Table of Contents

Features

- `Resize` at runtime, dropping what no longer fits.
- `Add/Remove` and check if an item fits from code.
- `Equipment slots` for all your RPG needs.
- `Custom shapes` for each item.
- Rearrange items by `draggin and dropping`, with visual feedback.
- `Move items between inventories`.
- Remove items by `dropping` them outside the inventory.
- Easily add `custom graphics` and change the size of your inventory.
- Supports `scaled canvases`.
- Tested thoroughly with over `75 Unit Tests`, and profiled using the Unity Profiler.
- Tested using all types of `Canvas render modes` (Screen Space Overlay, Screen Space Camera and World Space)

Example

A fully functional example is included with this reposetory and can be found in the folder "`Assets/Example`".

- `Inventory Overlay.scene` - the Unity Scene that contains an example using the Canvas render mode of Screen Space Overlay.
- `Inventory Camera.scene` - the Unity Scene that contains an example using the Canvas render mode of Screen Space Camera.
- `Inventory World Space.scene` - the Unity Scene that contains an example using the Canvas render mode of World Space.
- `Inventory.png` - includes all artwork used in the example.
- `ItemDefinition.cs` - a `ScriptableObject` implemetation of `IInventoryItem`.
- `SizeInventoryExample.cs` - a `MonoBehaviour` that creates and connects an Inventory with a Renderer, and fills it with items.
- `Items-folder` - Contains the `ItemDefinitions` used in the example.

Documentation

Below you can find documentation of various parts of the system. You are encouraged to look through the code, where more in-depth code docs can be found.

Getting Started

Creating a new inventory is simple. Remember that the inventory system rests within its own namespace, so don't forget to add `using FarrokhGames.Inventory`.

```
var inventory = new InventoryManager(8, 4); // Creates an inventory with a width of 8 and height of 4
```

The Inventory

Below is a list of actions methods and getters within `InventoryManager.cs`.

```
/// <summary>
/// Invoked when an item is added to the inventory
/// </summary>
Action<IInventoryItem> onItemAdded { get; set; }

/// <summary>
/// Invoked when an item was not able to be added to the inventory
/// </summary>
Action<IInventoryItem> onItemAddedFailed { get; set; }

/// <summary>
```

```
/// Invoked when an item is removed to the inventory
/// </summary>
Action<IInventoryItem> onItemRemoved { get; set; }

/// <summary>
/// Invoked when an item is removed from the inventory and should be placed
on the ground.
/// </summary>
Action<IInventoryItem> onItemDropped { get; set; }

/// <summary>
/// Invoked when an item was unable to be placed on the ground (most likely
to its canDrop being set to false)
/// </summary>
Action<IInventoryItem> onItemDroppedFailed { get; set; }

/// <summary>
/// Invoked when the inventory is rebuilt from scratch
/// </summary>
Action onRebuilt { get; set; }

/// <summary>
/// Invoked when the inventory changes its size
/// </summary>
Action onResized { get; set; }

/// <summary>
/// The width of the inventory
/// </summary>
int width { get; }

/// <summary>
/// The height of the inventory
/// </summary>
int height { get; }

/// <summary>
/// Sets a new width and height of the inventory
/// </summary>
void Resize(int width, int height);

/// <summary>
/// Returns all items inside this inventory
/// </summary>
IInventoryItem[] allItems { get; }

/// <summary>
/// Returns true if given item is present in this inventory
/// </summary>
bool Contains(IInventoryItem item);

/// <summary>
/// Returns true if this inventory is full
```

```
/// </summary>
bool isFull { get; }

/// <summary>
/// Returns true if its possible to add given item
/// </summary>
bool CanAdd(IInventoryItem item);

/// <summary>
/// Add given item to the inventory. Returns true
/// if successful
/// </summary>
bool TryAdd(IInventoryItem item);

/// <summary>
/// Returns true if its possible to add item at location
/// </summary>
bool CanAddAt(IInventoryItem item, Vector2Int point);

/// <summary>
/// Tries to add item att location and returns true if successful
/// </summary>
bool TryAddAt(IInventoryItem item, Vector2Int point);

/// <summary>
/// Returns true if its possible to remove this item
/// </summary>
bool CanRemove(IInventoryItem item);

/// <summary>
/// Returns true ifits possible to swap this item
/// </summary>
bool CanSwap(IInventoryItem item);

/// <summary>
/// Removes given item from this inventory. Returns
/// true if successful.
/// </summary>
bool TryRemove(IInventoryItem item);

/// <summary>
/// Returns true if its possible to drop this item
/// </summary>
bool CanDrop(IInventoryItem item);

/// <summary>
/// Removes an item from this inventory. Returns true
/// if successful.
/// </summary>
bool TryDrop(IInventoryItem item);

/// <summary>
/// Drops all items from this inventory
```

```
/// </summary>
void DropAll();

/// <summary>
/// Clears (destroys) all items in this inventory
/// </summary>
void Clear();

/// <summary>
/// Rebuilds the inventory
/// </summary>
void Rebuild();

/// <summary>
/// Get an item at given point within this inventory
/// </summary>
IInventoryItem GetAtPoint(Vector2Int point);

/// <summary>
/// Returns all items under given rectangle
/// </summary>
IInventoryItem[] GetAtPoint(Vector2Int point, Vector2Int size);
```

Items

Items inside the inventory are represented by the IInventoryItem interface. In the included example, this interface is attached to a `ScritableObject` making it possible to create, store and change item details in the asset folder.

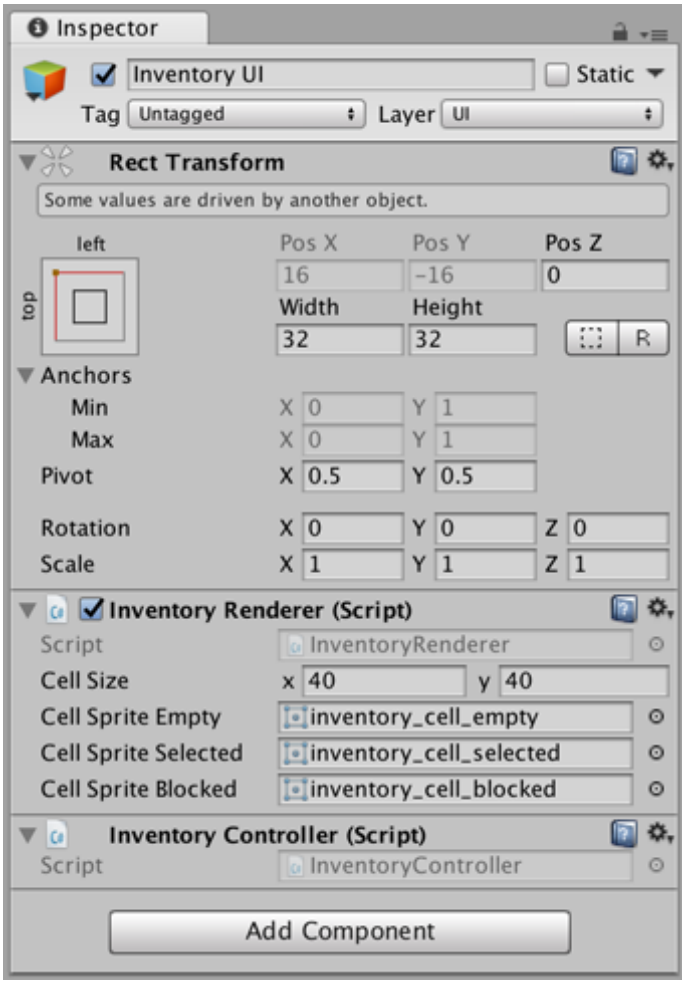
```
using UnityEngine;
using FarrokhGames.Inventory;

/// <summary>
/// ScriptableObject representing an Inventory Item
/// </summary>
[CreateAssetMenu(fileName = "Item", menuName = "Inventory/Item", order = 1)]
public class ItemDefinition : ScriptableObject, IInventoryItem
{
    [SerializeField] private Sprite _sprite;
    [SerializeField] private InventoryShape _shape;
    [SerializeField] private bool _canDrop;

    public string Name => return this.name;
    public Sprite Sprite =>  return _sprite;
    public InventoryShape Shape => _shape;
    public bool canDrop => _canDrop;

    /// <summary>
    /// Creates a copy if this scriptable object
    /// </summary>
```

```
    public IInventoryItem CreateInstance()
    {
        return ScriptableObject.Instantiate(this);
    }
}
```

The shape of an item is defined by the serializable `ItemShape.cs` class which has a useful property drawer.

Rendering the Inventory

The inventory system comes with a renderer in a MonoBehaviour called `InventoryRenderer.cs`.



Simply add this to a `GameObject` within your `Canvas`, and connect it to an inventory using the following code.

```
/// <summary>
/// Set what inventory to use when rendering
/// </summary>
/// <param name="inventory">Inventory to use</param>
public void SetInventory(InventoryManager inventory);
```

Please see the image at the top of this document as an example of how the rendering looks

---

Interacting with the Inventory

To enable interactions (drag and drop), add `InventoryController.cs` to the same `GameObject` as your renderer.

Other files included

Besides the actual inventory, there are sipport-classes included in the reposetory.

- `Pool.cs` - A generic pool of objects that can be retrieved and recycled without invoking additional allocations. Used by the `Renderer` to pool sprites.

You are free to use these support-classes under the same license, and their `Unit Tests` are included.

A huge thanks to farrohkgames for creating and sharing such an asset! You can find the original files here: [GitHub - FarrokhGames/Inventory: A Diablo 2-style inventory system for Unity3D]