ARPG Attributes, Items, and Abilities

Quick Links:

- Link to Online Document (most updated) (this document last updated 04/29/2021)
- Link to upcoming changes (1.22.0 - EXTENSIVE changes)
- Quick Start Guide
- Discord Invite
- Bug Submit Form

Playable Windows and WebGL Demo.

Click Here to Provide Feedback.

Website

Table of Contents

Playable Windows and WebGL Demo.

Click Here to Provide Feedback.

Website

RoadMap

Plans are subject to change based on available feedback from users.

Systems that will receive only minor, if any, updates.

- Attributes
- Items
- Equipment
- Looting
- Leveling
- Equipment Traits

Systems that will receive major planned updates in the future

- ~~Auras (Late October 2020)~~
- ~~Skills (renamed to Abilities) (Late October 2020)~~
- ~~Quests (January 2021)~~

Systems I'm keeping an eye on for possible revision

- ~~Status Changes~~

Unnamed systems are not a priority unless feedback indicates otherwise.

Animations and states will likely receive minor updates as new types of abilities and auras are added.

Planned deadlines

Late October 2020 Update

- ~~Migration to 2019.4 LTS version.~~
- ~~Major update for Auras that represents a 'final' version.~~
- ~~Major update for Skills (renamed to Abilities) that represents a 'final' version.~~
- ~~Inclusion of Progress Trees and an example with an ability tree.~~

January 2021 Update

- ~~Quest system that tracks progress by building on the progress trees.~~
- ~~Refinements to Abilities and Auras.~~
- ~~Minor changes to the save databases to work with the new Auras and Abilities.~~
- ~~Vendors.~~
- ~~Dual Wielding.~~
- ~~Editor Database Editing Window~~

ARPG Attributes, Items, and Abilities

April 2021 Update

- ~~Refine Master Database~~
- ~~Refine Actor Creator~~
- ~~Refine Wearable Creation~~
- ~~Refine Editor Workflow in general~~
- ~~Exporter to CSV (export the jsons to external spreadsheet editor, import them back into the project).~~
- ~~2D Triggers / Works in 2D now~~

June 2021 Update

- Factions
- Architectural Design Documents
- Documentation Update
- Interact Priority Class
- New Interact Detection Types
- Button based Ability Detection (hold A to charge ability, etc)
- Revisiting Enemy Movement and Player Movement Systems

Architectural Design

When possible, the systems are designed with a scriptable object that stores the data and an interface that links that data to an instance.

Typically the design follows that an instance implements an interface and that interface communicates to the relevant data (Scriptable Object, mainly). For instance, the 'Player Attributes' included in the demo implements the interface IAttributeUser and communicates with ActorAttributes (Scriptable Object).

The design maintains the ability to slap an interface onto existing code, and by implementing it you'll achieve the same functionality as what's in the demo.

Ability System

- An Ability User (IAbilityUser) calls into the Ability Controller with TryCastAbility by passing the Ability as a parameter. If the conditions are met, the Ability Controller starts the cooldown timer, delay timer, and Ability logic. Abilities must be equipped and learned by the AbilityController in order to use.

Attribute System

- An Attribute User (IAttributeUser) calls into the ActorAttributes to modify Attributes by AttributeType.

Inventory System

- An Inventory User (IInventoryUser) calls into the ActorInventory to modify Inventory items and Equipment.
- The ActorInventory also will check for a clothing interface (IWearClothing) and a Wearable Prefab (IWearable). Upon finding both, it will tell the user of IWearClothing to equip the object that derives from IWearable at the appropriate EquipmentSlot (Mapped on the object instance. An example is Clothing script on the player in the demo).

Aura System

- An Aura User (IAuraUser) calls into the AuraController with ToggleAura by passing an Aura as the parameter. Auras must be equipped and learned by the AuraController in order to use.
- An Aura Receiver receives the effects of an Aura. This means you can have things that cast auras be immune to their effects, or things that don't cast auras receive their effects.

Shopping System

- The ShopKeeper inherits IShopKeeper and IInteract. A shopper (IShopper) calls into IInteract.DoInteraction() and then the ShopKeeper performs OpenShop.

Questing System

- The QuestchainGiver inherits from IInteract and IQuestGiver. A quest user (IQuestUser) calls into IInteract.DoInteraction() and then the QuestchainGiver performs OpenQuestDialogue() and SetupQuest().

Basics

What I consider the fundamental pieces to the system. Everything is built from **Actor Stat**, **Inventory**, and **Items** classes. The rest branch out from these 3 pillars.

The common interfaces include IAttributeUser and IInventoryUser. Items is an abstract base class.

Ability Controller

The Ability Controller is a scriptable object that contains an actor's ability information such as tracking learned abilities, equipped abilities, and conditions for use. The main call is **TryCastAbility(Ability theAbility)**.

<u>Method 1:</u>

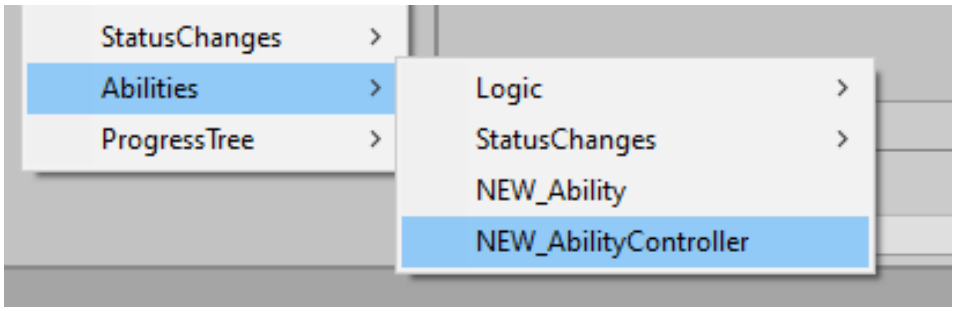
Duplicate an existing Class found under

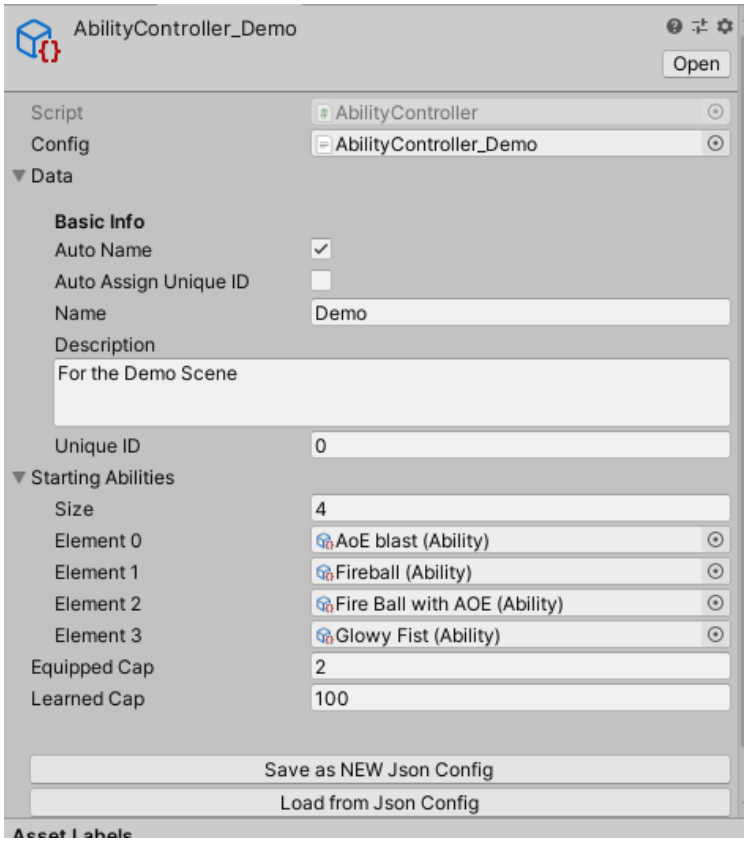
- *Assets/GWLPXL/ARPG/Data/Abilities/Controllers*

<u>Method 2:</u>

Right Click in the *Project Window*

- -> Create -> GWLPXL -> ARPG -> Abilities -> NEW_AbilityController





- *Equipped Cap* refers to the Abilities that are currently equipped (e.g. on your hotbar)
- *Learned Cap* refers to the number of Abilities in total one can learn.

Abilities

The Ability scriptable object holds the identifying information about a unique ability. It also contains an array of logics that perform the behavior once **StartCastAbility** has been called (typically through the ability controller, but not necessarily). In this example, the logics include the generic melee logic which enables and disables the damage components and a fire area of effect that leaves a damage over time fire effect.

<u>Method 1:</u>

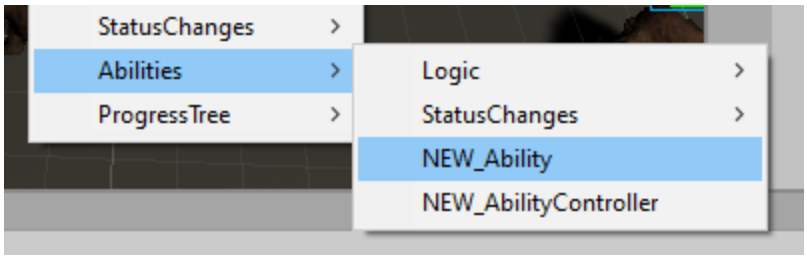
Duplicate an existing Class found under

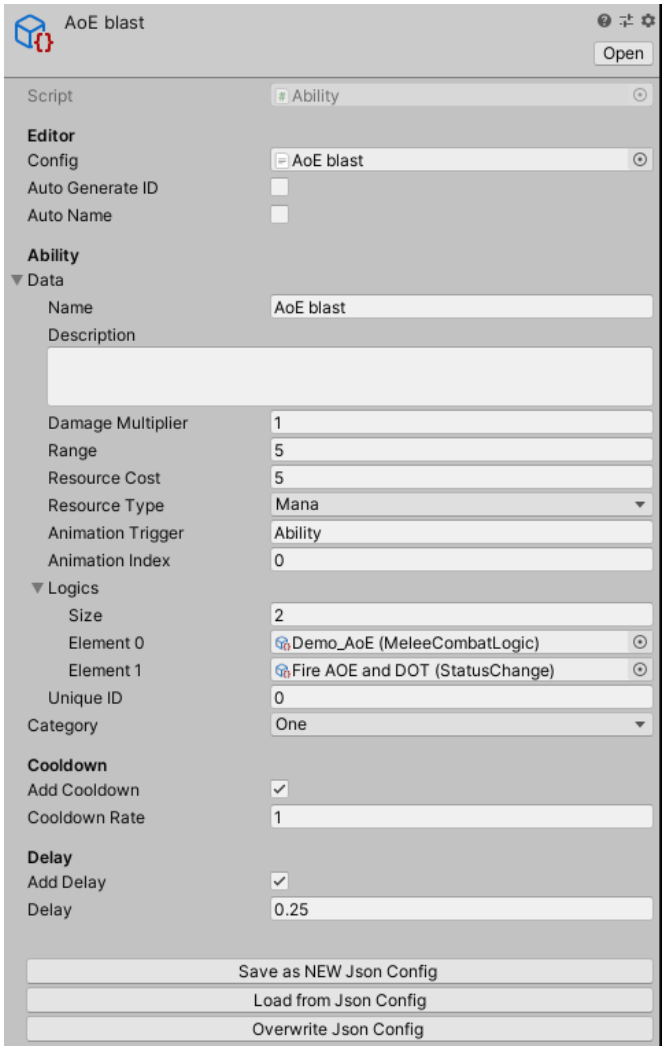
- *Assets/GWLPXL/ARPG/Data/Abilities/Ability*

<u>Method 2:</u>

Right Click in the *Project Window*

- -> Create -> GWLPXL -> ARPG -> Abilities -> NEW_Ability





- *Logics* initiate the gameplay behavior.
- *Category* limits an Ability's use. There can only be one ability in use at a time per category.
- *Cooldown* determines the Ability's duration.
- *Delay* determines a delay before the *Logics* initiate (e.g. you can delay the logic to time out with the animation).

Actor Class

Actor Class is a **Scriptable Object** that limits an actor's equippable equipment. If the class is left **null**, then the system assumes you have no restrictions.

<u>Method 1:</u>

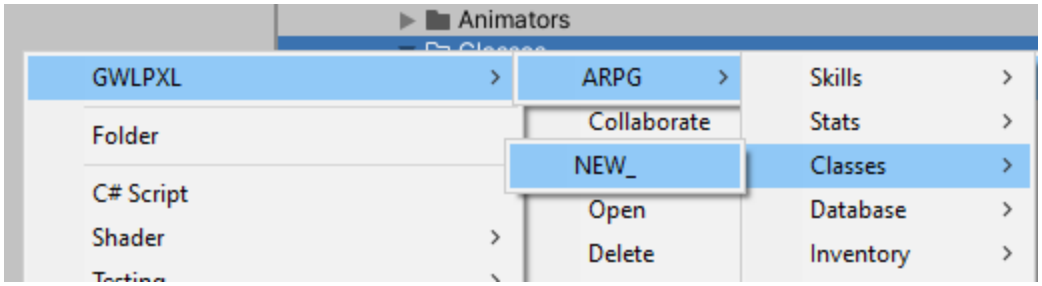
Duplicate an existing Class found under

- *Assets/GWLPXL/ARPG/Data/Classes*

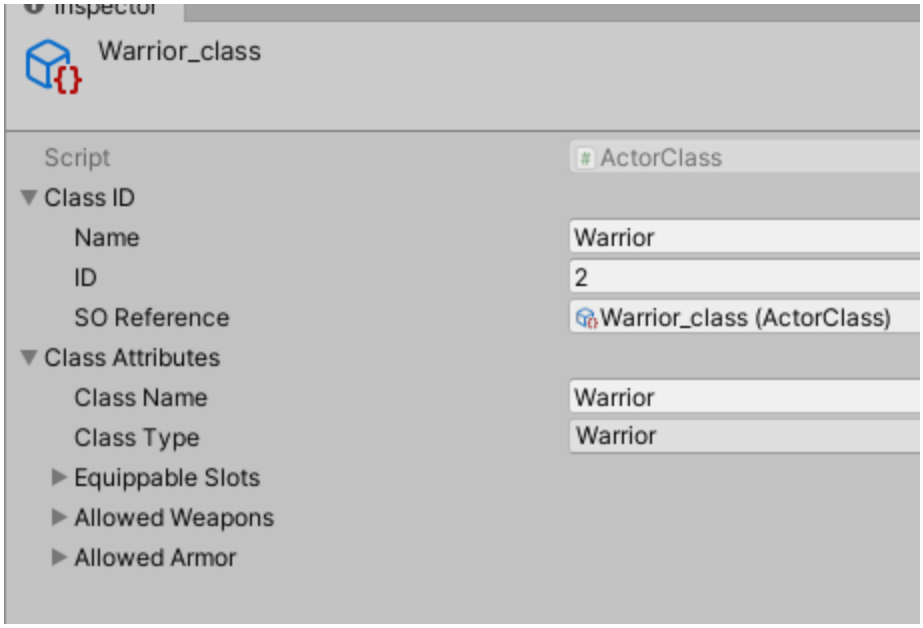
<u>Method 2:</u>

Right Click in the *Project Window*

- -> Create -> GWLPXL -> ARPG -> Classes -> NEW_



Actor Classes can be assigned to actors and used to limit their equipment capabilities.



- *Class ID* is for saving purposes, don't touch unless you know what you're doing.
- *Class Name* is the name the player sees.

- *Class Type* defines the type. Refer to [Creating a New Type of Actor Class](#) to create custom classes.

Class Type	Warrior
▼ Equippable Slots	
Size	16
Element 0	Head
Element 1	Chest
Element 2	Waist
Element 3	Legs
Element 4	Feet
Element 5	Left Shoulder
Element 6	Right Shoulder
Element 7	Left Hand
Element 8	Right Hand
Element 9	Neck
Element 10	Left Ring
Element 11	Right Ring
Element 12	Left Glove
Element 13	Right Glove
Element 14	Left Bracer
Element 15	Right Bracer

Equippable Slots define where equipment is located on an actor. Refer to [Creating a Custom Slot](#) to learn how to create custom slots.

▼ Allowed Weapons	
Size	4
Element 0	None
Element 1	Sword
Element 2	Axe
Element 3	Staff
▼ Allowed Armor	
Size	5
Element 0	None
Element 1	Cloth
Element 2	Leather
Element 3	Mail
Element 4	Plate

Allowed Weapons define which *WeaponTypes* can be equipped by the class. Refer to [Creating a New Type of Weapon](#) to learn how to create custom types.

Allowed Armor define which *ArmorMaterialTypes* can be equipped by the class. Refer to Creating a new Type of Armor to learn how to create custom types.

Actor Attributes

The Actor Stats class is a **Scriptable Object** that contains an actor's individual Attributes, i.e. *Stats*, *Resources, Elements*, and *Other.*

Method 1:

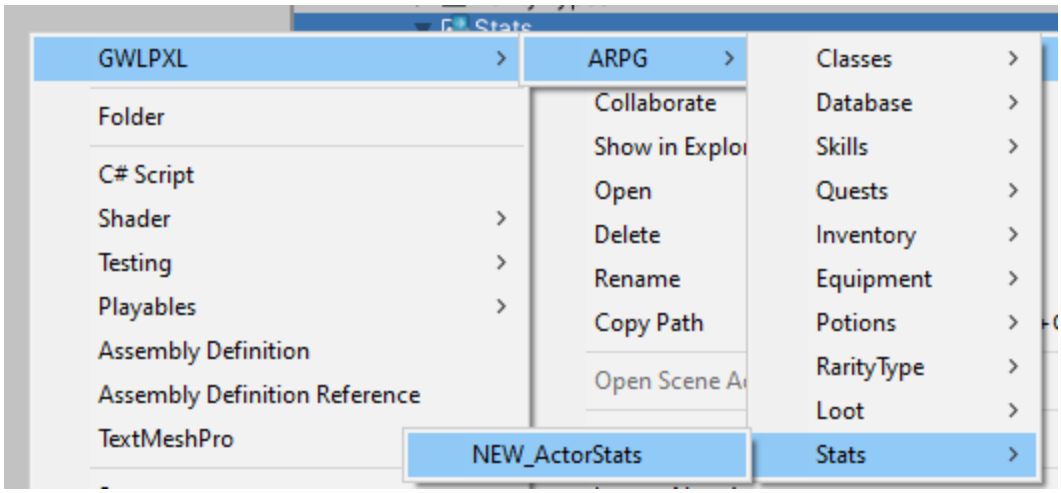
Duplicate an existing actor stats found under

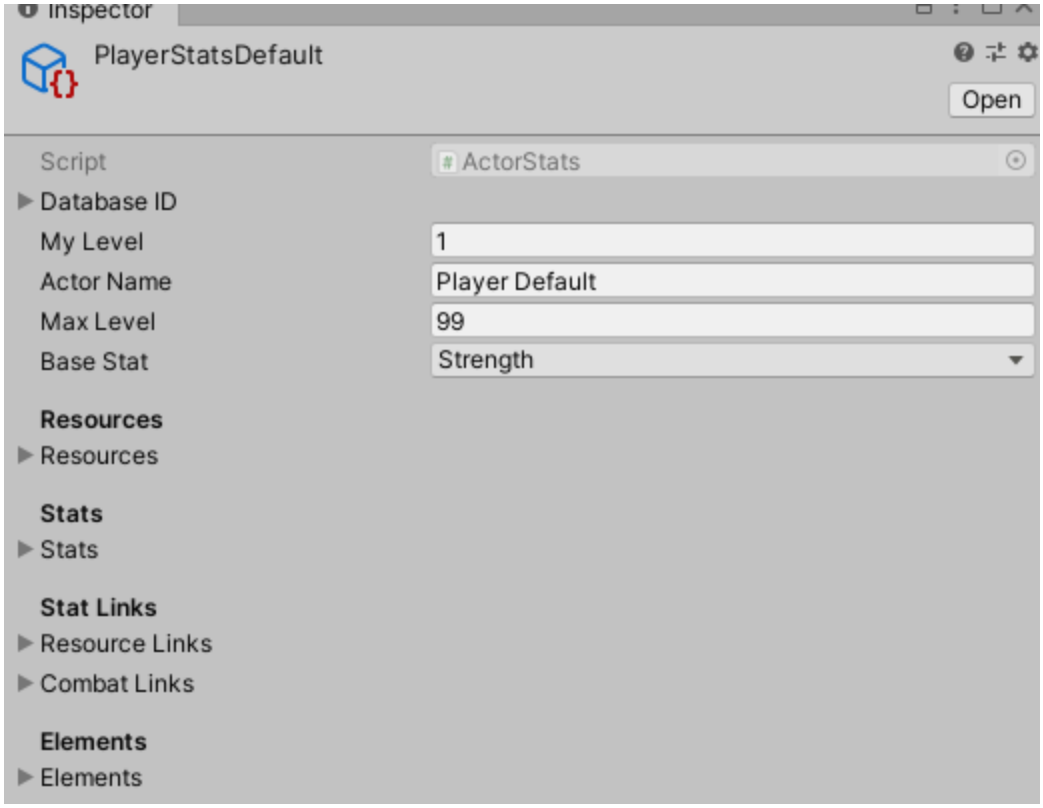
- *Assets/GWLPXL/ARPG/Data/ItemStatsInventory/ActorStats*

Method 2:

Right Click in the *Project Window*

- -> Create -> GWLPXL -> ARPG -> Stats -> NEW_ActorStats





- *Database ID* is for saving purposes, don't change unless you know what you're doing.
- *My Level* refers to the stat user's current level.
- *Actor Name* refers to the stat user's name.
- *Max Level* will cap the stat user's level.
- *Base Stat* refers to a primary stat for the stat user.

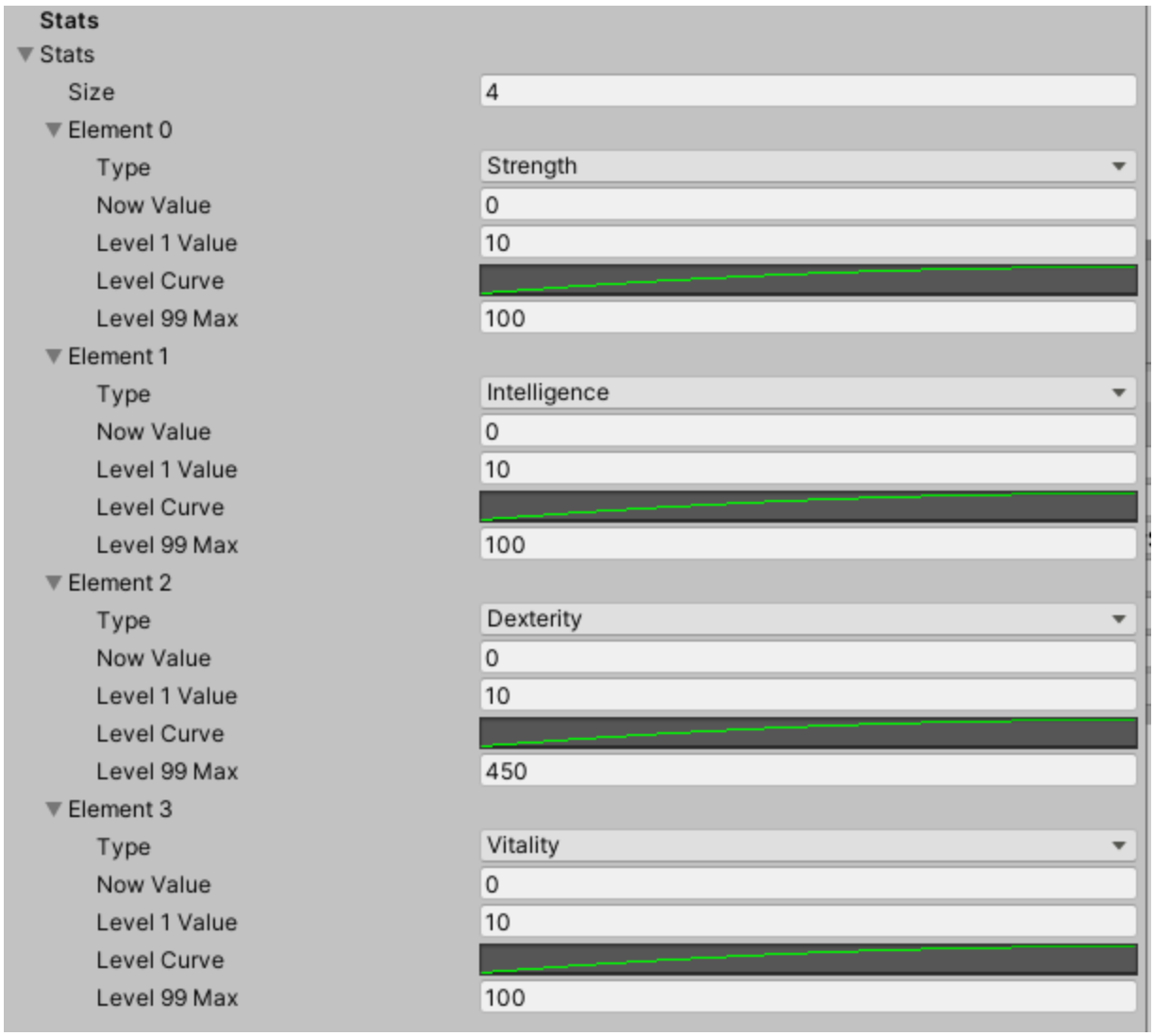
Resources refer to things that have a cost. **Health** and **Mana** are examples.

Resources increase on a leveling curve. In the example below, at level 1, the stats will start with '50' Health and '50' mana before any item or resource buffs are included.

- *Type* refers to the resource type. Refer to creating a new resource type to add custom types.
- *Max Value* is the maximum number for that resource.
- *Now Value* is the runtime value of that resource.
- *Level 1 Value* refers to the amount of resources the stat user has at *Level 1.*
- *Level Curve* defines the leveling curve for the resource.
- *Level 99 Max* refers to the maximum value that the stat user can gain from leveling.

Stats refer to things that do not have a cost. **Strength, Intelligence, Dexterity,** and **Vitality** are examples.

Stats increase on a leveling curve, similar to **resources** above. In the example below, at level 1, the stats will start with '10' of each stat listed before any item or stat buffs are included.



- *Type* refers to the stat type. Refer to creating a new stat type to add custom types.
- *Now Value* is the runtime value of that stat.
- *Level 1 Value* refers to the amount of stat the stat user has at *Level 1.*
- *Level Curve* defines the leveling curve for the stat.
- *Level 99 Max* refers to the maximum value that the stat user will have of the stat.

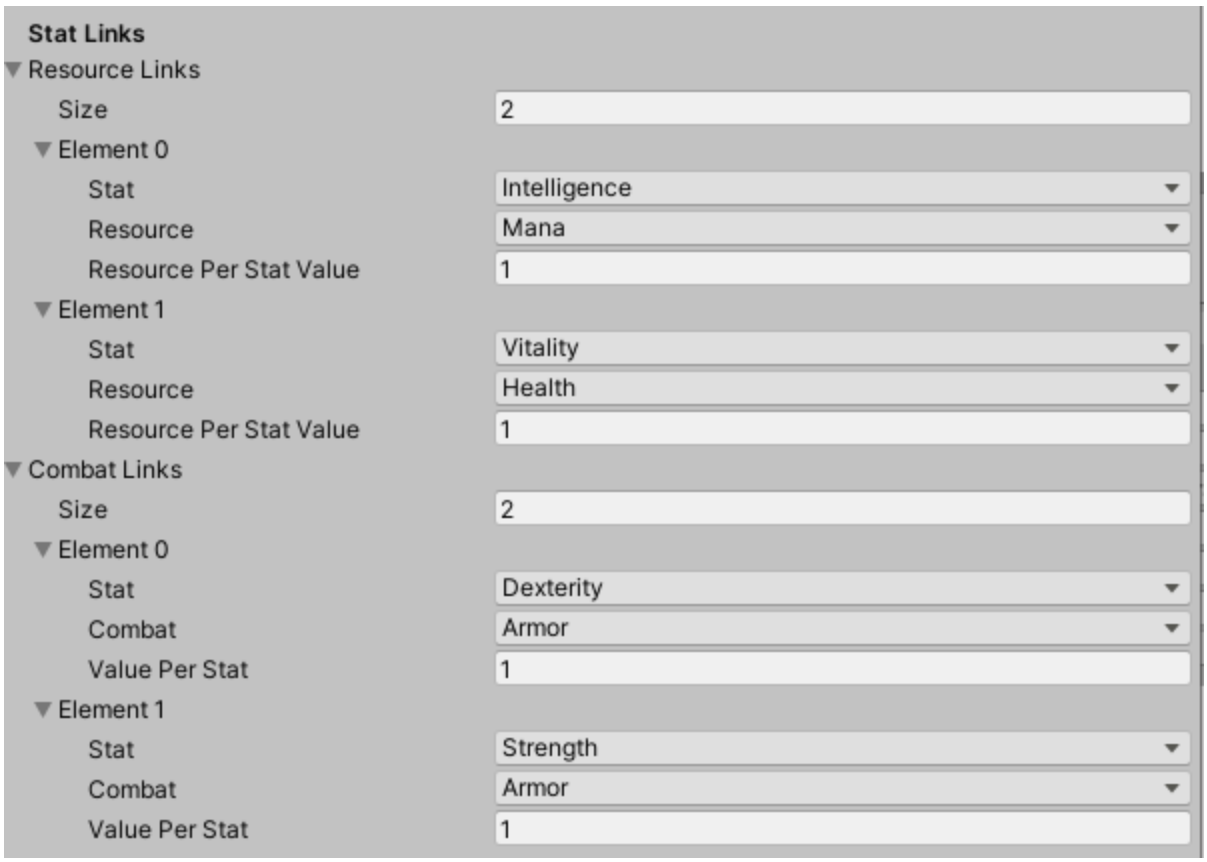
Stats can increase **Resources** or **Combat** values by using a **Stat Link.**

In the example below, for *Resource Links*

- for every 1 point of *Intelligence*, *Mana* is increased by 1.
- for every 1 point of *Vitality*, *Health* is increased by 1.

For *Combat Links*

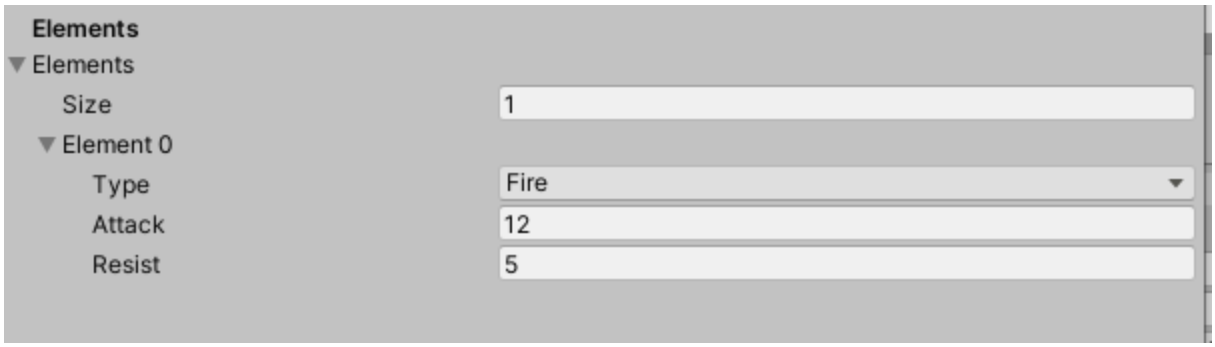
- for every 1 point of *Dexterity*, *Armor* is increased by 1.
- for every 1 point of *Strength*, *Armor* is increased by 1.



- *Stat* refers to the *Stat* that will increase a *Resource* amount.
- *Resource* refers to the *Resource* that will be increased.
- *Resource Per Stat Value* refers to the amount of *Resources* to be added/subtracting per *Stat* point.

Elements refer to elemental attack amounts and resist amounts.

The example below has *Fire* attack of 12 and a resist of 5. Attack and Resist are calculated based on subtraction. An attack of '12' against a stat user of '5' *Fire* resist will yield a damage result of '7'.



- *Type* refers to the *Element*. Refer to creating a new element type to create custom types.
- *Attack* refers to the attack amount of the element.
- *Resist* refers to the resist amount of the element.

Attributes

Attributes come in four flavors: Stats, Resources, Elements, and Other. These are located on the Actor Stats class.

See how to customize the different attributes to learn how to add more.

Aura Controller

The Aura Controller is a scriptable object that controls the use of auras. It will safely switch between auras.

<u>Method 1:</u>

Duplicate an existing actor stats found under

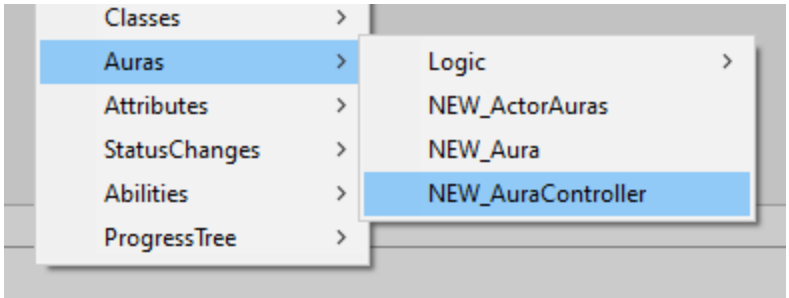
- *Assets/GWLPXL/ARPG/Data/Auras/Controllers*

<u>Method 2:</u>

Right Click in the *Project Window*

- -> Create -> GWLPXL -> ARPG -> Stats -> NEW_ActorStats

ARPG Attributes, Items, and Abilities



Aura

Auras can have a repeating effect (Pulse) and an area of effect (AOE). Auras can also be defined to only affect certain groups, so player aura will only help friendlies or you can create a player aura that hurts enemies. You can extend the Aura Group type.

<u>Method 1:</u>

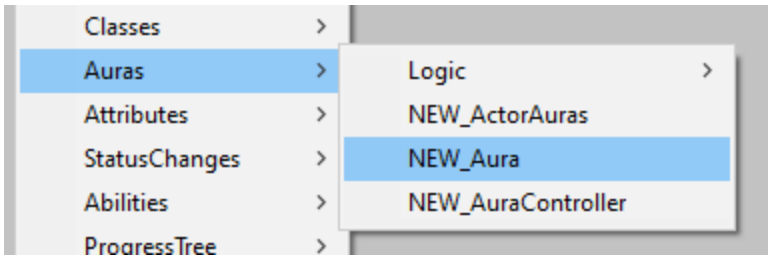
Duplicate an existing actor stats found under

- *Assets/GWLPXL/ARPG/Data/Auras/Types*

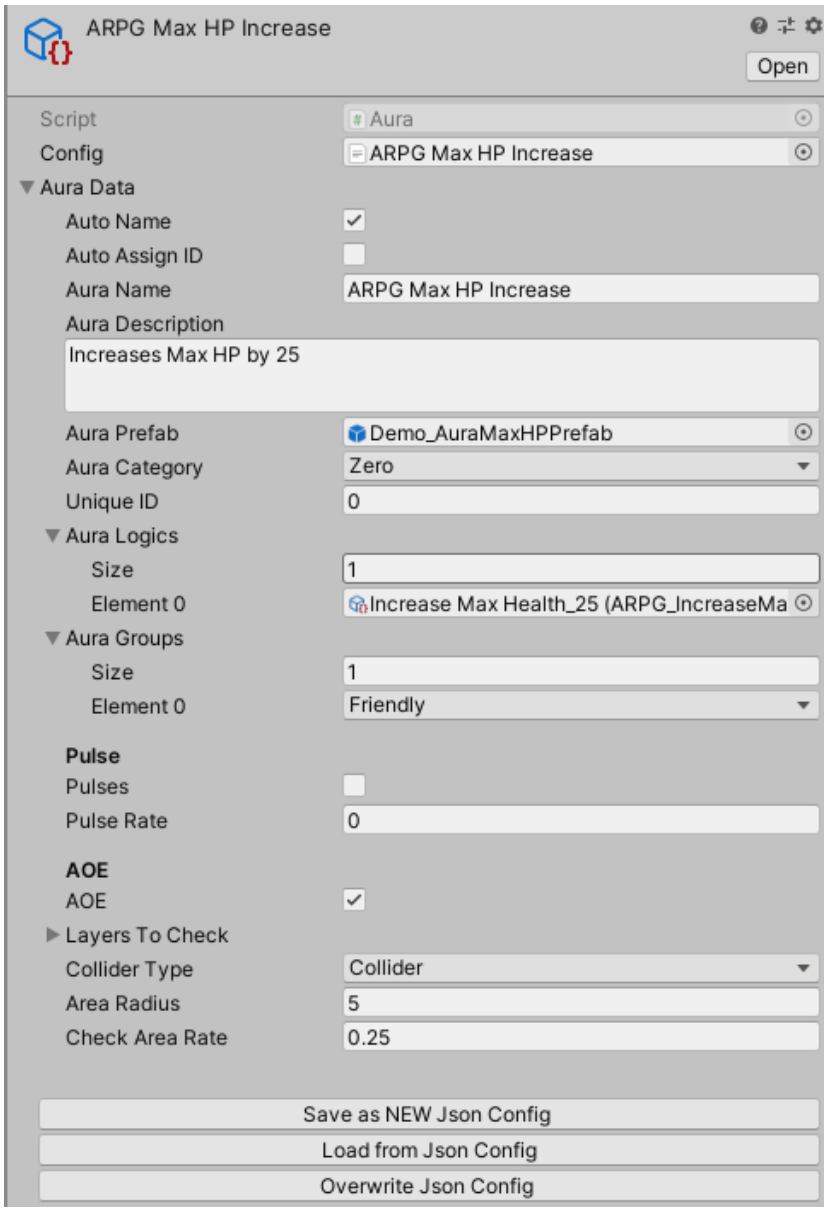
<u>Method 2:</u>

Right Click in the *Project Window*

- -> Create -> GWLPXL -> ARPG -> Stats -> NEW_ActorStats



Auras also contain interchangeable logics. The example below has the aura increasing max health by 25. Adding another logic will add another behavior of the aura.



Aura Logics initiate the gameplay behavior.

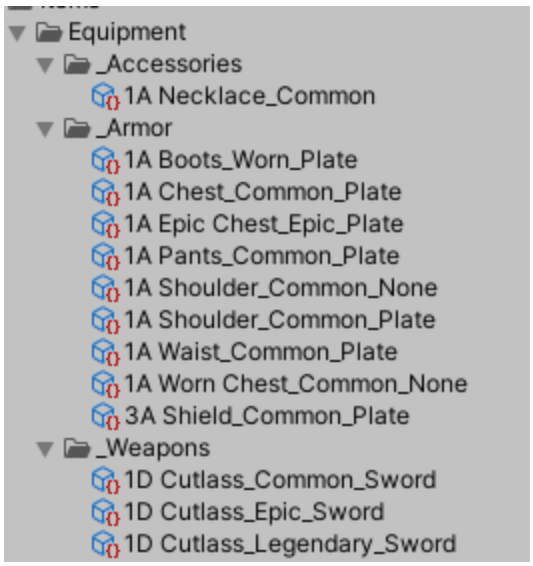
Equipment

Equipment are part of the *Item* system. They can the archetypal data for a piece of equipment, including its stats and traits.

<u>Method 1:</u>

Duplicate one of the Equipment Scriptable objects (Armor, MeleeWeapon, Accessory) already found in the project

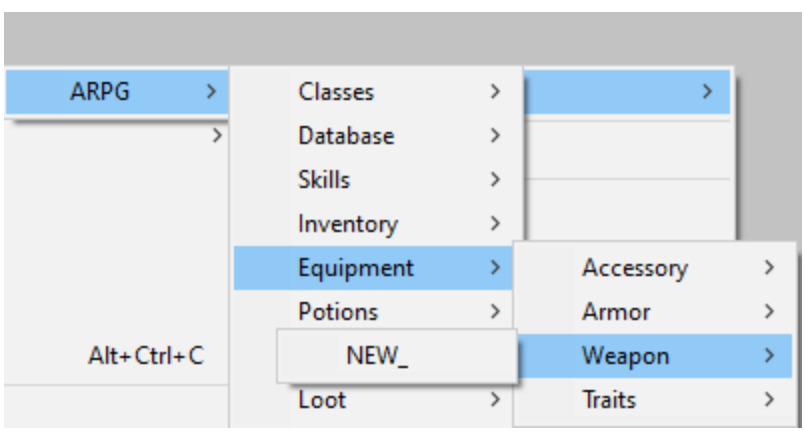
- *Assets/GWLPXL/ARPG/_Data/ItemStatsInventory/Items/Equipment*



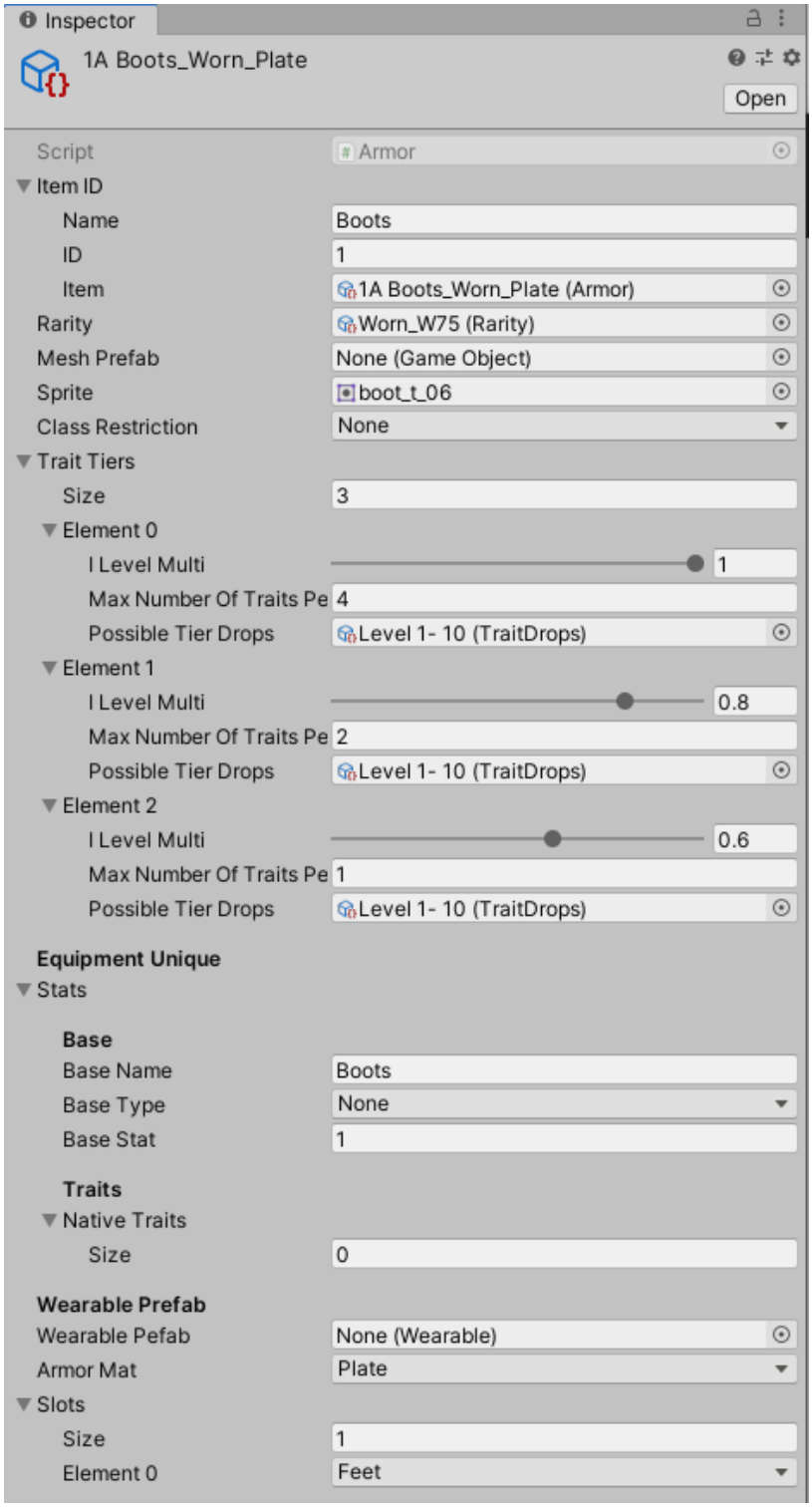
<u>Method 2:</u>

Right Click in the *Project Window*

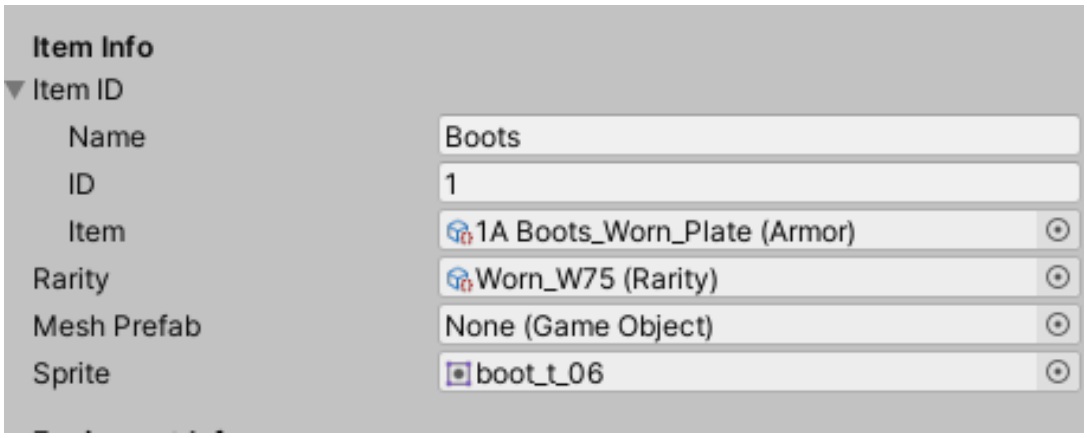
- -> Create -> GWLPXL -> ARPG -> Equipment -> [Piece you want] -> NEW_



Example of an equipment piece (keep scrolling for full description):



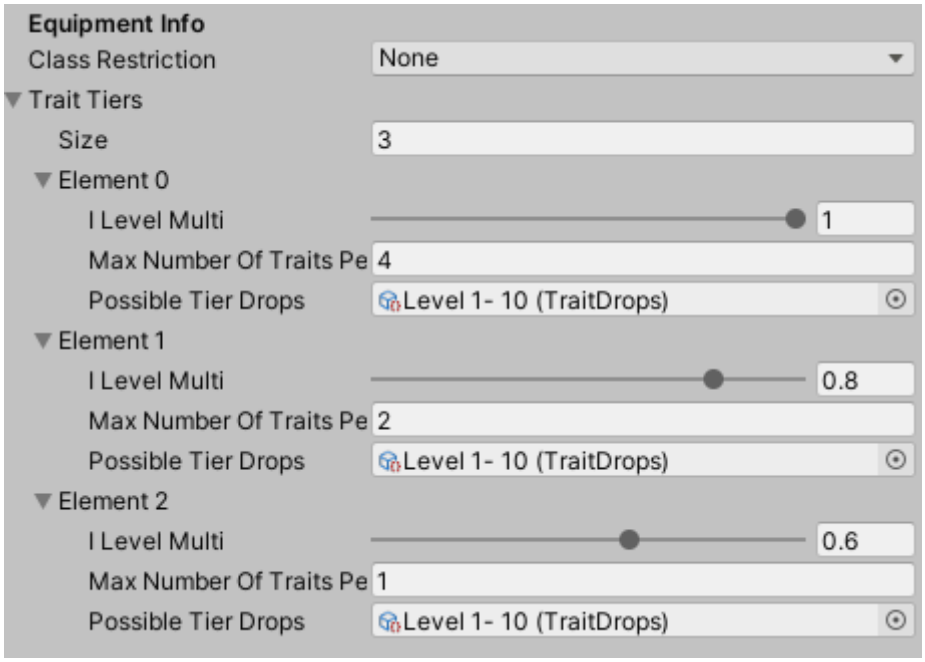
Item Info



- *Item ID* is for saving purposes, don't change unless you know what you're doing.
- *Rarity* determines its chance to drop (and also text color and possible # of traits to drop)
- *Mesh Prefab* is the mesh that'll drop as the loot.
- *Sprite* is the 2d representation of the item (such as in the inventory).

Equipment Info

This is the basic info for all equipment types (Armor, Weapon, Accessory). **ILevel** stands for **I**tem **L**evel**.**



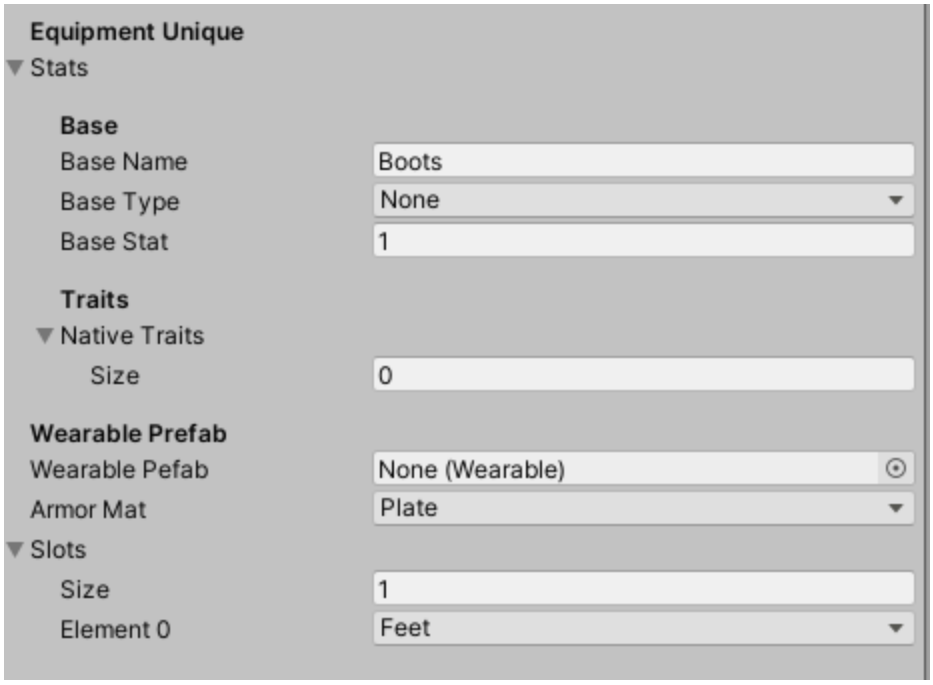
- *Class Restriction* allows you to restrict to certain classes.
- *Trait Tiers* allow you to create multiple tiers of possible random drops. In this example, there are three tiers.
- In this example,
    - *Element 0* receives full value (100%) of the iLevel when the item is dropped. It can only drop up to 4 of these (rarity also constrains drops, so be mindful) and the traits that drop come from the *Possible Tier Drops*.
    - *Element 1* and *Element 2* are repeats, except they receive 80% and 60% of the iLevel, respectively.

<u>Important notes for Equipment Info:</u>

- Why have trait tiers? This allows for easier balancing purposes. If some traits are more valuable than others (such as element attack), then you can restrict it from dropping too often or at too high of a value.
- This *I Level Multi* is applied **before** the individual traits multi is applied. For instance, if the iLevel is 20 and our first *Element 0* has a Multi of 0.6, then the individual traits will receive an iLevel of 12 (20 * 0.6). This number **may** be further reduced by their individual multis.
- Of course you can ignore this level of detail and only include 1 Trait tier that receives full value and has 999 traits (some number equal to or larger than your project's rarity drop range). These possible traits will not exceed the number set on the rarity.

Equipment Unique

Armor, Accessory, and Weapon look similar, but these are unique values for each equipment type.

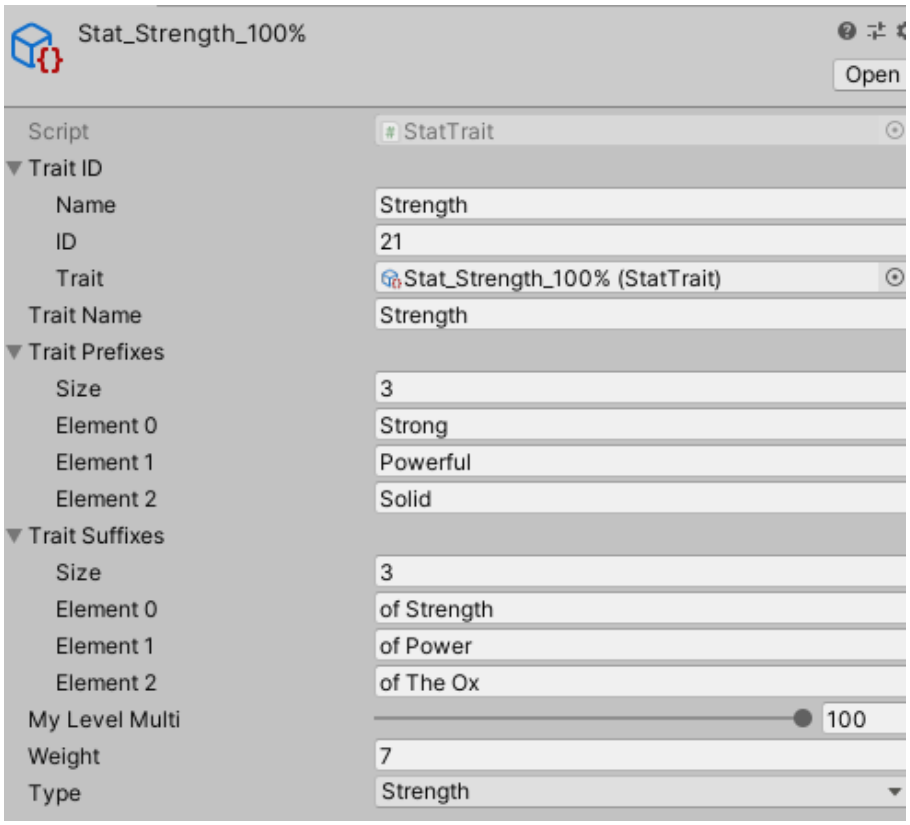


- *Stats* refer to the equipment's base values (**before** iLevel or traits are applied).
    - *Base Name* is the name the player sees.
    - *Base Type* defines what type of value the Base Stat is (armor or damage).
    - *Base Stat* is the starting value before Ilevel or traits are applied.
    - *Native Traits* allow you to attach a trait that is always on the equipment.
- *Wearable Prefab* is the prefab that has an attached "Wearable" script and allows you to plug the equipment into the "Clothing" system, so you can visually represent the equipment on the player.
- *Armor Mat* defines the armor material (applies only to armor. Weapons have weapon types such as sword, axe).
- *Slots* defines where the player wears this. You can have a piece of equipment occupy multiple slots.

The other types of equipment will have slight variations, but they are mostly obvious (e.g. Instead of *Armor Mat* the melee weapons have a choice of type of weapon such as Sword).

Equipment Trait

Equipment Traits is a **Scriptable Object** that allows the system to attach traits to *equipment* based on an item's level (ILevel).



- *Trait ID* is for saving purposes, don't change those unless you know what you're doing.
- *Trait Name* is the name shown to the player.
- *Trait Prefixes* are the randomly assigned prefixes to the dropped loot. The first native trait on the dropped equipment writes the prefix. For instance, if this was assigned to the **Native Traits** section of a dropped piece of equipment, then the equipment would start with one of the **Trait Prefixes** in its randomly generated name.
- *Trait Suffixes* are the randomly assigned suffixes to the dropped loot. The first random trait on the drop writes the suffix.
- *My Level Multi* is the **percentage** that the trait receives per iLevel. In this case, it's 20% per 1 iLevel (so an item that drops with an iLevel of 20 gets Ice Attack +4%). **Be mindful**, if the ILevel isn't high enough, then it could drop a trait with 0 since the system uses integers.
- *Weight* determines the chance the trait will be applied.
- *Type* is the element type of the attack.

The Equipment Traits all follow a similar pattern.

Inventory

Actor Inventory is a **Scriptable Object** that provides that track's an actor's held items, including items that are currently equipped.

Method 1:

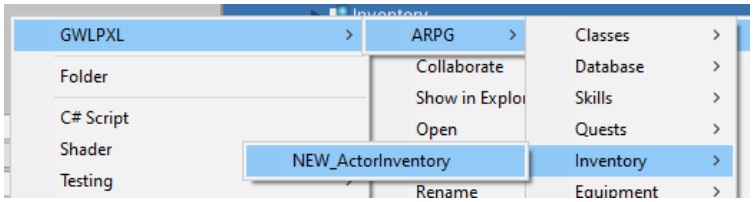
Duplicate an existing inventory found under

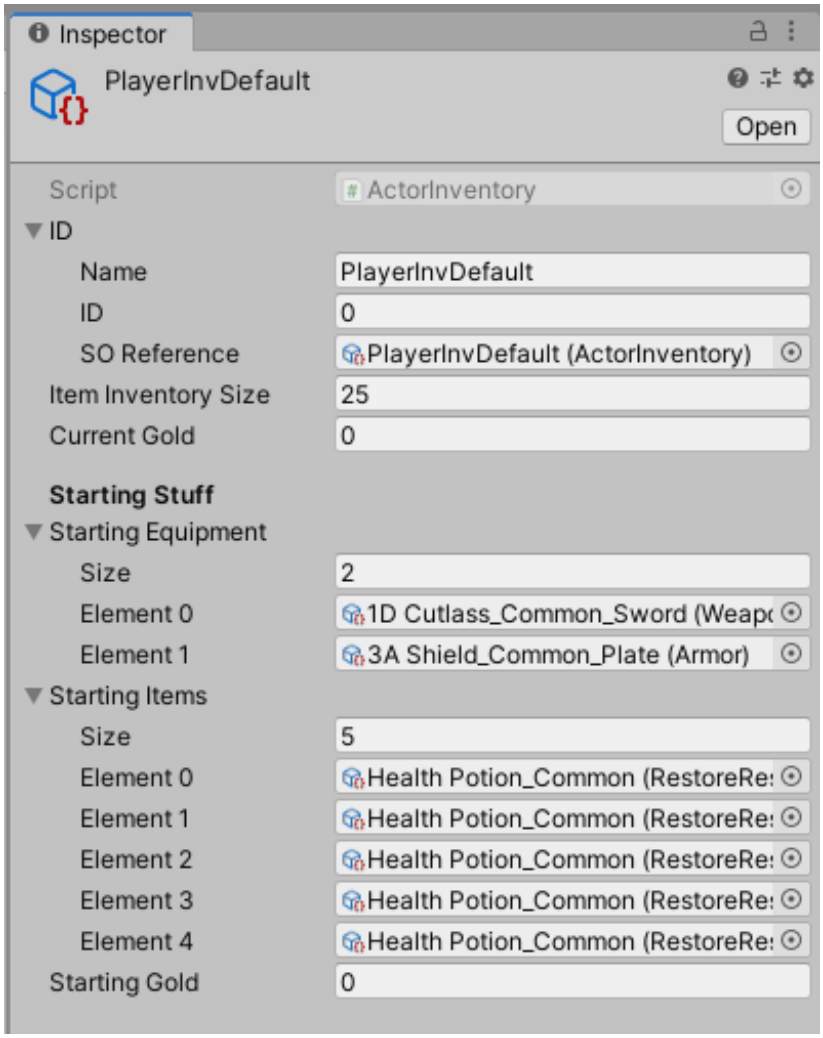
- *Assets/GWLPXL/ARPG/_Data/ItemStatsInventory/Inventory*

Method 2:

Right Click in the *Project Window*

- -> Create -> GWLPXL -> ARPG -> Inventory -> NEW_ActorInventory





- *ID* is for saving purposes.
- *Item Inventory Size* limits the number of unique items that can be held at a time.
- *Current Gold* is gold at runtime.
- *Starting Equipment* allows you to give actor equipment at the beginning that will be equipped.
- *Starting Items* does the same for items (that aren't equipped).
- *Starting Gold* is the gold at the beginning of the experience.

Loot Drops

Loot Drops is a **Scriptable Object** that holds possible drops (Loot Table). The loot drop percentages are determined by an item's *weight*, refer to creating a new type of rarity to customize and learn more about weights.

Method 1:

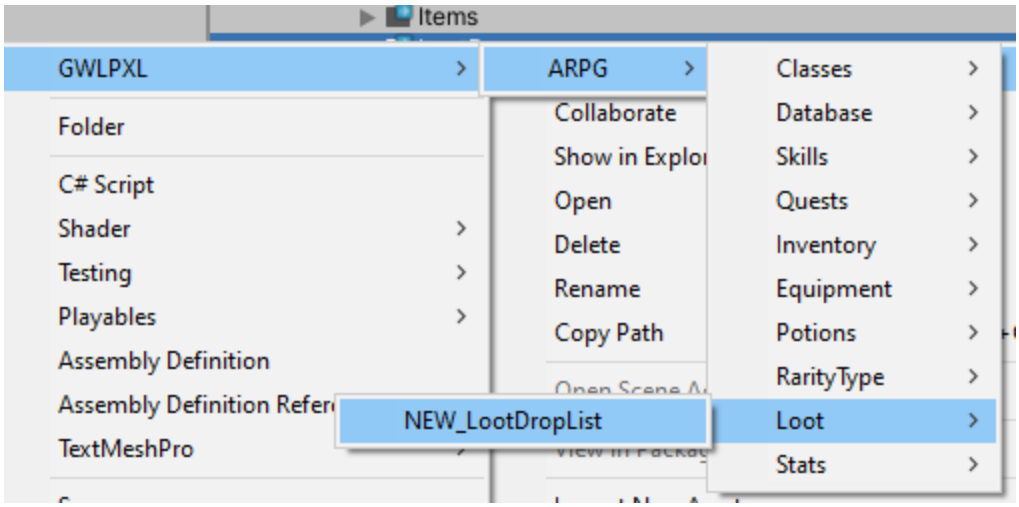
Duplicate an existing Loot Drop List found under

- *Assets/GWLPXL/ARPG/Data/ItemStatsInventory/LootDrops*

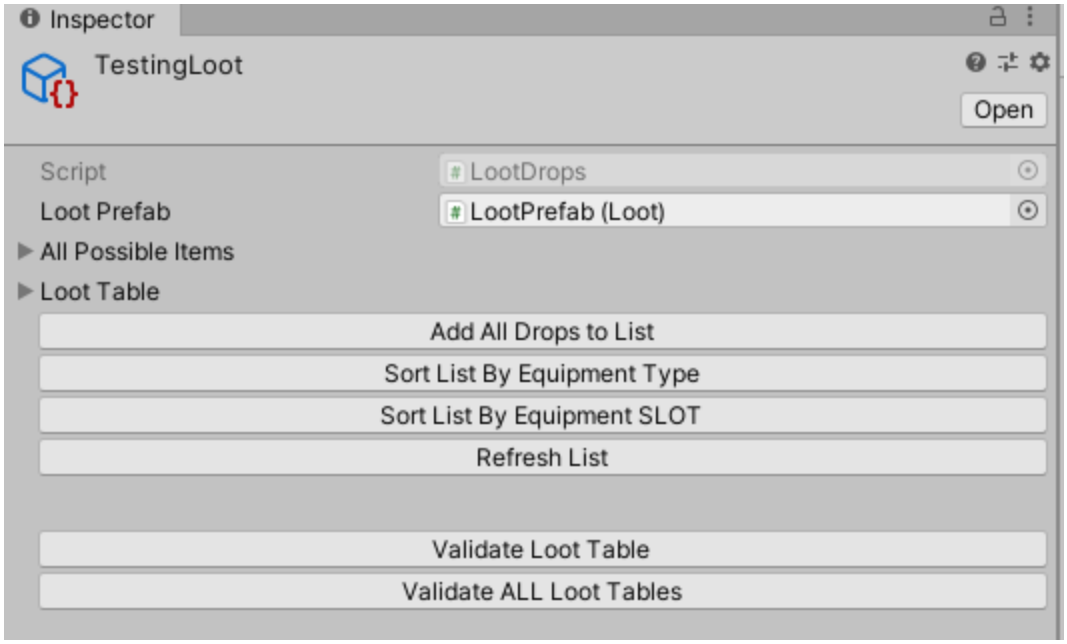
Method 2:

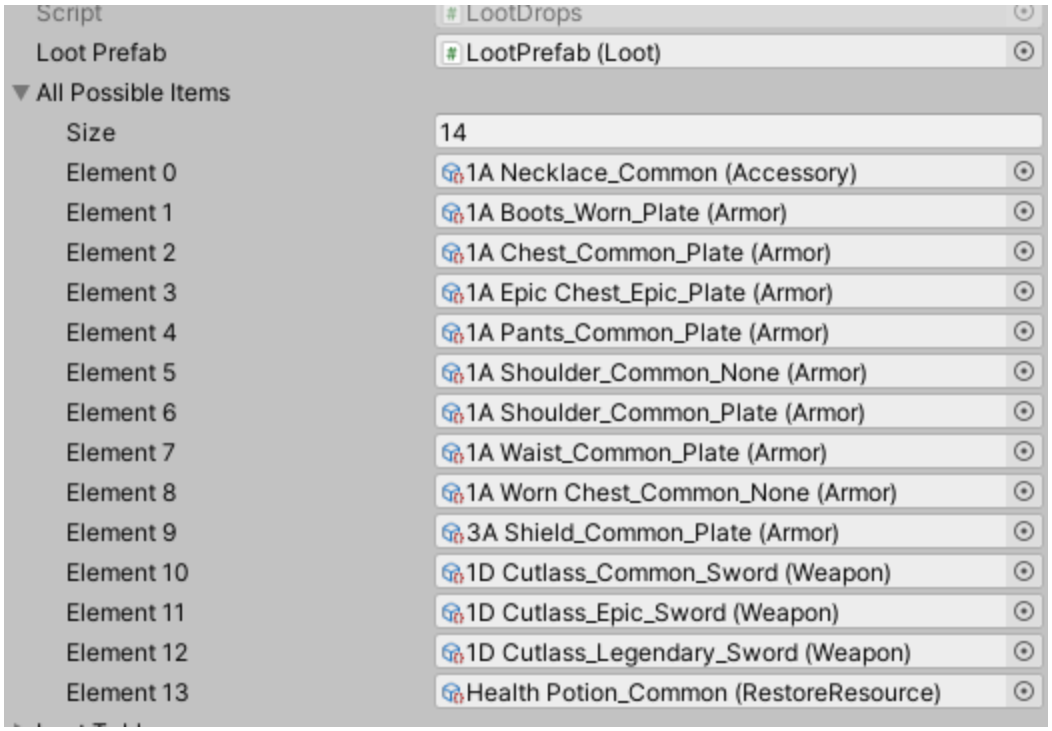
Right Click in the *Project Window*

- -> Create -> GWLPXL -> ARPG -> Loot -> LootDropList

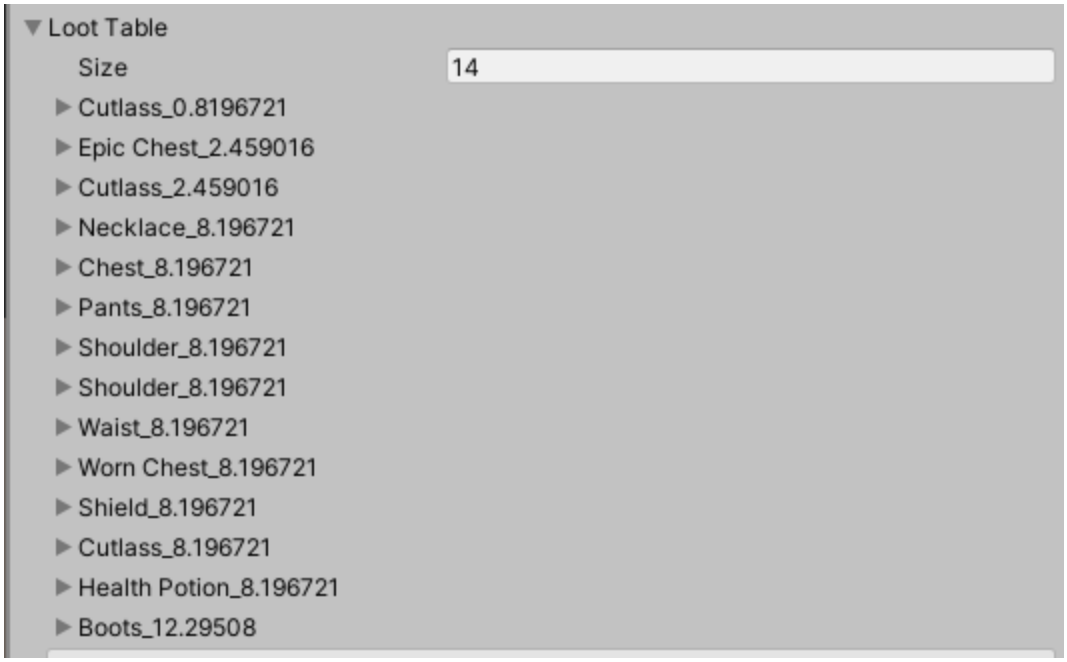


Scroll down for further explanation.

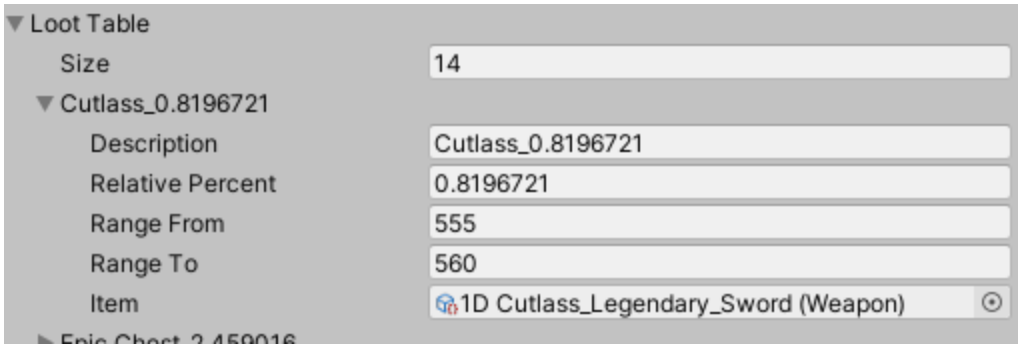




- *Loot Prefab* refers to the object that is created on the dungeon floor once loot is dropped. It is already configured and comes with a script that speaks to the dungeon canvas and updates its values based on the *equipment* that is dropped. Don't change unless you know what you're doing.
- *All Possible Items* is a list that contains any item that has the potential to drop.

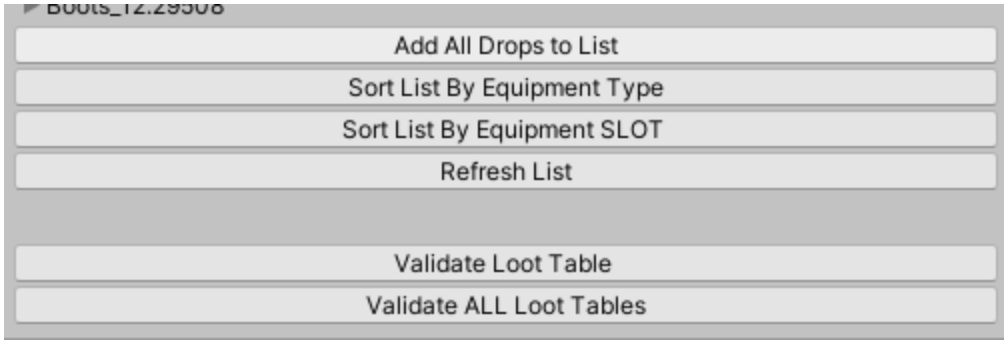


- *Loot Table* displays the drop percentages for each item in the *All Possible Items* list. It is auto named with the base name of the item plus the percent chance to drop (e.g. the first 'Cutlass' has roughly ~ 0.82% chance to drop while the 'Epic Chest' has roughly a ~2.5% chance to drop).



Expanding the items in the *Loot Table* shows the item's *description*, chance to drop, weighted range and scriptable object reference. Don't change the *Range From* and *Range To*, these are generated at runtime or when you hit the button "Validate Loot Table". These values are only exposed for designers.

The *Relative Percent* derived from the *Item Rarity* that is assigned to each drop. To learn more about creating custom raritys and weights, refer to creating a new type of rarity.

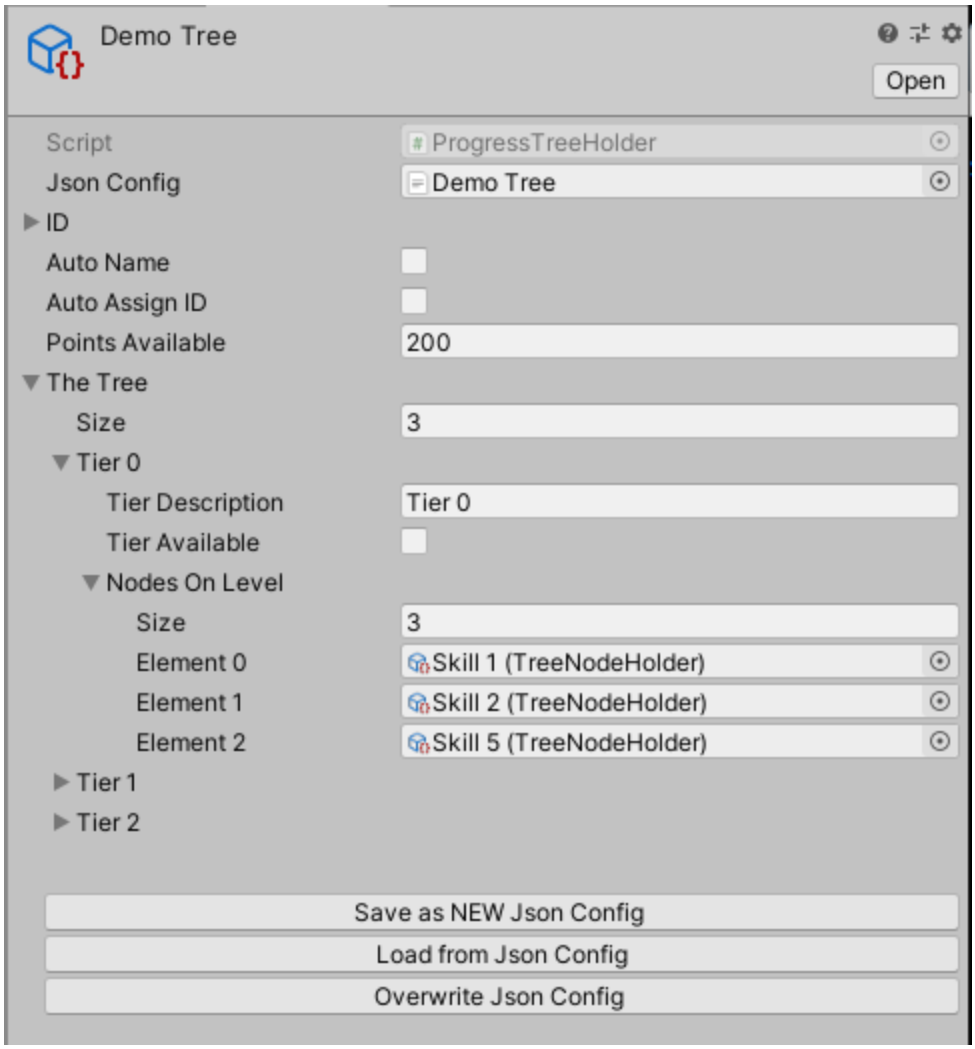


Clicking “Validate Loot table” generates the *Loot Table*.

Progress Tree Holder (Ability Tree in Demo)

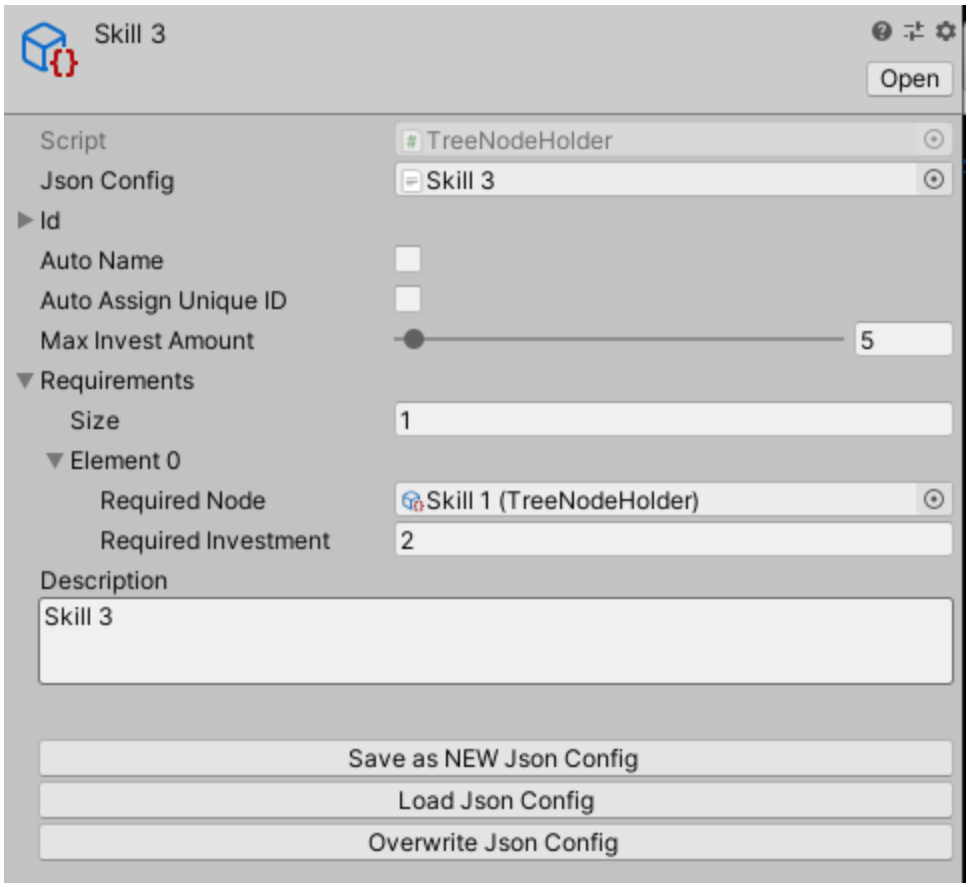
The progress tree is a generic tree that allows you to invest and divest points from certain nodes in order to progress down the branches. The example tree below and in the demo, show how to use it to unlock abilities. The system will also be leveraged for branching quests.

The tree consists of tiers, and each tier has nodes on it.

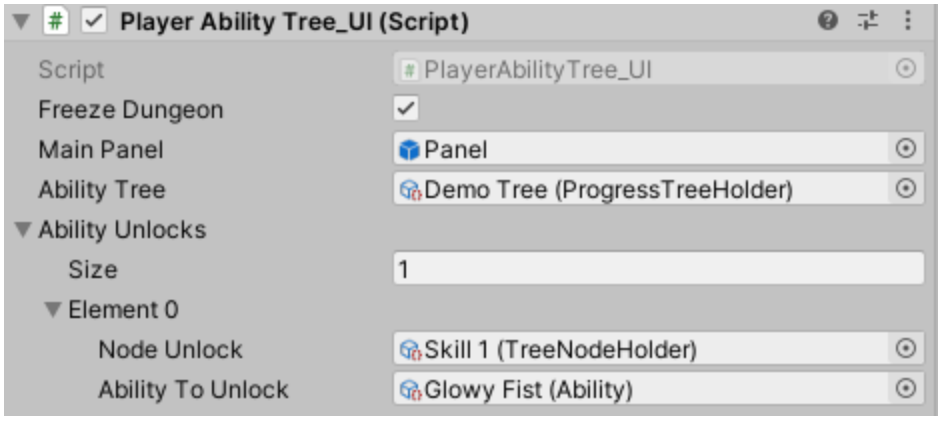


Progress Tree Node

Each node can require another node or nodes to unlock and a certain amount of points in the node.

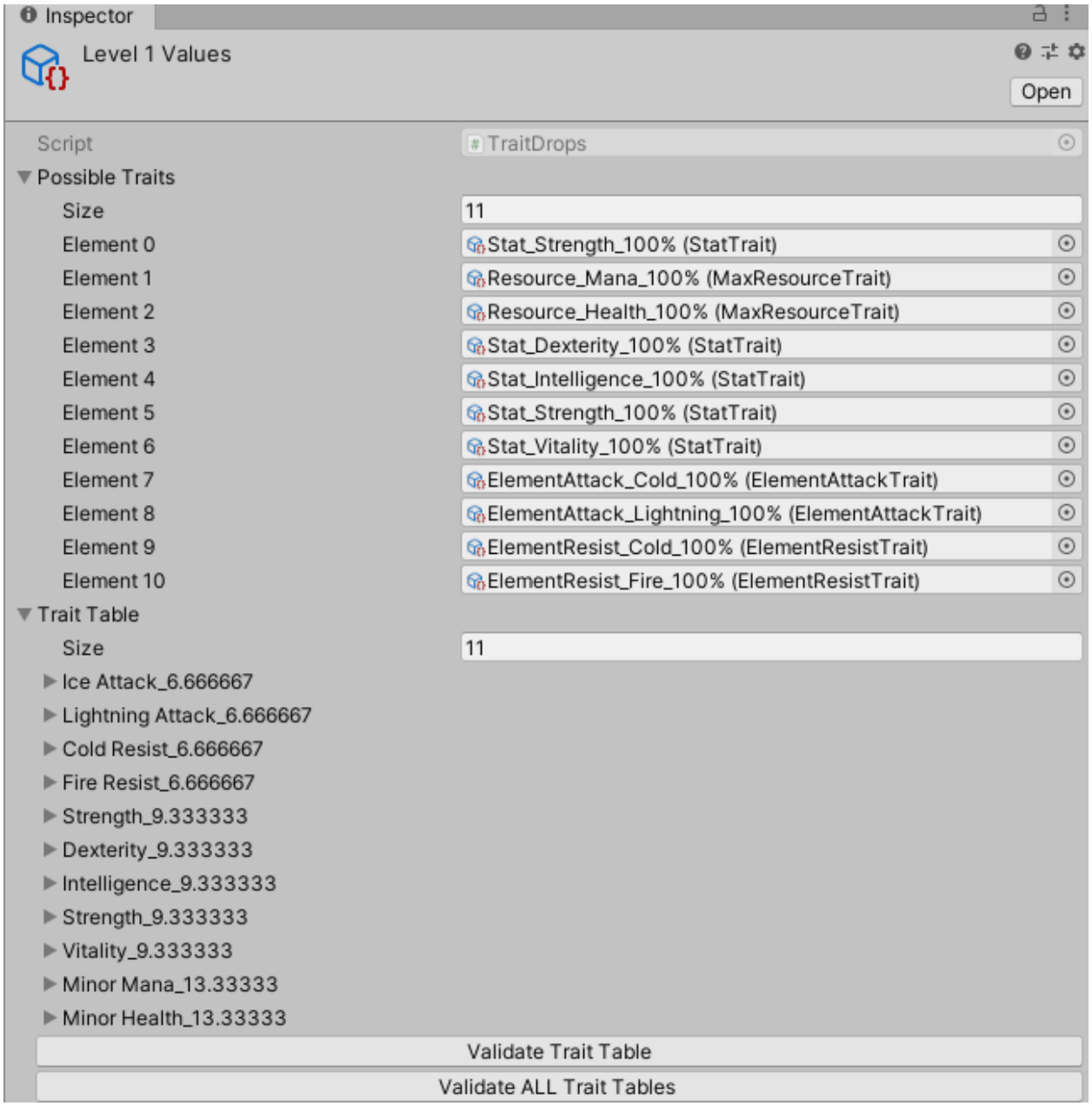


Example class that links the node lock status to learning an ability.



Trait Drop

Similar to the LootDrops, the Trait Drops Scriptable Object holds all the possible traits that could be added to a weapon at runtime. The 'Trait Table' determines the chance a particular Trait will be attached to the Equipment.



Click 'Validate Trait Table' to create and visualize the drop chances. The drop chances are determined by the 'weight' on the Traits, relative to all the other trait weights in the 'Possible Traits'.

Statics

Static classes that are used as helper classes for various utilities.

- **JsonConfig** - provides the Json saving and loading.
- **ItemHandler** - convenience functions that are common for item handling (Drop, Use, etc.)
- **PlayerDescription** - provides string descriptions to the player (e.g. character info)
- **TypeKeys** - hardcoded string keys for abstract classes for use in editor classes
- **Formulas** - common formulas, such as Player and Enemy damage
- **FindInterfaces** - convenience function that finds all interfaces of type in an active scene
- **EquipmentHandler** - convenience functions that are common for equipment

Status Changes

The system in charge of Damage over Time (DOT), Buffs, and Debuffs.

The ability system leverages it at the moment. Good contender for refactor.

Extending the System

Types can be found in

- namespace GWLPXL.ARPGCore.Types.com
- Assets/GWLPXL/ARPG/_Scripts/Types

Abilities

Ability Logic and AbilityCategory can be customized.

Extend Ability Logic.

```
using UnityEngine;

namespace GWLPXL.ARPGCore.Abilities.com
{

    Unity Script | 5 references
    public abstract class AbilityLogic : ScriptableObject
    {
        /// <summary>
        /// Called after the delay period, if any.
        /// </summary>
        /// <param name="skillUser"></param>
        /// <param name="theSkill"></param>
        4 references
        public abstract void StartCastLogic(IAbilityUser skillUser, Ability theSkill);


        /// <summary>
        /// Called after the cooldown period, if any.
        /// </summary>
        /// <param name="skillUser"></param>
        /// <param name="theSkill"></param>
        4 references
        public abstract void EndCastLogic(IAbilityUser skillUser, Ability theSkill);


    }
}
```

Inherit **AbilityLogic** and write your specific behavior under StartCast and EndCast.

See the examples used in the demo under

- Assets/GWLPXL/ARPG/_Scripts/Abilities/Scriptables/Logics

Creating Custom Ability Category

1. Open up the script “AbilityCategory”
2. Add a new enum value
3.
```
namespace GWLPXL.ARPGCore.Types.com
{
    2 references
    public enum AbilityCategory
    {
        Zero = 0,
        One = 1,
        Two = 2,
        Three = 3,
        Four = 4
    }
}
```
4. The example below adds the “Five” category.
5.
```
namespace GWLPXL.ARPGCore.Types.com
{
    2 references
    public enum AbilityCategory
    {
        Zero = 0,
        One = 1,
        Two = 2,
        Three = 3,
        Four = 4,
        Five = 5
    }
}
```
6. Save the Script.
7. You can now use your custom Ability Category.

Ability Buffs

Creating a Custom Ability Buff

Example Ability Buffs live in

- Assets/GWLPXL/ARPG/_Scripts/StatusChanges/Scriptables/AbilityBuffs

Ability Buffs are Status Changes that modify an Ability. These are typically instant effects or temporary effects that last during the duration of the Ability. They differ from Weapon Buffs in that they do not live on the weapon but instead use the caster as its scene object.

Ability Buffs only exist as Data on a Scriptable Object. Let's take a look at an example. The following example provides an instant Element Buff +Attack during the duration of the Ability.

```
[CreateAssetMenu(menuName = "GWLPXL/ARPG/StatusChanges/Abilities/NEW_WeaponElementBuff_Flat")]
//flat buff amount
Unity Script | 0 references
public class InstantElementBuff_Flat : AbilityStatusChange
{
    public ElementType Element;
    public int BuffAmount;
    [System.NonSerialized]
    Dictionary<ActorAttributes, int> buffed = new Dictionary<ActorAttributes, int>();//stats and buff amount

    10 references
    public override void ApplyStatus(IAbilityUser attacker)
    {
        IAttributeUser statUser = attacker.GetParentTransform().gameObject.GetComponent<IAttributeUser>();
        Apply(statUser);
    }
    10 references
    public override void RemoveStatus(IAbilityUser attacker)
    {
        IAttributeUser statUser = attacker.GetParentTransform().gameObject.GetComponent<IAttributeUser>();
        Remove(statUser);
    }

    1 reference
    private void Apply(IAttributeUser statuser)
    {
        ActorAttributes stats = statuser.GetRuntimeAttributes();
        buffed.TryGetValue(stats, out int value);
        if (value == 0)
        {
            stats.ModifyElementAttackNowValue(Element, BuffAmount);
            buffed[stats] = BuffAmount;

        }
        ARPGDebugger.DebugMessage("buffed " + Element + " " + BuffAmount + stats.ActorName, stats);
    }

    1 reference
    private void Remove(IAttributeUser statuser)
    {
        ActorAttributes stats = statuser.GetRuntimeAttributes();
        buffed.TryGetValue(stats, out int value);
        if (value != 0)
        {
            stats.ModifyElementAttackNowValue(Element, -value);
            buffed[stats] = 0;
        }
        ARPGDebugger.DebugMessage("removed " + Element + " " + -BuffAmount + stats.ActorName, stats);
    }
```

Ability Buffs derived from **AbilityStatusChange**. The ApplyStatus() and RemoveStatus() methods contain the unique behavior of the buff. So, you need to write that unique behavior and also the variables. The ApplyStatus() is called when an ability is first activated, and the RemoveStatus is called when the Ability is complete (typically after the cooldown is over).

Actor Class

Actor Classes are optional, but allow you to define limitations for what a character can equip. ClassType is customizable.

Creating a Custom Class Type

8. Open up the script “ClassType”
9. Add a new enum value
10.
```
8 references
public enum ClassType
{
    None = 0,
    Warrior = 1,
    Mage = 2,
}
```
11. The example below adds the “Thief” class.
12.
```
8 references
public enum ClassType
{
    None = 0,
    Warrior = 1,
    Mage = 2,
    Thief = 3
}
```
13. Save the Script.
14. You can now use your custom class type.

Attributes

Attributes are found on the Actor Stats class and form the base of the stats system.

Creating a Custom Stat Type

Stats are attributes that have flat values, a ‘now value’.

1. Open up the script “StatType”
2. Add a new enum value
3.
```
29 references
public enum StatType
{
    Strength = 0,
    Intelligence = 1,
    Dexterity = 2,
    Vitality = 3,
    None = 20
}
```
4. The example below adds “Charisma”.
5.
```
29 references
public enum StatType
{
    Strength = 0,
    Intelligence = 1,
    Dexterity = 2,
    Vitality = 3,
    Charisma = 4,
    None = 20
}
```
6. Save the Script.
7. You can now use the custom StatType.

Creating a Custom Resource Type

Resources are attributes that can be ‘consumed’ and ‘replenished’. As such, they have a ‘cap value’ and a ‘now value’, e.g. 72 / 100.

1. Open the script “ResourceType”
2. Add a new value to the enum.
3.
```
54 references
public enum ResourceType
{
    Health = 0,
    Mana = 1,
    None = 20
}
```
4. The example below adds “Stamina”.
5.
```
79 references
public enum ResourceType
{
    Health = 0,
    Mana = 1,
    Stamina = 2,
    None = 20
}
```
6. Save the Script.
7. You can now use your custom ResourceType.

Creating a Custom Element Type

Elements are attributes that have flat values, a ‘now value’.

1. Open the Script “ElementType”
2.
```
39 references
public enum ElementType
{
    None = 0,
    Fire = 1,
    Cold = 2,
    Lightning = 3
}
```
3. Add a new value to the enum. This example below adds Darkness:
4.
```
39 references
public enum ElementType
{
    None = 0,
    Fire = 1,
    Cold = 2,
    Lightning = 3,
    Darkness = 4
}
```
5. Save the script.
6. You can now choose your new ElementType.

Creating a Custom Other Type

‘Other’ attributes are ones that aren’t a flat value (Stat) or a consumed/replenished value.

1. Open the Script “OtherAttributeType”.
2.
```
5 references
public enum OtherAttributeType
{
    None = 0,
    CriticalHitChance = 1,
    CriticalHitDamage = 2
}
```
3. Add a new value to the enum. The example below adds “MovementSpeed”.
4.
```
5 references
public enum OtherAttributeType
{
    None = 0,
    CriticalHitChance = 1,
    CriticalHitDamage = 2,
    MovementSpeed = 3
}
```
5. Save the Script.
6. You can now use this new ‘OtherAttributeType’.

Auras

Extend Aura Logic

```
using UnityEngine;


namespace GWLPXL.ARPGCore.Auras.com
{
    /// <summary>
    /// base class for any new aura logic.
    /// Inherit from this abstract class to create your own logic.
    /// </summary>
    Unity Script | 4 references
    public abstract class AuraLogic : ScriptableObject
    {
        /// <summary>
        /// When Aura is applied.
        /// </summary>
        /// <param name="onUser"></param>
        /// <returns></returns>
        4 references
        public abstract bool DoApplyLogic(ITakeAura onUser);
        /// <summary>
        /// When Aura is removed.
        /// </summary>
        /// <param name="fromUser"></param>
        /// <returns></returns>
        4 references
        public abstract bool DoRemoveLogic(ITakeAura fromUser);

    }
}
```

Inherit from **AuraLogic.** Write the behavior in DoApplyLogic and DoRemoveLogic.

See the examples in

- Assets/GWLPXL/ARPG/_Scripts/Auras/Scriptables/AuraLogics

Creating Custom Aura Category

1. Open the Script “AuraCategory”.
2. Add an enum value.
3.
```
namespace GWLPXL.ARPGCore.Types.com
{
    2 references
    public enum AuraCategory
    {
        Zero = 0,
        One = 1,
        Two = 2,
        Three = 3,
        Four = 4,
    }
}
```
4. The example below adds “Five”.
5.
```
namespace GWLPXL.ARPGCore.Types.com
{
    2 references
    public enum AuraCategory
    {
        Zero = 0,
        One = 1,
        Two = 2,
        Three = 3,
        Four = 4,
        Five = 5
    }
}
```

   Save the Script.
6. You can now use your new AuraCategory.

Equipment

Creating a Custom Equipment Slot Type

7. Open the Script “EquipmentSlotsType”.
8. Add an enum value.
9.
```
35 references
public enum EquipmentSlotsType
{
    None = 0,
    Head = 5,
    Chest = 10,
    Waist = 15,
    Legs = 20,
    Feet = 25,
    LeftShoulder = 26,
    RightShoulder = 27,
    LeftHand = 30,
    LeftBracer = 31,
    RightBracer = 32,
    LeftGlove = 33,
    RightGlove = 34,
    RightHand = 35,
    Neck = 40,
    LeftRing = 45,
    RightRing = 50
}
```
10. The example below adds “Back”.
11.
```
35 references
public enum EquipmentSlotsType
{
    None = 0,
    Back = 1,
    Head = 5,
    Chest = 10,
    Waist = 15,
    Legs = 20,
    Feet = 25,
    LeftShoulder = 26,
    RightShoulder = 27,
    LeftHand = 30,
    LeftBracer = 31,
    RightBracer = 32,
    LeftGlove = 33,
    RightGlove = 34,
    RightHand = 35,
    Neck = 40,
    LeftRing = 45,
    RightRing = 50
}
```
Save the Script.
12. You can now use your custom slot.

Creating a Custom Equipment Type

The system already comes with ‘Armor’, ‘Weapon’, and ‘Accessory’. Adding anything beyond these categories will require you to inherit from “Equipment” and write the behavior yourself.

1. Open the Script “EquipmentType”.
2.
```
30 references
public enum EquipmentType
{
    None = 0,
    Armor = 5,
    Weapon = 10,
    Accessory = 15
}
```
3. Add a new enum value.
4. Create a new class.
5. Derive from “Equipment”.
6. Implement the abstract class, and complete the values.
7. You can now use your custom “EquipmentType”.

Creating a Custom Accessory Type

Accessory Types can be used to define accessories and who can wear them (which is stored in the Actor's Class).

1. Open the Script "AccessoryType".
2.
```
2 references
public enum AccessoryType
{
    None = 0,
    Necklace = 1,
    Ring = 2
}
```
3. Add a new enum value, the example below adds "Trinket".
4.
```
2 references
public enum AccessoryType
{
    None = 0,
    Necklace = 1,
    Ring = 2,
    Trinket = 3
}
```
5. Save the script.
6. You can now use your custom "AccessoryType".

Creating a Custom Armor Marial Type

Armor Materials Types can be used to define Armors and who can wear them (which is stored in the Actor's Class).

1. Open up the script "ArmorMaterial"
2. Add a new enum value.
3.
```
5 references
public enum ArmorMaterial
{
    None = 0,
    Cloth = 1,
    Leather = 2,
    Mail = 3,
    Plate = 4
}
```
4. This example adds "Diamond":
5.
```
5 references
public enum ArmorMaterial
{
    None = 0,
    Cloth = 1,
    Leather = 2,
    Mail = 3,
    Plate = 4,
    Diamond = 5
}
```

Creating a Custom Weapon Type

Weapon Types can be used to define weapons and who can wear them (which is stored in the Actor's Class).

1. Open up the script "WeaponType"
2. Add a new enum value.
3.
```
5 references
public enum WeaponType
{
    None = 0,
    Sword = 1,
    Axe = 2,
    Staff = 3
}
```
4. This example adds "Mace":
5.
```
5 references
public enum WeaponType
{
    None = 0,
    Sword = 1,
    Axe = 2,
    Staff = 3,
    Mace = 4
}
```

Items

Creating a Custom Item Type

'Items' is the base class that 'Potions' and 'Equipment' derive from. To create a new item type

1. Open up the script "ItemType"
2. Add a new enum value.
3.
```
13 references
public enum ItemType
{
    Equipment = 0,
    Potions = 1
}
```
4. The example adds "Socketable", such as gems for equipment ala Diablo.
5.
```
13 references
public enum ItemType
{
    Equipment = 0,
    Potions = 1,
    Socketable = 2
}
```
6. Create a new class "Socketables".
7. Derive that class from "Items".
8. Implement the abstract class.
9. Code the functionality for the new ItemType.

<u>You will need to write custom behavior for custom item types that aren't already included.</u> A beginning example is shown below.

1. make sure to use the namespace
    a. **GWLPXL.ARPCore.Items.com;**
2. Inherit from “Items”
3. Write the custom class.

```
0 references
public class Socketable : Items
{
    //write custom socketable behaviors here, e.g. "PlaceInSocket"
    //

    //this comes from the base items class
    15 references
    public override string GetItemName()
    {
        throw new System.NotImplementedException();
    }

    //this comes from the base items class
    5 references
    public override ItemType GetItemType()
    {
        return ItemType.Socketable;
    }
}
```

7.

Creating a Custom Potion Type

1. Open up the Script “PotionType”.
2.
```
5 references
public enum PotionType
{
    RestoreResource = 0

}
```
3. Add a new enum value, the example below adds “BuffStat”.
4.
```
5 references
public enum PotionType
{
    RestoreResource = 0,
    BuffStat = 1

}
```
5. Create a new class “BuffStat”.
6. Implement ‘Potions’ abstract class. (Remember to use the GWLPXL.ARPGCore.Items.com namespace).
7. Code the behavior.

IInteract

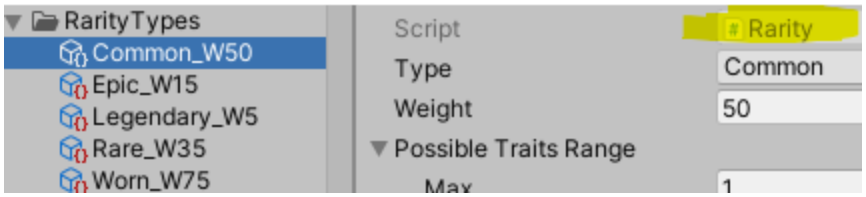
Found in

- Assets/GWLPXL/ARPG/_Scripts/Core

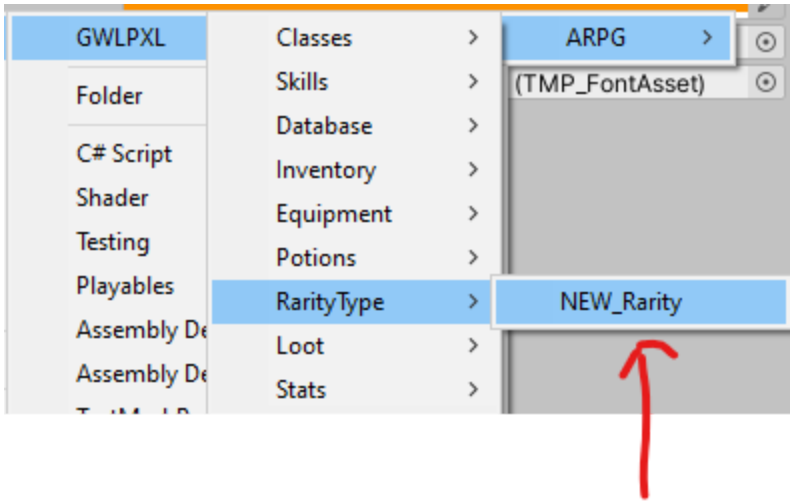
Used for non-combat interactions in the demo. “Loot.cs” and the OnClick objects in the demo scenes show examples of implementation.

Loot Rarity

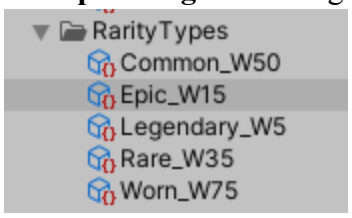
Creating a Custom Loot Rarity Type

1. Open up the “Rarity” script by either selecting the *Script* from the Scriptable Object or by searching for “Rarity” in the search bar.
2. 

3. Add a the new enum value as the new item rarity you want to add.
4.
```
6 references
public enum ItemRarity
{
    Common = 0,
    Worn = 1,
    Rare = 5,
    Epic = 10,
    Legendary = 15
}
```
5. For instance, if I wanted to add “Magical”, I might put it between Rare and Epic and assign it a number between Rare and Epic, like so:
6.
```
6 references
public enum ItemRarity
{
    Common = 0,
    Worn = 1,
    Rare = 5,
    Magical = 6,
    Epic = 10,
    Legendary = 15
}
```
    a. It’s not important that you assign numbers, but I find it good practice and way to keep enums consistent throughout the lifetime of the project.
    b. You can also just add it after legendary and continue on (e.g. Magical = 16)
7. Now create a new Rarity Scriptable Object by either
8. Right click -> Create -> GWLPXL -> RarityType -> NEW_Rarity

9. 

10. Or **duplicating** an existing rarity:

11. 

12. Assign the new values to the new Rarity Scriptable Object:

13. 

   a. The *Type* is the enum value.
   b. The *Weight* is how often it will drop, this is relative to the other weights. A higher weight will have a higher chance to drop than a lower weight. You will see the % drop rates once you create a LootDrop and Table.
   c. *Possible Traits Range* is how many **random** traits it might drop with. This **constrains** Equipment Trait values, e.g. this is the absolute maximum that could drop.

   d. *Color* is the color of the UI text and its UI tint in the inventory.
   e. *Loot Text Prefab* is the prefab of the loot text, you can configure this for different types of drops if you desire.
   f. *Font Assest* is from TextMeshPro, we assign different *Font Assets* because this is used to generate the differing colored texts.
14. To create a new *Font Asset*, navigate to
   a. Assets/GWLPXL/ARPG/TMPro_Fonts
15. Duplicate an existing *Font Asset*. Rename it, and assign it to the newly created rarity.
16. You can repeat the process for a new *Loot Text Prefab*,
   a. Assets/GWLPXL/ARPG/Prefabs/UI/Dungeon/LootText

Weapon Buffs

Creating a Custom Weapon Buff

Example weapon buffs can be found at

- Assets/GWLPXL/ARPG/_Scripts/StatusChanges/Scriptables/WeaponBuffs

Weapon Buffs consist of two parts. The first part is a Scriptable Object, which holds the data for the buff. The second is a Monobehaviour that lives on the weapon itself, and it's responsible for the buff behavior. Let us look at an example.

The following buff allows you to apply an additional FLAT damage amount to the weapon, a +X mod.

```
namespace GWLPXL.ARPGCore.StatusEffects.com
{

    [CreateAssetMenu(menuName = "GWLPXL/ARPG/StatusChanges/WeaponBuffs/New FLAT Damage")]

    Unity Script | 0 references
    public class ApplyAdditionalFlatDamage : WeaponStatusChanges
    {
        public DamageSourceVars_NoActor AdditionalDamage = new DamageSourceVars_NoActor();


        [System.NonSerialized]
        Dictionary<Transform, WeaponBuffTracker> trackers = new Dictionary<Transform, WeaponBuffTracker>();
        8 references
        public override void Apply(Transform[] weapons, IAbilityUser forUser)
        {
            Enable(weapons, forUser, trackers);
        }

        7 references
        public override void Remove(Transform[] weapons, IAbilityUser forUser)
        {
            Disable(weapons, forUser, trackers);
        }

        7 references
        protected override IWeaponStatusChange CreateIWeaponMono(Transform forTransform)
        {
            AdditionalDamageSource_NoActor source = forTransform.gameObject.AddComponent<AdditionalDamageSource_NoActor>();
            source.Vars = AdditionalDamage;
            return source;


        }
    }
}
```

The Scriptable contains the variables ("AdditionalDamage" in this case), a tracking dictionary "trackers", and overrides Apply(), Remove(), and IWeaponStatusChange from the base class. For the most part, you can copy and paste the trackers dictionary and Apply(), Remove() methods to any new custom buff.

The unique parts of the buff include the variables (these can be whatever, in this case "DamageSourceVars_NoActor" is just a plain C# class that holds float and int values).

The "CreateIWeaponMono" adds the component to the weapon. This component contains the unique behavior of the buff and must be made. Let's look at our "AdditionalDamageSource_NoActor" to see an example of one.

```
Unity Script | 2 references
public class AdditionalDamageSource_NoActor : MonoBehaviour, IWeaponStatusChange
{
    public DamageSourceVars_NoActor Vars = new DamageSourceVars_NoActor();
    bool active = false;
    IAbilityUser self = null;
    10 references
    public void DoChange(IAttributeUser other)
    {
        if (IsActive() == false) return;
        IReceiveDamage damaged = other.GetInstance().GetComponent<IReceiveDamage>();
        if (Vars.DamageMultipliers.PhysicalMultipliers.BasePhysicalDamage > 0)
        {
            damaged.TakeDamage(
                Vars.DamageMultipliers.PhysicalMultipliers.BasePhysicalDamage,
                Types.com.ElementType.None);

        }

        for (int i = 0; i < Vars.DamageMultipliers.ElementMultipliers.Length; i++)
        {
            if (Vars.DamageMultipliers.ElementMultipliers[i].BaseElementDamage > 0)
            {
                damaged.TakeDamage(
                    Vars.DamageMultipliers.ElementMultipliers[i].BaseElementDamage,
                    Vars.DamageMultipliers.ElementMultipliers[i].DamageType);
            }
        }
    }

    8 references
    public Transform GetTransform() => this.transform;

    16 references
    public bool IsActive() => active;

    19 references
    public void SetActive(bool isEnabled) => active = isEnabled;

    9 references
    public void SetUser(IAbilityUser myself) => self = myself;
}
}
```

This class derives from both **MonoBehaviour** and **IWeaponStatusChange**. For the most part, the GetTransform(), IsActive(), SetActive(), and SetUser() can be copy and pasted. The unique behavior must be written in the method“DoChange()”, as this runs when the damage is dealt.

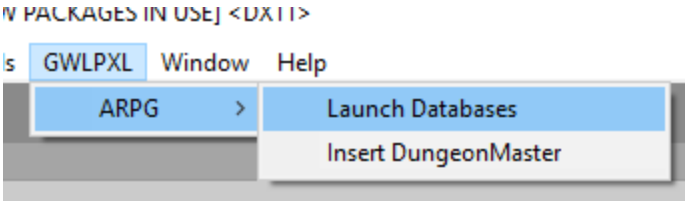
In this case, if the buff is active, then we use the variables (Vars) to call into the respective TakeDamage method of the other. This “DoChange()” method happens when the associated damage dealer (weapon) performs its damage. You can trace its logic in any of the IDoDamage derived classes (found on the damage dealer prefabs, Assets/GWLPXL/ARPG/Prefabs/DamageDealers).

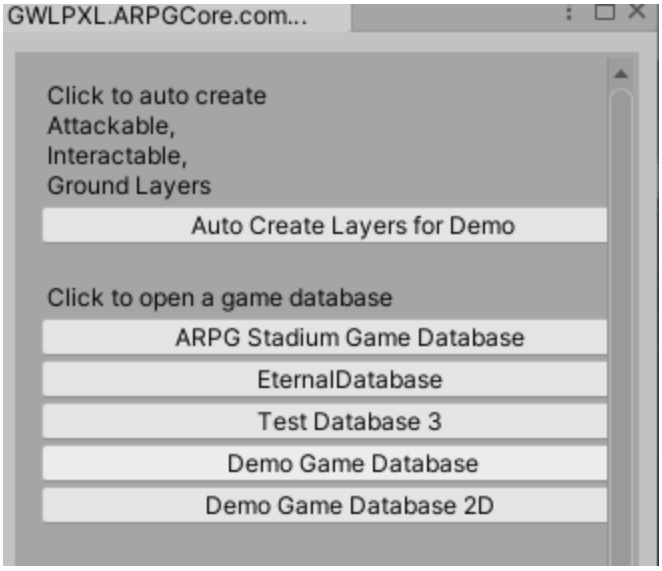
In short, you need a custom **MonoBehavior** component that derives from **IWeaponStatusChange** and has your unique custom logic in the DoChange() method. Then, you will need to create a Scriptable Object that derives from WeaponStatusChanges and adds your newly created component to the transform.

Wearables

Creating Wearables

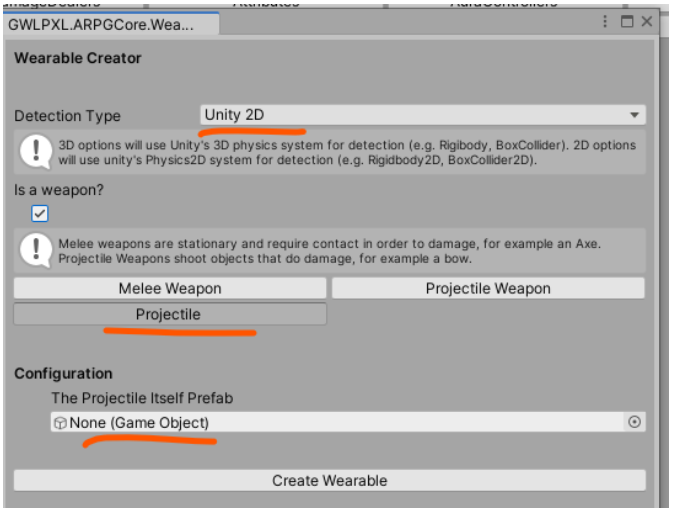
To create Wearables

1. Launch the Game Database.
   a. 

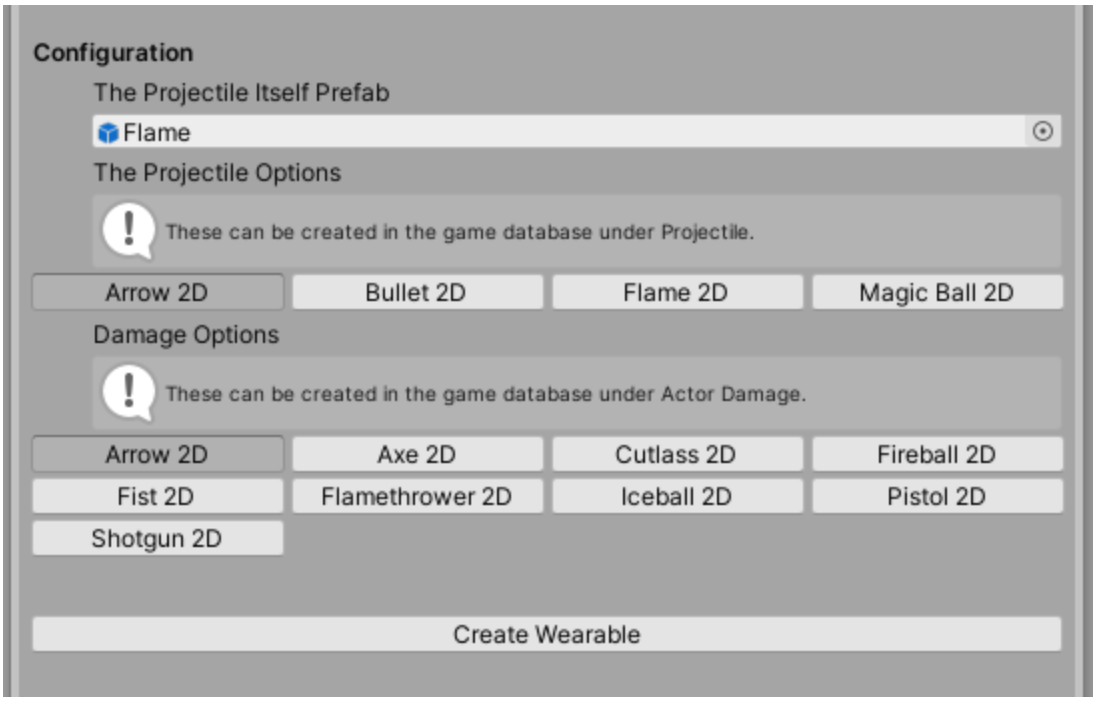
2. Choose your database.
   a. 

3. Click on “Wearables”
   a. 

4. Choose the **Detection Type**.
5. Enable **Is a Weapon** (if it does dmg, like melee and projectiles).
6. Insert the **Prefab** that represents that weapon.

a. 

7. You'll then see a list of options.

a. 

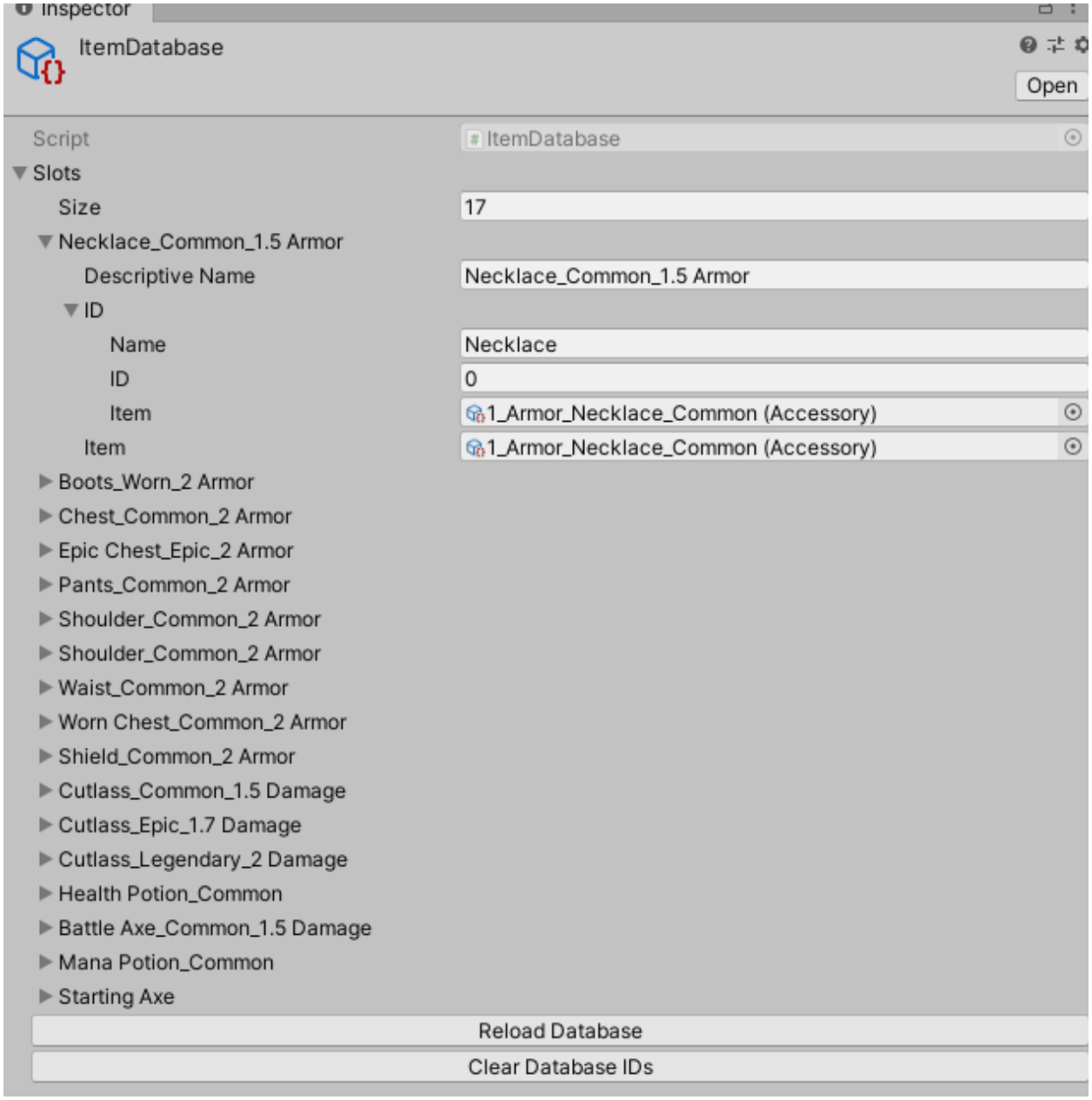
8. Choose the options and hit 'Create Wearable'.
9. Save it somewhere in the project.

Now you have a new Wearable that can be used! The system will auto equip all the necessary parts on to that prefab while keeping anything that's already on it.

Additional Systems

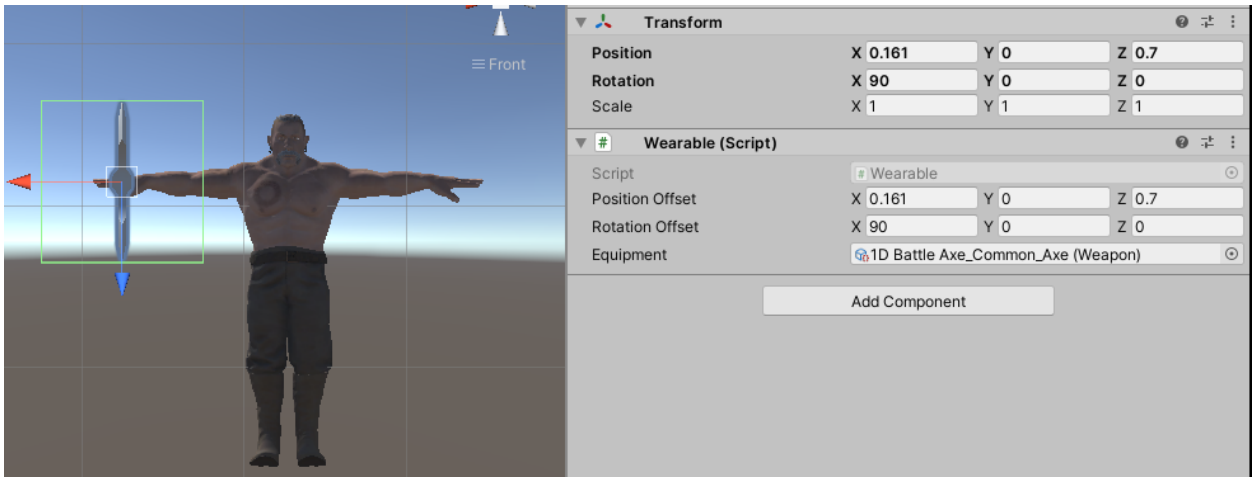
Saving

The saving system leverages scriptable objects as databases. When you save and load, you are saving the scriptable object's unique id and loading it by finding that unique id. If using the databases, the unique id is the index of the entry.



IWearable and the Clothing System

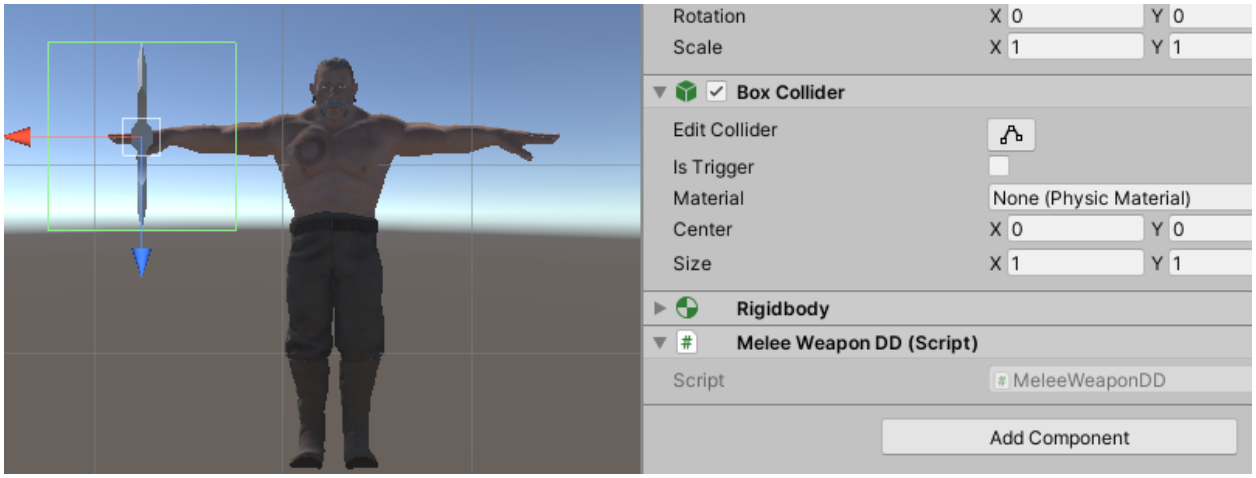
Any piece of equipment you want to wear must be a *Wearable*. In the example below, the axe is a 3D mesh that is parented to the right hand and positioned so it fits.



Once you have the position and rotation appropriately configured, copy those numbers over to the *Position Offset* and the *Rotation Offset*, respectively. This is the position and the rotation the system will use once the item is equipped.

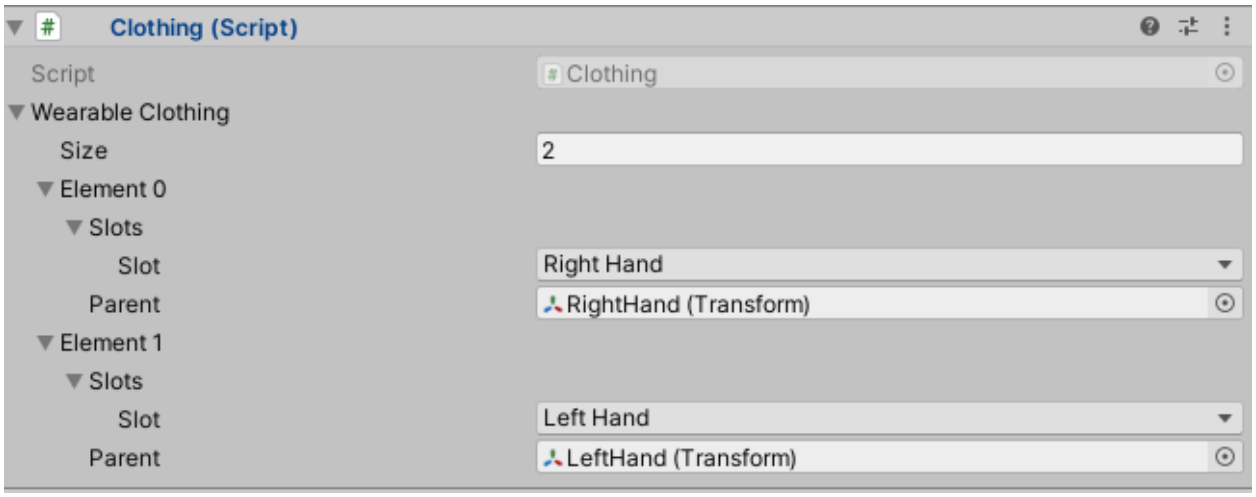
Assign the *Equipment* scriptable object, this is the *Equipment* you want to be represented by the *Wearable*.

For weapons, add a *MeleeWeapon_DD_Trigger* prefab as a child. In the example below, I've attached the *MeleeWeapon_DD_Trigger* to the axe and renamed it. This is the object that defines the hitbox.



To customize the hitbox, click *Edit Collider* on the attached collider. Fit it to the size of the hitbox you want.

Once you have *Wearables*, use the *Clothing* component to assign where to wear the *Wearables*. The *Clothing* component assigns *Slots* to *Transforms.* In the example below, I've assigned the *Slot* "Right Hand" to the 'RightHand' transform of my player object. I repeated the same with the left.



Interfaces & Other System Details

Namespaces and Scripts

The Scripts are categorized by their namespace (e.g. ‘GWLPXL.ARPGCore.Animations.com’ will access the Scripts under the ‘Animations’ folder seen below).

I want the system to be customizable for your own needs, which is why the Scripts are delineated as such.

Additionally, Each namespace typically has an Interface or interfaces that you can attach to existing Scripts in order to provide functionality (if you so choose).

Items abstract class

[]

Systems in Progress

Quests and IQuestUser

A system designed to track and save quests.

Status Changes

Including DOTS and buffs.

Rename IReceiveDOT to IREceiveDebuff, and a dot is a type of debuff.

Scriptable Game Events

Game Event & Game Event Listener.

Create a new Game Event.

Attach a Game Event Listener.

Example found in...

Upcoming update notes

1.21.9 (aiming for end of April)

1.22.0

Large changes! Back up before you update. Alternatively import into a new project and transfer your data over.

<STRONG>Major Changes</STRONG>

New interface IActorHub – Large change, this is a one step shop that defines all of the actor systems.

Re-organized the AbilityStatusChanges and WeaponStatusChanges – renamed them to AbilityMods and StatusMods and put them in their respective folders.

IAbilityUser new methods

IActorHub GetActorHub();

void SetActorHub();

Moved 3d actors over to the new custom statemachine

Moved the following classes to that namespace into new folder (see documentation for details in the Misc section)

New namespace, Ability.mod

Re-organized Player and Enemy structure, now easier to manage in the editor.

<STRONG>Abilities</STRONG>

Created a new namespace GWLPXL.ARPGCore.Abilities.Mods.com – now houses the ability mods and apply weapon mods

Ability Mods added

AoE Knockback

Knockback

Explosive

Dash

ARPG Attributes, Items, and Abilities

Leap

Deflect Projectile

Added ModHelper static class

Added functions for adding/removing mods

<STRONG>Attribute System</STRONG>

Added KnockbackResistance – reduces the force of a knockback

<STRONG>Combat System</STRONG>

Renamed IWeaponStatusChange to IWeaponModification for clarity

Changed CasterProjectile and ShooterProjectile to now include options for start, end, and in between times for the ability.

Caster Projectile – options for projectiles anytime during the cast

Shooter Projectiles – options for projectiles anytime during the cast

Added example chain lightning

Added AdditionalLauncher class

Added Knockback and Knockback AOE

Added Explosive Mod

Added example of spinning magic orb (uses the AdditionalLauncher class)

Added static class CombatHelper

Added new damage functions on the CombatHandler

Refactored damage dealers to common functions in the combathandler

<STRONG>Status Effect System</STRONG>

Added new Status Effects classes

ARPG Attributes, Items, and Abilities

Added Slow example

Added Freeze Example

Added StatusEffectHelper static class

Added Inflict Status effect class

<STRONG>Enemy AI System</STRONG>

Added new classes for the enemy ai – both 2d and 3d

Added an AI state machine for animations

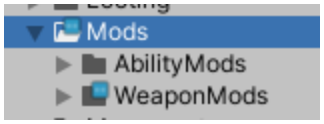
Added an AI blackboard for runtime data

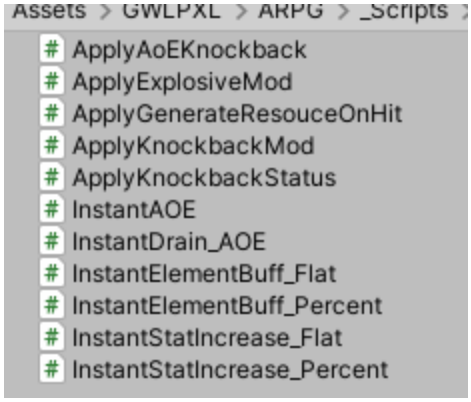
Added an AI simple brain to show how the blackboard can work with the statemachine

<STRONG>Misc</STRONG>

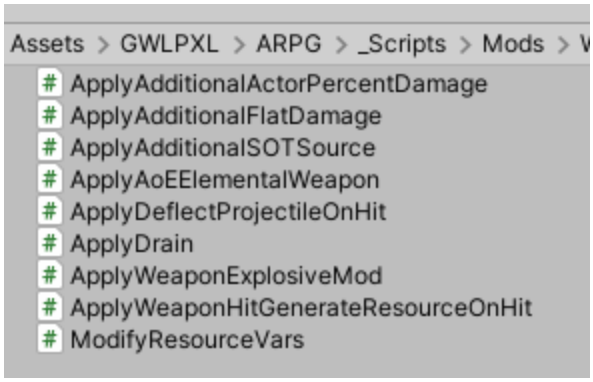
Added new FollowPlayerRotate class for camera movement (thanks to WaynesWorld Omaha)

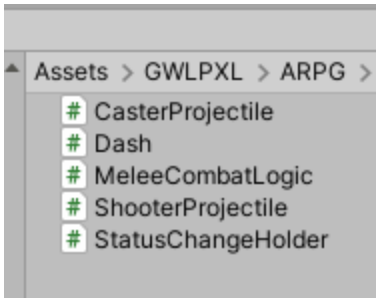
And much more. Check out Discord if you have questions.





ARPG Attributes, Items, and Abilities





ARPG Attributes, Items, and Abilities

1.21.8

Added new Art for the Demo scene and more example scripts for Interactables.

Added new example scripts and objects for Interactables (find them in the demo scene).

Updated Enemy Movement System (preview for 2D first, then moving to 3D)

- Added AI state machine for enemy movement
- 2d idle, 2d walker, 2d aggro
- (does not attack yet, that is next update)

Updated Player Movement System (preview for 2D first, then moving to 3D)

- Revamped 2d movement state machine
- Added 8 direction example of new 2d move system
- Added 4 direction example of new 2d move system
- Added interrupt options on ability – on damaged, on dead, on another ability used, on collision
- Added new ability interrupt classes
- Added slash 2d
- Added example interrupts on 2d character

ARPG Attributes, Items, and Abilities

1.21.7

<STRONG>Abilities</STRONG>

Modified IAnimator lookup on the AbilityUser script, looks on parent then looks in children

<STRONG>Combat</STRONG>

Modified IDoDamage, added new public method

Updated default damage dealers

<STRONG>Databases</STRONG>

Item stats can be imported/exported to json via the item database SO

Updated some brief descriptions on some of the systems in the documentation

<STRONG>Factions</STRONG>

Faction system experimental, see it in the new faction namespace

Added Faction Manager to the ARPGDungeonMaster_Singleton

Added example Faction Relations Template

Added Enemy Faction class

Added Player Faction class

Added Faction UnityEvents

<STRONG>Player Input</STRONG>

Added InteractMouse3D – if you want to use the Interact system with separate movement controls

<STRONG>Quests</STRONG>

ARPG Attributes, Items, and Abilities

Added quest requirement – faction requirement

Added quest reward – faction reward

Modified QuestChain Requirement abstract classes

Modified IQuestUser, IQuestGiver interfaces

<STRONG>Shopping</STRONG>

Added shopping requirement – faction requirement

Added Shop Requirement abstract class

Modified IShopper, IShopKeeper interfaces

<STRONG>Status Changes</STRONG>

Added DoChange(Transform other) method on IWeaponStatusChange interface

Added WeaponBuff – Deflect Projectiles

Added Deflect Projectiles examples both 2d and 3d

Modified IWeaponStatusChange with DoChange(Transform other) – used for movement changes such as deflecting projectiles

Example Invector Base and Override animators in

Assets/GWLPXL/ARPG/Data/Animations/Animators

ARPG Attributes, Items, and Abilities

1.21.6

<STRONG>2D</STRONG>

Added breakables in the 2d demo scene

Added shopkeeper in the 2d demo scene

Added new mouse over 2d class

Added quest giver in the 2d demo scene

Improved 2D brain class

<STRONG>Databases</STRONG>

Added ability to import/export trait databases

<STRONG>Fixes<STRONG>

Shop can now freeze the player and act as intended

Scaling on the HChu default fist has been fixed

Enemies would stop moving after restarting the scene

Fixed a bug on breakable creation that wasn’t assigning a reference appropriately

1.21.5

Game Database

- **Added 2D game database example.**
- **Added 2D options in the game database.**
- **Created Data2D folder, contains the 2D example database.**
- **Added new options to reflect these changes in the game database actor creator**

Core

- **Added Enemy Top Down brain 2D**

Combat

- **Added Projectile2D class**
- **Added MeleeWeaponDD2D class**
- **Added EnvironmentDamage2D class**
- **Fixed a bug with left handed melee weapons not working after a specific series of equip/unequips**

Movement

- **Added Enemy Top Down Mover 2D**
- **Added Player Top Down 2D Mover**

Player Inputs

- **Added Interact Mouse 2D**
- **Added Player Interact Input Class**
- **OnTriggerEnter2DInteract class**

CanvasUI

- **Changed the floating text. Easier now to allow other tweeners to control the text and added more options.**

ARPG Attributes, Items, and Abilities

ARPG Attributes, Items, and Abilities

1.21.4

<STRONG>Highlight</STRONG>

Added new Wearable Creation window – easily create melee weapons, projectile weapons and projectiles with a button press.

<STRONG>Databases</STRONG>

Added databases actor damage types

Added database projectiles

Added database for melee

<STRONG>Editor</STRONG>

Added new Wearable Creation window – easily create melee and projectile weapons with a button.

Added new editor window for actor damage types

Added new editor window for projectiles

Added new editor window for melee

<STRONG>Misc</STRONG>

Fixed shield scale issue in the demo

1.21.3

<STRONG>Abilities</STRONG>

Added “CanLoop” toggle on Abilities. Example of a looping Ability is the Flamethrower.

<STRONG>Animations</STRONG>

Created new animation slots for pistol, shotgun, and flame thrower.

<STRONG>Combat</STRONG>

ARPG Attributes, Items, and Abilities

Melee Weapon Damage Dealer and Projectile Damage Dealer now include Damage Options, additional multipliers, and optional Status Over Time effects.

Combat formulas! These have been moved to Scriptable Objects which you can override in order to write your own formulas if you desire. They are the same ones found in the static class, but are now easily in view in the editor.

Added new FlameThrower, Pistol, and Shotgun as example wearables in the Prefabs -> Items -> Wearables folder.

<STRONG>Misc & Fixes</STRONG>

Added new art for placeholder flamethrower, pistol, and shotgun.

Upgraded to TMPro_2.1.1 – has an error with TMP dropdowns unless you update – this is a UNITY bug but upgrade resolves (if you have an issue)

Fixed an issue where the item longbow on HChu was null causing errors.

Created examples of the Pistol and Shotgun archetypes on Hchu.

Expected release - Late January / Early February of 2021 - Going live once I record a video for the weapon buffs and ability buffs.

NEW Discord: https://discord.gg/XzR2FUW

1.21 Discord • https://discord.gg/XzR2FUW

Overall Notes

Art

Added a new example player character -HChu. Character is for demo purposes, contact me if you wish to use in a commercial product.

Ability System

Added new ability slots for default abilities.

Added optional ability requirements, e.g. weapon requirement for an ability.

ARPG Attributes, Items, and Abilities

Added SetAbiltiySpeed() – call into this to set the speed of abilities (only)

Changed delays to normalized percents (0 – 1), you may need to readjust these but was necessary for timing animations to speed increase/decreases.

Exposed Event for AbilityEnd

Animation System

Added Trigger and Animator states for Basic Attacks

Added to IANIMATE void TriggerBasicAttackAnimation(string trigger, int index);

Combat

IWeaponStatusChange added new GetTransform Method

Canvas

The old dungeon canvas has now been split into 3 parts:

Player UI, Floating Text, Loot Text.

User's can adjust the PlayerActionBar prefab attached to the PlayerDungeon Canvas to suit their needs. If you're using the "Scene_Canvases" prefab, the updates should automatically take effect.

Added 5 new interfaces and 6 monos for UI control

Input System

Refactored out the inputs for the following systems.

Player Mouse Input Class

Player Canvas Input Class

Player Ability Input Class

ARPG Attributes, Items, and Abilities

Player Aura Input Class

Inventory System

Exposed OnItemAdd and OnItemRemoved to Unity Events

Status Changes

Added new Modifiers for additional damage based on caster % and also flat

Added new Modifier that applies a SOT (Status Over Time)

Renamed the AbilityStatus derived changes to “InstantX”, reflecting that they are instantly applied once the ability is active. These typically only last the duration of the ability.

Refactored the weapon buffs, -> adds a mono that derives from IWeaponStatusChange. These buffs exist on the weapon and their effects happen on damage dealt.

Canvas UI

Removed Unnecessary Canvas Renderer components

TMPRO

Added a new font for “Enemy”, made it default for HP Info

Items

Added a new item type, “Currency”, used to drop gold or similar currency.

Shopping

Exposed new Unity Events for both the Shop Keeper and the UI

ARPG Attributes, Items, and Abilities

ARPG Attributes, Items, and Abilities

Version 1.2

Editor & Convenience Changes

- One button click to create Player, Enemy, Breakable, Searchable, ShopKeeper, or Quest Giver actors.
- New Game Database Scriptable Object
- New Project Settings Scriptable Object
- New Game Database Editor Window for all databases.
    - Abilities
    - Ability Controllers
    - Attributes
    - Auras
    - Aura Controllers
    - Classes
    - Inventories
    - Items
    - Loot
    - Questchains
    - Quests
    - Quest Logs
    - Traits
- Custom Editor Windows for the databases
- New Static Class & Functions for Actor Creations
- Unity Event Classes – integrated into the built-in scripts and accessible to all
    - UnityAbilityEvents
    - UnityAuraEvents

- UnityClothingEvents
- UnityCombatEvents
- UnityDamageEvents
- UnityDropLootEvents
- UnityInputEvents
- UnityItemEvents
- UnityLevelUpEvents
- UnityLootingEvents
- UnityQuestChainEvents
- UnityStatusChangeEvents

And much more! Video and further improvements soon.

New Systems

Shopping!

- Added classes ‘ShopKeeper’, ‘ShopScaler’, ‘PlayerSeller’, ‘PlayerShopper’
- Added IShopKeeper, IShopper, ISeller
- See ShopKeeper Gameobject in Demo_Room
- Full list of classes can be found in _Scripts/Shopping or the shopping namespace

Quests!

- Added classes ‘QuestChainGiver’, ‘PlayerQuester’, ‘PlayerQuestCanvasUser’
- Added IQuester, IQuestGiver
- Added ‘Quests’ and ‘Questchain’ Scriptable Objects
- Full list of classes can be found in _Scripts/Quests or the Quests namespace

ARPG Attributes, Items, and Abilities

Changes to Existing System

Ability

Added Shooter Projectile Logic

Combat

Added a Projectile Combatant Class

Added IProjectile and IShooter

Items

New added item, QuestItem (See Skeleton Key for example)

Saving

Added save functionality for quest progress

Added a switch to choose saving in binary or json plain text

Statics

Separated out 'Formula' class into new classes by responsibility

Added static class 'Combat Resolution' (resolves combat damage, damage formulas are here)

Added static class 'Combat Stats' (displays stats)

Status Changes

Added Weapon Buffs – applied to weapons, not to the caster itself.

Misc

ARPG Attributes, Items, and Abilities

Added a prefab to visualize AOE damage

Added art for a bow sprite and 3d model

Fixes

Damage Over Time was not always registering correctly

Fixed an error with ability saves

Future Changes

Editor Window Changes to improve workflow

Improvements to the existing changes

Shopping

Buying / Selling system with Shopkeeper

IShopper, ISeller, IShopKeeper

PlayerShopper, PlayerSeller, ShopKeeperCanvas_UI

Questing

Questing system - Questchains, QuestLogs, Quest Giver

IQuester, IQuestGiver, IQuestCanvasUser

PlayerQuestCanvasUser, PlayerQuester

Combat

Updated Projectile Weapon system (different than casting system)

Added **Projectile Weapon** class

ARPG Attributes, Items, and Abilities

added **shooter projectile ability logic**

added **Unity scene event integration** for Projectile and MeleeWeaponDD classes

Added **Damage multipliers** for Projectile and MeleeWeaponDD classes

Added **Damage multipliers** and **DOT** options for environmental damage

(description system does not take these new multipliers into account, only the traits for now)

Temporary weapon buffs

Bugs

Fixed a bug with saving abilities

Fixed bug where dots were not applying negative numbers (damage) correctly

Misc

Unity Event integration for the following classes:

Player health, Player Levels, Player Aura user, Player Aura Receiver, Clothing, Ability User, Player Quester, Player Seller, Player Shopper

Unity Scene Events across gameplay scripts (to hook external scripts into them)

Cleaned up the formulas for **CombatResolution**, **CombatStats**, and **Formulas**

Added ability to define UI layers to block raycasts on **Player Nav Mesh Mover** (so the player doesn't move if clicked on a UI element)

Added buy/sell cost to items

PlayerEventController

rename Sprites to sprites (may break references, sorry)

Custom Editors

New Databases and custom editor windows

Player Object creation with a click of a button

ARPG Attributes, Items, and Abilities