. You can access it using regular `GetComponent` methods or the Player component - if you already have access to an instance - by accessing the `inputs` attribute `(MyPlayerInputManager)player.inputs`. Don’t forget to cast it to your new class if you’re reading it directly from the Player, or else you’ll not have access to the latest methods.

You can use the following class as an example of how to read the action you just created:

cs

```cs
using UnityEngine;

public class InputActionTest : MonoBehaviour
{
    // This is a reference to your new Input Manager.
    private MyPlayerInputManager inputs;

    private void Start()
    {
        // I'm using GetComponent here because I've added this script to th
        // If you need to access it from somewhere else, you'll need a dif
        inputs = GetComponent<MyPlayerInputManager>();
    }

    private void Update()
    {
        // Read the value by calling the method you just created.
        // Since I defined it to return a bool value, I can use it like th
        if (inputs.GetMyActionDown())
        {
            // Every time I pressed the button corresponding to the action
            // the console will show the following message:
            Debug.Log("It's working!!!");
        }
    }
}
```

Note that I named the new action `myAction`, but it must have a meaningful name related to the context of its usage. As mentioned before, look at the built-in Player Input Manager implementation and use it as a reference for your new manager.

> The Platformer Project currently uses version 1.1.0-preview.3 of the Input System package.

Player Stats Manager

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

> The stats manager must have at least one active stats. So, make sure to assign at least one Player Stats scriptable to the first index of the Stats list.

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 clicking on the plus button from the “states” list in the Player State Manager component and selecting one state from the dropdown list. 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.

Gliding Player State

The **Gliding Player State** makes the Player fall slower, allowing it to adjust its falling direction better and travel longer distances.

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 manager.

Pole Climbing Player State

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

Rail Grinding Player State

The **Rail Grind Player State** uses Unity’s new Splines package to create different platforming mechanics that allows “on rails” segments. The Player will gain or lose speed based on the rails

angle, which is suitable for momentum-based games. It can also perform a grinding dash to gain instant speed or brake to reduce it.

For instructions about how to create custom Splines, see the Creating Grindable Rails section.

> You can configure the grinding physics and its sub-abilities under the Player Stats.

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.
        }

        public override void OnContact(Player player, Collider other)
        {
            // Executed at every Update where the Entity is in contact with a c
        }
    }
```

Transitions to this state 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.

Creating Grindable Rails

To create grindable Game Objects, add a new Spline to your scene through `Game Object > Spline`, then select one of the default shape or click on the Draw Splines Tool to make a custom one. You can read the official Splines documentation for more information here.

After drawing your new Spline, add the Spline Extrude component to make it visible in-game, then add the Mesh Collider component to make it solid. Finally, set the Spline Game Object tag to “Interactive/Rail”, so the Player can interact with it.

To ensure the Player won’t try to wall slide or climb the Spline Game Object, move it to the “Unstable Surface” layer mask. After that, you can hit play, and your new Spline must work fine.

> You can look at the default splines in the 'TestLevelScene' hierarchy under the `Test Level > Rails` Game Object.

Player Animator

The Player Animator is responsible for controlling the Player's animations. It sends data from the Player's components to Unity's Animator as animation parameters. Unity's default animator system handles all the animation transitions by evaluating the parameters it receives through the Player Animator component. However, if you need more parameters, you can change the Player Animator to access any other component and send the specific data you want using the Animation Parameters API.

Suppose you're creating your Player from the ground up. In that case, you'll need to add its model as a child of the Player prefab, assign an Animator Controller to the "Controller" property from the Animator component of your model, and finally assign its Animator component to the Player Animator "animator" field. Suppose you don't give any animator reference to the Player Animator. In that case, it will not be able to send animation parameters nor perform any animation transition, and your character will always be in the default animation state.

To make your custom character model share the same animations as the default character, you'll need to set up its rigging to be humanoid. Unity's Animator cannot retarget the animations if your character is not a humanoid, making your model play no animation when using the default character animator controller. There's an official Unity tutorial about humanoid rig configuration. https://youtu.be/pbaOGZzth6g

> You can have non-humanoid characters with custom animations. The rigging configuration is only needed if you want to use the default animations with a custom model.

You can find the Animator Controller used by the default character at `PLAYER TWO/Platformer Project/Examples/Animations/Models` .

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);

- Health (int);
- Jump Counter (int);
- Is Grounded (bool);
- Is Holding (bool);
- On State Changed (trigger).

The default Character's animator controller asset works by transitioning from "Any State" to the desired state you want using the State parameter. The State parameter corresponds to the index of the active Player State based on the Player State Manager "states" list. So if your Idle State corresponds to the element 0 from the state's list, when the State parameter has the value 0, the controller should transition from Any State to the Idle animation state, as long as there's a transition considering this specific value. For example, suppose you changed the state's list, and now your Idle State corresponds to element 1. In that case, you'll also need to update the controller transitions to consider this new value. Otherwise, the controller will play another animation that does not correspond to the Idle animation you expect.

You can change the controller's transitions to consider any other parameter you want, and there's no need to stick to the default character approach. I've made it this way because it's straightforward to visualize all animation transitions. The downside of this approach is that any change to the Player State Manager states list order will cause your controller to transition to the wrong animation, but it should only happen the first time you're building your character. After that, it's a matter of adding new transitions.

Replacing the template character's model

If your character's model is already configured to use the humanoid rig, delete Lily's model from the Lily prefab hierarchy and add your character's model in its place. Select your character in the hierarchy and assign Lily's controller and avatar to the Animator component in the Inspector window. Then, ensure that you assign your character to the Animator property in the Player Animator component of the Player's root Game Object. Finally, save your modified Player prefab either as a variant or an original prefab by dragging and dropping its root Game Object into a folder in the Project window.

> If you're creating your Player from the ground up, please ignore these instructions. There'll be no need to replace anything.

> If your model is not using the humanoid rig, you'll need to build another animator controller using your custom animations. Use the built-in one as a reference.

Holding Animation and Animation Layers

The default animator controller also uses Animation Layers and Avatar Mask to the model's upper body to handle holding objects' animations. This Layer is called the "Upper Body," and you can find it right after the Base Layer in the Animator window. All Animation Layers share the same parameters and handle transitions precisely in the same way. The main difference between the Upper Body layer and the Base Layer is its state transition logic, which uses an Empty Animation State when the character is not holding anything, so the model animates using the traditional upper body animation from the base layer. There's only a transition to the Holding Animation State when the parameter "Is Holding" is set to true, which will cause the Layer to override the animation from the base layer with the standing arms.

The default character's avatar mask can be found at
`PLAYER TWO/Platformer Project/Examples/Avatar Masks` .

Importing Custom Animations

When using custom animations, please toggle the "Bake Into Pose" option for all Root Transforms in the import settings, or else your custom animations will move away your character's model from the Player's origin. This happens because the asset does not use animation root motion, which may be present in your animations even if it doesn't seem to be. You can read more details about it at https://docs.unity3d.com/Manual/RootMotion.html.

Ledge Climbing Animation and Forced Transitions

The Ledge Climbing animation works uniquely, requiring a feature from the Player Animator called Forced Transitions. It detects when a specific transition happens, e.g., a transition from the Ledge Climbing Player State, and forces an animation clip to play based on its name on the desired Animation Layer. The sole purpose of this feature's existence is to solve a glitch from the Animator where the Ledge Climbing animation offsets to the wrong position for a single frame. Forcing the transition will guarantee that this problem won't show up, so you don't need it for anything else.

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 its settings.

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** works as an extension for Cinemachine's virtual cameras; more precisely, it's a component to be used with the Cinemachine Free Look camera. This component will move the virtual camera so that it follows the Player and leaves some room for visibility to allow precise jumps.

It'll also make the camera react to game events, giving or taking control depending on the circumstance, and correctly repositioning it when the Player respawns.

When using this component, you don't need to manually assign the Player to the camera "Follow" and "Look At" slots. Instead, it'll be done automatically on its initialization processes.

> You can choose not to use it if you want, but you'll lose some interesting behaviors.

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.

Player Events

The **Player Events** defines all the events that can be triggered by the Player component.

Player Events Listener

The **Player Events Listener** is a component to provide inspector-level access to the events triggered by the Player component. It's helpful to handle hitboxes.

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
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.

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.

Introduction

This page will help you to integrate the Platformer Project with the Emerald AI 3.0 asset from Black Horizon Studios. Note that **Emerald AI is not bundled with Platformer Project**, so you must buy it separately from the Asset Store to use it. Also, Emerald AI is an optional integration. If you don't own Emerald AI, ignore this page.

> Make sure to have a fresh install of Emerald AI. Then, use the built-in enemies prefabs from Emerald AI to test the integration before creating custom ones.

> This tutorial expects you to have already imported all the Emerald AI assets into your project. So please, read Emerald AI documentation first to install it properly before integrating it with the Platformer Project. https://github.com/Black-Horizon-Studios/Emerald-AI/wiki.

Damaging Player

Emerald AI uses Tags and Layers to detect its targets. So, to make it follow and attack the Player, you'll need to create a new Layer for your character and assign this same Layer to the `Detection Layers` under `Emerald AI System > Detection & Tags > Tags & Faction Options`. Also, make sure the property `Follower Tag` under this same section is set to `Player`.

> To learn more about how to create Layers and how to assign it to Game Objects, please access the official documentation: https://docs.unity3d.com/Manual/create-layers.html.

To make the Emerald AI system able to damage the Player, you must change the `Emerald AI Player Damage` class to get the Platformer Project Player class and call its `ApplyDamage` method. To achieve this, copy & paste the following method inside the `EmeraldAIPlayerDamage.cs` file:

cs

```
protected virtual void DamagePlayerPlatformerProject(int amount)
{
    if (TryGetComponent(out PLAYERTWO.PlatformerProject.Player player))
    {
        player.ApplyDamage(amount);
    }
}
```

Finally, call this same method right after the `DamagePlayerStandard` from line 20 by adding this code:

cs

```
DamagePlayerPlatformerProject(DamageAmount);
```

After that, Emerald AI enemies can follow, attack, and apply damage to your Players.

Damaging Emerald AI

The Platformer Project uses hitboxes to apply damage to other entities. These hitboxes detect collision through Unity's "On Trigger" callbacks. So, suppose you are using the default Enemies prefabs from Emerald AI. In that case, you must first add a Rigidbody component to them, or else the hitbox will be unable to detect and fire the collision event.

After that, you'll need to create a new class to extend from the default `EntityHitbox` component and override its `HandleCustomCollision` method to interact with Emerald AI System. To achieve that, create a new C# Script called `EntityHitboxEmeraldAI.cs` and copy & paste the following code into it:

cs

```
using UnityEngine;
using EmeraldAI;
using PLAYERTWO.PlatformerProject;

public class EntityHitboxEmeraldAI : EntityHitbox
{
    protected EmeraldAISystem m_emerald;
```

```
    protected override void HandleCustomCollision(Collider other)
    {
        if (other.TryGetComponent(out m_emerald))
        {
            HandleRebound();
            HandlePushBack();
            m_emerald.Damage(damage, EmeraldAISystem.TargetType.AI);
        }
    }
}
```

After creating the new Entity Hitbox script, replace all components with this class on the hitboxes of your character prefab. For example, the default hitboxes for Lily are under the Hitboxes Game Object. Therefore, each of its children has an Entity Hitbox component that must be removed and replaced by your newly created one. After that, the Player will be able to damage Emerald AI enemies.

> Please, don't forget to add a Rigidbody component to all your enemies' Game Objects, or else they will never receive damage from the Player.
