CMP302 Gameplay Mechanics Report

Michael Ennis

1902141

Terminology

“Slot” refers to each individual inventory slot found within the Inventory System. Both the Container and the Player inventories have these slots within their UI’s. These slots are each an FSlot struct found inside an array, which controls the position of all items found within the inventory system.

“Fill up” refers to a slot that the player drags an item from, and a slot that the player drags that item to, aka “Source” and “Destination” slot. If the destination slot contains the same type of item as the source slot, then it will attempt to add the quantity of source slot to the destination slot, to “Fill up” the slot. However, if the max stack size has already been reached for the destination slot, the remaining items will be replaced in the source slot.

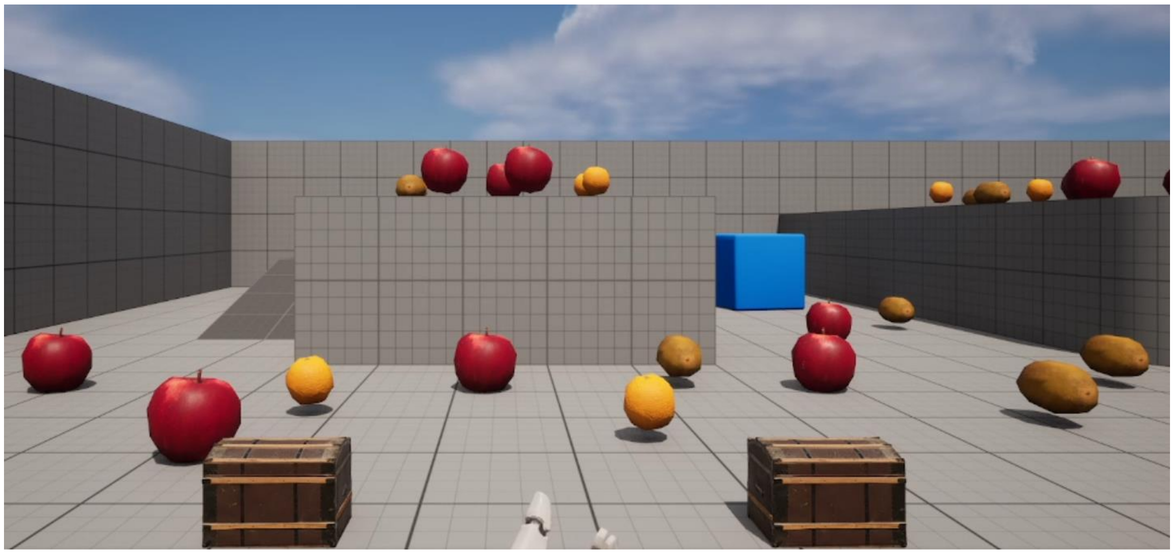
“Max Stack Size” refers to how many items a slot can store. Each item has its own unique max stack size, which is easily editable in the Item Data Table.

Video Demonstration Link: https://youtu.be/ybbXlI4o7wQ

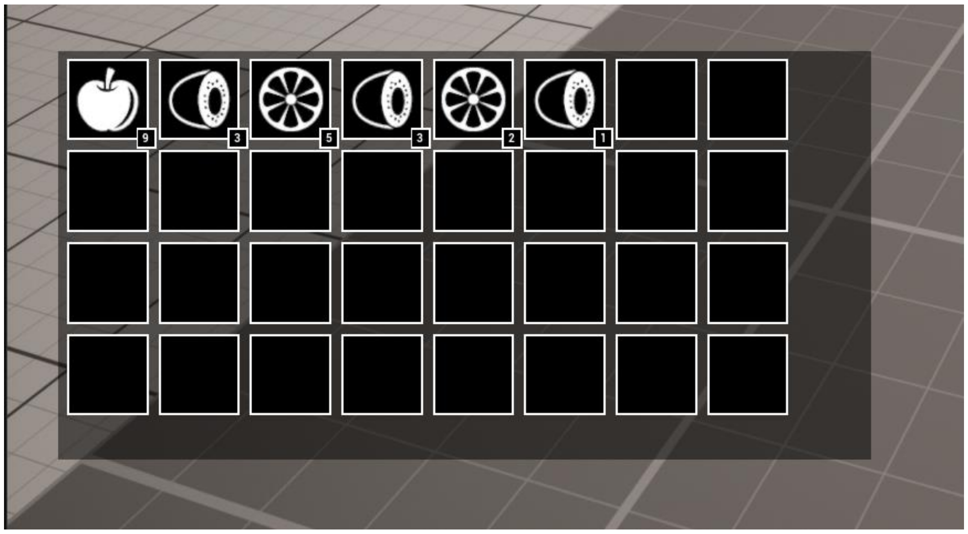
<u>Summary</u>

The mechanic is a simple modular inventory system, that can be applied to any object or player that requires the ability to store items. The inventory system is very adaptable, as items within it can swap slots with other items or “fill up” slots that contain the same type of items. This also works across the Chest inventory, found within the world, so that any normal inventory action that the player can choose to perform can also be done within the Chest inventory itself. This system takes inspiration from many RPG and Sandbox games that have excellent inventory systems, including Minecraft (Mojang Studios, 2011) and Terraria (Re-Logic, 2011).

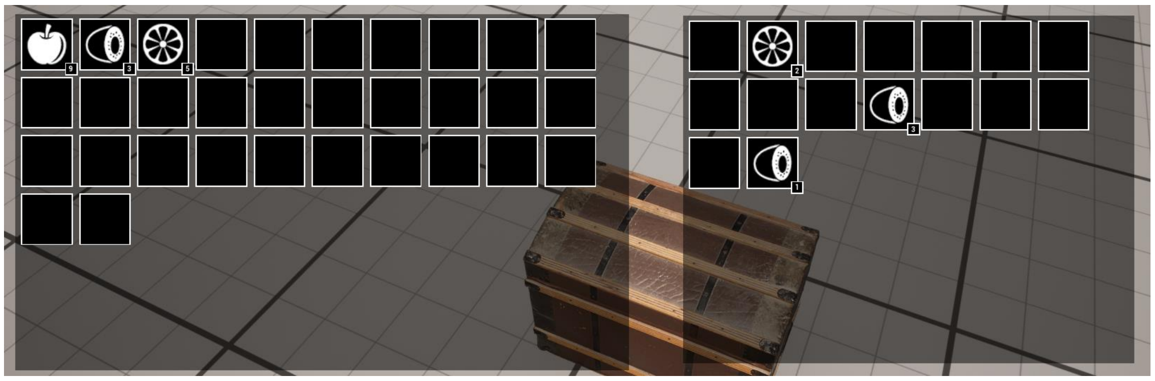
1. Inventory UI
    a. Upon pressing the “i” key, the UI opens
    b. The UI will close the next time the “i” key is pressed
    c. Upon pressing “Right Mouse Button” and hovering over an item, a menu will open
    d. In that menu, all or a single item in that slot can be dropped on the ground
    e. These items can later be picked up and readded back to the inventory system
2. Slots
    a. An Inventory UI (Includes container UI) have several slots to store items
    b. Each slot can store up to the max stack size of an item
    c. Slots can be swapped by dragging the items from one slot to another
    d. If the slots contain items of the same type, one slot can “Fill up” another.
3. Items
    a. Items can be picked up by looking at them and pressing the “E” key
    b. A small widget will be displayed to the screen showing the item’s name
    c. Items can also be dropped from the UI and placed back inside the world
    d. Each item in a slot has a counter that specifies how many items are in that slot
4. Container UI
    a. Upon pressing the “E” key and looking at the chest, the Container UI opens
    b. The Player Inventory System UI also opens inside the Container UI
    c. All slot operations can be performed inside the Container and across Container and Player inventories
    d. The Container saves its inventory and can be accessed later to add or remove items



World View



Player Inventory


Container + Player Inventory

Introduction

Many games rely on Inventory Systems to allow players some autonomy over storing and managing their own items. This is best shown in games like Minecraft (Mojang Studios, 2011) where the player inventory allows complete freedom over which items should be placed or stored in which slots. This allows for some creativity from the user in terms of item management and gives more agency to the player.

The purpose of this project was to create a simple, modular system that can easily be added to any players or containers throughout the world without having to design different inventory systems every time a new container is added to the world. It is also easily modifiable, as adding a Shop or Trading system would make use of a lot of this core functionality. This project is also an excellent example of how C++ can be integrated into already existing blueprints, as it takes over the role of data management, in terms of the slots and the operations, making the program more efficient.

The deliverables for this project are an Inventory System Component, that can be added to anything with the ability to store items. The project is designed to be easily modifiable, as such certain blueprints and UI can be further developed upon to add any functionality required by the game itself.

Overall Description

The primary aim of this product is to provide a simple-to-use, but highly modifiable, inventory system capable of adapting to anything the designers need. There are currently 4 different actors in the scene that are interactable; the Chest, Apple, Orange and Kiwi. The fruits each have different icons and different stack sizes to differentiate them. They also all have different Data Table entries to represent this.

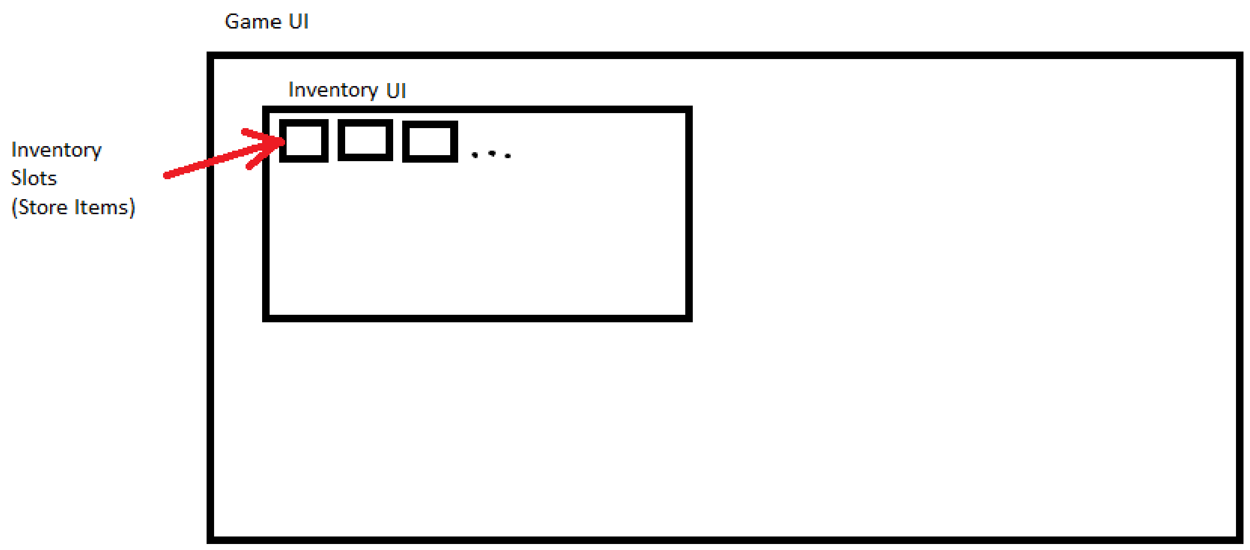
Product Functions:

1. Allow easy access to the User's Inventory System, which stores all their items
2. Show the user what items they are currently viewing in the world
3. Allow the user to pick up items and delete them from the world
4. Allow the user to drop items back into the world and delete them from their inventory system
5. Allow storage containers to store items inside of their own inventory systems and save the items contained within them
6. Allow a programmer the ability to further modify the already existing inventory system to accommodate any new features
7. Allow a designer or programmer the ability to further modify the UI to suit the specific game's theme
8. Allow a designer to add further Data Table entries for any new items that can be stored inside of the inventory system

This product is aimed towards an intermediate level of understanding of Unreal Blueprinting. The current implementation should not require any further modifications to work as intended, however any further systems, e.g., shops, will likely require modifications built using the core functionality provided by this product. The product has been designed and implemented with these goals in mind, however, and should be relatively simple to further improve the existing mechanic.

System Features

Player Inventory UI



Priority: 9

Description:

A Player Inventory UI that shows all items collected by the player. It also displays the quantity of items in a slot in the bottom right corner of an Inventory Slot. Other UI screens for different systems can also be displayed on the right side of the Game Screen, as such the Player Inventory UI only displays on the left. The items shown are stored in the Inventory System Component, in a C++ Struct.

Response Sequences:

The Inventory UI is shown by pressing the “i” key. After the key is pressed, all the items are displayed in a grid to the user. When the Inventory UI is visible, the mouse is enabled to allow for slot operations to take place, such as swapping slots and increasing the amount of an item in one slot.

For development, the Inventory System will open through the UE5 Blueprint Editor – and the C++ code will open Rider. As much of the main functionality of this will be done in C++ as possible, as C++ excells when implementing data structures.

Functional Requirements:

REQ-1: Receive Input from Player Controller

Must receive input from the player to open. The inventory system is also attached to a container, which opens with a different input key and under different conditions.

REQ-2: Display Inventory items

Must display Inventory Items from the C++ FSlot struct array, found within the Inventory System class, in a grid format.

REQ-3: Must enable mouse input

Must be able to allow for the player to use their mouse, in combination with the slots.

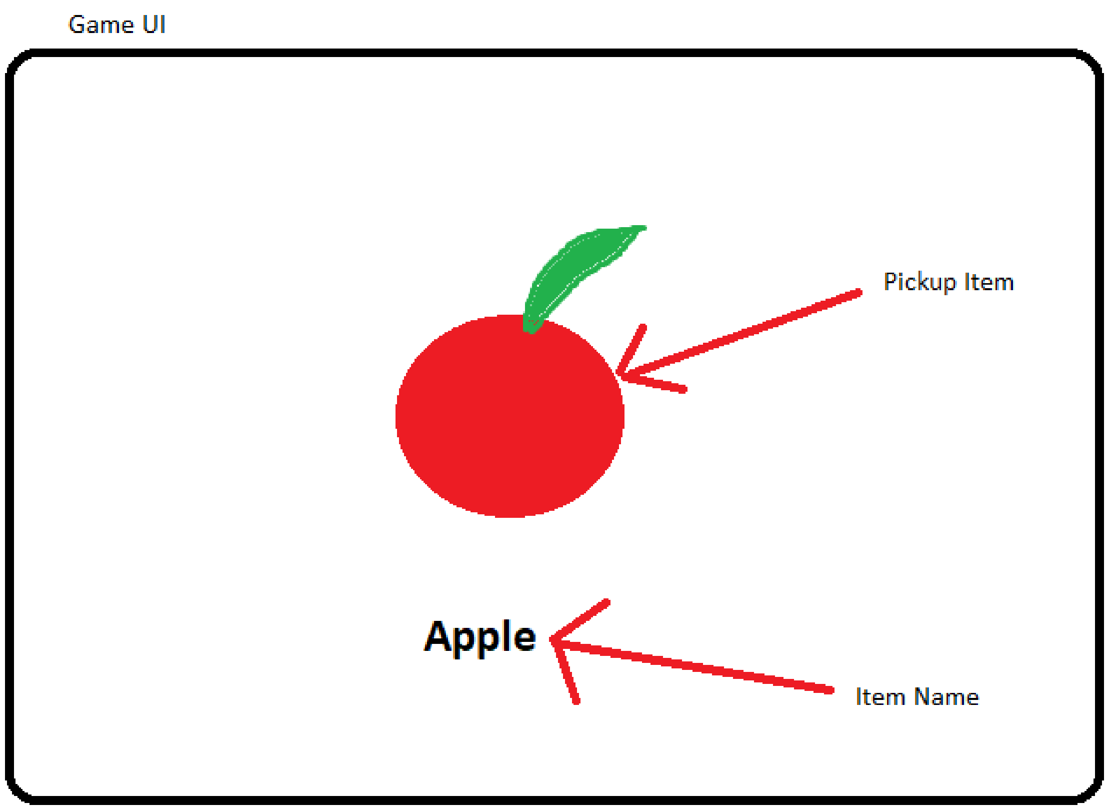
REQ-4: Must allow for Drag and Drop operations

Must let the player complete Drag and Drop operations, to enable organizing the inventory system.

REQ-5: Must Close after Receiving input from Player Controller

Must be able to return to the game after the player is finished structuring their inventory system.

<u>Pickup Items</u>



Priority: 9

Description:

An item that can be picked up by the player and placed in their inventory. When viewed by the player, a small display box should be rendered to the bottom of the screen showing the currently viewed item's name.

Response Sequences:

Players can add these items to their inventory by pressing the "E" key. These items will show a small display box when the player is directly looking at them, so that the player knows what kind of item they are picking up.

For development, the Item Data Component will open through the UE5 Blueprint Editor – C++ will not be used, except for storing the Name of the item.

Functional Requirements:

REQ-1: Receive Input from Player Controller

Must receive input from the player to interact with the item.

REQ-2: Item is being Looked At

Must be within a certain distance and looking directly at the item to be looking it.

REQ-3: Item should be added Inventory System

Must be added to the player's Inventory System after interacting.

REQ-4: Item should be deleted from the world

Must delete the Item after it's been added to the Inventory System

REQ-5: Items should be added to duplicate stack

Must add the item first to an already existing slot with this item in it, before creating a new stack

REQ-6: Must Display name of the Item to the Screen

Must add a small text box that displays the item before picking it up

Container Item

Priority: 7

Description:

Like a pickup item, except this item has an Inventory System component attached and cannot be added to an Inventory System itself. Can store items and save the contents of its Inventory System. When interacting with the Player, a separate Container UI will become visible, alongside the Player UI. This UI allows the player to perform Drag and Drop operations over both the UI.

Response Sequences:

Players can interact with this Item by pressing the “E” key. The Container UI will then be displayed to the screen, alongside the Player Inventory UI. When the Container UI is visible, the mouse is also enabled, and the Player can organize the items in these two inventories as required.

For development, the Inventory System will open through the UE5 Blueprint Editor – and the C++ code will open Rider. As much of the main functionality of this will be done in C++ as possible, as C++ excells when implementing data structures.

Functional Requirements:

REQ-1: Receive Input from Player Controller

Must receive input from the Player Controller to open the Container’s Inventory System

REQ-2: Container is being Looked At

Must be within a certain distance and looking directly at the container to be looking it.

REQ-3: Display Inventory Items

Must display both the Container UI and the Player Inventory UI’s items at the same time.

REQ-4: Must enable Mouse Movement

Must be able to allow for the player to use their mouse, in combination with the slots of both the Container and Player Inventory UI.

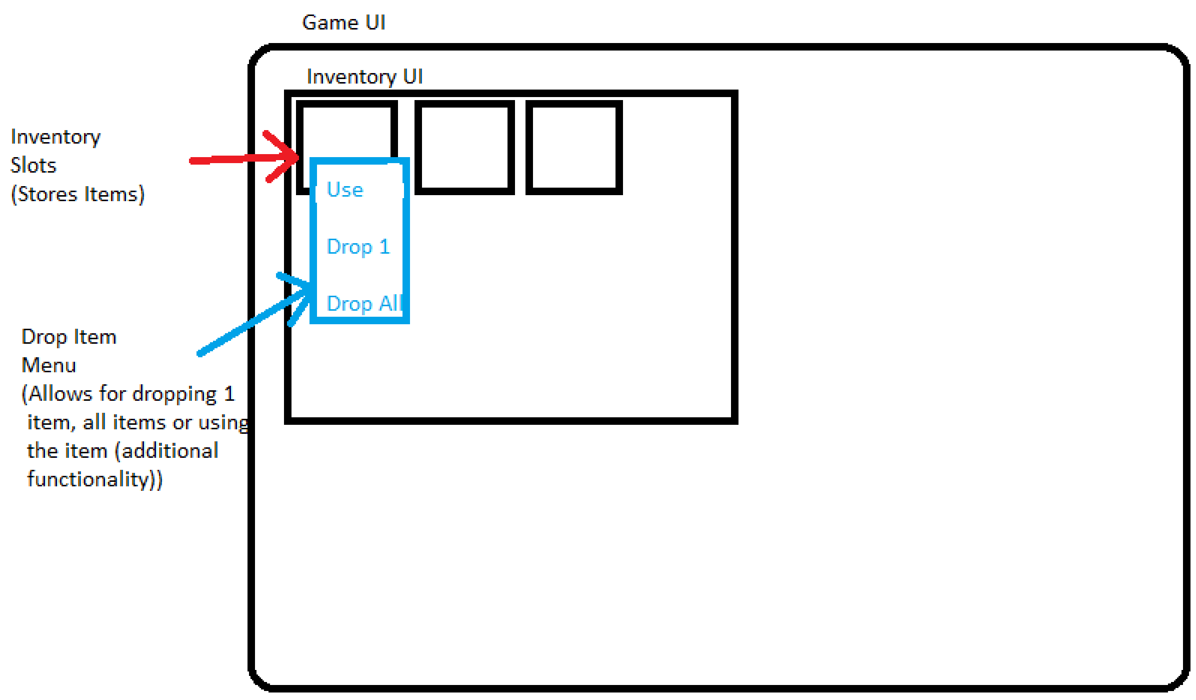
REQ-5: Must allow for Drag and Drop Operations

Must let the player complete Drag and Drop operations in both the Container and Player Inventory. Must also allow for the operations to succeed across both UIs.

REQ-6: Must Close after Receiving input from Player Controller

Must be able to return to the game after the player is done organizing both their own Inventory and the Container's Inventory.

Drop item Menu


Game UI
Inventory UI
Inventory Slots (Stores Items)
Use
Drop 1
Drop All
Drop Item Menu (Allows for dropping 1 item, all items or using the item (additional functionality))

Priority: 5

Description:

A drop-down menu with several buttons to drop items and use them. This drop-down menu enables items to be put back into the world after entering the Player's Inventory System. Also, later functionality could add something to consume the item, deleting it from the game rather than placing it.

Response Sequences:

Players can access this drop-down menu by using the "Right Click" mouse button, while hovering their mouse over a slot. When this menu is visible, the menu will disappear after the player clicks on one of the buttons or their mouse cursor moves far enough away from the menu.

For development, the UI Blueprint will open through the UE5 Blueprint Editor, to assign functionality to the buttons.

Functional Requirements:

REQ-1: Player Inventory must be visible

Must only be able to open the menu when the player has their Inventory UI open.

REQ-2: Mouse must be hovering over the slot

Must only be able to open the menu when the player is hovering over a slot with items inside.

REQ-3: Receive Input from Mouse

Must receive input from the mouse to select an option from the menu.

REQ-4: Must Drop Item into the world

Must place the item from the inventory into the world after pressing one of the “Drop” buttons

REQ-5: Must remove item from Inventory

Must remove the item from the inventory after placing in the world

Other Non-Functional Requirements

Performance Requirements:

While the game is in play, there should be very little interference with the system even with low frame rates. The only issue occurring could be the potential duplication of items when dropping them onto the floor if the frame rate drops too substantially.

Software Quality:

The software implementation will be kept to a high quality and concise standard. It will follow standard practice for programming with C++ including Header and Source files. It will be simplified as much as possible to lower the number of redundant lines, but still using an Object Orientated Approach, while also following the UE5 Coding Standards.

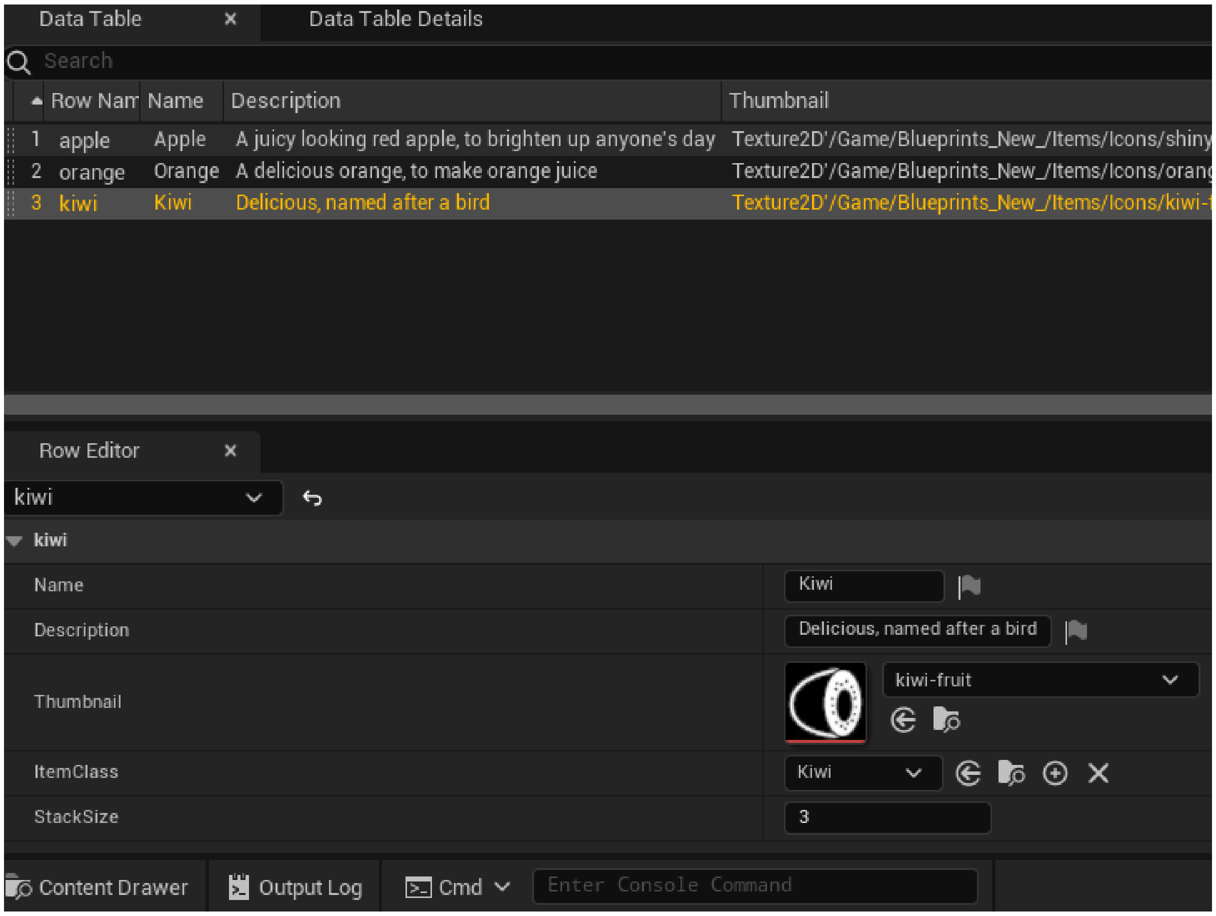
Any functions or variables found in C++ that are used in Blueprints will have a “c” attached to their name, to represent where they came from.

Method

Item Definitions:

Item definitions are stored in the “Item Data Table”. These definitions are used to find the correct name and sprite for an item, as well as the item’s max stack size and description. It also stores the prefabs for these items, for when the item needs to be dropped back into the world from the Player’s Inventory.

These definitions are used throughout the project; in the UI Widgets which display the item’s sprite in the slot, the Inventory System itself to determine which item is being moved to which slot, as well as the max stack size to make sure a slot does not go over the max.



	Row Nam	Name	Description	Thumbnail
1	apple	Apple	A juicy looking red apple, to brighten up anyone's day	Texture2D'/Game/Blueprints_New_/Items/Icons/shiny
2	orange	Orange	A delicious orange, to make orange juice	Texture2D'/Game/Blueprints_New_/Items/Icons/orang
3	kiwi	Kiwi	Delicious, named after a bird	Texture2D'/Game/Blueprints_New_/Items/Icons/kiwi-f

Name	Kiwi
Description	Delicious, named after a bird
Thumbnail	kiwi-fruit
ItemClass	Kiwi
StackSize	3

Item Interaction:

The player interacts with Items using the “E” key. This sets off the “Input Action: Interact” node in the event graph, which determines whether the item being interacted with is a Pickup item, or a Container Item. The item is a “Pick Up” item if it has the Data Table attached to it, otherwise it is a Container item.

This then calls the “Interact With” interface function, which is found in both types of items. If the item is a Pickup Item, then the “Interact With” function is called from the Item Data Component itself. This component calls the “Add to Inventory” function within the Player’s Inventory System and deletes itself from the game. However, if the item is a Container, then the “Interact With” function is called from within the Inventory System, which opens the Container UI alongside the Player Inventory UI.

The player can only interact with these items if they are within range and are directly looking at the item. This is determined by a Spherical Ray-Cast, which is cast along the Camera’s Forward Vector. This section is run from three C++ functions: “C Calculate Camera Line Trace”, “C Calculate Sphere Trace” and “C Set Hit Actor”. These functions first determine a line for the sphere trace to Ray-Cast along, then calculate the Ray-Cast itself and finally set “cLookatActor” to equal the target that was hit. Also, a bit of blueprint functionality allows for a Display Box to show the Item’s name at the bottom of the screen.

Inventory System:

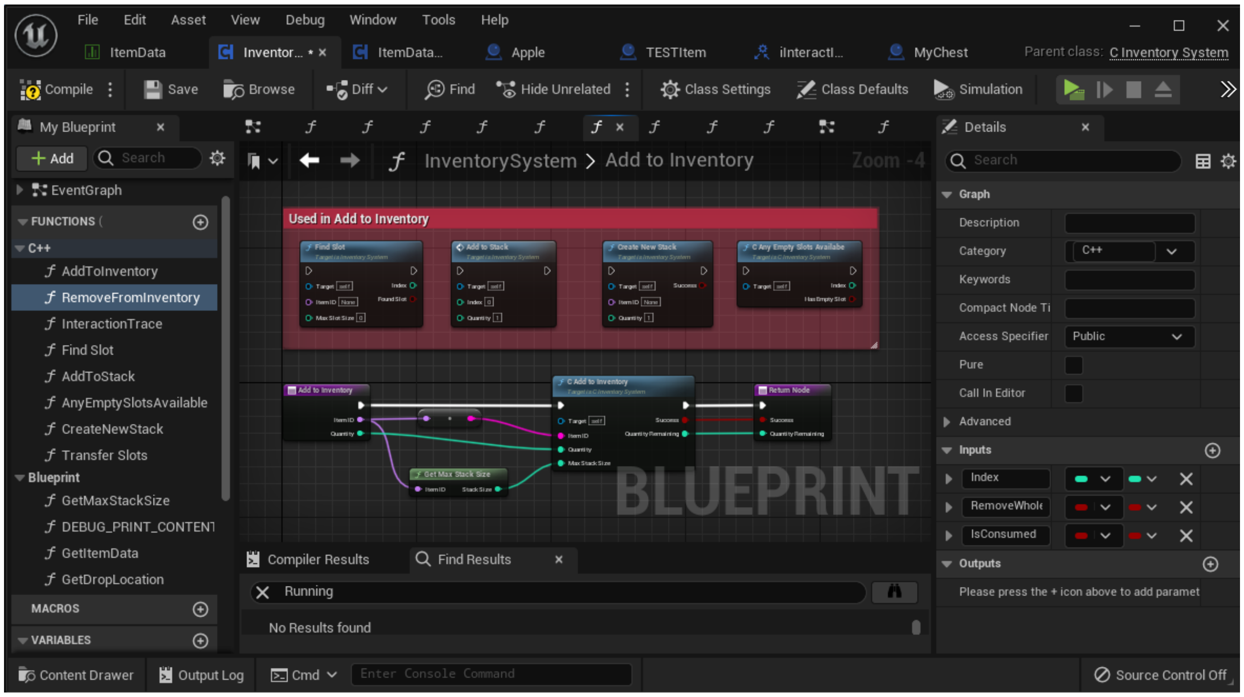
The Inventory System that is attached to both the Player and the Container are exactly the same, except different functions are called depending on whether the Player is using the Inventory System or the Container. All the core functionality of the Inventory System is calculated using C++, which were originally in blueprints but converted over to C++. The original blueprints were much more cumbersome to look through, compared to the C++. As such, this was determined to be the most important section of the mechanic to be converted.

There are numerous functions that are in either C++ or Blueprints inside of the Inventory System. Only small functions that are related to the Blueprinted Data Table remain, and a few custom events related to the UI. The core of the mechanic is entirely made in C++, including the functions: “Add to Inventory”, “Interaction Trace”, “Find Slot”, “Add to Stack”, “Any Empty Slots Available”, “Transfer Slots”, “Remove from Inventory”.

These functions allow for adding new items into the Inventory system, removing them when dropping into the world, adding items to an already existing stack, searching for empty slots in the system, transferring the contents of two slots and “Filling Up” a destination slot from a source slot. These functions use the “FSlot Struct” which contains an ItemID and Quantity, inside of an array to determine what item is in a slot, and how many items are in that slot. The ItemID is the same as the Item Name from the Item Descriptions.

Most of the C++ functions are called after something calls the “Add to Inventory” function; “Find Slot”, “Add to Stack”, “Create new Stack”, “Any Empty Slots Available”. These functions determine whether to create a new slot for this new item or whether to add it to an already existing slot. The other C++ function is “Transfer Slot”. This function is called during Drag and Drop operations, as such it allows for “Filling Up” slots and swapping the contents of slots. The C++ function to delete an item is “Remove from Inventory”. This is called when a “Drop” button is pressed in the drop menu. The “Interaction Trace” function is a set of three C++ functions used to determine whether a player is looking directly at an interactable item.

The Inventory System Is an Actor Component. It inherits from cInventoryComponent, which allows for Blueprint Code to run C++ functions.



Viewing Inventory:

Viewing the Player's inventory is called on the Inventory System event graph when "Input Action: Inventory" is called. This functionality is only used if the actor that it is attached to is a Player. If the Container is interacted with, both the Player and Container inventory will be displayed on the screen, allowing the player to move items between them.

Every time a Drag and Drop operation is complete, the inventory will effectively "Refresh" the UI Widget, causing it to be deleted and replaced with the new data. This allows for the Inventory System to stay updated regardless of how many Drag and Drop operations are completed by the user.

UI Widgets

Inventory Slot:

The Inventory Slot widget is used extensively throughout the different Inventory System's UI. One slot is created per array index that exists in the "C Contents" FSlot Array. The amount of these that exist within the specific Inventory System's UI is determined by the C Inventory Size variable. These slots also contain Blueprinting to allow for drag and drop operations, as the "On Drop" overridable function calls the "Transfer Slots" function, allowing for the player to swap slot contents between two different slots and for filling up the destination slot.

Inventory Grid:

The Inventory Grid widget is created when the player views their inventory, or the player views a Container's inventory. The grid itself stores all Inventory Slots inside of the Inventory System component, as it dynamically adds slots to its Scroll Box component. This allows for a dynamically sized inventory, as any number of slots can be added to the Scroll Box, and they will all be correctly organized.

Player Menu:

The Player Menu widget is a Canvas with an Inventory Grid inside. The main feature of this specific widget is Blueprinting, as it decides whether mouse input is allowed or if the inventory is shown. This is connected to the Player's Inventory System, as that decides when to display the Player Inventory UI to the screen.

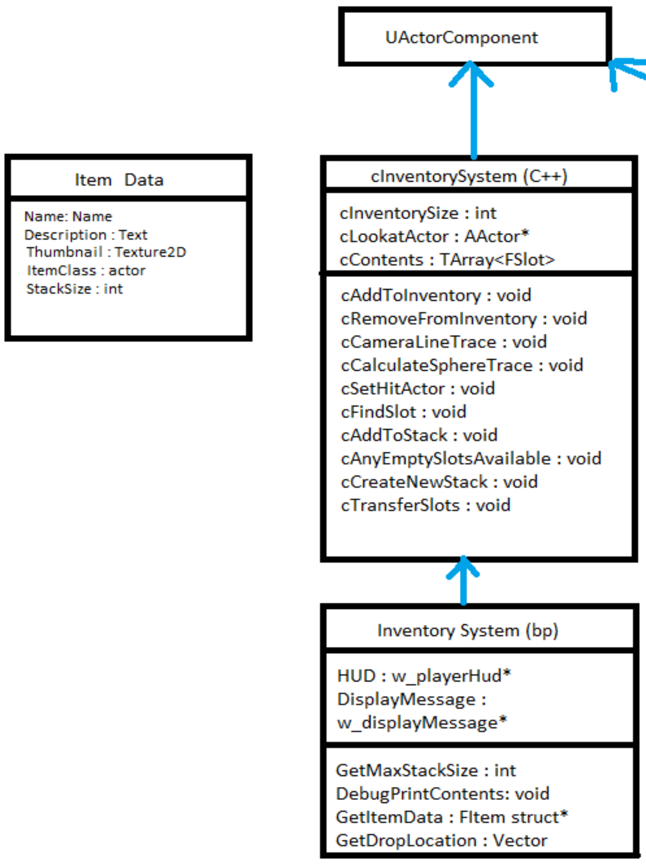
Display Message:

The Display Message widget is a small text box that displays the name of the item being hovered over. The text of this message is determined in the “Interaction Trace” function inside of the “Inventory System” component. If the message is empty, then this message will not display.

Drop Menu (Action Menu):

The Drop Menu widget is a small vertical box with three buttons inside; one for Use, one for Drop 1 and the other for Drop All. The Use button currently has no functionality; however, it could be easily expanded if this was added to a game with Health Bar functionality, or an Energy System. The Drop 1 and Drop All buttons call the “SDropItem” and “SRemoveItem” event, found within the Inventory System event graph. These events call the “Remove from Inventory” function and remove those specific items from the Player’s Inventory System. Also, the items being removed are then placed back int the world.

UML Diagram



Technical Discussion

Most of the core functionality of the project was converted manually from Unreal Blueprints to C++. The only functions that are not converted are those that interact either directly with UI widgets or with the Item Data Table. Specifically, the Data Table is still in blueprints as it is much easier to use in that form, as a Data Table in C++ would be far harder to update for Game Designers. The Inventory System makes use of Spherical Ray Casts, as this method is excellent at detecting which item the player is currently looking at. The Ray Cast channel by default is set to Ignored, as such any new items that are added must modify their collision presets manually. This means that items that are irrelevant to the Inventory System itself will not interfere with the Ray Casts.

A reference of the actor hit is used to check if the actor has a Data Table. This check allows the Inventory System to determine if the actor hit can be added to the Inventory, or if it is a Container item instead. The reference is also used to check the Data Table outside of C++, so that the C++ functions can access the correct ItemID and other variables without having access to the Blueprint Data Table.

If the item hit is a container, (as it has no Data Table), then another Inventory System Component which is attached to that item, will share its Container UI to the screen, and the player’s UI too. The system will not allow functionality of the Player to be used by the Container, as Player Input functions can only be called if the component is attached to a Player Controller. This allows for the same component type to be attached to anything that requires an inventory, making it very modular and dynamic.

Adding to the inventory is a unique function as it first searches for a currently in use stack with the same Item ID, and checks how close that stack is to its max. It will then attempt to add that item to that stack first, before creating a new inventory slot to store the new items. This is incredibly adaptive, as automatically sorting new items into an already player-organized inventory feels very intuitive and adds to the player experience.

Removing from the inventory is simpler, as it either removes the entire stack and places them all into the world or removes a single item from a stack instead. This is a relatively simple process in comparison to Adding to the Inventory, as such it only deals with a single slot. If the slot has no more items left inside of it, it will empty the entire slot.

Transferring Slots is a little more complicated, as it requires both the source and current Inventory system component to function appropriately. This works easily enough using just the Player’s Inventory component, but the two inventory systems are required to work in unison for drag and drop operations to work across the two of them. Also, it requires the “On Drop” override to work appropriately, as it makes use of when the user drags an item from one slot to another. Thankfully, this system works perfectly and can easily move items over from one inventory to another, even performing all drag and drop operations on the way too.

Development

I followed an online tutorial on how to implement an Inventory System using Blueprints and decided to convert the core functionality over to C++. This worked well, however at the start there were a few challenges. The core Data Table was already created in Blueprints and was more difficult to convert over to C++ in comparison to the FSlot Struct and the Array which stored it. Also, the Data Table in Blueprints seems more reasonable from a design perspective, as it is far easier to add to the table in BP than it is in C++.

Overall, nothing in the tutorial covered C++, so I effectively converted large swathes of Blueprints over to C++ entirely myself.

Conclusion

During development, converting some of the Blueprints directly into C++ originally seemed like a daunting task, however after seeing how little lines all those massive blueprints became, it was worthwhile doing for ease of readability. Leaving the C++ functions inside the blueprints themselves also makes it easier to recognize which functions have been converted already, as deleting them after they are put into C++ would have made me lose track of where everything was.

Overall, this project was extremely effective at improving my skills in C++ unreal, as so far, I have only really used Blueprints with unreal. The most interesting thing to me was that you can have multiple outputs from a singular function, as in common C++ coding practice, you can only return a single output, however Unreal C++ makes it extremely easy to return multiple outputs by allowing you to pass things by reference into the function, so they can be changed from inside of the function itself.

I think the project was very successful, as I completed everything that I set out to do. I wanted to create a simple, yet modular Inventory system and make it applicable to almost any object.

References

Apple, Orange, Kiwi and Chest textures and models:

Quixel. (n.d.). *Quixel Bridge*. [online] Available at: https://quixel.com/bridge.

Apple Sprite:

Lorc (n.d.). *Shiny apple icon*. [online] game-icons.net. Available at: https://game-icons.net/1x1/lorc/shiny-apple.html#download [Accessed 6 Apr. 2023].

Orange Sprite:

Delapouite (n.d.). *Orange icon*. [online] game-icons.net. Available at: https://game-icons.net/1x1/delapouite/orange.html#download [Accessed 6 Apr. 2023].

Kiwi Sprite:

Delapouite (n.d.). *Kiwi fruit icon*. [online] game-icons.net. Available at: https://game-icons.net/1x1/delapouite/kiwi-fruit.html#download [Accessed 6 Apr. 2023].

Inventory System Tutorial:

Laley, R. (n.d.). *Unreal Engine 5 Tutorial - Inventory System Part 1: Inventory Component*. [online] YouTube. Available at: https://youtu.be/vHT4MhmwacE [Accessed 6 Apr. 2023].