UI Toolkit Sample - Dragon Crashers
A mini manual

About this guide

This guide will help you understand the structure, organization, and game logic of the official Unity sample UI Toolkit Sample – Dragon Crashers. The sample shows game menus, from simple to advanced, implemented in UI Toolkit using UI Builder. We recommend you reference this page with the sample open in the Editor.

The guide mentions a number of features of UI Toolkit, but it does not cover each one in detail. UI Toolkit documentation is here.

The sample was released together with the e-book User interface design and implementation in Unity, which provides instructions for the fundamental techniques and features for creating UI in Unity.

Unity 2022 LTS is the recommended version of Unity for the sample.

Introduction to the game menus

The menu screens from *UI Toolkit Sample – Dragon Crashers*

When you open the sample in the Editor, you'll see a menu bar on the left that will help you navigate the modal main menu screens. Here's a brief introduction to each of the menus in the image above:

1. **The home screen**: This serves as a landing pad when launching the application. Use this screen to play the game or receive simulated chat messages.

2. **The character screen**: This involves a mix of GameObjects and UI elements. Explore each of the four *Dragon Crashers* characters. Use the stats, skills, and bio tabs to read the specific character details, and click on the inventory slots to add or remove items. Finally, level up each character in typical RPG fashion using acquired potions.

3. **The resources screen**: This links to Unity documentation, forums and other resources for making the most of UI Toolkit.

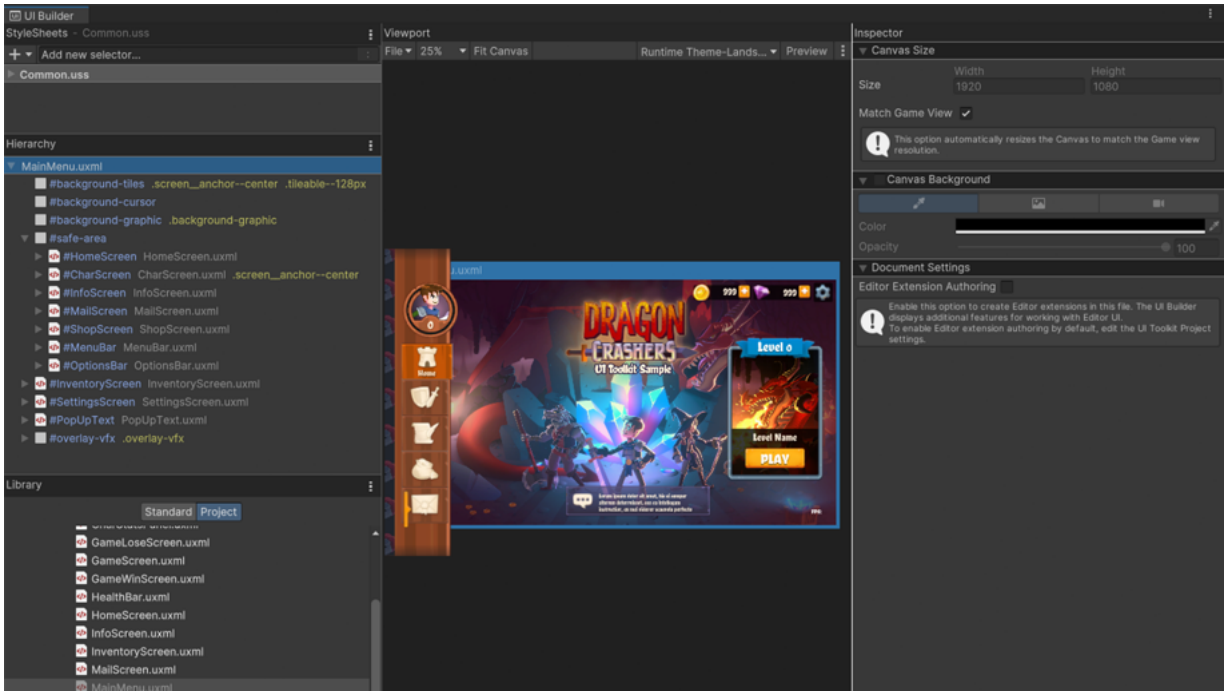
4. **The shop screen**: This simulates an in-game store where you can purchase hard and soft currency, such as gold or gems, as well as virtual goods like healing potions.

5. **The mail screen**: This is a front-end reader of fictitious messages that uses a tabbed menu to separate the inbox and deleted messages.

6. **Settings:** Click on the settings icon in the top-right corner to open up the settings screen. You can also click the gold and gem buttons to go directly to each respective tab of the shop.

7. **Backend:** In order to focus on UI design and implementation, the sample project simulates backend data like in-app purchases or mail messages, using ScriptableObjects. You can customize this stand-in data via the Resources/GameData folder.

Scenes and main interface



The MainMenu scene from the sample

The project includes two scenes under the /Scenes folder, **MainMenu** and **Game**.

The **MainMenu** scene is the hub for all meta aspects of the game UI, such as character selection, inventory, and shop. The MainMenu GameObject has a UI Document component that holds a reference to the **MainMenu.uxml** file. The UXML file is the **visual tree** asset.

You will need to switch to the Game view to view the interface because the UI Toolkit interface does not show in the Scene view.

The **Game** scene is the gameplay portion of the sample and shows in-game overlays like health bars, a drag and drop zone for potions, and a floating pause menu with a blur effect. The UI Document is part of the GameScreen GameObject.

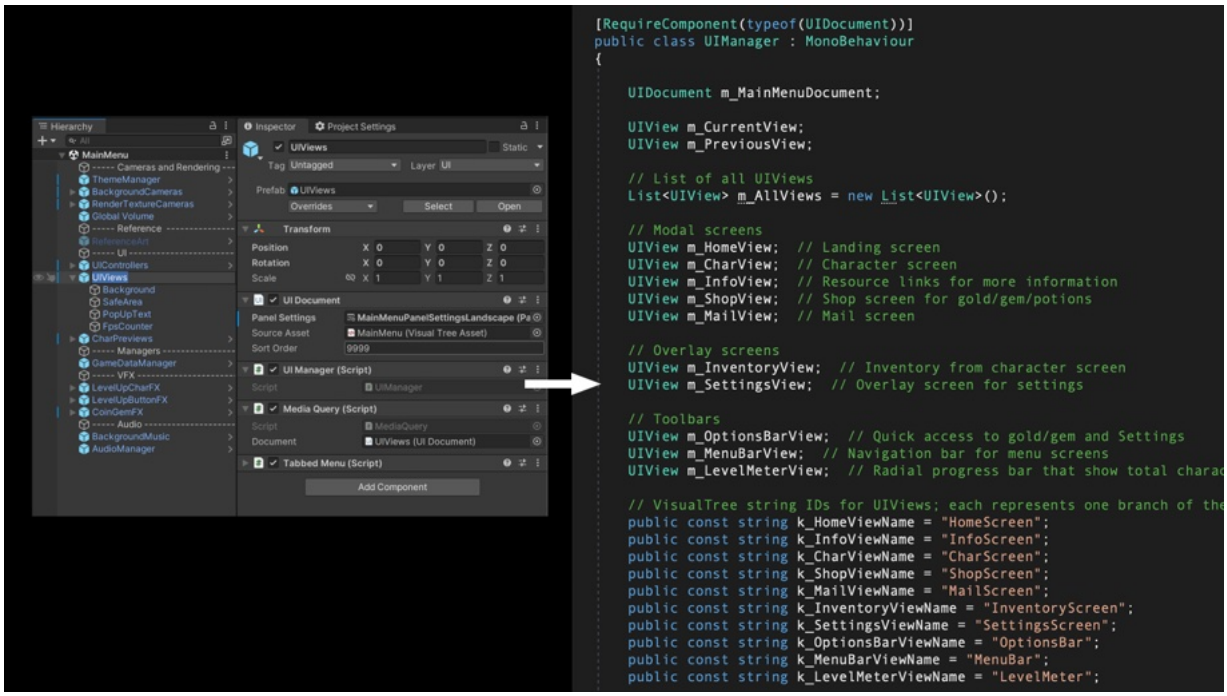
To modify the MainMenu interface in UI Builder, open the **MainMenu.uxml** file from the Project section of the Library window. Enable the **Match Game View** option in the Inspector to preview the elements with the same spacing as the Game View.

The MainMenu.uxml file is the parent visual element in which all the different screens, stored as UXML files, are embedded. Embedded elements function as templates and appear grayed out. To edit them in isolation, right click on the UXML file in the Hierarchy and select **Open Instance in Isolation**.

Different modal screens make up the interface: #HomeScreen, #CharScreen, #InfoScreen, #MailScreen, #ShopScreen, plus the #InventoryScreen and #SettingsScreen overlays. A #MenuBar used for navigation also draws on top of the other UI screens.

Clicking the buttons of the MenuBar navigates through the various screens by hiding and showing the appropriate Visual Elements in the Hierarchy.

UI management



The UI manager in the Hierarchy view

The **UIManager** on the UIViews GameObject serves as a high-level manager for various parts of the main menu. It keeps track of different UIs and shows and hides the screens that comprise the interface.

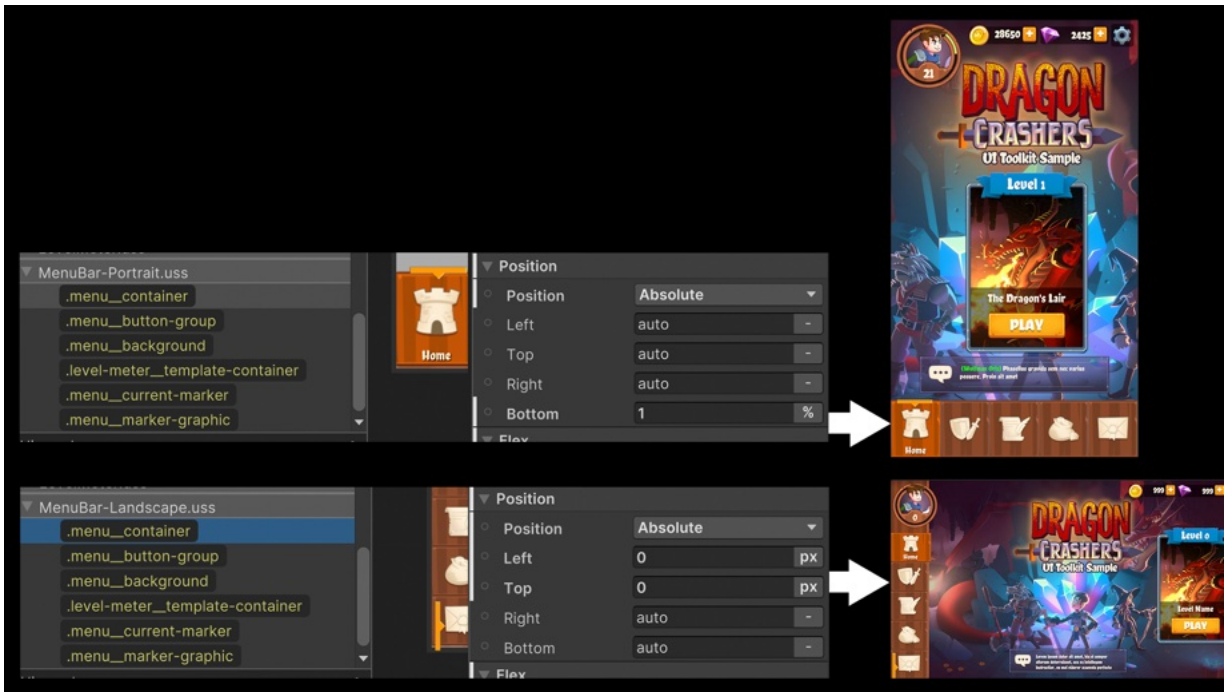
A **UIView** class acts as a base class for different portions of the UI. Each UIView is a functional unit of the interface. In the sample, the UIManager assembles the user interface from these UIView objects.

The UIView class does not derive from MonoBehaviour. Instead, the lifecycle of a UIView starts with its constructor, which can be used to initialize fields or settings.

Each interface screen has a dedicated UIView script — **HomeView**, **CharView**, or **InfoView** — that controls its behavior. These scripts, in turn, can be made of smaller UIView objects. For instance, MailScreen is managed by a **MailView** component, which references three other UIView components.

The navigation buttons in the **MenuBar** send event messages to the UIManager. These event messages call the **Show** and **Hide** methods on the correct UIViews.

Responsive UI for landscape and portrait



Different versions of the same selector used in each theme

You can adapt your UI to accommodate both portrait and landscape mode for mobile devices. One approach is to take advantage of Theme Style Sheets (TSS) in UI Toolkit.

The sample project includes a custom MediaQuery component attached to the UIViews GameObject. This listens for changes to the screen resolution and notifies a ThemeManager script if detected. The ThemeManager then toggles the active theme based on the updated aspect ratio.

Why use themes to handle different screen orientations?
Themes offer an efficient way to adjust the UI without needing a full redesign. A theme in UI Toolkit is simply a collection of Unity style sheets (USS) that work together.

When you define a TSS you have a quick mechanism to change several associated styles at once. This allows you to store portrait or landscape spacing and positioning within a specific theme and then apply that to the PanelSettings at runtime.

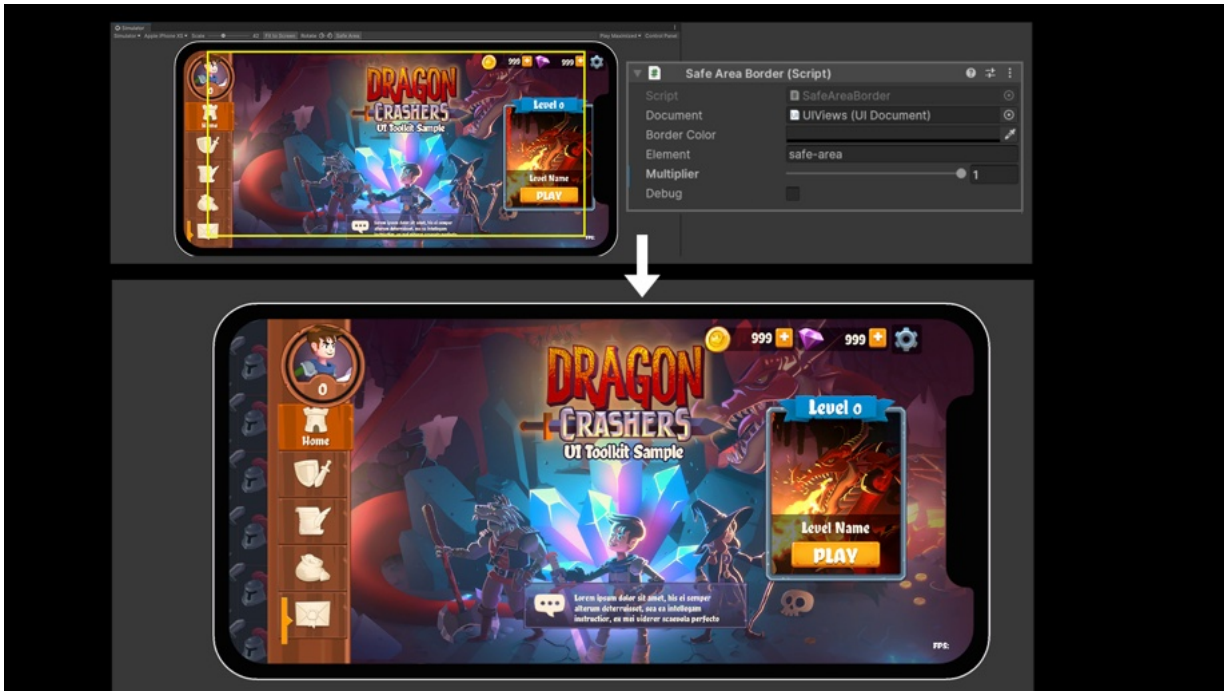
The sample features portrait and landscape base themes, each with two possible seasonal variations, resulting in six themes (e.g., RuntimeTheme-Landscape, RuntimeTheme-Portrait, RuntimeTheme-Landscape--Halloween, etc.).

Consider the **MenuBar.uxml** file. The RuntimeTheme-Landscape places the MenuBar buttons vertically on the left of screen, while RuntimeTheme-Portrait aligns them along the bottom. The UXML structure remains constant; only the theme changes. This allows for a seamless transition between styles without altering the visual tree hierarchy.

Though the UXML remains the same in each case (with the identical selectors), the layout can be customized for portrait or landscape as needed.

Note: When working in the UI Builder, make sure that you've chosen the correct theme to avoid unintended changes.

Safe area for mobile devices



Use the border width to keep UI elements within the safe area.

If you're building for mobile devices that have rounded corners or notches, the sample project demonstrates how to use the Screen.safeArea API for better placement of interactive UI elements.

Use the Device Simulator to preview the safe area on a variety of devices.

One technique is to place every modal UI screen within a single container. A script can then dynamically adjust the container's border width based on the safe area, ensuring that this border applies to all child elements.

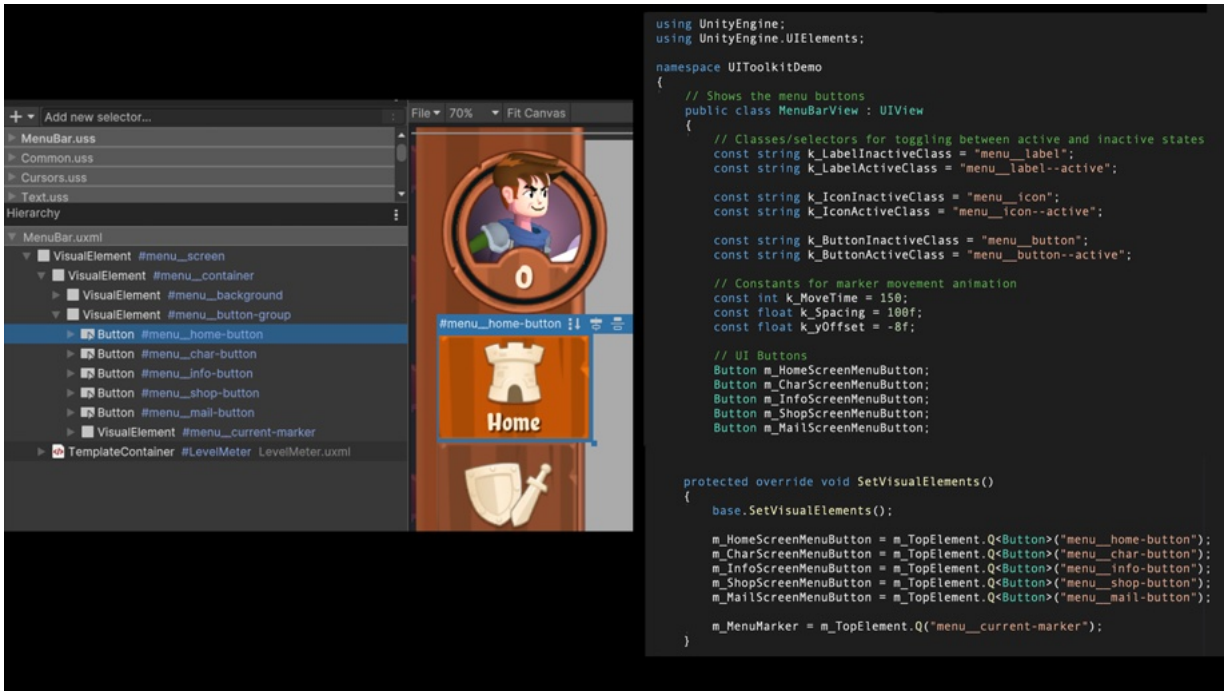
This safeguards against the unintentional hiding of any UI elements by the device's screen edges.

To better visualize the safe area, we employed a repeating fill pattern defined in USS or overlayed a color using a script.

The provided SafeAreaBorder script also features a multiplier adjustment, which allows fine-tuning of the border width to optimize screen real estate.

For UI elements that need to bypass the safe area, position them outside the designated safe-area container in the UXML Hierarchy. This enables you to integrate elements like background graphics or overlay effects with your menu screens.

Naming standards



Names of style sheets, buttons and visual elements in the sample

Designers and programmers can collaborate more effectively when they use naming conventions for visual elements and style sheets. Consistent naming helps to keep the hierarchy organized in UI Builder.

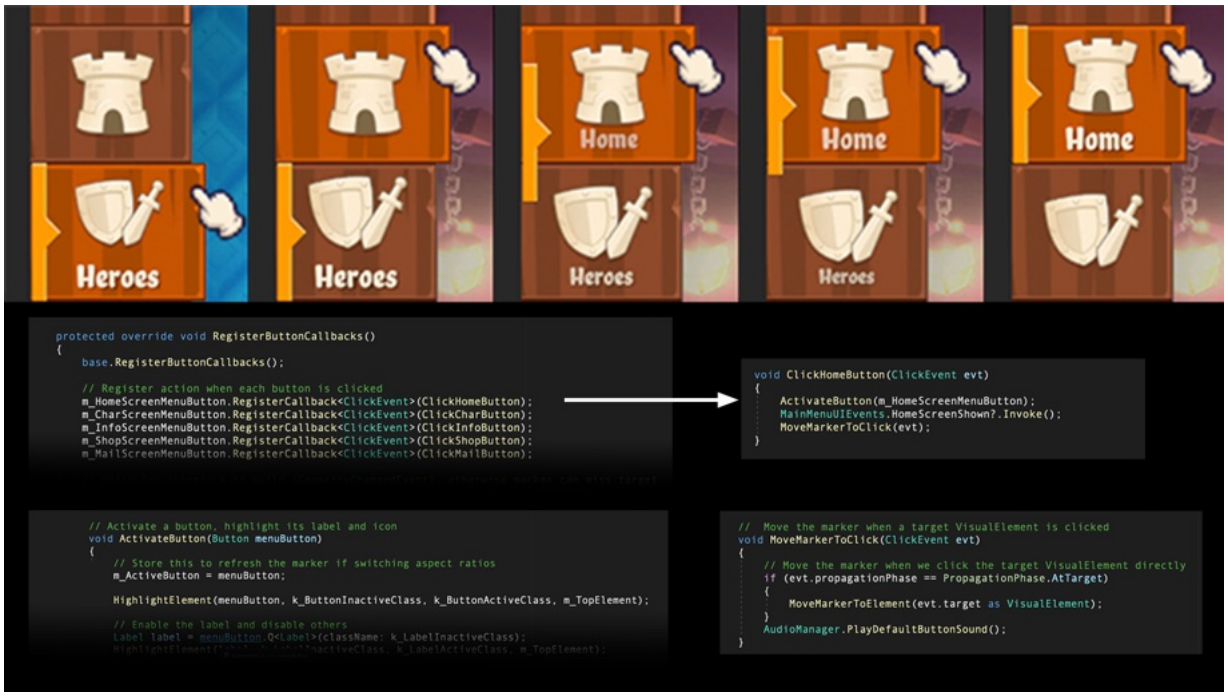
This is especially important for UI Toolkit, since string identifiers are used with UQuery to locate elements in the visual tree. For example, the query

`root.Query<Button>("foo").First();` locates the first button element with the identifier "foo."

While not required, the Block Element Modifier (BEM) naming convention is recommended for both visual elements and style sheets. BEM-formatted names like `menu__shop-button` or `button-label--selected` quickly convey an element's function, location, and relation to surrounding elements.

You can store element names as read-only constant variables or define them directly inline. For additional naming guidelines, consult the C# style sheet included in the project or download the e-book *Create a C# style guide: Write cleaner code that scales*.

Implementing functionality



Using code and USS transitions to create a nice transition effect

UI Toolkit comes with its own event system that connects user interactions to visual elements. Although UI artists might not set up events, understanding the process can help them design functional mock-ups that are easy to hand off to programmers.

In UI Toolkit, a callback is registered to an event like this:

```
myVisualElement.RegisterCallBack<MouseDownEvent>(myFunction);
```

This will trigger the designated `myFunction` when the user clicks `myVisualElement`.

Consider the MenuBar in the sample project. When you click a button, it highlights to a different color, and the orange arrow marker moves to the selection. This also notifies the UIManager to display the proper UI screen in the background.

The **ClickHomeButton** method does three things:

- **Style the Button:** A HighlightElement method swaps out style classes. The previously active button reverts to its inactive style.
- **Move the Marker:** The function MoveMarker uses the mouse click event and sets a new marker position.
- **Toggle the screen:** The MenuBar notifies the UIManager through event messages that the HomeButton has been clicked. The corresponding Home Screen becomes visible while the previous menu screen is hidden.

By registering callbacks, you can thus add complex runtime behaviors to your buttons and other elements.

The shop screen: A complex window

One of the most complex interfaces can be found in the shop screen

The shop screen is one of the more complex menus in the sample. This UI shows virtual goods organized under separate tabs: coins, gems, and potions. Items for purchase appear within a ScrollView in the main UI container.

The UI system uses four custom scripts that work with the **ShopScreen.uxml**:

- **ShopItemSO:** This ScriptableObject holds the statistics for each item that appears in the shop.

- **ShopItemComponent:** This translates game data from the ScriptableObject into UI text and graphics for a single item.

- **ShopScreenController:** This handles the actual game logic to set up the shop. The controller reads the ScriptableObject assets from the Resources and stores them in categorized lists (`List<ShopItemSO>`), one for each item type (coins, gems, potions).

- **ShopView:** This UI class renders the UI, updating the display based on events from the controller.

Shop items come in two layout sizes: Standard size and one for special promotions. These sizes correspond to the CSS selectors, `shop-item__size--normal` and `shop-item__size--wide`, in the **ShopItem.uxml**.

When the user opens the shop screen:

1. The ShopScreenController reads the ShopItemSO script into a categorized list, which is the data source for the available items in the shop.

2. Selecting a tab sends an event message to the controller, and this selects the corresponding items by category – coins, gems, or potions.

3. The ShopView script processes this filtered list and instantiates a visual tree asset for each shop item within the ScrollView's content area.

4. Each ShopItemComponent translates shop item data into visual elements, including text and images.

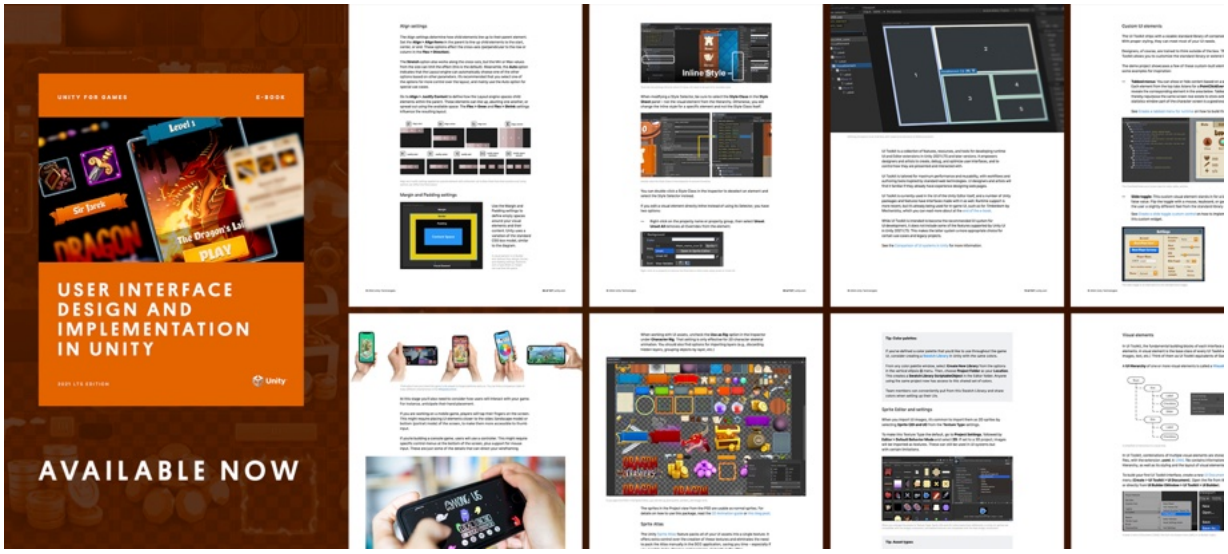
While the tabbed interface in ShopView may appear to be opening and closing tabs, it's more of a visual trick. Style changes simply affect the tab colors as the content in the Scrollview gets updated.

Additional notes:

- The MailView interface works in a similar fashion to toggle between the inbox and deleted messages. This contrasts the CharStatsView, where the tabbed interface cycles between three separate containers.

- Customization of the Scrollview's default dragger uses a selector that matches the default name in Unity and overrides specific parameters.

Get the e-book for full instructions



This article only covers the organization of the code and data in the sample. Of course, there are many more techniques and features in UI Toolkit to know about to unlock its full potential for your game UI, making it feel fun, juicy, and cohesive to players.

Written and reviewed by technical and UI artists – external and Unity professionals alike – the guide starts out by covering UI design and art creation fundamentals, and then moves on to in-depth instructional sections on UI development in Unity.

Download the e-book

You can more advanced e-books and articles for programmers, artists, technical artists, and designers in the Unity best practices hub.