Inventory and Shop System
Technical Document

Version 1.0.0

Accardo Cheng

December 13, 2025

Contents

List of Code Listings

1 System Description

This document presents a technical overview and user manual for the Inventory and Shop System developed using the Unity Engine. The system aims to provide a solution for problems related to inventory management and item customization found in modern games.

The design of player inventory and item shop has long been a core aspect of game development due to their significant influence on player progression and engagement, particularly in role-playing games. However, as gameplay systems have evolved, these designs have often become unnecessarily complex for both players and developers, especially in titles that do not require such depth to support their content. Players may find themselves navigating multiple layers of inventory menus and managing duplicated items with limited variety, resulting in a tedious and time-consuming experience. Similarly, developers can face difficulties when editing existing items, adding new entries, or maintaining consistency between inventory and shop interfaces, often due to over design and inefficient code structures.

To address these issues, this system provides a simple and clearly structured foundation packaged as a modular, interoperable, and customizable development framework. Its key features include:

- A single folder with all the necessary components, allowing easy import while reducing the risk of overwriting existing files.
- C# script functions support simple steps in creating new items, editing existing variables, and separating types to be placed in the world or the shop list.
- An on-screen inventory UI that eliminates the need for a separated window. Items can be collected by using an assigned input key, and their corresponding sprites are automatically displayed in a clean and linear layout.
- A shop menu consists of two separate panels. The interface dynamically extracts and displays item information in vertical lists. Purchased upgrades or equipment immediately apply their effects, with effect magnitudes editable through the Inspector and extensible through additional scripts.
- A playable demo scene in which all features of the system can be tested.

2 System Installation and Usage

2.1 Installation Process

1. **Importing the System.** Drag the `InventoryShopSystem` folder into the Unity `Assets` directory. The package contains three subfolders:

   - `Scripts` — all C# files implementing the core system behaviour (inventory logic, currency tracking, UI interactions).
   - `Items` — several pre-made item ScriptableObjects for demonstration.
   - `Sprites` — icon assets for items and UI elements.

   In addition, the package provides six prefabs for immediate implementation.

2. **Attaching Core Scripts (System Entry Points).** In the `Scripts` folder, select `Inventory.cs` and `CurrencyManager.cs` and attach them to the player object, then select `ShopUpgradeManager.cs` and attach it to the shop object. These scripts are the *primary entry points* of the system:

   - `Inventory.cs` initializes and manages the player's inventory, providing methods for adding, removing, and querying items.
   - `CurrencyManager.cs` tracks the player's currency and ensures correct updates during buying and selling.
   - `ShopUpgradeManager.cs` handles shop transactions by interfacing with both `Inventory.cs` and `CurrencyManager.cs`, enabling the player to purchase equipment or upgrades through the shop UI.

```
public bool Add(Item item)
{
    if (items.Count >= capacity)
    {
        Debug.Log("Inventory Full!");
        return false;
    }
    items.Add(item);
    return true;
}
```

Listing 1: Example code snippet from `Inventory.cs`

```
 public bool SpendCoins(int amount)
    {
        if (coins >= amount)
        {
            coins -= amount;
            return true;
        }
        else
        {
            return false;
        }
    }
```

Listing 2: Example code snippet from `CurrencyManager.cs`

```
if (currency.SpendCoins(upgrade.buyPrice))
    {
        inventory.Add(upgrade);
        Debug.Log("Upgrade purchased: " + upgrade.name);
    }
else
    {
        Debug.Log("Not enough coins.");
        return;
    }
```

Listing 3: Example code snippet from `ShopUpgradeManager.cs`

3. **Setting Up the UI.** Create a Canvas in the Unity Hierarchy, then drag the following prefabs underneath it:

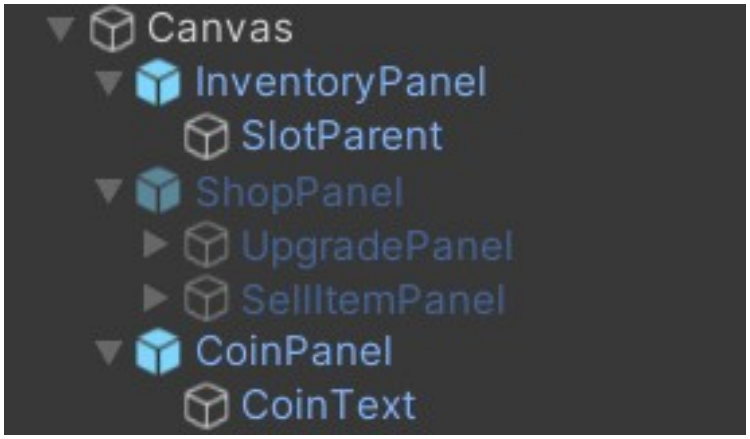

Figure 1: The Canvas tree after installation.

- `InventoryPanel` — constructs the inventory UI displayed at the top center of the screen. After placing the prefab in the Canvas, assign the `Player` object to the `Inventory` field in the Inspector, like in Figure 2.

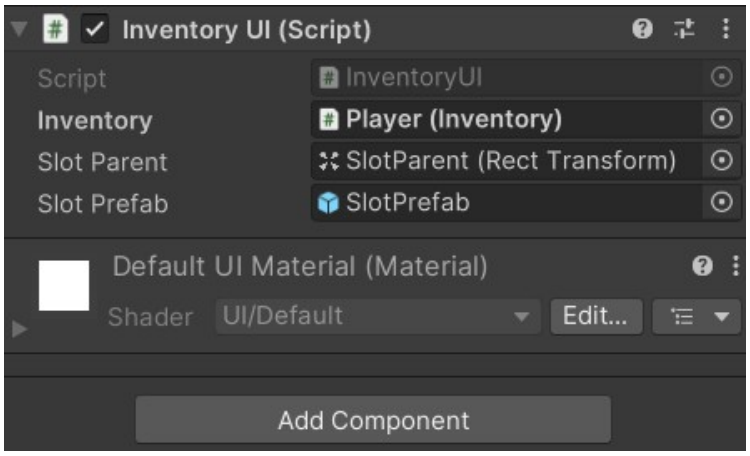

Figure 2: Fields under `InventoryUI.cs` in `InventoryPanel` inspector.

- `ShopPanel` — builds the buying and selling interface using the attached `ShopUIManager.cs` script. In the Inspector, assign the `Player` object to the `Inventory` and `Currency` fields, and assign the Shop object to the `Upgrade Manager` field.
- `CoinPanel` — displays the player's current currency using the attached `CurrencyUI.cs` script. After installation, assign the `Player` object to the `Currency` field in the Inspector.

4. **Additional Component: Item Pickup, Creation and Shop Trigger**
   - `ItemPickup.cs` This script allows the player to pick up items in the world using an assigned input key. Attach `ItemPickup.cs` to any in-scene item object that the player should be able to collect. When triggered, the script automatically adds the assigned item to the player's inventory.
   - `Item.cs` (ScriptableObject) This file defines the data structure for in-game items using Unity's ScriptableObject system.
   - `ShopTrigger.cs` This script allows the shop UI to open automatically when the player approaches a designated shop area and presses the assigned input key. Attach `ShopTrigger.cs` to a shop object with trigger collider in the scene.
5. **Supporting Prefabs.** The remaining prefabs (`SlotPrefab`, `SellItemUI`, and `UpgradeUI`) are internal UI components already referenced by

the main panels. Each includes its respective behaviour script (e.g., `SellItemUI.cs`, `UpgradeUI.cs`) and requires no additional setup.

2.2 Usage: Item creation and editing

The system includes the `Item.cs` ScriptableObject, which defines the data and visual representation of each in-game item. New items can be created directly inside the Unity Project window using the following steps:

1. Right-click in the Project window and select `Create` → `Inventory` → `Item`. This generates a new `Item.asset`.
2. Fill in the fields in the Inspector, including:

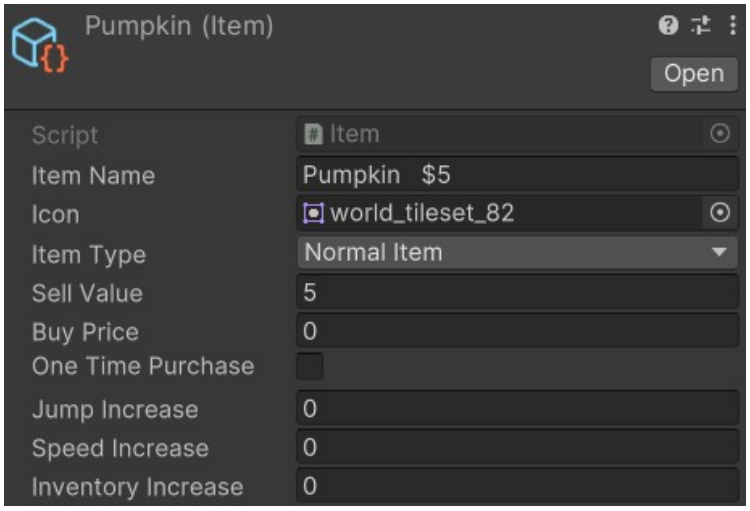

Figure 3: The item variables in inspector available for editing.

- **Name** — the display name of the item.
- **Icon** — sprite used in the inventory and shop UI.
- **Value** — currency cost or sell value.
- **Type** — Upgrade or Normal pickup item.
- **OneTimePurcahse** — Allow upgrade to be applied for one time.
- **Upgrade** — the magnitude of upgrade variables for player object.

3. Assign the created `Item.asset` to an in-scene object using `ItemPickup.cs`, allowing the player to collect it.

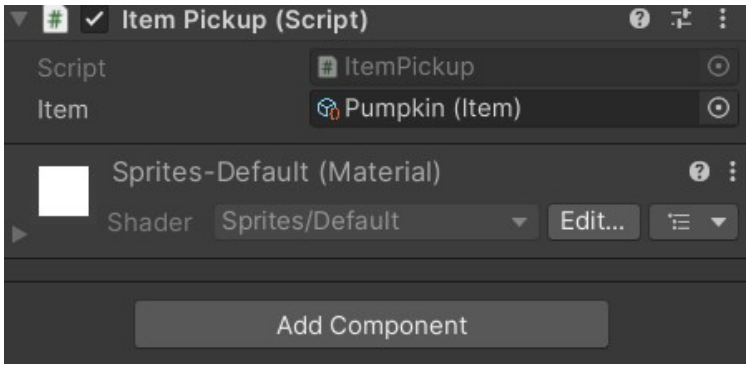

Figure 4: Assign item.asset to an in-game object.

4. To add the item to the shop list, set the item type to Upgrade and check the One Time Purchase box, then assign the item to `ShopUIManager.cs` in ShopPanel.

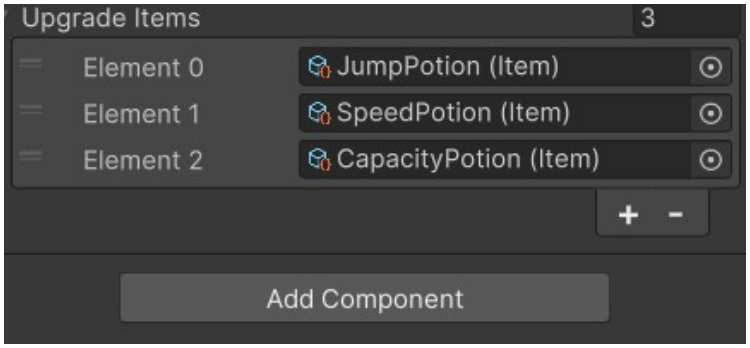

Figure 5: Add items to the shop list.

5. To add a new upgrade effect, use the following code example as reference, then adjust the magnitude in the item inspector.

```
public int healthIncrease;
```

Listing 4: Example `Inventory.cs` code to add upgrade to inspector

```
if (currency.SpendCoins(upgrade.buyPrice))
  {
     if (upgrade.healthIncrease != 0)
       {
         player.health += upgrade.healthIncrease;
       }
  }
```

Listing 5: Example `ShopUpgradeManager.cs` code to apply logic.

3 System Algorithms, Models, Mathematics

3.1 Inventory Model

The inventory system is modeled as a finite capacity container, represented as a set

$$\mathcal{I} = \{i_1, i_2, \dots, i_n\} \quad \text{where} \quad n \leq C,$$

where C is the maximum capacity of the inventory. An insertion is permitted only when the capacity constraint

$$n < C$$

is satisfied.

Each item i_k is defined as a tuple

$$i_k = (\text{name}, \text{icon}, \text{value}, \text{type})$$

allowing the system to generalize over both normal items and upgrade items.

3.2 Shop Sale Function

Let $V(i)$ denote the value of item i. Selling an item updates the player's currency:

$$\text{coins}_{t+1} = \text{coins}_t + V(i).$$

Simultaneously, the inventory is updated by removing one instance of i:

$$\mathcal{I}_{t+1} = \mathcal{I}_t \setminus \{i\}.$$

The operation is fully functional since the system enforces that the UI and inventory are always synchronized via the callback function:

$$\text{onInventoryChanged}() : \mathcal{I} \to \text{UI}.$$

Algorithm 1: Sell-One Procedure

Input: Item i, Inventory $\mathcal{I}$, Player currency C
Output: Updated inventory and currency
if $i \in \mathcal{I}$ **then**
    $\mathcal{I} \leftarrow \mathcal{I} \setminus \{i\}$;
    $C \leftarrow C + V(i)$;
    onInventoryChanged();
else
    return *Error: Item not found*;

3.3 Upgrade Purchase Model

A purchase is allowed if and only if the player has sufficient currency:

$$C \geq P(u),$$

where $P(u)$ is the upgrade price.

Upon successful purchase, player variables are updated:

$$\text{moveSpeed}_{t+1} = \text{moveSpeed}_t + \Delta v,$$
$$\text{jumpForce}_{t+1} = \text{jumpForce}_t + \Delta j,$$
$$C_{t+1} = C_t - P(u).$$

The upgrade can only be purchased once, expressed as:

$$u \notin \mathcal{I} \quad \text{before purchase.}$$

Algorithm 2: Upgrade Purchase Algorithm

Input: Upgrade u, Inventory $\mathcal{I}$, Player currency C
Output: Updated player attributes
if $u \in \mathcal{I}$ **then**
    return *Failure: Already owned*;
if $C < P(u)$ **then**
    return *Failure: Insufficient currency*;
$\mathcal{I} \leftarrow \mathcal{I} \cup \{u\}$;
$C \leftarrow C - P(u)$;
if $u.\Delta v \neq 0$ **then**
    moveSpeed $\leftarrow$ moveSpeed + $u.\Delta v$;
if $u.\Delta j \neq 0$ **then**
    jumpForce $\leftarrow$ jumpForce + $u.\Delta j$;
if $u.\Delta C \neq 0$ **then**
    capacity $\leftarrow$ capacity + $u.\Delta C$;
onInventoryChanged();

3.4 UI Synchronization via Observer Pattern

The UI synchronization mechanism follows the Observer Pattern [1], where inventory state changes automatically propagate to the rendering layer:

$$\text{onInventoryChanged} : \mathcal{I} \rightarrow \varnothing.$$

Whenever an item is added or removed, this callback is invoked:

$$\mathcal{I}_{t+1} = \begin{cases} \mathcal{I}_t \cup \{i\}, & \text{Add} \\ \mathcal{I}_t \setminus \{i\}, & \text{Remove} \end{cases}$$

$$\Rightarrow \text{onInventoryChanged}().$$

This ensures:

$$\forall t, \qquad UI_t = \mathcal{R}(\mathcal{I}_t).$$

Algorithm 3: Inventory UI Refresh Procedure

Input: Current inventory $\mathcal{I}$
Output: Updated UI slot icons
Clear all slot images;
for $k \leftarrow 1$ **to** $|\mathcal{I}|$ **do**
    $s_k \leftarrow$ slot at index k;
    $i_k \leftarrow$ item at index k in $\mathcal{I}$;
    Set icon of s_k to i_k.icon;

3.5 Shop System Model

3.5.1 Shop UI Auto-Population Model

The shop UI constructs a multiset of item counts:

$$M = \{(i, c_i) \mid i \in \mathcal{I},\ c_i = |\{x \in \mathcal{I} : x = i\}|\}.$$

Each pair (i, c_i) generates a row in the shop interface. Rows are regenerated whenever the shop menu is opened to ensure correctness:

$$\mathrm{UI}_{\mathrm{shop}} = \mathcal{G}(M),$$

where $\mathcal{G}$ is the row-generation function.

3.5.2 Algorithm: Selling Items from the Shop UI

Algorithm 4: Sell-One Action from UI

Input: Item i, Quantity c_i, Inventory $\mathcal{I}$, Currency C
if $c_i > 0$ **then**
    Remove item i from $\mathcal{I}$;
    $C \leftarrow C + V(i)$;
    Update quantity $c_i \leftarrow c_i - 1$;
    onInventoryChanged();
    if $c_i = 0$ **then**
        Remove UI row for i;

4 Demo Scenes

4.1 Inventory UI Display

As shown in Figure 6, the inventory UI is placed near the top center by default, and the currency UI on the top left displays the amount of coins acquired by the player.

Figure 6: Inventory and Currency UI

Use arrow keys to move around and pick up five items on screen with 'Z' key, the corresponding sprites are displayed on the interface:

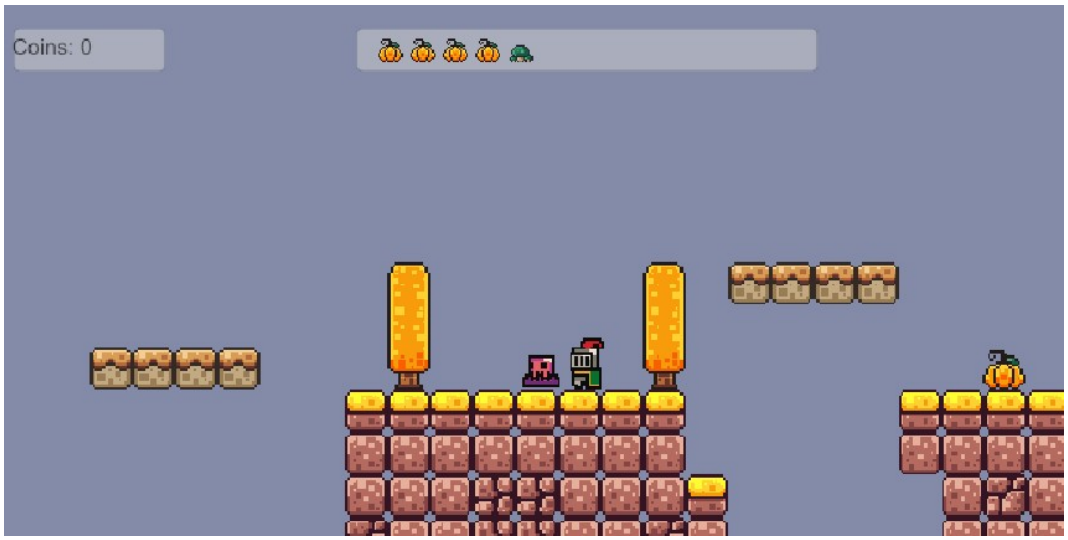

Figure 7: Inventory UI after picking up Items

4.2 Shop UI Display

Press 'Z' key near the shop object to access the trade menu, items in the inventory are listed on the right panel with *icon*, *name*, *price*, *quantity*, *Sell One* and *Sell All* button. Upgrades/equipment are presented on the left side, matching the items assigned to the inspector in Figure 5.

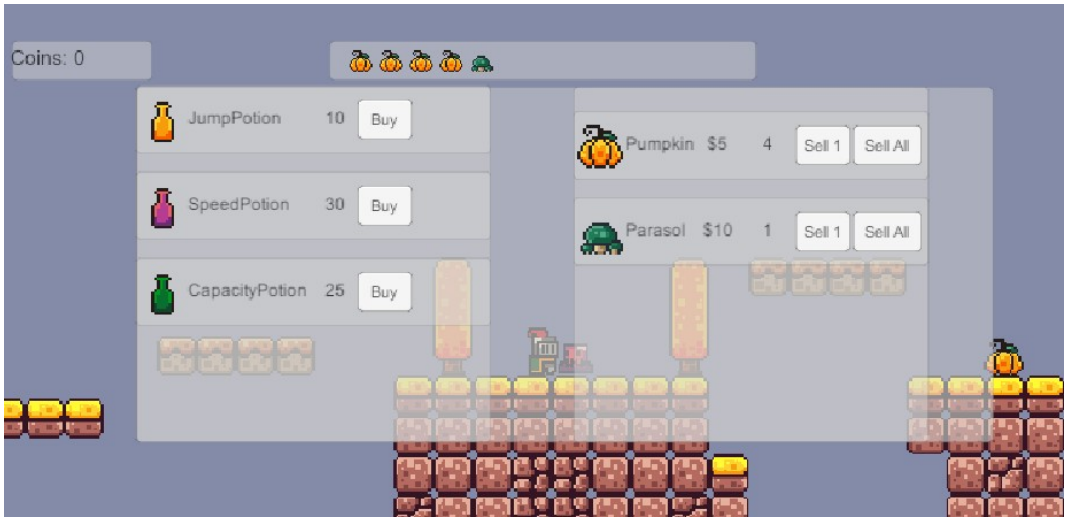

Figure 8: Shop UI with two panels

After pressing the sell one button on the Pumpkin item row, the quantity goes

down from 4 to 3, Inventory UI is updated to have 3 Pumpkin sprites displayed in slots, and the coin count is increased by 5.

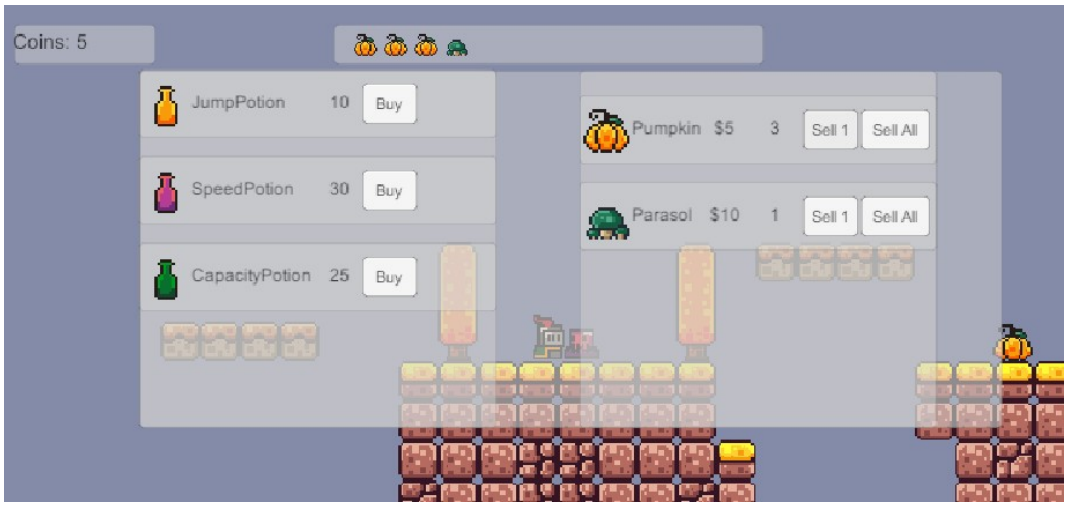

Figure 9: Updates in UI after selling one item

4.3 Upgrade Purchase

Since the current maximum inventory capacity is set to 5, player can not pick up the 6th item and the console will send "Inventory Full" message when attempt.

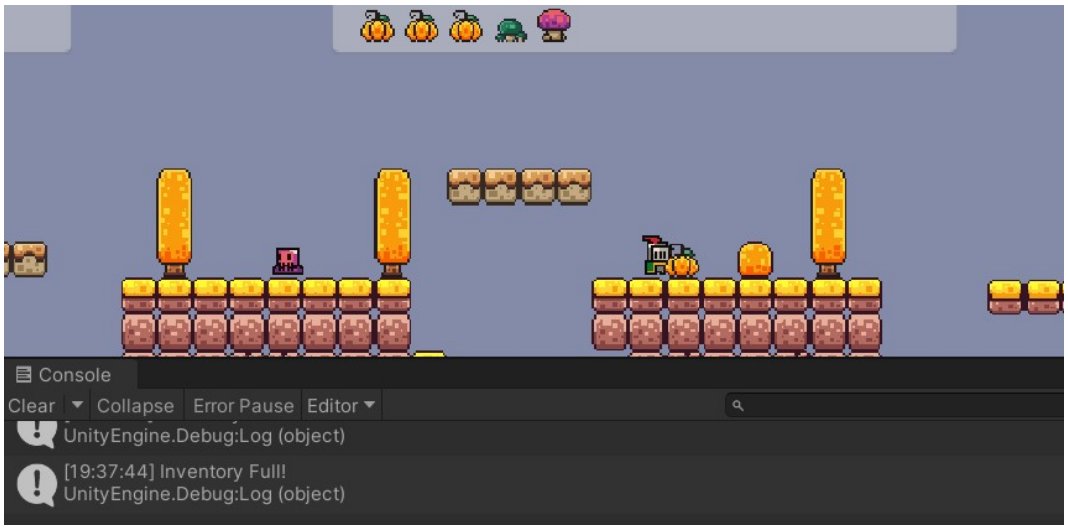

Figure 10: Exceed inventory capacity

After selling one parasol and two pumpkins, the player's coin total increases to 25. The Capacity Potion is then purchased from the shop, adding its sprite to the inventory UI and increasing the maximum inventory capacity by 5. The *Buy* button is disabled after the transaction due to the *One Time Purchase* option enabled in the Inspector.

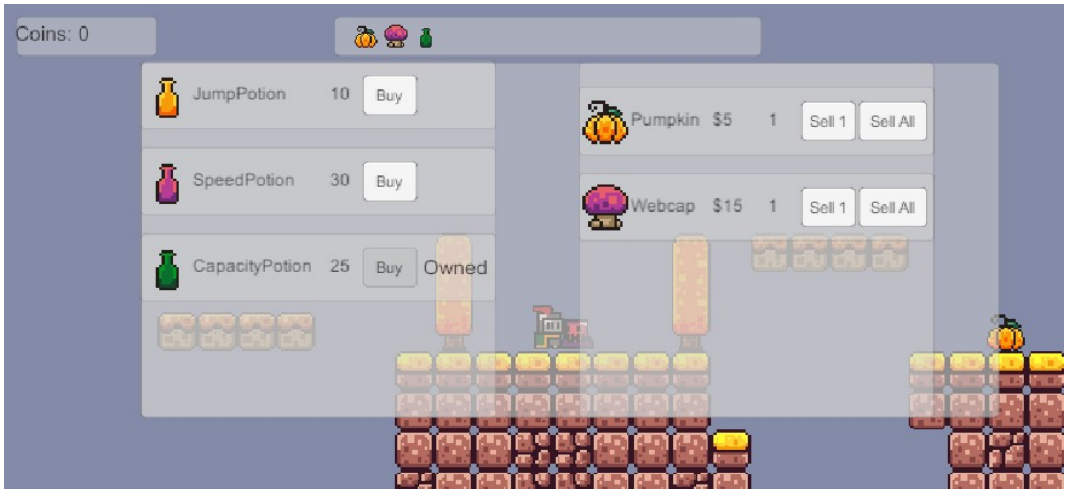

Figure 11: Updates in UI after purchasing an item

Check whether the upgrade effect is correctly applied by picking up 7 items, the capacity is shown to have increased to 10.

Figure 12: Upgraded Inventory with 10 capacity

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.