Gameplay Mechanics Development

Course Reference: CMP 302

-

Student Name: Marc Philippe Beaujean

-

Student ID: 1502932

Table of Contents

Introduction

The mechanic that was developed for this project was an inventory and item management system. Inventory systems are often a focal point in games that allow the player to pick up a large number of different items or resources. A physics handle that allows the player to move objects which are not collectable, was added as another way to interact with objects in the game world. The system borrows heavily from the item interaction system used in the game “Amnesia, the Dark Descent”, which included an inventory that allowed the player to store items, but also enabled interactions with physics objects in the world.

Requirements and Specifications

- *Purpose:* A large number of popular games, across a multitude of genres, have a heavy gameplay focus on looting and item scavenging, usually combined with an inventory to store those items (*Diablo, Player Unknown’s Battlegrounds, Minecraft etc.*). The purpose of this project was to create a similar system with high customisability for designers.
- *Intended Audience:* The resulting product and this document are intended for individuals that want to implement or learn how to create such a system in their games. The system specifically, as mentioned previously, is also created for a designer to use and adjust. Programmers that wish to add additional functionality to the system directly, can do so by accessing the source code. A potential end-user of the system would be a player, that is playing a game where the inventory created for this module has been implemented
- *Product Scope:* The project was initially confined to just including a basic inventory, as well as the mechanics associated with handling an inventory (picking up object, dropping them, etc.). However, this was raised to including a multitude of different item classes that can be used by the designer or another programmer to create new items with ease, which would then integrate into the inventory system seamlessly. Furthermore, the implementation and use of the physics handle was included to create a physics object grab mechanic. This document is also part of the project’s scope, as it aims to inform about the fidelities and specifications of the final product.

Functionality Overview

The mechanic in its core should allow the user to store items and utilise them, by retrieving them from the game environment. Thus, the inventory, it’s UI and a class that allows the player to pick up items, would make up the core components of the feature. Picking up and dropping items make up the main functions that the user needs to execute to be able to utilise this system effectively, if it were to be used in the context of a released and finished product. Supporting components to help demonstrate the feature, would include an extensive UI system, that makes it easier to visualise the inventory. In addition, various classes to specify a multitude of interact-able game objects help demonstrate the system’s functionality and also demonstrate some practical uses for the system.

Design and Implementation Constraints

The main constraint, which is specified by the assessment brief, is that I make the system using Unreal Engine. While I was able to get artistic help from fellow students (see references) to create some visual representations for the items created while working on this system, I was missing artistic knowledge of my own to make appealing and game-ready visuals. This is especially evident when looking at the UI of the system. Another

constraint was the time constraint, which stopped me from developing a fully featured game and more items to use with the inventory. There are also a handful of item interaction mechanics that I could have created to increase functionality. For example, by allowing the user to combine items to generate new ones, in other words a crafting system. In the version of Unreal Engine that I am using (4.17), there are multiple known bugs when using the “Drag and Drop Operation” in correlation with the UML button widget (Answers.unrealengine.com., 2017). Finally, the hardware that is used to develop the system and the coding language (C++) that I would be able to use, were also constraints imposed on developing the system.

Assumptions and Dependencies

In order for the system to be implemented in an Unreal Engine project, basic knowledge of the editor and blueprints is required. While the system handles a lot of elements for the user, in order for new items to be created that can be used with the inventory, the designer will need to know how to program the functionality for these. This can be done by overriding the virtual functions within the base classes of the object class, meaning that the designer does not have to worry about how the object interacts with the UI and the inventory container in general, as this is already done in the background. Furthermore, it is assumed that the computer is able to handle the processing requirements for Unreal Engine and running the system. Besides Unreal Engine, editing the system in code will require Visual Studio or another supported compiler.

Requirements and Specifications for System Components

The inventory needs to be flexible, meaning that it can recognise actors of a specific class as being objects that can be placed in the inventory. It should handle these objects equally in terms of how they can be rearranged within the UI, however specific distinctions between items should be made (for example, items that are not equip-able, like ammunition for a gun). The inventory should allow for the user to move items easily and feel satisfying when doing so. While there are no sounds, informative visual queues should indicate to the player how to interact with items in a way that feels familiar and intuitive. Whenever an event changes the contents of the inventory, the UI should update correspondingly. The player needs to have the ability to add items to the inventory by picking them up in the world and dropping them to make space. Space in the inventory should be limited, however the designer should be able to adjust the amount of objects that can be carried in the inventory blueprint and by using the UI widget designer. Other actors in the world, for example those that allow the player to interact with them using items they carry in their inventory (i.e a door that requires a specific key), should be able to access the contents of the player’s inventory or the player should be able to access the corresponding item. Finally, the player should be able to transfer objects dynamically between their inventory and other actors that also act as item “containers”, like chests. Because of the fact that the inventory UI is created in the Unreal Engine UI widget tool, designers should be able to adjust the visual appearance of the inventory. In addition, a designer should be able to specify what items are in the inventory when the game begins.

Items that are compatible with the inventory, should also have functionality that can be specified for when they are picked up, dropped, used by the player or have a passive effect when they are being carried. For objects that are equip-able, the player should be able to drag these objects to the action bar and receive visual feedback. These items should give visual feedback when they are equipped and have effects when the player wishes to use them. Objects which are limited in use, or are resources that other objects

depend on, should not be place onto the action bar, however they should be accessible to other actors or objects within the inventory if need be. For example, when the player is firing a gun that they equipped through their inventory, the gun class should get an indicator of the amount of ammo that the player has available, before firing. Furthermore, resource stacks should be combinable, so that they take up less space in the inventory (if the player has a batch of 24 bullets and picks up a batch of 20 bullets, the bullet slot in their inventory should contain 44 bullets without the new batch taking up a whole new slot). Each object needs to be represented in the game world, the inventory UI and, for equip-able items, in the state where the item can be used by the player. These representations should be easy for a designer to change and adjust in blueprints.

The quick access bar is a separate item container, that unlike the main inventory, can only hold items that can be used by the player (equip-able items). Using the quick access bar, the player can switch between items quickly and decide what item they want to use without having to open the entire inventory. The player should have the ability to alternate between all the items in the quick access bar using key shortcuts. The player should be able to add, replace and drop items in the action bar, but only if they are an object that can be equipped – otherwise, move or swap requests should be ignored. The number of items in the quick access bar should be editable in blueprints and the widget designer.

Many games that involve inventory and item management, allow the player to explore other item containers, that act like separate inventories. Often, these can be used to allow the player to store items that don't fit in their current inventory or merely incentivise them to explore the environment, by offering a container with a lot of items as a reward for their effort. As mentioned previously, a chest is an example of how this is often implemented in a multitude of games. Alternatively, this can also lead to interaction with other players or NPCs, like a trader where the player can trade items in their own inventory for specific resources. The external item container should be accessible via a UI that is reminiscent of the regular player inventory. It should be possible to swap items between the container and the inventory dynamically, as well as the quick access bar.

When interacting with items in the level, it is often useful to have visual feedback in regards to what objects one is about to pick up or interact with. A visual indicator, that displays text which describes the objects that are currently "in range" of the player, would help with this problem and make it easier for the end-user to utilise the system.

"Crafting" is a system that is becoming increasingly popular in games that make use of inventory-like mechanics. They are particularly helpful in games that require a lot of resource management, as they allow the player to use items which they have in excess, to generate items that they actually require to fulfil a specific task. For crafting to be implemented, objects need to have specific output classes that are generated when they are being materialised by the player. There would need to be another UI window, that allows for the combination of items by dragging them into the specified areas. Crafting has not been implemented for this course work.

Method

When developing the system, each element was developed in the order of its priority (the importance that this element had to making a functional system). The first step for creating the system, was to create the items that the player will interact with (named *CollectableObject* in the source code). These need to have a visual representation in the

world, hence a mesh component was added, which also handles the collision and physics for the transform. A basic line trace system was created, which was sent out from the player's view point. If the actor that is returned from the trace hit was cast-able to the item class, a function is called that disables the item's mesh component and disables it's collision. This effectively removes the item from the physical world, however it allows other classes to reference that same object while it is in the inventory itself. It is also computational less expensive than completely destroying the object and spawning it back in later, when the player drops it.

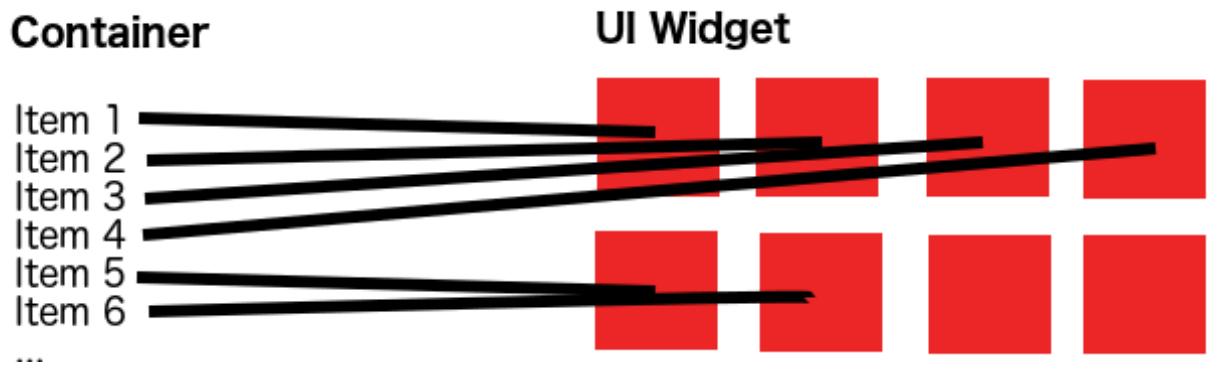
EquipableItem is a class that inherits from *CollectableObject* and as the name suggests, is an item that gives the player some kind of additional ability. For this, additional functions like *OnEquip* and *OnUnequip* are added, which give visual feedback for when the item is selected for use. In the specifically designed *FirstPersonCharacter* class (inherited from the Unreal class *Character*), an uninitialised mesh is assigned to the camera, called *ItemMesh*. When an item is equipped from the quick access bar, the item's mesh is passed and assigned to the mesh component on the player camera. When it is unequipped, that mesh is hidden (usually followed by the equip call of another item). Because all of these functions are virtual, they can be tailored for specific items to support animations or other effects. This is especially useful for the primary function of the *EquipableItem* class, which is *UseItem*. By default, this function is left undefined, however it allows a programmer to define any functionality they like for a given item that inherits from the class, by overriding the function. A lot of the key functions in the system can either be called or defined in blueprints, which makes it more accessible to designers who want to prototype new items and functionality.

Before discussing the main component of the system, the item inventory itself, it is important to talk about how the system has been designed with Unreal Engine's own classes in mind. A reference to the inventory and the quick access bar are created in a custom class that inherits from *CharacterController*. Since the character controller handles most of the UI by default in Unreal Engine, it seemed like a perfect fit to store these references in this class, given that it will require a lot of UI interaction. Furthermore, it naturally avoids a circular dependency between the items and the character itself, because these will need to reference the player when trying to implement unique functionalities (i.e. a gun that is fired by the player will need to know the players viewpoint, location, etc.). The quick access and inventory components are added to the character, then the character controller will establish a reference to them. Another argument for using the controller to handle the inventory and related input, is that it makes switching between characters easier. All that is required, is to de-reference the inventory of the previous character and add a reference to the new one.

The inventory itself (or *ItemContainer* in its base form) inherits from the *ActorComponent* class. Besides the aforementioned reasons, the system was designed in a way that any item with this component could store items and dynamically adapt to the UI, when the player interacts with it. In addition, it would mean that a large amount of code that would be in the player class, is now distributed amongst multiple classes. Collected items are added to a *TArray* within *ItemContainer.* UI widgets have references to the corresponding container that they are representing, with a widget of type *ItemSlot* referencing an item in the array. Using event delegates that trigger when a change occurs for a given container, the widget is notified and updates accordingly. When an item container component is first accessed by a player, a reference to an item of class *EmptySlot* is added to the array, until

the array's size is equal to the maximum number of item slots for that particular container (this is done using the *Init* function from Unreal Engine's *TArray* class). The maximum number of item slots for a given container is determined by multiplying the number of rows and columns specified in the inspector.

While the rows and columns variables don't directly effect the container classes themselves, they are passed to the UI widgets, so that the layout can be adjusted by a designer very easily. The *ItemSlot* widget acts as a visual representation of each item in the container array and holds a reference to the item it is representing, the index of the item in the array, but also the container itself. When the container widget is initialised, each *ItemSlot* widget is assigned to a panel widget that encompasses the entire widget, after which its position in the panel is assigned using the rows and columns variables. This means that the inventory dimensions will dynamically adjust to whatever value a designer sets them to. Furthermore, the *ItemSlot* widget can be customised very easily using the Unreal Engine widget designer.



Depiction of how container items are assigned to the widget slots

The *ItemSlot* widget also holds some of the core functionality for the mechanic, since it is heavily UI based. Unreal Engine supports multiple events for UI widgets, for example *OnDrag*, *OnDoubleClicked*, etc. Because a reference to the container is created when the widget is initialised, functions can be called that allow the user to make changes to the containers within the UI. For example, when an *ItemSlot* widget containing a reference to an item that is not of type *EmptySlot* is dragged onto another *ItemSlot*, that slot can call the *SwapItems* method in the container. This method takes the container of the dragged item as a parameter, so that items can be swapped between different containers dynamically (i.e. the player can swap items between the inventory and the quick access bar). Similarly, the *DropItem* method can be called when the widget is double clicked. As mentioned previously, the *EmptySlot* class is an item that is used to represent areas where items can still be added to the inventory. It is thus specifically designed to be ignored by any input event operations, however can still be swapped with other item slots in the UI.

Often in games, there are items that have multiple uses or are simply there to represent some sort of quantitative resource (i.e. a cartridge of bullets might hold up to 30 bullets per item). By default, the base *CollectableObject* class assumes that an item takes up multiple units per slot, however this can be defined by simply tuning the parameters in the editor. The variable *MaxUnitsPerSlot* specifies how many units a particular object can carry, before a new instance needs to created to carry the excess units. If the *MaxUnitsPerSlot* parameter is set to one, the UI will always display the item as a lone entity. Otherwise, it will indicate the amount of units stored in that particular item in the UI, next to its name.

The system is designed to identify items which store the same type of unit and combine them if possible, to minimize the amount of space taken up in the inventory. When an item is added to a container that does not have the maximum amount of units it can hold per slot, it checks if there are other objects of the same type that it can combine with. Once another object of the same type is identified that also does not contain the maximum number of units, the current number of units contained from the object containing fewer is added to the object containing more units. If an object exceeds the maximum number of units per slot after this operation, the remainder is reassigned to the other object and the process repeated, until the object's units are fully depleted (in this case the reference to the object is reassigned to *EmptySlot*) or all other objects of that type have reached their maximum unit capacity.

The *QuickAccess* class inherits from *ItemContainer*, but is very unique in how it handles the objects it holds. Specifically, it can only accept swap requests when both items are equip-able (the *EmptySlot* class is equip-able, but does not have any use functionality). Another major difference between other container classes and the quick access, is the variable *CurSelectedItem*. This item contains the index of the item that is currently equipped by the player. This is visually represented in the UI widget, by accessing the corresponding *ItemSlot* and changing it's background colour to red. The selected item can be fast toggled using the corresponding input, which either decrements or increments *CurSelectedItem*, un-equips the previous and equips the new selected item. When an item is swapped with the item that is currently equipped, that item is unequipped and the item that is newly dragged onto the slot is equipped.

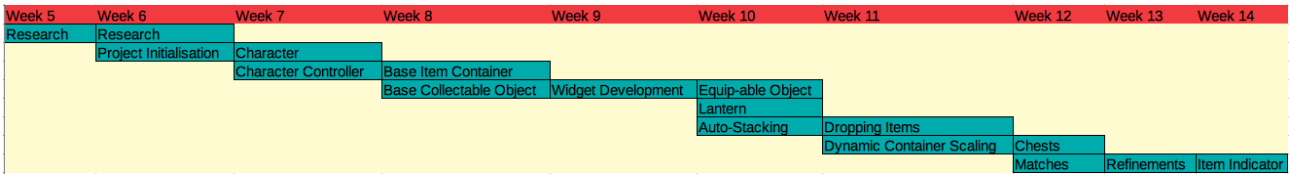
As can be observed by the demonstration scene, a couple of chest objects are added to the scene, that can be opened and allow the user to switch items between the open containers. This shows the dynamic nature of the system, as all that is needed is a base item container component and some simple logic that adds the information from the external container to the current UI widget. This is done, by creating an empty reference to an item container in the custom *CharacterController* class. If the player interacts with a chest object, the controller retrieves a reference to the container of the chest and updates the widget correspondingly. A lantern object was also added, to demonstrate how items can be equipped, unequipped and dropped back into the scene. The lantern can only be used if objects of the type *Matches* can be found in the player's inventory. These are used to the show how the system handles items that have multiple uses or represent a grouping of objects with the same type.

To make it easier for the user to distinguish what items they are about to pick up, a basic indicator was added, which updates a text box widget using an event delegate. The event is triggered whenever a line trace, which is called every tick on the player character, hits a collectable object. The parameter passed to the event is the indicator name of that object. This way, the indicator is updated dynamically, providing visual feedback of possible interactions within the game world.

Development

A waterfall development process was adapted for this project. The initial step, was to familiarize with the Unreal Engine API and learn how to use the visual scripting language provided for accelerated prototyping. This included reworking and exploring some of Unreal Engine's own classes, like the Character Controller, Actor, etc. Once this had been done, the base idea for the project was formed during week 7, from which onwards,

consistent development on the system began. Of course this meant, that several base classes had to be established before the more refined components of the system could be assembled. After the essential components of the system were in place, new features were added that were not initially planned, because there was still additional time available. Overall, the approach to this project can be described as modular, with elements of agile development as well.



Waterfall diagram of the development progress

During development, it was difficult to figure out the best way to avoid a circular dependency between the game objects and the player's character. Eventually, the solution consisted of moving a large portion of the input and item handling to the character controller, while using forward declarations to circumvent occasional conflicts. While it is common to prototype a large portion of a new mechanic in blueprints when using Unreal, the lack of knowledge in both using the engine and the general API meant that it was preferable to learn and start creating the functionality in C++ right away. Of course the final application had to include blueprints, because this is simply the best way to do UI Widgets in Unreal Engine.

Conclusion

When assessing the system critically, one of the main problems can be identified as the visual interface. While it is functional and meets the basic requirements for the system, it is not very visually appealing. The user experience would be highly enhanced, if some additional widgets would be used to make it more colourful, for example a border widget that encompass the individual container representations. In addition, there could be some way for the user to customise the visual appearance and layout of the inventory, that exceeds simply changing the number of columns and rows for the item slots. In addition, it would have been nice to include a way for the player to drop items from containers by dragging them out of the container, like in a lot of games. Unfortunately, due to time constrains and because this was a quite low priority feature, this was not implemented. Additionally, more types of items could have been added for demonstration purposes, to show how versatile the system is.

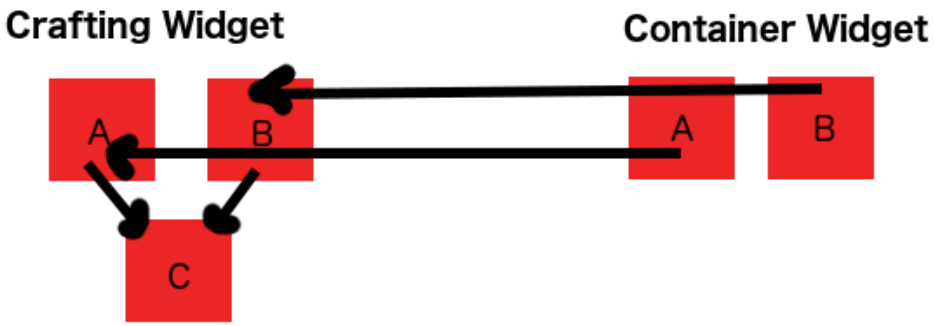
The development approach turned out to be fairly successful. While it is more common to prototype a mechanic using the visual scripting language when working in Unreal Engine, jumping straight into C++ proved to be a very effective way to save time. Due to solid, prior experience with C++, learning the Unreal Engine API was not too time consuming and challenging, even without familiarisation with the functions using blueprints. Due to the time that was saved by implementing the system in C++ and not having to convert from blueprints, additional functionality could be added, which was not initially intended to be part of the system (for example, having the inventory dimensions be dynamically adjustable). In addition, blueprints were still used in conjunction with C++ to create the widgets, which meant that general knowledge about blueprints was still obtained.

When developing the system, quite a few issues surfaced, which lead to suboptimal implementations of some of the functionality. The first being of course, that the widget UI was made using blueprints, which means that it will run less optimally than if it were written

in C++. This also has its upsides, the main one being that it is more accessible to designers, who can readjust the look and layout of the UI much easier. Another problem is the use of line traces. While it is a very effective way of detecting if an object can be picked up or not, it might be more optimal to use a temporary collision sweep on such a short distance, as it is less computationally expensive. Excessive use of the line trace method can slow down the game and reduce performance, so keeping its use to a minimum is desirable. Since the indicator system requires a ray trace every frame, this can be very costly over the course of an entire, finished game. In addition, the indicator is used exclusively to identify collectable objects in the world, but not for example when a player is near a chest that they can interact with. Ideally, the interaction indicator would be expanded upon to encompass everything in the scene, that might be of importance to the player.

Another problem with the mechanic, is that there are several bugs that are linked to the version of Unreal Engine that is used for this project. In version 17, there are known problems with dragging and dropping button widgets, which is what was used to create the *ItemSlot* widget. Specifically, a button can only be dragged with the right mouse button (and double clicked). In addition, if a drag on a button was detected, the button hover state wont reset after the item was dragged to another slot. These issues have been fixed in later (and earlier) builds of the engine (Answers.unrealengine.com, 2017).

The crafting mechanic outlined during the specifications section was not implemented. This would have been a great addition to the feature, because it would allow for items within the inventory to interact in a more complex manner than just being swapped around. To implement such a mechanic, another class would need to be added with its own designated UI widget. In addition, the class would need to know what each item combination should output, which would have been way out of scope for this project.



Potential widget layout for a crafting system

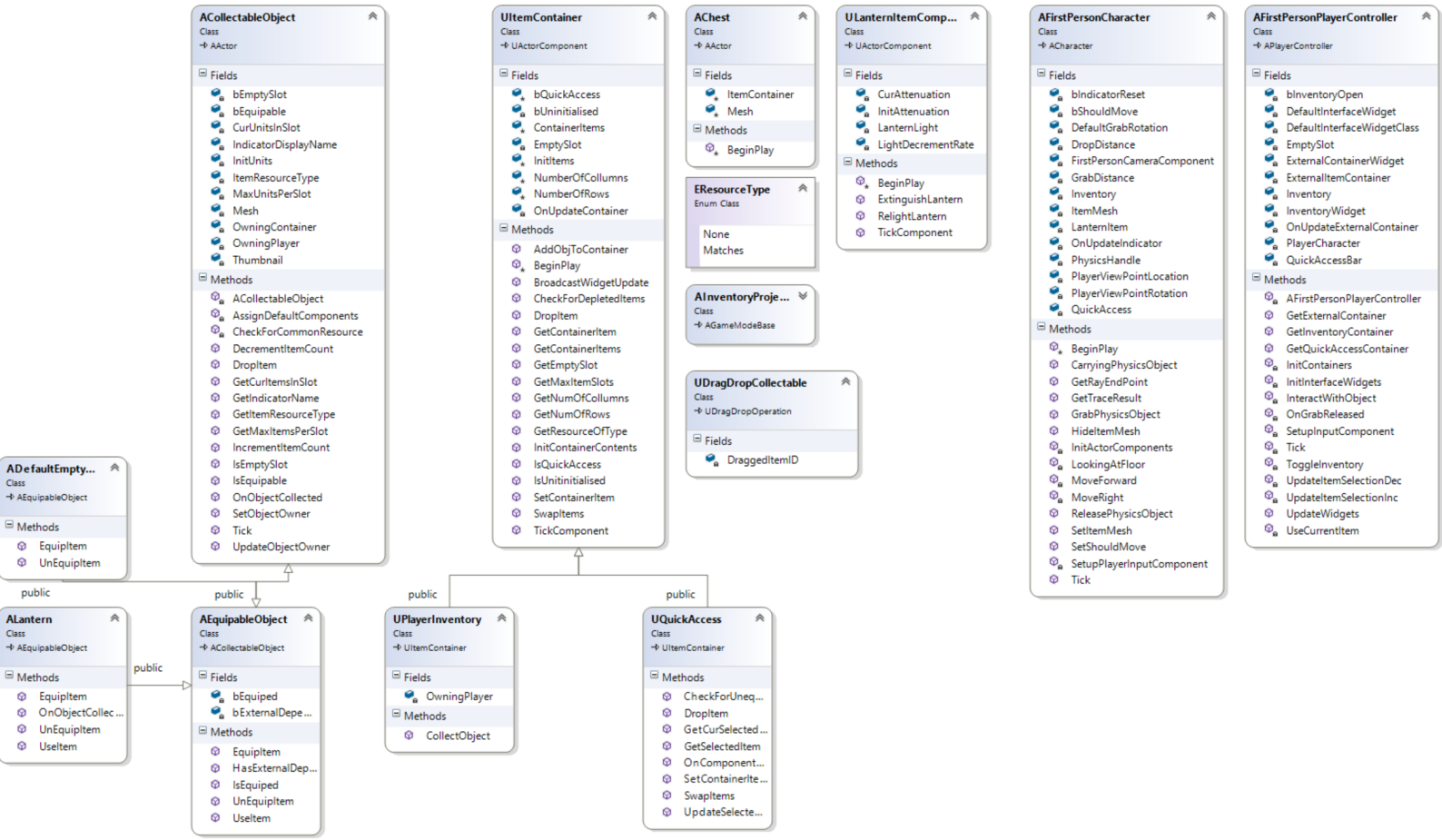
Overall, the delivered product manages to meet all the necessary specifications, while meeting plenty of additional requirements that improve upon the bare components of the mechanic. It was planned to be developed with accessibility for designers in mind, which seems to have been achieved for the most part. While it seems useful to have programmers expanding upon the system, like adding new items or improving item interactions within the containers, there is still plenty of playroom and customisation that can be implemented by someone who does not have a strong understanding of the C++ programming language. One of the strongest features for the system, is that it can easily be implemented and added onto any type of game. The components work well by

themselves, however the developer would need to rewrite the controller and character interactions to suite their respective game.

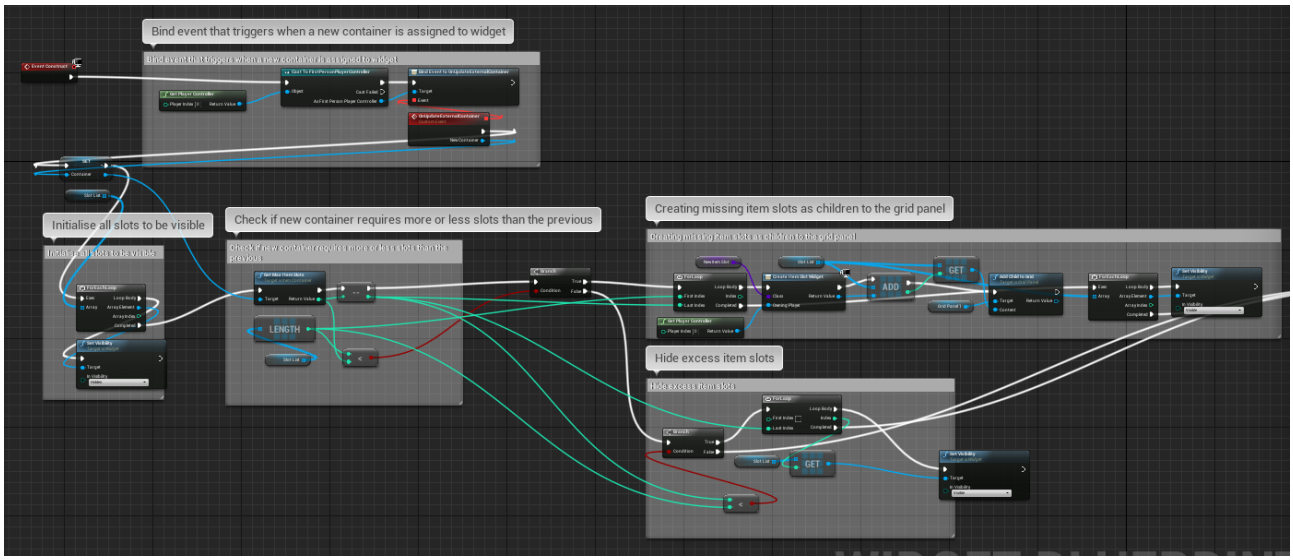
Demonstration Video

https://www.youtube.com/watch?v=sogcDUN4qrQ&feature=youtu.be

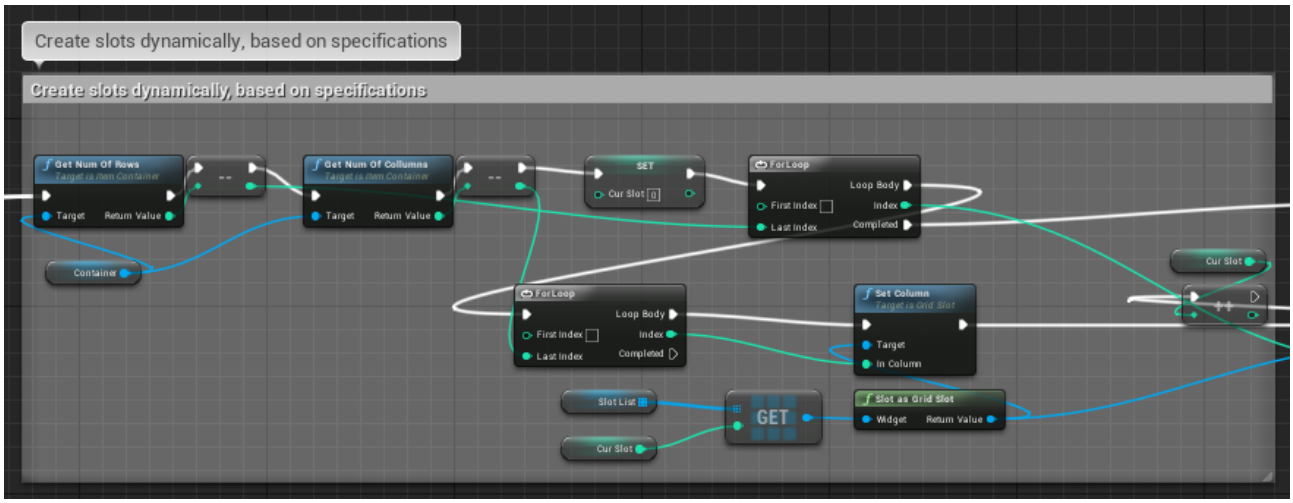
UML Class Diagram



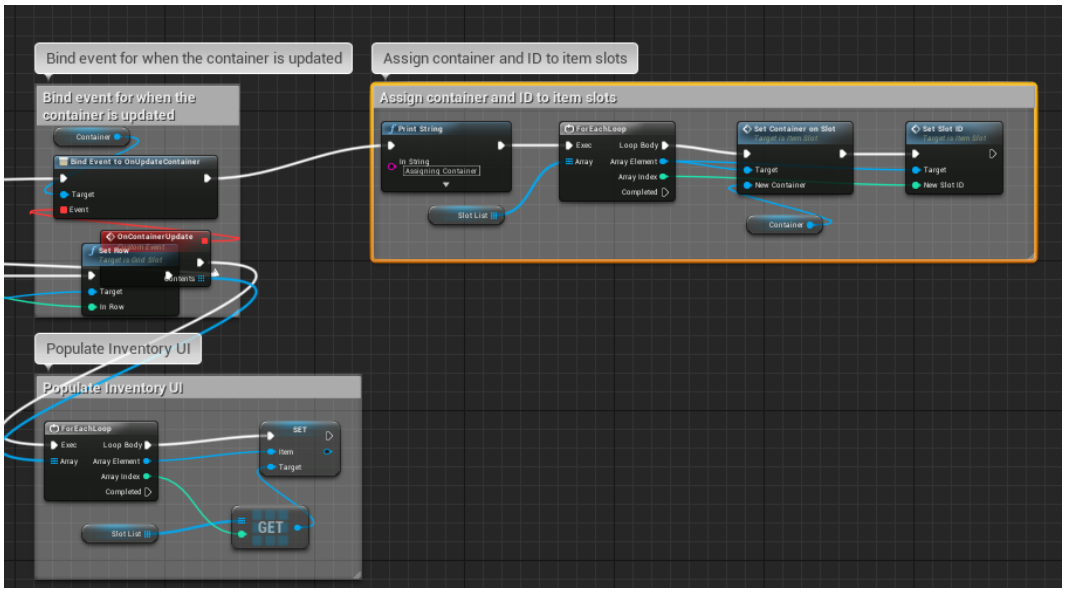
Widget Blueprints



Initialise item slots from container information

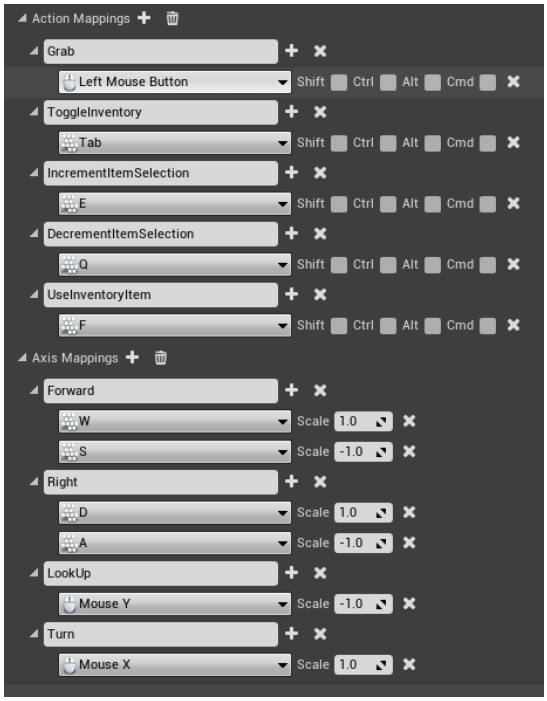


Assign rows and collumns position in pannel for each slot

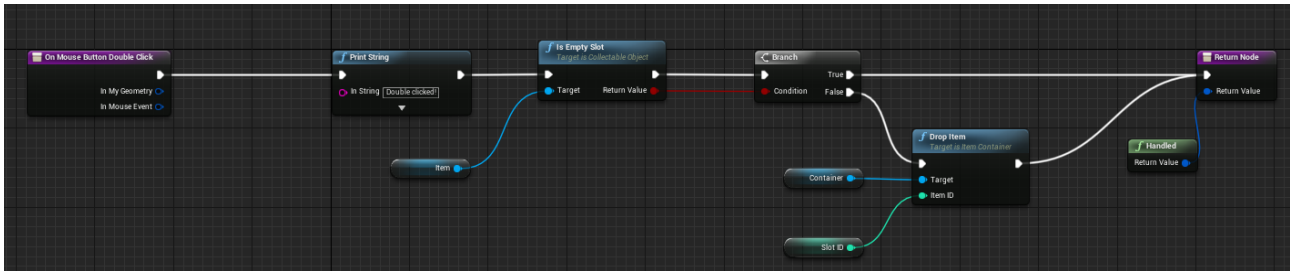


Bind update event to container, then assign container to slots and corresponding item ID

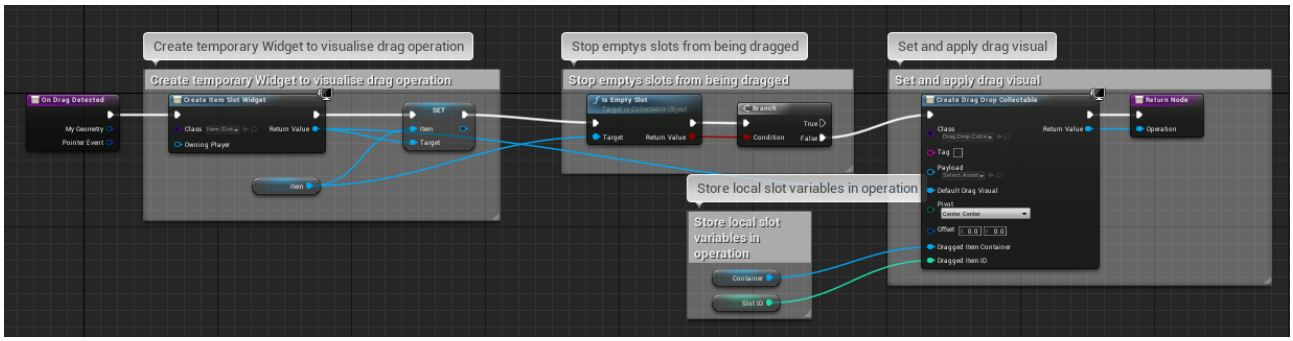
User Guide

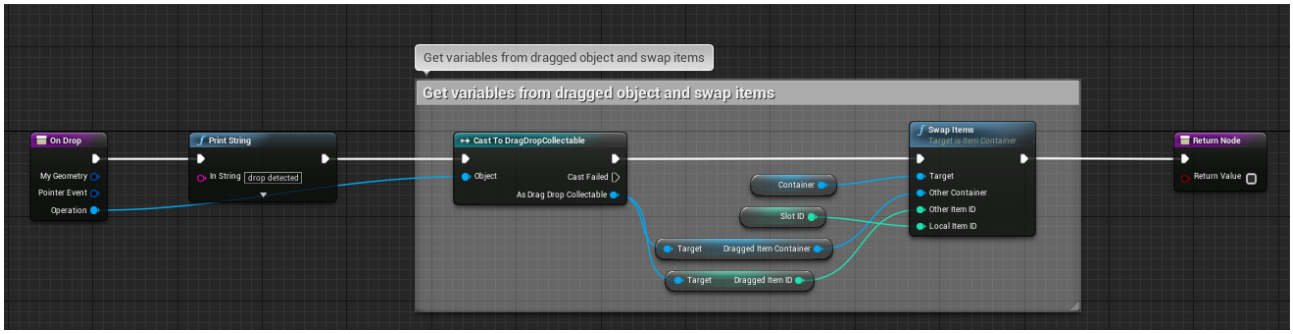


Input in the project settings



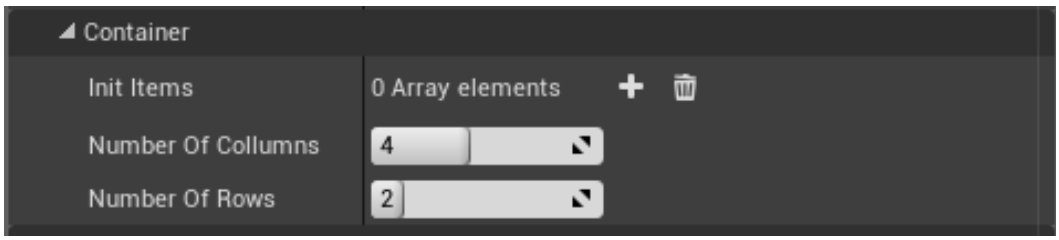
Double click to drop items from the inventory





Drag/Drop item widgets to swap them around

There are many requirements for the system to function properly in any given scene, fortunately a lot of them are already set up by the code provided (i.e. setting up the collision channels of the collectable objects to match those of the player trace, enabling physics on the collectable object's mesh components, etc.). However there are still many customisation options, which can be utilised by a designer. For example, items can be initialised in a container, by adding the specified items to the *InitItems* array in the container component blueprint. This will add the items to the container as soon as it is initialised. As mentioned previously, the size and layout of the container can also be specified in the corresponding blueprint.



The collectable item class also offers many blueprint exposed variables, that allow a designer to tweak and make adjustments easily. For example, the visual elements of the item can be defined, such as the mesh, thumbnail or indicator display name of the object.

Furthermore, the user can determine how many units should be contained by an object when it is initialised and how many units it can hold per slot. The *ResourceType* enum is used to specify if a given object is a resource that is used by the player or other objects. This makes it easier to find when searching the inventory for a specific type of item and doesn't require a cast, which can get quite expensive.

References

- Amnesia: The Dark Descent [Computer Game]. (2010). Frictional Games.
- Answers.unrealengine.com. (2017). *On Mouse Button Double Click Bug - UE4 AnswerHub*. [online] Available at: https://answers.unrealengine.com/questions/700718/on-mouse-button-double-click-bug.html?sort=oldest [Accessed 25 Apr. 2018].
- Docs.unrealengine.com. (2018). *C++ Programming Tutorials*. [online] Available at: https://docs.unrealengine.com/en-us/Programming/Tutorials [Accessed 26 Apr. 2018].
- McConnell, S. (2004). *Code complete*. 2nd ed. Redmond, Wash.: Microsoft Press
- YouTube. (2018). *UE4 C++ beginner tutorial*. [online] Available at: https://www.youtube.com/watch?v=EIBWsvSNRQs&list=PLboXykqtm8dzeQmW8ZgR4gVLUILM0sdVy [Accessed 14 Feb. 2018].

Special thanks to Justin Dolan for making the lantern and key model.