About

ARPG Project is a comprehensive Unity Asset that enables game developers to easily create 3D action RPG games. Designed with a focus on delivering the core functionalities of classic ARPG games, this template is an ideal solution for those who want to create action RPG games in Unity.

Built with C# and inspired by some of the most popular ARPG games in history, ARPG Project provides developers with all the tools they need to create compelling, high-quality games. The template includes a wide range of features, such as an intuitive character controller, stats system, enemy AI, inventory systems, skill system, and much more.

Whether you're a seasoned game developer or just starting out, ARPG Project makes it easy to create stunning games with rich gameplay mechanics. With its comprehensive feature set and easy-to-use interface, this template is the perfect solution for anyone who wants to create an engaging, immersive ARPG game in Unity. So why wait? Start building your dream game today with ARPG Project.

> Please note that ARPG Project requires some understanding of general Unity usage. You can learn more about Unity here: https://learn.unity.com/

> Replacing visual elements and making adjustments to some of the asset mechanics may not need much coding skills. But if you're looking to make bigger changes, C# knowledge is required.

Features

- 3D Character Movement;
- Enemy AI;
- Humanoid Rig Support;
- Save/Load (Binary or JSON);
- Stats Point Distribution;
- Health and Mana System;
- Skill System w/ Cool Down;
- Stash System (Store Items);
- Minimap & Icons;
- Level & Experience Progression;
- Inventory System;
- Dynamic Items' Attributes;
- Items' Durability;
- Damage Based on Stats;
- Defense Based on Stats;
- Weapons (Blades & Bows);
- Armor System;
- Consumable Items;
- Collectible Items (Loot & Drops);
- Destructible Objects;
- Quest System & Rewards;
- Buy or Sell Items (Merchant);
- Repair Items (Blacksmith);
- 3D Camera (Zoom and Rotate);
- Waypoints (Flash Travel).

Supported Packages

- Cinemachine;
- New Input System;
- Post-Processing;
- Shader Graph.

Buying the Asset

ARPG Project is available on Unity Asset Store: https://assetstore.unity.com/packages/slug/251134.

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 `Tutorial` inside the following directory:
   `PLAYER TWO/ARPG Project/Examples/Scenes` ;
10. Hit Unity’s Play button and start to mess around.

> Do **NOT** import ARPG Project into an existing project.

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.

The demo scene uses multiple real-time light sources, which is not ideal in forward rendering (URP) because of the maximum per-object light limitation. If you want to achieve similar visuals, you’ll need to stick with the built-in deferred rendering or use HDRP. Use URP only if your game won’t have so many real-time lights or if you can bake them to avoid the limitation.

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 **most are still pink**. This happens because this asset uses custom shaders to allow the highlight effect when hovering the mouse on certain objects. 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

> As mentioned above, the same scene in URP will not look the same because of the light limitation, regardless of the shader you’re using. In HDRP, you must adjust the light and emission intensities to match the sample scene.

After that, you must 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, add a vignette, enable the ambient occlusion, and turn on motion blur.

File Structure

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

You'll find two subfolders in the ARPG Project folder, which are `Core` and `Examples`.

The **Core** folder contains all the C# files that give life to the asset, and they are separated by concern, so there's no GUI logic out of the GUI folder nor Collectible stuff inside the Entity 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.

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.

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 `ARPG 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

Please note that currently, ARPG Project only supports mouse and keyboard inputs. While support for other input methods may be added in the future, the current version of the template is designed to be used with a standard mouse and keyboard setup. If you require support for other input methods, please consider this limitation before purchasing the asset.

Player Inputs

- Movement: "Left Click";
- Attack or Interact: "Left Click" the target;
- Attack w/ Skill: "Right Click" the target;
- Standing Attack: Hold "Left Shift" + "Left Click";
- Standing Skill Attack: Hold "left Shift" + "Right Click";
- Consume Items: "Q", "W", "E", "R" keys;
- Change Skills: "1", "2", "3", "4" keys;

Camera Inputs

- Zoom In/Out: Mouse Scroll;
- Rotate: “Left Alt” + Mouse Scroll.

GUI Inputs

- Toggle Skills Window: “S” key;
- Toggle Character Window: “C” key;
- Toggle Inventory Window: “V” or “I” keys;
- Toggle Quests Window: “L” key;
- Toggle Pause Menu Window: “Esc” key;

> You can change the button that correspond to a given action through the Input Actions assets located at `Assets/PLAYER TWO/ARPG Project/Examples/Inputs` folder.

Getting Help

If you need any support, feel free to contact me at playertwopublisher@gmail.com. Please, ensure to provide your **Invoice Number** somewhere in the email to speed up the support request.

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

> Make sure to read the documentation first before asking for help.

Game

The **Game** component controls general aspects of the Game, such as base stats for Player’s progression, base combat stats, items attributes settings, base prefabs, and character data.

> All your scenes must have one `__GAME__` prefab on it.

> When making changes to any of the Game domain components, apply all the overrides to propagate its changes through all the scenes.

Game Audio

The **Game Audio** component is a singleton responsible for playing audio effects, gui audios, and music. It also keeps reference of general purpose clips.

Game Controller

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

Game Database

The **Game Database** is a component that keeps reference to lists of data that must be indexed somehow.

Game Pause

The **Game Pause** is a component to control the pausing state of the game in a single place.

Game Save

The **Game Save** is responsible to save and load the game data from the persistent memory. There are two different save modes available: Binary and JSON. You can also specify the file name. 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. https://docs.unity3d.com/ScriptReference/Application-persistentDataPath.html.

Game Scenes

The **Game Scenes** is a component that controls the scenes loading.

Game Stash

The **Game Stash** is a component that creates a list of inventories that are shared between characters, so the Player can store or transfer Items to other Characters.

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 state of a playable scene.

Creating a playable Level

When creating a new playable Level, make sure to add the `__LEVEL__` prefab to it, it's a Game Object with all important components in order to make a new Level.

In the Level component, you can define a "Player Origin" reference, which is a Transform from your scene that will be used as the initial position of the Player. When instantiated, the Player will use this transform's position and rotation to be positioned in the scene.

The Level component also have many lists of objects that it will keep track of. You must assign objects to these lists only if you want their state to be persisted by the saving system. For example, if you want a enemy to stay dead after loading the game, add it to the “Entities” list.

The `Level Waypoints` must have a reference to all the available waypoints from your scene, if any.

The `Level Audio` will play a music when the Level start, chose whatever music you want, or leave it blank.

Your level will also need other prefabs, such as:

- `__GAME__` ;
- `__CAMERA__` ;
- `__CANVAS__` ;
- `__MINIMAP__` (optional).

> Note that, different than the `__GAME__` prefab, the `__LEVEL__` must **NOT** have its overrides applied, since its different from scene to scene.

Character

The **Character** domain is a crucial aspect of any game that involves character development and progression. It is responsible for storing and managing the runtime data of each individual character from the game, such as their stats, inventory, equipment, and quests.

At the core of the Character domain is the `Character` scriptable object. This object defines the initial data for each character, including their starting scene, stats points, equipment, consumable items, inventory items and coins, and skills. This makes it easy for users to create new characters and ensure they have a solid foundation for growth and development.

To support the Character scriptable object, the domain includes other objects to represent each of the character’s elements. These objects include `CharacterEquipments` , which store the character’s equipped items, `CharacterInventory` , which tracks their inventory, `CharacterInventoryItem` , which represents a single item in their inventory, `CharacterItem` , which defines the item’s data, `CharacterItemAttribute` , which defines the item’s attributes, `CharacterQuests` , which tracks their active quests, `CharacterScenes` , which manages their

current scene, `CharacterSkills`, which stores their acquired skills, and `CharcaterStats`, which keeps track of their various stats.

The Character Instance is an object that is instantiated when a character is created and is based on the scriptable object's initial data. Once created, the game will use it to initialize all subsequent character objects and build the entire character representation.

In addition to managing the character's runtime data, the Character Instance is also used to create the player Entity. The player Entity represents the actual playable character in the game world and provides the interface through which the player can interact with the game.

By organizing and managing all of this data in the Character domain, game developers can create more dynamic and engaging characters that evolve over time, providing players with a more immersive and satisfying gameplay experience.

Creating a new Character

To create a new character in Unity, start by right-clicking inside a folder in the "Project" panel. Then, select Create > PLAYER TWO > ARPG Project > Character > Character. This will generate a new file named "New Character," which you can rename as needed.

Once the new character file is created, left-click on it to access the list of properties in the "Inspector" panel. Here, you can define the prefab for the character's Entity, as well as its initial scene, stats, equipment, consumables, inventory, and skills.

Before you begin creating your own character, I highly recommend taking a look at the built-in character to understand how it's built. You can find the built-in character data in the "PLAYER TWO/ARPG Project/Examples/Data/Characters/Warrior.asset" folder.

> Please note, while the Character object is an essential part of creating a character in the ARPG Project, it's important to understand that it represents data and not the in-game playable character itself, which is from the Entity domain. So, the Character domain has nothing to do with Player behavior, animations or any visuals in general.

> After creating a new Character, make sure to add it to the Characters list from the `Game Database`.

Entity

The **Entity** is a powerful component that leverages Unity's AI system and the Character Controller to enable fluid movement for both players and enemies. With a suite of methods available, it offers granular control over velocity, waypoint movement, and attacks based on items or skills.

Entity Inputs

The **Entity Inputs** is a component responsible for allowing the Entity to react to Player inputs. It's build upon the New Input System API.

Creating a new Character

To create a new playable character, you'll need to create a new empty Game Object in your scene. Add the Entity component to it by dragging and dropping from the "Core" folder or through the "Add Component" menu by searching for "Entity". In the Entity component you can configure the base move and rotation speed, as well as the minimum attack distance, and the ground snap force, to keep the Player attached to the ground.

The "Target Tag" attribute is used to identify the tag of other entities that will be considered as enemies, allowing your Player to track and attack them.

The "Hitbox" attribute is a reference to a object with the `Entity Hitbox` component attached to it. Your entities must have one if you want to allow melee attacks. The Hitbox is a Game Object child of your Entity with a Collider component of any shape and the "Entity Hitbox" component attached to it.

To allow your Player to be detected by enemies and other systems, change its tag to "Entity/Player".

Use the `Entity Inputs` component to allow your Entity to be controlled by Inputs. Make sure to add an Input Action asset to it, with all your inputs defined. Also, you'll need at least one Camera in your scene tagged as "MainCamera," or else the Input System will be unable to Raycast and trigger the Entity's actions, and your console will show error messages.

Other required components to make your Character work are:

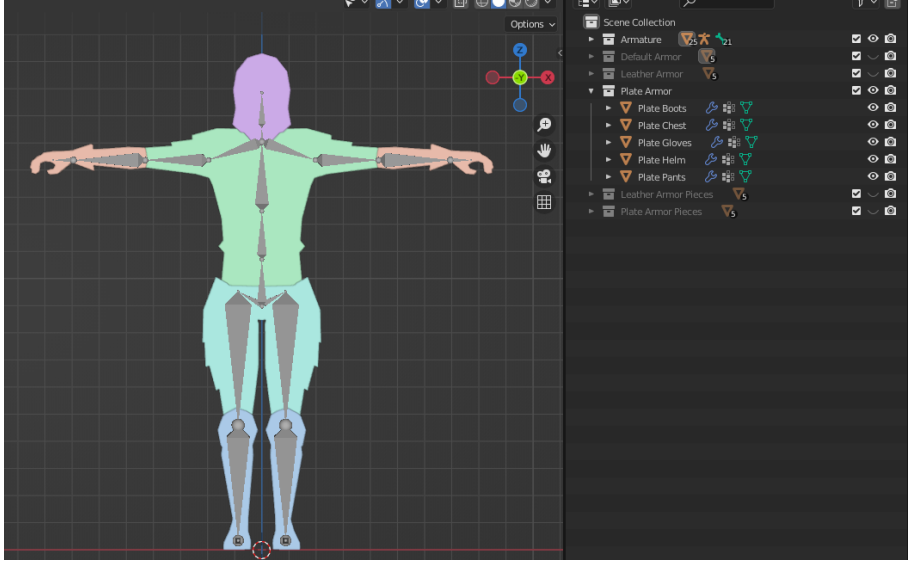
- Entity Stats Manager;
- Entity Skill Manager;

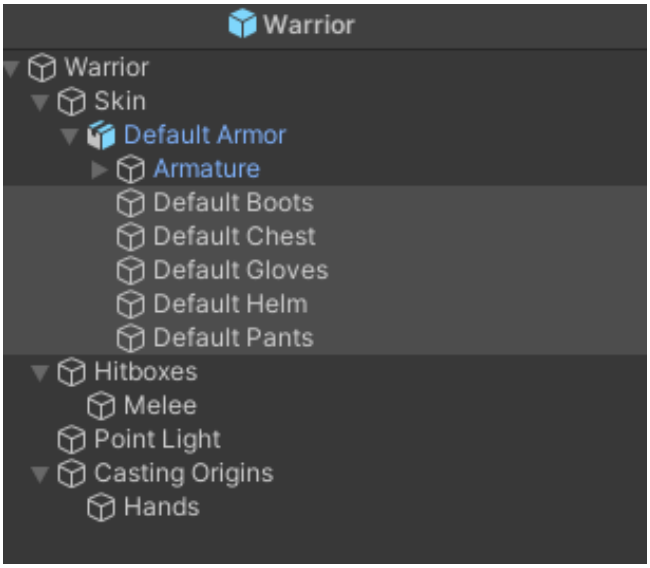
- Entity Item Manager;
- Entity Inventory;
- Entity Animator.

Most of these component don't need additional configuration, but requires some references to be set. For example, the `Entity Skill Manager` requires a casting origin to be used as a reference for the Player's hands, but not the actual hand bone, only a transform in front of the Player that closely matches the desired origin of hand casted skills. The `Entity Item Manager` also requires some references, like transforms to be used as weapon slots, the projectile origin to be used when soothing arrows, and the skinned mesh renderers which will be used by the armor system.

The armor system works by replacing each mesh rendering corresponded to a body part. To make it works, you must export your character separated by helm, chest, gloves, pants, and boots in a single FBX file. Note that, despite being separated in collections on Blender, all the armor parts shares the same bone structure. Organizing it in this way makes it easier to disable/enable the armor groups in Blender and to export each armor as individual FBX files.

> By design, the default model of your character is also structured as an armor.



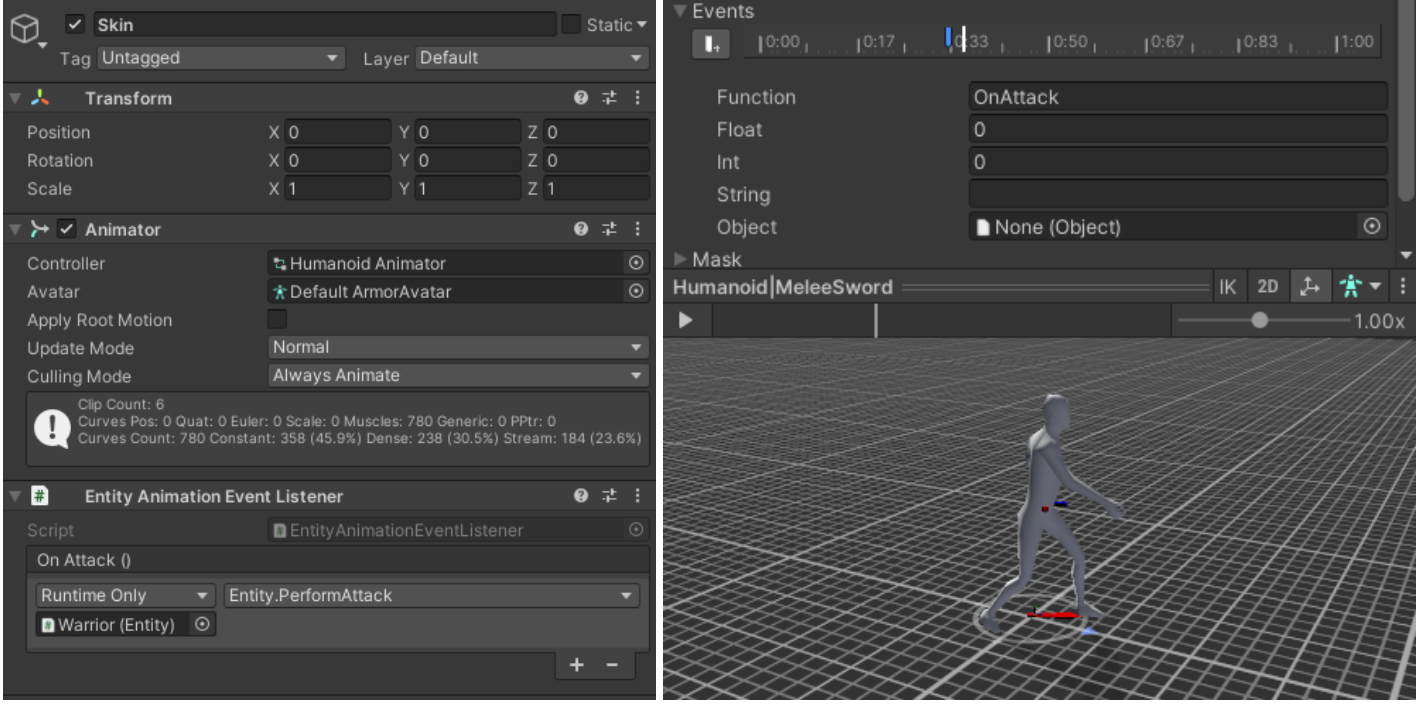


> Note that this armor system is based on mesh renderers, so there's no need to export all armors at once, nor having thousands of inactive Game Objects on your character's model hierarchy. You'll only need a way to reference each mesh renderer individually.

> In order to deform properly when animating, all the meshes must be skinned to the same bones.

You can download the Blend file I used to create the armors from here: https://drive.google.com/file/d/14klLc34uc2iQdjk0QKXnlZN2HLqE9Qos/view.

The `Entity Animator` is not optional because the attack action will be triggered by animation events. So you must have an `Entity Animation Event Listener` component attached to the same object that contains your Animator component. Also, make sure to call the "Perform Attack" from the Entity component in the Unity Event "On Attack" from the Event Listener.



> Make sure to create the events “OnAttack” in the Animation Import settings if you’re using your own animations (the default ones are all configured). This event will trigger both melee and skill attacks based on the context. https://docs.unity3d.com/Manual/AnimationEventsOnImportedClips.html.

The `Entity Animator` also uses Animator Override Controller to manage the Character’s stance based on the items it’s equipping. It looks fancy, specially if you have never used it before, but it’s actually pretty simples. Animator Override Controllers are just a “list” of animation clips that overrides the default animations from a given animator. For example, if the Character is using a sword, the Entity Animator will override the default “Idle animation” by the “Idle With Sword”.

> If you need more information about animator overrides, please visit the official documentation: https://docs.unity3d.com/Manual/AnimatorOverrideController.html

There are other components you can use to bring your character to life, such as:

- Entity Audio;
- Entity Feedback;
- Minimap Icon.

Entity AI

The **Entity AI** is a component used to move Entities using AI behaviors, such as patrolling, target searching, melee attacks, and skill attacks.

Creating a new Enemy

The process for creating an enemy follows the same principles as for creating a Character. In fact, Characters and Enemies use most of the same components, as they are fundamentally the same thing.

The main difference between Characters and Enemies is that enemies do not need to respond to input data. Instead, they require a special component called `Entity AI` to move around, which allows the Entity to seek targets and perform skills or melee attacks.

As with Characters, the “Target Tag” is used to identify the enemies’ targets, in other words, the Player. Therefore, set the target tag to your Player tag, and be sure to tag the enemy as “Entity/Enemy” so that it can be identified by other systems.

For enemies that use skills, make sure to check the Infinite Mana toggle in the “Bot Settings” section under the `Entity Stats Manager`. This allows your enemy to attack without losing Mana points. You may also want to uncheck the Can Level Up toggle so that the enemy does not earn experience by killing the Player.

> Since enemies are not initialized by the Character system, you will need to manually set their initial weapons and skills if you want them to use any. Also, note that enemies do not need the `Entity Inventory` component. Additionally, if you want to create armored enemies, build a variant with increased stats and equip it with the armor model instead of relying on the armor system.

If your enemy has custom animations, set the “Default Animations” override from the `Entity Animator` for one containing all of your custom animations.

> To make it easier for the Player to click on the Enemy, add an additional collider to it with bigger bounds and toggle the “Is Trigger” property.

Entity Stats Manager

The **Entity Stats Manager** calculates all the stats from the Entity, such as how much damage it applies, how much defense it has, how much mana and health points, etc. The Stats Manager also communicates with the Entity Item Manager to modify the stats based on the Item’s attributes.

Entity Animator

The **Entity Animator** is responsible for controlling the stances of a Entity based on the Entity Item Manager. It works by combining Unity’s `Animator Controller` and `Animator Override Controllers`, allowing the default animation system to be clean while supporting a variation of different animations based on the items in use. For example, the Entity can play a different animation while holding a sword or a bow.

Entity Animation Event Listener

The **Entity Animation Event Listener** is a component that propagates animation events created from the Animation import settings as Unity Events. To make it works, all you need to do is attach this component to a Game Object that also contains an Animator component.

Entity Audio

The **Entity Audio** component provides audio feedback for specific entity actions. For example, you can define multiple audio clips, having a wide range of audio variations, to be used when the Entity takes a regular or a critical hit. You can specify audios to play when it spots a target, which is great for enemy sounds.

Entity Camera

The **Entity Camera** component must be attached to your gameplay camera. It reproduces common behaviors on ARPG games, such as zoom In/Out, and rotation. The Entity Camera also uses its own set of Input Actions, using the New Input System.

Entity Feedback

The **Entity Feedback** is used to represent the damage. You can create a `Damage Text` object to show the entity's received damage in runtime. It's also to give for the Player some feedback of attack given or received.

Entity Hitbox

The **Entity Hitbox** is built to be used together with a Collider component of any shape. It applies damage to the entities inside the collider, in which its tag matches the "Target Tag" from your Entity, for a specific amount of time. You can also define a "default toggle duration," which is the default amount of time this hitbox will keep applying damage. By default, it only applies damage to one Entity at time.

Entity Inventory

The **Entity Inventory** is a component to store Item Instances.

> You'll need to adjust the GUI spacing if you decide to change the inventory capacity.

> The default cell size was hard coded to 52 pixels. You can change it in the Inventory file, but updating the GUI will be harder than just changing the capacity.

Entity Item Manager

The **Entity Item Manager** is a component to manage the items equipped by the Entity. It's different from the Entity Inventory, which only stores the items. It also manages the consumable items. The item manager references Transforms from the Game Object to be used as item slots, such as weapon slots. It's also responsible for managing the mesh renderers sued to equip armors visually.

> Only configure initial items if your Entity is a NPC. Characters will override these settings.

Entity Skill Manager

The **Entity Skill Manager** is a component to manage Entity Skills. It also keeps a reference to the casting origin that represents the hands.

> Only configure initial skills if your Entity is a NPC. Characters will override these settings.

Collectible

The **Collectible** is a Interactive abstract object that can hold an Item Instance data, when using the `Collectible Item` implementation, or a money amount, when using the `Collectible Money` implementation.

There's no need create new collectible prefabs. The collectible items and coins are generated in real-time using the Item Loot System, using two base prefabs, so you don't need to create a new one for each of your items.

You can take a look at the default collectibles base prefabs at `PLAYER TWO/ARPG Project/Examples/Misc` folder, and they are named as `Collectible Item` and `Collectible Money`. If you ever need to replace them, make sure to replace its references in the `__GAME__` prefab, under the "Collectible Prefabs" section from the Game component.

Creating new Collectibles

As mentioned, the collectible item itself is generated in real-time, the collectible prefabs are just a base. But if you need to create a new one, just make an empty Game Object, add a Collider component of your preferable shape, and add the Collectible script by dragging and dropping it, or through the “Add Component” button searching for “Collectible Item” or “Collectible Money”.

After that, save your Game Object as a prefab by dragging and dropping it in a folder from the “Project” panel.

> Collectibles are generally instantiated in runtime by the Item Loot component.

Items

This section covers the process of creating new items in the game. It provides step-by-step instructions on how to create different types of items, such as armor, weapons, potions, and shields, as well as how to create skill items that allow characters to add new skills to their list of available abilities.

Creating new Items

To create new items, simply right-click inside a folder in the “Project” panel and open the “Create > PLAYER TWO > ARPG Project > Item” section. Here, you’ll find a variety of items that you can create, including:

- Item Armor;
- Item Blade;
- Item Bow;
- Item Potion;
- Item Shields;
- Item Skill.

Regardless of the item you’re creating, all items will share certain attributes:

- Prefab: a Game Object that represents your item in the game world (can be its model);
- Price: the base cost of the item;
- Drop settings: the position and rotation offset of the item when dropped;
- Inventory settings: the item sprite, size, and stacking settings in the inventory.

Equipable items, such as armors, weapons, and shields, have additional attributes:

- Max durability: the maximum durability points the item has
- Required level: the minimum level the character must have to equip the item
- Required strength: the minimum strength the character must have to equip the item
- Required dexterity: the minimum dexterity the character must have to equip the item
- Weapon items, like blades and bows, have the following additional attributes:

Weapon items, like blades and bows, have the following additional attributes:

- Min damage: the minimum damage the weapon can cause
- Max damage: the maximum damage the weapon can cause
- Attack speed: the additional attack speed the weapon provides
- Attack clips: a collection of audio files that the weapon plays when being used

> When creating an item, be sure to assign it to the items list in the `Game Database` .

Creating new Armors

To create a new armor item, select “Armor” from the item creation menu. Note that the armor item only represents one piece of the armor, and you don’t need all the pieces to make it work. You must configure which part of the body this armor represents by changing the “Slot” property under “Armor Settings.” The “Rendering Settings” will define the mesh this armor will use to replace the skinned mesh renderer from the entity. You must also set the materials it will use in order.

> For more details about armor system setup, please visit the character creation section.

Creating new Blades

To create a new blade item, select “Blade” from the item creation menu. You can configure the blade to be single-handed or double-handed by changing the “Type” attribute under “Blade Settings”. You can also configure a position and rotation offset that will be applied to the blade game object when it’s instantiated in both character’s hands.

Creating new Bows

To create a new bow item, select “Bow” from the item creation menu. You can configure the bow to be a bow or a crossbow. The only difference will be the animator override they’ll use and the position of the item in the character’s hand. The projectile is a prefab that the bow instantiates when being used to attack, and the shot distance is the maximum distance that the target can be before the entity tries to attack it. Use the “Hand Settings” to adjust the item’s position in the entity’s hand.

Creating new Potions

To create a new potion item, select “Potion” from the item creation menu. The potions will have “Healing Settings”, which determine the amount of health or mana points they’ll restore when consumed.

Creating new Shields

To create a new shield item, select “Shield” from the item creation menu. You can adjust the defense points the entity will get by equipping the shield. You can also adjust the position of the game object in the entity’s hand by adjusting the transform settings.

Creating new Skills (Item)

To create a new skill item, go to the item creation menu and select “Item Skill”. It’s important to note that an Item Skill is not the skill itself, but rather an item that holds a reference to a skill. This allows the character to add the skill to their list of available skills if their stats meet the requirements.

Item Loot

The **Item Loot** component is used to drop items based on a set of properties provided by the Loot Stats. To use it, simply assign the `Item Loot` component to any Game Object you wish and then assign a `Loot Stats` to it. You can activate the loot by calling its public method, `Loot` .

> If the `Entity` component is added to the Game Object, the loot will automatically drop the items when the Entity dies.

Creating new Loot Stats

To create a new Loot Stats asset, right-click on a folder in the “Project” panel and select “Create > PLAYER TWO > ARPG Project > Item Loot Stats.” The Loot Stats have several attributes, including the loot chance, loop count, position settings, attribute settings, items, and money chance.

The loot chance attribute allows you to configure the chance of this loot dropping any items or money, on a scale of 0 to 1, where 0 means 0% chance and 1 means 100% chance.

If the loop count attribute is greater than zero, the loot will randomly pick a new item or money for each iteration.

The position settings section provides configuration options for the positioning of items. If you disable the “Random Position” property, no parameters under this section will take effect, and the items will always drop in the center. If you increase the radius, the items will drop in random positions within the specified range.

The attribute settings allow you to adjust if the items dropped have any additional attributes, as well as how many based on a minimum and maximum quantity. If the quantity is 0, the constraint will not be applied. This setting will only affect the attributes for Weapons and Armor items.

The items attribute lists the items that this loot can drop. The loot system will randomly pick an item for each loop.

Finally, the money chance attribute allows you to specify the chance of the loot dropping coins instead of an item, as well as the amount of coins that will be dropped.

Skills

This section provides instructions on how to create new skills and attack skills in ARPG Project. The text describes the steps that the user needs to follow to create a new skill, including the different types of skills that can be created, the properties that can be configured for a skill, and notes on how skills are casted to the target or the mouse position. Additionally, the text provides instructions on creating a new Attack Skill and the additional properties that can be configured for it.

Creating a new Skill

To create a new skill, right-click inside a folder in the “Project” panel and open the “Create > PLAYER TWO > ARPG Project > Skill” section. You will find a variety of skills you can create, including:

- Skill;
- Skill Attack.

Regardless of which one you decide to create, all skills will have the following properties:

- General Settings: where you can configure the skill’s icon, sound, and cooldown duration.
- Casting Settings: where you can define the visual representation that the skill will use and the origin of the cast, which can be on the caster or the target.
- Target Settings: where you can specify whether the skill can only be used against a target and if it must face the target direction.
- Animation Settings: where you can configure if the skill uses the default attack animation or if it will apply an animator override.
- Mana Settings: where you can specify if the skill’s usage consumes any mana and how much.
- Healing Settings: where you can configure if the skill restores the entity’s health and by how much.

> When the caster entity is tagged as a Player, skills configured with the cast origin to “Target” will be cast to the mouse position if no target is available. For example, when performing a free attack.

> To allow your Character to somehow equip your new skills, assign them to Item Skills.

> When creating a new skill, make sure to assign it to the Skills list in the `Game Database` .

Creating a new Attack Skill

To create a new attack skill, select “Attack Skill” under the Skill creation menu. Attack Skill will have the following additional properties:

- Use Melee Hitbox: If toggled, your skill will activate the Entity hitbox to perform its attack.
- Min Attack Distance: The minimum distance that allows the entity to perform this skill.
- Required Weapon: Set this property if your skill requires a specific weapon. For example, a blade attack skill must require a blade to be equipped.

- Damage Mode: This option will affect the calculation of the damage. “Regular” will use strength, and “Magic” will use energy.
- Min/Max Damage: Here, you can configure the additional damage this skill provides.

Skill Particle

The **Skill Particle** is a specialized component that you can assign to a Game Object with a Particle System. This component is particularly useful if you want a Particle System to deal damage to an entity’s target. You can also configure it to use particle collision events for more precise particle attacks. If you’re not using the collision events, be sure to add a Collider component to the same object that uses this component, or else your particle will not cause any damage. When you assign a prefab with this component to the “Casting Object,” the skill system automatically transfers the damage data to this component.

Quests

The **Quest System** adds an engaging layer to your game by giving players tasks to complete. These tasks can be created to collect items, defeat enemies, reach a specific location, or be manually triggered from custom code.

Creating new Quests

To create a new quest, right-click on a folder in the “Project” panel and select “Create > PLAYER TWO > Quest > Quest.” Name the file as desired. The main settings for the quest are related to its basic information, such as the Title, Description, Objective, Target Progress, and Completing Mode. Note that the Target Progress is only necessary if the Completing Mode is set to “Progress.”

If the Completing Mode is set to “Reach Scene,” the quest will automatically finish when the Destination Scene is reached. The Destination Scene must match the name of the scene in the Build Settings.

If the Completing Mode is set to “Progress,” the quest will be completed by collecting items or defeating enemies. You can also define a custom progress system by specifying a progression key and calling the AddProgress method from the Quest Manager, which can be accessed through the Character Instance.

If the Completing Mode is set to “Trigger,” the quest will only be completed by manually triggering it using the Trigger method from the Quest Manager. This Completing Mode is useful if you need to complete a quest using custom events in your game.

Under the Reward Settings, you can configure whether or not your Quest will provide rewards such as experience points, coins, or items. The reward will only be given if the “Has Reward” option is toggled and it will automatically go to the player as soon as the quest is completed.

> It is important to note that the player can only progress, trigger, or complete quests that have been accepted. Additionally, if multiple quests have the same destination scene or progress key, all of them will receive progress, but only if they have been accepted by the player.

Quest Giver

The **Quest Giver** is a Interactive Game Object responsible for assigning quests to the Player, so they can show up in the Quests Log. By interacting with them, a window will show up with all the quests details.

Creating Quest Givers

Creating a Quest Giver is easy. You simply need to assign the Quest Giver component to any Game Object in your scene and then add one or more quests to the quests list. As quests are completed, the Quest Giver will automatically show the next quest from the list.

> To make the Quest Giver more noticeable, it is recommended to add a Highlighter component so that the Game Object will become brighter when the player hovers the mouse on it.

Quest Enemy

The **Quest Enemy** is a component that can be attached to entities designed as enemies. By assigning an “Enemy key” to it, every time an enemy with this component is defeated, it will add progress to accepted quests where the “Completing Mode” is set to “Progress” and the “Progress Key” matches the key from the Quest Enemy component.

Quest Item

The **Quest Item** is an Interactive object that adds progress to quests that have a matching “Progression Key” and “Item Key” property. It is important to note that it will only add progress to active quests where the “Completing Mode” is set to “Progress.” The “Show Only When Quest Is Active” toggle can also be used to make the Game Object active only when a Quest with matching keys is accepted.

Quest Trigger

The **Quest Trigger** component is an example of how the “Completing Mode” can be used to complete a quest. In this case, the Quest Trigger component works by triggering the Quest completion when the player enters the Collider trigger area. To make it work, a Collider component of any shape must be added to the same Game Object attached with the Quest Trigger component.

Minimap

The **Minimap** works by saving an orthographic picture of the scene as a PNG file. This way, it’s possible for you to edit it in a photo editor program to make it more appealing. It’s also different from solutions using multiple cameras, that you generally see in tutorials on the internet. Making it more performative, since there’s no need to render the game scene more than once.

Creating a Minimap

To create a Minimap, drag and drop the `__MINIMAP__` prefab from the “PLAYER TWO > ARPG Project > Examples > Prefabs > Global” folder, or create a new Game Object and add the `Minimap` component to it. Adjust the settings under the “Map Volume” section to fit your entire scene inside the

yellow bounds. After that, tweak the “Capture Settings” as you like and hit the “Generate Minimap Texture” button.

The generated minimap image will be saved to a folder in the same directory of your scene matching your scene’s name. It’s also going to assign the generated texture automatically to the “Minimap Texture” field of the Minimap component.

> If you want to hide Game Objects from the captured image, you can either move them to a different Layer that is not captured by the Culling Mask, or disable these Game Objects from the Hierarchy.

Minimap Icon

The **Minimap Icon** is a component that can be added to any Game Object in order to make it appear in the Minimap. You can adjust the icon’s sprite initial scale and initial rotation. The “Rotation With Owner” property will make the icon rotate if the Game Object its attached to rotates in the game world.

> When adding Minimap Icon to an Entity, it’ll be automatically disable when the Entity is defeated.

Miscellaneous

Miscellaneous are components that are not complex enough to be separated on their domains.

Merchant

The **Merchant** is an Interactive component that can sell and buy items for the Player. The Merchant sections are divided by name and each of them can have multiple items. It can also sell Items with additional attributes by changing the “Attributes” properties, but will only affect equipable items, such as Weapons, Armors, and Shields.

The pricing system takes into consideration the base price of the item, its durability, and the amount of attributes it has. The base price is taken from the items data asset. You can configure the additional price per attribute in the Game component.

The durability rate, which is the current durability of the item divided by its max durability, will be multiplied by the price. So, if your item is broken, its price will be zero. Otherwise, if it's durability is at half, the final price will be half of the current price.

> Note that the merchant will always buy items by half of their final price.

> If the buy back section of the Merchant is full, the next items you sell will be discarded.

Blacksmith

The **Blacksmith** is a component that can be used to fix the Player's equipable items. It can repair individual items or the entire inventory. The final cost of the repairing is determined by the durability of the item, but if it's at 100% durability, the price will be zero. As the durability decreases, the repairing price increases from the min to the max repair cost. The cost to repair all the inventory is the sum of the cost of each item.

Waypoint

The **Waypoint** is a component that allows the Player to flash travel around the scene. You can setup them to automatically activates when the Player gets closer using the "Activation Radius." You can also specify the name of the Waypoint, so it can be easily identified by the Player when selecting it from the Waypoints Window.

> To save the Waypoints, add them to the Waypoints list from the Level Waypoints.

Destructible

The **Destructible** is a component that can be attached to any Game Object in order to make it receive damage from the Player. You can assign a regular and an “cracked” Game Object to represent its state. You can also combine it with the Item Loot to make it drop items.

> To save their state, add the destructible objects to the Game Objects list from Level.