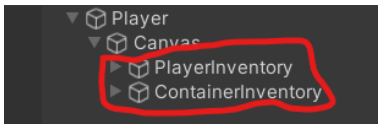
Mirror Inventory System By Trayson Cazier (Marshmallowboy)

Getting Started:

Create a new script, Call It Inventory Manager. This is how you will manage your inventory (duh). Make it inherit from MirrorInventory. I have put an example inventory manager in with the asset so I would recommend checking that out.
Anyway when you make that script put it on your Player Object. Then create a canvas (If you haven't already) then you should be able to just go to the example player object and copy over Player Inventory UI Object, and also Container Inventory UI Object.



Then copy it over to your canvas script.
From here you want to go to your inventory manager and then create item slots in your array, and assign each child object of the player inventory to it, like in the example.
The items index is where you store the database for the bobbles (bobbles are the physical item that stores data, like the gun and the magazine in the example) always leave ItemsIndex[0] as null, then from there put bobbles to their corresponding ID position, if a bobbles ID is 1, then put it in the 1 slot on the index. Speaking of bobbles,

How Bobbles Work:

On every bobbles there is an ISI script, or rather InventorySystemIdentification, this stores information about your bobble, this defines your id EVERY BOBBLE MUST HAVE A UNIQUE ID, AND IT CANNOT BE ZERO BECAUSE THAT MEANS IT IS NULL! So start with your bobble being one, the value is just a preset value, because your bobbles can store more data than just their id, for example, if you want an ammo counter on your gun, you just get a reference to your gun, and then get the ammo using

```
ammo = ActiveBobble.GetComponent<InventorySystemIdentification>().value;
```

That is just an example, you can also store a string, However adding values is its own beast.

Adding your own custom values:

To add your own values you must edit not only ISI (InventorySystemIdentification) but also the core MirrorInventory script.
Step 1:

```
public class InventorySystemIdentification : MonoBehaviour
{
    //This is the information you want to be on your item.
    public int id = 0;
    public int value;
    public string YourValue;
}
```

Add your value, it can be whatever value that can be passed through a parameter, int, string, bool, etc.

Step 2:
Go to ContainerInventory script, and above the class you will see:

```
public struct item
{
    public int id;
    public int value;
}
```

Change it to:

```
public struct item
{
    public int id;
    public int value;
    public string YourValue;
}
```

Step 3:
Go to this line in MirrorInventory:

```
public static item NullItem = new item { id = 0, value = 0 };
```

And change it to, for example:

```
public static item NullItem = new item { id = 0, value = 0, YourValue =
string.Empty };
```

Step 4:
In Function “StartInitialize”
Find this line:

```
InitializeItemCommand(container, new item { id = ISI.id, value = ISI.value
});
```

And change it to:

```
InitializeItemCommand(container, new item { id = ISI.id, value = ISI.value,
YourValue = ISI.YourValue });
```

Step 5:
In Function “LoadInventoryFromContainer”
Find these lines:

```
Items[i].id = ID.id;
Items[i].value = ID.value;
```

And then add your variable:

```
Items[i].id = ID.id;
Items[i].value = ID.value;
Items[i].YourValue = ID.YourValue;
```

Step 6:

In Function “LoadInventory”
Got to these lines:

```
GameObject temp = Instantiate(ItemsIndex[Item.id]);
temp.GetComponent<InventorySystemIdentification>().value = Item.value;
temp.transform.SetParent(ContainerSlots[i].transform, false);
```

Then add your value after the second line:

```
GameObject temp = Instantiate(ItemsIndex[Item.id]);
temp.GetComponent<InventorySystemIdentification>().value = Item.value;
temp.GetComponent<InventorySystemIdentification>().YourValue =
Item.YourValue;
temp.transform.SetParent(ContainerSlots[i].transform, false);
```

Step 7:
At this point you get the idea, Go to this in “UploadInventory”:

```
Items[i].id =
ContainerSlots[i].transform.GetChild(0).GetComponent<InventorySystemIdentif
ication>().id;
Items[i].value =
ContainerSlots[i].transform.GetChild(0).GetComponent<InventorySystemIdentif
ication>().value;
```

Change it to this:

```
Items[i].id = ContainerSlots[i].transform.GetChild(0).GetComponent<InventorySystemIdentification>().id;
Items[i].value =
ContainerSlots[i].transform.GetChild(0).GetComponent<InventorySystemIdentification>().value;
Items[i].YourValue =
ContainerSlots[i].transform.GetChild(0).GetComponent<InventorySystemIdentification>().YourValue;
```

Step 8:
In Save Inventory To String;
Change this:

```
Item = new item
{
      id =
itemSlot.GetComponentInChildren<InventorySystemIdentification>().id,
            value =
itemSlot.GetComponentInChildren<InventorySystemIdentification>().value
 };
```

To This:

```
 Item = new item
  {
      id = itemSlot.GetComponentInChildren<InventorySystemIdentification>().id,
            value =
itemSlot.GetComponentInChildren<InventorySystemIdentification>().value,
```

```
            YourValue =
itemSlot.GetComponentInChildren<InventorySystemIdentification>().YourValue
  };
```

Step 9:

Then in the same function, go to this line:

```
string itemToString = Item.id.ToString() + "," + Item.value + ":";
```

Change it to:

```
string itemToString = Item.id.ToString() + "," + Item.value + "," +
Item.YourValue + ":";
```

Step 10:

Go to “LoadInventoryFromString”

```
Item.id = int.Parse(itemListString[i].Split(',')[0]);
Item.value = int.Parse(itemListString[i].Split(",")[1]);
```

Change it to:

```
Item.id = int.Parse(itemListString[i].Split(',')[0]);
Item.value = int.Parse(itemListString[i].Split(",")[1]);
Item.YourValue = itemListString[i].Split(",")[2];
```

We don’t need the int.parse for this variable because it's a string.

Step 11(Final):

Then we go to in the same function:

```
GameObject temp = Instantiate(ItemsIndex[Item.id]);
temp.GetComponent<InventorySystemIdentification>().value = Item.value;
temp.transform.SetParent(itemsSlots.transform, false);
```

And just like when we changed LoadInventory we change it to:

```
GameObject temp = Instantiate(ItemsIndex[Item.id]);
temp.GetComponent<InventorySystemIdentification>().value = Item.value;
temp.GetComponent<InventorySystemIdentification>().YourValue =
Item.YourValue;
temp.transform.SetParent(itemsSlots.transform, false);
```

And then your done. See? Easy! 🙃

Functions:

Start Initialize:

```
public void StartInitialize(ContainerInventory containerInv);
```

Start initialize is a function in which should almost always be put into the start function of your Inventory Manager script, that is however, only if you only have one container in your scene, if you were to dynamically change the container, for example, if you had a raycast in an FPS environment, and the raycast set a container that it lands on as your container. In other words,

call this right after you have a reference to a container, nothing bad happens if you call this multiple times to the same container however. It just won't do anything if its already been initialized.

Parameters: ContainerInventory; Script; The current containers `ContainerInventory` script.

Load Inventory From Container:

```
void LoadInventoryFromContainer(SyncList<item>.Operation op, int index, item oldItem, item newItem)
```

You should pretty much NEVER have to call this function, StartInitialize does everything for you and this is just a callback. It does it for you. But if you must know, this function checks everything in the containers OBJECT inventory (the array on the container inventory), then loads it into the list inventory.

Load Inventory:

```
public void LoadInventory()
```

This updates your inventory to the synclist on the containerObjects, this should be called automatically so you don't actually have to worry about it in general.

Upload Inventory:

```
public void UploadInventory()
```

This is something that you have to worry about, this sends the data in the visual inventory. In the example InventoryManager, this is called every time the player lifts up their Fire1 key, so it updates every time they let go of m1. This is because the container inventory UI never changes until you have let go of the m1 button. I think you should always call this when m1 is up, but you can change this to your needs.
Example:

```
if (Input.GetButtonUp("Fire1"))
{
    UploadInventory();
}
```

Initialize Item Command:

```
[Command]
public void InitializeItemCommand(ContainerInventory containerInv, item id)
```

This should never be called from your Inventory Manager script. Except if you absolutely must add an id to a container, make sure you clear inventory first though, otherwise you will get more

Items in your items synclist than it is supposed to have.

Clear Inv List Command:

```
[Command]
public void ClearInvListCommand(ContainerInventory containerInv)
```

This should probably not be called from your Inventory Manager script. However if you need to completely delete all the data in a container, this is your best bet.

InventoryFull:

```
public bool InventoryFull(bool ContainerInventory)
```

This is what you use to check if an inventory is full.
Parameters: ContainerInventory; boolean; this determines whether the inventory it checks is the Container Inventory's UI is what is checked, if false, it checks the players inventory, if true, it checks the Container Inventory.

AddItemByID:

```
public void AddItemByID(int ID)
```

This adds an item to the player inventory, you cannot directly add items to container inventory without clearing it then using InitializeItemCommand, I might make a function for doing that however.

DestroyItemByID:

```
public void DestroyItemByID(int ID)
```

This does the same thing as AddItemByID but it destroys an item instead. Also these pick the nearest spot to instantiate/destroy top left to bottom right.

ReturnNearestSlotNumber:

```
public int ReturnNearestSlotNumber(bool ContainerInventory)
```

This returns the nearest slot number based on from top left to bottom right. I will explain this system in the next sections.

Parameters:Container Inventory; boolean; determines whether it gets the number in the Container Inventory or Player Inventory.

Save/LoadInventoryFromString:

```
public string SaveInventoryToString()
```

Saves an inventory to a string in a format that can be read by LoadInventoryToString(), the

purpose of this is to save the players inventory that can be stored on, for example, gamejolts servers.

```
public void LoadInventoryFromString(string InventoryAsString)
```

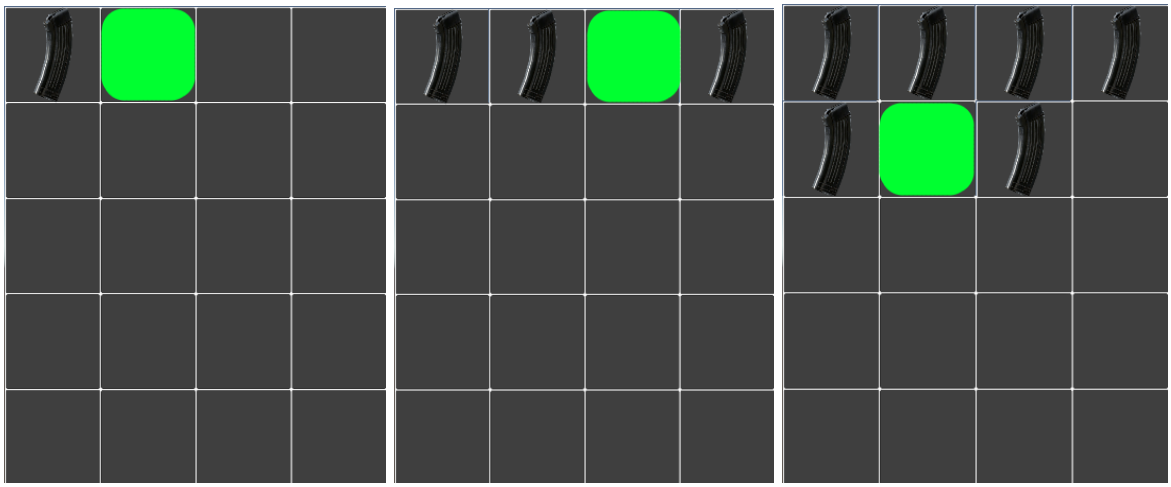
Loads the player inventory based on the string param put into it, you can generate this string from SaveInventoryToString();

Parameters: InventoryAsString; string; This is used to load from a specifically formatted string.

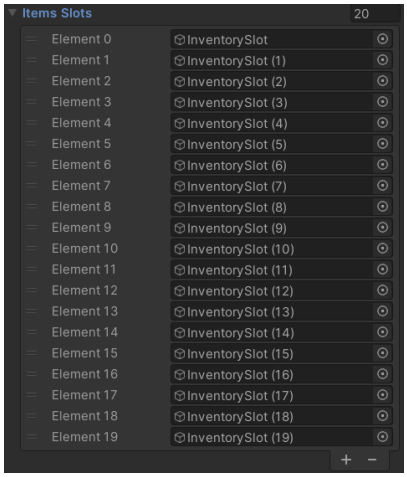
Systems:

TopLeft To Bottom Right Inventory System:

This system works by, for example in ReturnNearestSlotNumber, it return the number of the slot closest to the top right that isn't full, instead of explaining it through words I will add a graphic, the green is the spot it returns.



Inventory Arrays System:



The array on the left directly corresponds to the game objects on the right, you want to do the same with the container inventory.