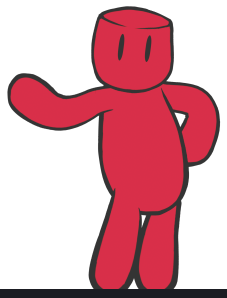
Grid-based Inventory Engine, User Manual

Manual v1.0.0



1. Introduction

Thank you for purchasing an Aldebaran Project product! With this document you will learn how do the following things:

- Set up this package,
- Add custom inventory items to your project.

You will also learn:

- About all the main systems of the package, including the built-in auto-sort function, the API for adding and removing items to the inventory, and how to extend this package.

We also provide some video documentation and tutorials at:

- https://www.youtube.com/channel/UCZI_GLg5CzcFDuXeN0J_-Rg

If you have any issues or suggestions, please contact us at aldebaranprojectdev@gmail.com, we will be very happy to receive your message and will respond as soon as we can. Do also tell us if you've made something cool with our software! It means a lot to us.

1.1. Credits and thanks

Software code and user manual written by Vinicius Krieger Granemann, inventory art by William da Silva Pereira.

A huge thanks to Matheus Novelli de Oliveira. for taking his time to help testing the inventory system and this manual, and to our friends and families for their support during the development of this project.

2. Implementation

2.1. Setting up this package

Create an empty Unity project or open an existing one. Inside Unity, go to *Window > Package Manager*. Inside the Package Manager window, click in *Packages* and select *Packages: My Assets*, and click *Download* if you have not yet downloaded it, and then click *Import*. In the *Import Unity Package* window, click *Import*. After the loading screen is finished, you should have a folder named "*Aldebaran Project*" inside your Assets folder.

If you have not yet imported Unity's TextMeshPro package, some errors will show in the console after loading the scene *InventoryExample* inside *Aldebaran Project/Grid-based Inventory Engine/Samples/Scenes*. A prompt will appear asking you to import TMPro Essentials. Accept to do so and the errors should go away.

Everything that will be shown here is present in this example scene. Every folder mentioned from now on will be referenced relative to *Aldebaran Project/Grid-based Inventory Engine*.

2.2. Creating custom items for your game

Inside the folder *Samples/Resources/Json*, you have a file called *items.json* which contains all the data about the existing items in your game. These are loaded at runtime (and so are very easy for your users to make mods with!). While developing your project in the editor, this JSON file will be imported from the Resources folder, **but in the build version of your game, you need to put a copy of the file items.json file inside Data/Json, relative to the executable of your game**. This path can be modified in the script *ItemList.cs*, if you wish.

Important: as seen in the example provided, all item prefabs, sprites and JSON must be placed in or in a subdirectory of the Resources folder while in the editor!

We have provided a few example items already in the items.json file, all you need to do is add new entries or modify the existing ones. By default, these are the properties you can edit for an item:

- “id_i”: item’s identification number in the list. You can use any positive integer, as long as all id_i are different.
- "id_s": item’s identification name inside the system, not the same as the label the player will see in the inventory. Similar to id_i, but used for finding items by name. You can use any text as long as it doesn't have spaces and all id_s are different.
- "type": item category. Used for description and auto-sorting, but you can also use it for your own purposes inside your game. **For creating new types, read section 2.3.**
- “name”: actual name shown to the player in the tooltip and item details systems.
- "weight": example value you can create and use in your systems. In this implementation, it just shows in the description window.
- “width”: item’s width in squares.
- “height”: item’s height in squares.
- “description”: string of text that appears in the details window when you inspect an item.
- “prefabPath”: file path relative to the Resources folder to the prefab you wish to spawn in the world when you drop the item. Do not use file extensions.
- “spritePath”: file path relative to the Resources folder to the sprite inside Unity that will represent the item in the inventory — you need to use an image file that you set to Sprite in the import settings inside Unity. Do not use file extensions.

Tip: as the Grid-based Inventory Engine uses TextMeshPro for displaying text, you can add rich text tags to displayed text properties, like name and description. A few examples are provided in the JSON file and shown in the example scene.

2.3. Creating custom item types (categories)

When creating a new item type, there are 3 things you need to care about.

1. In the JSON entry for your item, type the name of the category that you want after its type property — CustomType, for example.
2. In Unity, in the top-left corner go to Edit > Project Settings > Tags and Layers, and add a layer with the exact same name you've typed in the JSON, case-sensitive.
3. If you wish to control auto-sort priorities, edit the script *ItemList.cs* inside the *Runtime* folder. Near the end of the code you will see 3 examples of type priority settings.

```
42          // This is where you can add priorities to th
43          types.Add("Weapon", 0);
44          types.Add("Potion", 1);
45          types.Add("Equipment", 2);
```

To add a sorting rule to your custom type, you use `types.Add()` with your custom type name, like so:

```
42          // This is where you can add priorities to
43          types.Add("Weapon", 0);
44          types.Add("Potion", 1);
45          types.Add("Equipment", 2);
46          types.Add("CustomType", 3);
47      }
```

2.4. Creating custom item properties

In the *Runtime* folder you can find the *Item.cs* script, where item properties are defined.

```
4  public class Item
5  {
6      public int id_i;
7      public string id_s;
8      public string type;
9      public string name;
10     public float weight;
11     public int width = 1;
12     public int height = 1;
13     public string description;
14     public string prefabPath;
15     public string spritePath;
16 }
```

There are already provided several examples of properties respective to the ones you have in the JSON file. Let's say you wish to add a value that holds the value of an item, in gold coins. In the Item.cs script, add an integer variable called `value`, like so.

```
4  public class Item
5  {
6      public int id_i;
7      public string id_s;
8      public string type;
9      public string name;
10     public float weight;
11     public int width = 1;
12     public int height = 1;
13     public string description;
14     public string prefabPath;
15     public string spritePath;
16
17     public int value;
18 }
```

And in your item entries in the JSON file, add a *value* field with the desired number.

```
{
    "id_i": 2,
    "id_s": "ring",
    "type": "Equipment",
    "name": "Ring of <color=\"red\">IMPENDING DOOM<color=\"white\">",
    "weight": 0.025,
    "width": 2,
    "height": 2,
    "description": "<color=\"white\">Not very nice.",
    "prefabPath": "Sample Prefabs/ring",
    "spritePath": "Sample Sprites/ring6x6",
    "value": 15
}
```

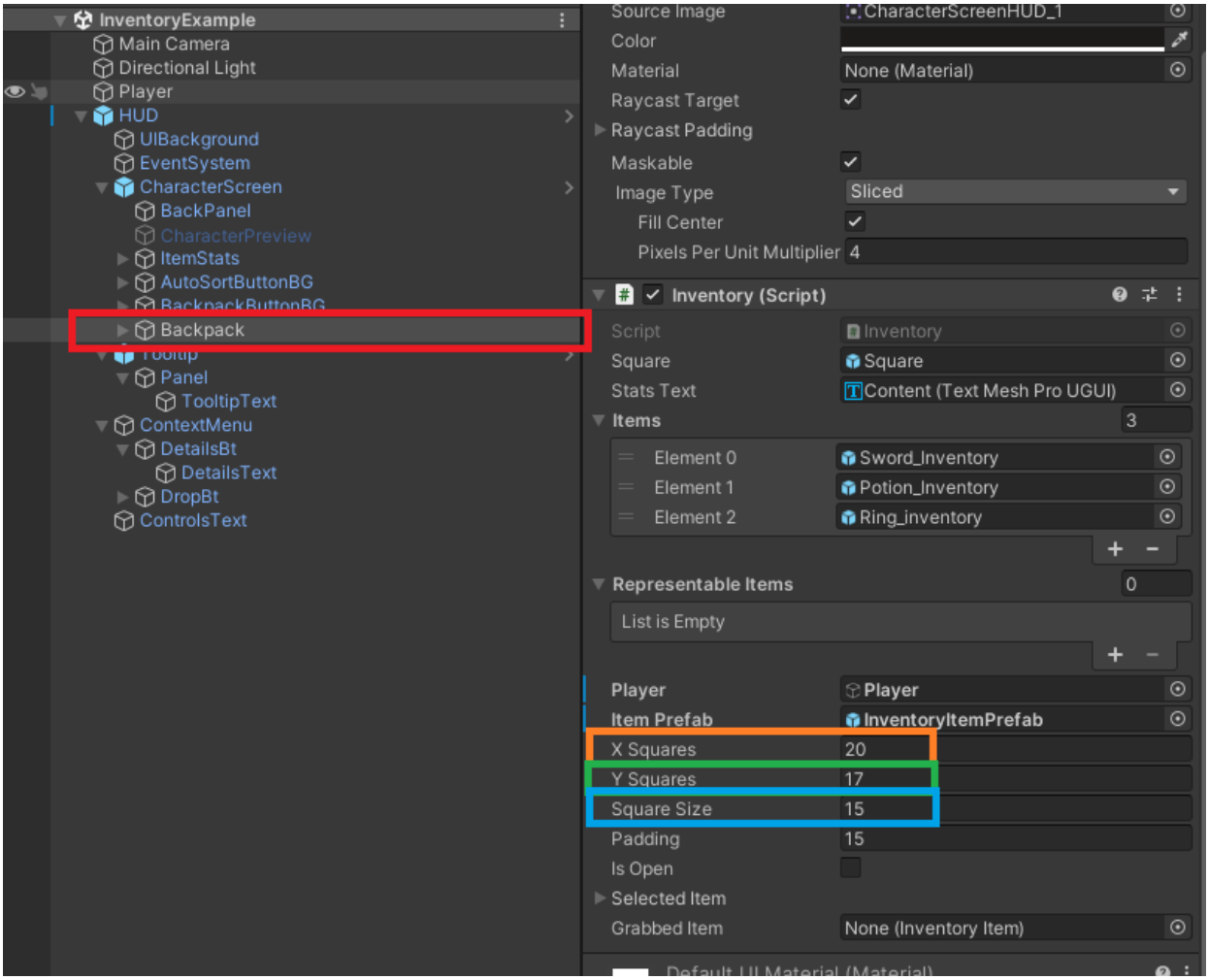
You can then access these properties via your scripts by getting a reference to the InventoryItem and accessing its M_item. For example:

```
7   public InventoryItem inventoryItem;
8
    Unity Message | 0 references
9   private void Start()
10  {
11      print(inventoryItem.M_item.value);
```

If you've set a reference to your item in the inspector, its value will be printed in the Unity console.

2.5. Changing grid and square size

To change the size of the grid in squares or of the squares themselves, in the HUD prefab in the hierarchy navigate to CharacterScreen > Backpack, and in the Inventory script in the inspector change the value of the variables "X Squares" for the number of squares in the horizontal direction, "Y Squares" in the vertical direction, and "Square Size" to decide the actual size of each square. This will automatically adjust the size of the items to fit the squares.


InventoryExample
Main Camera
Directional Light
Player
HUD
UIBackground
EventSystem
CharacterScreen
BackPanel
CharacterPreview
ItemStats
AutoSortButtonBG
BackpackButtonBG
Backpack
Panel
TooltipText
ContextMenu
DetailsBt
DetailsText
DropBt
ControlsText
Source Image
CharacterScreenHUD_1
Color
Material
None (Material)
Raycast Target
Raycast Padding
Maskable
Image Type
Sliced
Fill Center
Pixels Per Unit Multiplier
4
Inventory (Script)
Script
Inventory
Square
Square
Stats Text
Content (Text Mesh Pro UGUI)
Items
3
Element 0
Sword_Inventory
Element 1
Potion_Inventory
Element 2
Ring_inventory
Representable Items
0
List is Empty
Player
Player
Item Prefab
InventoryItemPrefab
X Squares
20
Y Squares
17
Square Size
15
Padding
15
Is Open
Selected Item
Grabbed Item
None (Inventory Item)

3. System functionality

3.1. Picking up and dropping items (adding and removing items from the inventory)

In the *Runtime* folder you can find a script called *PlayerExample.cs*, which contains example usage of the inventory system.

To add items to the inventory, first you need to get a reference to it. In the PlayerExample file we get it by the inventory's tag.

```
public class PlayerExample : MonoBehaviour
{
    Inventory inventory;

    Unity Message | 0 references
    private void Start()
    {
        inventory = GameObject.FindGameObjectWithTag("Inventory").GetComponent<Inventory>();
    }
```

Having a reference to the inventory, you use `inventory.AddItem(x)` to instantiate an item in the inventory, where *x* is the *id_i* (the numerical ID) you've set for the desired item on the JSON file.

```
private void Update()
{
    if (Input.GetKeyDown(KeyCode.Alpha1))
    {
        inventory.AddItem(0);
    }

    if (Input.GetKeyDown(KeyCode.Alpha2))
    {
        inventory.AddItem(1);
    }

    if (Input.GetKeyDown(KeyCode.Alpha3))
    {
        inventory.AddItem(2);
    }
}
```

This function can be called from anywhere, as long as you have a reference to the inventory.

3.2. Auto-sort

As explained in the previous section, the auto-sort algorithm sorts items with a lower priority number first. They are placed in a left-to-right, top-to-bottom order. In cases in which the algorithm doesn't manage to find placement for all items, nothing will happen. In the sample inventory there is a button which calls the `AutoSort()` function. Like the `AddItem()` function, it can be called from anywhere as long as you have a reference to the inventory.

3.3. Extending this package

All of this project's source code is provided with this package, so you can modify it as you wish. We would love to receive your message if you have created something with our software. Happy developing!