# Inventory and Crafting System Documentation

## Table of Contents

## Overview

The Inventory and Crafting System is a complete solution for managing items, equipment, and crafting in Unity games. It features a flexible and extensible architecture that can be easily integrated into any project.

### Key Features

- Drag and drop inventory management

- Equipment system with different slot types

- Crafting system with recipe management

- Auto-save functionality

- Tooltip system

- Context menu for item actions

- Resource gathering system (demo implementation)

- Full source code included

## System Requirements

- Unity 2022.3.52f1 or higher
- No third-party assets required
- TextMeshPro (included with Unity)

## Installation

1. Import the package into your Unity project
2. Ensure TextMeshPro is installed in your project
3. Copy the demo scene from `Assets/Scenes/DemoScene` to get started

### Included Prefabs

The system includes all necessary prefabs for functionality:

- `InventoryPanel`: Main inventory interface
- `EquipmentPanel`: Equipment slot interface
- `CraftingPanel`: Crafting interface and recipe list
- `ItemTooltip`: Item information display
- `ContextMenu`: Right-click action menu
- `NotificationPanel`: In-game notifications

### Demo Content

The package includes a simple demo scene to showcase the system's functionality. The demo demonstrates:

- Basic inventory operations
- Equipment system
- Crafting system
- Resource collection
- Save/Load functionality

Note: The promotional video showcases advanced models and assets that are not included in this package. These were used for demonstration purposes only. The package includes basic prefabs and demo content sufficient to understand and implement the system.

### Demo Scene Setup

1. Open the demo scene from `Assets/Scenes/DemoScene`

2. Press Play to test the following features:

  - Click resources to collect them

  - Open inventory (default key: Tab)

  - Drag and drop items

  - Right-click items for context menu

  - Craft items from available recipes

  - Save/Load system (F5 to save, F9 to load)

## DemoGame Scene NavMesh Setup

The DemoGame scene uses Unity's NavMesh system for character movement. To properly set up the NavMesh in your scene, follow these steps:

### 1. Enable Navigation Window

1. Open Window -> AI -> Navigation

2. This will open the Navigation window where you can configure NavMesh settings

### 2. Set Up NavMesh Surface

1. Select all the ground/terrain objects in your scene that the character should walk on

2. In the Navigation window, go to the "Object" tab

3. Check "Navigation Static"

4. Make sure the object has a collider component (Box Collider, Mesh Collider, etc.)

### 3. Configure NavMesh Settings

1. In the Navigation window, go to the "Bake" tab

2. Adjust the following settings as needed:

  - Agent Radius: 0.5

  - Agent Height: 2

  - Max Slope: 45

  - Step Height: 0.4

### 4. Bake NavMesh

1. Click the "Bake" button at the bottom of the Navigation window

2. A blue overlay will appear on walkable surfaces

3. This indicates where the character can navigate

### 5. Character Setup

1. Make sure your character object has these components:

   - NavMeshAgent component

   - Character Controller component

2. Configure the NavMeshAgent component:

   - Speed: 5

   - Angular Speed: 120

   - Acceleration: 8

   - Stopping Distance: 0.1

### Common Issues

- If the character can't move to certain areas, check if those areas are properly connected in the NavMesh

- If the character gets stuck, try adjusting the Agent Radius or regenerating the NavMesh

- Make sure all walkable surfaces are marked as "Navigation Static"

### Testing NavMesh

1. Enter Play mode

2. Click anywhere on the ground

3. The character should automatically find a path and move to that location

4. Blue lines will show the calculated path in Scene view

## Quick Start

1. Create an empty scene

2. Add the following prefabs to your scene:

   - `InventoryAndCraftingSystem/Prefabs/UI/InventoryCanvas`

   - `InventoryAndCraftingSystem/Prefabs/Managers/GameManager`

3. Set up your camera and player

4. Press Play to test the basic functionality

## System Components

### Core Managers

1. **InventoryManager**
   - Controls inventory operations
   - Required Inspector References:
     - Inventory Slot Prefab
     - Draggable Item Prefab
     - Slots Parent Transform
   - Key Settings:
     - Inventory Size
     - Auto Sort

2. **EquipmentManager**
   - Manages equipment slots and item equipping
   - Required Inspector References:
     - Equipment Slots List
     - Draggable Item Prefab

3. **CraftingManager**
   - Handles recipe management and crafting operations
   - Required Inspector References:
     - Recipe List
     - Crafting Panel
   - Key Settings:
     - Default Crafting Time

4. **SaveManager**
   - Manages save/load operations
   - Key Settings:
     - Auto Save Interval
     - Use Binary Format
     - Use Auto Save

### UI Components

1. **InventorySlot**
   - Represents a single inventory slot
   - Required Components:
     - Image
     - Button
   - Optional Components:
     - Tooltip

2. **EquipmentSlot**
   - Represents an equipment slot
   - Inspector Settings:
     - Slot Type (Helmet, Chest, etc.)

3. **CraftingPanel**
   - Displays available recipes and crafting progress
   - Required References:
     - Recipe List Content
     - Recipe Button Prefab

4. **ItemContextMenu**
   - Right-click menu for item actions
   - Required References:

   - Button prefabs for actions

## Creating Items

### Creating a New Item

1. Right-click in the Project window
2. Select Create > ScriptableObjects > Items > ItemData
3. Fill in the item details:
   - Name
   - Description
   - Icon
   - Max Stack Size
   - Item Type
   - Equipment Slot Type (if equippable)
   - Crafting Category (if craftable)

### Item Properties

- **isStackable**: Can multiple items stack in one slot
- **isEquippable**: Can the item be equipped
- **equipmentSlotType**: Which slot type this item can be equipped to
- **maxStackSize**: Maximum number of items in one stack

## Creating Recipes

### Creating a New Recipe

1. Right-click in the Project window
2. Select Create > ScriptableObjects > Crafting > RecipeData
3. Configure the recipe:
   - Output Item
   - Output Amount
   - Crafting Time

  - Required Ingredients

  - Category

### Recipe Properties

- **recipeName**: Display name of the recipe

- **craftingTime**: Time in seconds to craft

- **ingredients**: List of required items and their amounts

- **outputItem**: The item produced

- **outputAmount**: Number of items produced

- **category**: Recipe category for UI organization

## UI System

### Tooltip System

- Add TooltipTrigger component to any UI element

- Configure tooltip content in the inspector

- Supports dynamic content via code

### Context Menu

- Right-click on items for actions

- Customizable actions based on item type

- Built-in actions:

  - Use

  - Drop

  - Split Stack

  - Equip/Unequip

## Save System

### Save Data

- Inventory contents

- Equipment loadout
- Crafting progress
- Unlocked recipes

### Save Settings

- Auto-save interval
- Binary/JSON format
- Save file location
- Backup system

### Save Methods

```csharp
// Manual save
SaveManager.Instance.SaveGame();

// Manual load
SaveManager.Instance.LoadGame();

// Delete save
SaveManager.Instance.DeleteSaveFile();
```

## Demo Game

### Resource Collection System

1. Add ResourceNode component to collectible objects
2. Configure:
   - Resource Item
   - Resource Amount
   - Respawn Time
   - Collection Range

### Player Setup

1. Add PlayerController component

2. Add PlayerResourceCollector component

3. Configure NavMeshAgent settings

## Troubleshooting

### Common Issues

1. **Items not appearing in inventory**
   - Check if ItemData is properly configured
   - Verify InventoryManager references
   - Ensure proper prefab setup

2. **Crafting not working**
   - Verify recipe ingredients
   - Check CraftingManager setup
   - Confirm recipe references

3. **Equipment not equipping**
   - Check item's equipmentSlotType
   - Verify EquipmentManager setup
   - Confirm slot type matches

### Debug Mode

- Enable debug logs in each manager
- Use Unity's Debug mode for additional information
- Check console for error messages

## Implementation Guide

### Basic Setup Tutorial

1. **Scene Setup**

```
- Create a new scene
- Add a Main Camera
- Create an empty GameObject named "GameManager"
- Add required manager components to GameManager:
  - InventoryManager
  - EquipmentManager
  - CraftingManager
  - SaveManager
```

2. **UI Canvas Setup**

```
- Create a new Canvas (UI > Canvas)
- Set Canvas settings:
  - Render Mode: Screen Space - Overlay
  - UI Scale Mode: Scale With Screen Size
- Add the following UI prefabs as children:
  - InventoryPanel
  - EquipmentPanel
  - CraftingPanel
  - NotificationPanel
  - ItemTooltip
```

### Adding New Items

1. **Creating Item Data**

```
```

   - In Project window: Right-click > Create > ScriptableObjects > Items > ItemData
   - Configure basic properties:
     - Item Name
     - Description
     - Icon
     - Max Stack Size
   - For equipment items:
     - Check "Is Equippable"
     - Set Equipment Slot Type
   - For craftable items:
     - Set Crafting Category
     - Configure recipe requirements
   ```

2. **Example: Creating a Sword Item**
   ```
   - Create new ItemData
   - Set Name: "Iron Sword"
   - Set Icon: (drag sword sprite)
   - Set Max Stack Size: 1
   - Check Is Equippable
   - Set Equipment Slot Type: Weapon
   - Set Item Type: Weapon
   ```

### Adding New Recipes
1. **Creating Recipe Data**
   ```
   - Create > ScriptableObjects > Crafting > RecipeData
   - Set Output Item (drag ItemData)
   - Add Ingredients:

    - Click "+" to add ingredient
    - Drag required ItemData
    - Set amount needed
  - Set Crafting Time
  - Set Recipe Category
  ```

2. **Example: Sword Recipe**
  ```
  - Output Item: Iron Sword
  - Ingredients:
    - Iron Ingot x2
    - Wood x1
  - Crafting Time: 3.0
  - Category: Weapons

### Implementing Custom Item Behavior
1. **Custom Item Actions**
  ```csharp
  // Create a new script inheriting from ItemData
  public class UsableItem : ItemData
  {
      public override void Use()
      {
          // Add custom use behavior
          Debug.Log($"Using item: {itemName}");
      }
  }
  ```

2. **Custom Equipment Effects**

```csharp
public class EquippableItem : ItemData
{
    public override void OnEquip()
    {
        // Add equip effects
        Debug.Log($"Equipped: {itemName}");
    }

    public override void OnUnequip()
    {
        // Add unequip effects
        Debug.Log($"Unequipped: {itemName}");
    }
}
```

### Resource Collection System

1. **Setting Up Resource Nodes**

```
- Create an empty GameObject
- Add ResourceNode component
- Configure:
  - Resource Item (ItemData)
  - Resource Amount
  - Respawn Time
  - Collection Range
- Add a collider (must be trigger)
- Add visual mesh/sprite
```

2. **Player Resource Collection**

  ```
  - Add PlayerResourceCollector to player
  - Configure:
    - Collection Range
    - Collection Layer
    - Collection Key (default: Mouse0)
  ```

### Save System Integration

1. **Basic Save/Load**

  ```csharp
  // Add to your game manager
  private void SaveGame()
  {
      SaveManager.Instance.SaveGame();
  }

  private void LoadGame()
  {
      SaveManager.Instance.LoadGame();
  }
  ```

2. **Auto-Save Configuration**

  ```
  - Select SaveManager in inspector
  - Enable Use Auto Save
  - Set Auto Save Interval (seconds)
  - Optional: Enable Use Binary Format
  ```

### Common Integration Scenarios

1. **Adding Items Through Code**
  ```csharp
  // Give player an item
  public void GiveItem(ItemData item, int amount)
  {
      InventoryManager.Instance.AddItem(item, amount);
  }
  ```

2. **Checking Inventory Space**
  ```csharp
  // Check before adding items
  if (InventoryManager.Instance.HasSpaceForItem(item, amount))
  {
      InventoryManager.Instance.AddItem(item, amount);
  }
  ```

3. **Equipment Checks**
  ```csharp
  // Check if item is equipped
  public bool IsItemEquipped(ItemData item)
  {
      return EquipmentManager.Instance.GetAllSlots()
          .Any(slot => slot.CurrentItem == item);
  }
  ```

4. **Custom Crafting Conditions**
   ```csharp
   // Add custom crafting requirements
   public bool CanCraftItem(RecipeData recipe)
   {
       bool hasIngredients = CraftingManager.Instance.HasRequiredIngredients(recipe);
       bool hasRequiredLevel = PlayerLevel >= recipe.requiredLevel;
       return hasIngredients && hasRequiredLevel;
   }
   ```

### UI Customization
1. **Modifying Slot Appearance**
   ```
   - Select slot prefab
   - Modify in inspector:
     - Background image
     - Border
     - Highlight color
     - Selection effect
   ```

2. **Custom Tooltip Content**
   ```csharp
   public class CustomTooltip : ItemTooltip
   {
       protected override string GetTooltipText(ItemData item)
       {
           return $"{item.itemName}\n{item.description}\nCustom Info: {GetCustomInfo(item)}";
       }
   ```

  }
  ```

### Troubleshooting Guide
1. **Items Not Appearing**
   - Check if ItemData is in Resources folder
   - Verify InventoryManager references
   - Check console for errors
   - Ensure proper prefab setup

2. **Crafting Issues**
   - Verify recipe ingredients exist
   - Check ingredient amounts
   - Confirm CraftingManager setup
   - Verify recipe references

3. **Equipment Problems**
   - Check slot type matches
   - Verify item is equippable
   - Check equipment slot setup
   - Confirm drag-drop setup

4. **Save System Issues**
   - Check write permissions
   - Verify save location
   - Enable debug logs
   - Check file format settings

### Performance Optimization
1. **Large Inventories**
   - Use object pooling for slots
  ```

  - Implement virtual scrolling
  - Optimize icon loading
  - Cache frequently used items

2. **Multiple Crafting Queues**
  - Use job system for crafting
  - Implement queue management
  - Optimize progress updates

### Best Practices
1. **Code Organization**
  - Keep managers in separate scene
  - Use events for updates
  - Implement proper error handling
  - Follow singleton pattern

2. **Resource Management**
  - Organize items by category
  - Use proper naming conventions
  - Implement proper cleanup
  - Handle scene transitions

## Support and Updates
For support or questions, please contact:

[Your Contact Information]

## Version History
- 1.0.0: Initial release
- [Future versions will be listed here]