Generic UI List

Version 1.1.2
May 2020
© Kevin Foley 2019 - 2020
onemanescapeplan@gmail.com

Overview

The **UI List** package gives you a strong foundation from which you can quickly build lists to display any type of data. This package relies on C# generics, so a basic understanding of generics is necessary to work with the package.

Components of a UI List

To create a new kind of list, you'll need to write a small amount of code and create a prefab. A UI list is made up of the following elements:

- **Item Model**: A data type that represents the data for one item in the list. The model can be any C# data type (a primitive, struct, or class)
- **List Controller**: A component script which defines core functionality for the list
- **Item View script**: A component script which defines how data from a single model is displayed on an item view
- **Item View prefab**: A UI prefab used to display a list item
- **Selection Manager**: An optional component script that can be added to a list controller to add support for selecting/deselecting items in the list

Creating a list

For this example, we'll create a scoreboard - a list of player names and scores. This example is static - the scoreboard won't visually update if a player's score changes while the list is already displayed. To enable the items in the list to refresh, see the section "Data Binding".

The item model

First, we need to decide on the item model - the data type that will be displayed in our list. In this example, we want our list to display player names and scores, so we'll use a custom class:

```
public class Player implements IBindable {
    public string Name { get; set }
    public int Score { get; set }
    public Player(string name, int score) {
        this.Name = name;
        this.Score = score;
    }
}
```

The item view script

Next, we need to create a script that defines how data from our model is displayed on UI components. This will extend `UIListItemViewBase` with a concrete data type. Continuing our scoreboard example:

```
public class ScoreboardListItemView : UIListItemViewBase<Player> {

    [SerializeField] protected Text nameText;
    [SerializeField] protected Text scoreText;

    public override void Refresh() {
        if (model == null) return;
        nameText.text = model.Name;
        scoreText.text = model.Score;
    }
}
```

The `Refresh()` function is called by the base class when an item model is assigned to the view. This is where we display the data from our model on the UI elements in the view.

Notice that the `UIListItemViewBase` class is a generic class, so we have to specify the item model type (in this case, `Player`) when extending it.

The list controller script

We need a list controller which extends `UIListController` with concrete types. Continuing our scoreboard example:

```
public class ScoreboardListController : UIListController<Player,
ScoreboardListItemView> { }
```

That's it! Our list controller class is a one-line affair, because it inherits all of the functionality it needs from the generic base class `UIListController`. All we have to do is define the concrete types for our item model and item view script.

Building the UI

Lastly, we'll need to build the UI that displays our list. Start by creating a canvas and placing a **Horizontal Layout Group** or **Vertical Layout Group** component on a child of the canvas. If you want to have a scrolling list, you'll probably want to put the layout group inside a scroll view. Following the above example, add the **Scoreboard List Controller** component to the same GameObject as the layout group.

After setting up the list controller, add a child to the list and assign your item view component. Following the above example, add the **Scoreboard List Item View** component. Then add the two Text objects that will display the name and score, and link the item view component to them. Save this as a prefab, and assign the prefab to "prefab" field of the Scoreboard List Controller.

Using the list controller

Add one item:

```
listController.AddItem(new Player("Susan", 25)); //add to end of list
listController.InsertItem(0, new Player("Alan", 15)); //add at given index
```

Add a range of items to the end of the list:

```
var players = new List<Player>() { new Player("Tommy", 100), new Player("Sally", 250) };
listController.AddRange(players);
```

Set all of the items in the list:

```
listController.Data = players;
```

See if list contains an item:

```
bool inList = listController.Contains(player);
int index = listController.IndexOf(player);
```

Removing items:

```
bool foundAndRemoved = listController.RemoveItem(player);
listController.RemoveItemAt(index);
```

Clear the list:

```
listController.Clear();
```

Get the item count:

```
int count = listController.Count;
```

Pooling:

```
int itemsInPool = listController.CurrentPoolSize;
listController.MaxPoolSize = 10;
```

Data binding

Sometimes the data displayed in your list can change while the list is open. The scoreboard/leaderboard is a good example of this.

Say we have a deathmatch game in which the players can press Tab at any time to view the current scores. Bob presses Tab to view the score list, and sees that Alice has 3 kills. Just then, Alice gets another kill. We want her score to update immediately, without Bob having to close and re-open the score list. We need some way of updating the scores in the list as soon as they change.

A common but inefficient solution to this problem is to refresh the display every frame:

```
public override void Refresh() {
    nameText.text = model.Name;
    scoreText.text = model.Score;
}

public void Update() {
    Refresh();
}
```

That works, but it has unnecessary performance overhead. `Update()` gets called every frame, whether or not the data has actually changed. This is particularly problematic in lists with a large number of complex items.

Another common naive solution to this kind of problem is to give the data source (in this case, a Player class) a reference to the text field where the score is displayed. Such a solution might look something like this:

```
public class Player {
    public Text scoreText;
    private int score;

    public string Name {get; set;}
    public int Score {
        get { return score; }
        set {
            this.score = value;
            scoreText.text = value.ToString();
        }
    }
}
```

The above example is not a good solution! For one thing, it means the Player class relies on a reference to the UI, which is bad code design. For example, if we close the score list and the Text object is destroyed, the Player class will now contain an invalid reference. The next time we change the player's score, we'll get an error from Unity: `MissingReferenceException: The object of type Text has been destroyed but you are still trying to access it.` We could add a null check, but that is treating the symptom instead of the problem.

A solution

A better way to handle this is to use events to let the UI know when to update. The UIList package offers a simple way to do this: have your data class implement the IBindable interface, and invoke the ChangeEvent each time the model changes in a way that will require the UI to update.

```
public class Player implements IBindable {
    private string name;
    private int score;
    private UnityEvent changeEvent = new UnityEvent();
    public UnityEvent ChangeEvent => changeEvent;

    public string Name {
        get { return name; }
        set {
            this.name = value;
            changeEvent.Invoke();
        }
    }
    public int Score {
        get { return score; }
        set {
            this.score = value;
            changeEvent.Invoke();
        }
    }
}
```

Here we've implemented the `IBindable` interface and added the necessary private variable `changeEvent`. Notice that we invoke the `changeEvent` when the player's name or score is changed. That's it! When you use the `Player` class as the data model for the list, `UIListItemViewBase` will automatically detect that the model implements `IBindable` and listen to the `ChangeEvent`, and refresh the view when the player's name or score changes.

To see binding in action, check out the "Binding" example in `OneManEscapePlan\UIList\Examples\Binding.`

Pooling

Instantiating and destroying GameObjects has performance overhead. In many cases, the impact is too small to notice, but if you're constantly creating and destroying large numbers of objects, the application framerate can suffer - particularly on mobile devices, which have very limited CPU power compared to modern desktops and laptops.

Object pooling is a solution to significantly cut back on performance overhead by recycling objects. Rather than destroying instances when we no longer need them, we return them to a *pool*, where they remain inactive until needed. The next time we need an instance, instead of creating a new one, we simply grab an object from the pool.
The `UIListController` supports pooling item views for better performance. It has two properties that control pooling:

- **Max Pool Size**: the maximum number of item views that can be stored in the pool. Set to 0 to disable pooling, or -1 for unlimited pool size
- **Starting Pool Size**: the number of inactive item views that will be added to the pool when the list is first created. This can be useful if you expect that the list will grow some time after you create it, and want to reduce the CPU cost of displaying new items in the list in the future. Setting this value too high can waste memory and cause your application to hiccup when the list is first created.

Pooling reduces CPU usage when items are removed from and added to the list, but slightly increases memory usage (since unused items in the pool remain in memory as long as the list exists). Keep in mind that **pooling is most beneficial for frequently changing lists**. If you never add items to or remove items from the list after it is first created, pooling won't come into play.
See the `Performance Test` example to witness the power of pooling in action. In this example, each frame the list is cleared and 100 new items are added. The "elapsed" field indicates how long it took to regenerate the list over the last frame, in ticks, while the "average" field shows a running average.

Advanced list controller types

This package includes two additional types of list controller for advanced use-cases:

Scrolling from code

Use `ScrollingUIListController` as the base class for list controllers that need the ability to scroll to a specific item from code. This adds two public methods, `ScrollTo(ItemModel`

`model)` and `ScrollTo(int index)`. You **do not** need to use this class to support user-scrolling in a `ScrollRect`; it is only used for scrolling to specific items from code.

Virtualized lists

In a traditional list, we create one item view for each item of data in the list. For example, if our list is a scoreboard and we pass it 22 players, it's going to create and display 22 item views. This works fine for most lists, but can cause serious performance problems if our list contains too much data. Creating hundreds or thousands of item views at once can cause the application to noticeably stutter, and scrolling through all those item views can cause major framerate drops.

By contrast, in a ***virtualized*** list we only create as many item views as are necessary to fill the screen. When a view scrolls off one side of the screen, we immediately recycle it to the other side of the screen. This greatly improves performance both in creating the list and scrolling through it.

Version 1.1.0 of this package adds a new class `VirtualizedUIListController` which can be used as a base class for lists that require virtualization. Please note that **this is a BETA feature** and it may contain bugs or other issues.

One limitation of the `VirtualizedUIListController` is that it does not work well with Unity's `LayoutGroup` components; this means you must manually specify the size and alignment of your items in the list controller inspector. For this reason, **you should rely on regular list controllers for most cases and only use a virtualized list controller for lists that have so many items that it causes performance issues.**

Examples

The project includes several example scenes to demonstrate how to build and use lists, located in `Assets\OneManEscapePlan\UIList\Examples\`:

- **String List**: A simple list of names. Controls are provided to add items to and remove items from the list.
- **Binding**: Demonstrates how we can use the `IBindable` interface to automatically update a list item view when its model changes. Select an item in the list and then use the slider to change the value.
- **Multiselect List**: A list that supports selecting several items at once
- **Performance Test**: Use this to compare performance with pooling enabled and disabled. A list of 100 items is re-generated every frame.
- **Virtualized List:** Uses a virtualized list to display 1000 items with no performance issues.