The DragAndDrop Library

The purpose of this Library is to streamline the development of drag-and-drop interfaces such as game inventories, training screens and so on.

The scripts provided work with arrays or Lists of objects, and allow for adding, removing and re-ordering elements to those collections with minimal custom code. At the same time, function overrides allow for game specific logic and behaviour to be added to your drag-and-drop interfaces.

Video Links

The following videos demonstrate the contents of the DragAndDrop library in various levels of detail.

A quick introduction to the package (3 minutes)

https://www.youtube.com/watch?v=I_tdKhatPrM

An in-depth look at the Example scene (17 minutes)

https://www.youtube.com/watch?v=Yobqp0s3gIc

A live development example on how to create a working shop (32 minutes)

https://www.youtube.com/watch?v=CHA5LCSiuaw

Example Scene

On the left is a 4 x 5 Inventory backpack, with an equip screen to the right with 7 slots arranged in a vaguely humanoid formation, and a 4 slot potions belt below. You can use the left mouse to drag items between these three containers.
The potions belt can only receive potions. Anything else will snap back. The potion belt squares light up gold if a potion is dragged over them, and shrink and turn grey if anything else is, to let the user know that you can't place those items there.
If you drag something over the Equip screen, all squares will open or contract, to help guide the user to the right slot for placing the item.

On the right is a powerset list with four powers listed in it.
Below that is a powers bar. You can use the right mouse to drag powers from the powerset into the powers bar. Note that this doesn't remove powers from the powers set, and the same item can be dropped in multiple slots here.

While playing with this interface, select the Player object in the inspector. You'll see that moving items around in the UI causes its internal arrays and lists to update correctly.

DragAndDrop classes

The five core classes are in the \Assets\DragAndDrop\Scripts folder.

Draggable derived UI prefabs are dragged between **Slot** prefabs, with the game-specific rules and relationships to back-end data being managed by **ObjectContainer** derived classes.

Each **Slot** has a **Draggable** child that does not change parent. **Draggable** objects that represent an empty slot are typically turned off until something occupies that **Slot**. The **Draggable** moves with the mouse, but always snaps back to its rest position when released, with a little sleight-of-hand making it look like its moved from one position to another.

ObjectContainerList and **ObjectContainerArray** are **ObjectContainer** derived classes that manage underlying C# arrays or Lists of objects. Derive your own container classes from these and you'll have at minimum only one line of code to write to get everything working.

The ObjectContainer-derived classes typically keep track of a collection of Slots, and a collection of backend objects like an array or List.

Each Slot has a reference to its child Draggable, and may know its index within the collection.

Each Draggable has a reference to the object it represents.

Draggable

An abstract base class for any UI object that represents a single item, such as a Power, Inventory Item or anything else that can be dragged between containers. A **Draggable** represents a **UnityEngine.Object** derived object of some kind. This class does most of the heavy lifting.
You'll typically derive from this class and make one or more UI prefabs for it.
Your base class has to override the **UpdateObject()** function, which copies the data from the object that it represents into the UI elements of the prefab. Examples are **CharmUI** and **PowerUI**. Note that these classes are both used in multiple prefabs to give different representations of the same data. In each prefab you specify mouseButton as 0 or 1 to enable the object to be dragged with either the left or right mouse. (This defaults to 0, but you could set it to 1 if your item is also a **Button**, like the **PowerUI** prefabs do).

Public Member Variables

int mouseButton	0 = left mouse button to drag 1= right mouse button to drag Set in the inspector on a per-prefab basis.
static Draggable current	A reference to the Draggable (if any) currently being dragged by the user
Slot slot	The Slot in which this object lives. This will remain constant for the Draggable's lifetime, as each draggable remains a child of the Slot it was spawned in. Dragging operations just change the obj value between Draggables.
UnityEngine.Object obj	The object (eg inventory item, power) that this Draggable is representing

Private and Protected Member Variables

bool dragging	True if the draggable is currently being dragged by the user
Slot currentSlotOver	A reference to the Slot which the currently dragged object is over. Used to manage and trigger the ObjectContainer events used to highlight slots.
Canvas canvas	A cached reference to the parent Canvas we're inside. This is used to boost the canvas' sortingOrder so that the dragged object never goes behind other canvases but always remains on top.

Public member functions to override

void UpdateObject()	You must override this. In here you talk to any child UI components and change their state to reflect the object that you're displaying, stored in the **obj** member. This will probably involve casting the object first

Public member functions

void SetObject()	Sets the object we're representing, and calls UpdateObject()
void OnBeginDrag()	Drag handler for when a drag starts. It sets our dragged boolean so we'll follow subsequent mouse moves. Its makes us the top child sibling and boosts our canvas in the sortingLayer order so that we appear as a topmost UI item whilst dragged. Checks the Container's CanDrag function to veto this move.
void OnDrag	Moves us with the mouse. Also raycasts for slots under the mouse and updates containers and slots that we pass over to allow them to highlight or respond in other ways.
void OnEndDrag()	Checks the potential drop both ways – can we be placed in the slot we've been dragged over, and if we're displacing another item can that be placed in our parent container? This will update the backend data if the move is legal, and calls the appropriate events for successful or failed drags. Canvas sortingLayer is restored here.

Private member functions

private Slot GetSlotUnderMouse()	Raycasts under the mouse to find the top Slot object that we're over. Used internally when dragging an object to call events on the slots we pass over, and to determine where we're dropping a dragged object.
void SwapWith(Slot _slot, bool readOnlySource, bool readOnlyTarget)	Swaps the object being dragged with the object in a target slot's object. Here we check the legality of the move both ways – can we drop into the target container, and can the object being displaced (if any) get put into our parent container? ReadOnly flags tell us if either container is readonly

Slot

A Slot represents a single place in the array or list of items being manipulated. Each Slot has a child Draggable, which may be inactive if the slot is empty.
You can use the base Slot class as it is, or derive your own class from it to display extra data in the slot, as I've done with PowerSetSlot. If you derive a class you can override UpdateSlot() and have the slot's state change when the object it's holding changes.

Public Member Variables

Transform slot	If the Draggables are intended to be children of one of our child components, set that child Transform here. If set to null (as default) then Draggables are direct children of the Slot prefab.
ObjectContainer container	A back reference to the container this slot is a part of. This is used to query the drag and drop rules set up for the container.
int index	An index for this slot within its container. Used when managing arrays and Lists, to determine which element this slot corresponds to.
UnityEvent onDragEnterCanSlot	This event gets invoked when a legal object is dragged over this slot.
UnityEvent onDragEnterCannotSlot	This event gets invoked when an object is dragged over this slot that cannot be slotted here according to the container's rules.
UnityEvent onDragExit	This event is invoked when the currently dragged object leaves this slot. If you change the slot's state using the above events, restore it when this one is invoked.
UnityEvent onSlot	This event gets invoked when an object is slotted in this slot.
bool Unlocked	A public property for setting a slot to locked/unlocked status.
Draggable item	A public property for the Draggable that sits in this slot. Setting it causes UpdateSlot() to be called automatically, in case our slot needs to adapt its own components.

Public member functions to override

void UpdateSlot()	If the Slot has child components that need to be updated when the object it holds is changed, do so in here. See **PowerSetSlot** for an example of this

Public member functions

void OnDraggableEnter()	Calls the appropriate event when a Draggable is dragged over this slot. Called from Draggable code.
void OnDraggableExit()	Called from Draggable when a Draggable is dragged off this slot.

ObjectContainer

An abstract class that represents a collection of Slots. ObjectContainers allow you to specify rules as to how the collection behaves. What can be slotted in there, is the array read-only, and so on.
You specify a UI Prefab for the Slots (Slot Prefab) and one for for the Draggables that will sit inside them (Item Prefab). These are used in the MakeSlot function.
You can override any of the following functions:

Public Member Variables

Slot slotPrefab	The prefab to instantiate slots for holding items in this container
Draggable itemPrefab	The prefab used to instantiate items in this container
Slot[] preMadeSlots	An optional array of slots that have already been placed in the editor.
UnityEvent onDragBegin	This event gets invoked when an item is first dragged from a slot in this container.
UnityEvent onDragEnd	This event gets invoked when an object is successfully slotted in this container.
UnityEvent onDragFail	This event is called when the user attempts to slot something in this container and it is rejected.
UnityEvent onDragEnter	This event gets invoked when an object is dragged over any of this container's slots.
UnityEvent onDragExit	This event is called when a dragged object exits this container's slots airspace.

Public member functions to override

bool CanDrag(Draggable dragged)	Return false to prevent the item being dragged from this container.
bool CanDrop(Draggable dragged, Slot slot)	Returns whether a dragged item can be dropped in the described slot.
void Drop(Slot slot, ObjectContainer fromContainer)	This is called after a successful drop into this container. This is where you should update the back-end data.
void ThrowAway(Draggable	This event is called when a draggable is dragged from this

dragged)	container and dropped in empty space. This, for example, is where you might spawn an in-world item if the player removes something from their inventory.
void OnDragBegin()	By default calls the onDragBegin event. You can override this if you like to have hardcoded behaviour when you start to drag an object from this container.
void OnDraggableEnter()	By default, calls the onDraggableEnter event. Override this to hardcode behaviour when an object is dragged over this container's slots
void OnDraggableExit()	By default, calls the onDraggableExit event. Override this to hardcode behaviour when an object is dragged off this container's slots' airspace.
bool IsReadOnly()	Return true to make prevent the back-end data being updated, so that the contents of this container don't change when something is dragged from it.

Protected Member functions

Slot MakeSlot(UnityEngine.Object obj, Slot preMade = null)	Instantiates a Slot and its child Draggable for this container, based on the slotPrefab and itemPrefab members. **obj** is the object to pass to the Draggable via UpdateObject(). **preMade** allows you to pass an existing Slot through instead of instantiating a new one, although an itemPrefab will still get instantiated. ObjectContainerList and ObjectContainerArray call this internally in their CreateSlots fucntions. For an example of using this manually, see the **EquipScreen** class.

ObjectContainerArray

An abstract class derived from ObjectContainer, that works with a C# array of Objects, automatically creating a slot for each one.
Your derived class doesn't need to call MakeSlot directly, but CreateSlots, which will automatically clone the Slot Prefab and Item Prefab for each element of the array.

Private and Protected Member Variables

UnityEngine.Object[] objects	A reference to a C# array of objects that lives outside of this class in the back-end data, eg. an array of Items in an Inventory. This array gets manipulated by the container in having values replaced, but is not resized or reallocated by the container.
Slot[] slots	An array of child Slot prefabs that the container instantiates, to pair up with each entry in the objects array. Each slot is typically instantiated with a child UI prefab for the item.

Member functions

void CreateSlots(UnityEngine.Object[] array)	This attaches the array reference to this container. Slot and Draggable prefabs are created for each element of the array using the **slotPrefab** and **itemPrefab** members inherited from **ObjectContainer**. Manipulating the Draggables will then automatically change the contents of the array we've passed in here.
void HighlightSlots(bool on)	A utility function provided to call the DraggableEnter of DraggableExit function on all slots. This is intended to be called from events. See **EquipScreen** for an example.
override void Drop(Slot slot, ObjectContainer fromContainer)	Override called by the Draggable code, which updates the contents of our objects array when a drop occurs.

ObjectContainerList

An abstract generic class derived from ObjectContainer, that works with a C# List of the generic type, automatically creating a slot for each one.
As above, but works with Lists instead of C# arrays.

Private and Protected Member Variables

List<T> objects	A reference to a generic List of objects that lives outside of this class in the backend data, eg. A List of Items in an Inventory. This List gets manipulated by the container in having values replaced, but is not resized or reallocated by the container.
Slot[] slots	An array of child Slot prefabs that the container instantiates, to pair up with each entry in the objects array. Each slot is typically instantiated with a child UI prefab for the item.

Member functions

void CreateSlots(List<T> list)	This attaches the given List to this container. Slot and Draggable prefabs are created for each element of the List using the **slotPrefab** and **itemPrefab** members inherited from **ObjectContainer**. Manipulation of the Draggables will now modify the List automatically.
void HighlightSlots(bool on)	A utility function provided to call the DraggableEnter of DraggableExit function on all slots. This is intended to be called from events. See **EquipScreen** for an example.
override void Drop(Slot slot, ObjectContainer fromContainer)	Override called by the Draggable code, which updates the contents of our objects List when a drop occurs.

Tooltip classes

The Tooltip classes are located in the \Assets\DragAndDrop\Scripts\Tooltips folder.

They aren't part of the core DragAndDrop library, but work with it so well that I've added them in here.

This tooltip system works by having a script on the TooltipCanvas that polls under the mouse for an object with an ITooltip interface. If it finds one, it displays the tooltip box and sets its text to the found object's desired tooltip.

Once this canvas is set up, all you need to do is make your UI classes implement the IToolTip interface.

ToolTip

A class representing the UI object that pops up when the mouse hovers over a tooltip.

IToolTip

An interface for anything that's going to draw a tooltip

ToolTipSimple

A component that implements a very simple IToolTip interface, where the string is hardcoded by the designer in the inspector

Example classes and Prefabs

The following classes and associated prefabs in the example scene illustrate how to extend the DragAndDrop core classes to build a functional UI.

Backend classes

These classes form the basis of a very simple RPG system. They have no dependencies on the UI classes.

Power

A simple ScriptableObject with a name, an icon and a colour, used to represent a power like Punch, Laser Beam, Force Field etc. In a real game this would have data describing damage, energy cost, range and so on.

PowerSet

Nothing more than an array of Powers. In a full game could represent a themed powerset like Fire Blast or Force Fields which a player picks powers from.

Charm

A simple item like you collect in RPGS. They have a name, icon and colour, a type(eg potion, helmet, boots) and a level requirement in order to equip them.

Player

A simple RPG character with a backpack for storing any Charms, a belt for Potion Charms, and some equipment slots for storing currently equipped items like armour and jewellery.

Player also has an array of Powers, representing the powers that the player can use.

Power UI Classes

PowerUI

A Draggable-derived class for representing a Power in the UI

Public Member Variables

Image icon	A reference to a child Image component used to display the icon
Text label	A reference to a child Text component used to display the power's name

Public Member Functions

override void UpdateObject()	Updates the icon and label fields (if they exist) to match the Power stored in the obj member inherited from Draggable.

This is a typical implementation of Draggable, where we have a set of fields relating to child objects which we fill in the UpdateObject override.

Prefabs

PowerUILabel

This prefab has an icon and label field. Its intended to work like the power bar in a RPG where you can push buttons to activate the player's powers, so it has a Button on the icon. The mouseButton is set to 1 as a result so that right mouse drags the button, and left mouse is used to click the button.

PowerUINoLabel

As above, but has en empty label field, since the names are displayed on the slots in the PowerSet part of the UI. The null-pointer checks in UpdateObject ensure that this is not a problem.

PowersUI

Displays the powers array of the Player specified in the inspector.

ObjectContainerArray does all the hard work here, leaving this script with just four lines of code in Start to call CreateSlots!

Used by the **Canvas/PlayerPowers** object in the scene.

PowerSetUI

Displays the powers from a List in a Powerset specified in the inspector.

ObjectContainerList<Power> does all the hard work, although here we add an extra function to specify that this type of UI is readonly, and dragging a power out of here will not remove it from the powerset, or re-order the powers.

Used by the **Canvas/PowerSet** object in the scene.

PowerSetSlot

A Specialised **Slot** that displays some attributes of the power slotted in it, used to show the name and description of a Power in the PowerSet UI.

This overrides the UpdateSlot function to write the slotted power's name and description to a couple of child Text fields, using the exact same pattern as PowerUI does. Note how it gets the object slotted through the child Draggable.

Prefabs

PowerSetSlot

This prefab has text fields for the power name and description set up. It also has a round slot object to slot the power in as a child of the top level object where the Slot script sits, so it uses the Slot member variable to specify the correct transform.

Slot prefabs

RoundSlot

A basic Slot prefab with no special events.

Charm UI classes

CharmUI

A **Draggable** derived class for displaying a **Charm**

Public Member Variables

Image image	A reference to a child Image component used to display the icon

Public Member Functions

override void UpdateObject()	Updates the image (if it exists) to match the Charm stored in the obj member inherited from Draggable.
string getToolTipMessage()	Implements the IToolTip interface, to pass the Charm's description to the tooltip.

Prefabs

CharmUI

A simple Image with a child image, referenced by the Image field in the CharmUI script. MouseButton of 0 means charms are dragged around by the left mouse button.

Slot prefabs

SquareSlotBasic

This prefab has a slot that plays a noise when an item is placed in it, via the **OnSlot** event.

SquareSlotHighlight

This prefab has a slot that plays a noise when an item is placed in it, via the **OnSlot** event.

It also changes the image of its slot in response to the **OnDragEnterCanSlot, OnDragEnterCannotSlo**t and **OnDragExit** events.

CharmsArrayUI

A ObjectContainerArray derived class for displaying one of the Player's inventory arrays. The Player class has two suitable arrays, so we use an enum here to distinguish between them – belt or backpack.

Public Member Variables

Player player	The player whose charms we're displaying.
Player.CharmList charmList	An enum saying which Charm array to display, the belt or backpack.
Charm.CharmType charmType	A bitfield mask for which charm types we can slot in this UI. The Backpack can house All, the Belt is just for Potions.
Text description	An optional Text field for showing the description of the last charm dragged out.

Public Member Functions

override bool CanDrop(Draggable dragged, Slot slot)	Checks that the item we're trying to drop in here is of the right kind of charm. This prevents us putting anything but Potions in our belt.
override void OnDragBegin()	If we have a description text field, set it to show the Charm's description here. Note that we must call base.OnDragBegin() here for this to work.

This component is used on both the **Inventory** and **Belt** items in the scene's Canvas. Both use a GridLayoutGroup and ContentSizeFitter to arrange the child components that get created by **CreateSlots()** in Start().

In both, the OnDragFail event is set up to play a “whoops” sound when the wrong item is placed in one of their slots.

The Inventory uses the SquareSlotBasic prefab for its slots, so they make a noise when an item is slotted.

The Belt uses the SquareSlotHighlight prefab, which automatically changes the slot when an item is dragged over it: to an expanded gold image if the item can be slotted, or a contracted grey image if the item cannot be slotted.

EquipScreen

An ObjectContainer derived class that doesn't use the array or List variants, but illustrates how to manage other data. This handles a number of Charm variables in the Player class like amulet, armour, helmet and so on.

This class works with a set of Slots that are arranged spatially in a roughly humanoid pattern.

Public Member Variables

Player player	The player whose equipments we're displaying.

public Slot helmet;
public Slot amulet;
public Slot ring1;
public Slot ring2;
public Slot gloves;
public Slot boots;
public Slot armour;
	References to pre-exiting slots that have been placed in the Scene view.

public Slot[] slots;	An array of slots made up from the fields above, and initialised in Start(), so we can iterate through all slots in HighlightSlots

Public Member Functions

override bool CanDrop(Draggable dragged, Slot slot)	Checks that the item we're trying to drop in here is of the right kind of charm for the slot we're dropping it into. Also checks the item's level requirements against the Player's level. The Player is Level 0 by default, change it at runtime to change what you can slot.
override void Drop(Slot slot, ObjectContainer fromContainer)	When drop is called, all Slots have their child Draggables updated with new obj fields. This function writes those back into the backend data of the Player's fields.
void HighlightSlots(bool on)	A utility function for making all slots respond with their own events as if an object had been dragged onto or off them, depending on the bool value passed in.

The **Equipped** GameObject in the scene's canvas implements this component.

Note how it already has child slots set up to allow the designer to tweak the humanoid shape. These are passed through to MakeSlot to prevent it from instantiating a new Slot object, although it will still instantiate a new itemPrefab for each slot.

The various child slots have been given good descriptive names, and dragged into the Helmet, Amulet, Ring1, etc fields.

It uses the SquareSlotHighlight slot prefab, so that when HighlightSlots is called, each slot will expand or contract to show the player where they can place the dragged item.

It calls HighlightSlots on itself from the OnDragEnter and OnDragExit events, which get called when the dragged object is placed over any of its slots, or when it leaves the combined airspace of all its slots.

This makes the whole EquipScreen respond when dragging Charms near it.