

Ultimate Item System

Contents

1. Introduction:	2
1.1. Purpose of the document	2
1.2. Scope of the document	2
1.3 System overview	2
2. Concepts	3
2.1 Key terminology	3
2.2 Naming convention	3
2.3 Interfaces	3
3. Game Data Manager	5
3.1 Item Data	6
3.2 Item Prefab	8
3.3 Item Projectile	11
3.4 Crafting Recipe	12
3.5 Dust Data	16
4. Item	17
4.1 Class Diagram	17
4.2 Item Components	17
4.3 How to add already exist item?	24
4.4 How to create a new item type?	28
5. Inventory	34
5.1 Inventory structure	34
5.2 Player inventory code explanation	35
6. Crafting	38
6.1 Crafting structure	38
6.2 How to create a recipe for an item	40
7. Upgrade item	43
7.1 Upgrade item structure	43
7.2 How to create an upgradeable item	44
8. Save system	47

1. Introduction:

1.1 Purpose of the document

- The purpose of this document is to provide a comprehensive understanding of the item inventory system implemented in the project. This document aims to assist developers in understanding the codebase and providing guidelines for future development. Additionally, this document will provide a detailed explanation of how the inventory system operates, its components, and how they interact with each other.

1.2 Scope of the document

- The scope of this document is to provide a detailed explanation of the item inventory system source code. This document will cover the important structures of this project, like items, inventory, crafting, upgrades, saving systems, etc., and how you can extend items as you like.

1.3 System overview

- The item inventory system is a software component designed to manage the items that a player can carry or possess in a game. It is built using the C# programming language and utilizes the Unity game engine to provide a visual interface to the player.

-The system is composed of several classes, including the ***UIPlayerInventory.cs***, ***UIChestInventory.cs***, ***UICreativeInventory.cs*** classes, which is responsible for managing the user interface of the player inventory, chest inventory, creative inventory respectively, and the ***ItemData.cs*** and ***ItemSlot.cs*** classes, which hold information about the items in the inventory. The system also includes the ***ItemDataManager.cs*** class, which are responsible for holding, loading, and managing the item data, and the ***Player.cs*** class, which represents the player's character.

- The system is designed to be flexible and extensible, allowing for the addition of new item types and categories as needed. It provides several features, including the ability to sort and filter items by category, and the ability to craft new items using the items in the inventory.

- The system is intended to be used as part of a larger game project and can be integrated with other game systems and components as needed. It is designed to be easy to use, extend, and maintain, with well-organized and documented code that can be easily understood by other developers.

2. Concepts:

2.1 Key terminology

- a) *Item*: A unit of data representing an in-game item that can be stored in the inventory system.
- b) *Item slot*: A space in the inventory system where an item can be stored.
- c) *Inventory*: A collection of item slots that hold the items in the game.
- d) *Item type*: A type of specific item.
- e) *Item category*: A grouping of items based on their characteristics such as weapons, tools, resources, etc.
- f) *UI*: User interface, the graphical interface that allows the player to interact with the game.
- g) *Item Prefab*: A pre-made object in the game engine that can be duplicated and used in the game world.
- h) *EventTriggerType*: An enumeration of event types that can trigger actions, such as clicking a mouse button, pressing a key, etc.

2.2 Naming convention:

- Naming correctly helps you easily understand and manage later.

- 1) ItemData: **I_***.
 - 2) ItemPrefab: **IP_***.
 - 3) Crafting Recipe: **CR_***.
 - 4) Projectile Prefab: **PP_***.
 - 5) Upgradeable Item Data: **UI_***.
 - 6) Item Data script (inherited from Scriptable Object): ***Data** (ex: **BowData.cs**, **AnvilData.cs**, **ChestData.cs**, **SwordData.cs**...)
 - 7) Object Pooling script (inherited from UnityEngine.Pooling): ***Spawner**. (Ex: **ArrowProjectileSpawner.cs**, **SwordProjectile002Spawner.cs** ...).
 - 8) UI handle script: **UI***. (Ex: **UIAnvil.cs**, **UIInventory.cs**, **UIPlayerInventory.cs** ...)
- Interface: **I***. (Ex: **ICollectible.cs**, **IUseable.cs**, **IPlaceable.cs** ...)

2.3 Interfaces:

- 1) **IUseable**: This interface can be attached to game objects as items that can be used by a player or character. The interface can contain a method for using the object, such as "Use()", which can be called by the player or character.
- 2) **IShowDamage**: This interface can be attached to game objects such as enemies that can display damage when attacked or health information.
- 3) **IPlaceable**: This interface can be attached to game objects that can be placed or built in the game world (anvil, crafting table, chest, combat dummy, etc.).
- 4) **IEquippable**: This interface can be attached to game objects that can be equipped by a player or character (armor).

- 5) **IDroppable**: This interface can be attached to game objects that can be dropped by a player or character (All items implement).
- 6) **ICollectible**: This interface can be attached to game objects that can be collected by a player or character. (All items implement).
- 7) **IDamageable**: This interface can be attached to game objects that can receive damage (enemies, combat dummy, etc.).
- 8) **ICanCauseDamage**: This interface can be attached to game objects that can cause damage to other objects (all projectiles implement this).

3. Game Data Manger

- The GameDataManager class is responsible for managing item data, item prefabs, projectile prefabs, crafting recipes, and dust particle data in the game.

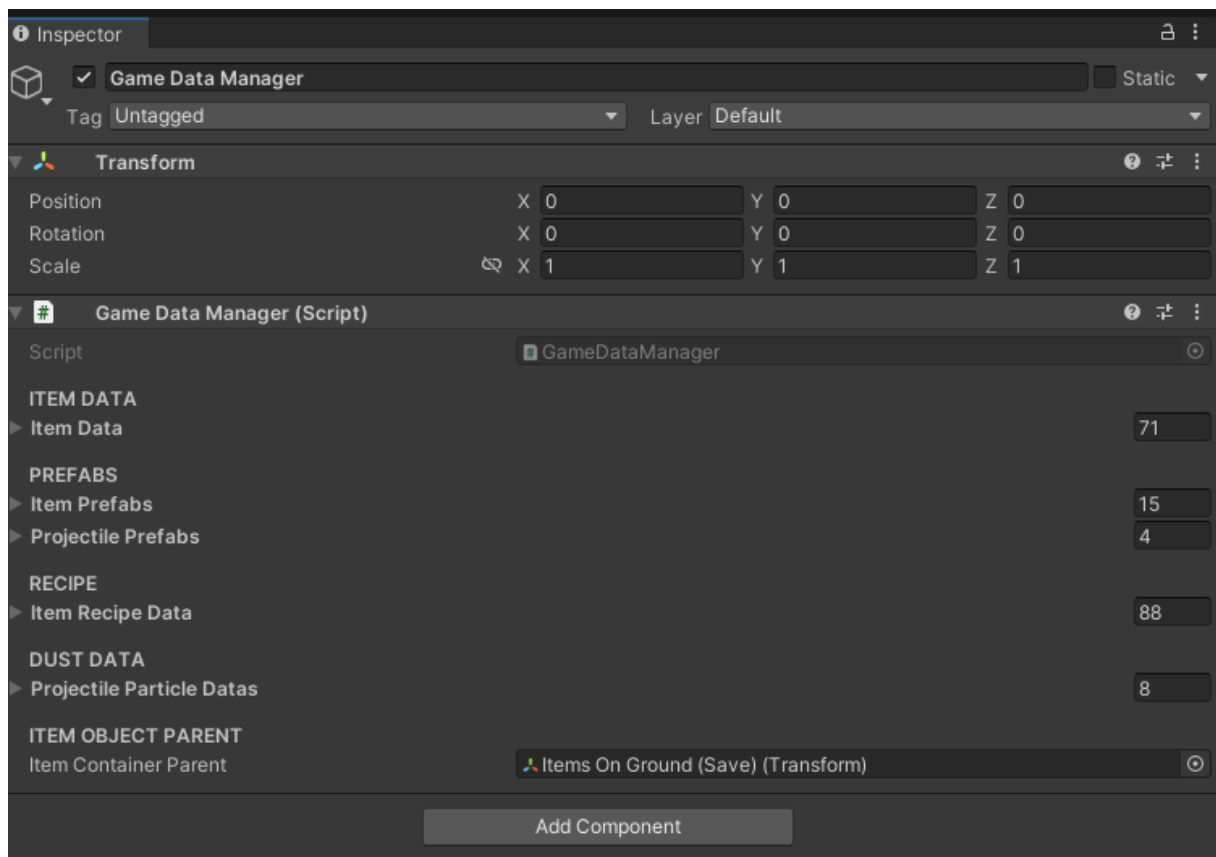


Image 3.1: Game data manager view in Unity Inspector.

3.1 Item Data

- This is a list of all items in the game. When you create a new item, you need to add references to it to use it in the game.

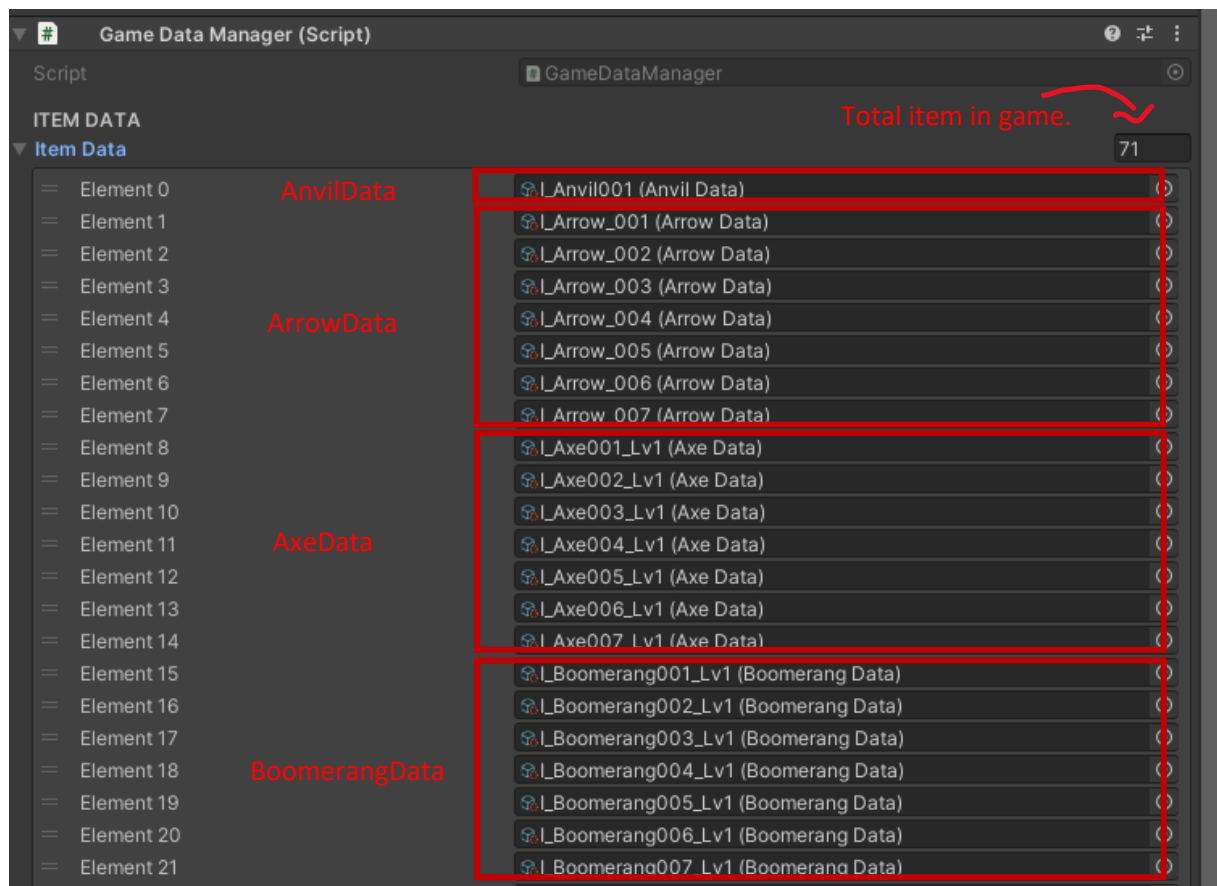


Image 3.2: Item data is stored in GameManager.cs.

- Item data is a scriptable object that inherits from **ItemData.cs** and has the naming convention: **I_***.
- In the image above, **ArrowData.cs** is inherited from **ItemData.cs**, so you can create many arrow types with specific properties to identify this item. I already created some item data, which can be found in *Assets>Create>ScriptableObject>Data>*

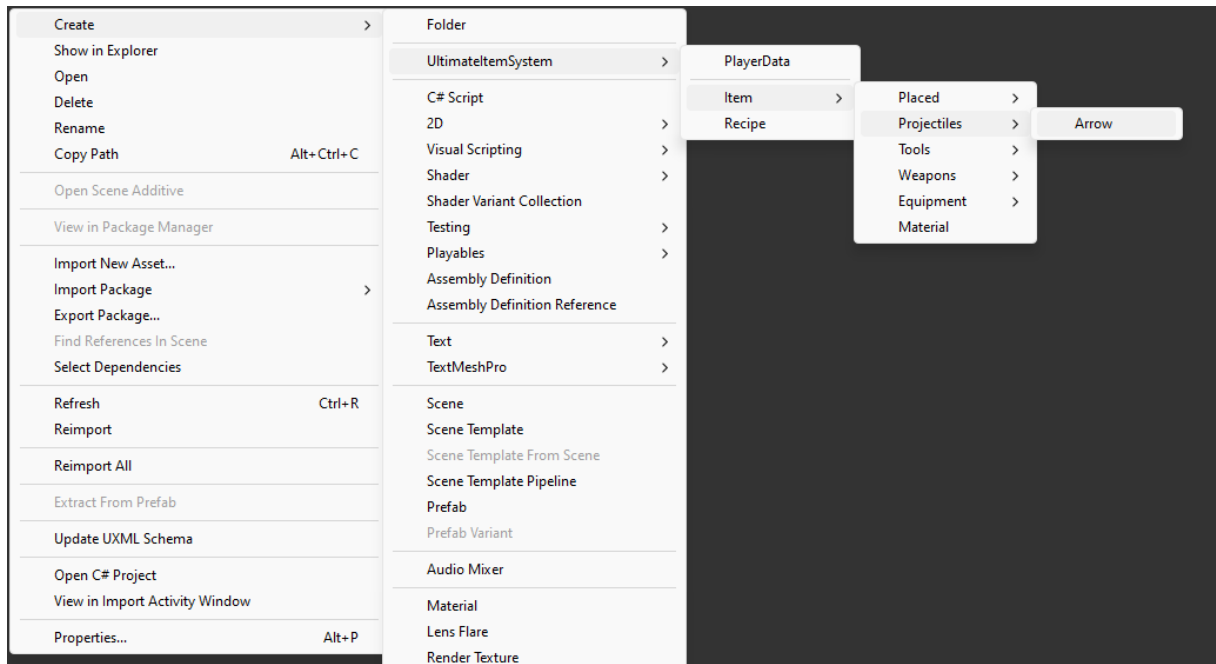


Image 3.3: Arrow data

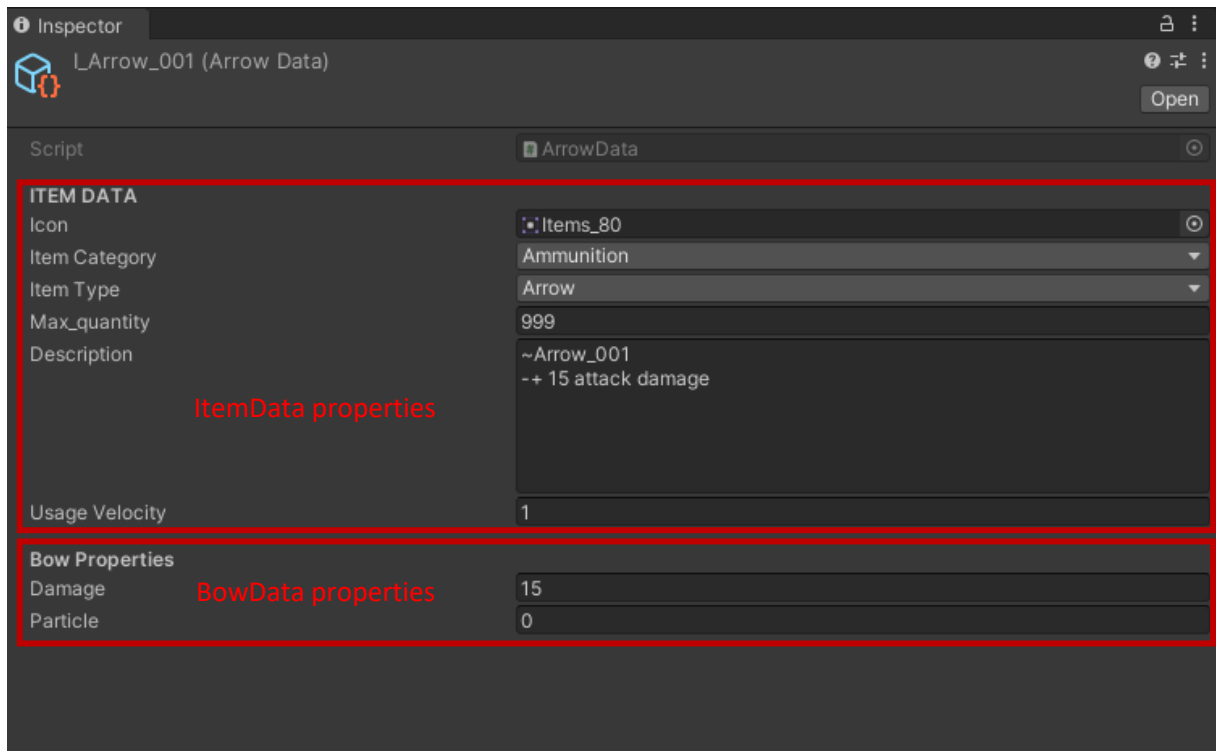


Image 3.4: Inspector view of the ArrowData.cs scriptable object

- Two parts: the `ITEM DATA` header is base **ItemData.cs** properties, and the other is other item properties based on what type of item you want to create.
- It seems very straightforward except:
- + Usage Velocity: Use times in one second – larger means faster (1 means 1 time in 1 second, 2 means 2 times in 1 second, and so on).

3.2 ItemPrefab

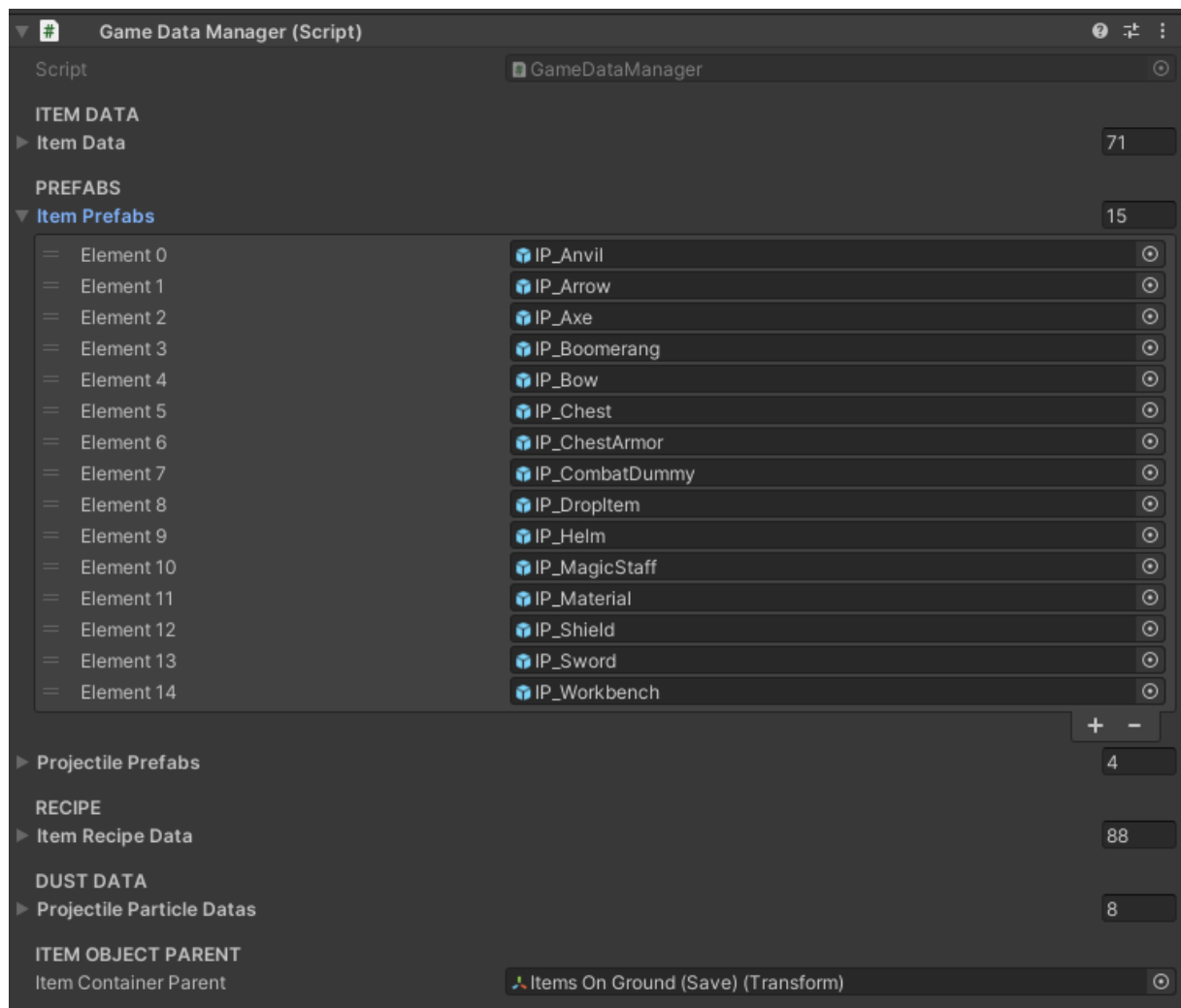


Image 3.5: Item prefabs are stored in GameDataManager.cs.

- Naming convention: **IP_***
- Each item type must have their prefab have the same name in order to create object and handle item logic. Example with a bunch of arrow data above, you only need one **IP_Arrow** prefab.

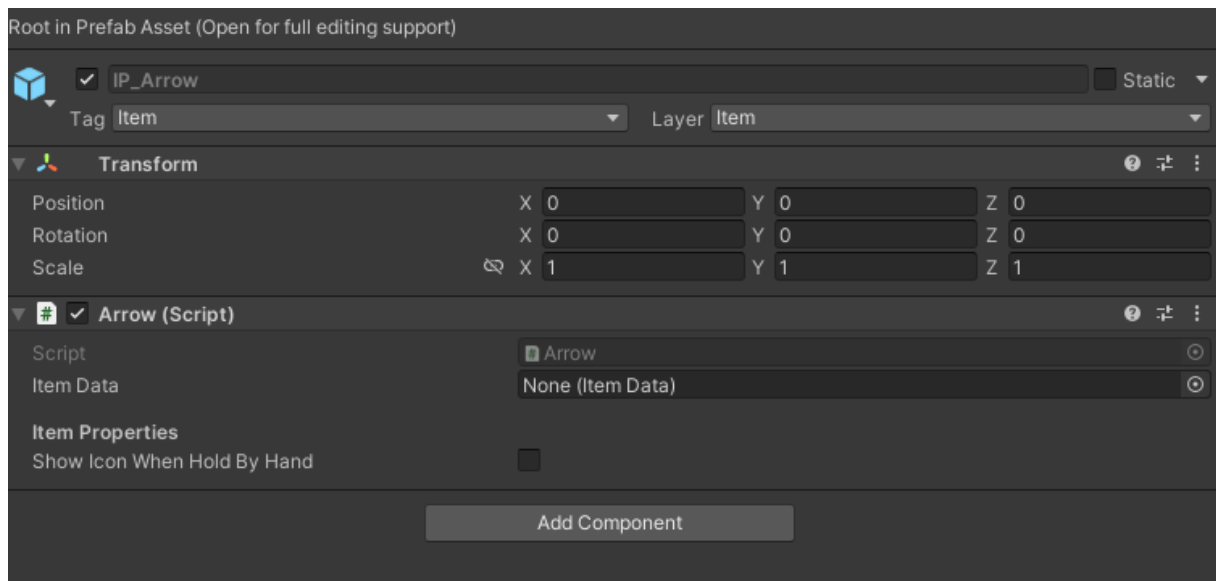


Image 3.6: Unity Inspector view of the IP_Arrow prefab.

- **NOTE:** Name of the item script attach into prefab must same as the item type of this, otherwise it will error.

- Why do we need the same name as item type?

-> Because in **GameDataManager.cs**, I used the method name **GetItemPrefab(string name)** in order to get prefab objects from GameDataManger. Example when you want to create a bow object you need to use this method to get the Bow prefab and pass the **ItemType.Bow** into this method.

```

/// <summary>
/// Gets the item prefab for the item with the specified name.
/// </summary>
/// <param name="name">The name of the item prefab to retrieve.</param>
/// <returns>The prefab for the item with the specified name, or null if no prefab with that name exists.</returns>
5 references
public GameObject GetItemPrefab(string name)
{
    if (itemPrefabByNameDict.ContainsKey(name))
        return itemPrefabByNameDict[name];
    else
    {
        throw new System.Exception($"Not found item prefab name {name} in GameDataManager.cs.");
    }
}

```

Image 3.7: Method gets a prefab by name.

```
/// <summary>
/// Enumeration of different types of items that can exist in the game.
/// </summary>
26 references
public enum ItemType
{
    // Armor
    ChestArmor,
    Helm,
    Shield,
    Shoes,

    // Weapons
    Bow,
    Sword,
    Axe,
    Hammer,
    Boomerang,
    MagicStaff,

    // Ammunition
    Arrow,
    Bullet,

    // Consumables
    Buff,
    Consumption,

    // Crafting materials
    Material,

    // Furniture
    Chest,
    Workbench,
    Anvil,

    // Miscellaneous
    CombatDummy,
    Currency,
    LightSource,
    Null
}
```

Image 3.8: Item type.

3.3 Projectile Prefabs

- As its name implies, it is a projectile when you use a specific weapon.
- Example: when using a sword, you need to instantiate a sword object from the sword prefab. The sword object has a Sword.cs script handles the logic when you use an item. The logic as swinging sword will instantiate a PP_SwordProjectile_001.

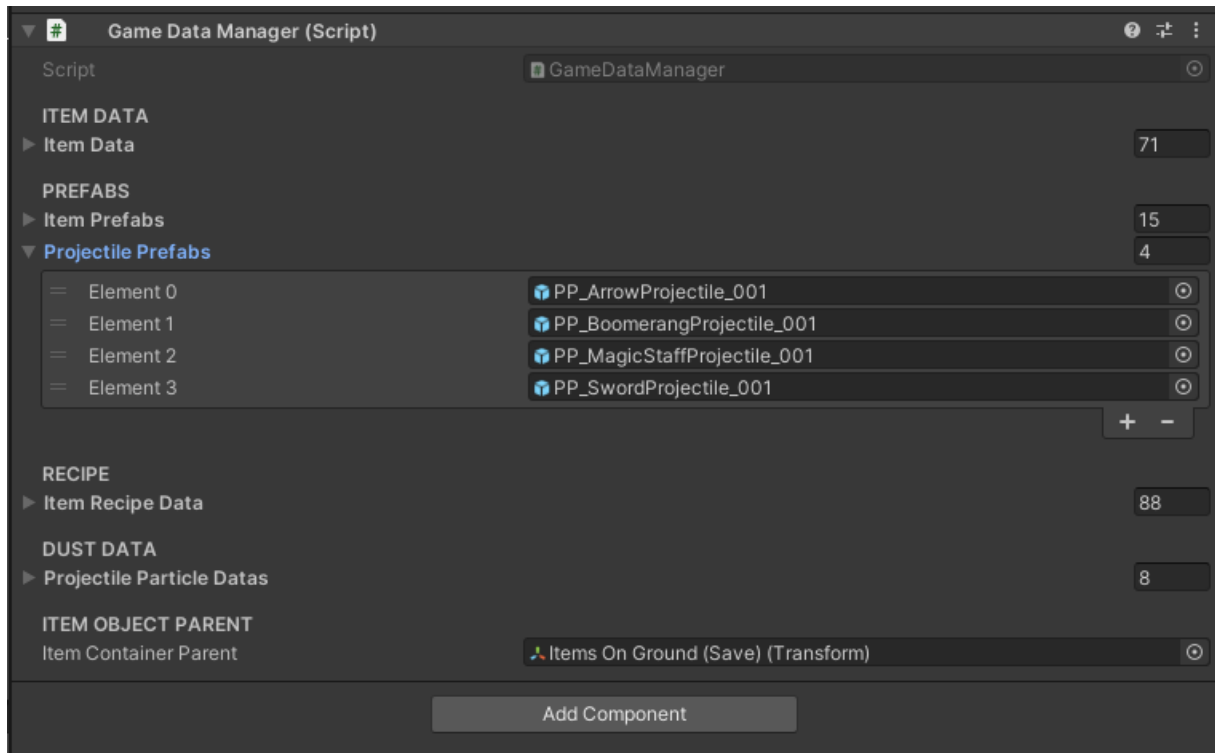


Image 3.9: Projectile prefabs are stored in GameDataManager.cs.

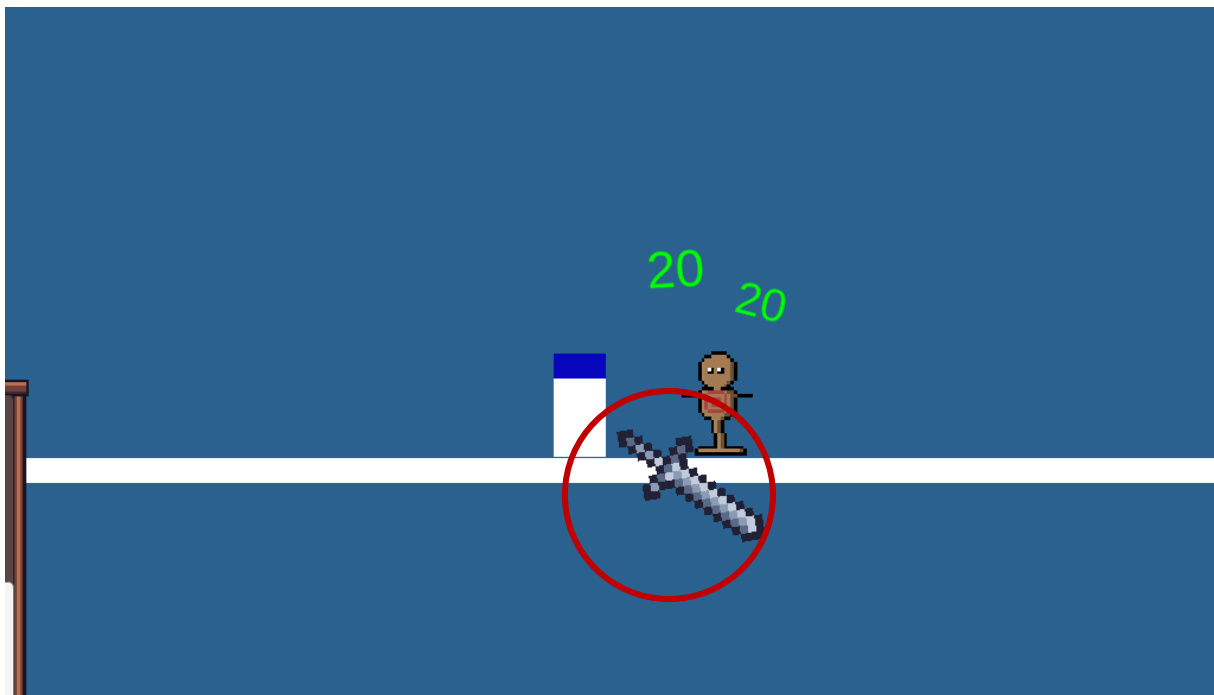


Image 3.10: Object PP_SwordProjectile_001 was created when you Sword item.

3.4 Crafting Recipe

- Naming convention: **CR_***.
- This is a list of all recipe data in the game. When you create a new item recipe, you need to add references into **GameDataManger.cs** to use it in the game.

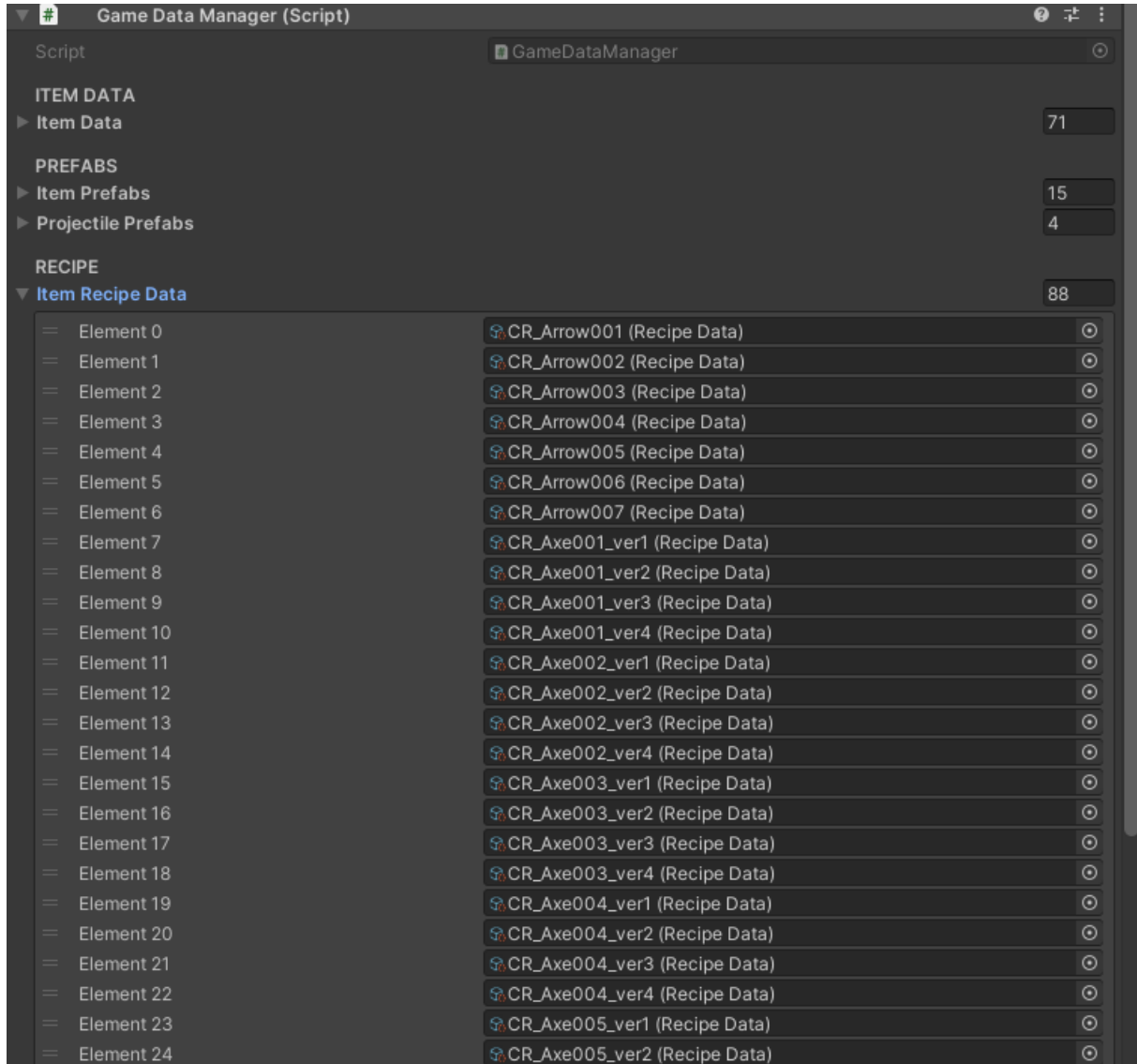


Image 3.11: Crafting recipes data are stored in *GameDataManger.cs*.

- One item can have many recipes, but only one recipe can exist.
- Example:



Image 3.12: Crafting recipe for axe version 01.



Image 3.13: Crafting recipe for axe version 02.

- Although the recipes differ, the result is the same. But because this recipe already exists, you cannot create another one like it; otherwise, an error will occur.

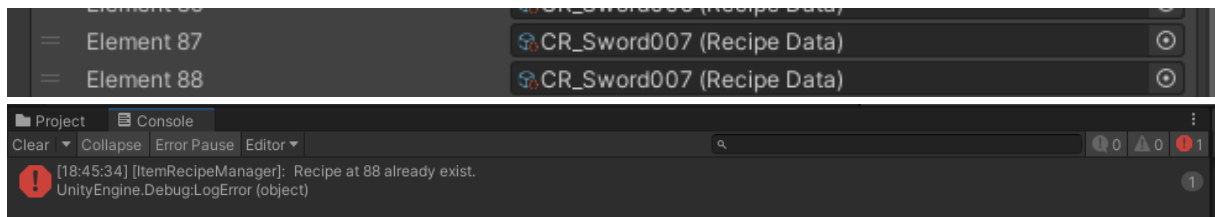


Image 3.14: Error when there are two identical recipes in the data.

- How to create a new recipe?

-> ItemRecipe is a scriptable object, so you can create it by accessing *Assets>Create>UltimateItemSystem>Recipe*.

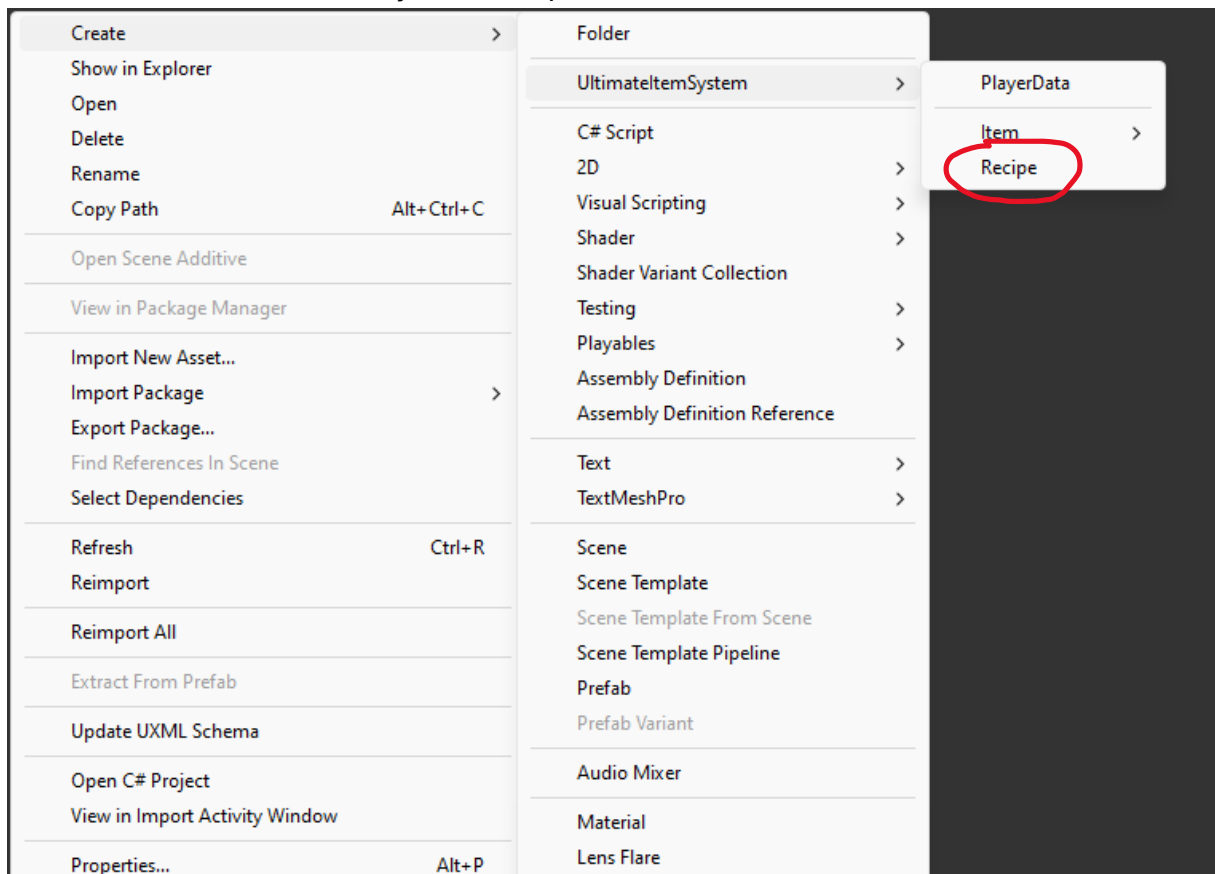


Image 3.15: Item recipe data.

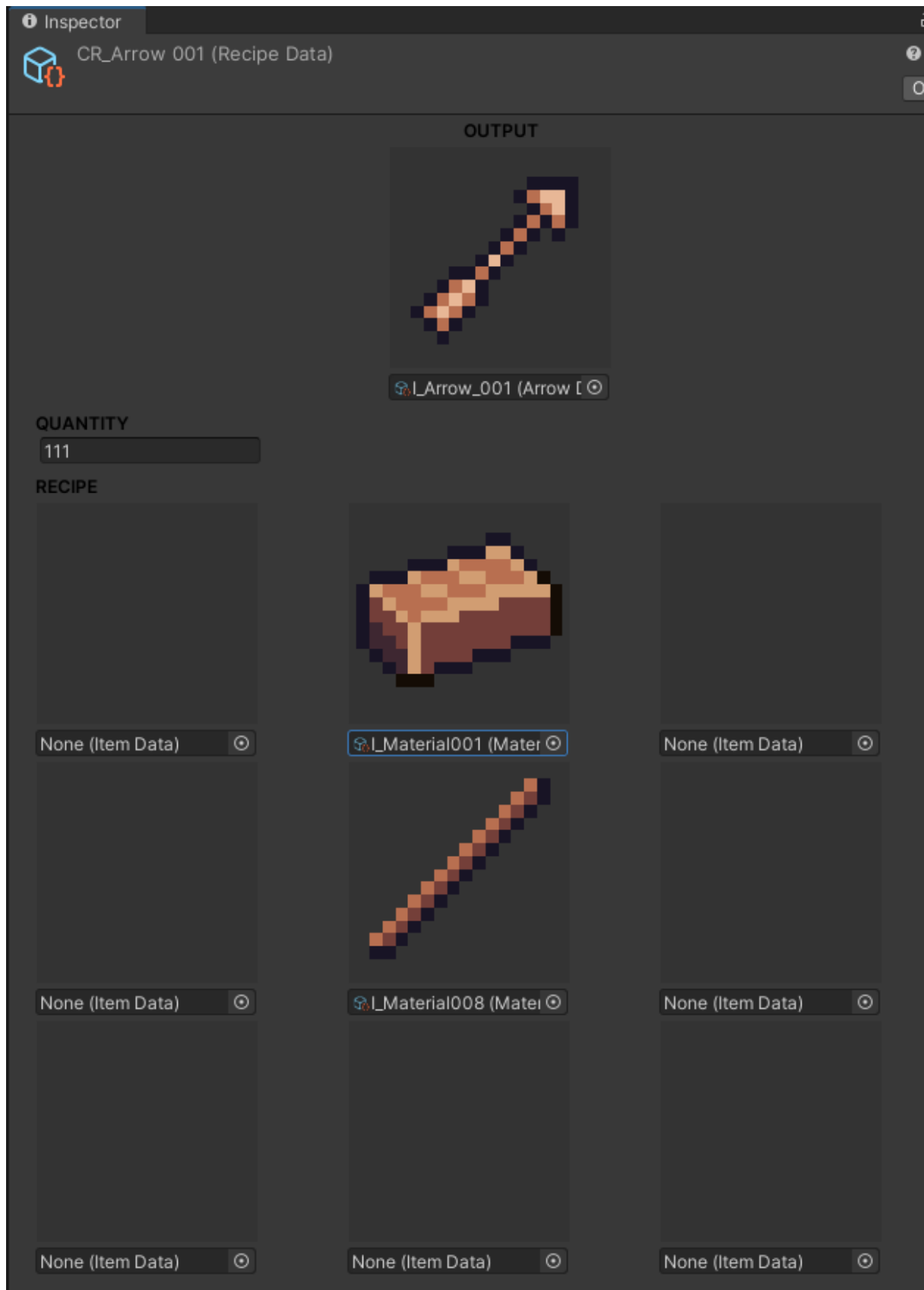


Image 3.16: Arrow recipe data editor view.

-Each recipe has a grid of 3x3 item data as an input source, 1 output item, and output quantity.

3.5 Dust data

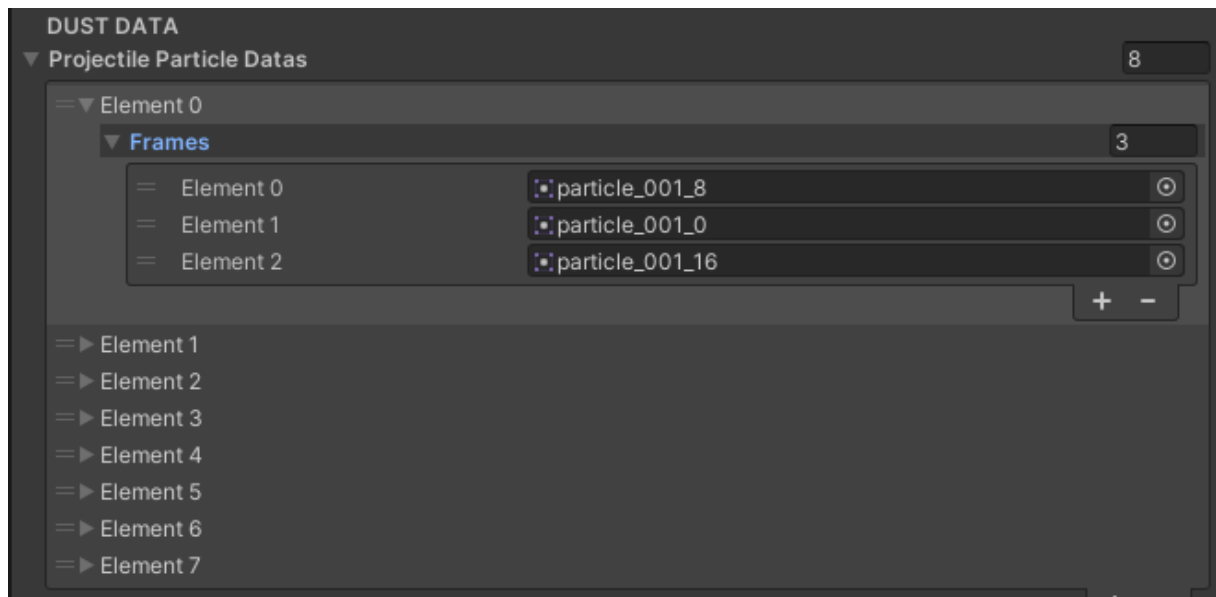


Image 3.17: Dust data are stored in GameManager.cs.

- This is an extension part that can be found in items such as an arrow or a magic staff. So, I'll not talk about this.

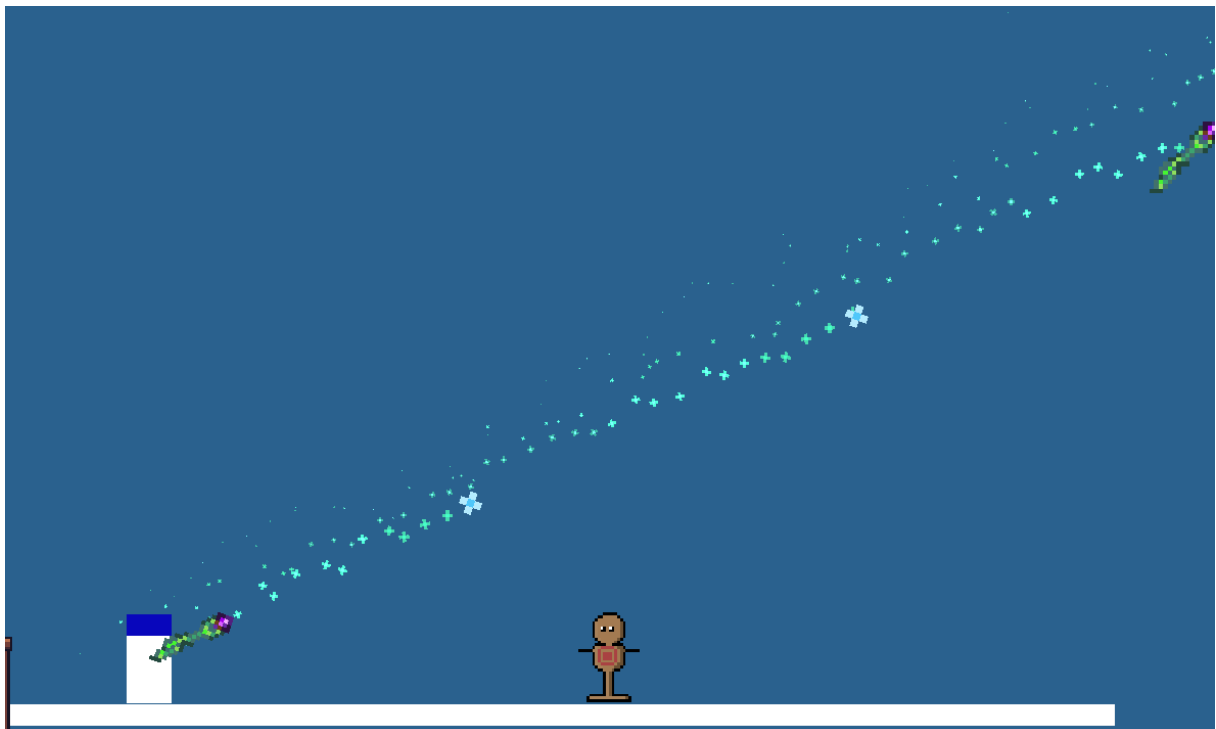


Image 3.18: Particle dust in game view.

4. Item

4.1 Class Diagram

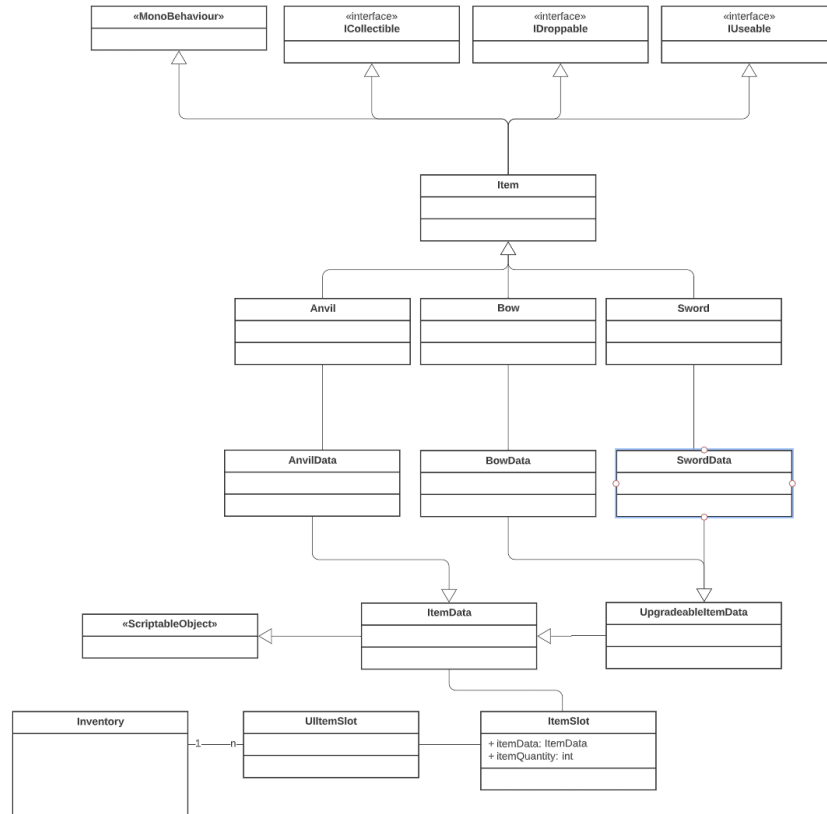


Image 4.1: Class diagram of item

4.2 Item Components

a) Item.cs

- `Item.cs` script is the base class for all item implementations in the inventory system. It is responsible for handling the logic associated with an item when it is used by the player by inheriting and implementing interfaces like ***IUseable***, ***IDroppable***, ***ICollectible***.

This base class provides a foundation for other item classes to inherit from and build upon, allowing for a modular and extensible item system.

- The scriptable object **ItemData.cs** is a base class that holds the item data for each item in the game. This includes properties such as the item's name, description, icon, and category.

b) **ItemSlot.cs**

- **ItemSlot.cs** scripts represent a slot that can hold an item in the inventory. It consists of two properties: *ItemData* and *ItemQuantity*.
 - + *ItemData*: is specific data of an item inherited from **ItemData.cs** (Ex: **IronSwordData.cs**, **DiamondSwordData.cs**).
 - + *ItemQuantity*: the quantity of item in this slot. (Must be less than maxQuantity of this item's data).
- The **ItemSlot.cs** class has several methods that allow for interaction with the slot. The `AddItem()` method takes an *ItemData* instance and a quantity. `RemoveItem()` method removes the specified quantity of items from the slot, ...

c) **UIItemSlot.cs**

- **UIItemSlot.cs** script is responsible for managing the UI element of an item slot in the inventory. It is attached to the prefab of the item slot game object in the UI and communicates with the **ItemSlot.cs** script to get and set the data of the item that the slot represents.
- The script includes methods to update the visual representation of the item slot when the data changes, such as the item icon, count, and background color.
- The **UIItemSlot.cs** script communicates with other UI scripts in the inventory system, such as the **UICreativeInventory.cs** script to perform actions when the item slot is clicked or selected. For example, when the player clicks on an item slot, the **OnSlotClicked()** method is called and passes the slot's index to the **UICreativeInventory.cs** script to handle the event.

d) **ItemPrefab**

- The item prefab in the game is responsible for displaying and holding the logic of each item in the game world. It has a parent script called **Item.cs** which is responsible for implementing the logic for each item in the game.
- The **Item.cs** script is inherited by other item scripts such as **Sword.cs**, **Bow.cs**, and others to implement their specific functionality. These scripts are attached to the prefab as components.
- The item prefab also contains a visual representation of the item in the game world. This is achieved using a 3D model, or sprite, which is attached to the prefab as a child object. The model or sprite is rendered in the game world to show the appearance of the item.
- Additionally, the item prefab also contains colliders and rigidbodies (if it has **IPlaceable** interface) to allow the item to interact with other objects in the game world, such as being picked up by the player or colliding with other items or obstacles.

- Overall, the item prefab plays an important role in the game as it allows players to interact with items in the game world, pick them up, and use them to progress through the game.

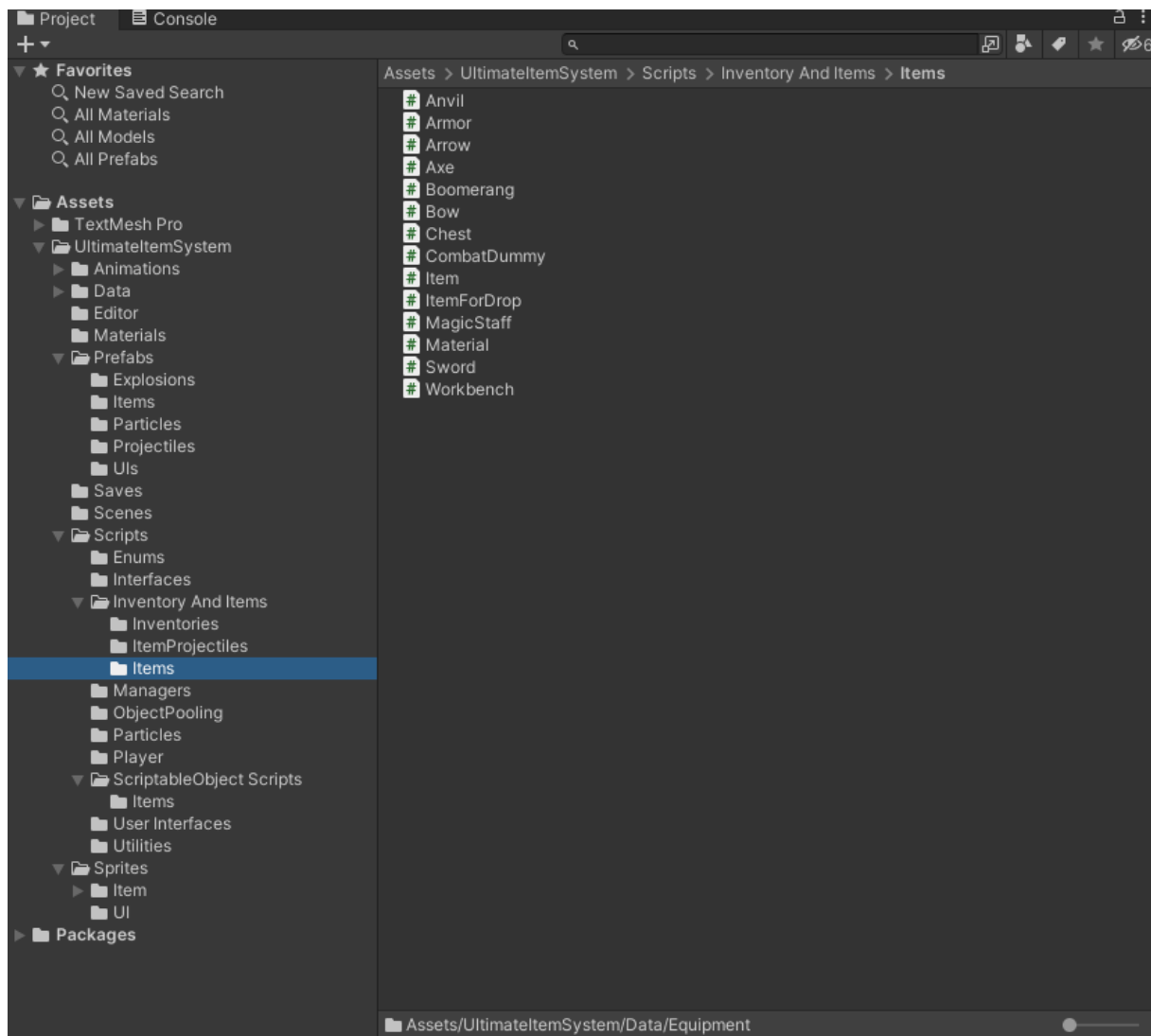


Image 4.2: Item classes

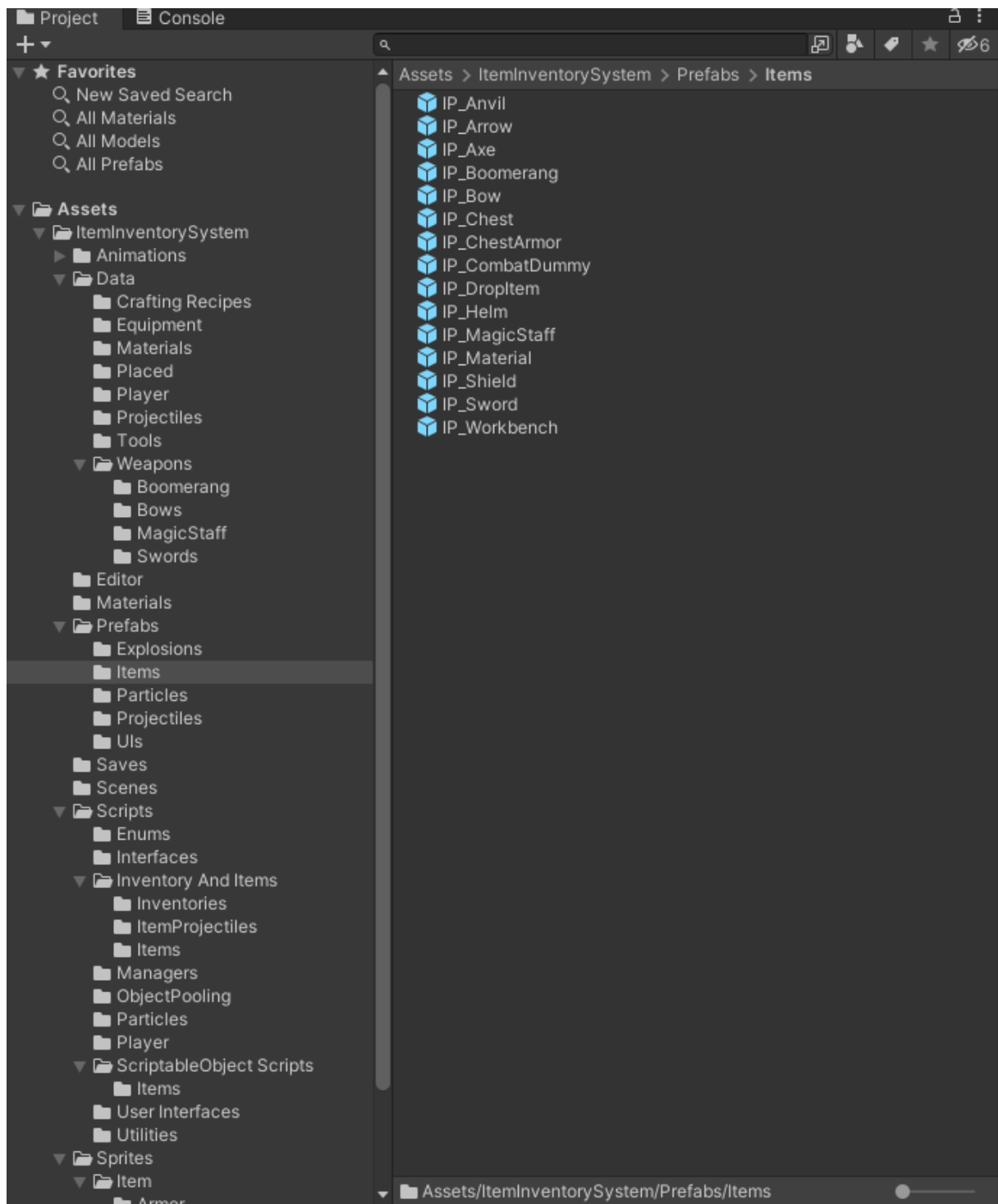


Image 4.3: Item prefabs

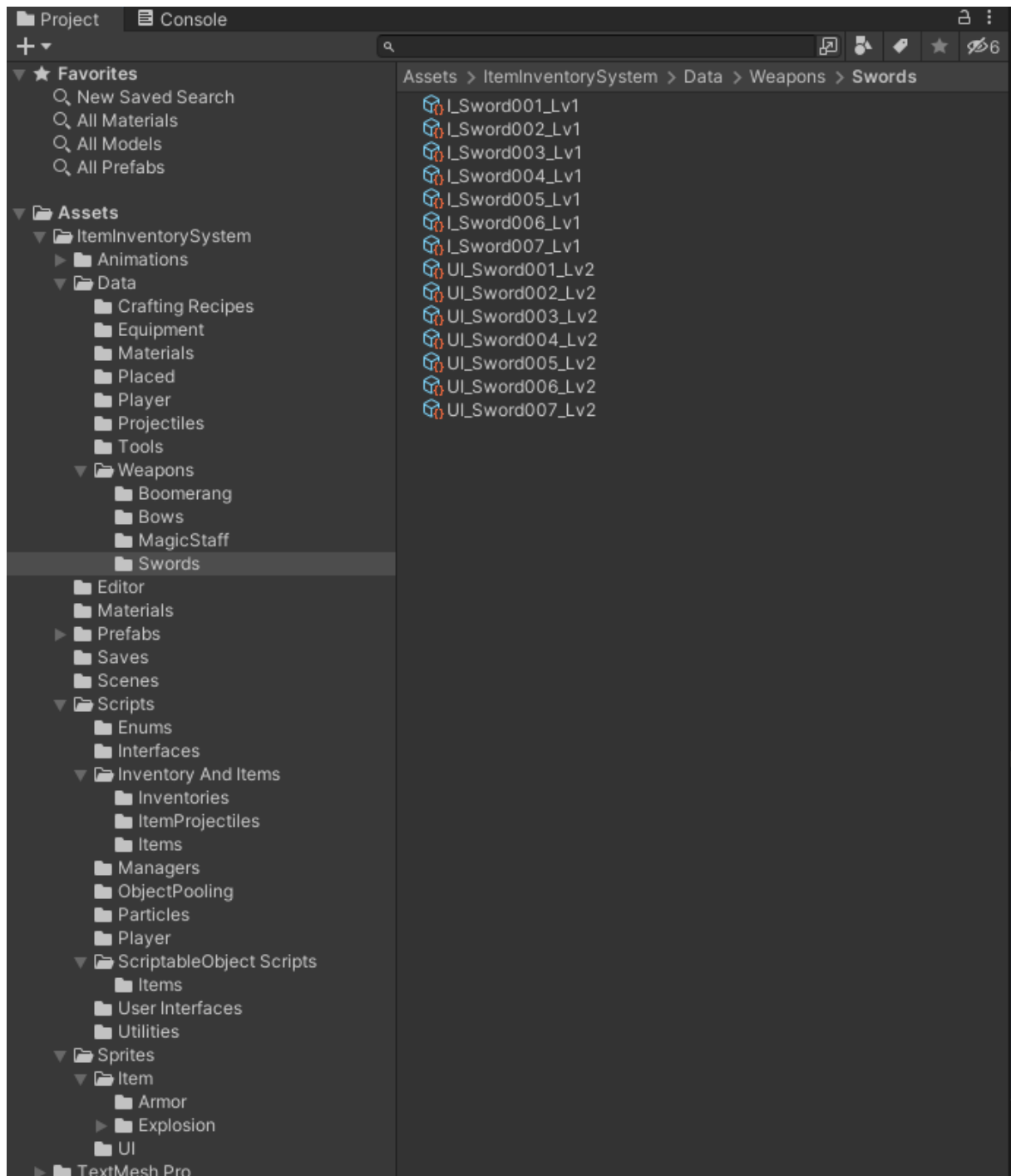


Image 4.4: Item data (Sword data)

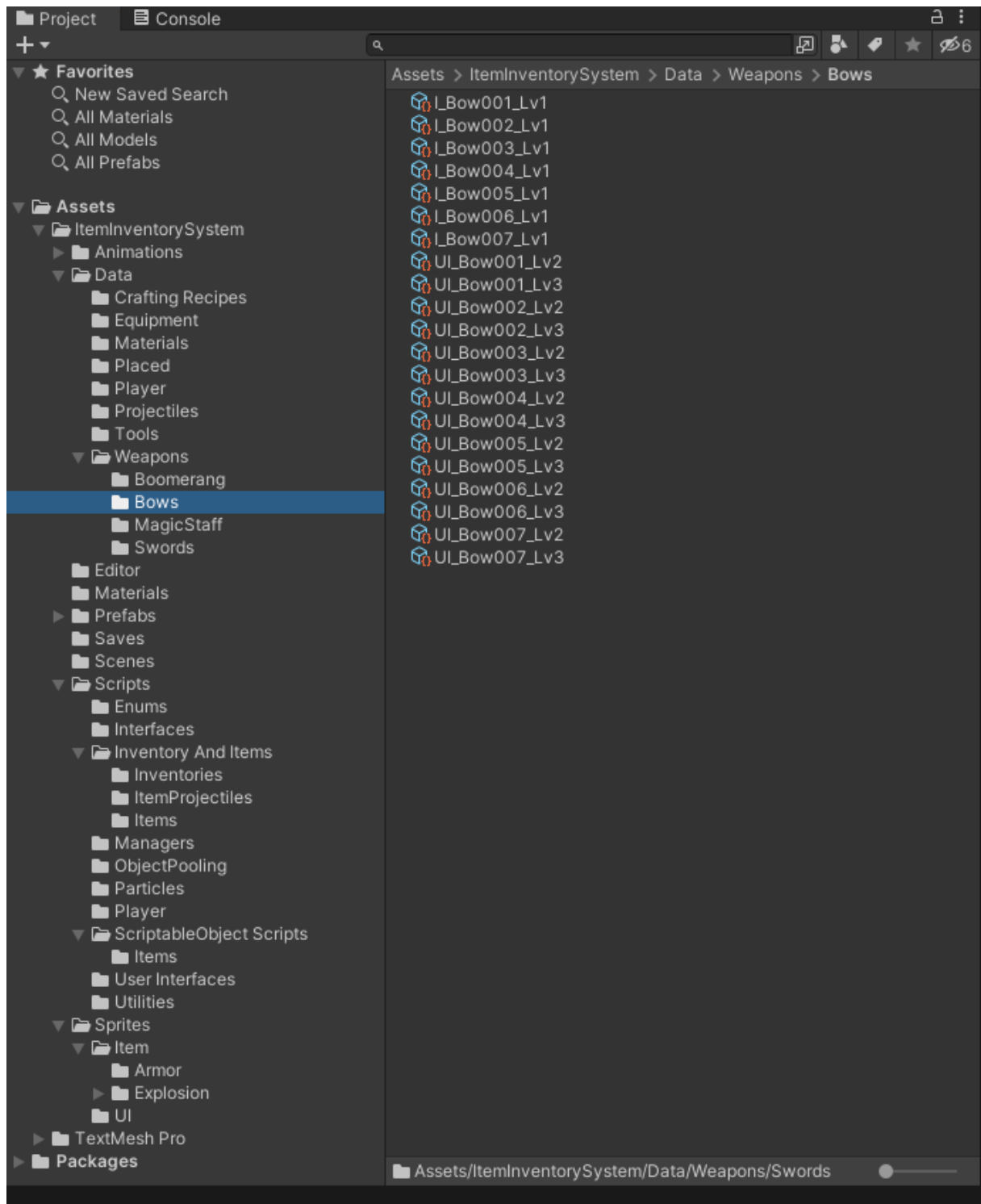


Image 4.5: Item data (Bow data)

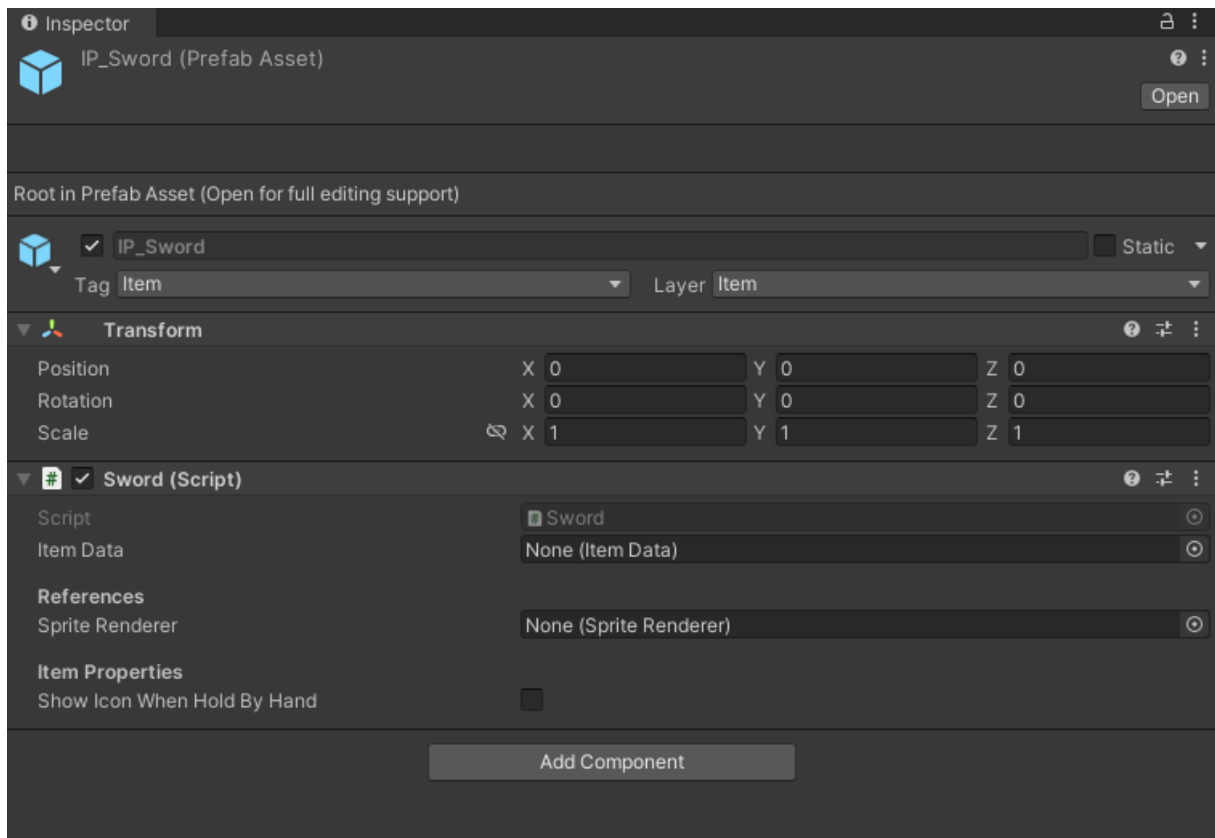


Image 4.6: Item prefab (Sword prefab)

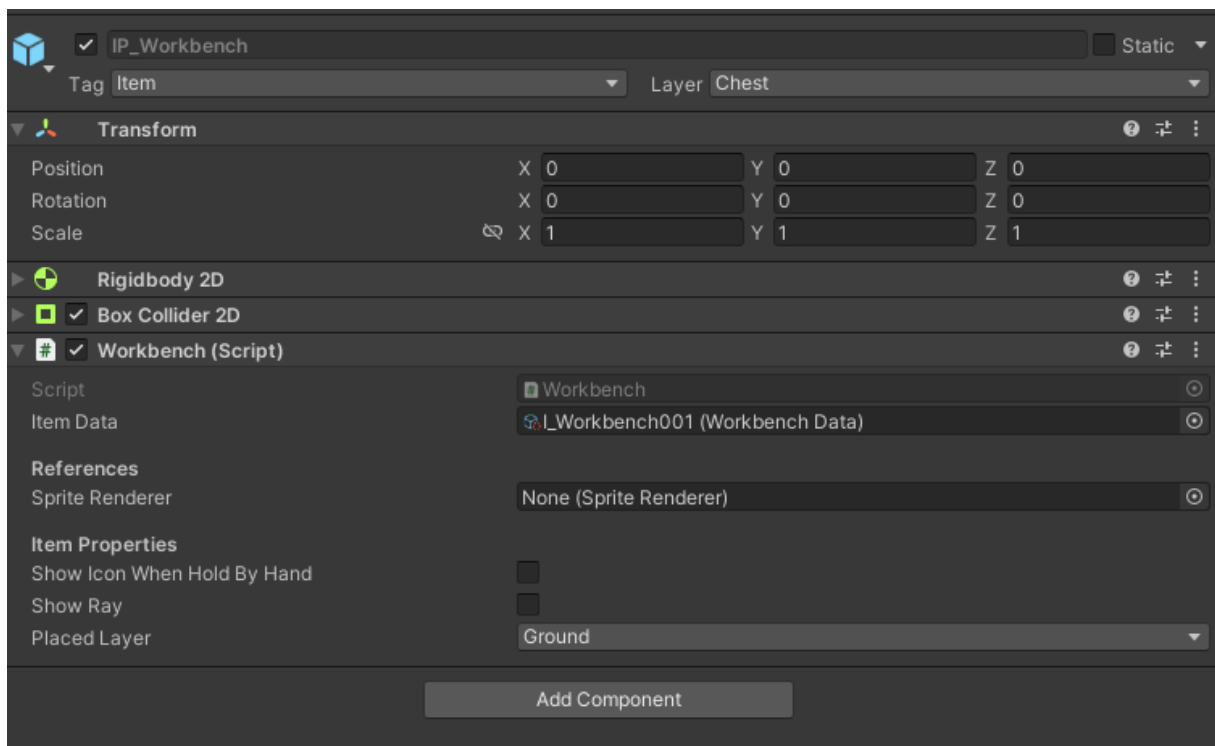


Image 4.7: Item prefab (Workbench prefab)

4.3 How to add already exist item?

- All existing items I've created can be found in ItemDataManager under Item Prefabs.

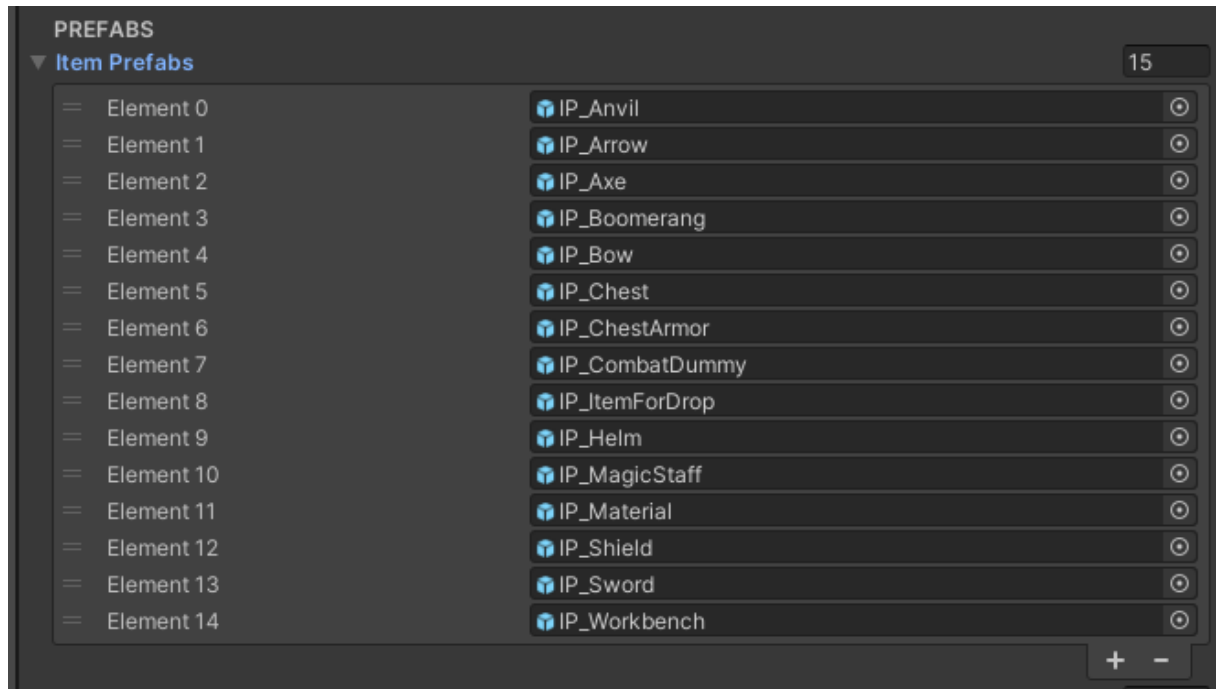


Image 4.8: All item prefabs in GameDataManager.cs

- Example: If you want to make a new sword item, you need to follow the following steps:

+ Step 1: Create new sword data in *Assets>Create>UltimateItemSystem>Item>Weapons>Sword*.

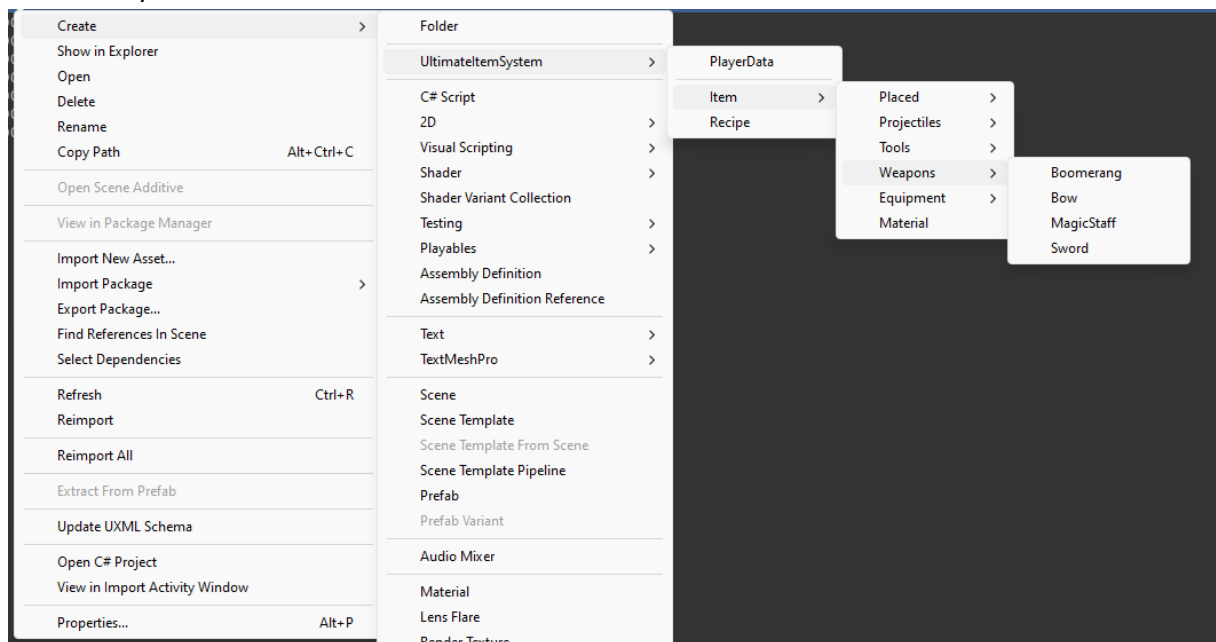


Image 4.9: Sword data in UltimateItemSystem

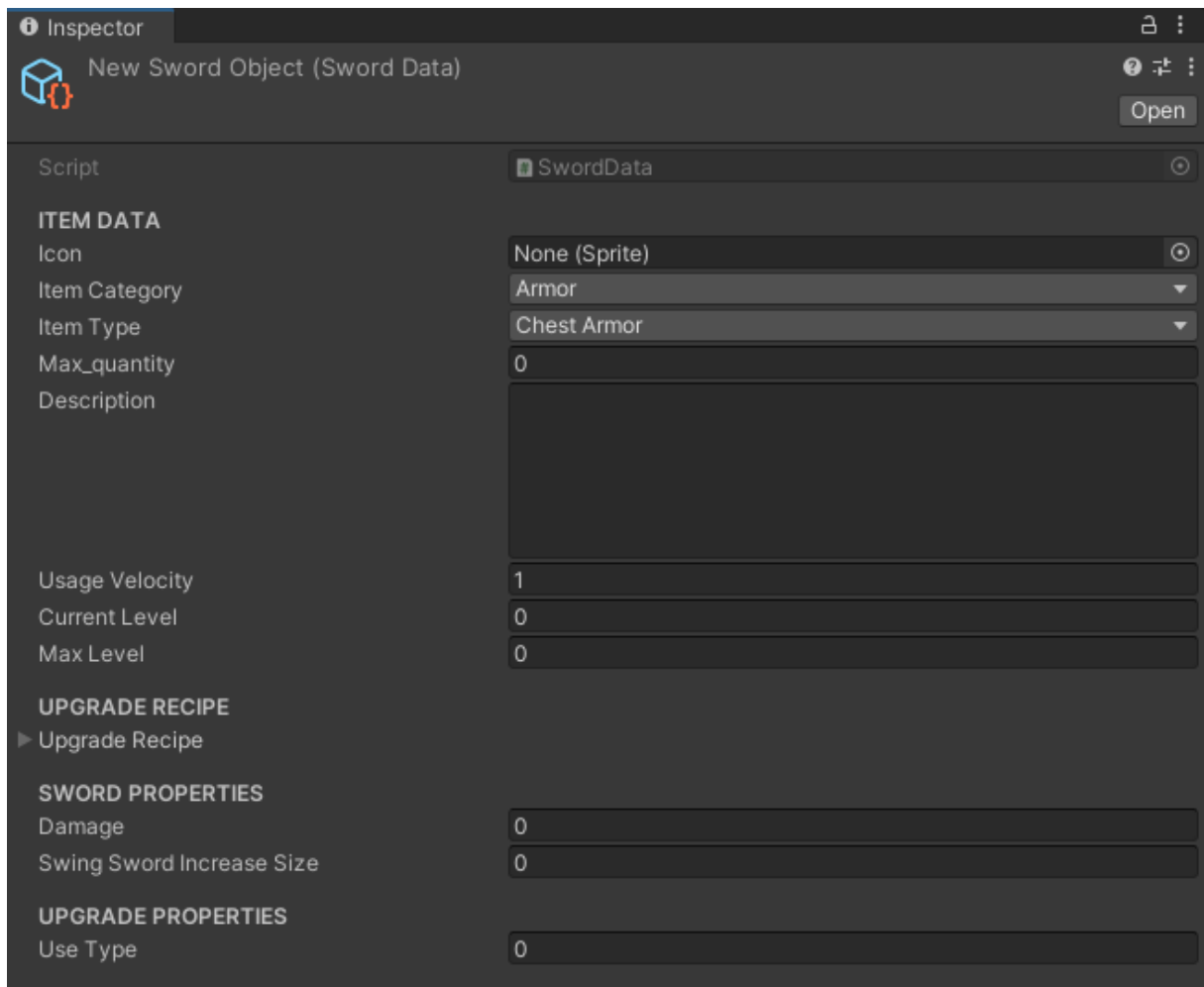


Image 4.10: Unity Inspector view of default sword data.

+ Step 2: Rename sword data with following format of item data: I_*. (Ex: I_Sword001_Lv1)

*** NOTE:** This is just a naming convention; if you don't name your files this way, it won't cause an error; you don't have to do what I do. However, it will make it easier for you to collect all data references later.

+ Step 3: Play with these properties (Based on the existing sword data, you only need to change the sprite of this).

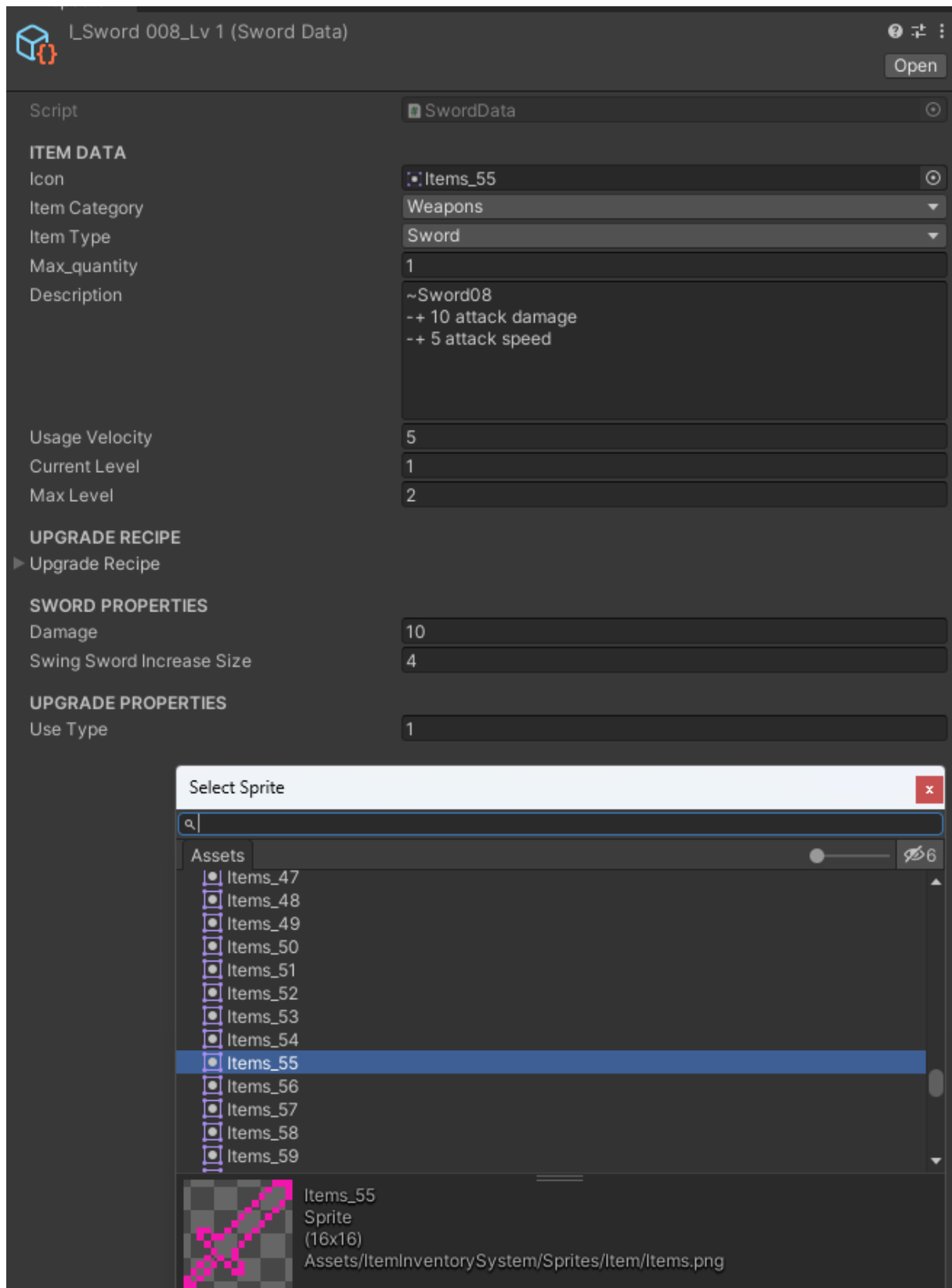


Image 4.11: Unity Inspector view after filling sword data.

+ Step 4: Drag this sword data into GameDataManager.

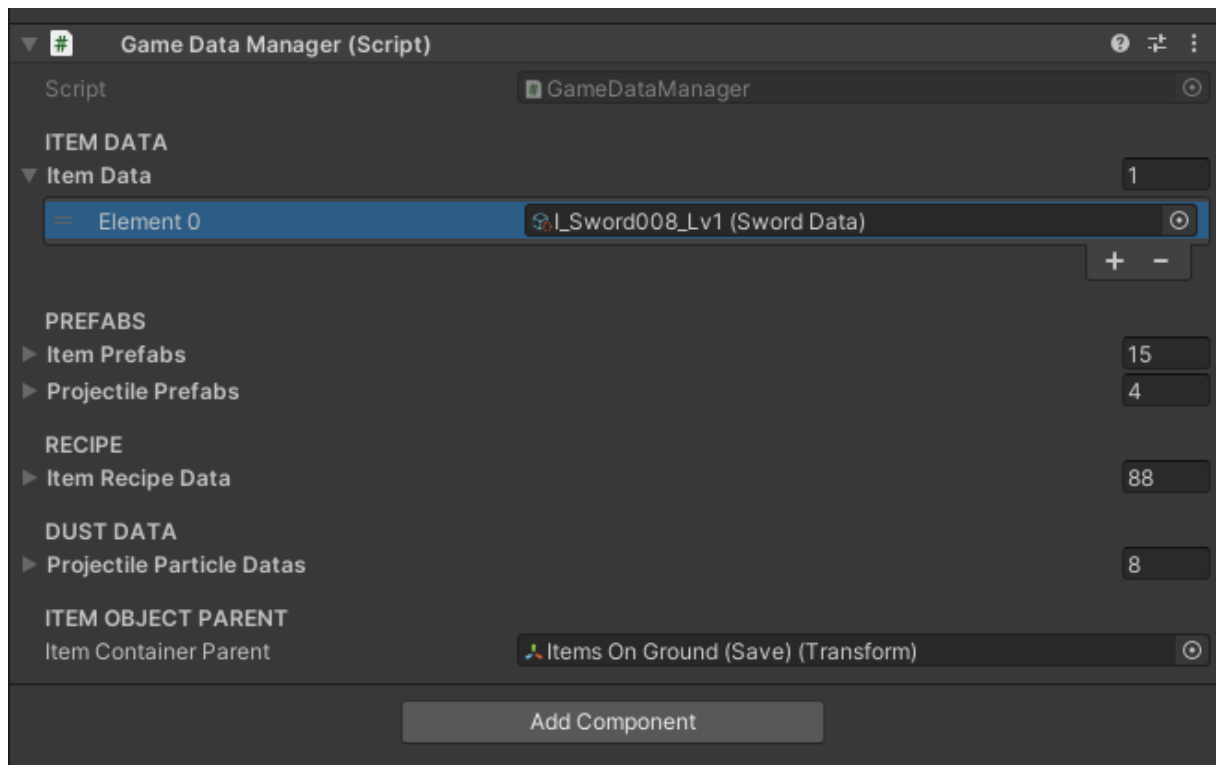


Image 4.12: Add item data to GameDataManager.cs in Unity Inspector

+ Step 5: Finish. When you press play, you will see the item exist in creative inventory.

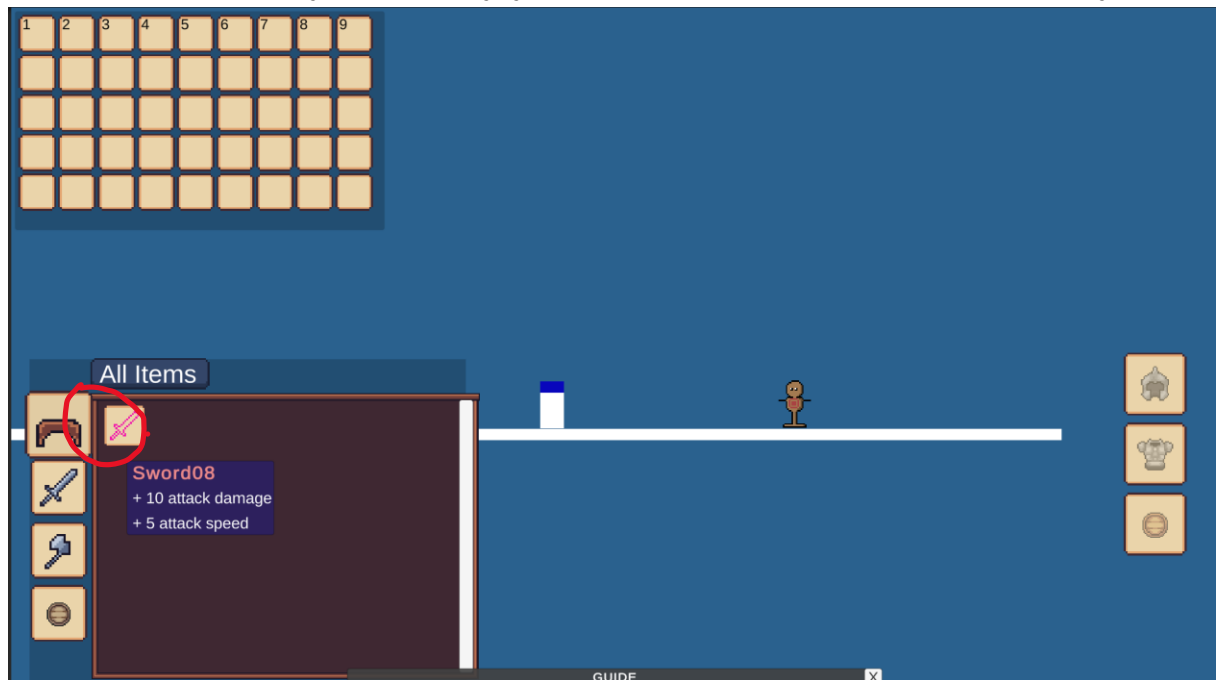


Image 4.13: Sword item data in creative inventory.

* **HINT:** To grab all item data into *GameDataManager* only need enter **!_*** in search bar like this:

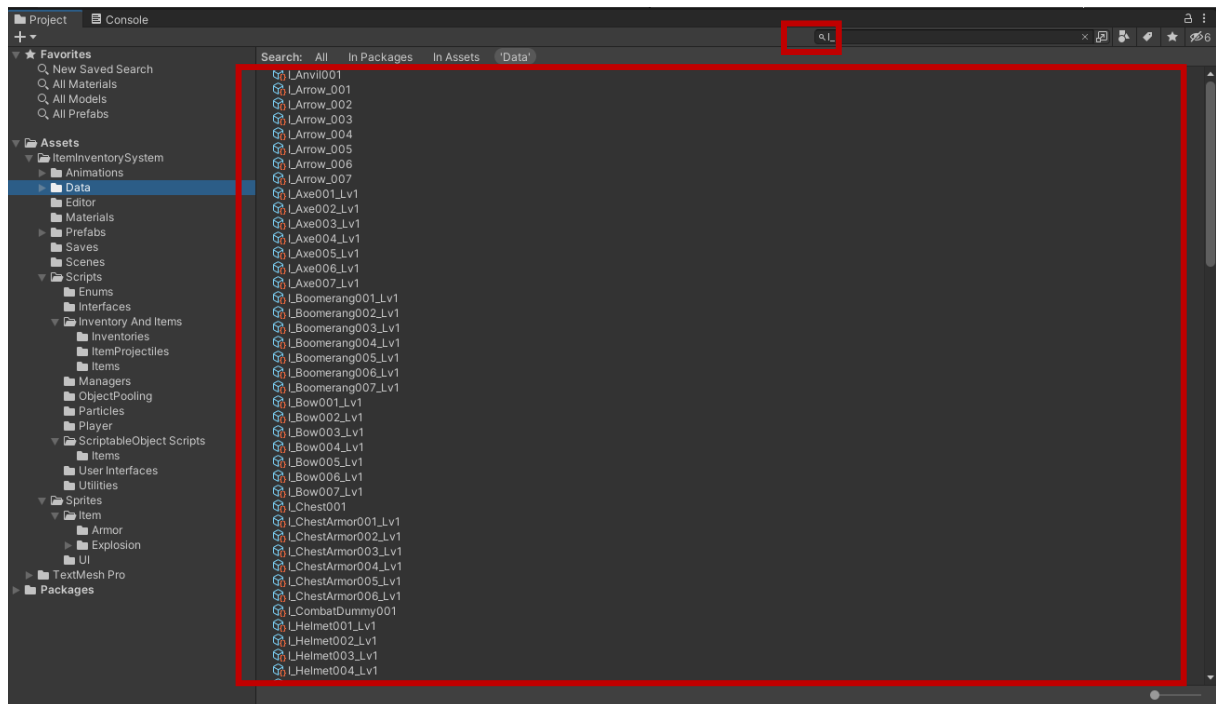


Image 4.14: Sword item data in creative inventory.

- Yes, the power of naming conventions.

4.4 How to create a new item type?

- To create a new item type that does not yet exist, three things must be defined:

- 1) Item data derived from *ItemData.cs* (handler data).
- 2) Item derived from *Item.cs* (handler logic).
- 3) Item prefab.

- After you have defined three of them, drag this into *GameDataManager.cs* in the Unity Inspector.



Image 4.15: Item data and item prefab container in *GameDataManager.cs*

* Example: Create a speed booster potion.

- Step 1: Find *ItemType.cs* enum add ItemType name **SpeedBoosterPotion**

```
20 references
public enum ItemType
{
    // Armor
    ChestArmor,
    Helm,
    Shield,
    Shoes,

    // Weapons
    Bow,
    Sword,
    Axe,
    Hammer,
    Boomerang,
    MagicStaff,

    // Ammunition
    Arrow,
    Bullet,

    // Consumables
    Buff,
    Consumption,

    // Crafting materials
    Material,

    // Furniture
    Chest,
    Workbench,
    Anvil,

    // Miscellaneous
    CombatDummy,
    Currency,
    LightSource,
    Null,

    SpeedBoosterPotion
}
```

Image 4.16: Item type enum

- Step 2: At *Assets>UltimateItemSystem>Scripts>ScriptableObject Scripts>Items*

Create new script name **SpeedBoosterPotionData.cs** derived from **ItemData.cs** like this:

```
[CreateAssetMenu(fileName = "SpeedBoosterPotion", menuName = "UltimateItemSystem/Item/Consumable/SpeedBoosterPotion", order = 51)]
public class SpeedBoosterPotionData : ItemData
{
    [Header("SpeedBoosterPotion properties")]
    public int boosterSpeed;
}
```

Image 4.17: SpeedBoosterPotionData.cs

- Step 3: Create new potion data and rename it with naming convention (**I_***) like the sword data above.

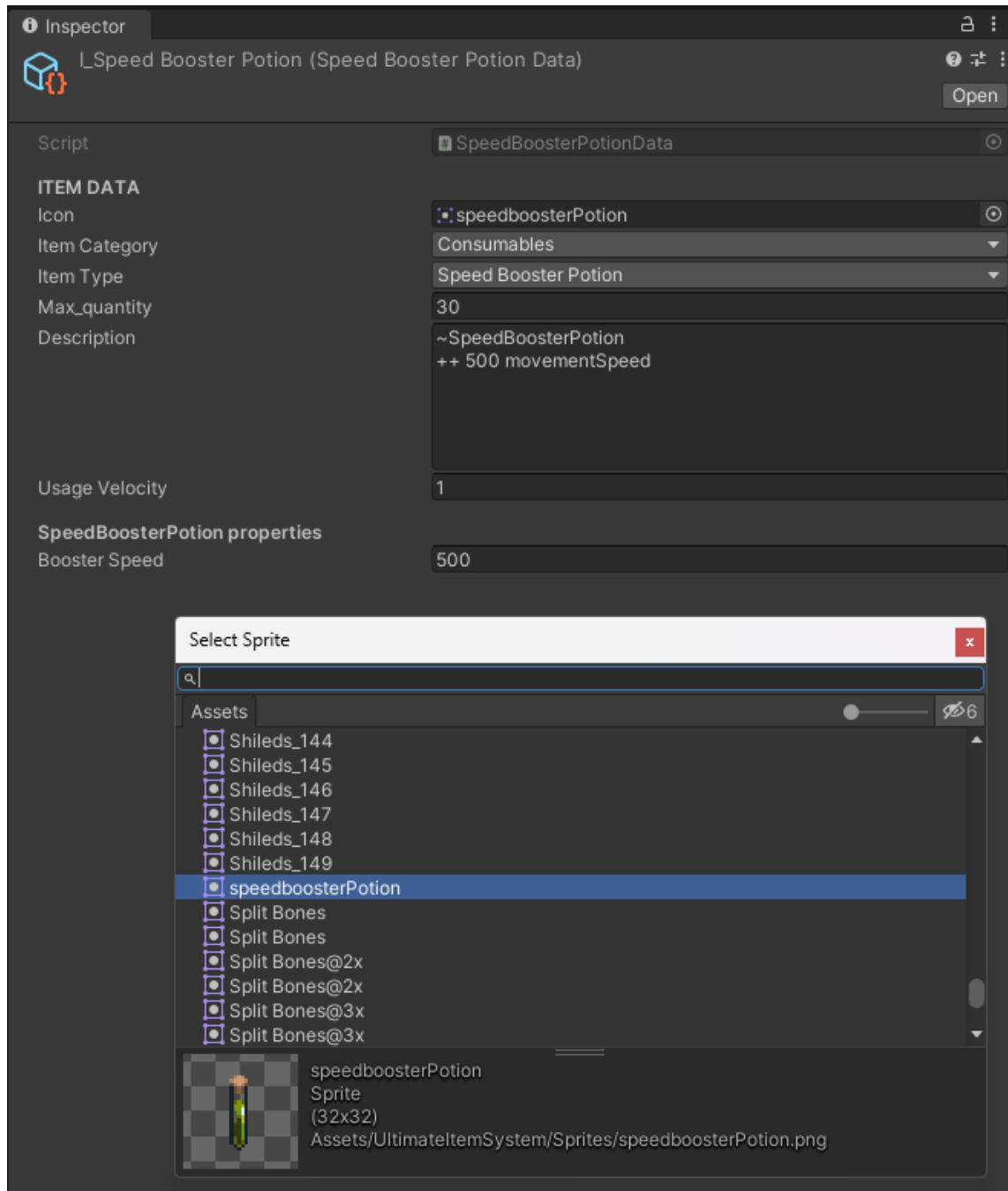
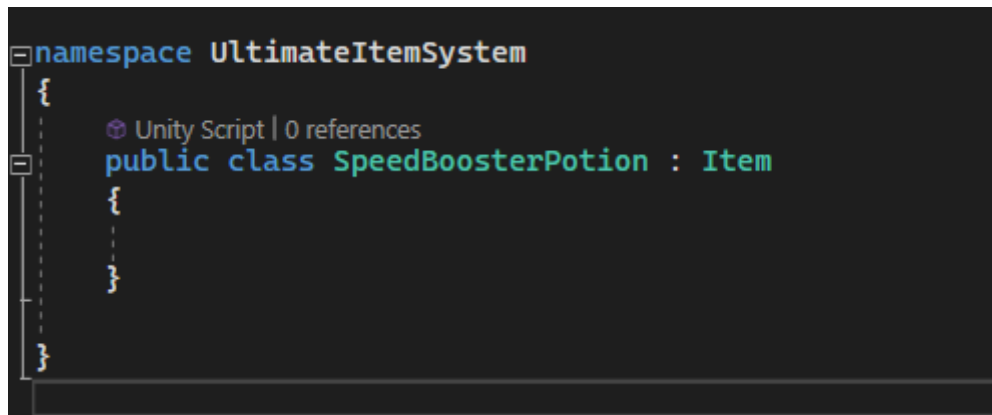


Image 4.18: SpeedBoosterPotionData.cs in Unity Inspector view.

- Step 4: Add this item data into **GameDataManager.cs** in Unity Inspector.
- Step 5: At *Assets>UltimateItemSystem>Scripts>Inventory and Items>Items* - Create **SpeedBoosterPotion.cs** (The name must be the same as the item type) derived from **Item.cs** for handler logic of the item.



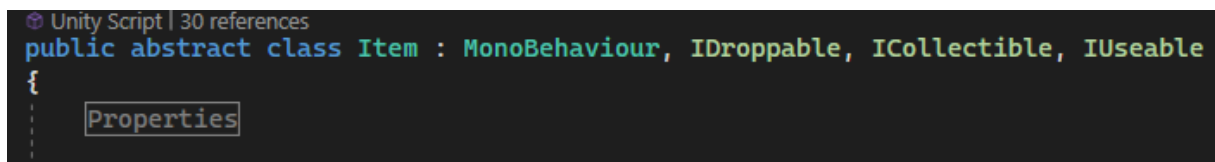
```

namespace UltimateItemSystem
{
    [Unity Script | 0 references]
    public class SpeedBoosterPotion : Item
    {
        //
    }
}

```

Image 4.19: SpeedBoosterPotion.cs

- If you go to the definition of Item.cs, you will see that it is derived from some interfaces like **IDroppable**, **ICollectible**, and **IUseable**. So, because we want to use this potion to increase player movement speed we only need to override the implementation of **IUseable**.

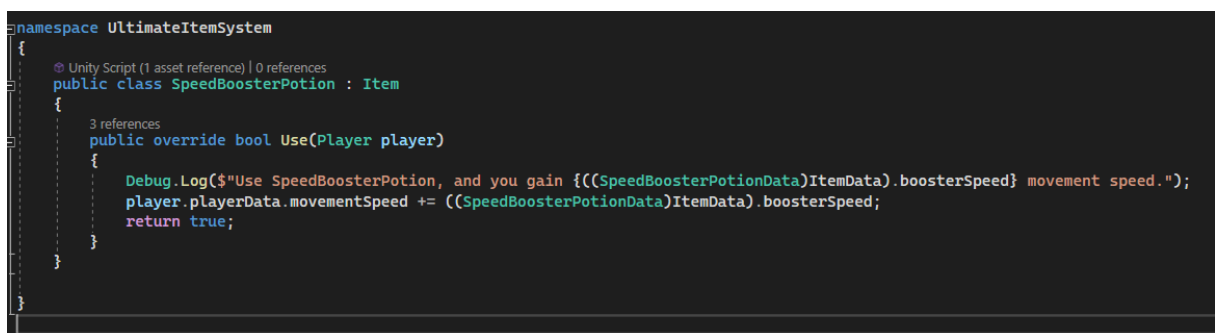


```

[Unity Script | 30 references]
public abstract class Item : MonoBehaviour, IDroppable, ICollectible, IUseable
{
    //
}

```

Image 4.20: Definition of base class Item.cs



```

namespace UltimateItemSystem
{
    [Unity Script (1 asset reference) | 0 references]
    public class SpeedBoosterPotion : Item
    {
        [3 references]
        public override bool Use(Player player)
        {
            Debug.Log($"Use SpeedBoosterPotion, and you gain {(SpeedBoosterPotionData)ItemData}.boosterSpeed movement speed.");
            player.playerData.movementSpeed += ((SpeedBoosterPotionData)ItemData).boosterSpeed;
            return true;
        }
    }
}

```

Image 4.20: SpeedBoosterPotion.cs implements an IUseable interface.

- Step 6: After you've created the item class, you'll need to make a prefab to represent it in the game world. Go to *Assets>UltimateItemSystem>Prefabs>Items>* - Make a new item prefab by cloning an existing one (for example, **PP_Sword**) and renaming it **IP_SpeedBoosterPotion** (*IP - Item prefab*). Replace **Sword.cs** with **SpeedBoosterPotion.cs**.

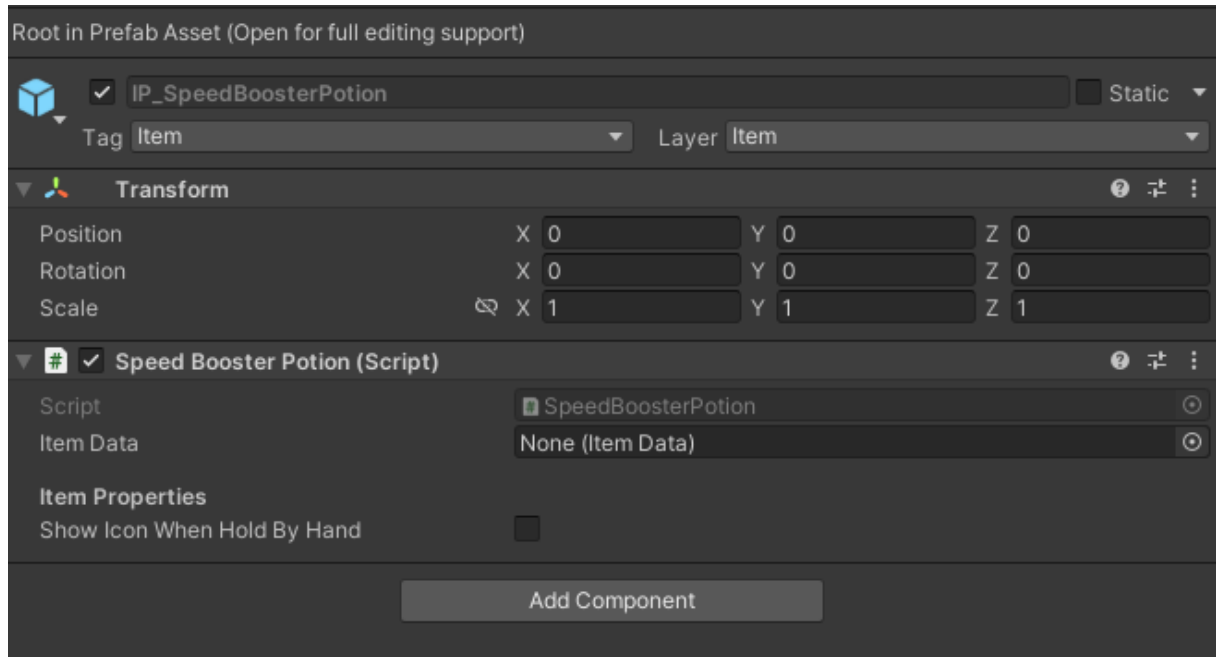


Image 4.21: SpeedBoosterPotion prefab in the Unity Inspector view.

- Then you drag this item prefab into **GameDataManager.cs** -> ItemPrefab in the Unity Inspector.

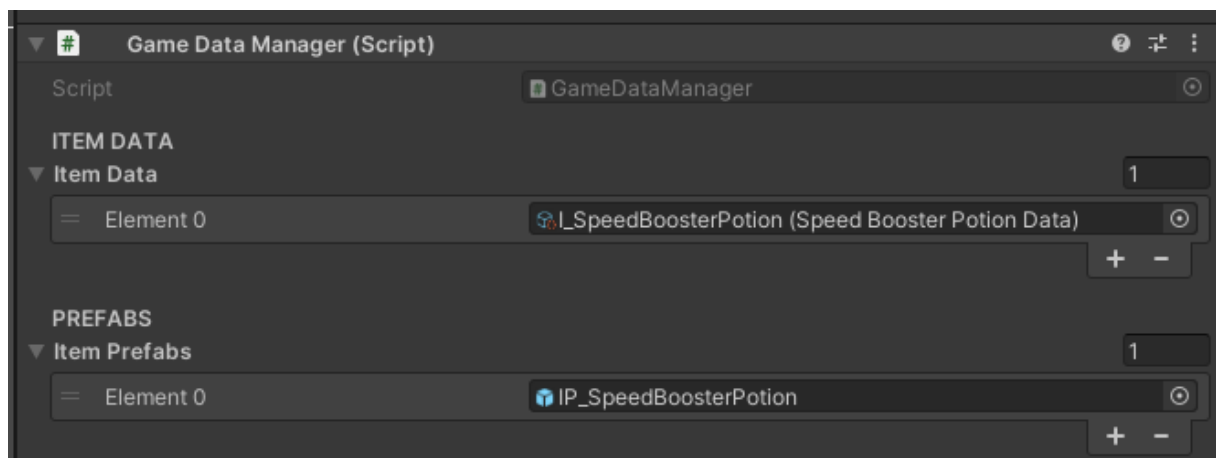


Image 4.22: Fill Item data and item prefab in GameDataManager.cs.

- Step 7: Enter play mode now; when you use this potion, you will receive a message and move around to feel your movement speed improve.

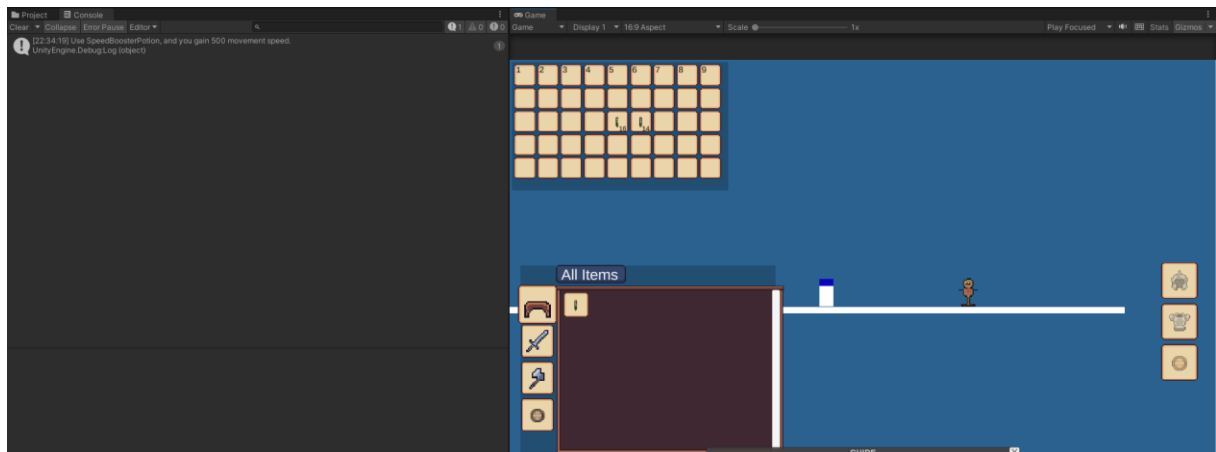


Image 4.23: SpeedBoosterPotion already appears in play mode.

***NOTE:** Many things still need to be improved, such as removing items after use, resetting player data, and so on, but I will not cover this. Try to find what works best for you.

5. Inventory

- Inventory is a fundamental system in the game that allows players to store and manage items. There are three types of inventories: *Player Inventory*, *Chest Inventory*, and *Creative Inventory*. Because the mechanics of these inventories are fixed, I already added a lot of comments to the code. You don't need to know about these, but if you want to learn more about how codes work, the structure of the inventory in this project is listed below.

5.1 Inventory structure

- Below are some classes for handling inventory in this package:

- + **ItemSlot.cs**: Keep track of item data and quantity.
- + **UllItemSlot.cs**: Display item slot
- + **PlayerInventory.cs**: Hold list of item slots, has some methods like add, remove item, ...
- + **UIPlayerInventory.cs**: Display inventory, handle logic such as click, drag, and so on...
- + **ChestInventory.cs**: Hold list of item slots, has some methods like add, remove item, ...
- + **UIChestInventory.cs**: Display inventory, handle logic such as click, drag, and so on...

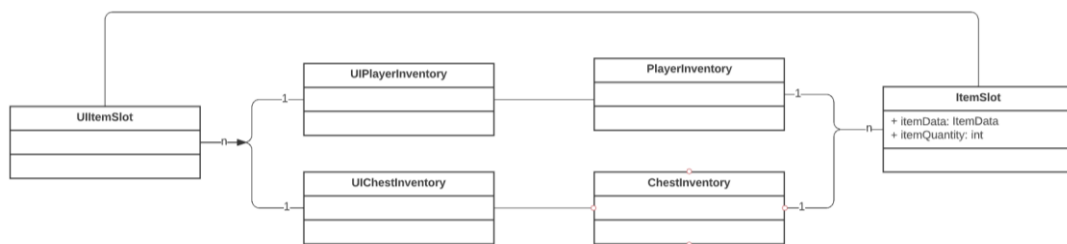


Image 5.1: Class diagram of inventory

- In detail, these classes are responsible for:

+ **ItemSlot.cs**: This is a visual representation of an inventory item, which contains the item's icon, name, and quantity.

+ **UllItemSlot.cs**: The UllItemSlot class is responsible for managing the UI of the item slot. It displays the item's sprite and provides the player with the ability to interact with it. It interacts with other components like **UICreativeInventory.cs** and **ItemSlot.cs** to manage the UI of the inventory.

+ **PlayerInventory.cs** is a script that manages the player's inventory. It is responsible for keeping track of the items that the player has collected and the quantity of each item. It contains methods for adding and removing items from the inventory, checking if the inventory contains a specific item.

+ **UIPlayerInventory.cs** is a script responsible for handle the user interface for the player inventory system. This class contains some method like `UpdateInventory`, and method for do with player interactive like `OnPointerDown`, `OnLeftClick`, `OnDrag`, ...

- The structure and logic of both *player inventory* and *chest inventory* are the same, so I'll just explain **PlayerInventory.cs** and **UIPlayerInventory.cs**.

5.2 Player inventory code explanation

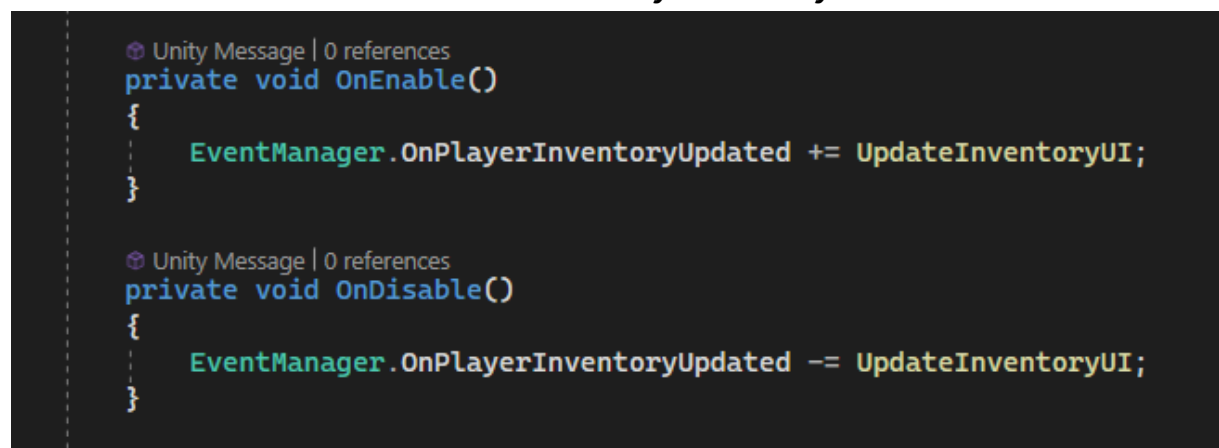
a) **PlayerInventory.cs**

- The **PlayerInventory.cs** class manages the player's inventory. It contains methods to add, remove, and get items from the inventory. The inventory is represented as a list of *ItemSlot* objects, where each slot can hold one item. The class also includes constants for the size of the inventory, and references to other components in the game (details are in the source code comments).

b) **UIPlayerInventory.cs**

- The **UIPlayerInventory.cs** class is responsible for managing the UI elements of the player's inventory. This is also in charge of handler logic when players interact with the UI, such as drag, click, etc.

- Below are some of the methods handled in **UIPlayerInventory.cs**



```
⊕ Unity Message | 0 references
private void OnEnable()
{
    EventManager.OnPlayerInventoryUpdated += UpdateInventoryUI;
}

⊕ Unity Message | 0 references
private void OnDisable()
{
    EventManager.OnPlayerInventoryUpdated -= UpdateInventoryUI;
}
```

Image 5.2: An event is triggered when player inventory changes, which will update this UI.

```

for (int i = 0; i < playerInventory.Capacity; i++)
{
    GameObject slotObject;

    if(i < playerInventory.WidthSize)
    {
        slotObject = Instantiate(numericItemSlotPrefab, this.transform);
        slotObject.GetComponent<UINumericItemSlot>().slotIndexText.text = $"{i + 1}";
    }
    else
        slotObject = Instantiate(itemSlotPrefab, this.transform);

    slotObject.GetComponent<UIItemSlot>().SetIndex(i);
    slotObject.GetComponent<UIItemSlot>().SetData(null);
    Utilities.AddEvent(slotObject, EventTriggerType.PointerClick, (baseEvent) => OnClick(baseEvent, slotObject));
    Utilities.AddEvent(slotObject, EventTriggerType.PointerEnter, delegate { OnEnter(slotObject); });
    Utilities.AddEvent(slotObject, EventTriggerType.PointerExit, delegate { OnExit(slotObject); });
    Utilities.AddEvent(slotObject, EventTriggerType.BeginDrag, (baseEvent) => OnBeginDrag(baseEvent, slotObject));
    Utilities.AddEvent(slotObject, EventTriggerType.EndDrag, (baseEvent) => OnEndDrag(baseEvent, slotObject));

    itemSlotList.Add(slotObject);
}

```

↖ UI interactive event

Image 5.3: Events added to UI game objects.

- To display inventory, this code simply instantiates *UIItemSlot*. Each *UIItemSlot* will have an event such as *OnClick*, *OnEnter*, *OnBeginDrag*, etc. to handle logic when the player clicks or drags UI elements.

```

2 references
private void OnLeftClick(int index)
{
    handHasItem = itemInHand.HasItemData();
    slotHasItem = playerInventory.inventory[index].HasItem();

    if (handHasItem == false)
    {
        if (slotHasItem == false)
        {
            //Debug.Log("HAND: EMPTY \t SLOT: EMPTY");
        }
        else
        {
            //Debug.Log("HAND: EMPTY \t SLOT: HAS ITEM");
            itemInHand.Swap(ref playerInventory.inventory, index, StoredType.PlayerInventory, true);
        }
    }
    else
    {
        if (slotHasItem == false)
        {
            //Debug.Log("HAND: HAS ITEM \t SLOT: EMPTY");
            itemInHand.Swap(ref playerInventory.inventory, index, StoredType.PlayerInventory, true);
        }
        else
        {
            //Debug.Log("HAND: HAS ITEM \t SLOT: HAS ITEM");
            bool isSameItem = ItemData.IsSameItem(playerInventory.inventory[index].ItemData, itemInHand.GetItemData());
            if (isSameItem)
            {
                ItemSlot remainItems = playerInventory.inventory[index].AddItemsFromAnotherSlot(itemInHand.GetSlot());
                itemInHand.SetItem(remainItems, index, StoredType.PlayerInventory, true);
            }
            else
            {
                itemInHand.Swap(ref playerInventory.inventory, index, StoredType.PlayerInventory, true);
            }
        }
    }
}

```

Image 5.4: Inner logic code for the left-click event.

- The code above describes the logic that occurs when a player left clicks in a player inventory slot. It contains **four** parts:

- 1) When the player does not hold an item and the slot clicked is empty.
- 2) When the player does not hold an item and the slot clicked has items.
- 3) When the player holds an item and the slot clicked is empty.
- 4) When the player holds an item and the slot clicked has items.

- The logic of each case depends on what you need. You can change it as you see fit, but it may take some time to understand. The other event follows a similar logic: experiment on yourself to find out what works best for you.

6. Crafting

6.1 Crafting structure

- You can find craft logic handles and craft UI handles at:

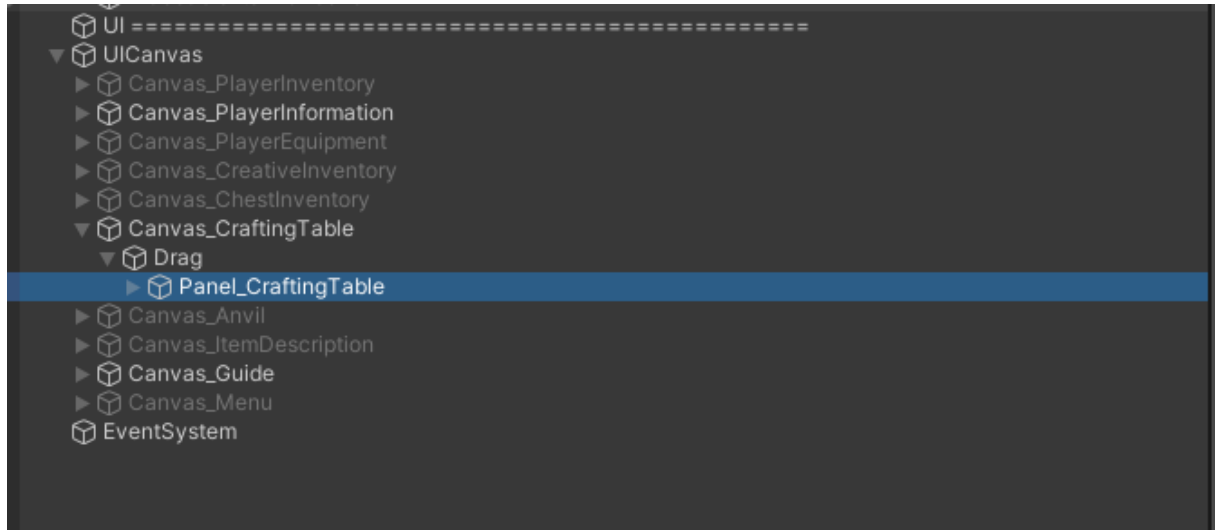


Image 6.1: GameObject component handling crafting mechanics.

- It contains two scripts: **CraftingTable.cs** and **UICraftingTable.cs**. Like player inventory, CraftingTable contains data and methods used by **UICraftingTable.cs** when players click, drag, etc. on UI elements.

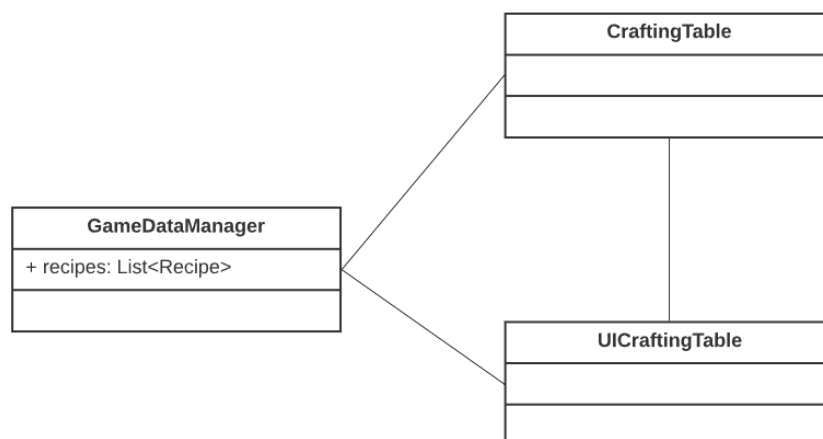


Image 6.2: Class diagram of crafting mechanics.

- The logic behind crafting mechanics is that when you add an item to the crafting table, it will show up in the UI, create a recipe instance, and be looked up in the recipes dictionary of data in **GameDataManager**. If find the correct recipe, the output item will appear and you will be able to obtain it; otherwise, nothing will happen.

- Some features:

- + Crafting table items can be used to open and close crafting mechanics.
- + You can move the UI by right-clicking on the gray border.
- + When the crafting table is open, you can find crafting suggestions by right-clicking on the item you want to craft in your creative inventory.

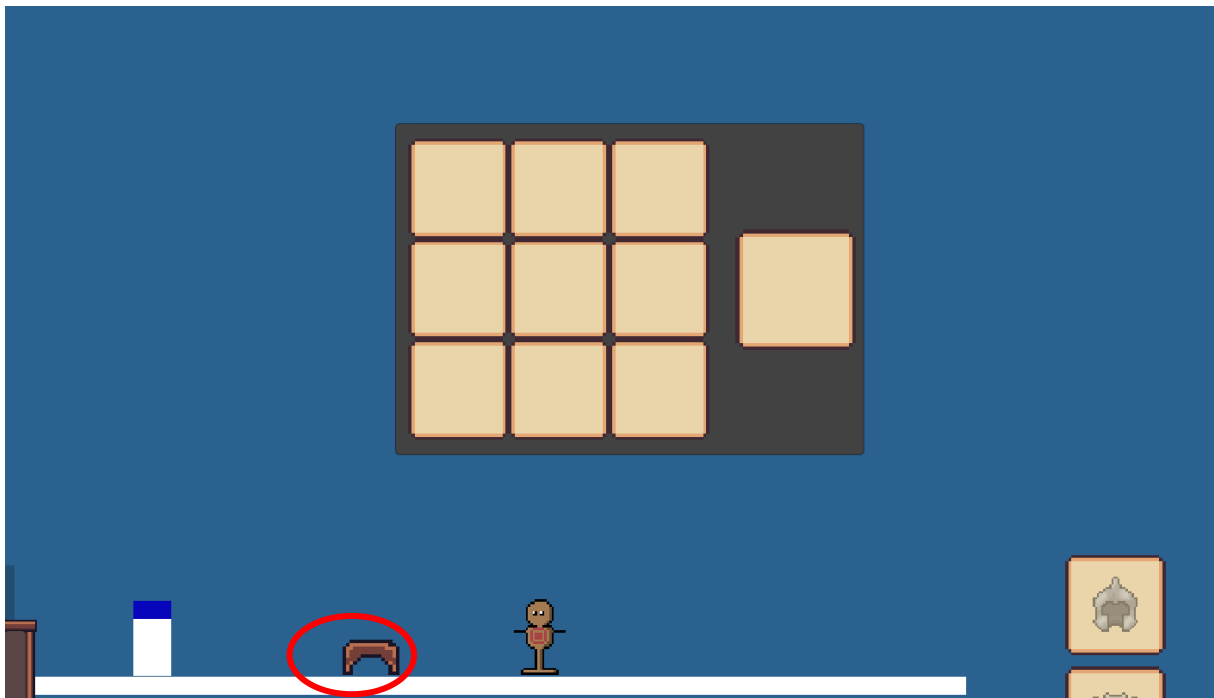


Image 6.3: Crafting table item.



Image 6.3: Suggestions for crafting items.

6.2 How to create a recipe for an item.

- If you already have an item, the Create Item recipe is very simple.
- Step 1: In *the Assets>UltimateItemSystem>Data>CraftingRecipe>* section, - Create a new recipe or copy an existing one and rename it with the naming convention **CR_***.

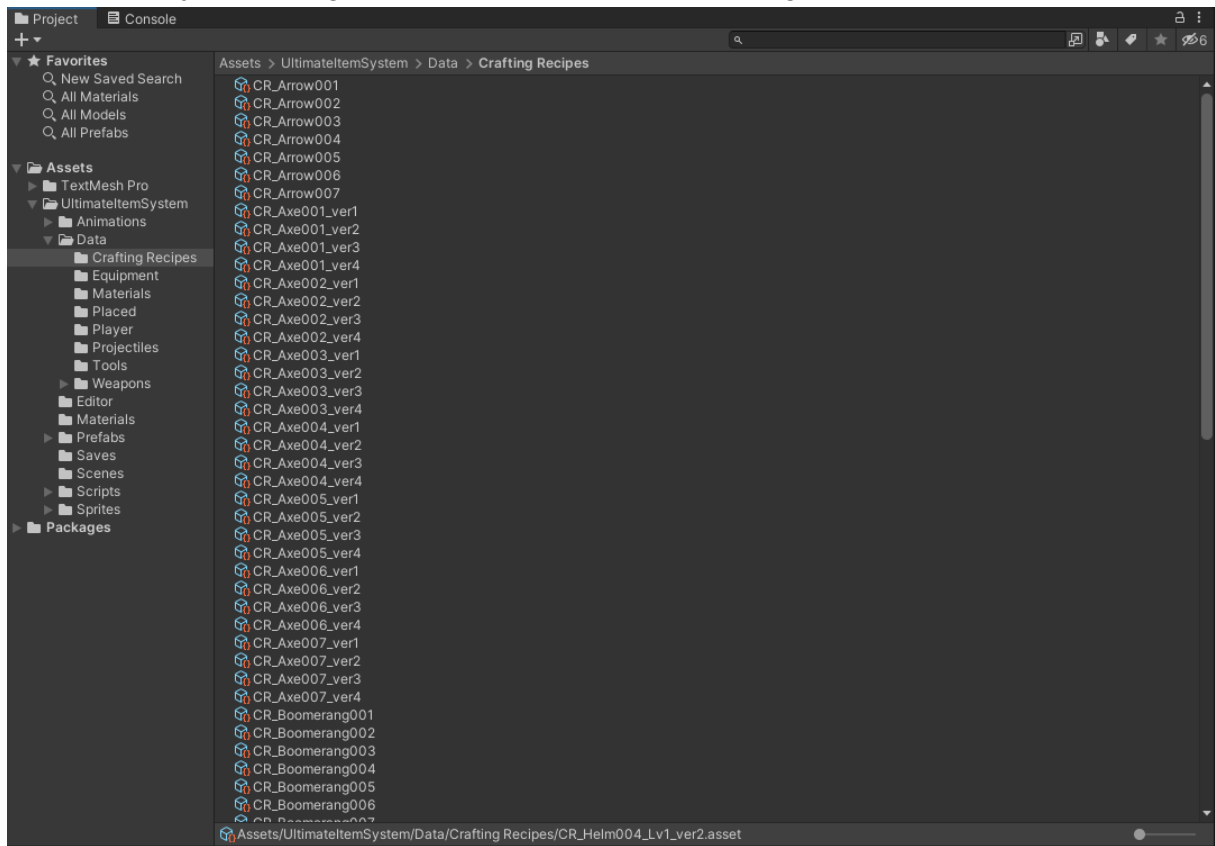


Image 6.4: Crafting recipes data.

- Step 2: Back to SpeedBoosterPotion I'll create this item recipe like this:

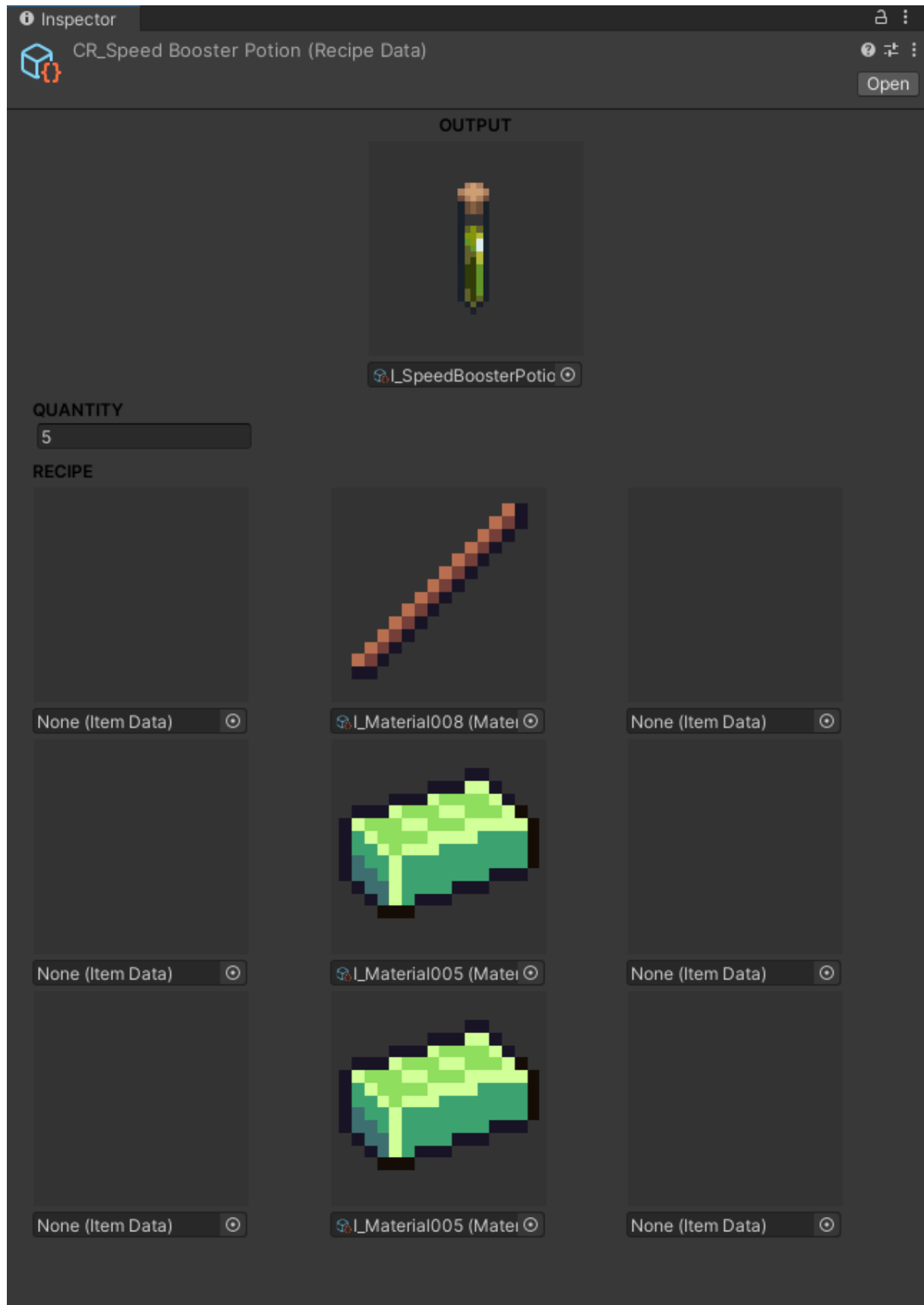


Image 6.5: SpeedBossterPotion recipe.

- Step 3: Drag the SpeedBoosterPotion recipes into the **GameDataManager.cs** recipes.

- Step 4: Finish, enter play mode, and you will see the recipe already applied.

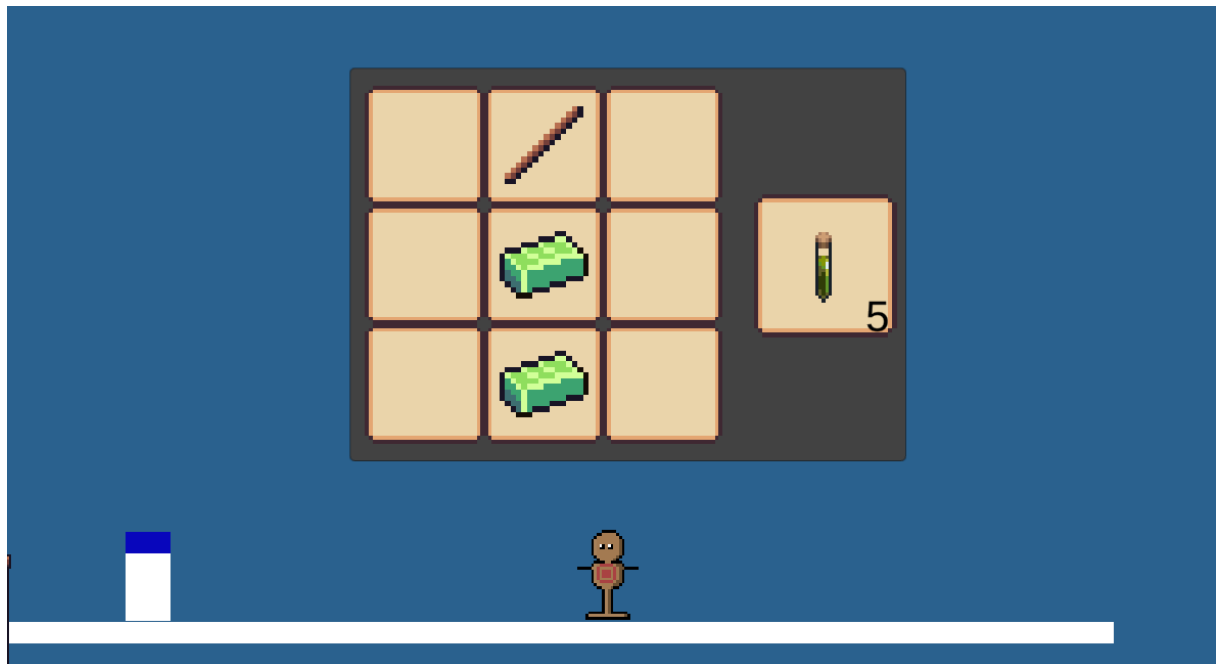


Image 6.6: Craft SpeedBossterPotion in the crafting table.

7. Upgrade items

7.1 Upgrade item structure

- Upgrade items are a mechanic that allows players to improve the quality of their items. Some items, such as a sword, a bow, armor, and so on, can be upgraded in this project. The **Anvil.cs** and **UIAnvil.cs** scripts are responsible for implementing upgrade items through anvil.

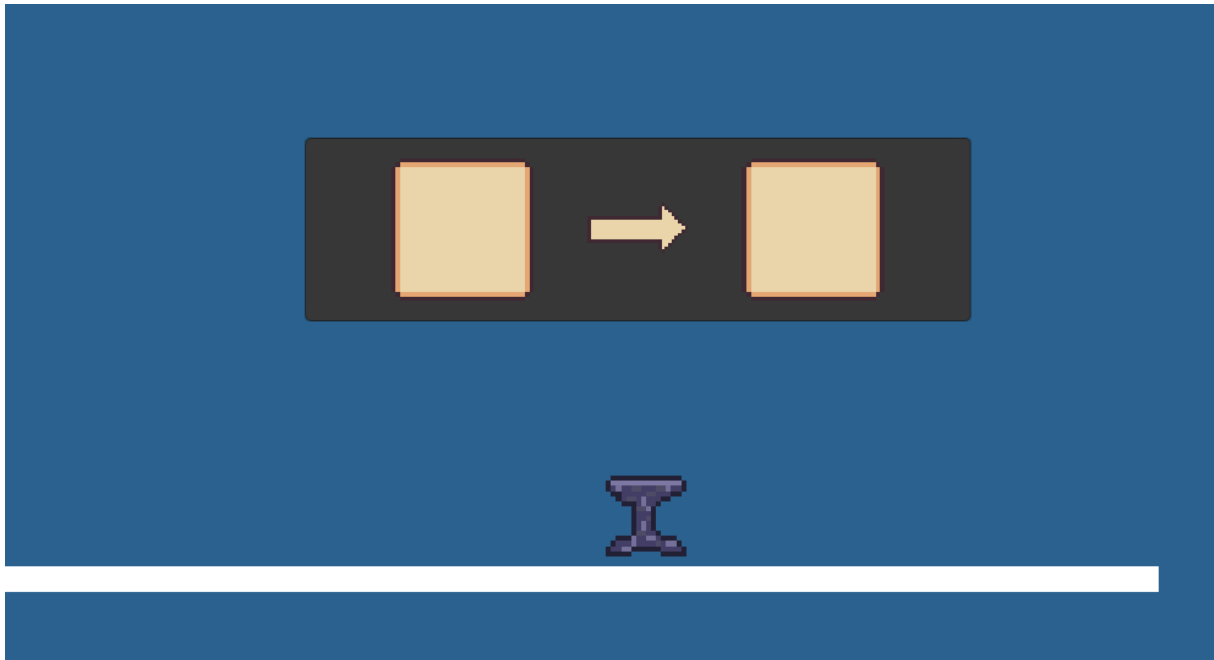


Image 7.1: Anvil item and the default UI of anvil.



Image 7.2: Anvil item when it has an item upgrade.

- When all the materials that require upgrading are complete, you can take the output item.

7.2 How to create an upgradeable item.

- Example: How to create an upgrade for SpeedBoosterPotion item.

- Step 1: First and foremost, your item data must inherit from UpgradeableItemData.cs rather than ItemData.cs. UpgradeableItemData.cs is a ScriptableObject, and it inherits from ItemData.cs but has some properties for upgrading items. So, we need to modify a little bit in SpeedBoosterPotionData.cs scripts.

```
namespace UltimateItemSystem
{
    [CreateAssetMenu(fileName = "SpeedBoosterPotion", menuName = "UltimateItemSystem/Item/Consumable/SpeedBoosterPotion", order = 51)]
    public class SpeedBoosterPotionData : UpgradeableItemData
    {
        [Header("SpeedBoosterPotion properties")]
        public int boosterSpeed;
    }
}
```

Image 7.3: SpeedBoosterPotionData.cs after modification.

- Yes, only change inherit from **ItemData.cs** to **UpgradeableItemData.cs**. When you look at the Unity Inspector, you will notice that there are some additional parts that need to be filled.

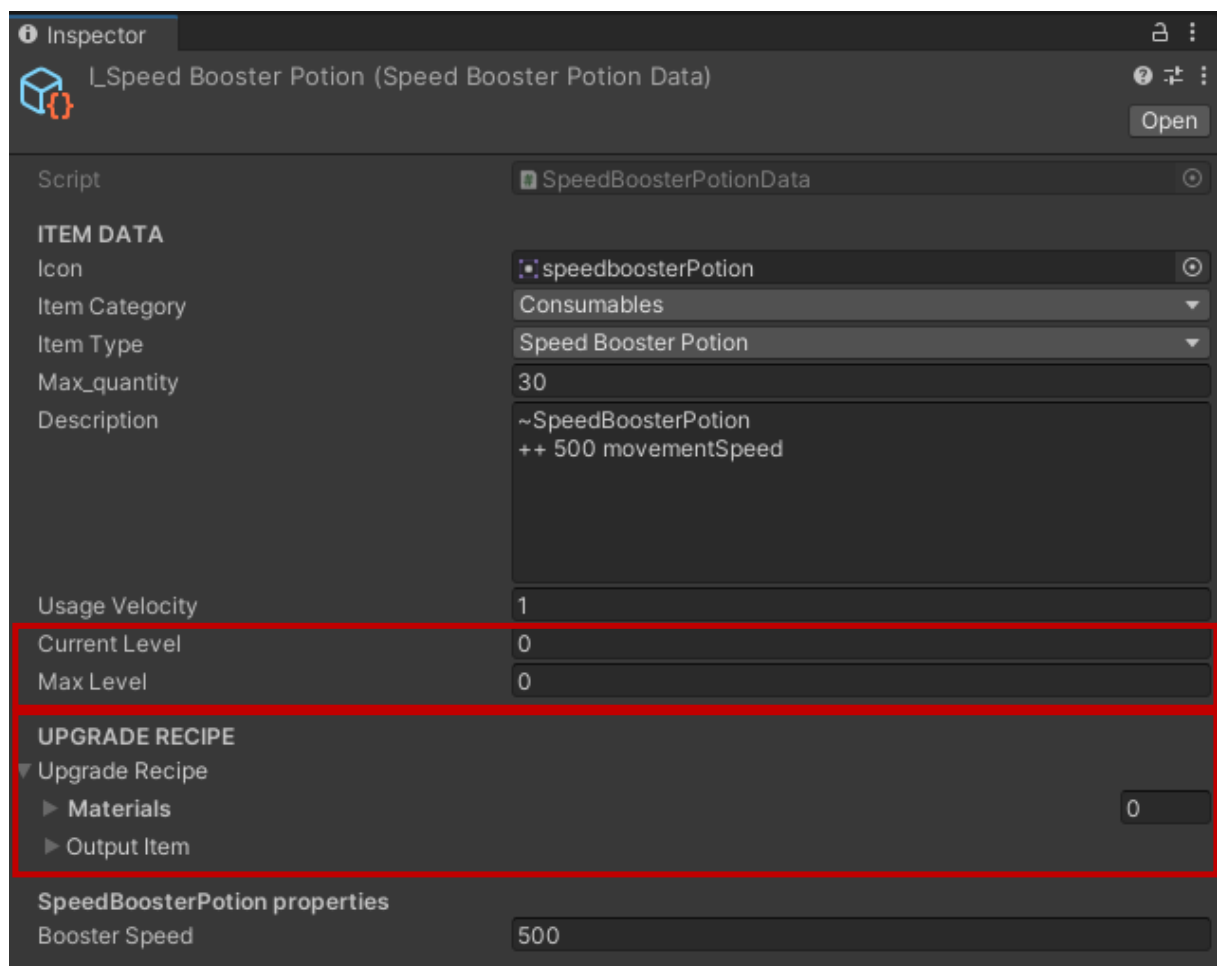


Image 7.4: Unity Inspector view of SpeedBoosterPotionData.cs after modification.

- **Current Level**, and **Max Level** parts are irrelevant. We'll focus on the **UPGRADE RECIPE**.
- As it names materials is list of material need to be upgrade that item, and output item is the output item when materials that require upgrading are complete. (Recommend a maximum of **8** materials slots because the UI only shows **8**, if you want to add more, edit **UIAnvil.cs** and change the maximum number of UI slot elements).

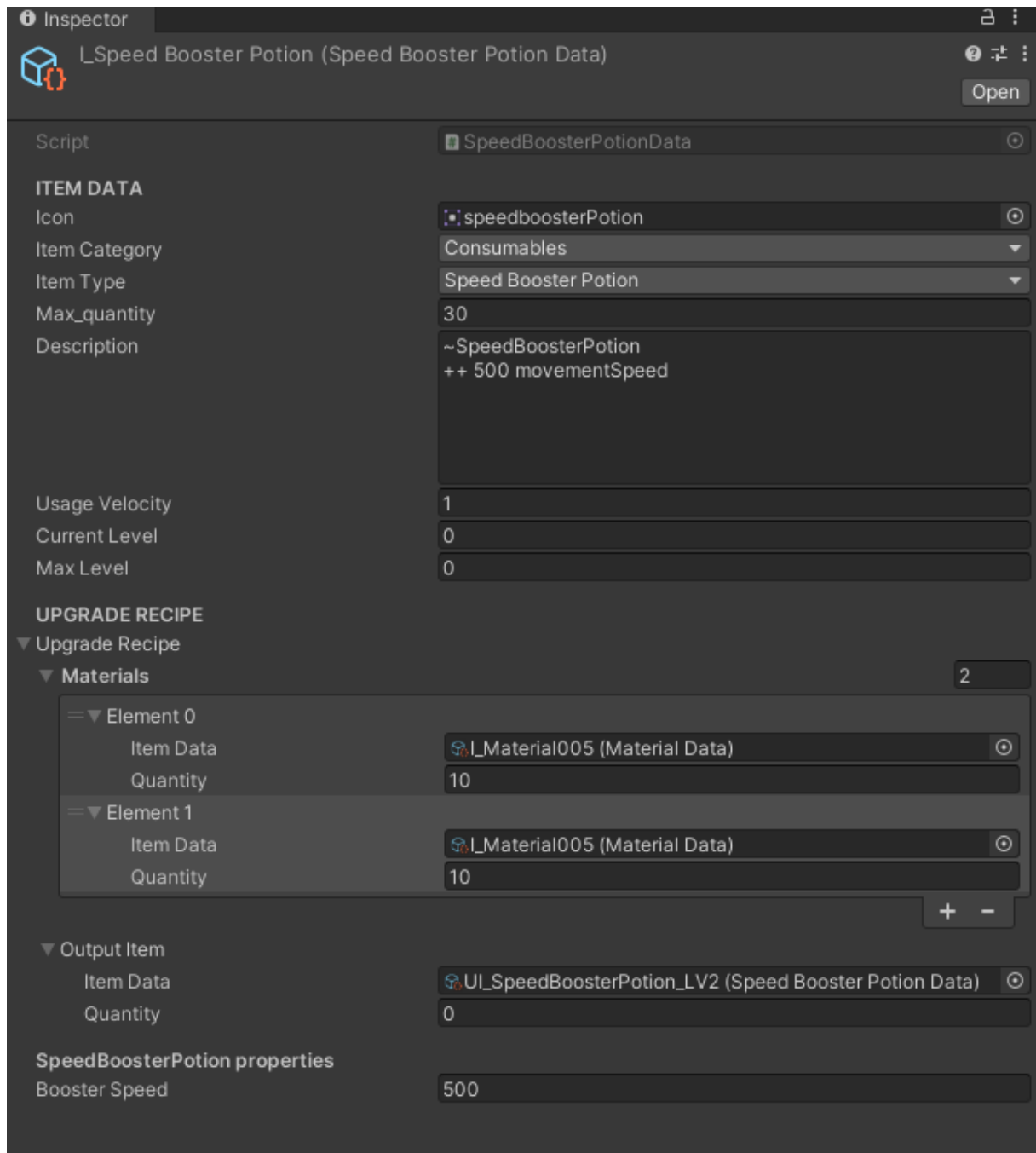


Image 7.5: Unity Inspector view of SpeedBoosterPotionData.cs after modification.

- After fill some data like this. Now we have the upgrade item working properly.
- *NOTE: Item data upgrades must have a naming convention (UI_*).**

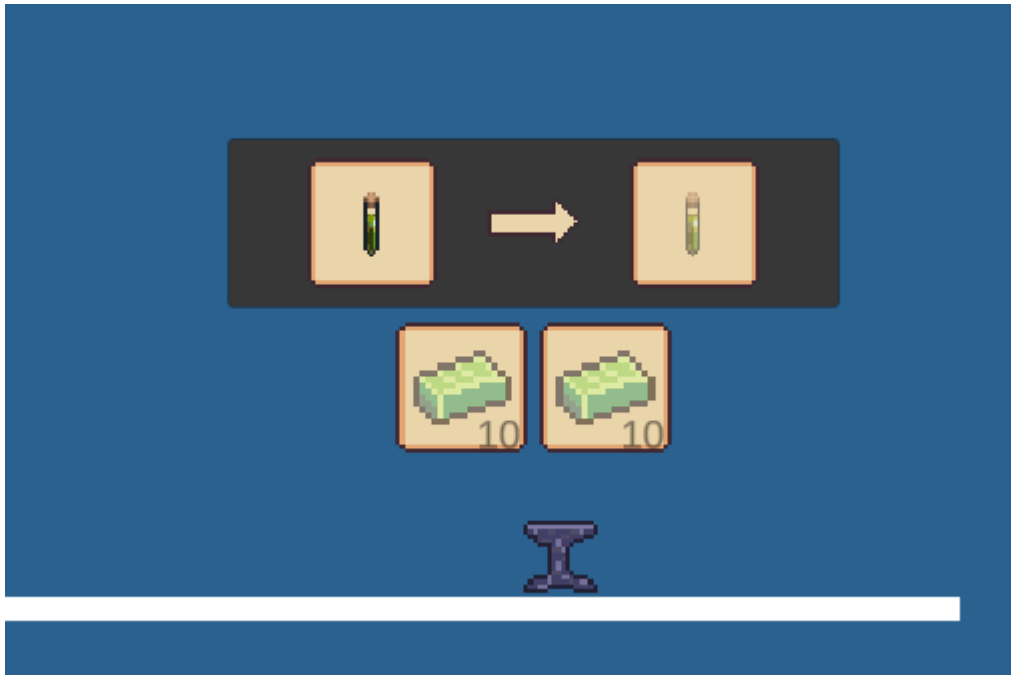


Image 7.6: SpeedBoosterPotionData can now be upgraded.

8. Save system

- The save system is an important aspect of any game. It allows players to save their progress and resume the game later. The save system is made up of two classes - **SaveManager.cs** and **SaveData.cs**.
- **SaveManager.cs** has some methods like: *Save()* and *Load()* for saving and loading data from **SaveData.cs**.
- **SaveData.cs** class is responsible for storing and retrieving game data. It has some method for converting game data into a json file or from a *json* file back to game data.
- Some data save in this package
 - + *Player inventory.*
 - + *Chest inventory.*
 - + *Item on ground (Item that drop by player).*
 - + *Item placed on ground* (Combat dummy, chest, anvil, crafting table)

9. Supports

- Gmail: conglapdev@gmail.com

