Tutorial – Drag and Drop Shop

This tutorial covers a potential project brief for your assessment.

Make sure you've completed the following series as it makes use of the concepts introduced there.

- *Tutorial – Unity User Interface*
- *Tutorial – Unity UI Layouts*
- *Tutorial – Unity Event Systems*

Designing a Drag and Drop UI

Consider PC games you may have played, where you have an inventory of items represented by little icons in a grid, or a list. You can approach a shopkeeper and see their store's inventory and your own inventory, and drag items between the two with the mouse to buy and sell them.

Torchlight, for example, uses a GridLayout-like arrangement, with each item represented by a small icon with tooltips to give you more detailed information.

World of Warcraft uses something more like a VerticalLayout, with each item represented by a longer thinner row, which displays some details on each item at all times.

These items can be drag-and-dropped into your equipment slots.

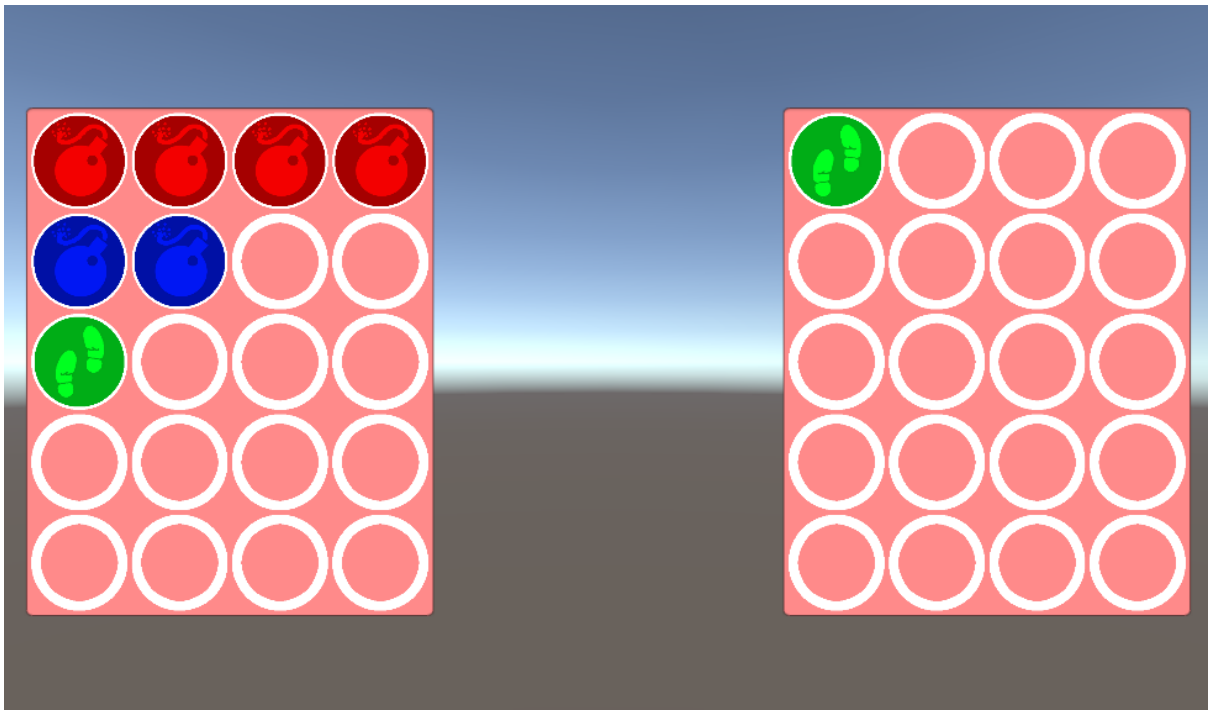
Despite looking quite different, these two systems are the same under the hood.

- A backend array of object is represented by cloning a prefab to represent each object.
- These prefabs are placed in a parent container which is responsible for laying them out spatially.
- The prefabs, or parts of them, can be picked up by the mouse and dropped into the other container
- Doing this updates the backend arrays and other data (eg your equipment and how much money you have) and the front end, so that the two stay consistent.

It's a very common UI paradigm, very tactile and satisfying to use. We'll discuss here how to go about implementing it.

Designing a Drag and Drop UI

You can create your own simple drag and drop UI in Unity, for example with a store screen on the left and the player's inventory on the right. Items can be dragged between the two with the mouse.



You'll need the following classes, or similar:

ShopItem – a ScriptableObject back-end class that specifies the icon, colour and so on for each item

Inventory – a back-end class with an array or List of ShopItems (that typically doesn't change size during gameplay, but can support null entries for empty slots). This will be a MonoBehaviour that you can stick on GameObjects like the Player or Shop.

ShopItemUI – a front end MonoBehaviour that displays a single ShopItem, using the pattern described in the earlier tutorial series. Different prefabs can be set up for the same script.

Slot - a front-end class for the empty slots which sit in the inventory, with or without items inside them.

InventoryUI – a frontend class for managing and cloning Slot and ShopItemUI prefabs in response to showing the entire inventory that we've created.

The InventoryUI will work slightly differently from the ActionListUI in the previous tutorial. For each ShopItem in the Inventory it should clone a Slot prefab as its own child, and then clone a ShopItemUI prefab as a child of that cloned Slot.

The Init() function in the example code below is used to link up references between the Slot and the ShopItemUI and the InventoryUI

```
    public Inventory inventory;

    public ShopItemUI itemPrefab;
    public Slot slotPrefab;

    Slot[] slots;

    // Start is called before the first frame update
    void Start()
    {
        // create a slot for each item
        slots = new Slot[inventory.items.Length];
        for (int i=0; i < inventory.items.Length; i++)
        {
            slots[i] = Instantiate(slotPrefab, transform);
            // create a shop item UI and feed the item to it
            slots[i].itemUI = Instantiate(itemPrefab, slots[i].transform);
            slots[i].itemUI.SetItem(inventory.items[i]);
            // create a slot
            slots[i].Init(this, i, slots[i].itemUI);
        }
    }
```

You should write up your own version of this technical design, and create a UML diagram of the classes and their inter-relationships.

The details of the Drag and Drop process

When we drag an item, we want to have it follow the mouse until we let go, at which point we decide what to do with it. If there's a slot underneath the mouse, we drop it there (maybe swapping with an existing item). If there isn't we snap the dragged object back to its original slot.

It's tempting to think that we re-parent the dragged item to its new slot when we move objects around. In practice it's easier to use a little smoke and mirrors. The dragged item always snaps back to its original parent slot, but we change the items being displayed in both the original and the new target slot. This gives a sense of visual continuity without disrupting the scene. Each Slot and its child ShopItemUI remain in the same place in the hierarchy, with the ShopItems being represented by them changing as we drag objects around.

We also want the backend objects to change when we perform a successful drag and drop. If we have references to the inventories, and know which index each slot represents, then we can carry this out easily enough.

This means each Slot needs a reference to their ShopItemUI, and have a reference to their InventoryUI, and know their index in the array. We will also require that the dragged item has a back-reference to its parent slot.

Unity's drag and drop interfaces

You can make the ShopItemUI moveable with the mouse by implementing the following interfaces in the UnityEngine.EventSystems namespace: IBeginDragHandler, IDragHandler and IEndDraghandler.

```
using UnityEngine.EventSystems;

public class ShopItemUI : MonoBehaviour, IBeginDragHandler, IDragHandler,
IEndDragHandler
{
```

Each of these requires that you implement a single function, which Visual Studio will guide you through.

For the most basic behaviour, you'll need to do the following:

void OnBeginDrag(PointerEventData eventData)

Set a dragging Boolean member variable to true.

We also want to make sure that the dragged object appears above everything else. An easy way to do this is to make it a child of the parent Canvas (find this using GetComponentInParent<>) and call Transform.SetAsLastSibling() on it. Store the original parent in a member variable for when we finish the drag.

```
public void OnBeginDrag(PointerEventData eventData)
    {
        if (originalParent == null) originalParent = transform.parent;
        if (canvas == null) canvas = GetComponentInParent<Canvas>();

        transform.SetParent(canvas.transform, true);
        transform.SetAsLastSibling();

        dragging = true;
    }
```

void OnDrag(PointerEventData eventData)

Move the transform of the object to the mouse position, like so. This isn't called every update, but is called every frame that the user moves the mouse.

```
    public void OnDrag(PointerEventData eventData)
    {
        if (dragging)
            transform.position = eventData.position;
    }
```

void OnEndDrag(PointerEventData eventData)

We do the bulk of the work here.

- See if there's a Slot under the mouse using EventSystem.RaycastAll

```
List<RaycastResult> hits = new List<RaycastResult>();

public void OnEndDrag(PointerEventData eventData)
{
    // is there a slot underneath us?
    Slot slotFound = null;
    EventSystem.current.RaycastAll(eventData, hits);
    foreach (RaycastResult hit in hits)
    {
        Slot s = hit.gameObject.GetComponent<Slot>();
        if (s)
            slotFound = s;
    }
```

- If there is, take that slot's index and get the backend Inventory from the Slot's container. Swap the items around in the backend data. Each Slot's ShopItemUI child will need to update to reflect the new item that's being held in there (which may be null)

- Snap the dragged ShopItemUI back to its parent position. Do this by restoring it's parent to the original parent we stored, and then setting transform.localPosition to zero.

Once you've got this working for the ShopItemUI you might want to create a base class for it called Draggable, and move the drag and drop implementation into that class so you can re-use this behaviour on other components in order to drag and drop them.

Extending this functionality

Adding in money

If you follow this outline, you should be able to drag items between the two containers and see the backend arrays update in real time in the inspector.

You could then look at introducing money.

When you are about to swap the two items in the OnEndDrag handler function, you should check to see if the player has enough money to perform the transaction. Note that the drag and drop can worth both ways. If the player drags a 50 gold potion on to a 500 gold crown in the store, then the overall transaction would cost them 450 gold, so you need to check that they have that amount of money.

If the swap is successful you also need to subtract that money from the player (and show it in the game HUD somewhere).

Generalising the containers

The InventoryUI class here holds a reference to an Inventory, but all it uses the Inventory for is an array of ShopItem objects.

InventoryUI could be derived from a more general ObjectArrayContainer which has a virtual function for getting a reference to an array of objects. (Note that in C# an array of a derived class can be assigned to an array of the base class, e.g.

```
        ShopItem[] items = new ShopItem[20];
        object[] things = items;
```

This could form the basis for a drag and drop library that can manipulate any array of objects, and not just ShopItems.

If you want to make a base class that manipulates Lists instead of arrays, you can do that too. You'll need to create a generic class that can take a type T to pass on to the List.