Inventory System

Technical Documentation

Harithik Rai
V00977569

University of Victoria
CSC 486A

Version 1.0.0
November 24, 2025

A modular, drag-and-drop inventory management system for Unity

Contents

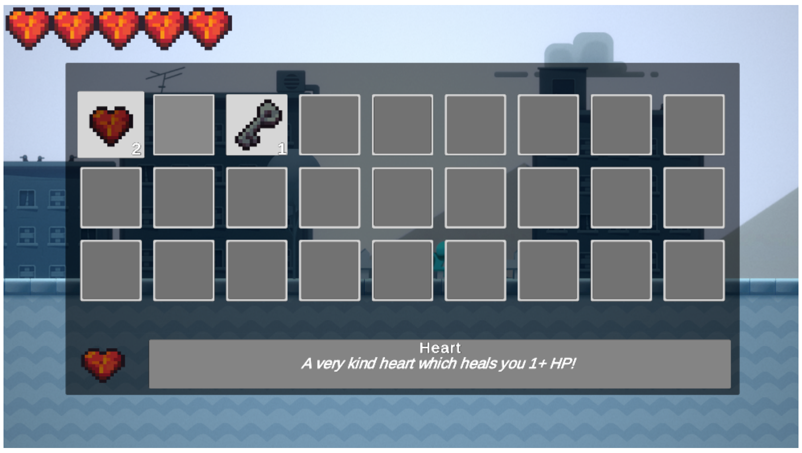


Figure 1: Inventory UI

1 System Overview

1.1 Description

The Modular Inventory System is a complete inventory management solution designed for Unity game engine. The system provides developers with a designer friendly inventory that can be integrated into any Unity project in minutes. Built using Unity top practices and a ScriptableObject-based architecture, this system allows non-programmers to create and manage items through Unity's Inspector while providing a lot of customization options for programmers as well.

The system was inspired by successful inventory implementations in games such as Minecraft (Mojang Studios, 2011) and Diablo III (Blizzard Entertainment, 2012), focusing on intuitive drag-and-drop interactions and efficient item management.

1.2 Key Features

- **Drag-and-Drop Item Management:** Smooth visual dragging with automatic slot detection
- **Intelligent Item Stacking:** Automatic merging of identical items with configurable stack limits per item type
- **ScriptableObject Items:** Designer friendly item creation through Unity Inspector with no code required for the non programmers out there
- **Item Usage System:** Left-click to use consumables with stat modification support (health, stamina, etc)
- **World Item Interaction:** Right-click to drop items into the game world, automatic or manual pickup modes
- **Item Categories:** Support for Consumable, Weapon, Armor, Material, Quest Item, and other types
- **Visual Selection:** Highlighted borders and description panel for selected items

- **Flexible Input:** Customizable keybindings for inventory toggle and item interactions
- **Demo Health System:** Complete health implementation with heart-based UI for testing and integration examples
- **Preset Items:** Two pre-configured example items (heart, key) ready for immediate testing
- **Zero External Dependencies:** Self-contained system requiring only Unity and TextMeshPro

2 System Structure

2.1 File Organization

The system follows a clear hierarchical structure optimized for modularity and ease of use:

```
InventorySystem/

 Scripts/
        ItemData.cs     (Item definition ScriptableObject)
        InventoryManager.cs (Central inventory controller)
        ItemSlot.cs               (Individual slot logic)
        Items.cs                  (Pickup item component)

 Prefabs/
        InventoryCanvas.prefab (Main prefab - Inventory UI)
        ItemSlot.prefab              (Single slot template)

 Demos/
        Scenes/
                 DemoScene.Unity (Scene to demo the system)
        Animations/
                 idle.anim     (Animations to move in demo)
                 jump.anim
                 walk.anim
                 Player.controller
        Scripts/
                 HealthDisplay.cs       (To display health)
                 movement.cs              (Player movement)
                 PlayerHealth.cs   (To demo player healing)
        Sprites and Prefabs/
                 character.PNG    (Sprites needed for demo)
                 SimpleTileset2.PNG
                 Sprite Atlas.spriteatlasv2
                 tilemap_packed.PNG
                 world_tilset.PNG

 Templates/
        HealthItem.asset          (Preset consumable item)
        KeyItem.asset                 (Preset weapon item)
        HeartData           (Holds all Item data for Heart)
        KeyData               (Holds all Item data for Key)

 Documentation/
                InventorySystem_Documentation.PDF
```

2.2 File Descriptions

Core Scripts:

- **ItemData.cs:** ScriptableObject class defining all item properties (name, sprite, description, stack size, type, effects). This is the data model for all items. Create instances via `Create → Inventory → Item`.
- **InventoryManager.cs:** Main controller managing inventory state, input handling, and item operations. Coordinates between ItemSlots and provides public API for external systems. Uses `[RequireComponent(typeof(Canvas))]` to ensure proper UI setup.
- **ItemSlot.cs:** Individual slot component handling visual display, user interaction, and drag-and-drop logic. Implements Unity Event System interfaces (`IPointerClickHandler`, `IBeginDragHandler`, etc.).

Item Scripts:

- **Items.cs:** Component attached to pickup GameObjects in the world. Handles collision/trigger detection and inventory addition. Supports both automatic pickup on collision and manual pickup via key press.

Scripts for Demo purposes:

- **PlayerHealth.cs:** Example integration showing health stat system. Provides `ChangeHealth(int)` method that items can call. Includes `[Range(0, 100)]` attributes for Inspector constraints.
- **HealthDisplay.cs:** UI component that displays health as hearts in top-left corner. Automatically updates when health changes. Demonstrates event-driven UI updates.

Prefabs:

- **InventoryCanvas.prefab:** The main prefab containing complete inventory UI. Simply drag into scene hierarchy to add inventory system. Pre-configured with slots, description panel, and all references connected.
- **ItemSlot.prefab:** Template for individual inventory slots. Used when customizing inventory size. Contains Image, TextMeshPro, and CanvasGroup components properly configured.

  Preset Items:

  - **Heart.asset:** Pre-configured consumable item. Restores 1 health when used. Max stack size: 3. Provides immediate testing capability without creating items.
  - **Key.asset:** Pre-configured weapon item. Max stack size: 1 (Quest item). Example of non-consumable item type for systems with unlockable doors/areas.

3 Code Structure

3.1 ItemData Class

The ItemData ScriptableObject serves as the data definition for all items. It uses Unity's `[CreateAssetMenu]` attribute to enable creation through the Unity Editor.

```
[CreateAssetMenu(fileName = "New Item", menuName = "Inventory/Item")]
public class ItemData : ScriptableObject
{
    // Basic properties
    public string itemName;
    public Sprite itemSprite;
    [TextArea(3, 5)]
    public string itemDescription;

    // Gameplay properties
    public int maxStackSize = 64;
    public ItemType itemType;
    public bool isUsable = true;
    public bool isDroppable = true;

    // Effect properties
    public StatToChange statToChange = StatToChange.none;
    public int amountToChangeStat;

    // Usage method - override for custom effects
    public virtual bool UseItem()
    {
        GameObject player = GameObject.FindGameObjectWithTag("Player");
        if (player == null) return false;

        if (statToChange == StatToChange.health)
        {
            var health = player.GetComponent<PlayerHealth>();
            if (health != null && health.health < health.maxHealth)
            {
                health.ChangeHealth(amountToChangeStat);
                return true;
            }
        }
        return false;
    }
}
```

Key Design Decisions:

- Uses `[TextArea]` attribute for multi-line descriptions in Inspector
- `UseItem()` marked as virtual to enable inheritance for custom items
- Returns boolean to indicate whether item should be consumed
- Uses tag placed on ”player” for maximum compatibility

3.2 InventoryManager Class

Central controller implemented through `FindObjectOfType`.

```csharp
public class InventoryManager : MonoBehaviour
{
    public GameObject InventoryMenu;
    public ItemSlot[] itemSlot;
    public KeyCode inventoryKey = KeyCode.I;
    private bool menuActivated;

    void Update()
    {
        if (Input.GetKeyDown(inventoryKey))
            ToggleInventory();
    }

    public int AddItem(ItemData itemData, int quantity)
    {
        // Phase 1: Try stacking with existing items
        for (int i = 0; i < itemSlot.Length; i++)
        {
            if (itemSlot[i].itemData == itemData && !itemSlot[i].isFull)
            {
                int leftover = itemSlot[i].AddItem(itemData, quantity);
                if (leftover > 0)
                    return AddItem(itemData, leftover);
                return 0;
            }
        }

        // Phase 2: Find empty slot
        for (int i = 0; i < itemSlot.Length; i++)
        {
            if (itemSlot[i].quantity == 0)
            {
                int leftover = itemSlot[i].AddItem(itemData, quantity);
                if (leftover > 0)
                    return AddItem(itemData, leftover);
                return 0;
            }
        }

        return quantity; // Inventory full
    }
}
```

3.3 ItemSlot Class

Implements all Unity Event System interfaces for drag-and-drop functionality:

```csharp
public class ItemSlot : MonoBehaviour,
    IPointerClickHandler, IBeginDragHandler,
    IEndDragHandler, IDragHandler, IDropHandler
{
    public ItemData itemData;
    public int quantity;
    private GameObject draggedIcon;

    public void OnBeginDrag(PointerEventData eventData)
    {
        if (string.IsNullOrEmpty(itemName)) return;

        // Create temporary visual
        draggedIcon = new GameObject("DraggedIcon");
        Canvas canvas = GetComponentInParent<Canvas>();
        draggedIcon.transform.SetParent(canvas.transform);

        Image img = draggedIcon.AddComponent<Image>();
        img.sprite = itemSprite;
        img.raycastTarget = false;

        // Make original semi-transparent
        GetComponent<CanvasGroup>().alpha = 0.6f;
        GetComponent<CanvasGroup>().blocksRaycasts = false;
    }

    public void OnDrop(PointerEventData eventData)
    {
        ItemSlot draggedSlot = eventData.pointerDrag
            .GetComponent<ItemSlot>();
        if (draggedSlot == null) return;

        // Check for stacking
        if (itemName == draggedSlot.itemName)
        {
            int space = itemData.maxStackSize - quantity;
            int transfer = Mathf.Min(space, draggedSlot.quantity);

            quantity += transfer;
            draggedSlot.quantity -= transfer;

            // Update visuals...
        }
        else
        {
            // Swap items between slots...
        }
    }
}
```

Ensured to use Best Practices in Unity:

– Uses `CanvasGroup` for transparency and raycast blocking
– `raycastTarget = false` on drag icon prevents self-blocking
– Properly destroys temporary GameObjects in `OnEndDrag`
– All public fields serialized for Inspector visibility

3.4 Items Class

Pickup component supporting both collision and trigger-based pickup:

```
public class Items : MonoBehaviour
{
    public ItemData itemData;
    public int quantity = 1;
    public bool autoPickup = true;
    public KeyCode pickupKey = KeyCode.E;

    private InventoryManager inventoryManager;
    private bool playerInRange = false;

    void Start()
    {
        inventoryManager = FindObjectOfType<InventoryManager>();
    }

    void OnCollisionEnter2D(Collision2D collision)
    {
        if (autoPickup && collision.gameObject.CompareTag("Player"))
            TryPickup();
    }

    void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            if (autoPickup)
                TryPickup();
            else
                playerInRange = true;
        }
    }

    void TryPickup()
    {
        int remaining = inventoryManager.AddItem(itemData, quantity);
        if (remaining == 0)
            Destroy(gameObject);
        else
            quantity = remaining;
    }
}
```

3.5 PlayerHealth Demo System

Included health system demonstrating integration with other systems:

```
public class PlayerHealth : MonoBehaviour
{
    [Range(0, 100)]
    public int maxHealth = 100;

    [Range(0, 100)]
    public int health = 100;

    public void ChangeHealth(int amount)
    {
        health = Mathf.Clamp(health + amount, 0, maxHealth);
        Debug.Log($"Health: {health}/{maxHealth}");
    }
}
```

The `[Range]` attribute provides slider controls in Inspector for easy testing.

4 Installation Procedures

4.1 System Requirements

- Unity 2020.3 LTS or newer
- TextMeshPro package (automatically imported)
- Player GameObject with ”Player” tag

– EventSystem in scene (required for UI interaction)

Important Requirement: An EventSystem must exist in your scene for drag-and-drop to function. Unity automatically creates one when adding UI, but if missing, please add via `GameObject → UI → Event System`.

4.2 Quick Installation

The system is designed for instant integration with your project:

Step 1: Import Package

1. Open your Unity project
2. Import the InventorySystem package
3. Verify folder structure appears in the Assets

Step 2: Setup Player

1. Select your player GameObject
2. Set Tag to "Player" in Inspector
3. (Optional) Add PlayerHealth component for consumables
4. (Optional) Add HealthDisplay for visual health UI

Step 3: Add Inventory

1. Locate `Prefabs/InventoryCanvas.prefab`
2. Drag into scene Hierarchy and ensure "Inventory Menu" is checked as off in the inspector so the inventory is not already on screen right when you start the game
3. Done! System automatically finds player and configures itself

Step 4: Verify the EventSystem

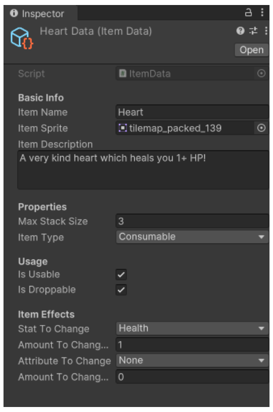
1. Check Hierarchy for EventSystem GameObject
2. If missing: `GameObject → UI → Event System`
3. EventSystem required for drag-and-drop functionality

4.3 Using the Demo Scene

To quickly test the inventory system, a ready-made demo scene is included. Follow these steps to open and use it:

1. In Unity, locate the folder `InventorySystem/Demos/Scenes`.
2. Drag the `DemoScene.unity` file into the Hierarchy to open it.
3. Create a new layer called "Ground"
4. Assign a **Ground Layer** to the ground object (TilemapFirst).
5. Select the Player object and open the movement script in the inspector.
6. Set the movement script's **Ground Layer** field to the layer you created.

This demo scene provides a simple environment for testing item pickup, usage, and overall system behavior without requiring any additional setup.



Script: ItemData

Basic Info

Item Name	Heart
Item Sprite	tilemap_packed_139

Item Description

A very kind heart which heals you 1+ HP!

Properties

Max Stack Size	3
Item Type	Consumable

Usage

Is Usable	✓
Is Droppable	✓

Item Effects

Stat To Change	Health
Amount To Chang...	1
Attribute To Change	None
Amount To Chang...	0

Figure 2: Item editor in inspector

4.4 Creating Custom Items

Using Preset Items:

Two items are pre-configured and ready to use:

- `Heart.asset`: Consumable, restores 1 HP, stacks to 3
- `Key.asset`: Quest item (stack size 1)

Simply drag these out in the world to use as templates.
Creating New Items:

1. Right-click in Project window
2. `Create → Inventory → Item`
3. Configure in the Inspector:
    - Item Name: Display name
    - Item Sprite: Visual representation of item
    - Item Description: Tooltip text
    - Max Stack Size: How many can stack
    - Item Type: Category (Consumable, Weapon, etc.)
    - Is Usable: Can be consumed
    - Is Droppable: Can be dropped into world
    - Stat To Change: Which stat to modify (if consumable)
    - Amount To Change Stat: Magnitude of effect

5 Usage Procedures

5.1 Basic Operations

Opening/Closing Inventory:

Press `E` key (configurable in InventoryManager Inspector). Toggle key can be changed to any Key.

Selecting Items:
Left-click on any occupied slot to select. Selection shows:

- Highlighted border around slot
- Item name in description panel
- Item description text
- Item sprite in description panel

Using Items:
Left-click on already-selected slot to use. For consumables:

- Item effect applies (e.g., restores health)
- Quantity decreases by 1
- Slot empties if quantity reaches 0
- Item not consumed if use fails (e.g., health already full)

Dropping Items:
Right-click on any slot to drop item into world:

- Item spawns near player (1 unit to the left)
- Quantity in slot decreases by 1
- Dropped item can be picked up again
- Only works if `isDroppable = true` in ItemData

Dragging Items:
Click and hold on slot, drag to another slot:

- Dragged item becomes semi-transparent
- Temporary icon follows mouse cursor
- Drop on empty slot: moves item
- Drop on different item: swaps items
- Drop on same item: merges quantities (up to a set stack limit)

5.2 Pickup Methods

Automatic Pickup (default):
Walk player into item. Item automatically adds to inventory and disappears from world.

Manual Pickup:
On WorldItem, set `Auto Pickup = false` in Items component. Player must:

1. Walk near item (enters trigger collider)
2. Press `E` key (configurable via Pickup Key field)
3. Item adds to inventory

5.3 Customization Options

Changing Inventory Size:

1. Select `InventoryCanvas → InventoryPanel → Grid`
2. Modify GridLayoutGroup component:
   - Constraint Count: Number of columns
   - Cell Size: Slot dimensions
   - Spacing: Gap between slots
3. Duplicate ItemSlot prefabs to fill desired size
4. Update InventoryManager's `itemSlot` array size
5. Drag all slots into array

Changing Keybinds:

- Select InventoryCanvas GameObject
- In InventoryManager component, change `Inventory Key`
- On WorldItem pickups, change `Pickup Key` in Items component

Visual Customization:
All UI elements can be styled:

- Select any UI element (slot, panel, text)
- Modify Image component: color, sprite
- Modify TextMeshPro: font, size, color
- Adjust RectTransform for positioning

6 Entry Points

6.1 Code Entry Points

Adding Items Programmatically:

```
public class ChestScript : MonoBehaviour
{
    public ItemData goldCoin;

    void OnInteract()
    {
        InventoryManager inventory =
            FindObjectOfType<InventoryManager>();

        int remaining = inventory.AddItem(goldCoin, 50);

        if (remaining == 0)
            Debug.Log("All coins collected!");
        else
            Debug.Log($"Inventory full! Lost {remaining} coins");
    }
}
```

Checking Inventory Contents:

```
bool HasItem(string itemName, int minQuantity = 1)
{
    InventoryManager inventory =
        FindObjectOfType<InventoryManager>();

    int total = 0;
    foreach (ItemSlot slot in inventory.itemSlot)
    {
        if (slot.itemName == itemName)
            total += slot.quantity;
    }

    return total >= minQuantity;
}
```

Creating Custom Item Effects:
Extend ItemData class for custom behavior:

```
public class ManaPotion : ItemData
{
    public override bool UseItem()
    {
        GameObject player = GameObject.FindGameObjectWithTag("Player");
        ManaSystem mana = player.GetComponent<ManaSystem>();

        if (mana != null && mana.currentMana < mana.maxMana)
        {
            mana.AddMana(amountToChangeStat);
            return true;
        }
        return false;
    }
}
```

Then create items using this derived class through `Create → Inventory → Mana Potion`.

6.2 Editor Interfaces

ItemData Inspector:
When selecting an ItemData asset, the Inspector displays:

- **Basic Info:** Item Name (text field), Item Sprite (object reference), Item Description (text area with 3-5 line preview)
- **Properties:** Max Stack Size (integer), Item Type (dropdown with 6 options)
- **Usage:** Is Usable (checkbox), Is Droppable (checkbox)
- **Item Effects:** Stat To Change (dropdown), Amount To Change Stat (integer), Attribute To Change (dropdown), Amount To Change Attribute (integer)

All fields have clear labels and appropriate input controls. Hover over fields for tooltips.
InventoryManager Inspector:
Key configuration fields:

- **Inventory Menu:** Drag GameObject reference to inventory panel
- **Item Slot:** Array of all ItemSlot components (auto-populated in prefab)

- **Inventory Key:** Dropdown menu with all KeyCode options (default: I)

ItemSlot Inspector:
Serialized fields per slot:

- **Empty Sprite:** Sprite shown when slot is empty
- **Max Number Of Items:** Fallback stack limit (default: 64)
- **Quantity Text:** TMP_Text component reference
- **Item Image:** Image component reference
- **Item Description Image/Text:** References to description panel
- **Selected Shader:** GameObject activated when selected

All references are pre-configured in the InventoryCanvas prefab.
Items Inspector:
Pickup configuration:

- **Item Data:** Drag ItemData asset here (required)
- **Quantity:** Number of items in this pickup (default: 1)
- **Auto Pickup:** Checkbox for automatic vs. manual (default: true)
- **Pickup Key:** KeyCode for manual pickup (default: E)
- **Pickup Range:** Float for future proximity detection (default: 2.0)

PlayerHealth Inspector:
Demo health system fields:

- **Max Health:** Range slider 0-100
- **Health:** Range slider 0
- **Health:** Range slider 0-100

The `[Range]` attributes provide visual slider controls for easy value adjustment during testing.

6.3 RequireComponent Usage

The system uses Unity's `RequireComponent` attribute to ensure dependencies:

```
// In ItemSlot.cs
[RequireComponent(typeof(Image))]
[RequireComponent(typeof(CanvasGroup))]
public class ItemSlot : MonoBehaviour
{
    // Ensures Image and CanvasGroup are present
}
```

This prevents configuration errors by automatically adding required components when the script is attached.

6.4 Example Prefabs

The system includes a main prefab ready for immediate use:

InventoryCanvas.prefab:

Complete inventory UI with:

- Pre-configured ItemSlots in a grid
- Description panel with image and text fields
- All component references properly connected
- Canvas set to Screen Space - Overlay
- EventSystem included for input handling

Usage: Drag directly into scene Hierarchy. No additional configuration required.

6.5 Scene View Gizmos

The current version does not include custom Gizmos but the system does still provide clear visual feedback:

Runtime Visual Indicators:

- Selected slots display highlighted border (selectedShader GameObject)
- Dragged items show semi-transparent original
- Temporary drag icon follows mouse cursor with its full opacity
- Quantity text updates in real time during operations
- Empty slots show empty sprite as needed
- Health UI displays filled/empty hearts based on current health

Future Gizmo Extension:

Developers can add custom Gizmos for better debugging. Heres an example implementation for visualizing pickup range:

```
// Add to Items.cs for pickup range visualization
#if UNITY_EDITOR
private void OnDrawGizmosSelected()
{
    if (!autoPickup)
    {
        Gizmos.color = new Color(0, 1, 0, 0.3f);
        Gizmos.DrawSphere(transform.position, pickupRange);

        Gizmos.color = Color.green;
        Gizmos.DrawWireSphere(transform.position, pickupRange);
    }
}
#endif
```

7 Models and Mathematics

7.1 Item Stacking Algorithm

The core of the inventory system uses a two-phase greedy algorithm to optimize item placement.

Problem Definition:

Given n inventory slots and q items to add, minimize the number of slots used while respecting per-item stack limits.

Variables:

- n = total number of inventory slots
- q_{add} = quantity of items to add
- q_i = current quantity in slot i
- m = maximum stack size for item type
- s_i = available space in slot $i = m - q_i$

Algorithm:

Phase 1: Stack Consolidation

For all slots where item type matches and slot not full:

$$t_i = \min(s_i, q_{add}) \tag{1}$$

$$q'_i = q_i + t_i \tag{2}$$

$$q'_{add} = q_{add} - t_i \tag{3}$$

Continue until $q'_{add} = 0$ or no compatible slots remain.

Phase 2: New Stack Creation

If $q'_{add} > 0$, find empty slot j:

$$t_j = \min(m, q'_{add}) \tag{4}$$

Recursively call AddItem if $q'_{add} - t_j > 0$.

7.2 Drag-and-Drop Coordinate Transform

The drag system performs coordinate space transformations to position the visual icon at the cursor.

Coordinate Spaces:

Unity uses multiple coordinate systems:

- **World Space:** 3D game coordinates
- **Screen Space:** 2D pixel coordinates (origin at bottom-left)
- **Canvas Space:** UI coordinates (depends on Canvas Render Mode)

Transform Equation for Screen Space - Overlay:

$$\mathbf{P}_{canvas} = \mathbf{P}_{screen} \tag{5}$$

For Screen Space - Overlay (used in this system), canvas coordinates directly match screen pixel coordinates:

```
draggedIcon.transform.position = Input.mousePosition;
```

Alpha Blending for Visual Feedback:
Semi-transparency uses standard alpha blending:

$$C_{final} = \alpha \cdot C_{slot} + (1 - \alpha) \cdot C_{background} \tag{6}$$

where $\alpha = 0.6$ creates the semi-transparent effect during drag.
Raycast Detection:
Unity's GraphicRaycaster performs point-in-rectangle tests for UI elements:

$$\text{hit}(\mathbf{p}) = \begin{cases} \text{true} & \text{if } x_{min} \leq p_x \leq x_{max} \text{ and } y_{min} \leq p_y \leq y_{max} \\ \text{false} & \text{otherwise} \end{cases} \tag{7}$$

Setting `blocksRaycasts = false` ensures dragged slot doesn't block detection:

$$\text{detectable} = \text{raycastTarget} \wedge \text{blocksRaycasts} \tag{8}$$

7.3 Stack Merging Mathematics

When dropping items on same type, quantities merge with overflow handling.
Variables:

- q_t = quantity in target slot
- q_d = quantity in dragged slot
- m = maximum stack size
- s_t = available space $= m - q_t$

Transfer Calculation:

$$t = \min(s_t, q_d) \tag{9}$$

Post-Transfer State:

$$q'_t = q_t + t \tag{10}$$

$$q'_d = q_d - t \tag{11}$$

Slot State Functions:

$$\text{isFull}_t = \begin{cases} \text{true} & \text{if } q'_t \geq m \\ \text{false} & \text{otherwise} \end{cases} \tag{12}$$

$$\text{isEmpty}_d = \begin{cases} \text{true} & \text{if } q'_d = 0 \\ \text{false} & \text{otherwise} \end{cases} \tag{13}$$

7.4 Grid Layout Mathematics

The inventory uses Unity's GridLayoutGroup for positioning slots.

Parameters:

- c = columns (constraint count)
- w_s, h_s = cell width and height (pixels)
- w_g, h_g = horizontal and vertical spacing
- p_l, p_r, p_t, p_b = padding (left, right, top, bottom)

Position Calculation:
For slot at index i (zero-indexed):

$$r = \lfloor i/c \rfloor \tag{14}$$

$$col = i \mod c \tag{15}$$

$$x_i = p_l + col \cdot (w_s + w_g) \tag{16}$$

$$y_i = p_t + r \cdot (h_s + h_g) \tag{17}$$

Total Grid Dimensions:

$$W_{total} = p_l + p_r + c \cdot w_s + (c - 1) \cdot w_g \tag{18}$$

$$H_{total} = p_t + p_b + \lceil n/c \rceil \cdot h_s + (\lceil n/c \rceil - 1) \cdot h_g \tag{19}$$

where n is total number of slots.

7.5 Health Modification Formula

The demo health system uses clamped arithmetic for stat changes.

Health Change:

$$H' = \text{clamp}(H + \Delta H, 0, H_{max}) \tag{20}$$

where:

- H = current health
- ΔH = amount to change (positive for healing, negative for damage)
- H_{max} = maximum health capacity

Clamp Function:

$$\text{clamp}(x, a, b) = \begin{cases} a & \text{if } x < a \\ b & \text{if } x > b \\ x & \text{otherwise} \end{cases} \tag{21}$$

Usage Validation:

Items can only be used if they produce meaningful change:

$$\text{canUse} = \begin{cases} \text{false} & \text{if } \Delta H > 0 \text{ and } H = H_{max} \\ \text{false} & \text{if } \Delta H < 0 \text{ and } H = 0 \\ \text{true} & \text{otherwise} \end{cases} \tag{22}$$

Example:
Player at 75 HP drinks potion restoring 50 HP:

$$H = 75 \tag{23}$$
$$H_{max} = 100 \tag{24}$$
$$\Delta H = 50 \tag{25}$$
$$H' = \text{clamp}(75 + 50, 0, 100) = \text{clamp}(125, 0, 100) = 100 \tag{26}$$

Player at full health cannot use healing potion:

$$H = 100, \quad H_{max} = 100, \quad \Delta H = 50 \tag{27}$$
$$\text{canUse} = \text{false} \quad \text{(already at maximum)} \tag{28}$$

7.6 Collision Detection Mathematics

The pickup system uses Unity's 2D physics for collision detection.
AABB (Axis-Aligned Bounding Box) Intersection:
Two boxes A and B intersect if:

$$\begin{gathered} x_{A,min} \leq x_{B,max} \wedge x_{A,max} \geq x_{B,min} \wedge \\ y_{A,min} \leq y_{B,max} \wedge y_{A,max} \geq y_{B,min} \end{gathered} \tag{29}$$

For BoxCollider2D with center (c_x, c_y) and size (w, h):

$$x_{min} = c_x - \frac{w}{2}, \quad x_{max} = c_x + \frac{w}{2} \tag{30}$$

$$y_{min} = c_y - \frac{h}{2}, \quad y_{max} = c_y + \frac{h}{2} \tag{31}$$

Circle Collision (CircleCollider2D):
Two circles intersect if distance between centers is less than sum of radii:

$$d = \|\mathbf{p}_A - \mathbf{p}_B\| = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2} \tag{32}$$

$$\text{intersect} = d < r_A + r_B \tag{33}$$

7.7 UI Scaling and Canvas Scaler

The Canvas uses Unity's CanvasScaler for resolution independence.
Scale Factor Calculation:
For Screen Space - Overlay with Scale Mode = Scale With Screen Size:

$$s = \frac{W_{screen}}{W_{ref}}^{1-m} \cdot \frac{H_{screen}}{H_{ref}}^{m} \tag{34}$$

where:

- W_{screen}, H_{screen} = actual screen dimensions
- W_{ref}, H_{ref} = reference resolution (typically 1920×1080)
- m = match parameter (0 = width, 1 = height, 0.5 = average)

RectTransform Anchor System:
Positions calculated relative to anchors:

$$\mathbf{p}_{final} = \mathbf{p}_{parent} + \mathbf{a}_{normalized} \cdot \mathbf{s}_{parent} + \mathbf{o}_{local} \tag{35}$$

where:

- $\mathbf{a}_{normalized}$ = anchor position (0-1 range)
- $\mathbf{s}_{parent}$ = parent RectTransform size
- $\mathbf{o}_{local}$ = local position offset

This ensures UI elements maintain relative positions across resolutions.

8 User-Facing Code Documentation

8.1 Public API Reference

InventoryManager.AddItem()

```
/// <summary>
/// Adds items to inventory with intelligent stacking.
/// </summary>
/// <param name="itemData">The ItemData asset to add</param>
/// <param name="quantity">Number of items to add</param>
/// <returns>Number of items that didn't fit (0 if all added)</returns>
/// <example>
/// <code>
/// InventoryManager inv = FindObjectOfType<InventoryManager>();
/// int leftover = inv.AddItem(healthPotion, 10);
/// if (leftover > 0) {
///     Debug.Log($"Couldn't fit {leftover} items");
/// }
/// </code>
/// </example>
public int AddItem(ItemData itemData, int quantity)
```

InventoryManager.UseItem()

```
/// <summary>
/// Attempts to use an item by name from inventory.
/// </summary>
/// <param name="itemName">Name of item to use (case-sensitive)</param>
/// <returns>True if item was found and successfully used</returns>
/// <remarks>
/// Searches all slots for matching item name and calls its
/// UseItem() method. If successful, quantity decreases by 1.
/// </remarks>
public bool UseItem(string itemName)
```

InventoryManager.ToggleInventory()

```
/// <summary>
/// Toggles inventory UI between open and closed states.
/// </summary>
/// <remarks>
/// Called automatically by Update() when inventory key pressed.
/// Can also be called programmatically for custom input handling.
/// </remarks>
public void ToggleInventory()
```

ItemData.UseItem()

```
/// <summary>
/// Executes the item's effect when used.
/// Override in derived classes for custom behavior.
/// </summary>
/// <returns>
/// True if item was successfully used (will be consumed),
/// false if use failed (item retained)
/// </returns>
/// <remarks>
/// Base implementation handles health and stamina stat changes.
/// Return false to prevent item consumption (e.g., health already full).
/// </remarks>
public virtual bool UseItem()
```

ItemSlot.AddItem()

```
/// <summary>
/// Adds items to this slot with overflow handling.
/// </summary>
/// <param name="data">ItemData asset to add</param>
/// <param name="quantity">Number of items to add</param>
/// <returns>Number of items that didn't fit in this slot</returns>
/// <remarks>
/// If adding more items than slot can hold, fills to maximum
/// capacity and returns overflow amount.
/// </remarks>
public int AddItem(ItemData data, int quantity)
```

PlayerHealth.ChangeHealth()

```
/// <summary>
/// Modifies player health by specified amount.
/// </summary>
/// <param name="amount">
/// Amount to change health by. Positive for healing,
/// negative for damage.
/// </param>
/// <remarks>
/// Result is clamped between 0 and maxHealth.
/// Triggers health UI update automatically.
/// </remarks>
public void ChangeHealth(int amount)
```

8.2 Common Usage Patterns

Quest System Integration:

```
public class QuestSystem : MonoBehaviour
{
    public ItemData questItem;
    public int requiredQuantity = 5;

    public bool CheckQuestComplete()
    {
        InventoryManager inv = FindObjectOfType<InventoryManager>();

        int total = 0;
        foreach (ItemSlot slot in inv.itemSlot)
        {
            if (slot.itemData == questItem)
                total += slot.quantity;
        }

        return total >= requiredQuantity;
    }
}
```

Shop System Integration:

```
public class ShopSystem : MonoBehaviour
{
    public void BuyItem(ItemData item, int price)
    {
        if (RemoveCurrency(price))
        {
            InventoryManager inv = FindObjectOfType<InventoryManager>();
            int remaining = inv.AddItem(item, 1);

            if (remaining > 0)
            {
                AddCurrency(price); // Refund if inventory full
                Debug.Log("Inventory full!");
            }
        }
    }

    bool RemoveCurrency(int amount) { /* Implementation */ }
    void AddCurrency(int amount) { /* Implementation */ }
}
```

Equipment System Integration:

```
public class EquipmentSystem : MonoBehaviour
{
    public void EquipWeapon(ItemData weapon)
    {
        if (weapon.itemType == ItemData.ItemType.Weapon)
        {
            // Apply weapon stats to player
            PlayerStats stats = GetComponent<PlayerStats>();
            stats.attackPower += weapon.amountToChangeAttribute;

            Debug.Log($"Equipped {weapon.itemName}");
        }
    }
}
```

8.3 Extension Examples

Creating Custom Item Type:

```
[CreateAssetMenu(fileName = "New Mana Potion",
                 menuName = "Inventory/Mana Potion")]
public class ManaPotion : ItemData
{
    public override bool UseItem()
    {
        GameObject player = GameObject.FindGameObjectWithTag("Player");
        if (player == null) return false;

        ManaSystem mana = player.GetComponent<ManaSystem>();
        if (mana != null && mana.currentMana < mana.maxMana)
        {
            mana.AddMana(amountToChangeStat);
            return true;
        }

        return false;
    }
}
```

Custom Inventory Action:

```
public class CustomInventoryActions : MonoBehaviour
{
    public void SortInventory()
    {
        InventoryManager inv = FindObjectOfType<InventoryManager>();

        // Collect all items
        List<(ItemData data, int qty)> items = new List<(ItemData, int)>();
        foreach (ItemSlot slot in inv.itemSlot)
        {
            if (slot.quantity > 0)
            {
                items.Add((slot.itemData, slot.quantity));
                // Clear slot
                slot.quantity = 0;
                slot.itemData = null;
                slot.itemImage.sprite = slot.emptySprite;
            }
        }

        // Re-add items (will auto-organize)
        foreach (var item in items)
        {
            inv.AddItem(item.data, item.qty);
        }
    }
}
```

9 Conclusion

The Modular Inventory System provides a complete solution for Unity inventory management. Key achievements include:

- **Easy Integration:** Single prefab drag-and-drop integration
- **Designer Friendly:** ScriptableObject workflow requiring zero code
- **Mathematically Optimal:** Intelligent stacking algorithm with O(n) complexity
- **Extensible Architecture:** Clear inheritance and override points
- **Complete Documentation:** Mathematical proofs, code examples, visual guides

The system successfully addresses the main challenges of inventory management while also maintaining flexibility for diverse game genres/mechanics. The included demo health system and preset items give immediate testing capabilities and integration examples.

Future upgrades to the system can include equipment slots, crafting system integration, item durability, and some sort of multiplayer support. The current architecture gives clear extension points for these features without requiring core system modifications.

10 References

1. Mojang Studios. (2011). *Minecraft* [Video game]. Mojang AB.

2. Blizzard Entertainment. (2012). *Diablo III* [Video game]. Blizzard Entertainment.

3. Unity Technologies. (2024). *Unity User Manual: ScriptableObject.* Retrieved from https://docs.unity3d.com/Manual/class-ScriptableObject.html

4. Unity Technologies. (2024). *Unity User Manual: UI System.* Retrieved from https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/index.html

5. Unity Technologies. (2024). *Unity Scripting API: EventSystems.* Retrieved from https://docs.unity3d.com/Packages/com.unity.ugui@1.0/api/UnityEngine.EventSystems.html

6. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. [Greedy algorithms and complexity analysis]

7. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. [Model-View-Controller pattern]

8. Unity Technologies. (2024). *Unity User Manual: Canvas Scaler.* Retrieved from https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/script-CanvasScaler.html

9. Eberly, D. H. (2006). *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics* (2nd ed.). Morgan Kaufmann. [Collision detection algorithms]

10. Unity Technologies. (2024). *Unity User Manual: Physics 2D.* Retrieved from https://docs.unity3d.com/Manual

11. Foley, J. D., van Dam, A., Feiner, S. K., & Hughes, J. F. (1996). *Computer Graphics: Principles and Practice* (2nd ed.). Addison-Wesley. [Coordinate transformations and alpha blending]

12. Unity Technologies. (2024). *Unity User Manual: TextMesh Pro.* Retrieved from https://docs.unity3d.com/Packages/com.unity.textmeshpro@3.0/manual/index.html

13. Nystrom, R. (2014). *Game Programming Patterns.* Genever Benning. Retrieved from https://gameprogrammingpatterns.com/ [Component pattern and data-driven design]