OpenRPG: Inventory

Mar 04, 2021

Contents:

1 Overview **1**

1.1 Features . . . 1
1.2 Project Structure . . . 2
1.3 Inventory System . . . 3
1.4 Equipment System . . . 5
1.5 Save / Load . . . 5
1.6 Migration . . . 7
1.7 Assets Reference . . . 7
1.8 Changelog . . . 7

CHAPTER 1

Overview

Welcome to the OpenRPG Inventory system documentation! This **generic, modular** and **efficient** inventory system will allow you to provide any actor you want with items container functionaly. You dont only get a typical player inventory, you can also have storages, sellers, lootable monsters and much more sharing the same system under the hood!

It has been designed to be flexible enough to suit all your customization requeriments (item effects, containers behaivors, etc) using component inheritance to always construct all your new functionality from a solid base and expand it as you like.

I will try my best to keep this documentation as well organized as possible, to help you understand every aspect of the system. First we will take a look at the project structure and how it is organized. Then we will talk about each system individually, showing you the necessary steps to set up each one successfully, how do they work and how to adapt it to the needs of your game. Finally, I will show you how to migrate the system correctly into you project and how to integrate it completely into your game, covering every needed aspect you need to know ;)

Hint: If you like it dont forget to leave a star on GitHub and also it would be a huge help if you give me all your feedback about errors you find or some features that you would like to be implemented!

1.1 Features

1.1.1 Inventory System

- Component based, add items container functionality to **any actor** you want!
- Comes with **3** types of containers implemented: *player inventory*, *storage* and *shop*.
- **Extremely flexible** to add/modify custom functionality per container type using **component inheritance**.
- Easy to use component **API**: Add Item(s), Remove Item(s), Swap, Split, Stack, Transfer, Drop...
- **Data driven** items from DataTable (**10+** example items).

- Supports different **types** of items (consumable, equipment, material, quest, miscellaneous and loot).
- **Rich features** set per item with **15+ properties** (name, description, type, rarity, usable, stackable, shop value, droppable, etc).
- Create your **own** containers (chests, sellers, lootable monsters, etc).
- Easy to set **custom functionality per item**.
- Supports items **Drag&Drop** functionality (between different containers or in the same container).
- **Built-in** save and load system (disk file and game session persistence).
- Clean UI examples.
- Extremely easy to **integrate** into your project.

1.1.2 Bonus: Equipment System

- Integrated with inventory system keeping **dependencies** to the **minimum**.
- Easy to **integrate** into your character (just **1** actor component).
- **Data driven** equipment data from DataTable (**5+** examples).
- Supports multiple **slot types** (armor, helmet, boots, shield, weapon, etc).
- Equip and unequip from **any items container** (not only player inventory, for example **directly** from storage).
- Supports simple click and **Drag&Drop** functionality.
- Supports **item swapping** from items containers (no need to unequip and then equip).
- **Drop** from equipment slot **directly to the world**.
- **Built-in** save and load functionality.

1.2 Project Structure

This section will cover the project folder structure and a brief description of the main files on it, to help you get more comfortable when looking at the project.

When you download it from GitHub (after leaving a star :P) and you open it with Unreal Engine, the first thing you will see at the root of the project are the `Demo`, `OpenRPG_Inventory` and `OpenRPG_Equipment` folders. Lets quickly see what each one contains:

1.2.1 Demo

This folder contains blueprints, icons, materials, widgets, stuff from the *Third Person Template* and other assets used just to test and explore the system. Lets focus on the blueprints and widgets folders as the other ones are not relevant to the system functionality itself:

Blueprints

Here you will find the implementation of the main game blueprints (character, player controller, game instance, etc). They will help you a lot when you have to implement the system into your own project, because all of them has the needed implementation to make the system work (components initialization, save/load, connection with the UI, etc). The `BP_DemoCharacter` also has a few debug controls to test the system by yourself.

There are also other blueprints used to test some of the already implemented containers, like the `BP_DemoStorage` to help you understand how to create your own storage actors or `BP_DemoShop` which implements a common shop keeper.

Lastly, you can also find a folder with all the classes for each of the example items created. Dont worry about them at the moment because later you will learn how to add your custom functionality to the items. Same applies to the character, player controller and so on. I will cover how to integrate the system with them in each system's section.

Widgets

Here you will find some clean demo widgets like the interaction tooltips, the windows that contains the items grid, etc. But the most important one is the `WB_DemoMain`, because its the widget added to the viewport and displayed in the game, which contains all the other widgets. That is, instead of creating one window for player inventory, another one for storage and then adding each of them individually to the viewport, you just manage them in this main widget, adjusting them easily as you want. When this one is added to the viewport, the rest of the widgets are added too, and its very easy to access them if you need just getting a reference to the main widget.

Note: Its a good practice to create a main HUD widget as the `WB_DemoMain` as explained before. However, this system doesnt care about how you manage your widgets. You will learn how this can be done in further sections.

1.2.2 OpenRPG_Inventory

This folder contains all the blueprints, data structures, datatables and widgets needed to make this system work. In other words, its the core of the system. All the components, base item classes, interfaces, functions library and so on are located under the `Blueprints` folder, which as you can suppose, is by far the **most important folder** in the whole project.

There are other folders like the `Structs` and `Enums` that contains all the data structures and extra data types used by the system, as well as a small items database using this data structure in the `DataTables` folder.

Finally, the widgets folder contains the most important widgets of the system: the item slot and the slots container grid. Both handle most of the UI functionality with the components. Here you will also find the tooltip widget, showed when the mouse is over an item slot to show information about the item it contains.

1.2.3 OpenRPG_Equipment

Finally, as the name suggests, this folder groups all the equipment functionality, with the same schema as the `OpenRPG_Inventory` folder. This system was created with the design of the inventory system always in mind, so you will find a lot of similarities when diving into the folders, but with much less complexity.

1.3 Inventory System

In this section we will explore the inventory system and how to use it. Let me first explain you that when this project started, this system was just purely that, an inventory system that was attached to the main player character.

However, as long as my programming knowledge was improving, I decided to generalize this system to make it what it is now, an **items container** system that can be attached to any actor, not just the player character. To achieve this, I have made an extensive use of **component inheritance**, which is the key to get **flexibility** and **generic behaivour** at the same time.

1.3.1 Blueprints

Adding new container types

To create a new type of container different from the example ones, you have to follow this simple steps:

1. Create a new entry in the `e_ContainerType` enum under `OpenRPG_Inventory/Enums` folder.
2. Create a new items container grid widget to show the items coming from your new type of container. To do that, just open the `WB_Main` widget under the `Demo/Widgets` folder. Grab a `WB_DemoWindow` widget and move it to the desired place. Now under the **Settings** category of that widget, modify the value of the `ContainerType` variable with the type of your new container.
3. Go to your player controller and in the functions `GetContainerWidget` and `GetMainContainerWidget` from the `BPI_ContainerWidgets` interface add your new widget window and its container to the return value.

Thats it! Now you have a new widget that will be updated with the containers of your new type. The new containers will use the player controller to extract the necessary references from the widget you have added. Lets check now how to use this new type creating a new container!

Creating a new container

To create a new component, just navigate to the `/OpenRPG_Inventory/Blueprints/Components` folder and you will see there is a `BP_ItemsContainer` which is the **base** for all the containers you create, apart from the example ones. To create a new one, just right click that component and select **Create child**. Open that new component and now you will see you have access to a few variables under the **Settings** category in the **Details panel**:

TODO: Extend and correct variables names

- `ContainerType`: This sets the type that your new component is going to have.
- `ContainerSize`: The amount of default slots that the container will have when initialized.
- `SlotsPerRow`: The amount of slots that should be displayed per grid row in the UI.
- `DefaultItemsRow`: Identifies a set of items that should be added automatically to the container when its initialized. Rows of this type can be created in the `OpenRPG_Inventory/Datatables/DT_ItemsList`.
- `AdaptSize`: If true, the `ContainerSize` value will be ignored and the size will be adjusted to the size of the items set you defined for your `DefaultItemsRow`. If the row is not valid then this setting has no effect.
- `StackSplitAmount`

Just customize them as you need. Then its time to customize some of your new container's behaivour by overriding a set of functions that will be called under certains events:

TODO: Add the rest and explain overridable functions

- `OnSlotClicked`: Called when a slot is right clicked in the UI. You can see some implementation examples in the example components. In the case of the `BPC_PlayerInventory`, it executes the functionality of using an item, and in the case of the storage, it moves the item in that clicked slot to the player's inventory.
- `HandleSlotDrop`: Called when an item slot is dropped from another container is dropped into an slot of this new container.
- `HandleEquipmentSlotDrop`: Same idea as before but only executed when this drop comes from an equipment slot.
- `SaveContainer`: Add extra behaivour when saving data from this container. For example, in `BPC_PlayerInventory`, the player's money is also saved when calling this function.

- `LoadContainer`: Same idea as above but for loading purposes.

Adding items container to an actor

TODO: Explain the process and the initialization pipeline.

Example containers

TODO: Explain each of the default containers.

1.3.2 Interfaces

1.3.3 Data Structure

1.3.4 Widgets

1.4 Equipment System

1.4.1 Blueprints

1.4.2 Interfaces

1.4.3 Data Structure

1.4.4 Widgets

1.5 Save / Load

In this section we will be covering one aspect that is common for the inventory and equipment system: **the built-in save and load functionality**. It is important to understand how both systems (specially containers) handles their data to be persistent **even when the game is closed** and how that data is **organized** in the blueprints that take care of this work.

When we talk about save and load functionality, **2 aspects** must come to your mind:

- Data persistent **across different maps** and linked to the **current game session**.
- Data persistent **across different game sessions**.

To explain this concepts better, lets see which blueprints are related to each one.

1.5.1 GameInstance

This blueprint has the property of being persistent across maps. That means that if your player travels to a new map and the items container actors in the old map gets destroyed (and also the player), then all the data they holded in the old map would be lost, but **this blueprint is the same instance always across different maps**. Because of that, its the perfect place to keep all the data persistent even if the player goes to a different map.

In this project, the `BP_DemoGI` is used with the purpose explained before. Further we will see the functions, interfaces and variables it uses to act as we want.

However, all the save and load functionality isnt complete, because if the game is closed then the game instance is also destroyed, so the saved data there would be lost when we open the game later. We would need something like an external file on the hard drive to store the data we want to keep across different game sessions. For that reason, we need another kind of blueprint, which is the `SaveGame` class.

1.5.2 SaveGame

This blueprint is a special engine class created for the purpose we need. If we want to save or load something from an external file, Unreal can create a file with the `.sav` extension, which can hold all the data we need to keep across game sessions. The interface with that file is just an instance of the class `SaveGame`, so we just need to create a custom instance and add there all the variables we need to keep, just like a normal actor.

The instances created for this project are:

- `InventorySave`: It holds all the variables needed to keep all the items containers and the player data related to the inventory functionality.
- `EquipmentSave`: Holds all the variables needed for the equipment system.

Attention: Please give a look at the functions used to interact with this type of class here

Note: You may be wondering why the `GameInstance` blueprint used in this project is part of the demo content and doesnt come in the `OpenRPG_Inventory` folder if it is such an important blueprint for the save and load system to work properly. That is because you will probably have your own `GameInstance` already implemented with your own logic, so I prefer to give you the tools to easily integrate the logic into your own class rather than obey you to use the `BP_DemoGI`. Of course, if you are not using a custom `GameInstance` in your project at the moment, you can perfectly use this one.

Now that you understand the main blueprints used by this system and why they are used, lets see how they are connected between them and how the containers and the player sends and receive data from them.

1.5.3 Interfaces

To make the communication as transparent as possible, I have provided a few blueprint interfaces to abstract the communication between the components and your game instance. These interfaces are:

- `BPI_ContainerSave`
- `BPI_PlayerInventorySave`
- `BPI_EquipmentSave`

Each of them have 2 functions with the syntax **SaveXXXX** and **LoadXXXX** depending on the type of data they save. For example, `BPI_EquipmentSave` has the functions `SaveEquipment` and `LoadEquipment`, while the interface `BPI_ContainerSave` has the functions `SaveContainer` and `LoadContainer`.

Those functions have a common input parameter called `SaveName`, which is a string label that **identifies** the data that you are going to save or the data you want to load. Think of the data as a **package** and this string as the **ID** of that package. This concept is very important because when you want to save or load any actor data, you need to pass this string to let the `GameInstance` identify what data package should be loaded or how it should be saved. You can just create an editable string variable in the actors that have any component data you want to handle (storage actors, the player, etc), and then use it when you save or load any data.

Note: In the BPI_ContainerSave interface you will notice there is an extra boolean input called DiskSaveable in the SaveContainer function. That is used to avoid saving containers that you would not want to be saved on disk. For example, you probably dont want to save the loot from a monster in the disk, but a storage or the player inventory should be.

There is also an extra interface called BPI_SaveManager. This one is used by the GameInstance to communicate with the SaveGame classes. Following the same schema as the other interfaces, it has the functions SaveDataToDisk and LoadDataToDisk. However, this functions doesnt have any input or output parameters, they are used to save or load a state of your game to the disk. Because of this, LoadDataToDisk can be called only when the game starts to dump all the saved data to your game instance, and then work with the data there. By the other hand, the SaveDataToDisk can be called when you reach some checkpoint in your map, or just when a UI menu button used to save the game is clicked.

If you are still reading this, congratulations! At the moment, you know which classes are used in the saving and loading process and why. Now you just need to add the interfaces commented above to your game instance and implement them with the data structure variables you prefer. My recommendation would be to use a **map** variable type, which is also called as **dictionary**. That is because the data stored has the structure of SaveName -> Data, which is the structure that this data type is made for, and the blueprint code needed is just a few nodes and works very fast. Just give a look at the BP_DemoGI blueprint and you will see how I have used it!

1.6 Migration

1.7 Assets Reference

1.8 Changelog

1.8.1 Improvements

- Reworked save and load functionality using GameInstance as the manager to keep data across levels and dump desired data to disk. Before, components were able to save data directly to disk, but its better to let the GameInstance handle that for flexibility.
- Added function EquipFromContainer to abstract container origin (not only player inventory), so now an item can be equipped directly from any container like storage for example.
- Added function UpdateEquipmentVisuals to encapsulate all the visuals functionality.
- Removed OnSaveExtraData and OnLoadExtraData events for child container types, now to expand save and load functionality (for example player's inventory money) just override the SaveContainer and LoadContainer functions to add the necessary code.
- Added OnEquipmentSlotDrop event with default functionality already implemented (unequip basically), but can be overriden to suit specific needs.
- Added function CanBeUsed in BP_BaseItem to remove the UsedSuccessfully boolean. Now its easier to override if an item should be used or not.
- Event OnItemUsed in BP_BaseItem is now called OnStartItemEffect.
- Added events OnFinishItemEffect and OnItemUseFailed in BP_BaseItem.

1.8.2 Bugfixes

- Fixed bug where non equippable items were used when dragged to an equipment slot.

1.8.3 TODO

- Add widget to show keyboard controls and shotcuts when using the system.
- Improve the example scene.
- Add some real functionality to an item instead of just explaining how to do it.
- Add functions to dynamically increase the amounts of slots in container.