Department of Mechanical and Mechatronics Engineering

Design and Optimization of a Modular Crafting System in Unity Using Scriptable Objects

A Report Prepared For:

The University of Waterloo

Prepared By:

Grace Dice

150 University Ave W.

Waterloo, Ontario, N2N 2N2

Sept 22, 2025

150 University Ave W.
Waterloo, Ontario, N2N 2N2

Sept 22, 2025

Prof William Owen
Associate Chair, Undergraduate Studies
Department of Mechanical and Mechatronics Engineering
University of Waterloo
200 University Ave W.
Waterloo, Ontario, N2L 3G1

Dear Professor Owen,

This report was written to fulfill my 2B work term requirement and is a self-directed project that was done during my role as an ME101 first-year tutor at the University of Waterloo. This is my 2nd work term report submission.

During the time I spent in that role, I assisted in teaching C++ and RobotC. The curiosity of the students and my own creativity caused me to be inspired to create my own coded project, in which I came upon the idea to design and create a modular crafting system in Unity using C# and Scriptable Objects.

The system itself includes a dynamic 1x5 grid to support crafting recipes, an inventory that allows for stackable/non-stackable items, and an event-driven UI that gives real-time, visual feedback to the user. Applying engineering principles such as the Single Responsibility Principle and performance optimization gave a system that can be expanded without modifying the core code. This is in terms of creating recipes or even creating more items and visual entities. The end outcome produced a scalable prototype of a gaming crafting system that was tested to handle over 25 recipes and keep low and efficient frame times.

This report in its entirety was written by me and has not received any prior academic credit at this or any other institution. This report was aided by the insights gained from watching Game Maker's Toolkit videos on YouTube [1]. Information about Unity and its structure and the basics of game creation was taken from these videos, yet the coding itself was informed by my background in coding. Three years of previous experience in Java, Javascript, C++, and C# were used to design this system.

I hope this report provides a clear and insightful overview of my work,

Grace Dice
ID: 21066386
2B Mechanical Engineering

Table of Contents

List of Figures

List of Tables

Summary

Using Scriptable Objects and Unity, this report covers the design of a modular and optimized crafting system. The project itself addresses a need for a flexible prototype crafting system for games as seen in RPG, survival, and simulation games.

The system allows players to combine materials in a 1x5 grid and create outputs based on predefined recipes. The basis of this is supported by the inventory, which holds 8 slots of materials up to 16 items or only 1 for those that are non-stackable. An event-driven user interface (UI) is triggered throughout creation to update the visuals and give real-time feedback, along with extensibility to add recipes and items without touching any code.

The system architecture is created in three distinct layers: the data layer using Scriptable Objects for items and recipes, the logic layer handling crafting and inventory management, and the UI layer that provides interactive visuals. Design choices such as 1D arrays for fast access and event-driven UI were evaluated along with the design structure through tradeoffs and benchmarks using Unity's Profiler. The overall testing spanned over grid mismatches, stack merging, UI consistency, and edge cases like empty grids in the crafting process.

Results for the report include the ability to support at minimum 25 recipes with correct, correctly output crafting results with frame times staying close to 10.53 ms on average, cleanly update the UI, manage an 8-item inventory, and produce scalable performance metrics.

1.0 Introduction

Almost every role-playing (RPG), survival, and simulation game includes a core mechanic, which is the crafting system. It allows players to combine raw materials into new or more useful items and spans over a range of complexity and design throughout games. Every one needs a framework that supports dynamic evaluation of inputs, feedback responsiveness, and extensibility. The creation of these systems not only involves creative design but also the disciplined engineering principles to function at a higher capacity.

This project explores how major core engineering principles such as modularity, optimization, and testing can be applied in a digital software environment. It leverages Unity's built-in dependencies and improves flexibility and reusability across gameplay systems [2]. Throughout development, Unity's Profiler tool was used to analyze the performance and responsiveness of the system by assessing runtime behavior [3].

The goal was not only to build a working prototype but also to create a scalable and efficient representation of best practices in gaming software architecture.

1.1 Problem Definition

The goal of this project is to develop a flexible, modular crafting system in Unity while being capable of evaluating combinations of items arranged in a 1x5 grid. The system must determine whether a recipe is valid and provide immediate feedback on the outcome. Additions to this include an inventory that supports both stackable and non-stackable items and extensibility features that allow for new recipes to be added without modifying any code.

The system will be built using Scriptable Objects to define recipes as modular, asset-based data units. Doing this allows for easy recipe management through the Unity editor, and logic and data are able to remain recoupled (a principle aligned with the Single Responsibility Principle in software design) [4]. By avoiding rigid and hard-coded definitions, the system is able to remain adaptable and maintainable as new crafting content is introduced.

Using the Profiler in Unity, the system will be tested for memory and CPU performance under various conditions and eliminate any runtime efficiency concerns, with the ultimate goal being to create a crafting engine that minimizes overhead, avoids redundant operations, and scales with growing complexity.

1.2 Project Context

This project was undertaken during the Spring 2025 term, during the time of working as an ME101 tutor. The project was inspired by curiosity and the desire to explore game development after interaction with tutoring first-year students in coding.

The crafting system is both a technical challenge and a creative endeavor by building it from the ground up in C# using Unity. The aim was for the integration of design practices into a domain often associated with rapid prototyping and artistic iteration and to showcase how engineering methodologies (like modular architecture, performance analysis, and scalable design) can enrich software development within creative industries. Utilizing a mechanical engineering background, there is the ability to apply the same analytical rigor to a complex software problem.

2.0 System Architecture and Design Rationale

The crafting system uses Scriptable Objects to achieve a more flexible and maintainable game design structure and is structured into four primary modules: the data layer, the logic layer, the UI layer, and the player interaction layer. All of these were made to adhere to the Single Responsibility Principle to allow each component to handle its own specific function and remain decoupled from the others [4]. This gives better maintainability and also aids in future extensions, such as adding new recipes or more UI features. Full code implementation for these classes can be found in the Appendix for reference.

The architecture diagram (Figure 1) expresses the flow of data and the interactions between each layer, with the data layer feeding into the logic layer, which in turn drives the UI layer, all connected through the player interactions. As the foundation, the data layer uses Scriptable Objects for item and recipe definitions and gives a centralized editor-configurable repository for game assets. The logic layer comprises the CraftingSystem and Inventory classes; it handles the core mechanics of recipe matching and item management. The UI layer consists of the UI_CraftingSystem and UI_Inventory classes and gives real-time visual feedback to the user, and the player interaction layer integrates input handling and environmental interactions.

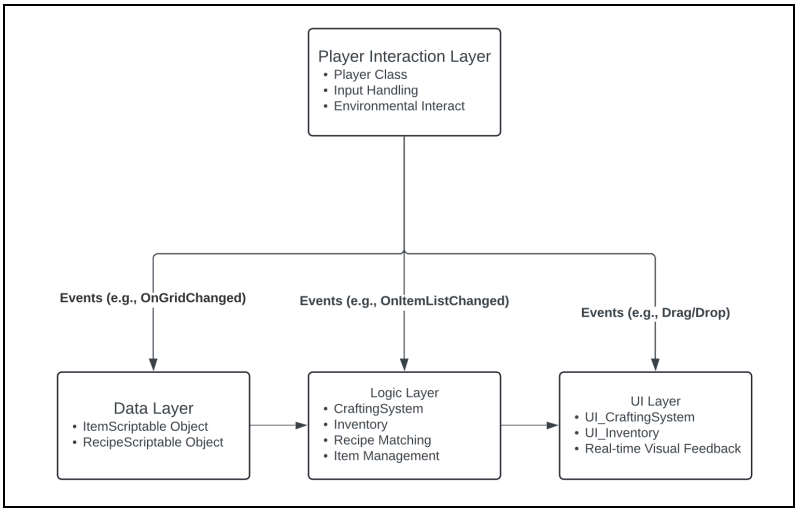

Figure 1. Code Architecture Diagram

2.1 Scriptable Objects as Data Models

Scriptable Objects are used as the primary data model and are implemented in the ItemScriptableObject and RecipeScriptableObject classes. These editor-configurable assets give designers the ability to create and modify game components without any need to edit the code. Each ItemScriptableObject defines properties like itemName (string), itemSprite (Sprite), and isStackable (boolean), which determine its identity, visual representation, and stacking behavior. Similar to this, RecipeScriptableObject contains the reference information for an array of five ItemScriptableObjects of the designer's choice and one for the output item.

The use of Scriptable Objects gives the benefits of their shared reference model and reduces memory usage by each game object referencing the same asset instance instead of duplicating data in memory. Their editor-friendly nature allows non-programmers to modify any game content without coding knowledge and is easier for collaboration in team settings. Lastly, this data model supports serialization and seamless save and load functionality for game states. By decoupling data from this behavior, it eliminates any need for hard-coded dependencies such as the quote below and aligns with modularity principles.

```
itemArray = new Item[1, GRID_SIZE];
biscottiRecipe = new Item.ItemType[1, GRID_SIZE];
biscottiRecipe[0,0] = Item.ItemType.Egg;
biscottiRecipe[0,1] = Item.ItemType.Honey;
biscottiRecipe[0,2] = Item.ItemType.Almond;
```

```
biscottiRecipe[0,3] = Item.ItemType.WheatFlour;
biscottiRecipe[0,4] = Item.ItemType.None;
```

The impact this has on the system is that it stays maintainable, adaptable, and capable of supporting new items and recipes without any code alterations.

2.2 Crafting Grid and Recipe Matching

The crafting system is designed as a 1x5 grid and is coded as a one-dimensional itemArray in the CraftingSystem class to allow linear item placement and recipe evaluation. Each inputted item is compared against the ingredients defined in each RecipeScriptableObject to determine the correct output. The recipe matching process iterates through the recipeList and checks for exact matches between the grid's itemArray and the recipe's Ingredients array; upon successful matching, the system generates the output item and clears the grid to prepare for the next crafting attempt.

As shown in the recipe matching logic flowchart below (Figure 2), the logic for recipe matching is quite straightforward and efficient. The process begins with looping through the recipeList and comparing each recipe's ingredients to the items in the itemArray. If a match is found, then the output is set and the grid is cleared; otherwise, outputItem is set to null and the OnGridChanged event is invoked to update the UI.

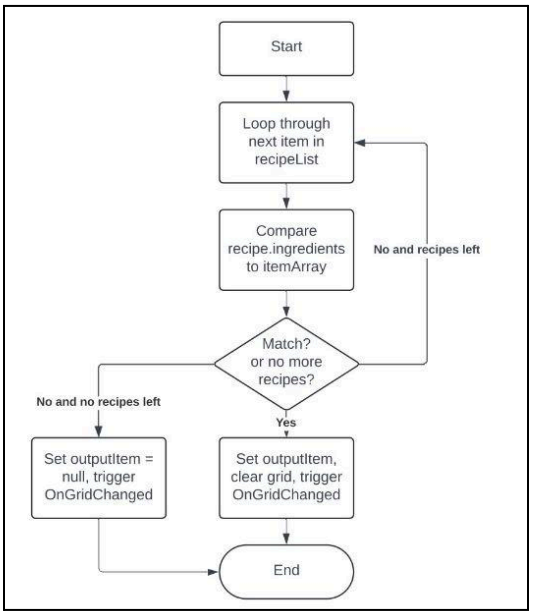

Figure 2. Recipe Matching Logic Flowchart

2.3 Event-Driven Updates

To provide real-time feedback, the crafting system uses an event-driven architecture based on the Observer Pattern [5]. There are two primary events: OnGridChanged in the CraftingSystem class and OnItemListChanged in the Inventory class, which both drive updates to the UI layer. Whenever an item is added, removed, or modified in the crafting grid, the OnGridChanged event is triggered; this is through the SetItem or RemoveItem methods. The same structure and logic are used and applied to the OnItemListChanged.

Both the UI_CraftingSystem and UI_Inventory classes subscribe to these events to update either of their respective visuals when necessary. This eliminates the need for continuous polling. For example, the UI_CraftingSystem class listens for OnGridChanged to refresh the crafting grid and output slot visuals and supports the drag-and-drop functionality through the IDropHandler interface. In parallel to this, the UI_Inventory uses the same structure but with OnItemListChanged to update its 2x4 grid while making sure that the item quantities are correctly reflected in the UI. Overall, this approach reduces any CPU overhead, because updates occur only in response to specific actions, and allows for better performance efficiency [6].

3.0 Component Descriptions and Code Flow

This section gives an overview of the key components in the crafting system, describing their roles, fields, methods, and interactions. The class diagram (Figure 3) shows each relationship between these components, including ItemScriptableObject, RecipeScriptableObject, CraftingSystem, Inventory, Player, and UI_CraftingSystem, as well as their dependencies and event-driven communication. By following the Single Responsibility Principle, each class focuses on a specific function, and no overlap happens.

The code begins with the data defined in Scriptable Objects, which are then processed by the logic layer to handle any crafting and item management. These all then communicate change to the UI layer via events and give real-time feedback to the user. The Player class integrates all of these systems together and handles user input and environmental interactions. The separation of each piece allows for independent development and testing of each function/class.

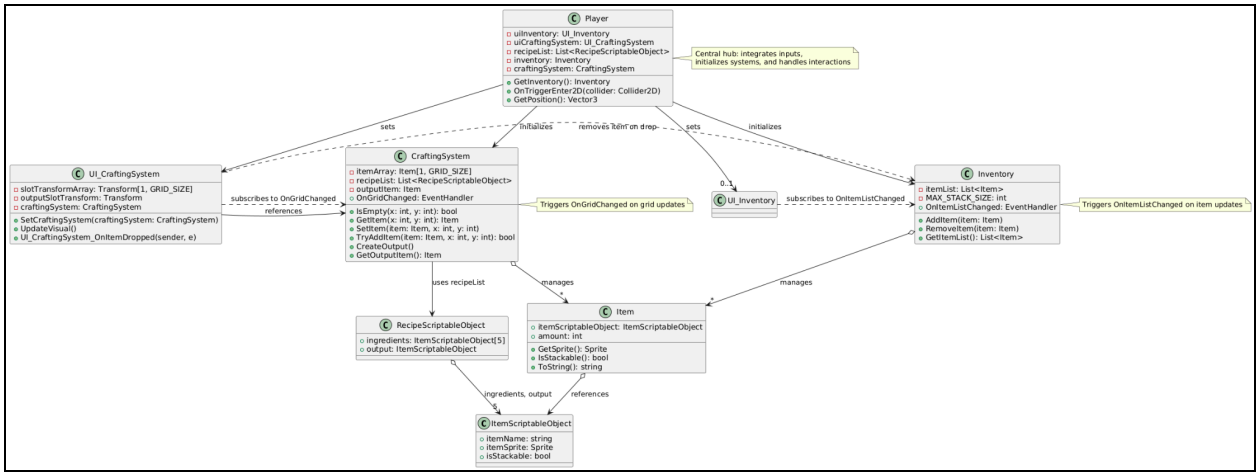

Figure 3. Unified Modeling Language (UML) Class Diagram of Modular Crafting System

3.1 ItemScriptableObject

This class is the data model for defining item properties within the Unity Editor. Its main purpose is to encapsulate item-specific attributes and to allow designers to create items without touching the code. The class includes the following fields:

- itemName (string): specifies the item's names, used for identification and display.
- itemSprite (Sprite): defines the visual representation of the item in the UI.
- isStackable (boolean): determines whether the item can be stacked in the inventory, affecting how quantities are managed.

Those fields are used to create assets in the editor, as can be seen below in Figure 4.

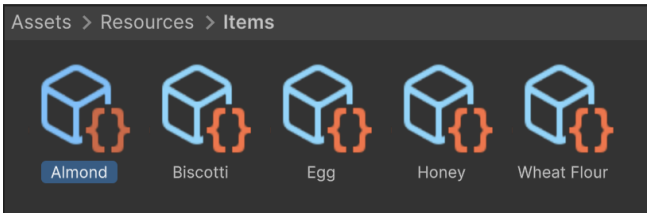

Figure 4. ScriptableObject Assets for Item Definitions in Unity

This class is referenced by other components in the system, such as the Inventory and CraftingSystem to access item properties during the runtime. For example, the GetSprite method in the Item class retrieves the itemSprite from the associated ItemScriptableObject. For UI rendering. By creating a centralized

location for all the data, the system reduces memory usage through the shared references and supports serialization, which aligns with the goals defined in Section 2.1.

3.2 RecipeScriptableObject

This class defines crafting recipes and encapsulates the data required to map a set of input items to an output item. Its purpose is to provide a flexible and configurable structure for recipes, enabling the adoption of new recipes without any code altering. The class includes the following fields:

- Ingredients (array of 5 ItemScriptableObjects): an array specifying up to five input items required for the recipe.
- Output (ItemScriptableObject): the resulting item produced when the recipe is matched.

These fields are configured in the Unity Editor and stored in a recipeList within the CraftingSystem class. This is used during the recipe matching process, where the CraftingSystem compares the ingredients to the items in the crafting grid. This data-driven approach eliminates any hard-coded recipes and allows for scalability.

3.3 Inventory Class

This class manages the player's collection of items, supporting both stackable and non-stackable items in a dynamic list. Its purpose is to handle item addition, removal, and querying while maintaining inventory state. Key methods include:

- AddItem: adds an item to the inventory, merging quantities for stackable items if they already exist.
- RemoveItem: removes items and updates quantities, ensuring proper state management.
- GetItemList: returns the current list of items for UI display or other components.
- OnItemListChanged: an event triggered whenever the inventory changes, notifying subscribers like the UI_Inventory class.

This class uses a dynamic list to support flexible inventory sizes, currently limited to eight slots for prototyping. This design supports the system's requirement for handling both stackable and non-stackable items and integrates with the event-driven UI updates described in Section 2.3.

3.4 CraftingSystem Class

This class is the core logic component, managing the 1x5 crafting grid and performing recipe matching. Its purpose is to process player inputs in the grid and generate crafting outputs based on predefined recipes. The class includes the following fields:

- itemArray: a 1D array representing the 1x5 grid storing Item objects
- recipeList: a list of RecipeScriptableObject instances containing all valid recipes.
- outputItem: the resulting item produced from a successful recipe match.

Key methods include:

- isEmpty: checks if a grid slot is empty, returning true if itemArray[x] is null.
- GetItem: retrieves the item at a specified grid index.
- SetItem: places an item in the grid and triggers the CreateOutput method.
- CreateOutput: iterates through recipeList to match itemArray against recipe ingredients, setting outputItem and clearing the grid on success.
- OnGridChanged: an event triggered after grid modifications to update the UI.

The CraftingSystem ensures efficient recipe matching by using a 1D array for fast index-based access, as discussed in Section 2.2.

3.5 Player Class

This class integrates the crafting and inventory systems with the player interactions and handles movement and item pickup. It serves as the central hub for user-driven actions and connects the logic layer to the game world. Key functions include:

- Initialization: sets up references to the CraftingSystem and Inventory instances during the Start method.
- Item loading: loads all ItemScriptableObject assets from the Resources folder using loadAllItemsFromResources.

- Pickups: handles item collection via OnTriggerEnter2D, adding items to the inventory when the player collides with ItemWorld objects.
- Movement: processes input using Unity's Input System, updating the player's position via a Rigidbody2D component with movespeed set to 3f.

The class allows for easy integration of crafting and inventory mechanics with gameplay.

3.6 UI_CraftingSystem Class

This class manages the visual representation of the crafting grid and output slot, giving an interactive interface for players. It is to display the state of the CraftingSystem and handle drag-and-drop interactions. Key functions include:

- Event listening: subscribes to the OnGridChanged event to update the grid visuals whenever the CraftingSystem changes.
- Drag-and-drop: implements the IDropHandler interface via OnDrop to process dropped items, updating the grid via CraftingSystem.SetItem.
- Visual updates: uses UpdateVisual to clear and repopulate the grid with instantiated pfUI_Item prefabs, aligning them to slotTransformArray positions.

The class initializes a slotTransformArray to manage the 1x5 grid layout, locating slots via gridContainer.Find("grid_" + x + "_" + y). It displays the item quantities and the output item. This approach eliminates polling, enhancing performance as validated in Section 4.2, and supports the system's requirement for real-time UI feedback (Section 1.1).

4.0 Engineering Analysis

The purpose of this section is to go over the design and performance of the system and ensure it meets the required objectives of flexibility, scalability, and efficiency as outlined in Section 1.1. By analyzing design choices, benchmarking runtime performance, and testing the system, this analysis validates the system's adherence to engineering principles. There are three main areas of focus: design choices and trade-offs, performance analysis, and testing and debugging to ensure reliability.

4.1 Design Choices and Tradeoffs

The crafting system's architecture was formed around several design decisions, each evaluated against alternative approaches to balance the functionality, performance, and maintainability. Table 1 summarizes these choices, their options, the selected approach, and the reason behind each decision.

Table 1. Design Decision Matrix for Modular Crafting System

Aspect	Options	Chosen	Reason
Data Storage	Dictionary, Array	1D Array	Fast index-based access, simpler implementation for a 1x5 grid.
Recipe Storage	JSON, Code, Scriptable Objects	Scriptable Objects	Editor-friendly, modular, supports serialization, and reduces hard-coding.
UI Refresh	Polling, Events	Event-Driven	Better performance by eliminating continuous polling.
Item Comparison	Name, SO Reference	SO Reference	Less error-prone, it allows type safety and consistency using Unity's asset references.

Data Storage: a 1D array was chosen for the itemArray in the CraftingSystem class over using a dictionary due to its fast index-based access. Dictionaries, while offering flexible key-based lookups, introduce overhead that isn't necessary for a fixed-size grid [7].

Recipe Storage: Scriptable Objects were selected over JSON or hard-coded code for their editor-friendly interface, which gives designers the ability to create and modify recipes without coding. This choice also supports serialization for save/load functionality.

UI Refresh: An event-driven approach was chosen to eliminate any need for polling, reducing CPU usage and allowing for updates only when needed.

Item Comparison: comparing items by using ItemScriptableObject references instead of using string names guarantees type safety and lowers errors, as string comparisons are more likely to have errors or naming inconsistencies

All of these choices were made to create a system that is performant, maintainable, and aligned with the project goals.

4.2 Performance Analysis

To evaluate the efficiency and scalability of the crafting system, performance benchmarks were conducted using Unity's Profiler window. The tests focused on measuring the frame time required for recipe matching under changing recipe list sizes. Both the original hard-coded configuration and the refactored Scriptable-Object-based configuration were profiled under identical conditions, creating the same recipe and outcome. In Table 2, each data point represents the average result of the independent runs to ensure consistency and reduce variability, except where additional analysis was necessary to understand runtime instability.

Table 2. Comparison of Hardcoded vs. Scriptable Recipe Matching Performance.

Number of Recipes	Old Configuration (ms)	New Configuration (ms)
1	5.88	-
4	6.43	6.33
7	*11.27 (see Table 3)*	*6.13 (see Table 3)*
10	-	6.87
16	-	8.98
20	8.46	-
23	-	10.53
30	9.53	-

The results indicate that the scriptable configuration consistently outperforms the hard-coded version, especially as the recipe increases in numbers. While the old configuration exhibits irregular performance spikes, the new configuration maintains a smoother and more predictable growth in frame time, as seen in the above table.

To further analyze the performance consistency, six profiling runs were executed for each configuration at 7 recipes, as can be seen in Table 3:

Table 3. Observed Frame Time Variability in Old Configuration at 7 Recipes.

Run #	Old Frame Time (ms)	New Frame Time (ms)
1	9.03	7.32
2	17.56	5.72
3	7.11	6.68
4	8.75	6.96
5	11.51	5.81
6	11.66	6.26
Average	**11.27**	**6.13**
Standard Deviation	**±3.68**	**±0.64**

This data reinforces that the new system is superior, and not only is the performance better, but the variance is lower. The old configuration shows a significant amount of instability, with frame times swinging from 7.11 ms to 17.56 ms, while the scriptable system stays tightly grouped around its mean. The old configuration's instability is likely due to inefficient array handling and redundant checks in the hard-coded recipe logic. The Scriptable Object's data reflects a more optimized design and overall validates the system's scalability and performance efficiency and supports larger recipe sets.

4.3 Testing, Debugging, and Challenges

To ensure the crafting system's robustness, a testing strategy was created, covering functional correctness, edge cases, and performance stability. The following scenarios were tested to verify system behavior:

- Grid Mismatch: 15 tests confirmed that the CraftingSystem.CreateOutput method produces no false positives when incorrect ingredients are placed in the grid. For example, mismatched ItemScriptableObject references result in a null outputItem, showing the accurate recipe matching.

- Stackable Items: The Inventory.AddItem method correctly merges stackable items by comparing ItemScriptableObject references and incrementing the amount field. This prevents any duplicate entries for items like “Honey” with isStackable set to true, which can be seen below.

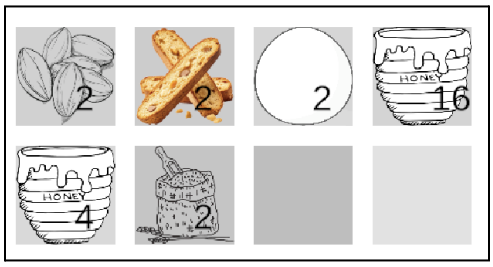

Figure 5. In-game Inventory With Max 16 Stacked Items

- UI Consistency: both UI classes accurately reflect every change triggered by OnGridChanged and OnItemListChanged events, and the visuals update in real-time without any discrepancies.

- Edge Cases: Tests covered scenarios such as empty grids, overstacked items (exceeding 16 items), and null references. As an example, UI_CraftingSystem. OnDrop prevents crashes when invalid items are dropped. Similarly, in the DragDrop class, if an item is dragged into blank space or no slot, it will go back to its previous spot.

Testing was done by using tools such as Unity’s Debug.Log for runtime diagnostics and the Profiler for detecting any memory leaks or performance bottlenecks. After checking many scenarios and achieving correct recipe matching and outputs, no memory leaks were seen, and the event-driven architecture consistently updated the UI. All of these things confirm that the system is reliable enough and ready for use as a functional prototype and meets all objectives outlined in Section 1.1.

During the creation process, there were a lot of challenges with the drag-and-drop functionality. From items duplicating themselves after being used in crafting to items failing to be registered when dropped onto the crafting grid. The duplication was due to the object not getting deleted from the inventory and only the UI updating, and not the actual code. This caused the item to stay in the inventory unknowingly and also be used in the crafting process, allowing a duplication of that item. It was fixed by deleting that item used from the inventory, not only the UI. Another problem was items failing to register in the grid after they were dropped. This led to inconsistencies in the Inventory and the UI_Inventory states and stemmed from the triggering of the OnGridChanged, and debug logs were added to make sure data passed to CraftingSystem.TryAddItem. As a fallback, DragDrop.OnEndDrag was created to return the item back to its original slot if the attempted drop isn’t valid.

5.0 Results

The development of the modular crafting system in Unity yielded a prototype that successfully demonstrates the application of engineering principles to game development, as shown by the system's performance, functionality, and scalability. The outcomes highlight the system's ability to handle the crafting mechanics while keeping its efficiency and extensibility, meeting the object set in Section 1.1. Specifically, the system supports recipe matching for over 25 configurations, producing correct outputs based on the defined RecipeScritableObject ingredients and grid placements. The UI integration is seamless, with drag-and-drop functionality, allowing players to interact with items without disruption. Real-time updates are achieved by event-driven mechanisms and allow the changes to be seen immediately without polling. Inventory management effectively supports both stackable and non-stackable items, with logic in the Inventory class to merge stacks up to a maximum of 16 units and assign slots in a 2x4 grid. Scalability is shown by adding over 25 recipes without modifying the core code, and new recipe assets can be created in the Unity Editor and loaded into a recipe list for matching.

The metrics collected during testing further validate the system's efficiency. Frame time remains low and predictable, with the new configuration showing a standard deviation of 0.4 ms compared to the 3.68 of the hard-coded version. Memory usage is optimized through shared Scriptable Object references, with no leaks detected in the Profiler during prolonged sessions involving crafting and UI updates. Overall, these metrics confirm the system's ability to handle increased complexity without significant degradation, supporting its use in future larger game projects.

This achievement marks the successful completion of all project objectives, creating a functional prototype that serves as the foundation for crafting systems in RPG, survival, or simulation games. The prototype not only meets the original requirements but also has the ability to scale up and be a reusable component for future games and self-directed learning in game engineering.

6.0 Conclusions

The modular crafting system developed in this project demonstrates a successful application of engineering principles to game development, all coming together into a scalable prototype that achieves the objectives outlined in Section 1.1. By using Unity's Scriptable Objects and C# scripting, the system shows the integration of modularity, optimization, and testing methodologies, resulting in a framework for a crafting system in video games.

6.1 Conclusions

This project has delivered a modular and extensible crafting system, created by the use of Scriptable Objects to decouple data from behavior. Key features of the system include the use of Scriptable Objects for defining item properties (itemName, itemSprite, and isStackable) and recipes (ingredients array and output), all reducing any hard-coding in the main classes and supporting serialization for save/load functionality. Event-driven updates allowed for real-time UI updates, which were facilitated by the Observer Pattern through the OnGridChanged and OnItemListChanged events. The separation of all of the concerns is shown through the layered architecture: the data layer (Scriptable Objects), the logic layer (CraftingSystem and Inventory), and the UI layer (UI_CraftingSystem and UI_Inventory).

Engineering principles were followed through the process, including the design to be modular, optimizing the system by event-driven code, using 1D arrays for fast access, and testing against any edge cases like grid mismatches or null references.

The outcome gained is a scalable prototype that validates game mechanics, easily supporting over 25 recipes with low frame times and consistent performance. This work's significance lies in demonstrating self-directed learning, bridging foundational programming skills from ME101 tutoring with practical application in interactive software design, and creating innovation in game development while adhering to disciplined methodologies.

6.2 Next Steps/Recommendations

To take this project further, there are several recommendations to improve future development. A visual recipe editor could be implemented as a custom Unity Editor tool to allow designers to create and modify RecipeScriptableObject assets intuitively through a graphical interface. This would give a more streamlined recipe configuration, reduce the need for manual description, and extend the system's editor-friendly nature.

Object pooling for UI elements, such as item sprites and slot visuals, would reduce memory fragmentation and allocation overhead during frequent updates. Specifically in scenarios where the inventory changes or in drag-and-drop operations. By pre-allocating and reusing UI objects, this would minimize garbage collection pauses and improve runtime performance in larger game sessions [8].

Automated testing using Unity's Test Framework can be used to cover edge cases comprehensively, including empty grids, overstacked items, null references, and recipe mismatches. Integrating unit testing for key methods would give regression-free updates and create a more reliable system as it scales.

Finally, for optimizing matching in large recipe lists, transitioning to a dictionary-based lookup in the CraftingSystem could replace the current approach and reduce time complexity for recipe checks. This would be helpful for systems with hundreds of recipes to maintain low frame times and support expanded content.

Overall, recommendations position the system for better use and to keep improving the system.

References

[1] Game Maker's Toolkit, "Game Maker's Toolkit," YouTube. Accessed: Sept. 02, 2025. [Online]. Available: https://www.youtube.com/channel/UCqJ-Xo29CKyLTjn6z2XwYAw

[2] Unity, "Unity ScriptableObject," Unity Documentation. Accessed: Aug. 21, 2025. [Online]. Available: https://docs.unity3d.com/6000.2/Documentation/Manual/class-ScriptableObject.html

[3] Unity, "Unity Profiler," Unity Documentation. Accessed: Aug. 21, 2025. [Online]. Available: https://docs.unity3d.com/6000.2/Documentation/Manual/Profiler.html

[4] GeeksforGeeks, "Single Responsibility in SOLID Design Principle," GeeksforGeeks. Accessed: Aug. 21, 2025. [Online]. Available: https://www.geeksforgeeks.org/system-design/single-responsibility-in-solid-design-principle/

[5] GeeksforGeeks, "Observer Design Pattern," GeeksforGeeks. Accessed: Aug. 22, 2025. [Online]. Available: https://www.geeksforgeeks.org/system-design/observer-pattern-set-1-introduction/

[6] S. R. Gothi, "Event-Driven Microservices Architecture for Data Center Orchestration," vol. 16, no. 2, p. 12, June 2025.

[7] T. Juszczak, "List vs Dictionary performance," Prographers. Accessed: Aug. 22, 2025. [Online]. Available: https://prographers.com/blog/list-vs-dictionary-performance

[8] Unity Learn, "Unity Learn Introduction to Object Pooling," Unity Learn. Accessed: Aug. 24, 2025. [Online]. Available: https://learn.unity.com/tutorial/introduction-to-object-pooling

Appendix

DrapDrop Class

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.EventSystems;
using UnityEngine.UI;

public class DragDrop: MonoBehaviour,
IPointerDownHandler,
IBeginDragHandler,
IEndDragHandler,
IDragHandler {
  private RectTransform rectTransform;
  private CanvasGroup canvasGroup;
  private Image image;
  private Item item;

  private Transform originalParent;
  private Vector3 originalPosition;

  private Canvas canvas;

  private void Awake() {
    image = GetComponent < Image > ();
    if (image == null) {
      Transform childImage = transform.Find("image");
      if (childImage != null) {
        image = childImage.GetComponent < Image > ();
      }
      else {
        Debug.LogError("No Image component found on object or its child named 'image'");
      }
    }

    rectTransform = GetComponent < RectTransform > ();
    canvasGroup = GetComponent < CanvasGroup > ();

    canvas = GetComponentInParent < Canvas > ();
    if (canvas == null) Debug.LogError("Canvas not found on parent! DragDrop won't scale drag delta
correctly.");
```

```
  }

  public void OnBeginDrag(PointerEventData eventData) {
    Debug.Log("OnBeginDrag");

    originalParent = transform.parent; // Store original slot transform
    originalPosition = transform.localPosition; // Store original position within that slot
    transform.SetParent(canvas.transform); // Drag on top of UI canvas
    canvasGroup.alpha = 0.6f;
    canvasGroup.blocksRaycasts = false;

    Debug.Log("Begin drag from: " + originalParent.name);
  }

  public void OnEndDrag(PointerEventData eventData) {
    Debug.Log("OnEndDrag");
    canvasGroup.alpha = 1f;
    canvasGroup.blocksRaycasts = true;

    bool droppedOnSlot = eventData.pointerEnter != null && eventData.pointerEnter.GetComponent <
IDropHandler > () != null;

    if (droppedOnSlot) {
      // Don't need to do anything — item has been added to crafting system,
      // and UpdateVisual will create the proper item in the correct slot.
      Destroy(gameObject); // Remove dragged UI object
    }
    else {
      // Snap back to original slot
      if (originalParent != null && originalParent.gameObject.activeInHierarchy) {
        transform.SetParent(originalParent, false);
        rectTransform.anchoredPosition = originalPosition;
      }
      else {
        Debug.LogWarning("Original parent was destroyed or inactive. Rebuilding drag item.");
        Destroy(gameObject); // fallback, avoid hanging item
      }
    }

  }

  public void OnPointerDown(PointerEventData eventData) {
    Debug.Log("Down Pressed");
```

```
  }

  public void OnDrag(PointerEventData eventData) {
    rectTransform.anchoredPosition += eventData.delta / canvas.scaleFactor;
  }

  public void SetSprite(Sprite sprite) {
    image.sprite = sprite;
  }

  public void Hide() {
    gameObject.SetActive(false);
  }

  public void Show() {
    gameObject.SetActive(true);
  }

  public void SetItem(Item item) {
    this.item = item;
    Sprite sprite = item.GetSprite();
    SetSprite(sprite);
    Debug.Log("Set sprite to: " + sprite ? .name);

  }

  public Item GetItem() {
    return item;
  }
}
```

CraftingSystem Class

```
using System;
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class CraftingSystem {
  public const int GRID_SIZE = 5;
  public event EventHandler OnGridChanged;
  private Item[, ] itemArray;
  private Item outputItem;
```

```
//private Item.ItemType[,] biscottiRecipe;
private List < RecipeScriptableObject > recipeList;

public CraftingSystem(List < RecipeScriptableObject > loadedRecipes) {
  itemArray = new Item[1, GRID_SIZE];
  recipeList = loadedRecipes;
}

public bool IsEmpty(int x, int y) {
  return itemArray[x, y] == null;
}

public Item GetItem(int x, int y) {
  return itemArray[x, y];
}

public void SetItem(Item item, int x, int y) {
  itemArray[x, y] = item;
  CreateOutput();

}

public void IncreaseItemAmount(int x, int y) {
  GetItem(x, y).amount++;
  OnGridChanged ? .Invoke(this, EventArgs.Empty);
}

public void DecreaseItemAmount(int x, int y) {
  GetItem(x, y).amount--;
  OnGridChanged ? .Invoke(this, EventArgs.Empty);
}

public void RemoveItem(Item item, int x, int y) {
  if (item == null && x == 0 && y == 0) // Special case for clearing output
  {
    outputItem = null;
    OnGridChanged ? .Invoke(this, EventArgs.Empty);
    return;
  }
  SetItem(null, x, y);

}

public bool TryAddItem(Item item, int x, int y) {
```

```
  if (IsEmpty(x, y)) {
    SetItem(item, x, y);
    return true;
  }
  else {
    if (item.itemScriptableObject == GetItem(x, y).itemScriptableObject) {
      IncreaseItemAmount(x, y);
      return true;
    }
    else {
      return false;
    }
  }
 }

 private ItemScriptableObject GetRecipeOutput() {
  foreach(var recipe in recipeList) {
    bool match = true;
    for (int i = 0; i < GRID_SIZE; i++) {
      var required = recipe.ingredients[i];
      var current = itemArray[0, i];

      if (required != null) {
        if (current == null || current.itemScriptableObject != required) {
          Debug.Log($ "Mismatch at index {i}: Required {required?.name}, Found
{current?.itemScriptableObject?.name}");

          match = false;
          break;
        }
      }
      else if (current != null) {
        Debug.Log($ "Slot {i} should be empty but found {current.itemScriptableObject?.name}");

        match = false;
        break;
      }
    }
    if (match) {
      Debug.Log($ "Recipe matched: {recipe.name}");
      return recipe.output;
    }
  }
  Debug.Log("No matching recipe found");
```

```
    return null;
  }

  private void CreateOutput() {
    var result = GetRecipeOutput();
    if (result != null) {
      outputItem = new Item {
        itemScriptableObject = result,
        amount = 1
      };
      for (int y = 0; y < GRID_SIZE; y++) {
        itemArray[0, y] = null;
      }
    }
    else {
      outputItem = null;
    }
    OnGridChanged ? .Invoke(this, EventArgs.Empty);
  }

  public Item GetOutputItem() {
    //Debug.Log("GetOutputItem returning: " + outputItem?.itemType);
    return outputItem;
  }

}
```

Inventory Class

```
using System;
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Inventory {
  private List < Item > itemList;

  public event EventHandler OnItemListChanged;
  private const int MAX_STACK_SIZE = 16;

  public Inventory() {
    itemList = new List < Item > ();

  }
```

```
public void AddItem(Item item) {
  if (item.IsStackable()) {
    bool itemAdded = false;
    foreach(Item inventoryItem in itemList) {
      if (inventoryItem.itemScriptableObject == item.itemScriptableObject) {
        // Check if adding the new amount would exceed the stack limit
        if (inventoryItem.amount + item.amount <= MAX_STACK_SIZE) {
          // Add to existing stack if within limit
          inventoryItem.amount += item.amount;
          itemAdded = true;
        }
        else {
          // Fill the current stack to the limit and create a new stack for the remainder
          int remainingAmount = (inventoryItem.amount + item.amount) - MAX_STACK_SIZE;
          inventoryItem.amount = MAX_STACK_SIZE;
          if (remainingAmount > 0) {
            // Create a new stack with the remaining amount
            itemList.Add(new Item {
              itemScriptableObject = item.itemScriptableObject,
              amount = remainingAmount
            });
          }
          itemAdded = true;
        }
        break; // Exit loop after handling the matching item
      }
    }
    if (!itemAdded) {
      // Add a new stack if no matching item was found or if all stacks are full
      itemList.Add(new Item {
        itemScriptableObject = item.itemScriptableObject,
        amount = Math.Min(item.amount, MAX_STACK_SIZE)
      });
      // If the incoming amount exceeds the stack limit, create additional stacks
      int remainingAmount = item.amount - MAX_STACK_SIZE;
      while (remainingAmount > 0) {
        itemList.Add(new Item {
          itemScriptableObject = item.itemScriptableObject,
          amount = Math.Min(remainingAmount, MAX_STACK_SIZE)
        });
        remainingAmount -= MAX_STACK_SIZE;
      }
    }
```

```
    }
    else {
      // Non-stackable items are added directly
      itemList.Add(item);
    }
    OnItemListChanged ? .Invoke(this, EventArgs.Empty);
  }

  public void RemoveItem(Item item) {
    if (item.IsStackable()) {
      Item itemInInventory = null;
      foreach(Item inventoryItem in itemList) {
        if (inventoryItem.itemScriptableObject == item.itemScriptableObject) {
          inventoryItem.amount -= item.amount;
          itemInInventory = inventoryItem;
        }
      }
      if (itemInInventory != null && itemInInventory.amount <= 0) {
        itemList.Remove(itemInInventory);
      }
    }
    else {
      itemList.Remove(item);
    }
    OnItemListChanged ? .Invoke(this, EventArgs.Empty);

  }
  public List < Item > GetItemList() {
    return itemList;
  }
}
```

Item Class

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

[System.Serializable]

public class Item {
  public ItemScriptableObject itemScriptableObject;
  public int amount;
```

```
  public Sprite GetSprite() {
    return itemScriptableObject.itemSprite;

  }

  public bool IsStackable() {
    return itemScriptableObject.isStackable;
  }

  public override string ToString() {
    return $ "{itemScriptableObject.itemName} (x{amount})";
  }
}
```

ItemScriptableObject Class

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

[CreateAssetMenu(menuName = "ScriptableObjects/ItemScriptableObject")]

public class ItemScriptableObject: ScriptableObject {
  //public Item.ItemType itemType;
  public string itemName;
  public Sprite itemSprite;
  public bool isStackable = true;

}
```

ItemWorld Class

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class ItemWorld: MonoBehaviour {

  [SerializeField] private Transform pfItemWorld;

  /*public static ItemWorld SpawnItemWorld(Vector3 position, Item item)
    {
       Transform transform = Instantiate(ItemAssets.Instance.pfItemWorld, position, Quaternion.identity);
```

```
        ItemWorld itemworld = transform.GetComponent<ItemWorld>();
        itemworld.SetItem(item);
        return itemworld;
    }*/
    public static ItemWorld SpawnItemWorld(Vector3 position, Item item) {
        GameObject go = GameObject.Find("ItemWorldSpawner");
        if (go == null) {
            Debug.LogError("No object named 'ItemWorldSpawner' found to access prefab.");
            return null;
        }

        ItemWorld spawner = go.GetComponent < ItemWorld > ();
        if (spawner == null) {
            Debug.LogError("ItemWorldSpawner does not have ItemWorld component.");
            return null;
        }

        Transform transform = Instantiate(spawner.pfItemWorld, position, Quaternion.identity);

        ItemWorld itemWorld = transform.GetComponent < ItemWorld > ();
        itemWorld.SetItem(item);
        return itemWorld;
    }

    public static ItemWorld DropItem(Vector3 dropPosition, Item item) {
        Vector3 randomDir = new Vector3(Random.Range( - 1f, 1f), Random.Range( - 1f, 1f)).normalized;
        ItemWorld itemWorld = SpawnItemWorld(dropPosition + randomDir * 1f, item);
        itemWorld.GetComponent < Rigidbody2D > ().AddForce(randomDir * 1f, ForceMode2D.Impulse);
        return itemWorld;
    }

    private Item item;
    private SpriteRenderer spriteRenderer;
    private void Awake() {
        spriteRenderer = GetComponent < SpriteRenderer > ();
    }

    public void SetItem(Item item) {
        this.item = item;
        spriteRenderer.sprite = item.GetSprite();
    }

    public Item GetItem() {
        return item;
```

```
  }

  public void DestroySelf() {
    Destroy(gameObject);
  }

}
```

Player Class

```
using UnityEngine;
using UnityEngine.InputSystem;
using System.Collections.Generic;

public class Player: MonoBehaviour { [SerializeField] private UI_Inventory uiInventory;
  [SerializeField] private UI_CraftingSystem uiCraftingSystem;
  [SerializeField] private List < RecipeScriptableObject > recipeList;

  public float moveSpeed = 3f;

  private Rigidbody2D rb;
  private Animator animator;
  private Vector2 moveInput;
  private Inventory inventory;
  private CraftingSystem craftingSystem;

  void Start() {
    rb = GetComponent < Rigidbody2D > ();
    animator = GetComponent < Animator > ();
    if (rb == null) Debug.LogError("Rigidbody2D is missing!");
    if (animator == null) Debug.LogError("Animator is missing!");

    rb.bodyType = RigidbodyType2D.Dynamic;
    rb.gravityScale = 0f;
    rb.freezeRotation = true;
    rb.collisionDetectionMode = CollisionDetectionMode2D.Continuous;
    Debug.Log("Player script started!");

    craftingSystem = new CraftingSystem(recipeList);
    uiCraftingSystem.SetCraftingSystem(craftingSystem);

    inventory = new Inventory();
    uiInventory.SetPlayer(this);
    uiInventory.SetInventory(inventory);
```

```
    //ItemWorld.SpawnItemWorld(new Vector3 (1, 0), new Item {itemType = Item.ItemType.Almond,
amount = 1});
    //ItemWorld.SpawnItemWorld(new Vector3 (0, 1), new Item {itemType = Item.ItemType.Egg, amount
= 1});
    //ItemWorld.SpawnItemWorld(new Vector3 (1, 1), new Item {itemType = Item.ItemType.Honey,
amount = 1});
    // Load all items from Resources/Items
    ItemScriptableObject[] allItems = Resources.LoadAll < ItemScriptableObject > ("Items");

    foreach(ItemScriptableObject itemSO in allItems) {
      inventory.AddItem(new Item {
        itemScriptableObject = itemSO,
        amount = 2
      });
      Debug.Log($ "Added {itemSO.itemName} to inventory");
      if (itemSO.itemName == "Honey") {
        inventory.AddItem(new Item {
          itemScriptableObject = itemSO,
          amount = 18
        });
      }
    }
  }

  public Inventory GetInventory() {
    return inventory;
  }

  private void OnTriggerEnter2D(Collider2D collider) {
    ItemWorld itemWorld = collider.GetComponent < ItemWorld > ();
    if (itemWorld != null) {
      inventory.AddItem(itemWorld.GetItem());
      itemWorld.DestroySelf();
    }
  }

  void Update() {
    // Reset movement
    moveInput = Vector2.zero;

    // Read input (Input System)
    if (Keyboard.current.wKey.isPressed || Keyboard.current.upArrowKey.isPressed) moveInput.y += 1f;
    if (Keyboard.current.sKey.isPressed || Keyboard.current.downArrowKey.isPressed) moveInput.y -= 1f;
```

```
    if (Keyboard.current.aKey.isPressed || Keyboard.current.leftArrowKey.isPressed) moveInput.x -= 1f;
    if (Keyboard.current.dKey.isPressed || Keyboard.current.rightArrowKey.isPressed) moveInput.x += 1f;

    // Normalize to prevent diagonal speed boost
    Vector2 normalizedMove = moveInput.normalized;

    // Update Animator parameters for Blend Tree
    animator.SetFloat("MoveX", normalizedMove.x);
    animator.SetFloat("MoveY", normalizedMove.y);
    animator.SetBool("IsMoving", moveInput.sqrMagnitude > 0.01f);
  }

  void FixedUpdate() {
    rb.linearVelocity = moveInput.normalized * moveSpeed;
  }

  public Vector3 GetPosition() {
    return transform.position;
  }

}
```

RecipeScriptableObject Class

```
using UnityEngine;

[CreateAssetMenu(menuName = "ScriptableObjects/RecipeScriptableObject")]

public class RecipeScriptableObject: ScriptableObject {
  public ItemScriptableObject[] ingredients = new ItemScriptableObject[5];
  public ItemScriptableObject output;
}
```

UI_CraftingItemSlot Class

```
using UnityEngine;
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.EventSystems;

public class UI_CraftingItemSlot: MonoBehaviour,
IDropHandler {
  public event EventHandler < OnItemDroppedEventArgs > OnItemDropped;
```

```
  public class OnItemDroppedEventArgs: EventArgs {
    public Item item;
    public int x;
    public int y;
  }

  private int x;
  private int y;

  public void OnDrop(PointerEventData eventData) {
    Item item = eventData.pointerDrag ? .GetComponent < DragDrop > () ? .GetItem();

    if (item == null) {
      Debug.LogWarning("Dropped item is null in UI_CraftingItemSlot.OnDrop");
      return;
    }
    OnItemDropped ? .Invoke(this, new OnItemDroppedEventArgs {
      item = item,
      x = x,
      y = y
    });
  }

  public void SetXY(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

UI_CraftingSystem Class

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using TMPro;

public class UI_CraftingSystem: MonoBehaviour {
  private Transform[, ] slotTransformArray;
  private Transform outputSlotTransform;
  private Transform itemContainer;
  public CraftingSystem craftingSystem;

  [SerializeField] private Transform pfUI_Item;
```

```
  private void Awake() {
    Transform gridContainer = transform.Find("gridContainer");
    itemContainer = transform.Find("itemContainer");
    outputSlotTransform = transform.Find("outputSlot");
    slotTransformArray = new Transform[1, CraftingSystem.GRID_SIZE];

    for (int x = 0; x < 1; x++) {
      for (int y = 0; y < CraftingSystem.GRID_SIZE; y++) {
        slotTransformArray[x, y] = gridContainer.Find("grid_" + x + "_" + y);
        UI_CraftingItemSlot craftingItemSlot = slotTransformArray[x, y].GetComponent <
UI_CraftingItemSlot > ();
        craftingItemSlot.SetXY(x, y);
        craftingItemSlot.OnItemDropped += UI_CraftingSystem_OnItemDropped;
      }
    }

  }

  public void SetCraftingSystem(CraftingSystem craftingSystem) {
    this.craftingSystem = craftingSystem;
    craftingSystem.OnGridChanged += CraftingSystem_OnGridChanged;
    UpdateVisual();
  }

  private void CraftingSystem_OnGridChanged(object sender, System.EventArgs e) {
    UpdateVisual();
  }

  private void UI_CraftingSystem_OnItemDropped(object sender,
UI_CraftingItemSlot.OnItemDroppedEventArgs e) {
    Debug.Log(e.item + " " + e.x + " " + e.y);

    // Remove 1 of the item from the inventory
    Player player = Object.FindFirstObjectByType < Player > ();
    if (player != null) {
      Inventory inventory = player.GetComponent < Player > ().GetInventory();
      if (inventory != null) {
        inventory.RemoveItem(new Item {
          itemScriptableObject = e.item.itemScriptableObject,
          amount = 1
        });
      }
    }
```

```
  // Add a copy to crafting system
  Item cloneItem = new Item {
    itemScriptableObject = e.item.itemScriptableObject,
    amount = 1
  };
  craftingSystem.TryAddItem(cloneItem, e.x, e.y);
}

public void UpdateVisual() {

  // Clear items from slot transforms
  for (int x = 0; x < 1; x++) {
    for (int y = 0; y < CraftingSystem.GRID_SIZE; y++) {
      foreach(Transform child in slotTransformArray[x, y]) {
        Destroy(child.gameObject);
      }
    }
  }

  for (int x = 0; x < 1; x++) {
    for (int y = 0; y < CraftingSystem.GRID_SIZE; y++) {
      if (!craftingSystem.IsEmpty(x, y)) {
        CreateItem(x, y, craftingSystem.GetItem(x, y));
      }
    }
  }

  if (craftingSystem.GetOutputItem() != null) {
    Debug.Log("Output item exists: ");
    CreateItemOutput(craftingSystem.GetOutputItem());

  }
  else {
    Debug.Log("No output item to display.");
  }

}

private void CreateItem(int x, int y, Item item) {

  Transform itemTransform = Instantiate(pfUI_Item, slotTransformArray[x, y]);
  itemTransform.localPosition = Vector3.zero; // Ensure it aligns inside the slot
  itemTransform.localScale = Vector3.one; // Reset scale just in case
  itemTransform.GetComponent < DragDrop > ().SetItem(item);
```

```
    // Optional: show item amount if > 1
    TextMeshProUGUI amountText = itemTransform.Find("text") ? .GetComponent < TextMeshProUGUI
> ();
    if (amountText != null) {
      amountText.SetText(item.amount > 1 ? item.amount.ToString() : "");
    }

  }

  private void CreateItemOutput(Item item) {
    Debug.Log("Creating output UI item: " + item.itemScriptableObject.itemName);
    foreach(Transform child in outputSlotTransform) {
      Destroy(child.gameObject);
    }
    Transform itemTransform = Instantiate(pfUI_Item, outputSlotTransform);
    itemTransform.localPosition = Vector3.zero;
    itemTransform.localScale = Vector3.one;
    DragDrop dragDrop = itemTransform.GetComponent < DragDrop > ();
    if (dragDrop != null) {
      dragDrop.SetItem(item);
    }
    else {
      Debug.LogWarning("DragDrop component missing on output item!");
    }
    TextMeshProUGUI amountText = itemTransform.Find("text") ? .GetComponent < TextMeshProUGUI
> ();
    if (amountText != null) {
      amountText.SetText(item.amount > 1 ? item.amount.ToString() : "");
    }
  }

}
```

UI_Inventory Class

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.UI;
using TMPro;

public class UI_Inventory: MonoBehaviour {
  private Inventory inventory;
  private Transform itemSlotContainer;
```

```
private Transform[, ] itemSlotArray; // 2x4 array to hold slot transforms
private Player player;

private const int ROWS = 2;
private const int COLS = 4;

[SerializeField] private Transform pfUI_Item;

public void SetPlayer(Player player) {
  this.player = player;
}

private void Awake() {
  // Find the container that holds the pre-placed slots
  itemSlotContainer = transform.Find("itemSlotContainer");
  itemSlotArray = new Transform[ROWS, COLS];

  // Initialize the slot array by finding pre-existing slots
  for (int y = 0; y < ROWS; y++) {
    for (int x = 0; x < COLS; x++) {
      string slotName = $ "itemSlot_{y}_{x}";
      Transform slotTransform = itemSlotContainer.Find(slotName);

      if (slotTransform == null) {
        Debug.LogWarning($ "Slot not found: {slotName}");
        continue;
      }

      itemSlotArray[y, x] = slotTransform;
      slotTransform.gameObject.SetActive(true);
      SetupSlotInteractions(slotTransform, x, y);
    }
  }
}

public void SetInventory(Inventory inventory) {
  this.inventory = inventory;
  inventory.OnItemListChanged += Inventory_OnItemListChanged;
  RefreshInventoryItems();
}

private void Inventory_OnItemListChanged(object sender, System.EventArgs e) {
  RefreshInventoryItems();
}
```

```
 private void RefreshInventoryItems() {
  // Get the current item list
  List < Item > itemList = inventory.GetItemList();

  // Update each slot in the 2x4 grid
  for (int y = 0; y < ROWS; y++) {
   for (int x = 0; x < COLS; x++) {
    Transform slotTransform = itemSlotArray[y, x];
    if (slotTransform == null) continue;

    // Clear previous item
    foreach(Transform child in slotTransform) {
     Destroy(child.gameObject);
    }

    int itemIndex = y * COLS + x;
    if (itemIndex < itemList.Count) {
     Item item = itemList[itemIndex];

     Transform itemTransform = Instantiate(pfUI_Item, slotTransform);
     itemTransform.localPosition = Vector3.zero;
     itemTransform.localScale = Vector3.one;

     DragDrop dragDrop = itemTransform.GetComponent < DragDrop > ();
     if (dragDrop != null) {
      dragDrop.SetItem(item);
     }

     // Optional: show item amount if > 1
     TextMeshProUGUI amountText = itemTransform.Find("text") ? .GetComponent <
TextMeshProUGUI > ();
     if (amountText != null) {
      amountText.SetText(item.amount > 1 ? item.amount.ToString() : "");
     }
    }
   }
  }
 }

 private void SetupSlotInteractions(Transform slotTransform, int x, int y) {
  RightClickHandler rightClick = slotTransform.GetComponent < RightClickHandler > ();
  if (rightClick == null) {
   rightClick = slotTransform.gameObject.AddComponent < RightClickHandler > ();
```

```
    }

    rightClick.OnRightClick = () =>{
      // Get item from child prefab
      DragDrop dragDrop = slotTransform.GetComponentInChildren < DragDrop > ();
      if (dragDrop == null) return;

      Item slotItem = dragDrop.GetItem();
      if (slotItem != null) {
        Item singleItem = new Item {
          itemScriptableObject = slotItem.itemScriptableObject,
          amount = 1
        };
        inventory.RemoveItem(singleItem);
        ItemWorld.DropItem(player.GetPosition(), singleItem);
        RefreshInventoryItems();
      }
    };

    Button button = slotTransform.GetComponent < Button > ();
    if (button != null) {
      button.onClick.RemoveAllListeners();
      button.onClick.AddListener(() =>{
        Debug.Log($ "Left-clicked slot {x}_{y}");
      });
    }
  }

}
```

UI_InventorySlotDropHandler Class

```
using UnityEngine;
using UnityEngine.EventSystems;

public class UI_InventorySlotDropHandler: MonoBehaviour,
IDropHandler {
  private int x;
  private int y;

  public void SetSlotPosition(int x, int y) {
    this.x = x;
    this.y = y;
  }
```

```
  public void OnDrop(PointerEventData eventData) {
    Item item = eventData.pointerDrag ? .GetComponent < DragDrop > () ? .GetItem();

    if (item == null) {
      Debug.LogWarning("Dropped item is null in UI_InventorySlotDropHandler");
      return;
    }

    // Add the item to the player's inventory
    Player player = Object.FindFirstObjectByType < Player > ();
    if (player != null) {
      player.GetInventory().AddItem(item);
    }
  }
}
```