Aria Technical Design Document - Beta

Overdew

Engine

We decided as a team to use Unreal Engine 4 to make our game.

We chose this Engine as it is entirely free to use, has many features for both art and programming as well as being the engine in which we were most experienced.

The engine version we are using is Unreal Engine version 4.26, as it is the most recent version.

Game classes

Our game uses a mix of C++ and blueprint classes to define functionality. Whether we use C++ or blueprints depends on certain factors such as: performance, stability, modularity, extensibility and ease of use (for designers).

For example we use C++ as the base class for most classes for performance, and then use blueprints for extensibility and modularity. We do this with our character classes detailed below.

Characters

In our game characters are any controllable actor in the world that resembles a lifeform.

The player and all types of enemies in our game are characters which derive from the base character class.

Base Character class

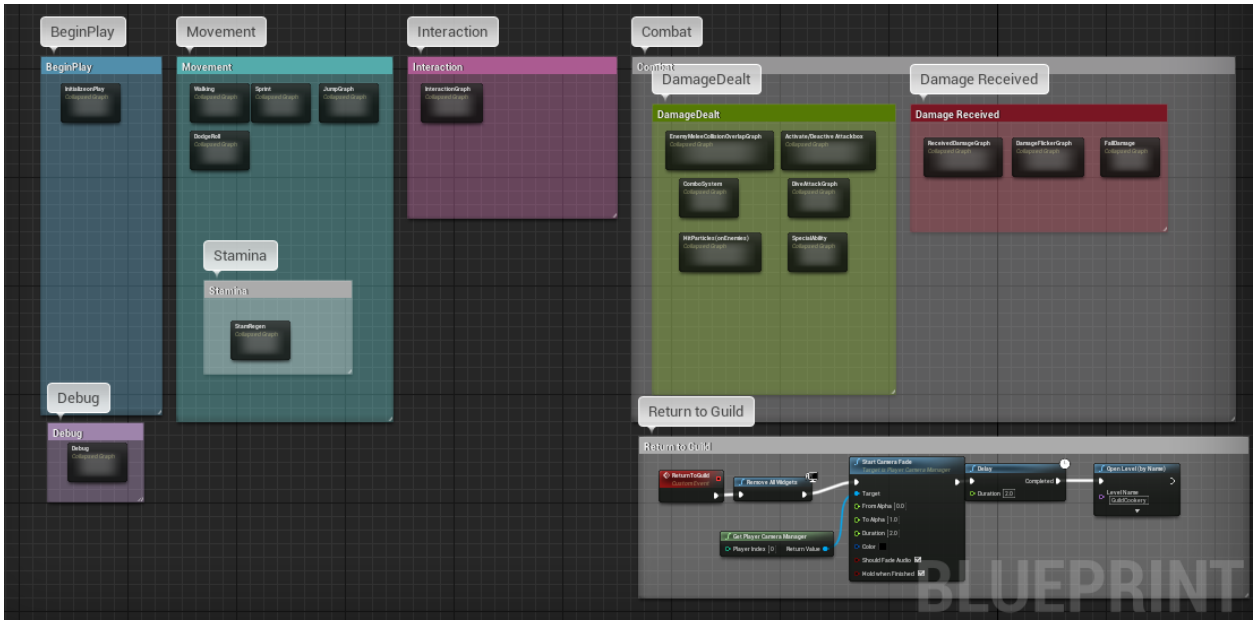
Each character in the game is a derived class of the base character C++ class.

This class holds all the base information that each character should have such as health, whether or not it is dead, etc. While also declaring functions which can be overridden such as the BaseAttack() and Die() functions.

These functions can be overridden for each derived class, allowing us to use polymorphism when handling/calling behaviour of our characters (using something such as using a generic AI controller).

Player class

The base player class of our main character is also written in C++, which we then derive into a Blueprint class to make it easier for our designers to use and add functionality. Most of the logic for our character is in the blueprint class as it's much easier to go in and make fast changes or prototype behaviours.



As you can see there is quite a lot going on in the player blueprint class. A lot of the logic in these graphs hasn't been refactored into a good state due to time being spent implementing the other systems required for the game.

Base Enemy class

Our Base enemy class is a blueprint class that is derived directly from our c++ base character class.

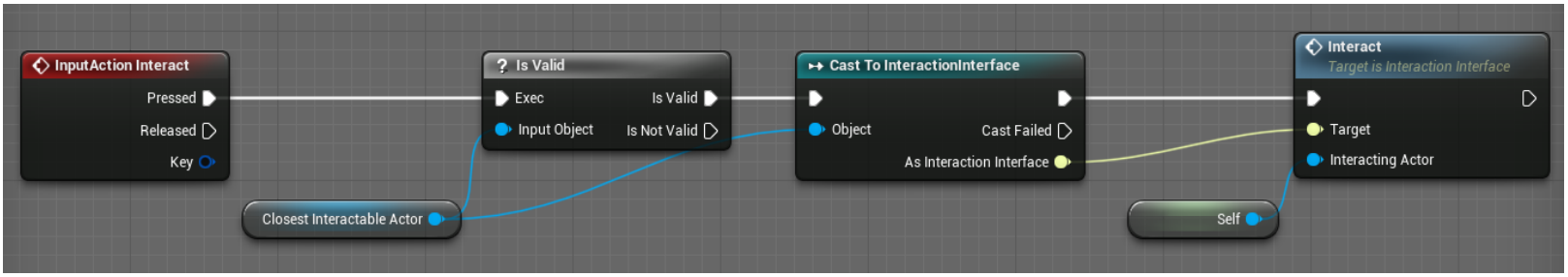
This blueprint class implements the default behaviour for each enemy in our game, such as showing UI, taking and applying damage, dropping items, implementing the Die() function of the BaseCharacter C++ class etc.

Each enemy is derived from this Base Enemy class and goes on to define the functionality for the BaseAttack() function of the BaseCharacter C++ class so each enemy can define how their attacks behave.

Interaction

Interaction is implemented through the use of a C++ Interface. All this interface does is define an Interact() function, which allows any class to inherit from this interface and implement this function.

This becomes extremely useful when you're able to cast any class to this interface to see if it implements it, and if it does then you can call this Interact function. This allows us to define a custom interact behaviour for any actor/object in our world with our player only



having to make one call to the interface to see if the object has the function defined.

It's as simple as sorting the closest intractable object and running the event above to run its Interact() function.

Quests

We use quests in our game to control how our player plays the game, they are used as a means of guiding the player through levels and giving designers the ability to have custom behaviour run based on specific quest events.

Quest Manager

The quest manager is a C++ class that controls and manages the quests in the game, it is responsible for initialising quests and defining which quests are locked and unlocked and which one is the current active quest.

A designer can simply add the classes of the quests that need to be initialised when the game is run, the quest classes are UObjects so they must be defined at runtime from the classes default values.

Base Quest

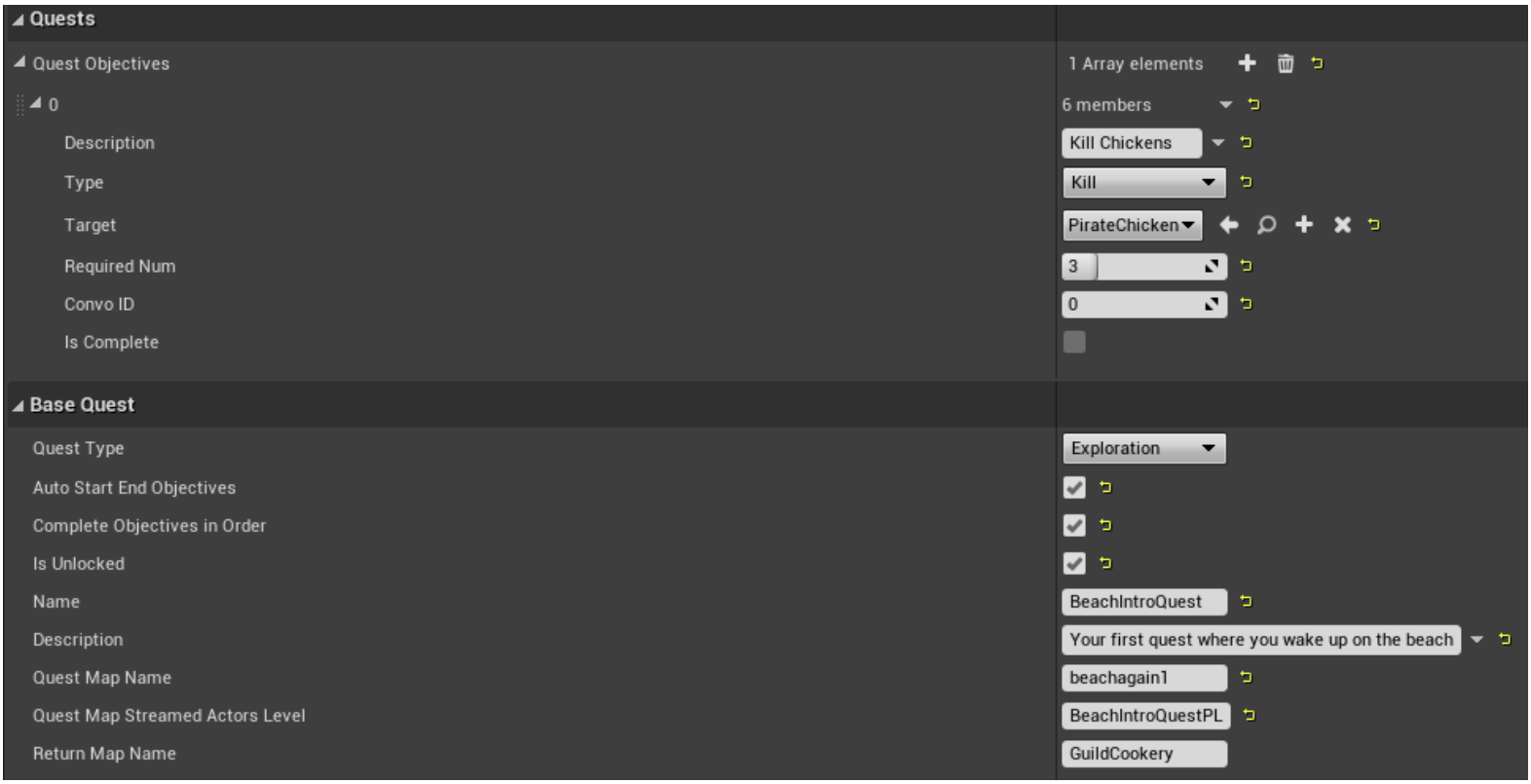
The BaseQuest class is a C++ class that defines the behaviour for quests. A BaseQuest stores an array of "objective" classes (defined below), these objectives have to be completed by the player to complete the quest.

Quests can be created as blueprint classes deriving from the base quest class, and all configurable values are exposed to blueprints from C++. In the quest blueprint you can define an array of objectives that will need to be completed by the play to finish the quest.

The quests have many configurable options:

- A toggle to automate a "go to quest", "return from quest" and "hand in quest" objective for each quest, helping guide the player to and from quests using UI and also saving the designer time so they don't have to define it themselves for every quest.
- A toggle that requires the objectives to be completed in order. This option allows designers to be able to structure quests to a defined order (especially useful for narrative based quests).
- A toggle that has the quest unlocked in the quest board at the start of the game.
- The quest map name, which is the map name you must define so the game knows which map you must be taken when starting the quest (there is currently no validation checking for whether the entered map name exists, the designer must manually make sure they enter the correct name).

This is how a quest is defined. There is an array to populate with the objectives required to complete the quest, as well as all the quest details and configurable options below.



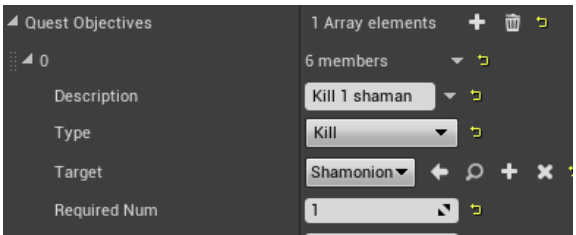
Objectives

Objectives are a struct of information that defines a single objective to be completed for part of a quest. Each objective has an objective type and objective target. The type defines what things will trigger checking the completion of this objective, and the target is the specific class type that will need to be checked to complete the objective.

There are 5 different types of objectives:

- Collect: Checked when picking up items in the world (currently not used in quests).
- Location: Checked when reaching a certain location (overlapping a collision box).
- Kill: Checked when killing an enemy.
- Interact: Checked when interacting with an interactable class in the world;
- Dialogue: Checked when talking to an NPC that has dialogue options.

If you trigger the completion check with the correct class type of the objective then you will complete that objective, e.g. If you have a kill objective with



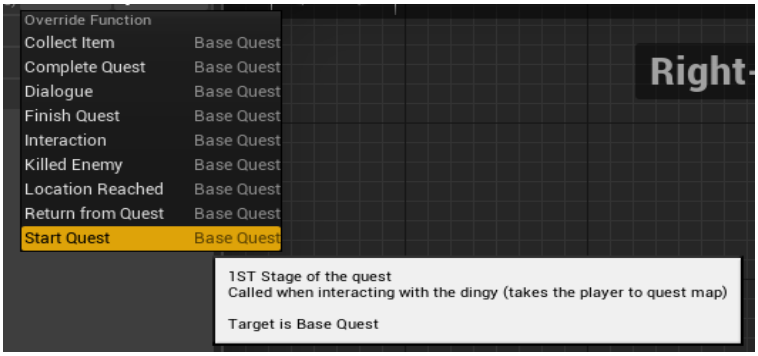
the target type of a chicken, then killing a chicken will complete the objective.

The description of the objective will be shown in the UI while doing a complete in order quest.

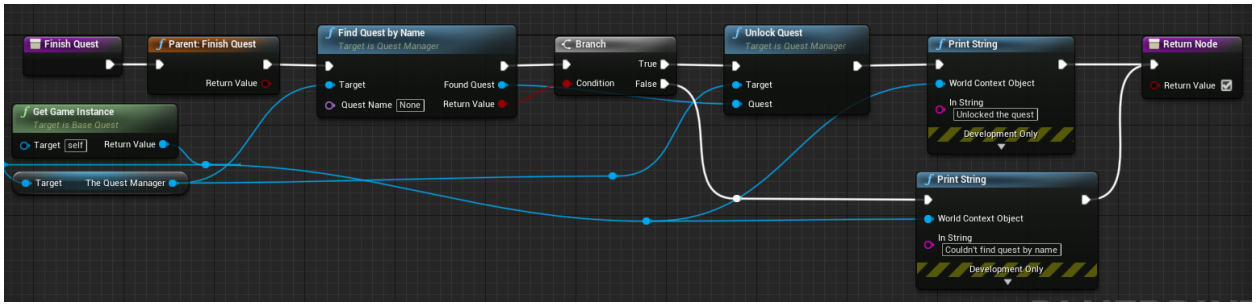
The required num defines how many times a single version of this objective must be done before it is considered completed.

Overridable functions

The quests also have functions called at each stage of completion, which can be overridden in the blueprint to do anything you want.



For example if you wanted a quest to unlock another locked quest on completion you can override the FinishQuest() function and have the quest manager unlock a quest by name when the quest is finished.

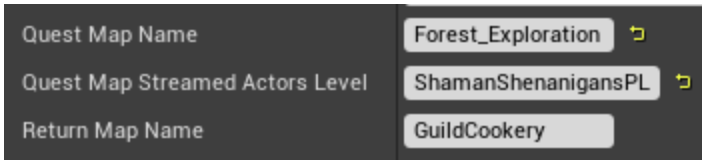


Streamable Level Instances

Quests are also able to be configured to load in a streamable level instance when loading into the level.

This allows designers to create a sub-instance of a level, add any sort of actors or objects into the level, and have those specific actors be streamed in when the quest is being played, allowing you to change/configure the map for each specific quest.

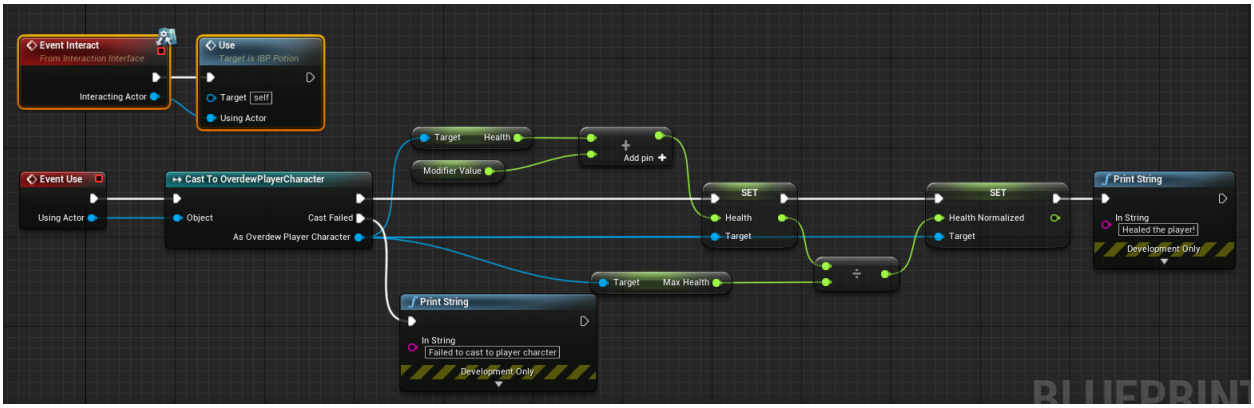
If a quest requires you to kill a Shamonion, then you can add a level instance which contains that actor, then define the quest to load that streamed actors level by giving it the name, and it will stream it in at runtime.



Items

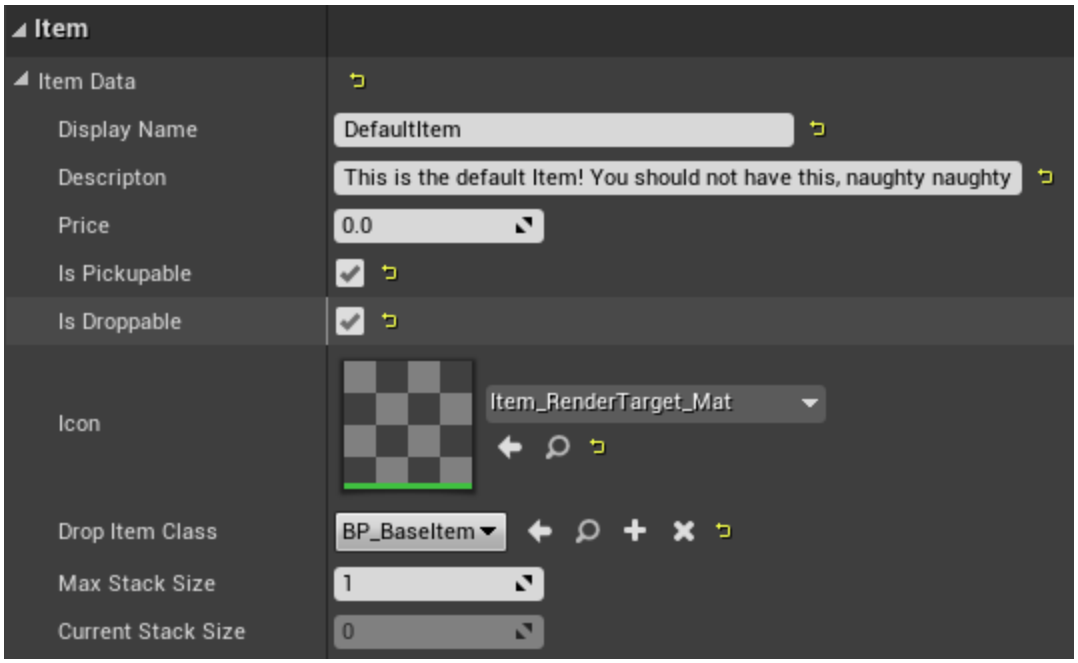
The base Item actor class is a C++ class that defines the functionality for items in our game.

Each item in our game is derived from the base class to be able to implement its own functionality. The item class also inherits the Interaction interface and has an overridable Use() function that allows each item class to have custom behaviour upon being used either from interacting with it in the world or using it through the inventory UI.



We chose to derive our items from the actor class so we can easily spawn them as physical objects in the world while also being able to define custom functionality for each item in the item's blueprint. The above picture is of a Potion item which heals the player upon use.

Item Data

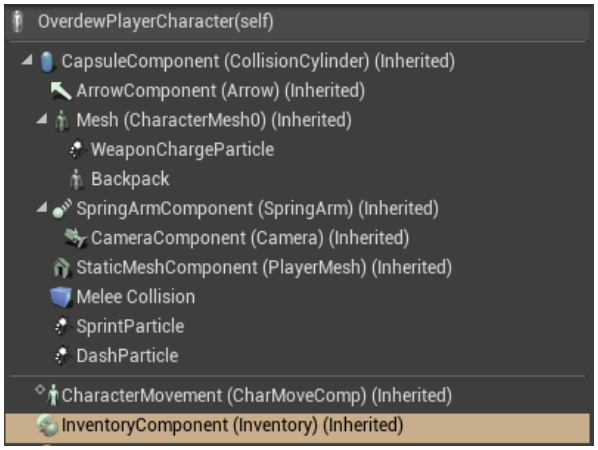


Items store a struct of ItemData, which holds all the variables an Item will use (as seen above).

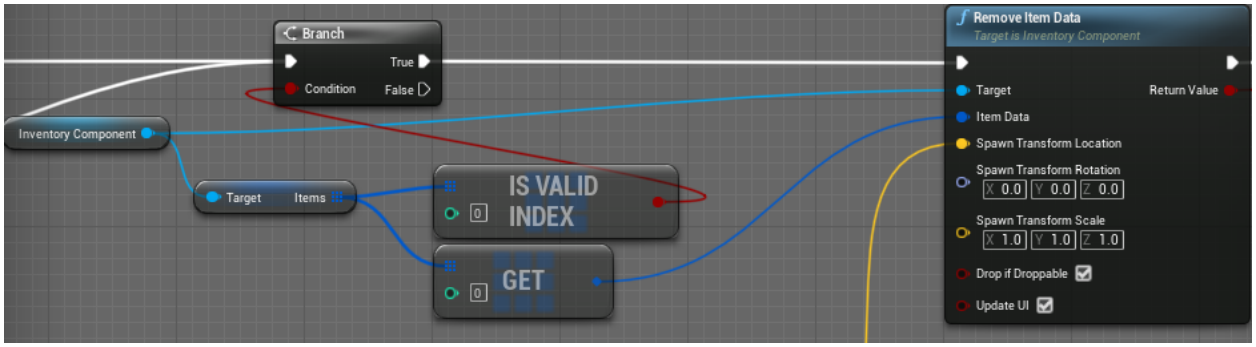
The item data must store a “Drop Item Class” which is a reference to the item’s blueprint actor class for the purposes of being able to spawn the item actor from the item data struct.

Inventory Component

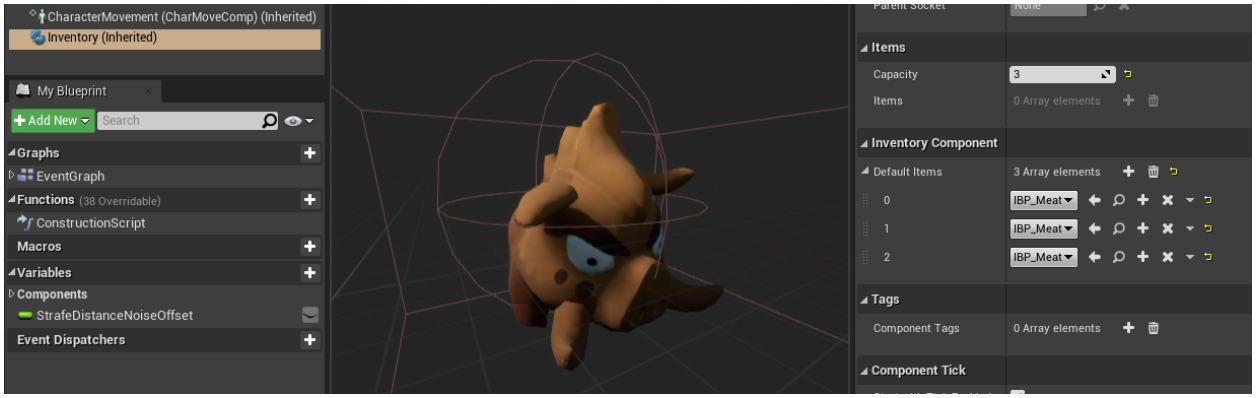
The inventory component is a C++ class that simply handles the storing of Items in our game. As it is an actor component, it can be attached to any actor in our world and be manipulated to do anything we want through the use of blueprints.



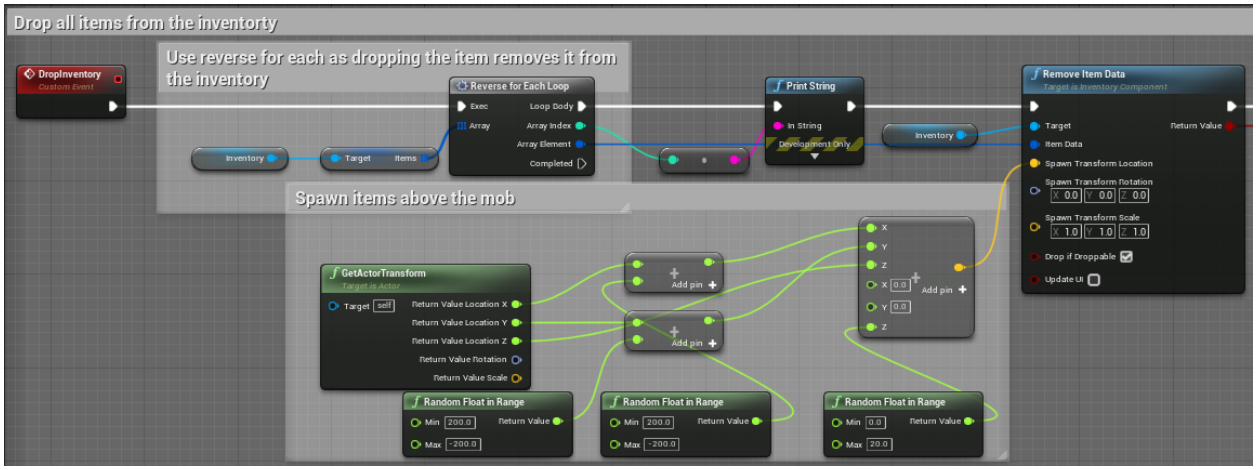
Here you can see the player has an inventory component.



Above is how an Item is removed from the inventory. If it is a droppable item, then the Remove Item Data function will also spawn the item in the world.



An inventory is initialised with items by adding to the array of default items in the inventory blueprint details section. Here you can see this Boar has 3 items of meat by default.

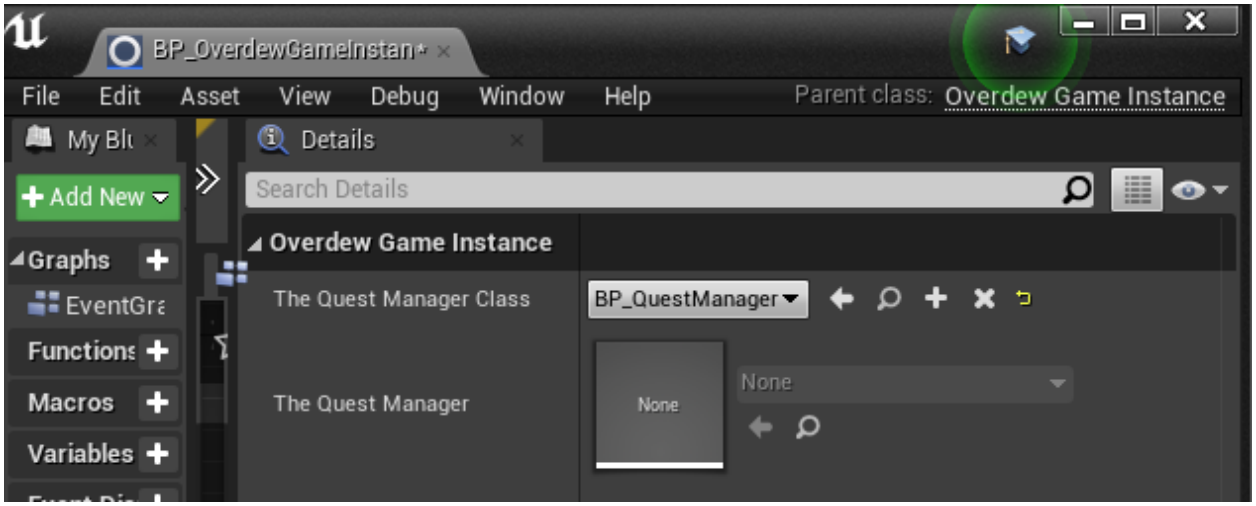


Here the BaseEnemy is defined to drop its inventory on death. So the boar above will end up dropping those 3 meat items upon death if it is killed by the player.

Game instance

The game instance is persistent across different scenes/levels of the game session, so it is responsible for storing data across scenes/levels when the game is played.

The game instance instantiates an instance of the quest manager at runtime, so that quests are properly managed and stored across scenes in the game.

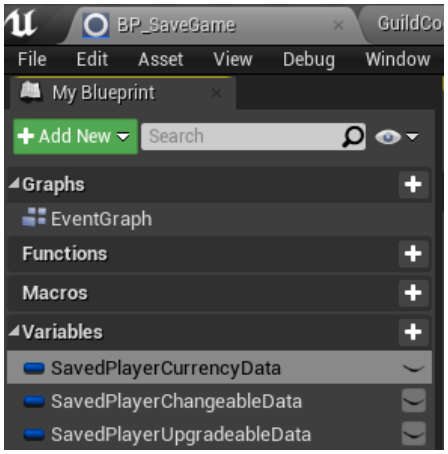


The quest manager also manages game saves and the loading/saving of player data between levels and play sessions.

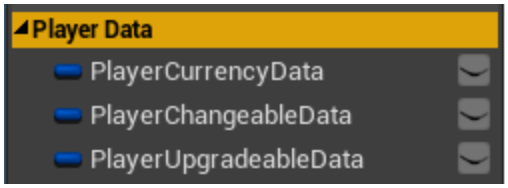
Game saves

We use game saves to keep progression across different play sessions of the game. Unreal has decent built-in functionality for saving data through the use of the “Save Game” class.

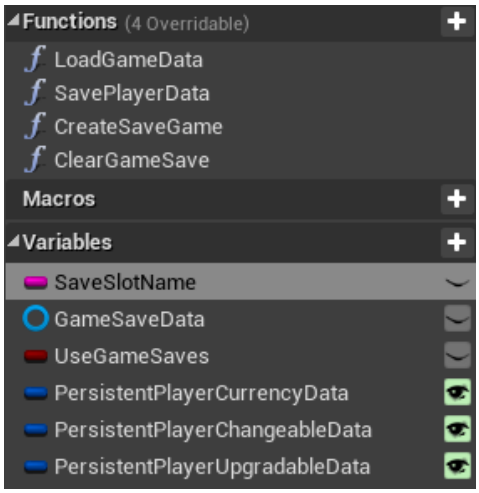
I was able to create a SaveGame blueprint that stores 3 different structs of data for the player. These include the player’s currency data, changeable data such as current health, and upgradeable data such as max health and max flask charges.



There are 3 different instances of these structures saved across different classes.



The player stores its own version of these 3 structs to use directly, so that when the game instance needs to save this data, it simply needs to just copy it to the SaveGame blueprint's instance of these structs before making the game save.



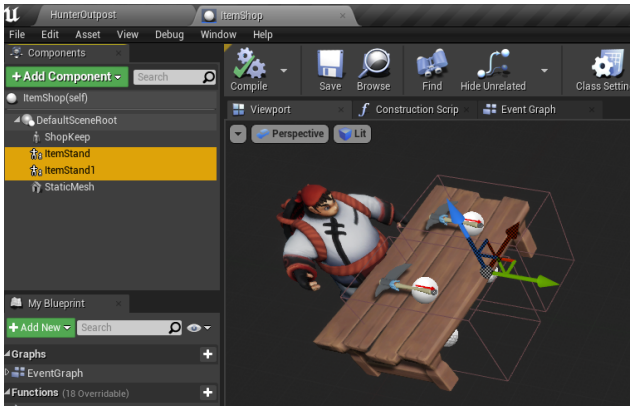
The game instance is responsible for implementing the functionality of the game saves so it can be called from anywhere.

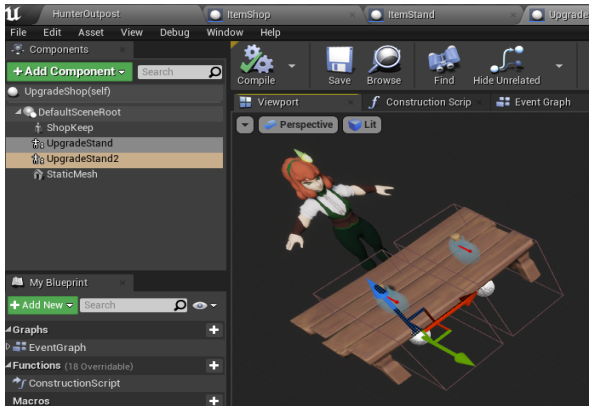
These are the functions that the game instance has defined, along with its own copy of the saveable data that it can use if we have disabled game saves and it needs to keep values persistent across levels.

Shops

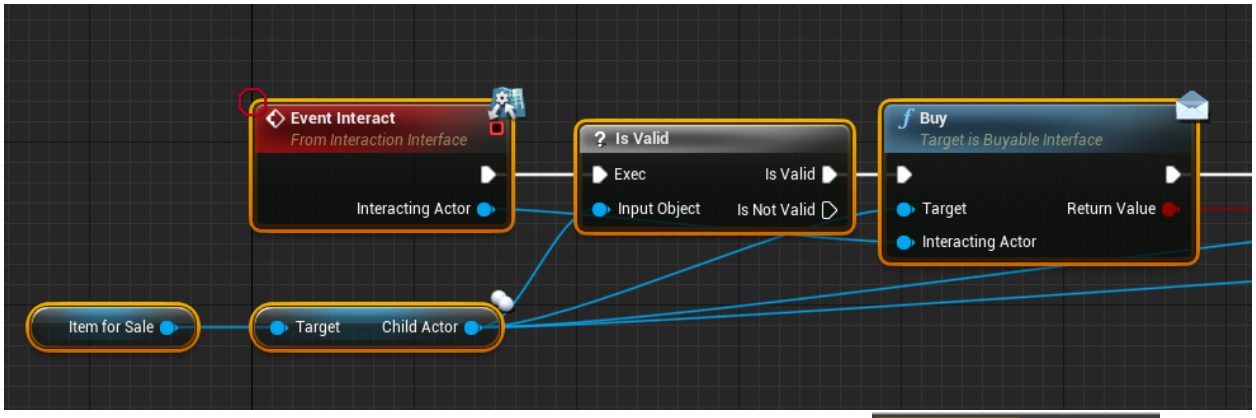
There are two shop classes in the game: The weapons shop, and the upgrades shop.

Each shop holds child actors of either an Item stand, or an Upgrade stand.

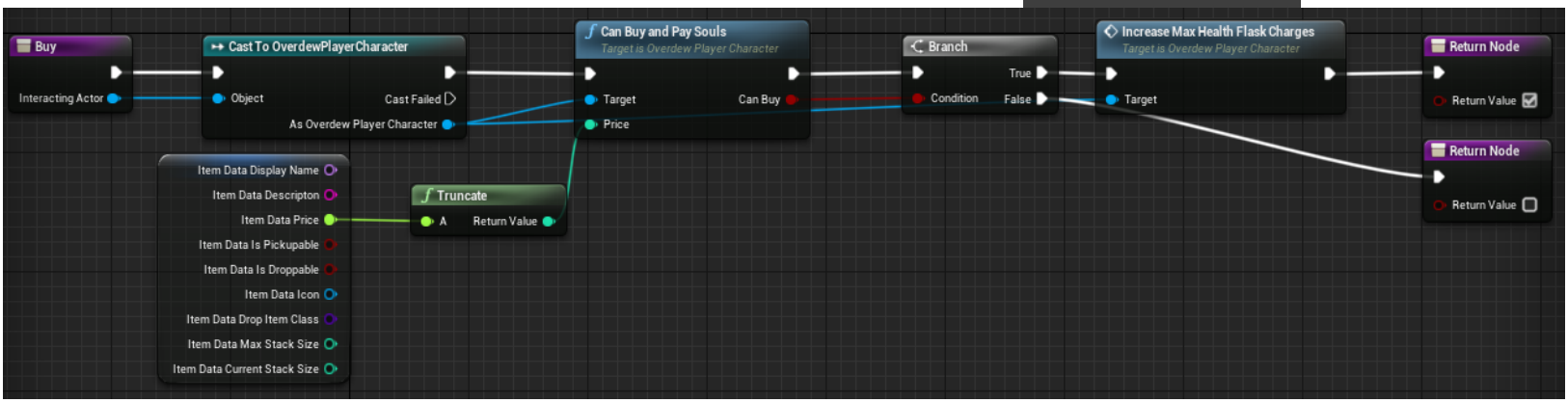




These stands are blueprint classes who also hold child actors of buyable items. They hold the functionality to be interacted with, and call the buy function from the C++ “Buyable Interface” on the buyable child actor Item. As such:



For the upgrade shop we can create items and define what happens when their buyable interface function is called.



The weapon shop works only slightly differently as weapons aren’t derived from the items class, but the overall logic is basically the same. The buy interface for every weapon is implemented in the base weapons class instead as they are all equipped on purchase.