QuizU documentation

Documentation for the QuizU - an official Unity sample demonstrating MVP, state pattern, managing menu screens and much more using UI Toolkit available on the Unity Asset Store.

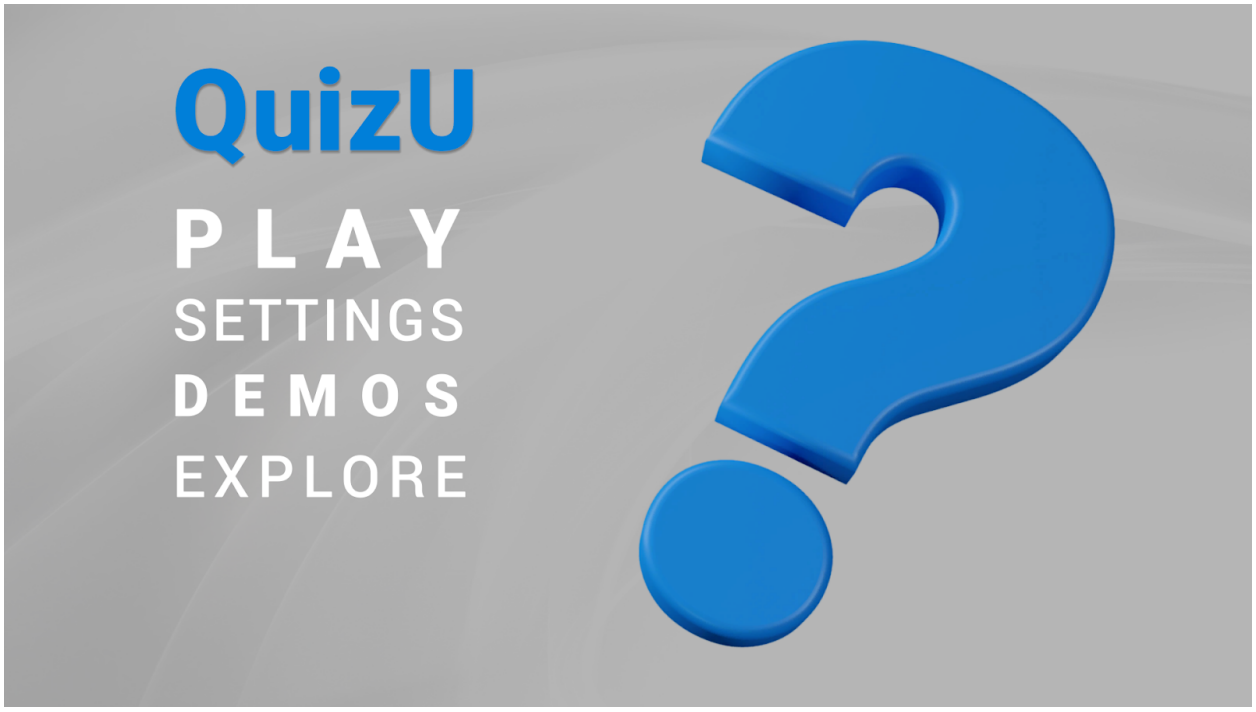
Version 1.01 Nov 1st 2023

Table of contents

Welcome to the new sample project QuizU

QuizU is a sample of an interactive quiz application that shows how UI Toolkit components can work together, leveraging various design patterns, in a small but functional game, complete with multiple screens and game flow management.

QuizU was created mainly with programmers in mind, explaining UI Toolkit concepts and tools from a developer perspective. QuizU implements common design patterns and event-driven architecture. Think of it as a study tool or a starter project for a larger application. And don't forget to play the game, too – it's a fun way to test how well you know Unity or perhaps prepare for a Unity certification.



Test your knowledge in QuizU or use it as a starting point for creating your own quiz game.

Key features

UI Toolkit demos
QuizU is designed to help make it easier to get started using the UI Toolkit. The project provides 10 small, digestible samples that demonstrate different aspects of UI Toolkit.

A mini quiz game with UI management
The sample project showcases a UI-centric quiz game that implements various patterns

for managing UI and game flow. It offers practical insights into handling multiple UI screens or breaking a large UI into smaller displays.

The QuizU project is a UI Toolkit-based game sample.

Design patterns

QuizU also showcases several core design patterns, including the usage of the state and model-view-controller (MVC) design patterns. These can help anchor your game architecture, so new features don't break your existing application.

Event-driven development

Here, the game components communicate with each other through events, promoting loose coupling for scalability and testability.

UI Toolkit demos

The QuizU project in Unity UI Toolkit offers a variety of standalone demo scenes, each serving as a bite-sized showcase of a specific UI Toolkit topic.

Think of these scenes as a set of recipes while exploring the UI Toolkit. Here's a brief sampling of what's inside:

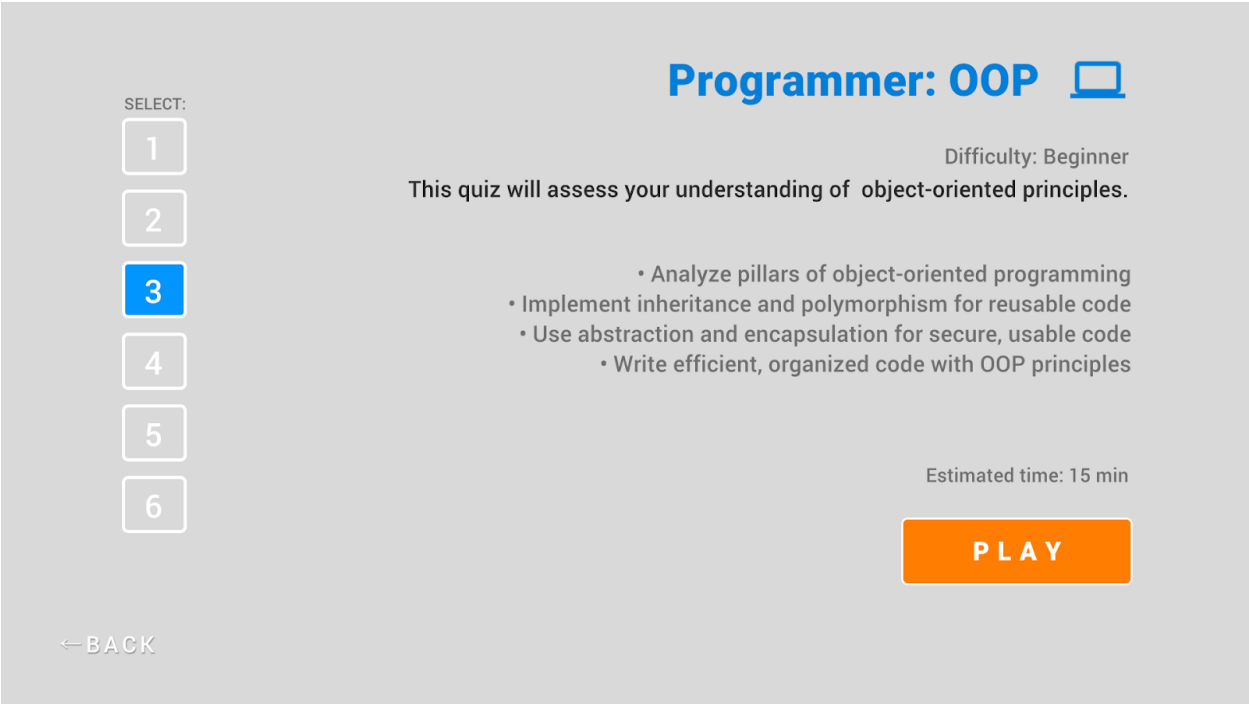
- **UXML and Visual Trees:** UXML files form a hierarchical structure of UI elements. These visual trees serve as a blueprint for your user interface.
- **Flexbox:** The Flexible Box Layout Model (flexbox) provides an efficient layout model for arranging UI elements dynamically within a container.
- **Unity Style Sheets (USS):** USS allows developers to customize UI elements with predefined styles. Reskinning your UI is just a matter of swapping style sheets.
- **UQuery:** UQuery simplifies the process of searching through a complex hierarchy of UI elements, enabling seamless navigation to specific UI components within the visual tree.
- **Pseudo-classes:** Pseudo-classes can be used to create interactive and animated UI elements with minimal extra code, adding extra 'juice' to your visual interface (e.g. enlarging a button when hovering over it or changing a text field color after selection).
- **UI Toolkit Event System:** UI Toolkit has its own complementing event system, designed to handle your UI's clicks, changes, and pointer input, even across complex hierarchies.
- **Manipulators:** Encapsulating related event callbacks into a single class, a manipulator promotes reusability and makes it easier to define user interactions (e.g. a click-and-drag manipulator for an inventory system, a gesture manipulator for a pinch-to-zoom effect, etc.).
- **Custom Controls:** The demo shows how to define and instantiate custom VisualElement through UxmlFactory and UxmlTraits classes. These custom controls can then be reused through scripts or the UI Builder.

As you dive deeper into the UI Toolkit, use these demo scenes as a source of inspiration or a reference guide. Then, you'll be ready to create your own rich, interactive user interfaces.

The mini-game

While the sample demos can help you understand specific topics in UI Toolkit, we've also compiled many of these techniques into a mini-game so you see how UI Toolkit fits into a more complete project.

Select a Quiz topic and then test your own Unity knowledge.



Test your Unity knowledge in QuizU.

The visual style is minimalistic so you can focus on the mechanics of putting the UI together without getting lost in the design details. This way you can see clearly how the UI elements combine into a cohesive user interface. It also makes it easier for you to reskin and apply your own theme using the project as a boilerplate template.

To help you manage your UI Toolkit-based interfaces, the project also demonstrates a stack-based navigation system for handling multiple screens. An upcoming article in this series explains this navigation system in detail, but for now, here are its main components in brief:

- Several Visual Tree Assets combine into a single UXML in UI Builder (UIScreens.uxml). Each visual tree corresponds to one UI in the application. A UI Document component reads this master UXML and then generates the relevant on-screen UI from that.

- A stack-based state machine – or "screen stack" – manages the UIs. Each screen is a fullscreen UI under the control of a UIManager class and is treated as a layer of a stack. Think of it like working with a pile of plates: You can only interact with the top plate, and to get to a plate beneath the top one, you have to remove plates on top of it. The last plate you put on is always the first one you take off.

- When a new screen opens, the UI Manager pushes onto the top of the stack and makes it the active state. When a screen is closed, it pops off the stack. This is a simple and effective way to navigate between modal screens.

- More complex UI Screens can be broken into smaller visual parts. The main Game Screen is a composite of several smaller displays. Each smaller “sub-screen” is handled independently; it can interact with the other smaller sub-screens or with the main UI Screen. This keeps everything more modular and reusable.

Whether you are creating a menu system for a game jam or developing a more complex project, this stack-based navigation system can provide a flexible and scalable framework for managing multiple UI screens. Learn more in our stack-based navigation system section.

QuizU: Exploring the demo scenes

UI Toolkit offers a comprehensive suite of features and tools to support you in building runtime UIs for game applications and Editor extensions. UI Toolkit introduces a new workflow and architecture that offers several improvements and advantages over UI development with UGUI.

UI Toolkit comes with a learning curve but also a number of advantages:

- **Scalability and Performance:** UI Toolkit is designed to be more efficient and performant, particularly for complex user interfaces.

- **Style Separation:** UI Toolkit separates styling from the logic using USS (Unity Style Sheets), similar to CSS in web development. This can make your UI code cleaner and easier to maintain. Designers can iterate on aesthetics without touching any code.

- **UXML Templates:** You can define reusable templates and instantiate them in your code. This can make it easier to design complex UI layouts.

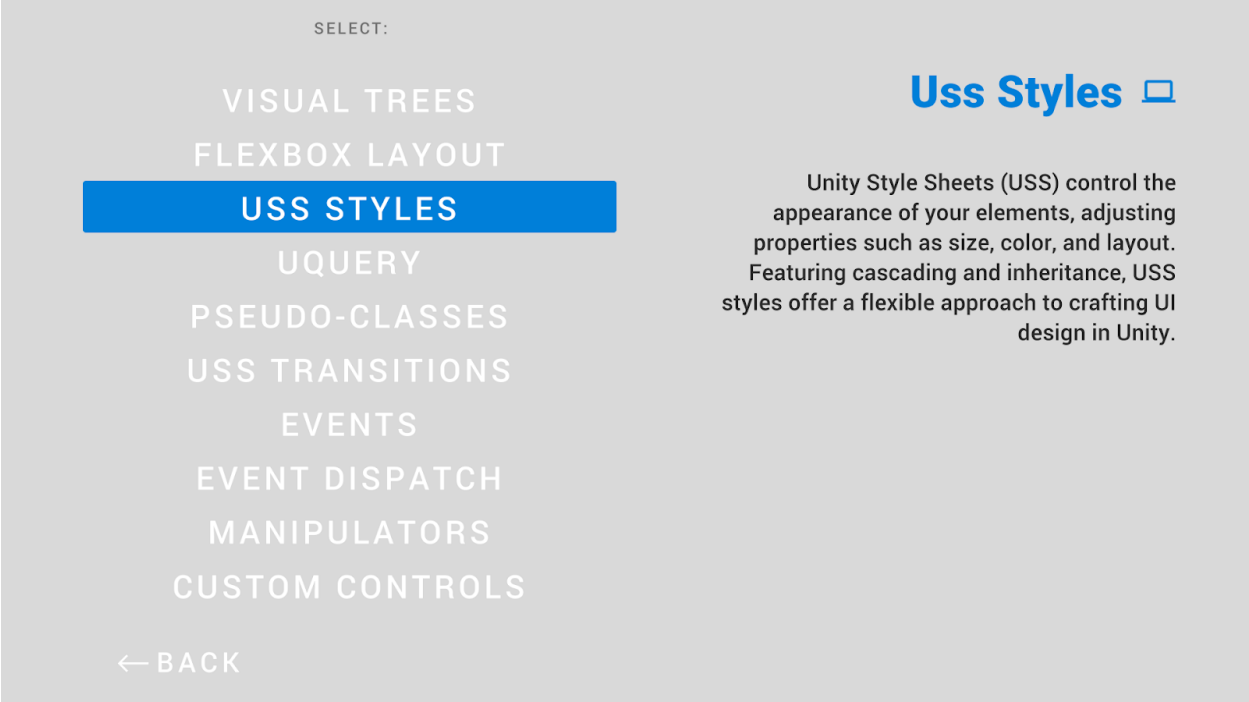
- **Unified support for runtime and Editor:** The UI Toolkit works in both the Unity Editor and at runtime, allowing you to use the same system for creating custom Editor windows and in-game UI.

The QuizU sample includes several standalone scenes that illustrate key concepts within the UI Toolkit. Each one represents a specific technique or feature. Consider these scenes as a set of recipes to inspire and guide you as you evaluate UI Toolkit for your next project.

Demo selection

Navigate to the Demo Selection Screen via the Main Menu screen, or if you prefer, disable the bootloader scene **(Quiz > Don’t Load Bootstrap Scene on Play)** and load each scene manually.

Let's break down some concepts highlighted in these demo scenes.



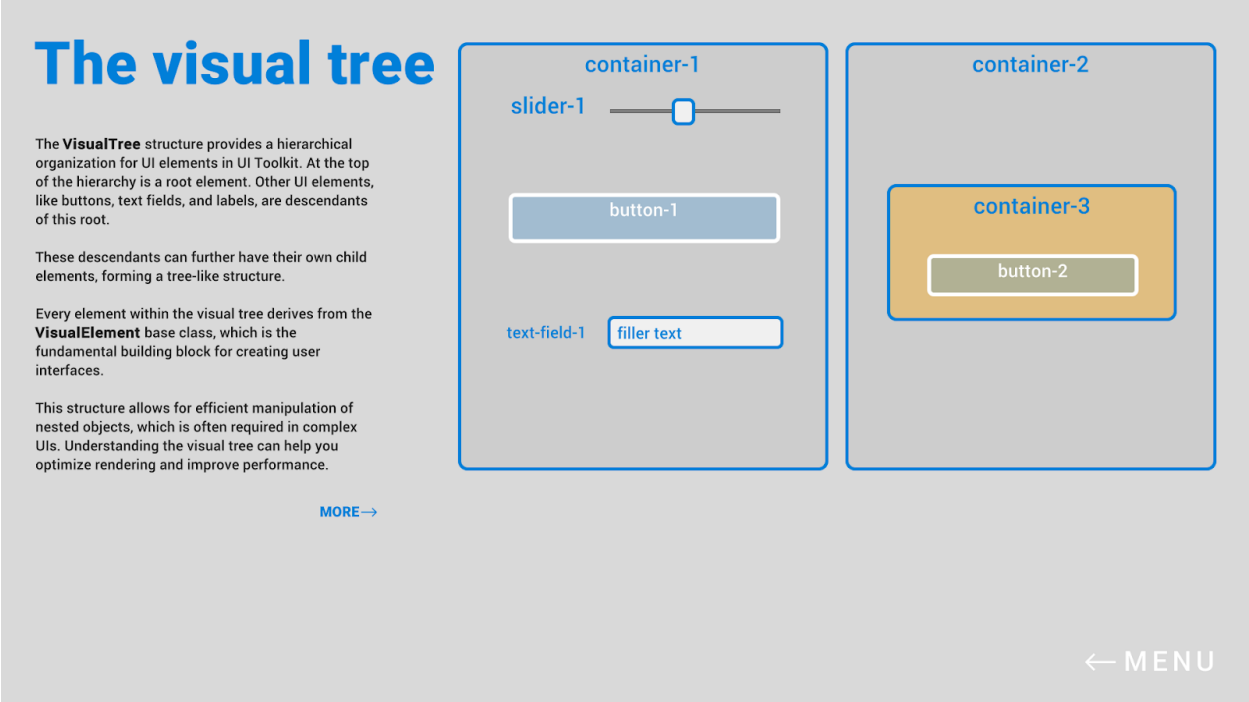
Select a demo from the Demo Selection Screen.

UXML and visual trees

UXML, or Unity XML, is the markup language used to structure and organize user interfaces in UI Toolkit. It provides a blueprint for constructing a visual tree, a hierarchy of UI elements. Every element declared in a UXML file corresponds to one node in the visual tree.

UXML files can also serve as reusable templates for UI elements, much like prefabs do for GameObjects. When a UXML file (or template) is loaded or instantiated, a corresponding visual tree is created in memory.

Consider UXML files as building blocks for UI structures and controls. UXML supports nesting, allowing your team members to work on individual UXML files and then combine them into a master hierarchy. This can prevent merge conflicts and promote modularity.



The demo introduces the visual tree hierarchy.

GroupBox versus VisualElement

The example features a GroupBox to group elements visually with a built-in border and label. Alternatively, you can customize a VisualElement with your own styling and add a Label to achieve a similar look.

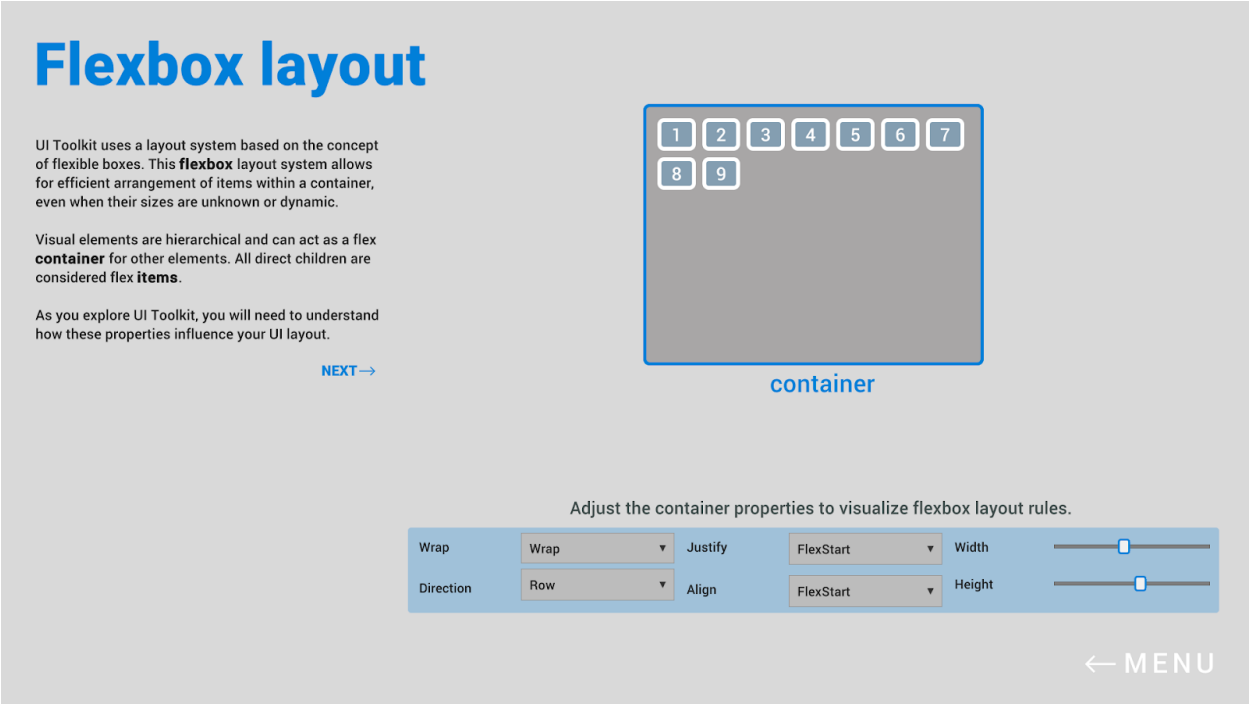
Flexbox rules and layout

Flexbox is a layout model inspired by web technologies that allows for the arrangement of UI elements within a container, even when their sizes are unknown or dynamic.

These rules allow for responsive design, adapting designs to various screen sizes and resolutions. Both row and column direction, control of the wrapping of elements, and managing the growth and shrinkage of elements based on available space are all possible with flexbox rules.

Building on the concept of visual trees in UI Toolkit, visual elements can act as flex containers using the flexbox layout rules. When functioning as parents, these containers can help organize child elements in an efficient manner.

Our **Flexbox** demo scene walks you through the basic rules of working with a flex container and its children. For more information on element positioning, see the UI Toolkit Layout Engine documentation.

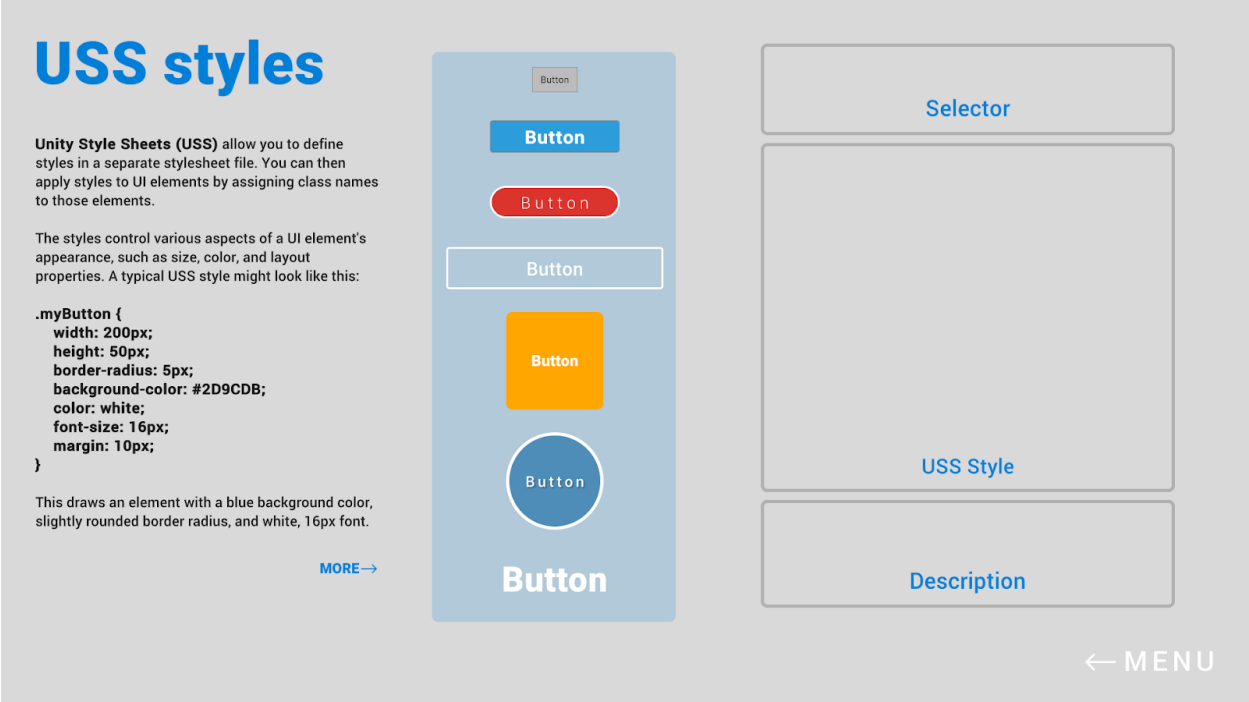


Adjusting the Flexbox properties.

USS styles

Unity's UI Toolkit provides a flexible system of styling using USS (Unity Style Sheets). Similar to their CSS equivalents, USS style sheets allow you to change the appearance of your elements by swapping predefined styles.

In the demo scene, compare the stylesheet code for a default button and several variants with different styles applied. Modifying font styles, colors, sizes, and positions is as easy as applying .uss files.

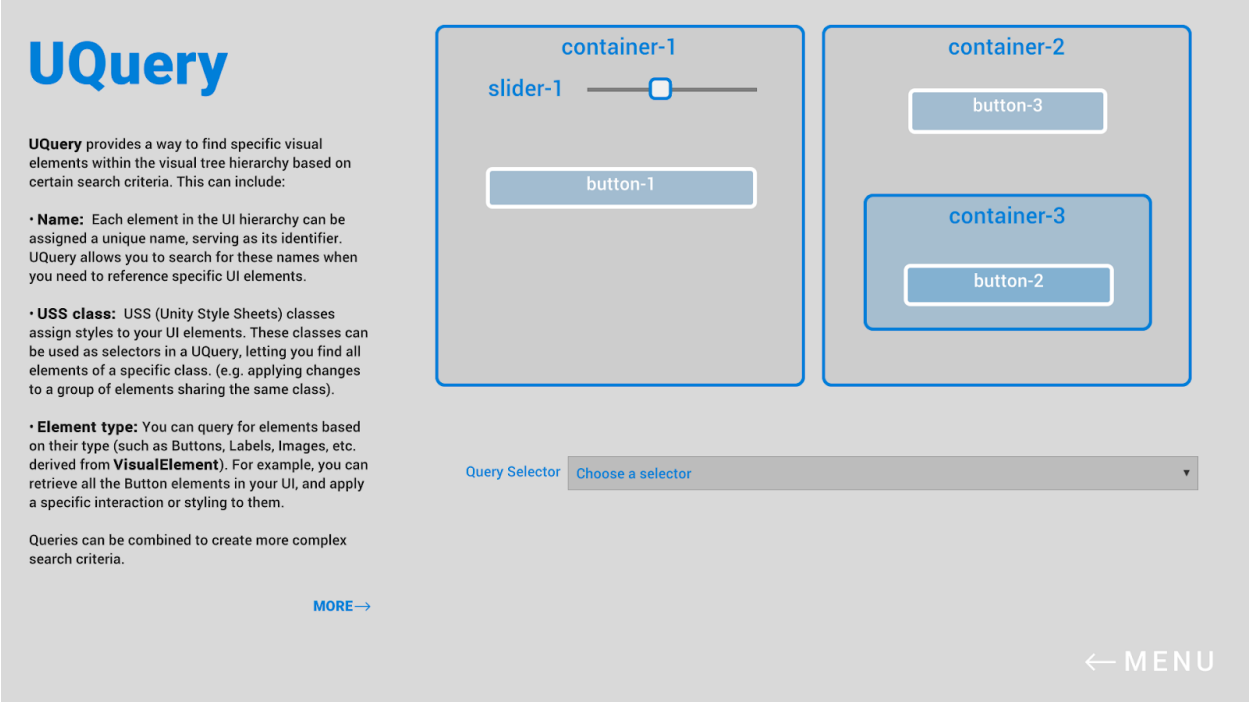


Change an element's appearance through USS styles

UQuery

UQuery is a feature that simplifies finding elements within a complex hierarchy in UI Toolkit.

This demo scene demonstrates its usage, by providing a query selector which highlights the selected UI elements, making it easier to interact with specific UI components in the visual tree.

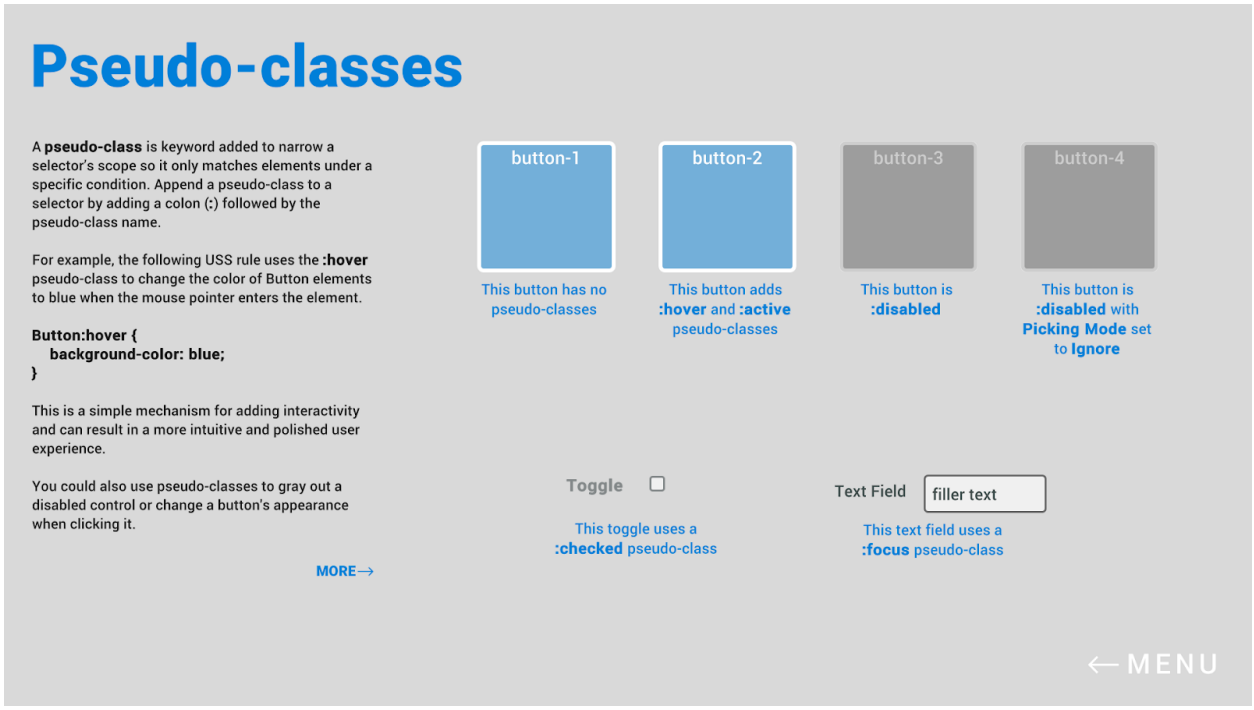


UQuery can make it easy to search through a complex hierarchy.

Pseudo-classes

Pseudo-classes are special selectors that target UI elements based on their state or certain characteristics, such as **:hover**, **:active**, and **:focus**, similar to their function in CSS.

This demo scene shows the basic usage of pseudo-classes for simple animations and interactivity. With a few extra styles and no extra code in most cases, we can make the UI elements respond to the user's input.

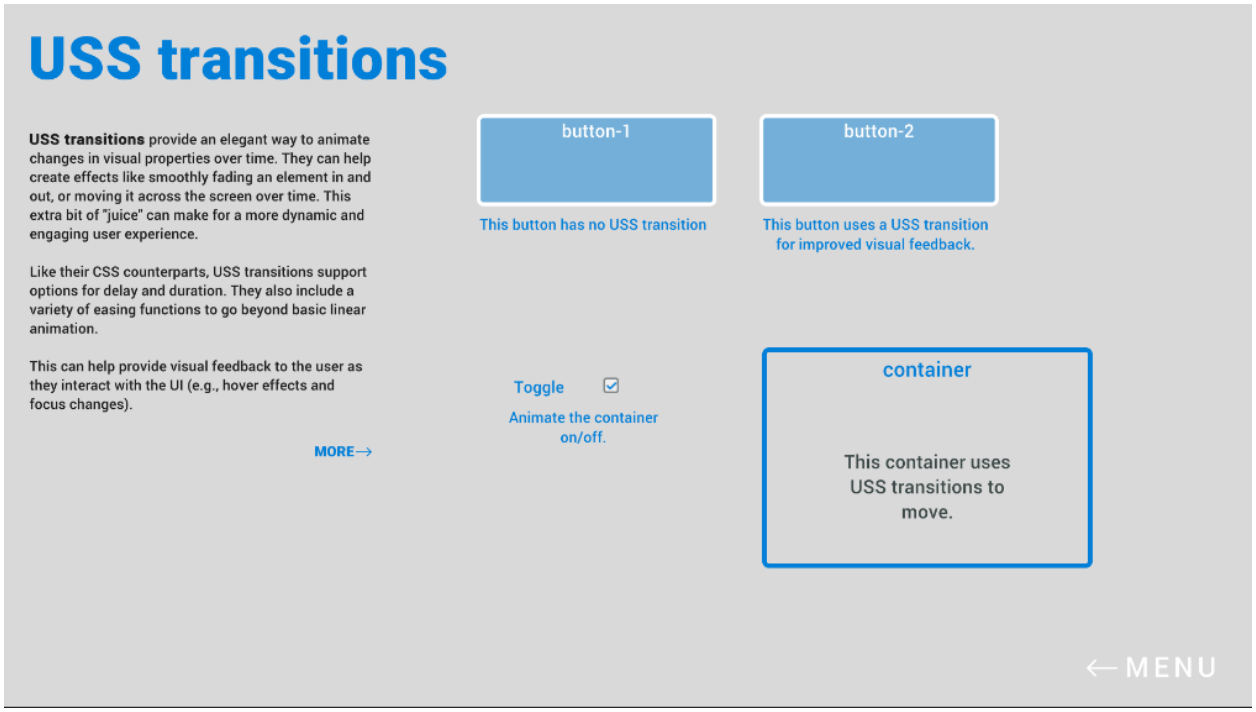


Pseudo-classes target elements based on their state.

USS Transitions

USS transitions allow you to modify UI properties over time, making your UI come alive with simple animations.

The demo provides some simple examples of how to add USS transitions for improving the visual feedback to the user.



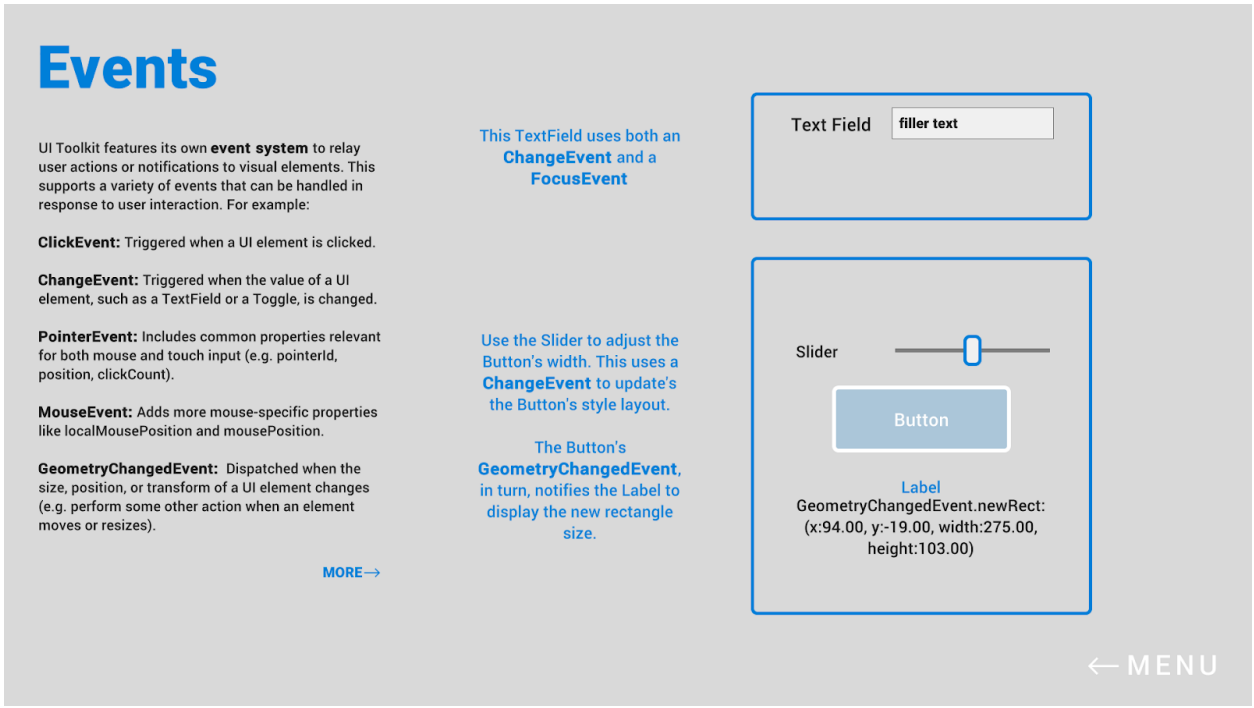
USS transitions provide an easy way to animate your UI for improved visual feedback.

Events

UI Toolkit comes with its own event system, designed to cater to user interface events (e.g. click events, change events, pointer/mouse events, etc.).

Note that this UI Toolkit event system can complement other events in your application, such as the Input System events or System delegates. While the UI Toolkit concentrates on UI-specific actions, the Input System can broadly handle raw device-level inputs. Standard .NET events provide a general way to interact with anything else. You can connect one type of event to another for the desired behavior.

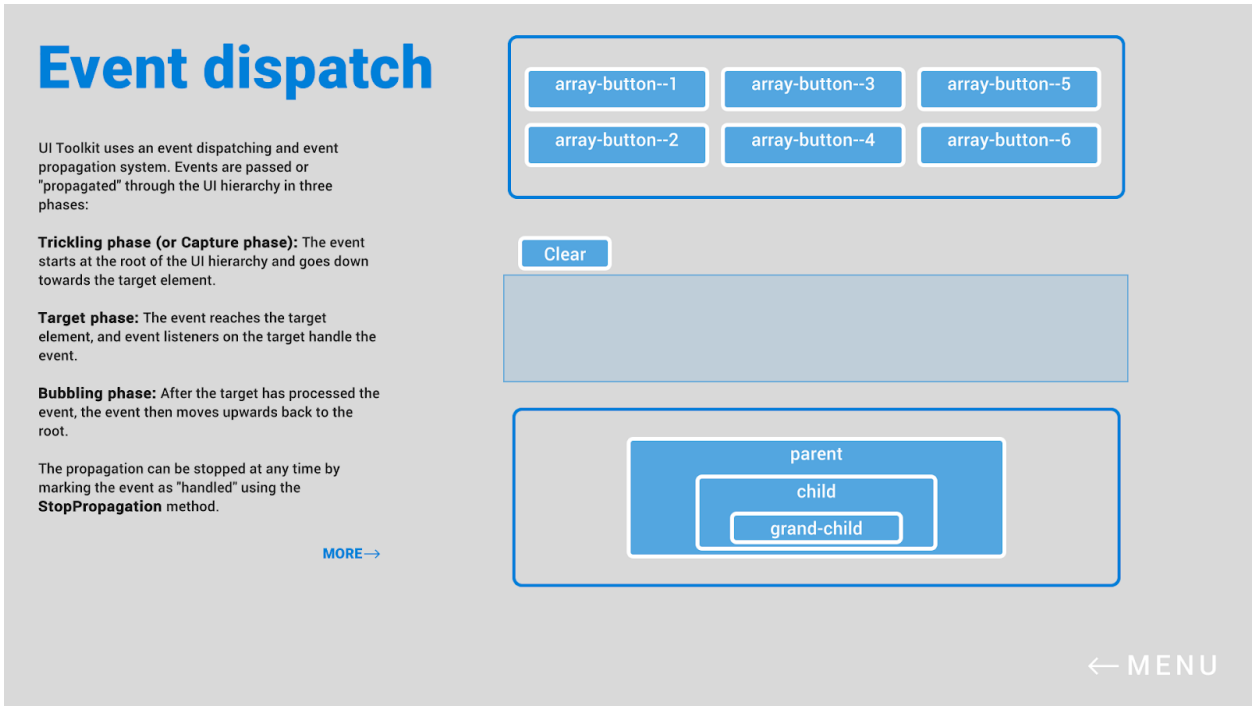
The demo scene shows how separate elements – a Slider, Button, and TextField – can communicate using UI Toolkit events.



Create interactions with the UI Toolkit event system.

Event dispatch and propagation

The UI Toolkit event system operates through a system of event propagation that allows events to traverse the visual tree hierarchy. The demo shows how an event trickles down, reaches its target, and then bubbles up. Use event dispatch to your advantage when setting up a container with several child elements.



Events can propagate through the visual tree.

Manipulators

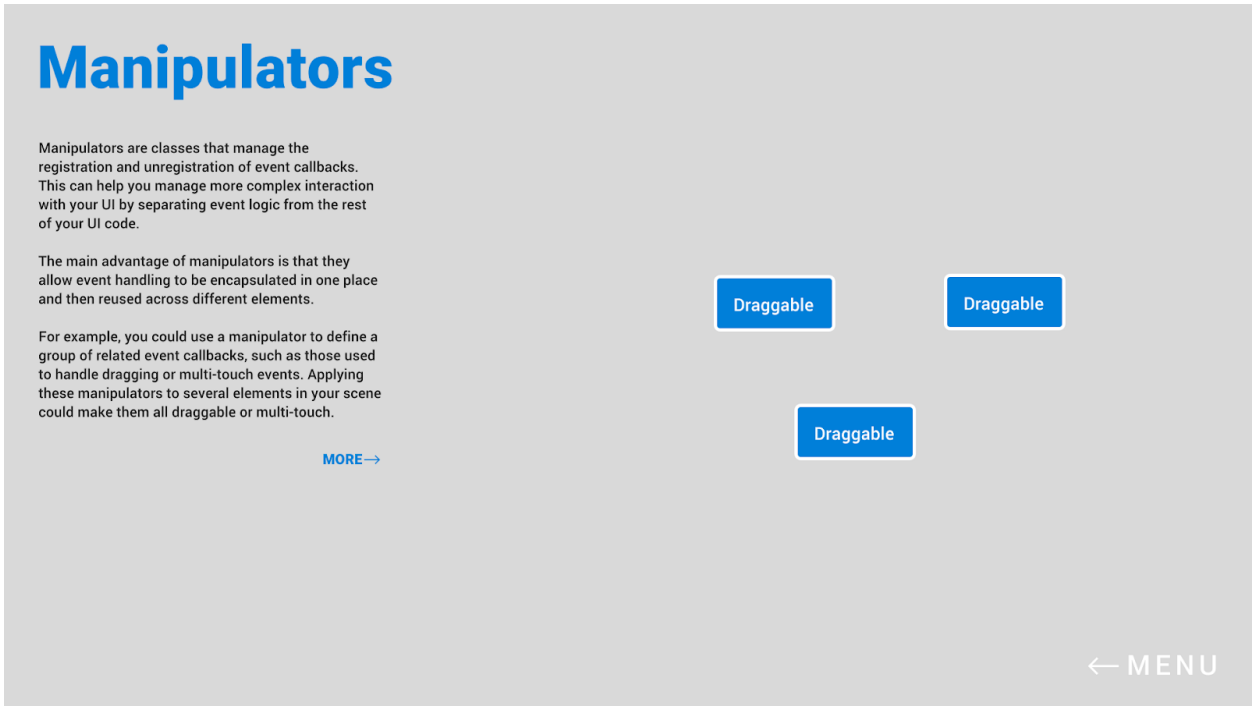
A manipulator is a utility class that can help you handle specific types of user interaction with UI elements.

Imagine click-and-drag interaction or a pinch-to-zoom operation. These can involve several related event callbacks to work properly – the response to the mouse click, the dragging action, and then the mouse release. Manipulators allow you to bundle those event callbacks together for easier management.

Once the manipulator is created, it's reusable and ready to add to your UI elements. Simply attach them to any elements that need the functionality.

In summary, a manipulator essentially streamlines setting up that user interaction, so you don't have to handle each callback one by one.

This demo showcases a custom manipulator that makes an element draggable with the mouse pointer. Making game pieces or floating windows is then just a matter of adding the example manipulator to the parts of your UI that need that specific user interaction.



Manipulators can be used for creating interactive behavior for your UI, such as the draggable elements in this example.

Custom controls

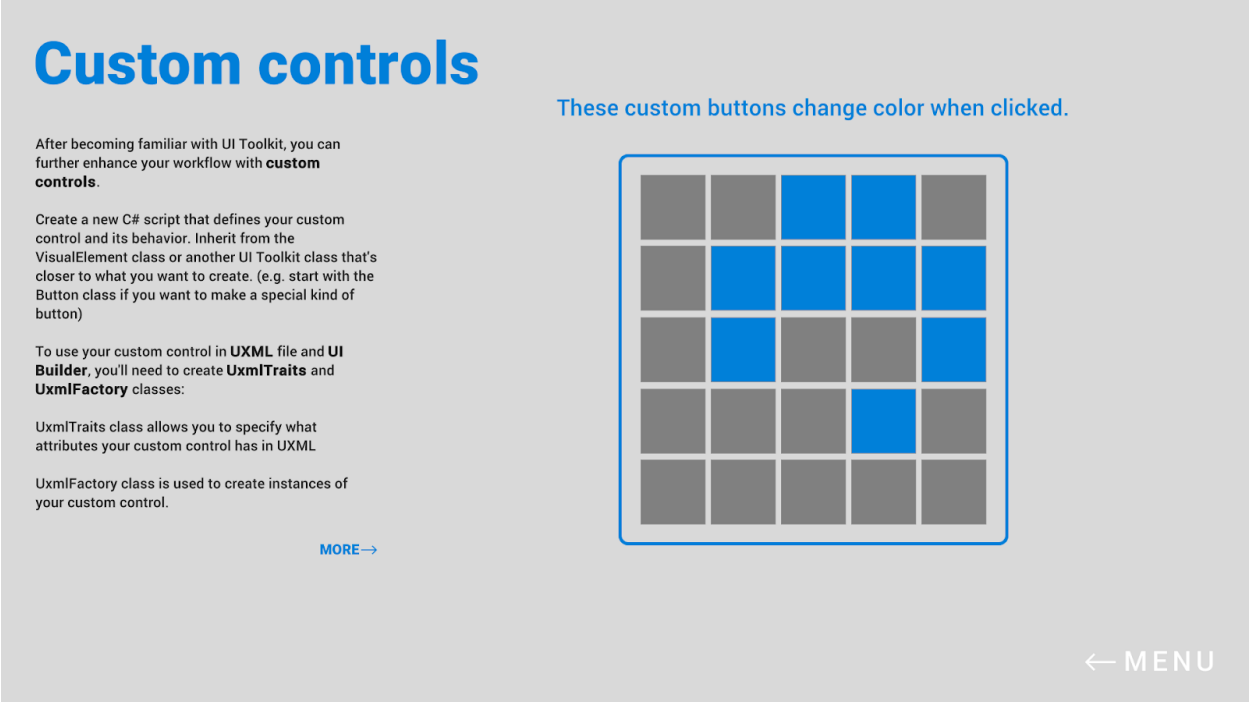
UI Toolkit also allows you to create custom controls that extend the functionality of the base UI Toolkit classes, giving you additional features or behaviors specific to your application's needs.

The UxmlFactory class functions as a link between UXML and your C# code. This factory class allows UI Toolkit to instantiate your custom component when it's encountered in a UXML document. In UI Builder, this allows you to drop these custom controls into your Hierarchy.

The UxmlTraits class defines the properties and fields for your custom element. Custom UxmlTraits appear in the UI Builder's Inspector.

The demo illustrates how UxmlFactory and UxmlTraits classes are used to define and instantiate a custom VisualElement in Unity's UI Toolkit.

The example showcases a custom button that can toggle between two color states.



A custom Button can be reused within the UI Builder.

Continuing with UI Toolkit

As you further your understanding of the UI Toolkit, we hope these demo scenes inspire you with new ideas, or serve as references for specific techniques and patterns.

We also suggest exploring the comprehensive examples in the documentation. These showcase how to assemble elements into functional UIs for both runtime and Editor scripting.

Design patterns

Game programming patterns are reusable solutions to common problems in software development. They can help streamline your code, making it more readable and maintainable.

The QuizU game implements several of these patterns, including:

- The **state pattern** allows an object to alter its behavior when its internal state changes. This pattern can reduce a more complex object into a predetermined set of states. The QuizU project uses a SequenceManager and UIManager that show

different types of state machines.

- The **model-view-presenter (MVP) pattern** maintains a strict separation between the app's data (Model), UI (View), and the controlling intermediary that binds the two (Presenter). This improves readability and maintainability.

- The **observer pattern** can help decouple objects to reduce their interdependencies. This can improve testability and reduce class size. QuizU uses System.Action delegates as well as the UI Toolkit event system.

We'll look in detail at each of these patterns in the posts to follow, so stay tuned.

Learn more in the guide Level up your code with game programming patterns. This e-book provides insights into the SOLID principles and various design patterns, introducing practical examples in Unity.

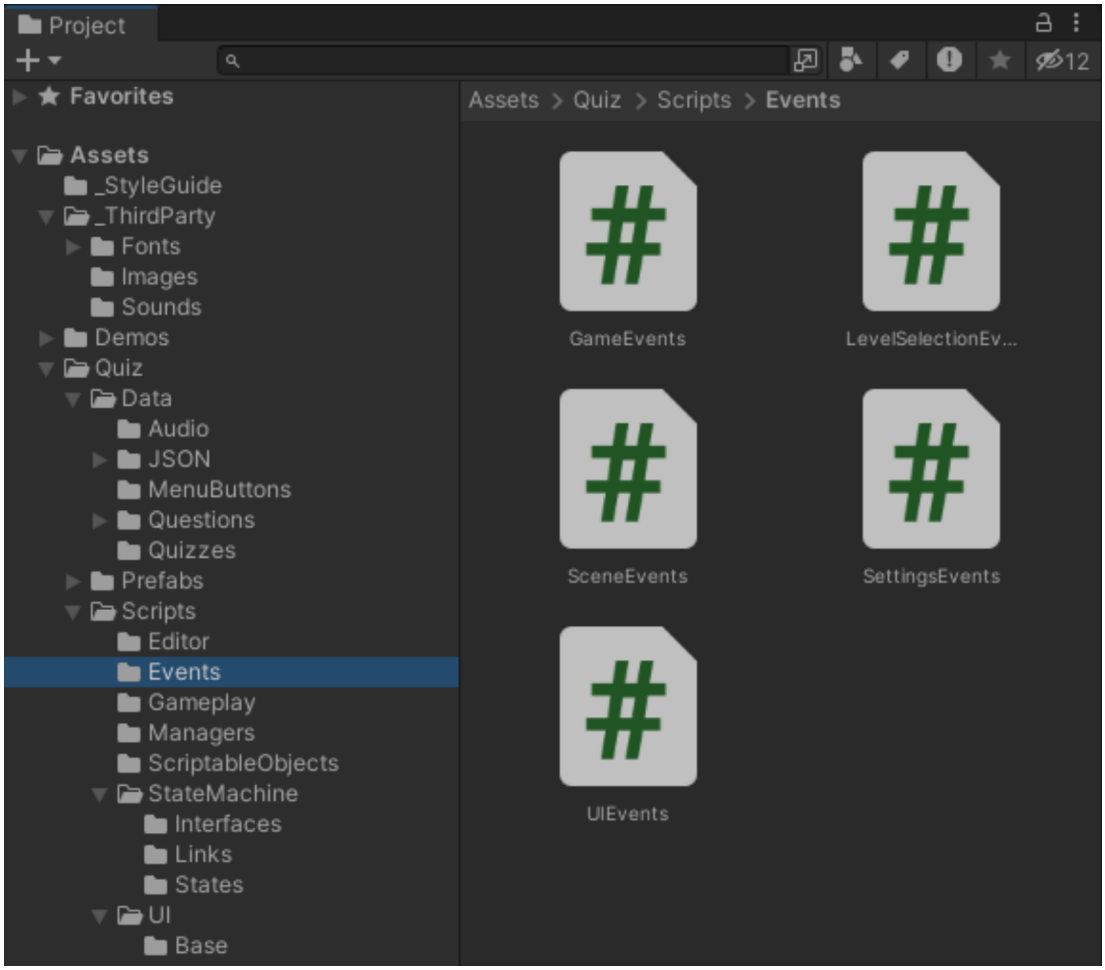
Event-driven development

The observer pattern governs how objects in the sample project communicate. In QuizU, almost everything that "happens" in the application is the result of an event.

Using static events can let objects pass messages without directly referencing each other because they are globally accessible from anywhere in the application, provided they have the appropriate access level. This decoupling makes your components easier to maintain and debug – without affecting the rest of the game's systems. Static events add a layer between the publishing objects that broadcast the signals and the objects which may be listening.

One benefit of event-driven development is that you can reduce reliance on the singleton pattern. Singletons can be convenient in smaller applications due to their global accessibility and persistent states; however, they can cause problems in larger, complex systems. Singletons often produce tightly-coupled code, making it difficult to debug or extend. Plus, the ease of access that singletons offer often leads to misuse or overuse.

Instead, event-driven development uses the publish-subscribe model for messaging. Events can allow for modular components that can be tested more easily.



Different parts of the application use events to communicate. You can find the events in the Scripts/Events folder.

Want to input your quiz answer, play a sound or queue up the next question? You can implement them as events. Invoke messages using the static event class and pass along any required data as a payload. Any object can listen for specific events and execute any handlers in response.

The events in QuizU fall into the following categories, with separate classes for each:

- **Game Events (\Quiz\Scripts\Events\GameEvents.cs):** These events manage the core game flow. They update questions, handle user-selected answers, display feedback, and manage game states (start, pause, abort, win, lose).

- **Level Selection Events (\Quiz\Scripts\Events\LevelSelectionEvents.cs):** These events allow the user to choose a quiz from the Level Selection Screen.

- **Scene Events (\Quiz\Scripts\Events\SceneEvents.cs):** These events manage loading, unloading, and progress updates of different scenes in the game.

- **Settings Events (\Quiz\Scripts\Events\SettingsEvents.cs):** These events handle changes to master, SFX, and music volume.

- **UIToolkit Events (Quiz\Scripts\Events\UIEvents.cs):** UI Toolkit maintains a separate set of events in order to interact with UI elements, such as buttons, sliders, and input fields. They process everything from clicks and drags to value changes, and can be used to trigger updates or transitions in the game.

By dividing the game logic into these different event classes, QuizU can handle complex game state transitions and interactions in a clean and organized way. For example, when a user selects an answer, the AnswerSelected event is triggered in the **GameEvents.cs**. That, in turn, updates the game state and UI in **UIEvents.cs**.

This breaks the application into small “islands of code” that can function more independently from each other. UI, game logic, settings – they all just need to listen for events and react accordingly.

BootloadScreen

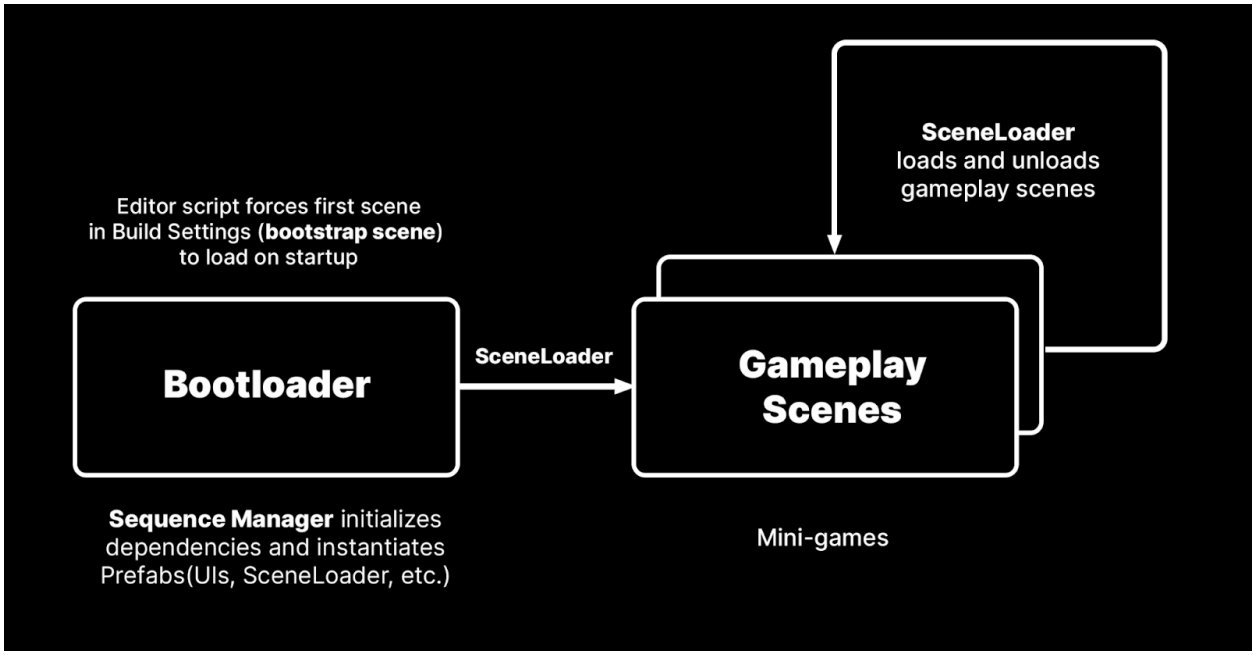
The demo project uses a couple of common techniques to help start the game application in a consistent and predictable state. One of these is a **Scene Bootstrapper** (or bootloader) which is an editor extension script responsible for setting up the game's initial state. This initialization code is separate from the game logic and ensures that all dependencies are set up correctly for the objects in the scene.

The bootstrapper configures essential GameObjects, managers, or services when a scene is loaded to avoid dependency issues.

If your Unity application spans several scenes, the bootloader can force loading a specific bootstrap scene, which is the first scene from the Build Settings. When exiting Play mode, the Editor reloads the previous scene.

Another component in the bootstrap scene, the **Sequence Manager**, can then instantiate essential prefabs on scene load. In this specific demo project, everything needed to create the game is a prefab, including a camera, SplashScreen, UI menus, and a SceneLoader.

The **SceneLoader** then additively loads (and unloads) any gameplay scenes as needed. In most cases, these scenes are composed of prefabs.



The Bootloader initializes dependencies to control flow of project scenes.

Project scenes

Each mini-game level is a separate Unity scene and appears in the Build Settings. Disable the SceneBootstrapper in the GameSystems menu if you want to explore those individual scenes.

Many projects also include a staging area for the main menu after the bootstrap scene. This demo project omits a main menu scene.

Other features

C# Style Guide

The **StyleExample.cs** file in **_StyleGuide** folder is a project-specific style guide for C# naming conventions, formatting rules, and usage guidelines, most of which are applied in QuizU. This promotes readability and consistency in coding, especially when working in a team environment.

While there is no “right style” other than what works best for you and your team, the style we are following here is inspired by general industry standards for C#. In our demo we

decided to make a few tweaks from the original guide as an example. You can learn more by checking out our e-book Create a C# code style guide for more information about style guides.

Utilities

The project includes some helpful smaller scripts not specific to the quiz application which can be found in the **\Quiz\Scripts\Utilities** folder:

- **EventRegistry:** This utility class manages the registration and unregistration of UI Toolkit events. Because so much of the game flow hinges on setting up events, the EventRegistry makes that process easier. See Event handling in UI Toolkit for more information.
- **Coroutines**: This helper class provides static methods for managing coroutines. This can help a System.Object or ScriptableObject run a coroutine on a separate MonoBehaviour.
- **DestroyOnLoad:** Apply this to temporary objects to remove them during the scene load process.
- **NullRefChecker:** If you are using the Inspector to set required dependencies, use this to validate your fields. Bypass this behavior using the custom **Optional** attribute.
- **Tooltip:** This shows how to use a UI Toolkit Manipulator to make a simple tooltip that stays within the boundaries of the screen.

ScriptTemplates

The **ScriptTemplates** folder includes two files to set up formatting, namespaces, and base classes for new MonoBehaviours and ScriptableObjects. Similar to the utilities, they are not specific to the demo and you can modify or create additional templates as needed.

ScriptableObjects

Finally, we use ScriptableObjects as data containers to help us manage information such as questions, quizzes, and menu data. Using ScriptableObjects, we can easily separate data from logic which simplifies data management for more streamlined development.

For more about ScriptableObjects, see the e-book Create modular game architecture with ScriptableObjects from our best practices guides.

QuizU: State pattern for game flow

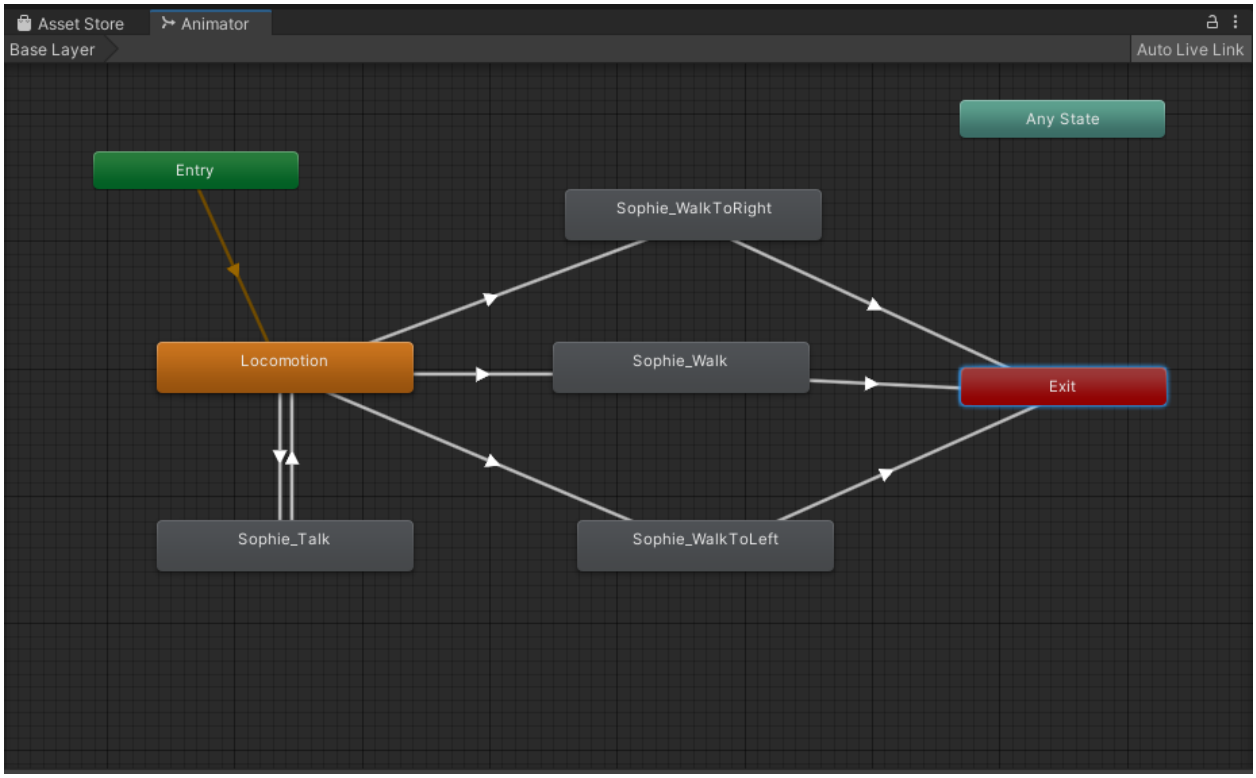
The state pattern is a software design pattern that can represent complex behaviors as a simpler set of internal states. Each state encapsulates the behavior and allowable actions within that state.

As an example, consider a 3D animated character in Unity. Its character rig might run, jump, attack, and idle, using a hierarchy of animated transforms. Though it's possible, working directly with the many different individual parts of a character is difficult. The sheer number of objects is unwieldy and can be hard to manage.

Instead, often you'll use an AnimatorController. This allows you to wrangle those hundreds of moving parts into clear and concise states: Jumping, running, attacking, idling, etc.

The AnimatorController is functioning as a state machine. It tracks the character's current state and manages transitions to other states. It does this based on predefined rules and triggers. As the developer, you only need to manage the logic of transitions between states, and then the state machine takes care of the rest.

You can visually represent a state machine as a graph, much like an AnimatorController:



The AnimationController is an example of a state machine.

This is the essence of the state pattern: It distills a complex system into manageable states.

The principle of state machines can be applied to numerous other aspects of your application. UI screens, AI behavior, level or scene management, etc. are potential candidates for using them.

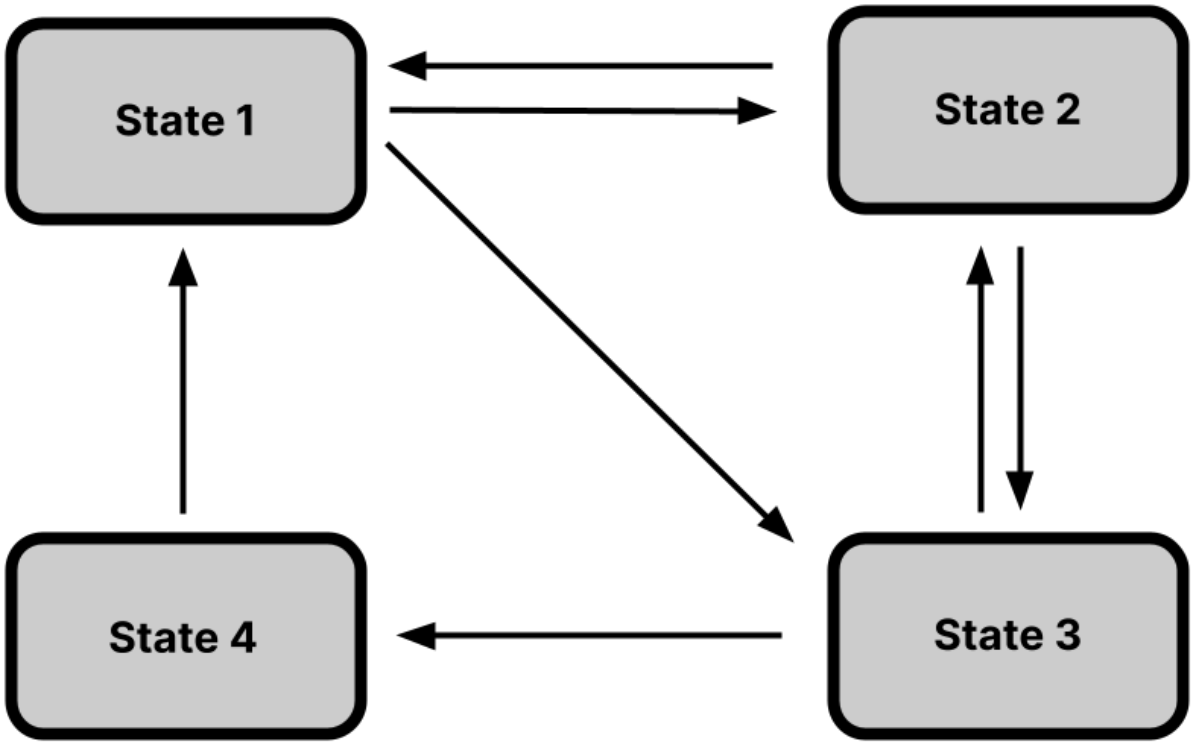
One common use for the state pattern is to describe the main game loop and game states. Let's walk through how to set up a state machine for the general game flow of the QuizU application.

Check out the Level up your code with game programming patterns e-book for an introduction to the state pattern.

Components of the state machine

You can think of a state machine as a map of possibilities. Like our AnimatorController, it can be represented visually as a graph, in which each state corresponds to a single node.

Only one state, the current one, can be active at a time. This current state switches between nodes in this graph, based on specific conditions or events. Imagine a choose-your-own-adventure book, where the current page (the state) changes depending on the choices (the conditions or events) you make.



Visualize a state machine as a graph.

The state machine is a collection of these states. Each state, in turn, can connect to one or more other states. These "links" evaluate conditions or events to determine how the transition happens. The arrows also illustrate how the pattern allows you to define varying rules for what state can transition to other states.

The IState interface

Rather than use a series of if-else conditions to change states, the original Gang of Four state pattern suggests that you define each state independently. When adding new states, you don't want to affect the existing ones.

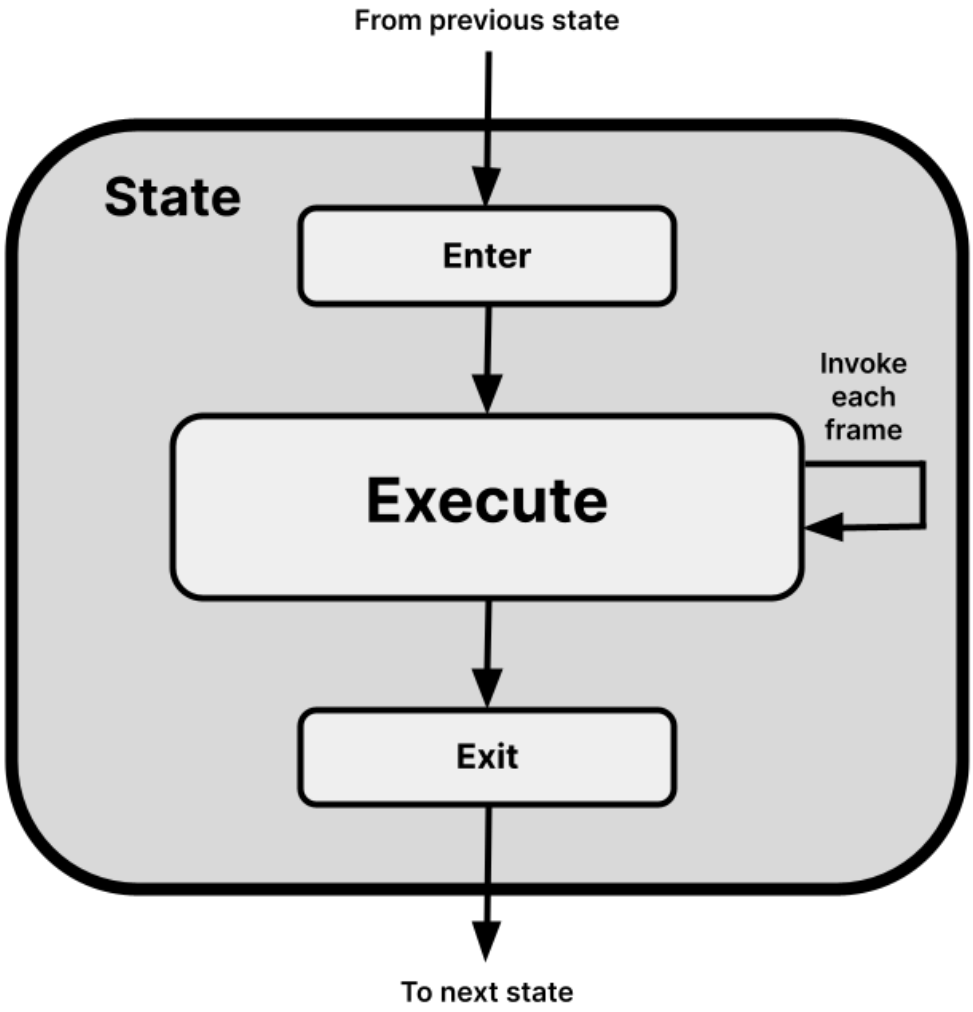
So, keep each state in its own object. Think of it as one "mode" of behavior.

The pattern itself doesn't care about what that behavior is. A state could handle UI changes, animation playback, or anything else – it's just a matter of how you want to apply it.

The only requirement is that each state needs some means of transitioning to another state when the time is right. What each state does is up the state itself. This approach makes everything more flexible.

In keeping with the idea of composition over inheritance, we'll define an interface, **IState (\Quiz\Scripts\StateMachine\Interfaces\IState.cs)**, that represents each state's barebones functionality. Interfaces allow you to define clear "contracts" between the states and the context (or state machine) that uses them. It ensures every state has the necessary methods that the context might call.

Each state encapsulates its behavior in its own script, making the code modular and easier to understand and maintain.



Each state in the state pattern.

The IState interface requires the following methods:

- **Enter:** This method is called when a GameObject first enters a particular state. Use this for setup, initializing variables, etc.
- **Execute:** This coroutine defines the main actions or behavior the GameObject should perform every frame in this state and is started after Enter().
- **Exit:** This method is called when the object leaves this state. It's useful for cleanup, resetting variables, etc.

- **ValidateLinks:** The owner state-machine calls this method to examine all the links of the state and determines if it should transit to the next state. It does so by determining if any conditions to transition to another state have been met. If not, the state machine stays in the current state.

- **AddLink** and **RemoveLink:** These methods will define how to set up transitions from this state to another one.

- **EnableLinks** and **DisableLinks**: These methods make all the links from this state to others active or inactive.

To follow along in the project go to the folder **\Quiz\Scripts\StateMachine \IState.cs**.

The ILink interface

To check on the conditions for exiting the state, we define another interface, the **ILink.**

```
public interface ILink
{
   bool Validate(out IState nextState);

    void Enable(){}

    void Disable(){}
}
```

This contains a **Validate** method to determine whether conditions are right to transition to another state. If the exit check passes, Validate returns true and provides the next state we should transition to. Otherwise, it returns false.

ILink also has methods to **Enable** and **Disable** its links.

Remember that the concrete states (below) and their links will actually contain the implementation logic. The interfaces are simply functional blueprints.

The StateMachine class

The **StateMachine (\Quiz\Scripts\StateMachine.cs)** class is the “brain” that processes how the states and their links work together.

It contains a **CurrentState** property to track its one active **IState**. This is the current behavior of the object. The logic within the CurrentState's ILinks determines when to shift to the next state.

In essence, the StateMachine is just a playback controller. It only knows that it contains a series of states and understands they are connected using links. It doesn't understand the details of how each state works or how their transitions happen. Everything executes depending on the current state's internal logic.

This pattern allows you to create as many states as you need. Define a state, add transitions, and the state machine is expanded.

The StateMachine includes these methods:

- **SetCurrentState**: This function changes the current state, CurrentState, to a new one. It also stops any previously running state.

- **Run:** This function turns on the main loop of the StateMachine. It starts the state, lets it run, and then waits until it's complete.

- **Stop:** This function stops the state machine from processing and clears the current state.

- **Loop**: This is an update loop that runs continuously. For every frame, the state machine checks the status of the current state. If the conditions are right, it finishes anything left to do in the current state and transitions to the next active state.

- **Skip:** This function can immediately stop a state that's currently running.

Concrete States

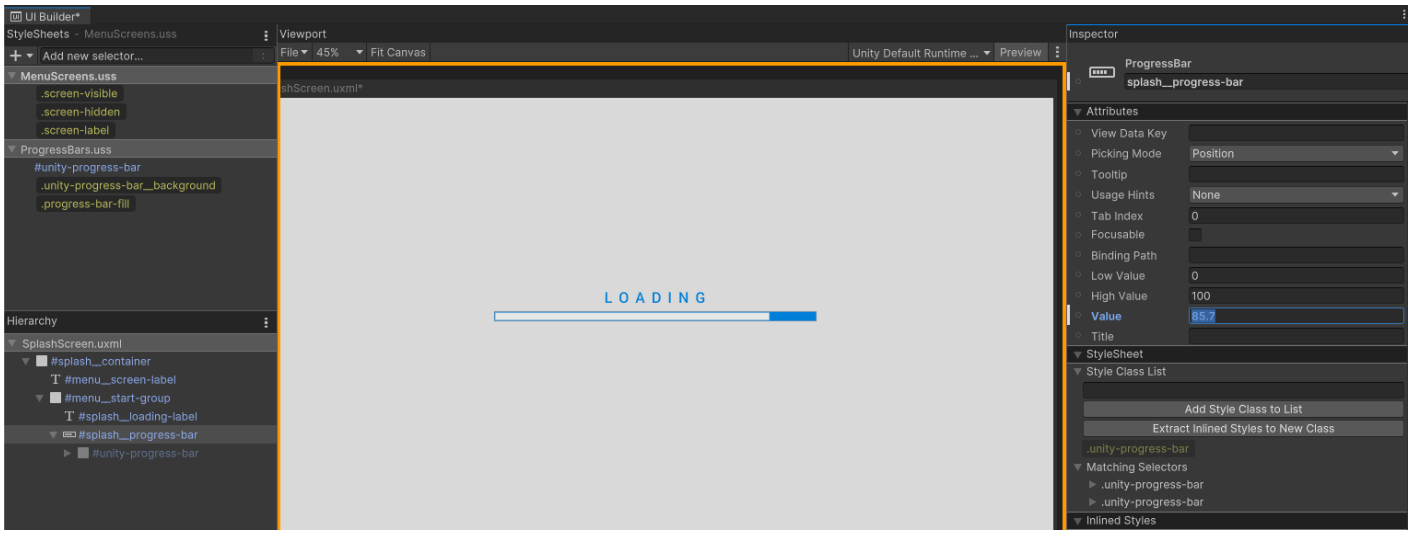
Of course, interfaces are just empty blueprints. To bring our state machine to life, we need *concrete states* that do something.

First, we define an **AbstractState (\Quiz\Scripts\StateMachine\States\ AbstractState.cs)** class to serve as a base class; this ensures all of the states share consistent implementation (e.g., the links all work the same way). The rest of our concrete classes can then derive from AbstractClass.

QuizU includes two example states (both available in **Quiz\Script\StateMachine\States**):

- The **State** class executes a specified Action just once when entering the state. This can be useful for anything, from setting a variable to playing back a sound or triggering some action.

- The **DelayState** class pauses the state machine for a given duration before it executes. A specified `Action<float>` can then run every frame. This works for tasks that loop continuously, such as animating a loading bar on the Splash Screen. An optional Action can also run on exit.



The DelayState updates the Splash Screen loading bar.

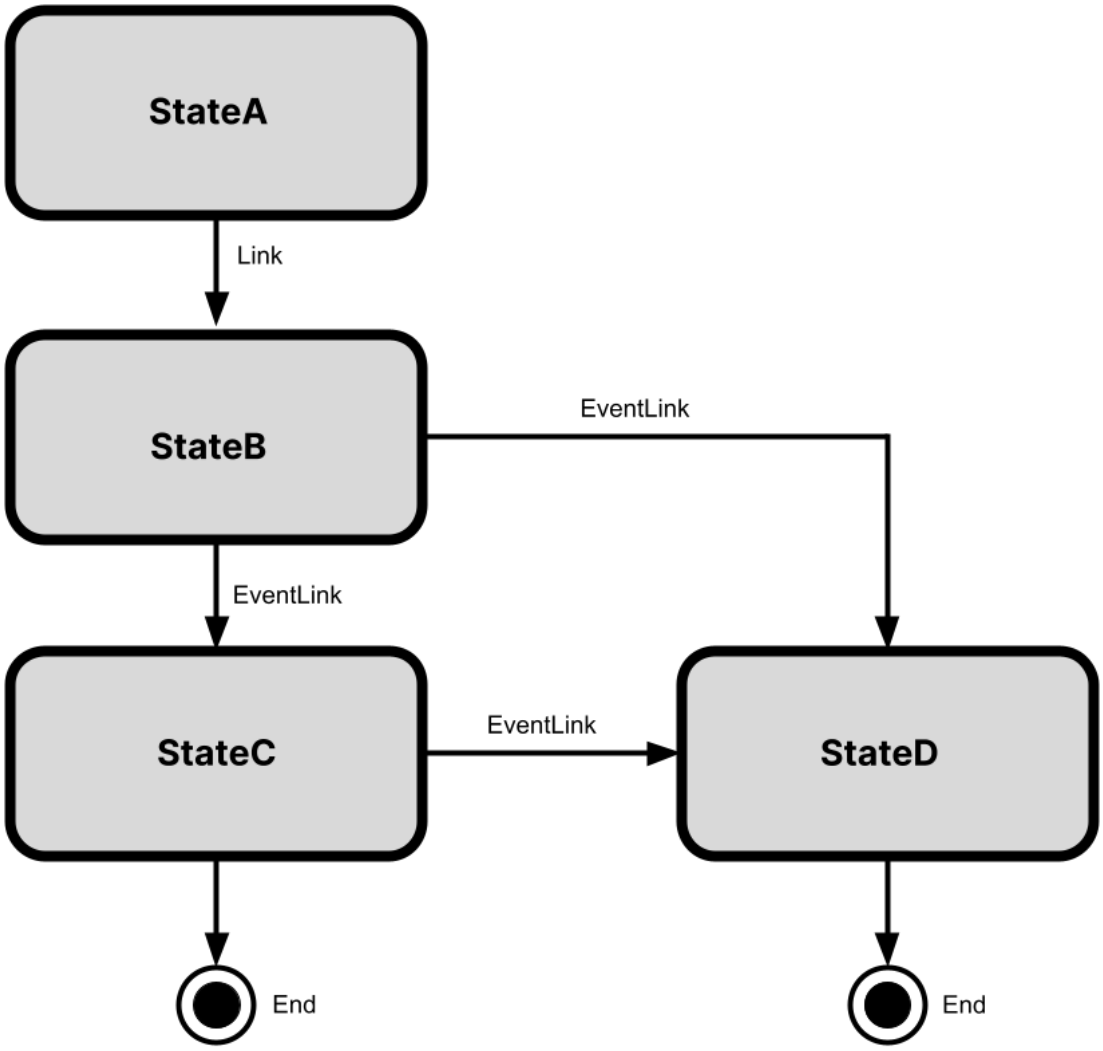
Transition links

The concrete classes will also require some corresponding ILinks (both available in Quiz\Script\StateMachine\Links\):

- The **Link** class, the simplest implementation of the ILink interface, transitions to another predefined state once the current state finishes execution.

- The **EventLink** class, on the other hand, waits for an event. The state machine remains in its current state until that event is raised. This is useful, for example, if the transition depends on user input.

The transition links can now connect concrete states together into a basic graph. This is everything that we need to build game flow. This diagram shows an example how a few states might connect with each other.



The Sequence Manager

Once we define some basic concrete states and links, we can create an instance of the state machine anywhere we need it.

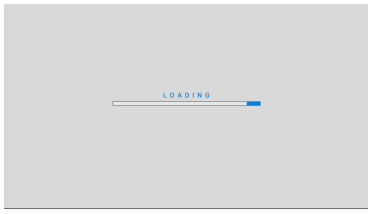
One place we can deploy a state machine is inside the **Sequence Manager (\Quiz\Scripts\Managers\SequenceManager.cs)**. In QuizU, this is a MonoBehaviour that tracks the overall game flow. We can formalize the main game loop as a series of IState objects.

Game states

Remember that the pattern's goal is to reduce something complex down to a few management states.

Implementing the state pattern early on helps to maintain a clean and organized project. As your game evolves from a small demo and increases in complexity, you can manage that new complexity with additional states.

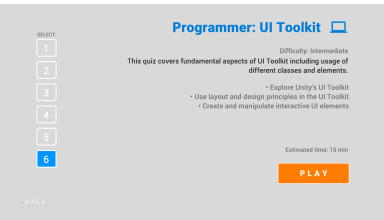
Each state is its own object. Our sample project breaks the application's game flow into the following:

- **SplashScreenState:** This is the initial startup state which loads necessary assets (e.g. textures, 3D models, sounds, shaders, etc.) and displays a loading progress bar.
- 

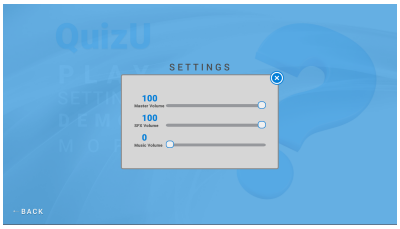
- **StartScreenState:** This is a placeholder state after the Splash Screen commonly used in games. In QuizU, we show the game logo until the user presses the start button.

- **MainMenuState:** This state shows the main menu, where users can select different options, such as starting a game, choosing a level, or changing settings.

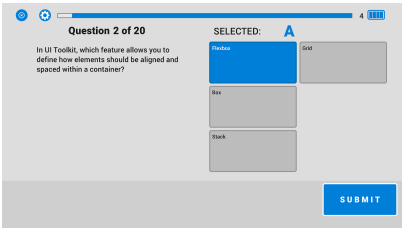
- **LevelSelectionState:** This state shows a UI to select a game level. QuizU has a number of different quizzes, but you could expand this to include difficulty settings, load game options, etc.



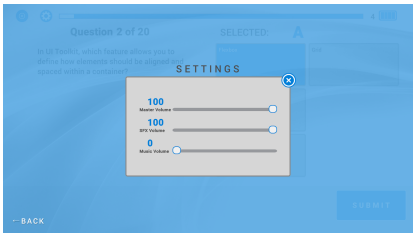
- **MenuSettingsState:** This state lets users change or customize application/game settings.



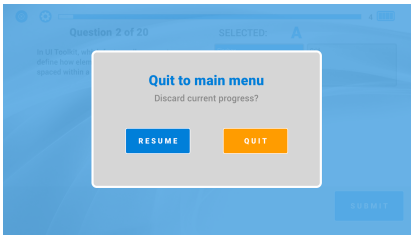
- **GamePlayState:** This is the core state where the quiz gameplay takes place. This state is active when the user is playing the game.



- **GameSettingsState**: Similar to MenuSettingsState, this one is accessible during gameplay. This screen allows players to adjust game settings without leaving the game. In QuizU the options are limited to audio volume settings, but this could be expanded to your needs.



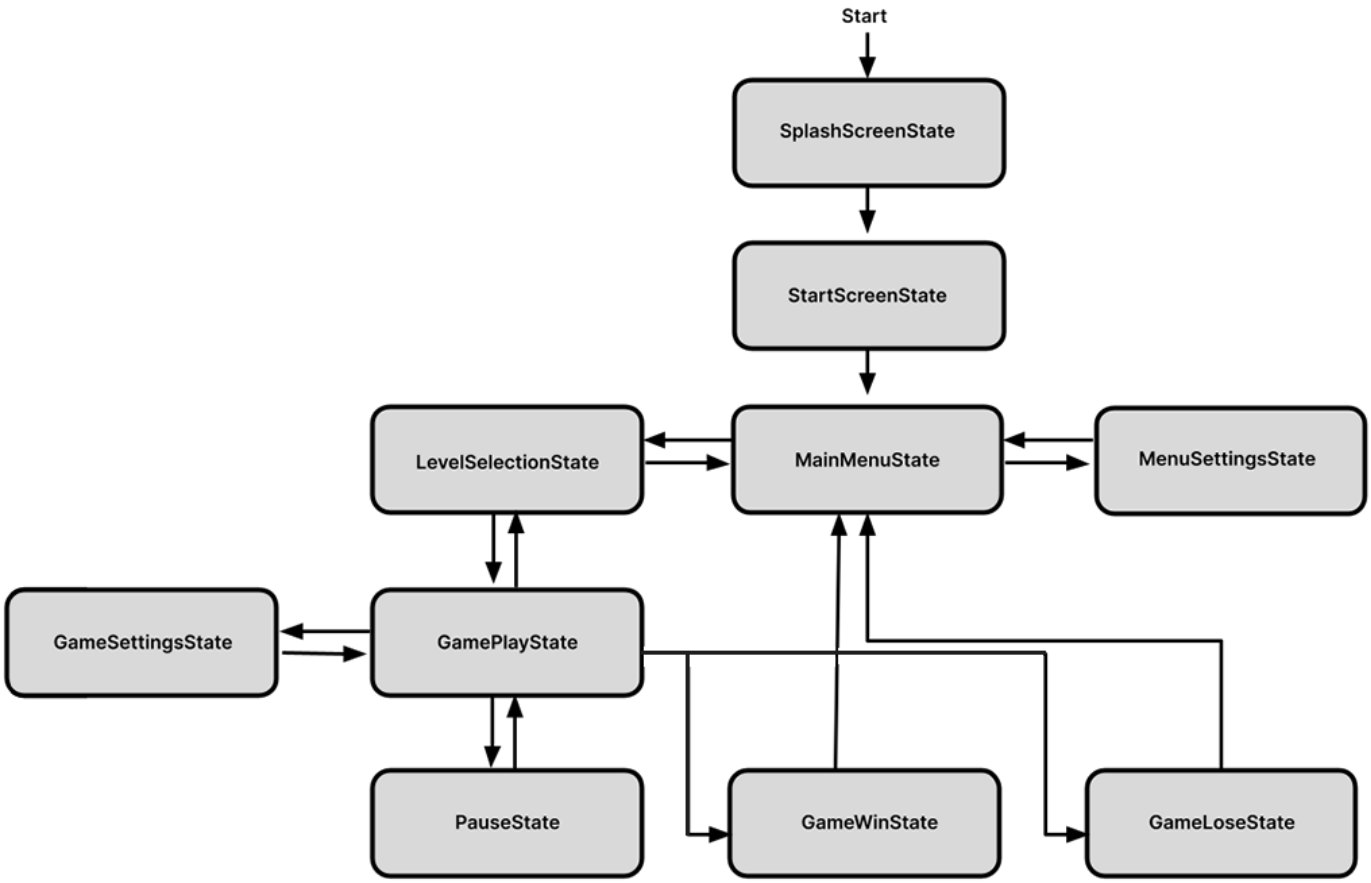
- **PauseState:** This state pauses the game during gameplay. It shows a Pause Screen where the user can continue gameplay or return to the menus.



- **GameWinState:** This state is shown when the player wins the game. It displays a "Win" screen with relevant information and options for replaying the game or returning to the main menu.

- **GameLoseState:** Similar to the GameWinState, it shows a "Lose" screen with relevant options.

Though we define these in code, a visual representation of the states might look like this:



Visualizing the game states as a graph

Each state represents one behavior of the game application at runtime. The SequenceManager transitions from one state to another, based on certain events or conditions.

State machine setup

In the **SequenceManager** (QuizU\Assets\Quiz\Scripts\Managers\SequenceManager.cs), fields are defined for the state machine and each IState.

- Set m_StateMachine to a new empty instance.
- Create a new field for each unique state with type IState.

This shows part of the SequenceManager in the QuizU project:

```
public class SequenceManager : MonoBehaviour
{
    …

    StateMachine m_StateMachine = new StateMachine();

    IState m_SplashScreenState;
    IState m_StartScreenState;
    IState m_MainMenuState;

    …

    private void SetStates()
    {

        m_SplashScreenState = new DelayState(m_LoadScreenTime,
SceneEvents.LoadProgressUpdated,
            SceneEvents.PreloadCompleted, "LoadScreenState");

        m_StartScreenState = new State(null, "StartScreenState");
        m_MainMenuState = new State(null, "MainMenuState" );
    }
}
```

Then in the SetStates method, use the constructor to create a new State or DelayState for each. In the above example:

- **m_SplashScreenState** is a **DelayState**. It raises the **SceneEvents.LoadProgressUpdated** event every frame for a delay of **m_LoadScreenTime** seconds.

  Once the delay is over, the state machine triggers the **SceneEvents.PreloadCompleted** event.

- **m_StartScreenState** and **m_MainMenuState** both implement the State concrete class. Essentially, these are placeholder states. We pass in null as the default action, which means the states don’t do anything specifically, even when they are active.

In Start, we initialize the SequenceManager and call the SetStates.

```
 // Define the state machine's states
private void SetStates()
{
            // Executes GameEvents.LoadProgressUpdated every frame and
GameEvents.PreloadCompleted on exit
```

```
            m_SplashScreenState = new DelayState(m_LoadScreenTime,
SceneEvents.LoadProgressUpdated,
                SceneEvents.PreloadCompleted, "LoadScreenState");

            m_StartScreenState = new State(null, "StartScreenState");
            m_MainMenuState = new State(null, "MainMenuState");
…

}
```

Note that even if a state doesn't perform any additional actions, it still has utility. For example, imagine if a UI element or visual effect needs to work differently during the StartScreen state or the MainMenu state. It can read the CurrentState of the SequenceManager to know when to change behaviors.

Adding link transitions

Each state includes an **AddLink** method to set what specific conditions or events trigger the state change.

Here's a snippet of the SequenceManager's **AddLinks** method where we set up those individual calls to AddLink:

```
public class SequenceManager : MonoBehaviour
{
    …

    private void AddLinks()
    {
        m_SplashScreenState.AddLink(new Link(m_StartScreenState));

        m_StartScreenState.AddLink(new EventLink(UIEvents.MainMenuShown,
m_MainMenuState));

        …
    }
}
```

These lines set up a transition from the SplashScreen State to the StartScreen State automatically and a transition to the MainMenu State once the UIEvents.MainMenuShown event is raised.

As you add more states and more link transitions, define them here.

Benefits of a state machine

Using a state machine has several advantages:

- **Scalability:** As a game grows more complex, the number of states can increase without impacting the existing code. This adheres to the Open/Closed Principle (OCP) which states that a class should be open for extension but closed for modification. Managing these states using a state machine in separate scripts can be easier than having one large monolithic script full of if-then or switch statements.

  Remember that in a state machine, each state is a standalone entity. Changing one state does not affect another.

- **Maintainability:** Each state handles its own logic with an explicit structure. This makes the code cleaner and more readable. Multiple team members can work on different states simultaneously without stepping on each other.

- **Reduced Complexity:** A state describes its own conditions for transitioning to the next state. The logic for each state is self-contained and doesn't spill out to other states. This keeps the classes short and easy to understand.

  Contrast this to modifying if-then statements where a change in one condition can have cascading effects down the chain.

In a small application like QuizU, the benefits of using a state machine may not be apparent. The purpose of this demo, however, is to show how to use such programming design patterns in game UI in a simple, pared-back project so the methods of implementation are really clear to you. This is something to keep in mind as you explore the project: It’s ultimately meant to be instructional and educational.

As an application grows, however, imagine how you can:

- **Poll for the current state:** The SequenceManager keeps the CurrentState as a public property. Anything that needs to listen for state changes can reference this and then respond.

- **Debug better:** Having the current state can help pinpoint issues or troubleshoot unexpected behavior.

- **Manage transitions:** Use state changes to notify other objects when moving between different scenes or levels in a game.

- **Pause the game:** Imagine creating a global pause state that doesn't need to set the TimeScale to 0. By referencing the SequenceManager's state machine, anything "pausable" can listen to the current state of the game.

- **Handle events:** The AbstractState includes an Enter, Exit, and Execute method. Use these to run a single event when starting or ending the state, or execute some logic every frame.

  Events can do anything that you need them to do, e.g. play a sound or a visual effect. You're only limited by your creativity.

Further reading

The specific implementation of QuizU's state machine is adapted from the Runner Template project. This is a feature-complete state machine ready to be used in your own project. See the original Runner Template (available in the Unity Hub) for more example states and link transitions.

If you're interested in more software design patterns, see the free e-book Level up your code with game programming patterns. Or, check out these articles on design patterns:

- Object pooling
- The state pattern
- The factory pattern
- The observer pattern
- The MVP/MVC pattern
- The command pattern

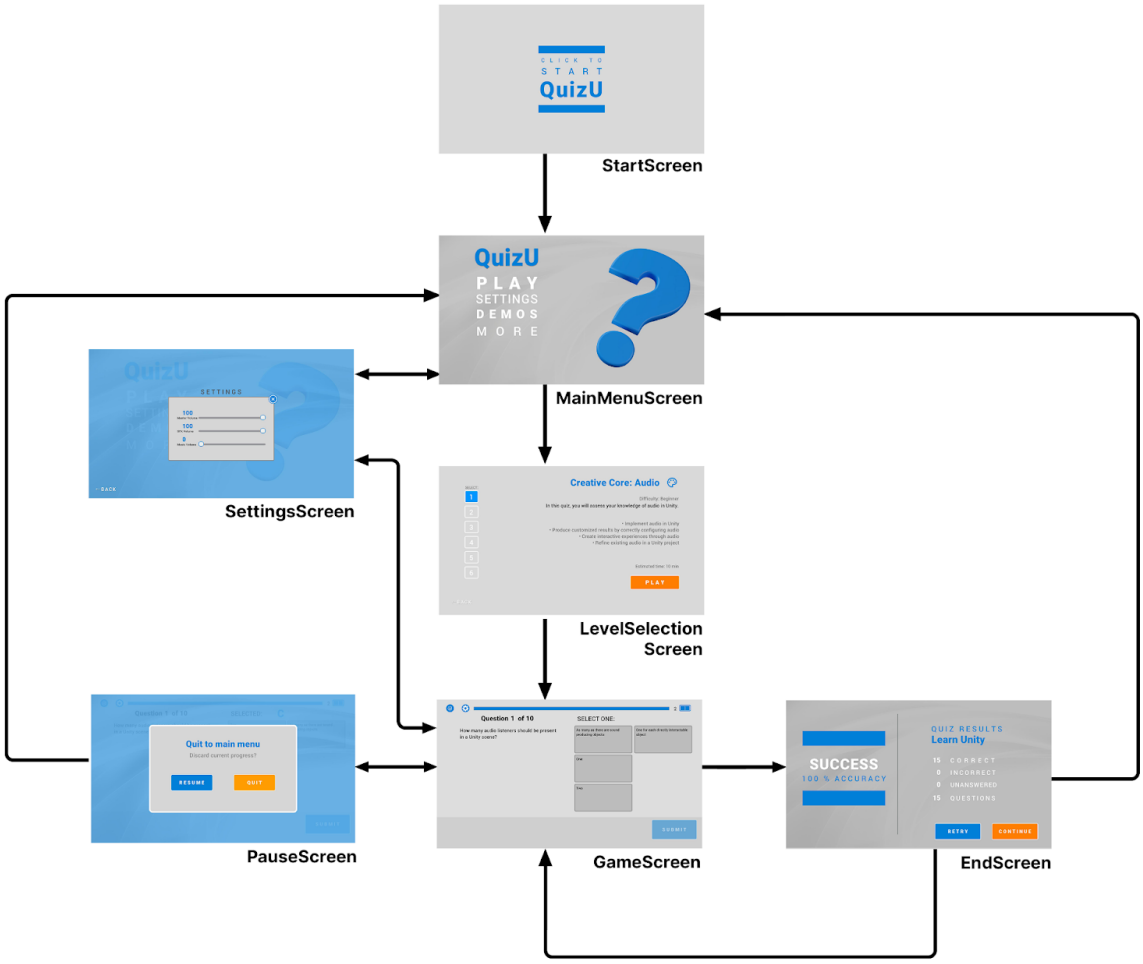
You can also find more advanced best practices guides in the documentation.

Managing menu screens in UI Toolkit

The QuizU mini-game uses seven main screens. These typify what you might find in many applications:

- **Start Screen:** This is the first screen, typically featuring the game's logo/ branding and a "start" or "play" button.
- **Main Menu Screen:** From this screen the player can navigate to various parts of the game such as the levels, settings, or other menus.
- **Settings Screen:** This is where the player can adjust game preferences, like audio settings.
- **Level Selection Screen:** This screen allows the player to choose the game level or stage they wish to play.
- **Game Screen:** This is where the main gameplay happens. In the context of the QuizU game, this is where questions are presented, and answers are collected.
- **Pause Screen:** This screen is displayed when the game is temporarily halted, providing options to quit or resume the game.
- **EndScreen:** This win/lose screen displays the player's score or performance when the game is complete while showing options to replay or return to the main menu.

These screens connect to form the UI navigation flow:



The UI navigation flow connects each UI screen.

Setting up UXMLs

In Unity's UI Toolkit, UXML files are akin to blueprints for UI structures. They define hierarchies of UI elements, creating what's known as a VisualTreeAsset. Each of these UXML files can be reused, nested, and combined with other UXMLs to construct complex and interactive UIs.

The workflow in UI Toolkit differs from the traditional UGUI approach. Instead of populating the Hierarchy with multiple UI objects, you'll primarily work with UI elements through UXML files – either through UI Builder or directly from scripts.

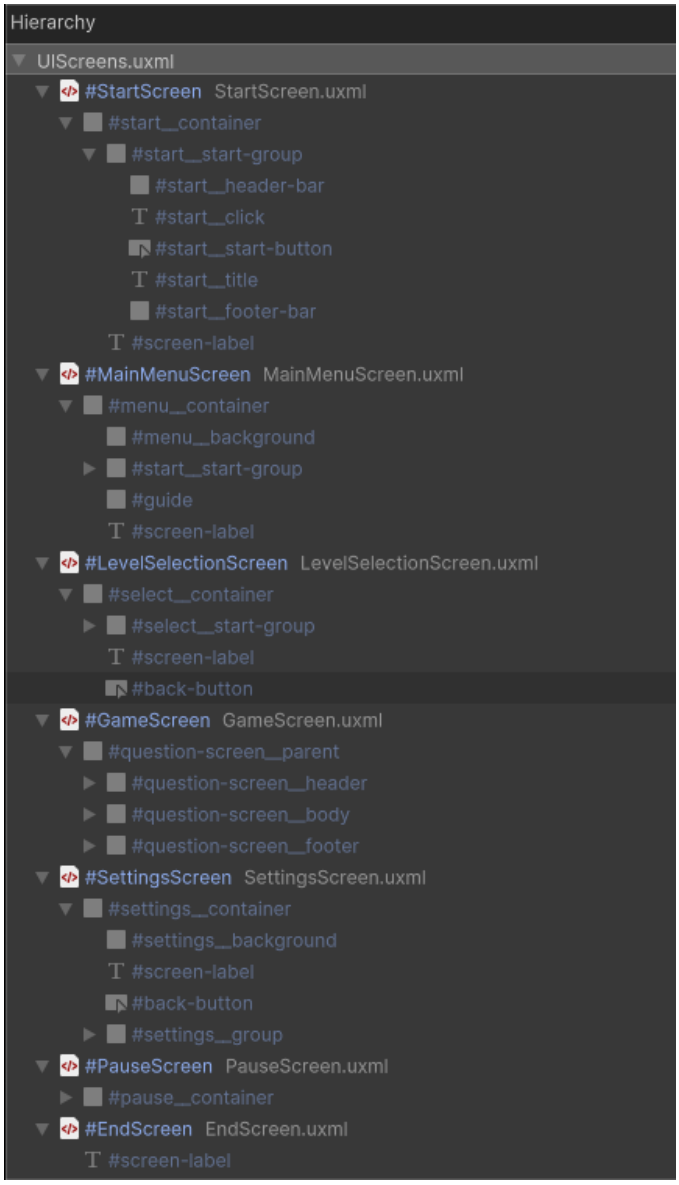
Ultimately, you only need a single UI Document and Panel asset to render the UI on the screen. The custom UI management will then display part of the visual tree as needed.

Within a project, each screen can have its own UXML file which holds the layout and arrangement for that particular screen. This allows each UI designer or developer to work on individual UXMLs, reducing merge conflicts with teammates.

In the QuizU sample project, these separate UXMLs are assembled into one master file, the **UIScreens.uxml**, which is then referenced inside the UI Document. Every screen's visual tree is designed to occupy the entire space of the parent container (**positioning: absolute, width: 100%, height: 100%**), ensuring each interface is independent of the others.

This workflow can benefit larger teams. Everyone can be responsible for a smaller, self-contained portion of the UI which can later be added to a unified UI Document. In this methodology, UXMLs are analogous to nested Prefabs, but without the actual GameObjects.

Look through the UIScreens.uxml in the QuizU sample to see how it's assembled.



Assemble the UXMLs for use in one UI Document.

Screen stack navigation

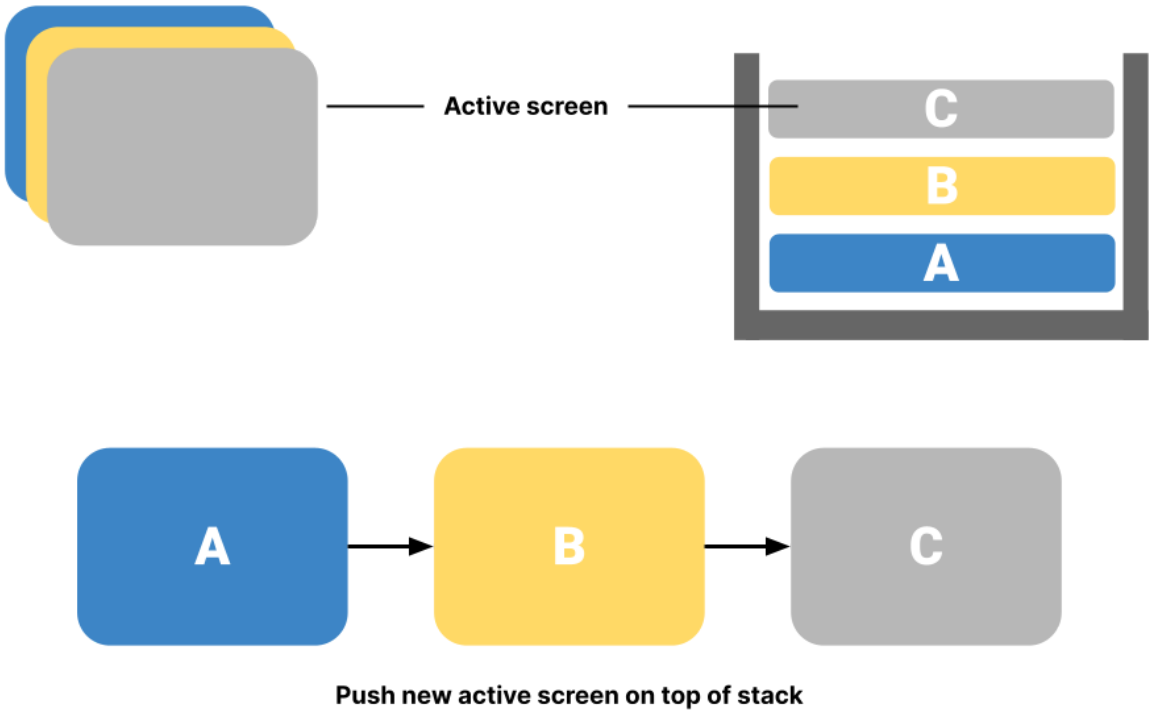
The QuizU sample uses a pattern called the “screen stack” or “stack-based state machine” for turning UIs on and off. The idea is to manage screens by stacking them on top of each other.

By keeping your UI screens in a stack, this pattern provides a built-in mechanism to

maintain user navigation history, similar to a web browser's back button.

This Last-In-First-Out (LIFO) style of navigation stores each fullscreen UI as a layer, with only the top layer active at a time. This combines the idea of using a stack data structure with a finite-state machine (FSM).

Though it shares similarities with the setup from State patterns for game flow, it is more simplified for this use case. Here, each screen functions as one state of the FSM. The state machine can only be in one state at a time, known as the *current* or *active state*.



Managing UIs with Stack-based State Machine or Screen Stack.

Each screen, or state, represents one "unit" of the UI, essentially everything visible on the screen at one given time. As these units become active, they are pushed onto the stack. When they are no longer needed, they are popped off the stack. The topmost screen on the stack is always the active state. Transitions between states occur in response to certain conditions.

Opening a new screen pushes a new state onto the stack, and that becomes the active state. Closing a screen or navigating back pops the top state from the stack, making the previous state the active one.

For instance, selecting ‘Play’ from a main menu screen pushes the gameplay screen onto the stack. Opening a pause menu from the game then adds the pause screen to the

stack. Resuming the game pops the pause screen off the stack, returning control to the gameplay screen.

While this pattern is fairly widespread, its implementations can vary. To illustrate one basic approach using UI Toolkit, we've created some sample classes, the **UIScreen** and **UIManager**. Both scripts are available in the (**QuizU\Assets\Quiz\Scripts\UI**) folder.

UIManager versus SequenceManager

In the sample project, both the UI Manager and the Sequence Manager operate at the same time and have some overlapping features. This particular application is UI dependent, and many events, like button clicks, can affect both managers.

Here, the UI Manager handles tasks related to the UI, using the UI Toolkit. On the other hand, the Sequence Manager oversees the general flow of the entire application.

When you start combining UI elements with GameObjects in your game, having these two separate managers will become more logical, as they'll work together to handle different aspects of the game.

UI Screen

The **UIScreen** class works in conjunction with the **UI Manager** to form the UI logic of the QuizU user interface.

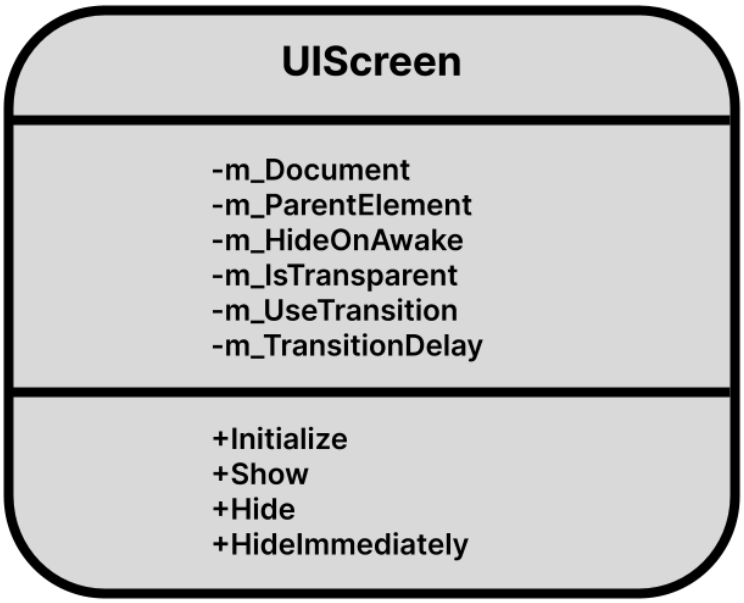
UIScreen is an abstract base class that provides the framework for creating one functional “unit” of the user interface. Therefore, the QuizU project includes classes like **GameScreen**, **StartScreen**, **PauseScreen**, etc., all of which derive from **UIScreen**.

The base UIScreen class includes a few methods:

- **Initialize:** This performs some basic setup, such as querying a UIDocument to find the topmost element within the Visual Tree Asset.
- **Show:** This method shows the UI element with a transition if enabled.

- **Hide:** This method hides the UI element with a transition if enabled.
- **HideImmediately:** This method hides the UI element without a transition.

Essentially UIScreens are UI objects that can show or display themselves.



Each UI Screen can show or hide itself.

Note how UIScreen also provides a number of properties and settings, such as:

- **m_HideOnAwake:** Whether the UI screen should be hidden when the game starts
- **m_IsTransparent:** Whether the UI screen should be partially see-through
- **m_UseTransition:** Whether the UI screen should use a transition when hiding or showing itself
- **m_TransitionDelay:** The amount of time to wait before showing the UI screen after it is initialized

Also, take these into consideration when working with UIScreens:

- Use the **m_ParentElement** field to access the topmost element of your UI screen's hierarchy. This can be useful for UQuery operations. For larger UI hierarchies, searching from a smaller branch of the visual tree can be faster than querying from

the `rootVisualElement`.

- Enable **m_UseTransition** to fade the UIScreen on or off using USS style classes. Use the style's transition duration and the **m_TransitionDelay** field to adjust the timing. When the transition finishes, a TransitionEndEvent then turns the parent element off entirely. This adds a small visual polish versus toggling the screen on or off abruptly.

- The **EventRegistry** is an optional helper class that uses the IDiposable pattern to manage the registration and unregistration of event callbacks – much like how you would subscribe or unsubscribe to System.Actions or UnityEvents. While the garbage collector usually takes care of removing the callback with its associated VisualElement, using the EventRegistry can help in certain situations.

- In this demo project, each UI screen has one corresponding UXML file with all the visual elements. Then, they combine into one UIScreens.uxml. You can nest even more visual trees if you require additional complexity.

Once we have a UI Screen that can turn on and off, we'll need another class, the UIManager, to maintain the screen stack.

Optimization tip

To reduce overhead, the UIScreen derives from a System.Object instead of a MonoBehaviour. Though they lose the ability to set fields in the Inspector, they can reference project assets through **Resources.Load** or **Addressables**. Also, you can initialize specific fields in the UIScreen constructor.

UIManager

The **UIManager** class is responsible for turning UI screens on and off. It uses a stack to manage all of the UIScreens under its control, with only one screen active at a time.

A stack works somewhat similar to Lists but uses a Last-In-First-Out (LIFO) data structure. This means that the last element added is the first one to be removed, similar

to stacking a deck of cards. This is useful for situations where you need to reverse the order of elements or maintain a specific order of actions, such as going back in your navigation. Here are a few takeaways from the QuizU implementation:

- The **m_History** stack starts with the Main Menu Screen, which functions as the “home screen.” Showing a UIScreen means pushing onto this LIFO stack. The top screen is always visible and active.

  This history stack maintains a collection of previously shown screens, allowing the system to "go back" until it reaches the home screen.

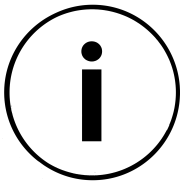
- UIScreens can be partially transparent or see-through. This provides an overlay effect, where the top screen's see-through areas reveal elements of the screens below. This allows screens underneath to be inactive but visible.

- Game events are tied to each screen. UIManager registers event listeners in the OnEnable method. For instance, the **UIEvents.MainMenuShown** event activates the **UIEvents_MainMenuShown** method, showing the main menu screen.

- UIScreen instances are stored in a master list using Reflection. This can help hide or show all of the UIs at once.

- The **UIManager** unregisters events in the OnDisable method. This prevents errors from dangling event listeners if the UIManager instance is deactivated or destroyed.

Used together, the **UIScreen** and **UIManager** form the “screen stack” or “stack-based screen” pattern often used in game development.



Adapting UI design patterns

Consider the screen stack design presented here as one approach for how you might work with UI Toolkit. Evaluate your application's needs and adapt it accordingly.

Some projects may benefit more from a "tab-based" or "drawer-based" navigation scheme. You can combine these with the screen stack design or use a different UI pattern altogether.

Keep in mind that each choice in software design comes with tradeoffs. For example, our stack uses a set of pre-instantiated UXMLs combined into a single file; essentially this acts as a custom-created pool where we deactivate screens not currently in use. If you have a large number of UIs, you may need to balance this with memory usage. It may be more efficient to instantiate the VisualTreeAssets at runtime.

UI Toolkit gives you a lot of flexibility here. Weigh the pros and cons of each design pattern and then decide as a team how to proceed.

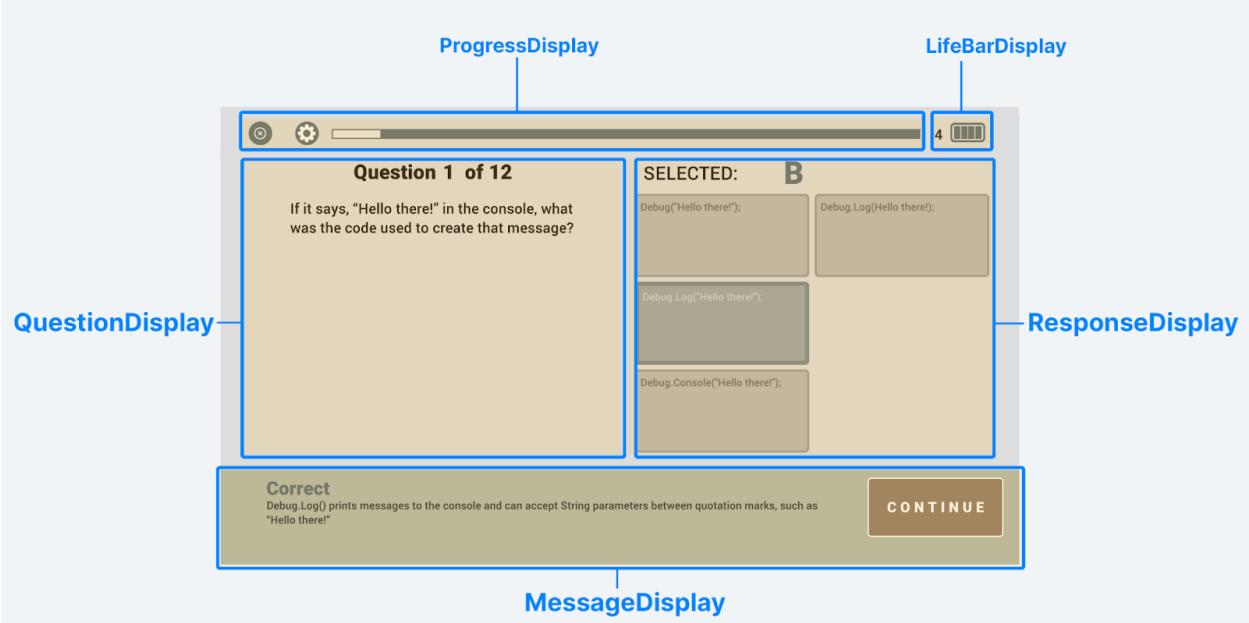
GameScreen example: Controlling other UIs

If you're building a game that requires more complex UI Screens, it can be beneficial to divide them into smaller, more manageable components. The **GameScreen.cs** file in the QuizU sample shows one example of how to break the UI logic into smaller scripts.

Though the UI consists of one UXML file, the C# logic spans several UI classes:

- **GameScreen:** This is the controller for several other UI displays.
- **QuestionDisplay:** This UI displays and formats the current question.
- **ResponseDisplay:** This displays the user's response to a question.
- **MessageDisplay:** This message bar at the bottom of the screen gives feedback to the user for correct and incorrect answers.
- **ProgressDisplay:** This progress bar at the top of the screen represents the user's progress through the quiz.
- **LifeBarDisplay:** This icon shows the remaining lives or guesses the user has left.

This organizational scheme divides the GameScreen like so:



The GameScreen encompasses several smaller classes.

The GameScreen class functions loosely as the controller for its smaller constituent displays. It's responsible for creating and disposing of the smaller displays under its management:

```
public class GameScreen : UIScreen
{
    ResponseDisplay m_ResponseDisplay;
    QuestionDisplay m_QuestionDisplay;
    MessageDisplay m_MessageDisplay;
    ProgressDisplay m_ProgressDisplay;
    LifeBarDisplay m_LifeBarDisplay;

    public GameScreen(VisualElement rootElement): base(rootElement)
    {
        m_ResponseDisplay = new ResponseDisplay(rootElement);
        m_QuestionDisplay = new QuestionDisplay(rootElement);
        m_MessageDisplay = new MessageDisplay(rootElement);
        m_ProgressDisplay = new ProgressDisplay(rootElement);
        m_LifeBarDisplay = new LifeBarDisplay(rootElement);
        m_LifeBarDisplay.AssignTooltip("Guesses remaining");
    }

    public override void Disable()
    {
        base.Disable();

        m_ResponseDisplay.Dispose();
        m_QuestionDisplay.Dispose();
```

```
        m_MessageDisplay.Dispose();
        m_ProgressDisplay.Dispose();
        m_LifeBarDisplay.Dispose();
    }
}
```

Note the following about this implementation:

- The **GameScreen** class does not inherit from MonoBehaviour, reducing any unnecessary overhead. This foregoes built-in life cycle events (like OnEnable, Awake, and Start) for initialization in lieu of using a constructor.

- Each of the smaller displays implements the `IDisposable` interface. This allows the GameScreen to dispose of them at once when invoking the `Disable` method (see Event Handling in UI Toolkit for more information about using an EventRegistry).

- In keeping with the base UIScreen class, the GameScreen constructor uses the topmost VisualElement, `rootElement`, to initialize the sub-displays.

- The smaller displays exercise a great deal of autonomy from the GameScreen itself. The GameScreen will destroy and dispose of them in Disable but otherwise, the m_ResponseDisplay, m_QuestionDisplay, et al. function as in independent UIs. They communicate with the rest of the interface via events.

Of course, implementations can vary depending on your needs. Your version might use only some of these techniques. For example, if you need the GameScreen logic most closely connected to the individual displays, each Display could reference the GameScreen more directly (or use local events).

Further reading

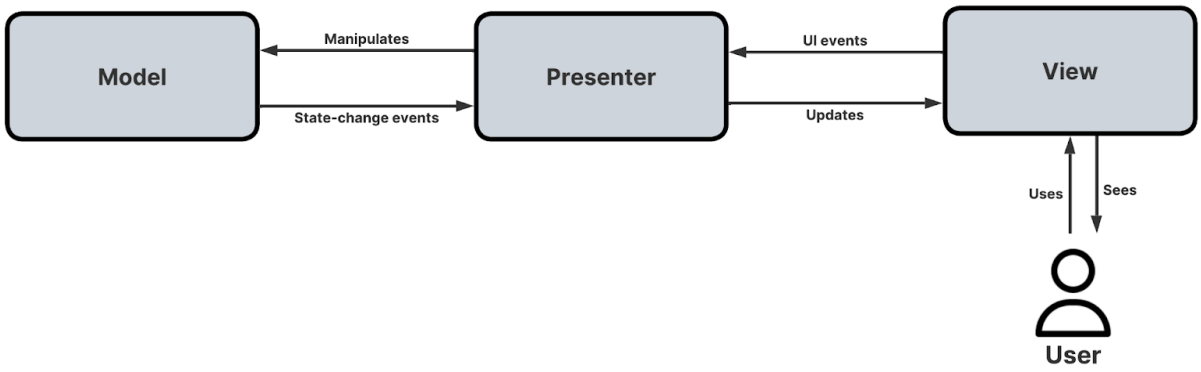
We hope that the QuizU project can help you get started in UI Toolkit to create UI systems for your Unity projects. Remember that you can always find more support in the Forums and documentation.

If you're interested in more software design patterns, please make sure to see the free e-book Level up your code with game programming patterns. You can also find our other best practices guides in the documentation.

QuizU: Model View Presenter pattern

The Model-View-Presenter (MVP) design pattern is an architectural pattern that helps us maintain separation of concerns, something that's especially helpful as we add features to the project.

MVP divides our code into three distinct parts:



Visualize the MVP pattern.

- **Model:** This contains the data and the rules that govern this data. For example, this could include the game's current state, game attributes, or logic regarding that data (e.g. rules for leveling up a character, health, or scoring). The model has no knowledge of the View.

- **View:** This is the user interface of your application. It displays the data to the user and sends user interactions (like button clicks) to the Presenter. The View in the MVP pattern does not directly interact with the Model.

- **Presenter:** This script sits between the Model and the View. It handles user input events from the View, updates the Model as conditions change in the game, and updates the View to reflect those changes.

Essentially, the View, or user interface, does not directly connect with the data that it represents. It instead relies on the Presenter to tell it what to do. Though we might have a separate script for UI logic, the View doesn't handle any "business logic" for our game. That's safely tucked away in a separate silo.

These separate layers of software are useful for:

- **Scalability**: Having distinct parts of the software that handle specific tasks makes it easier to manage and grow your codebase.

- **Modularity**: With MVP, changes in one component don't ripple through the entire system. Each component can be developed, tested, and modified independently. This makes your code easier to debug.

- **Reusability and Maintainability:** Parts of your code (like Models or Presenters) can often be reused across different parts of your application. If there's a problem, it's quicker to isolate and fix bugs in a specific layer than having to sift through code spaghetti.

If you ever need to rewrite the UI – let's say you want to update your UGUI interface to UI Toolkit – MVP makes that process easier. Changing any one of the three parts of MVP has minimal impact on the others. This is also a big plus when working on a large team where the work is divided among several developers.

MVP is a variation of the MVC (Model-View-Controller) family of software patterns. Both are patterns for structuring code into three parts, but in the MVC, a Controller (instead of a Presenter) handles user input and manipulates the Model. The Model can then directly update the View.

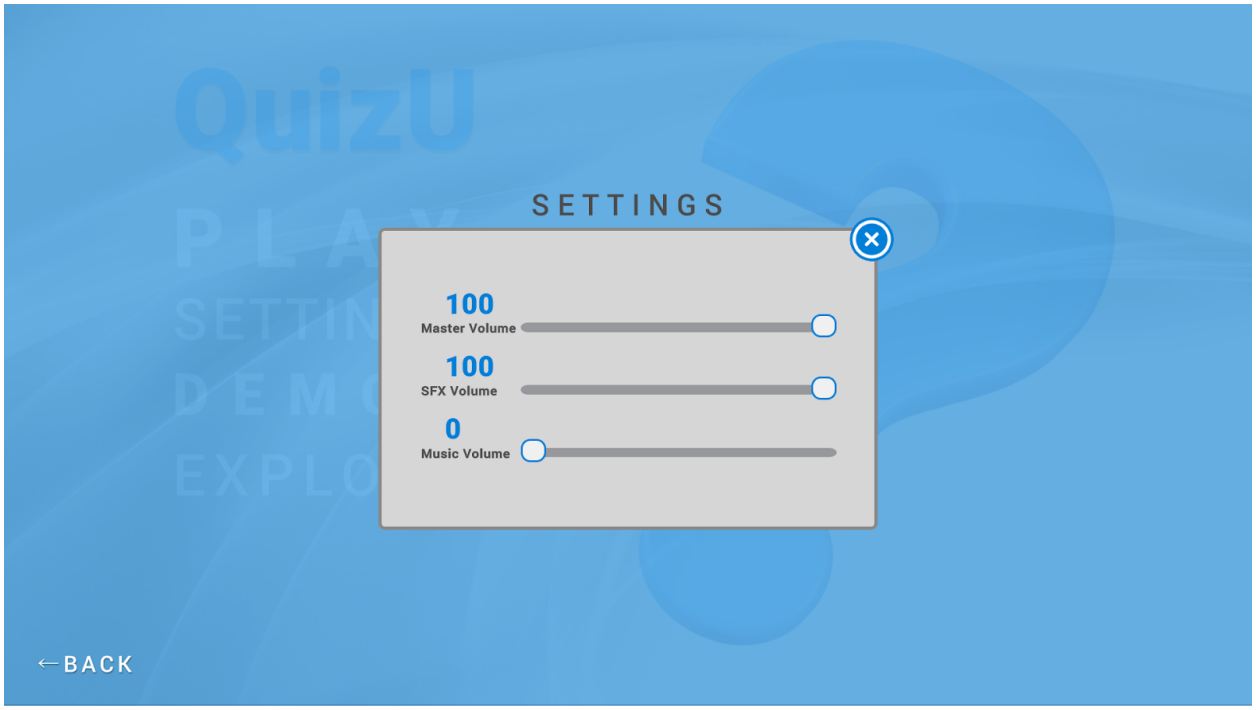
If you want to read more about MVC and MVP, see the free e-book, Level up your code with game programming patterns.

QuizU: MVP with UI Toolkit Example

Let's walk through a simple example in the QuizU project to see one way to set up MVP using the UI Toolkit-based interface. The SettingsScreen is a basic interface that allows the user to change master volume levels as well as the levels for the individual sound effects or background music.

To see it in action, open the **Boot** scene from the Scenes folder of the project. Then, select **Settings** from the menu.

The UI only contains a few UI Sliders like so:



The Settings Screen.

To build this screen in QuizU, we use MVP to structure our assets and classes:

- The Model consists of the **AudioSettings** (**\Assets\Quiz\Scripts\ ScriptableObjects.cs**) ScriptableObject, which connects to the **MainAudioMixer** asset. This contains audio configuration data, including the master volume, sound effects volume, and music volume. The Model also holds sound effect AudioClips that correspond to various game events (e.g. winning the game, answering incorrectly, etc).

- The View includes the **SettingsScreen** class. It's responsible for rendering the UI and responding to user interactions. In this case, that involves manipulating the UI sliders for audio settings. The **SettingsScreen.uxml** and **SettingsScreen.uss** define the structure and style of the UI.

- The Presenter – the **SettingsPresenter** class – acts as the glue between the Model and the View. After receiving user input from the View, it processes and updates the Model accordingly. If external changes happen to the Model (e.g. resetting when starting the game or loading data from a save), it can also refresh the View to match.

Let’s take a closer look at each part and see how they work.

The Model: AudioSettings

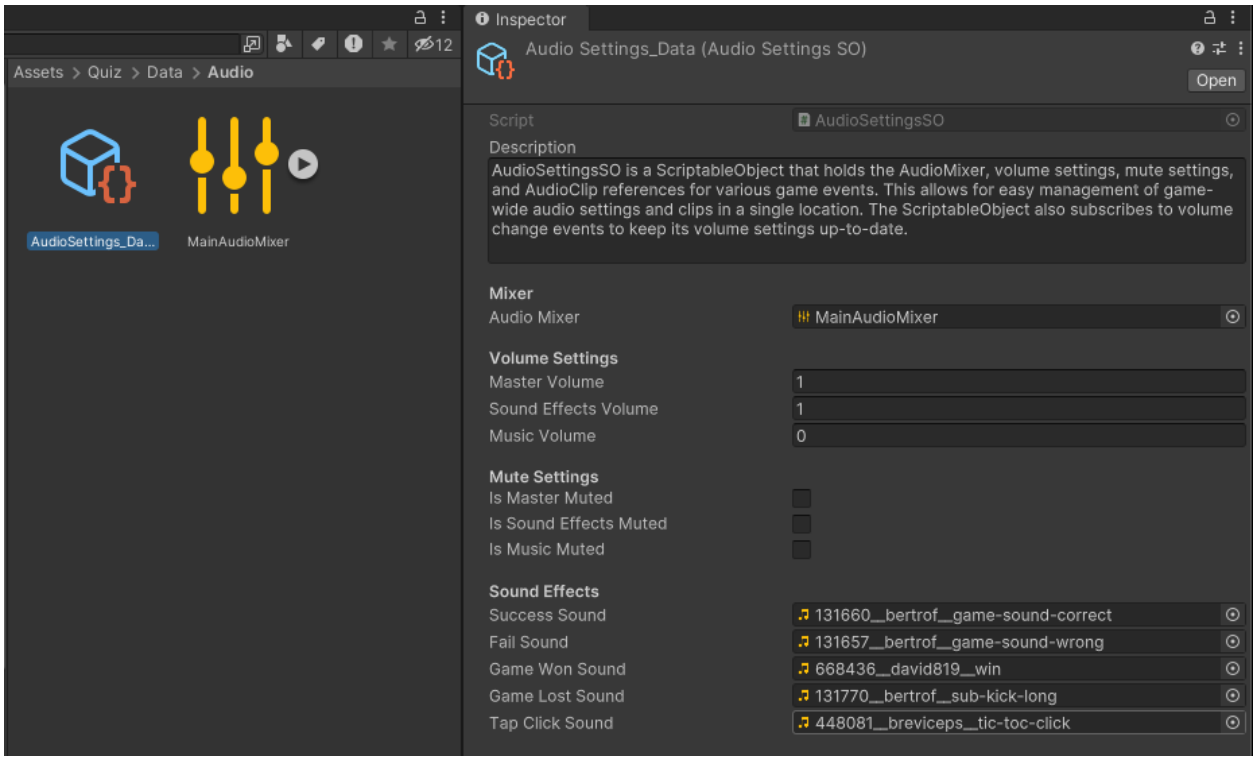
In MVP, everything in our **SettingsScreen** represents some value stored elsewhere. While you can use MonoBehaviours for data storage, ScriptableObjects offer a more optimized approach. They not only serialize their fields in the Inspector but also allow for a project-level Model that's accessible from any scene.

The **AudioSettingsSO (QuizU\Assets\Quiz\Scripts\ScriptableObjects\AudioSettingsSO.cs)** ScriptableObject includes:

- A reference to the **MainAudioMixer** that will control the sound levels.
- **Float properties** represent the master volume, sound effects volume, and music volume from a range of 0 to 1.
- A collection of AudioClips stands in for various **in-game sounds** (correctly answering, incorrectly answering, passing the quiz, failing the quiz, or just clicking a button).

Because the ScriptableObject is an asset at the project level, it can only refer to other project-level assets like prefabs. It can't directly reference objects from the Scene Hierarchy. This is inconvenient sometimes, but in this case, it's a good thing.

Remember, this Model knows nothing about any Views or Presenters - it stores and updates its data as directed. Using a ScriptableObject just enforces that.



The AudioSettings ScriptableObject

The View: Settings Screen

The View represents the user interface and consists of two parts:

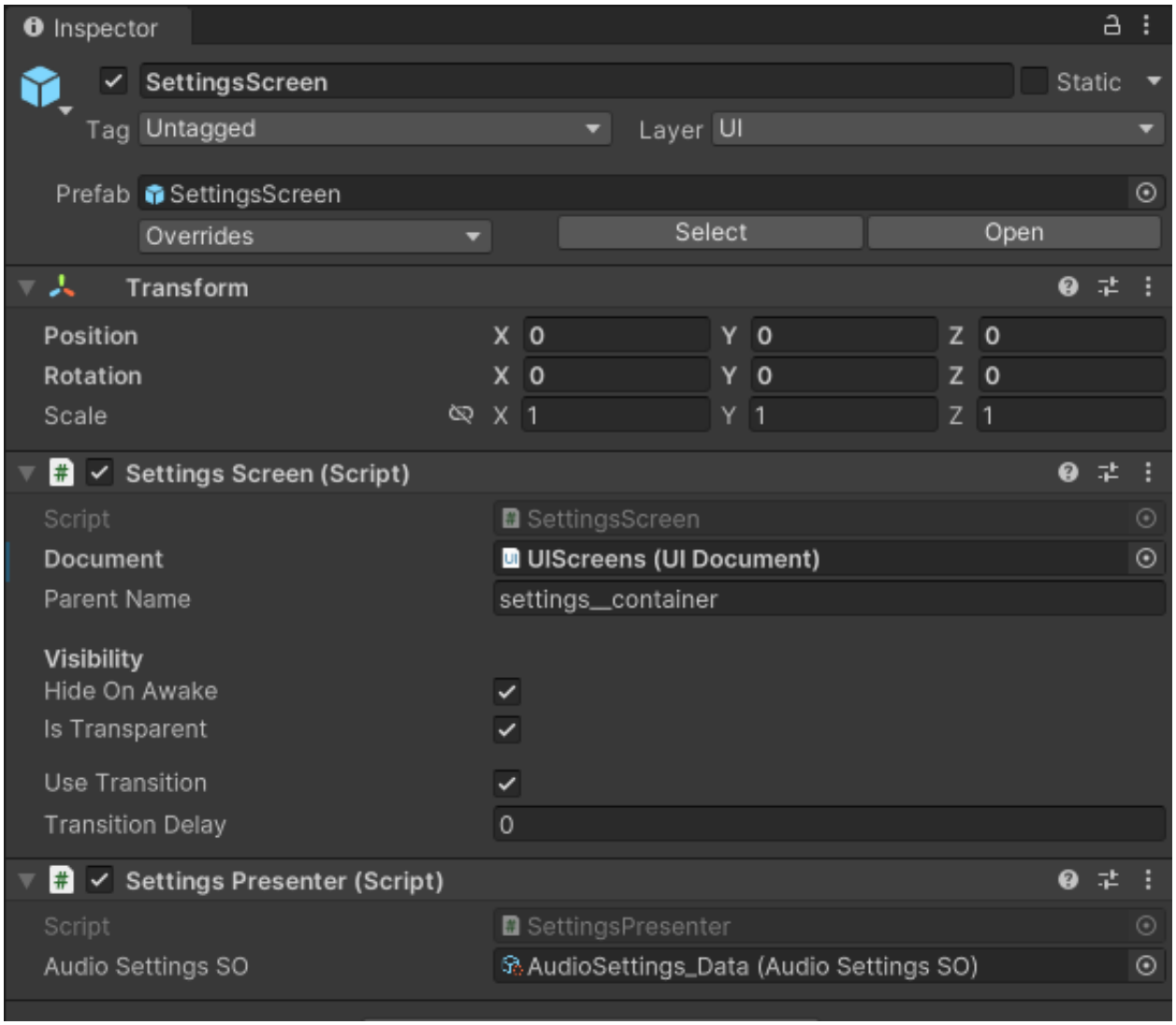
- **The UI Toolkit-based interface:** This includes the SettingsScreen.uxml and SettingsScreen.uss that define the visual tree hierarchy and its style sheets.
- **The SettingsScreen class:** This custom script manages all the UI elements (e.g. the sliders) and updates their state to match the data in the Model.

The **SettingsScreen** class **(Quiz\Scripts\UI\Screens\SettingsScreen.cs)** references all of the interactive elements within the parent container's visual tree and registers callbacks for sliders and button clicks.

It then determines what to do when the user applies input. In this implementation, that means updating each slider's corresponding label element and then notifying the Presenter when the UI has changed.

Because the View doesn't reference the Presenter, it communicates through events. Every time the user drags a slider handle at runtime, it raises a ChangeEvent with the new float value.

For example, when the user adjusts the master volume slider, this invokes the event handler (MasterVolumeChangeHandler). That updates the **m_MasterVolumeLabel.text** and then raises the **SettingsEvents.MasterSliderChanged** event. The Presenter listens to this event and then notifies the Model.



The SettingScreen represents the View logic.

The Presenter: SettingsPresenter

Central to the pattern, the **SettingsPresenter** (**\Quiz\Scripts\Managers\SettingsPresenter.cs**) sits between the View and the Model. It handles changes from the View, updates the Model, and vice versa. The Presenter receives user input events from the View, invokes appropriate methods on the Model, and performs necessary

calculations/manipulation of the data. This ensures that View and Model stay in sync but are decoupled.

In the QuizU sample, the **SettingsPresenter** class is a MonoBehaviour. It subscribes to events raised by both the **SettingsScreen** (View) and the **AudioSettingsSO** ScriptableObject (Model).

```
public class SettingsPresenter : MonoBehaviour
{
    ...

    // Event subscriptions
    private void OnEnable()
    {
        // Listen for events from the View/UI
        SettingsEvents.MasterSliderChanged +=
SettingsEvents_MasterSliderChanged;
        SettingsEvents.SFXSliderChanged += SettingsEvents_SFXSliderChanged;
        SettingsEvents.MusicSliderChanged +=
SettingsEvents_MusicSliderChanged;

        // Listen for events from the Model
        SettingsEvents.ModelMasterVolumeChanged +=
SettingsEvents_ModelMasterVolumeChanged;
        SettingsEvents.ModelSFXVolumeChanged +=
SettingsEvents_ModelSFXVolumeChanged;
        SettingsEvents.ModelMusicVolumeChanged +=
SettingsEvents_ModelMusicVolumeChanged;
    }

    // Event unsubscriptions
    private void OnDisable()
    {
        SettingsEvents.MasterSliderChanged -=
SettingsEvents_MasterSliderChanged;
        SettingsEvents.SFXSliderChanged -= SettingsEvents_SFXSliderChanged;
        SettingsEvents.MusicSliderChanged -=
SettingsEvents_MusicSliderChanged;

        SettingsEvents.ModelMasterVolumeChanged -=
SettingsEvents_ModelMasterVolumeChanged;
        SettingsEvents.ModelSFXVolumeChanged -=
SettingsEvents_ModelSFXVolumeChanged;
        SettingsEvents.ModelMusicVolumeChanged -=
SettingsEvents_ModelMusicVolumeChanged;
    }

    …
}
```

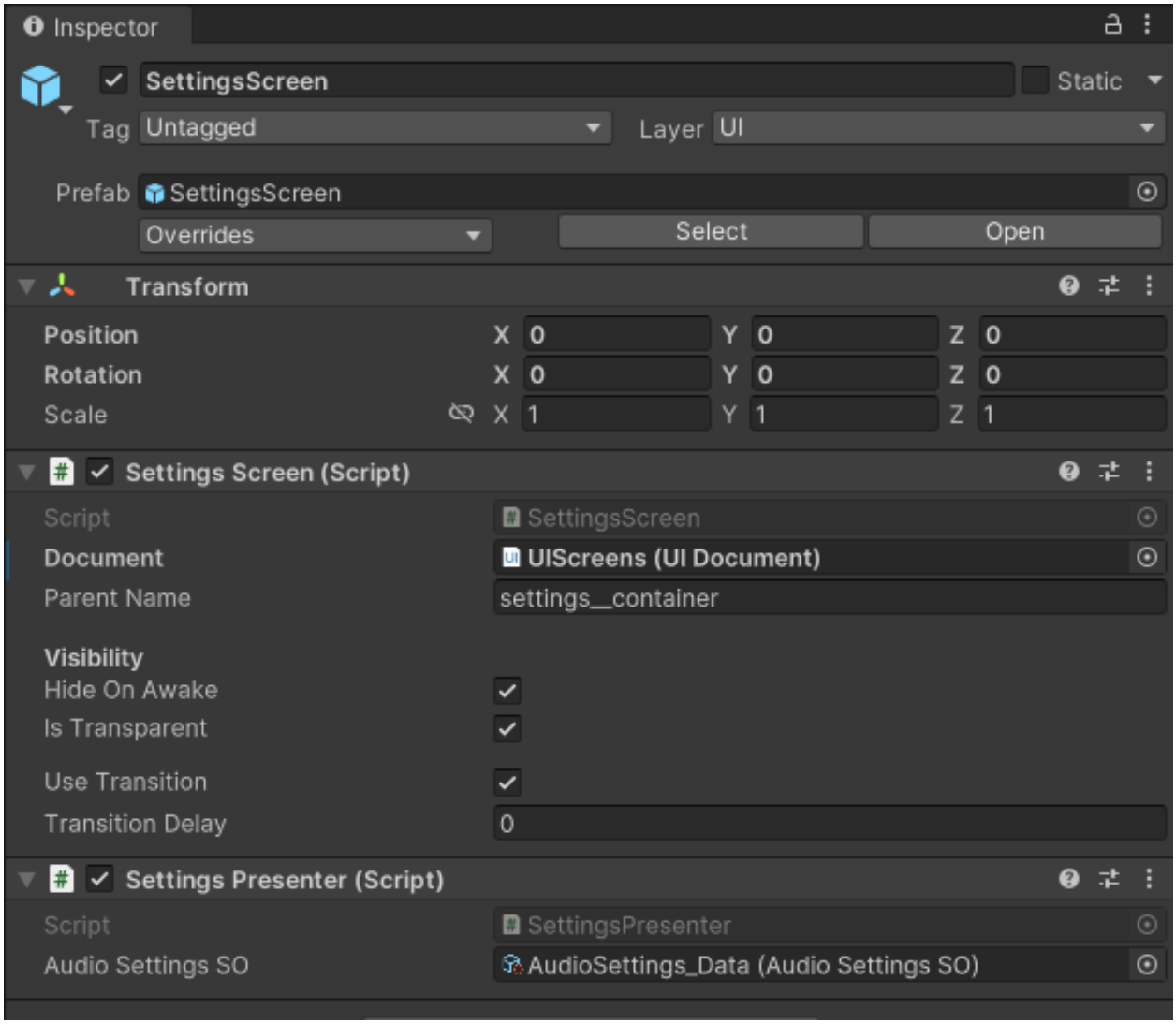
As you examine the SettingsPresenter, you'll note:

- Whenever the user moves a volume slider, the Presenter receives an event and notifies the Model to update.

- Likewise, if a game event changes the Model directly (e.g. the volume levels loaded from saved settings or loading for the first time), the Presenter notifies the View via events.

- The SettingsPresenter does not use UnityEngine.UIElements. This is another way to enforce the separation of concerns. Only the View will reference and manage UI elements. Instead, it sends messages between the user-interface (the View) and the AudioSettings ScriptableObject data (the Model).

- For convenience, the SettingsPresenter can maintain direct references to the View and the Model. In some circumstances, we might need to let the Presenter bypass events. For example, we can take advantage of the MonoBehaviour lifecycle events (Start or Awake) to initialize slider values when we enter Play mode.



The SettingsPresenter is the intermediary between the Model and View.

SettingsEvents

As you've seen in the previous examples, the View, Model, and Presenter from QuizU use public static delegates from the SettingsEvents class to communicate.

These aren't events in the strict C# sense – we want external objects to raise them – but they function as events for messaging purposes:

```
public static class SettingsEvents
{
        // Presenter -> View: sync UI sliders to Model
        public static Action<float> MasterSliderSet;
```

```
        public static Action<float> SFXSliderSet;
        public static Action<float> MusicSliderSet;

        // View -> Presenter: handle user input
        public static Action<float> MasterSliderChanged;
        public static Action<float> SFXSliderChanged;
        public static Action<float> MusicSliderChanged;

        // Presenter -> Model: update volume settings
        public static Action<float> MasterVolumeChanged;
        public static Action<float> SFXVolumeChanged;
        public static Action<float> MusicVolumeChanged;

        // Model -> Presenter: model values changed (e.g. loading saved
values)
        public static Action<float> ModelMasterVolumeChanged;
        public static Action<float> ModelSFXVolumeChanged;
        public static Action<float> ModelMusicVolumeChanged;
}
```

These delegates sit in between the objects that need to communicate. That lets them trade messages while staying decoupled.

This approach helps the Presenter, View, and Model each do what they do best. The View focuses on UI interactions, the Model manages data, and the Presenter coordinates between the two.

Further reading

Like other design patterns, you'll want to adapt MVP to your needs and style. For example, if the benefits of a ScriptableObject aren't apparent, maybe use a MonoBehaviour instead. The goal is to strike a balance between a consistent architecture and the flexibility to mix and match where necessary.

Though we hope that you don’t often need to do a massive rewrite of your UI code, using patterns like MVP can help prepare you for when it’s necessary. Then, you’ll be ready when the application scales and grows with your team.

If you’re interested in more software design patterns, please make sure to see the free e-book Level up your code with game programming patterns. You can also find our other best practices guides in the documentation.

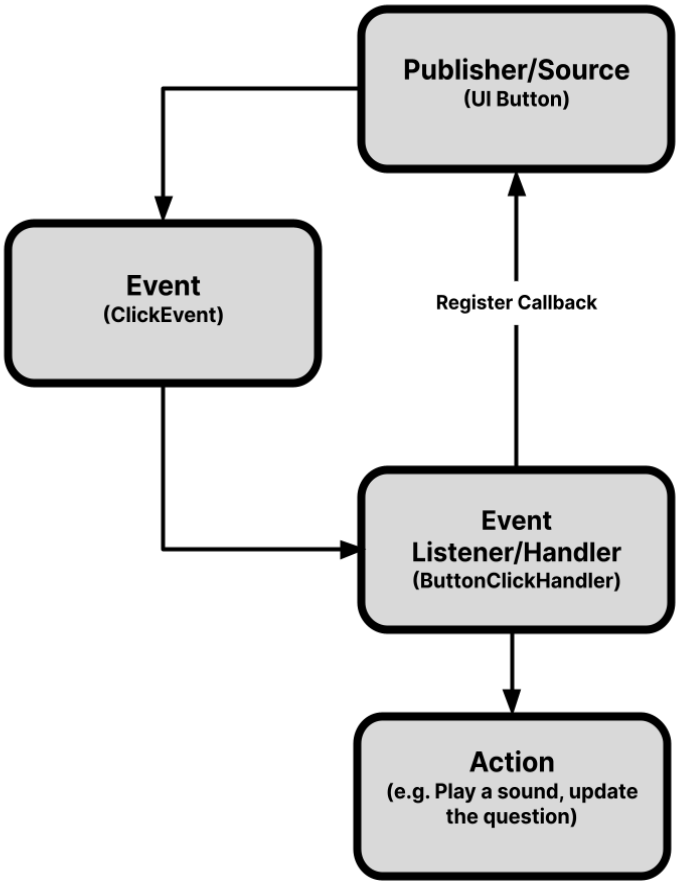
Finally, just a reminder that QuizU is not our only UI Toolkit demo. It complements two big pieces of content we released last year to help you get started with UI Toolkit:

- UI Toolkit sample – *Dragon Crashers*: This demo is on the Asset Store. It's a slice of a full-featured interface added to the 2D project Dragon Crashers, a mini RPG, using the Unity 2021 LTS UI Toolkit workflow at runtime.
- *User interface design and implementation in Unity*: This free e-book covers UI design and art creation fundamentals, and then moves on to instructional sections on UI development in Unity, mainly with UI Toolkit, but also with a chapter covering Unity UI.

QuizU: Event handling in UI Toolkit

UI development is event-driven. It revolves around anticipating user actions and responding to them. These interactions can be as straightforward as clicking a button or entering text, or as complex as executing drag-and-drop operations.

Consider our interactive QuizU game. When a user clicks a button on the Game Screen, the UI responds by displaying a new question. The application flows through a sequence of events and corresponding responses.



Events use a publisher-subscriber model.

Registering events

Every user input – a mouse click, key press, or pointer action – is an event. You handle events by registering event callbacks to specific UI elements. These callbacks are functions that get executed when the event occurs. For example, you can register an action to a button's ClickEvent so it executes some logic in response to a user clicking the button.

UI Toolkit uses a dedicated event system to make setting up events flexible and dynamic. An EventDispatcher listens for user interactions or script commands and then sends these events to the appropriate UI elements in your visual tree.

Setting up event handlers on UI elements often resembles something like this:

```
public class SomeClass()
{
      …

      // Associates UI elements with event handlers
      private void RegisterCallbacks()
      {
          // Registers a callback that triggers when the slider value
changes

m_Slider.RegisterCallback<ChangeEvent<float>>(SliderChangeHandler);

          // Registers callback that triggers when the button is clicked
          m_Button.RegisterCallback<ClickEvent>(ButtonClickHandler);

          // Registers callback that triggers when the button's layout
changes
          m_Button.RegisterCallback<GeometryChangedEvent>
(ButtonGeometryChangedHandler);
      }

       // Event handler that logs the new slider value when it changes
       private void SliderChangeHandler(ChangeEvent<float> evt)
       {
             Debug.Log("Slider value changed: " + evt.newValue);
       }

}
```

In this example, you identify specific elements that need interaction then register a callback to one of their UI Toolkit events. Here we use the RegisterCallback method from CallbackEventHandler to set up callbacks for a slider and a button:

- When the user drags the slider, the UI Toolkit event system raises a ChangeEvent with a new float value. The SliderChangeHandler method then processes the float value and performs an action.

- When the user presses the button and triggers the ClickEvent, a ButtonClickedHandler method executes in response.

- When something changes the same button's layout (position/width/height/transform), that triggers a GeometryChangedEvent. A different event handler can then respond. (e.g. adjust its text label, change color, etc.).

Note how elements can have more than one event, depending on what they need to do.

You can organize this according to the needs of your application. In this code snippet, we've grouped the RegisterCallback lines into a method for convenience; then we just need to invoke RegisterCallbacks when setting up the UI.

This is a small sampling of the events available. See the documentation for a complete Event reference and more about handling events.

Alternate callback syntax

In addition to using the RegisterCallback method, some events have alternative ways of setting up their handlers, as a matter of convenience.

- **ChangeEvents:** When working with a UI element that has a mutable value like a Slider or TextField, we can use the **RegisterValueChangedCallback** method. You can then use this to set up the event handler for the above Slider's ChangeEvent<float> just a little differently.

  Use the RegisterValueChangedCallback method with a ChangeEvent<T> where T is any struct type, like float, int, bool, etc.

- **ClickEvents:** Buttons have an even more simplified way to handle click events. When a button is clicked, instead of using RegisterCallback<ClickEvent>, you can use the **clicked** property.

  This `clicked` property is short for the button's `clickable.clicked`. Clickable is a manipulator available to every button that tracks mouse events. Using this syntax

is similar to using System.Action.

```
// Alternative way to add callback to the ChangeEvent<float>

m_Slider.RegisterValueChangedCallback(SliderChangeHandler);

private void SliderChangeHandler(ChangeEvent<float> evt)
{
    Debug.Log("Slider value changed: " + evt.newValue);
}

// Alternative way to handle a ClickEvent on the Button

m_Button.clicked += ButtonClickHandler;

private void ButtonClickHandler()
{
    Debug.Log("Button was clicked!");
}
```

Using event data

Events can also send custom data to their listeners. This data can provide useful context. For example, a **ClickEvent** has data that includes properties like **clickCount** (how many times the mouse button was pressed) and **mousePosition** (where the mouse click occurred).

Consider a button in your UI. Let's say you want to track the number of times it has been clicked and the position of the last click. The ClickEvent can assist with that:

```
public class SomeClass()
{
    int clickCount = 0;
    Vector2 lastClickPosition;

    private void RegisterCallbacks()
    {
        // ClickEvent callback for button
        m_Button.RegisterCallback<ClickEvent>(ButtonClickHandler);
    }

    private void ButtonClickHandler(ClickEvent evt)
    {
        clickCount = evt.clickCount;
        lastClickPosition = evt.mousePosition;

        // Perform your action here
        Debug.Log($"Button clicked {clickCount} times. Last click position:
{lastClickPosition}");
```

```
    }
}
```

In the `ButtonClickHandler` method, we're extracting the `clickCount` and `mousePosition` from the ClickEvent object. This information can then be used for debugging, analytics, or to drive gameplay mechanics.

Most UI events in Unity have their own specific data that you can use in context. For example, when the value of a Slider changes or when you type into a TextField, a **ChangeEvent<T>** happens.

For a complete list of events and their associated data, see the Event Reference.

Event dispatch and propagation

Events are dispatched to a specific visual element using a propagation path, as seen in the EventDispatch demo scene.

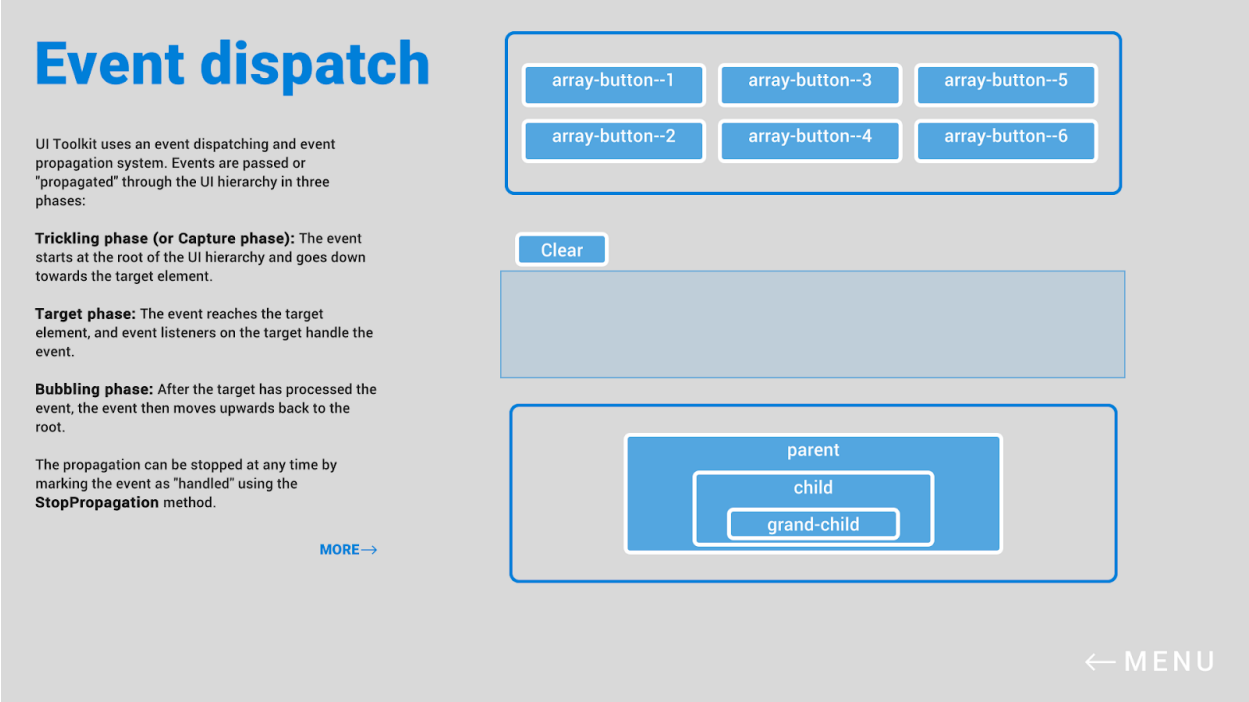
An event traverses the visual tree in phases: trickling down, at the target, and bubbling up. See this manual page for more detail.

To handle events in the appropriate phase, the EventBase class includes two important properties:

- **target**: This property refers to the visual element where the original event occurred. For instance, if the user clicks a button, the target of the ClickEvent would be that specific button.
- **currentTarget**: As an event moves along the propagation path, the `Event.currentTarget` property updates to the element currently handling the event.

These properties can help determine how and when to handle an event. For example, a callback registered on a parent element could handle click events for several child elements.

By checking the target property of the event, you can determine which child element was clicked and then handle the event. Take a closer look at the EventDispatchDemo script for sample usage.



The EventDispatchDemo scene shows how events propagate.

Unregistering events

In most cases, the lifetime of your controller object either matches or outlasts that of the UI, making it unnecessary to unregister event handlers. The garbage collector will usually reclaim callbacks along with their associated visual elements without issue.

However, there are specific cases where unregistering events is important:

- **External Object References:** If a callback refers to an external object that gets destroyed or reset (e.g. a game audio system or some other GameObject), unregister the event to avoid errors.

- **Context-Sensitive Buttons:** For buttons that change their behavior based on the game's context, unregister the existing callback and register a new action (e.g. a jump button that becomes an interact button).

- **Static Events:** If a VisualElement has registered a callback to a static event, that event will continue to hold a reference to the callback even after the element is destroyed. Unregister the callback to free the object for garbage collection.

To handle these situations, you can create a method (UnregisterEvents in the below example) where you unregister the event handlers:

```
public class SomeClass()
{
    …

    private void UnregisterEvents()
    {
        // Slider callbacks

m_Slider.UnregisterCallback<ChangeEvent<float>>(SliderChangeHandler);

        // ClickEvent callback for button
        m_Button.UnregisterCallback<ClickEvent>(ButtonClickHandler);
        // GeometryChangedEvent callback for button
        m_Button.UnregisterCallback<GeometryChangedEvent>
(ButtonGeometryChangedHandler);
    }
}
```

Though unregistration is often not a requirement in this case, getting into the habit of cleaning up after yourself can be a good thing. Be aware of your project's specific needs and unregister callbacks when necessary. For everything else, rely on the garbage collector.

The Event Registry pattern

Registering and unregistering a large number of callbacks, however, can be cumbersome.

A more maintainable solution is using a helper utility to manage your UI Toolkit events. This event registry can help you keep track of all the callbacks, making it easier to unregister them when needed.

To implement this, we create a class named **EventRegistry** that implements **IDisposable**. The interface just really means we need to add a public Dispose method like this:

```
public class EventRegistry : IDisposable
{
    // Single delegate to hold all unregister actions
    Action m_UnregisterActions;

    // Registers a callback for a specific VisualElement and event type (e.g.
ClickEvent, MouseEnterEvent, etc.).
    public void RegisterCallback<TEvent>(VisualElement visualElement, Action<TEvent>
callback) where TEvent : EventBase<TEvent>, new()
```

```
    {
        EventCallback<TEvent> eventCallback = new EventCallback<TEvent>(callback);
        visualElement.RegisterCallback(eventCallback);

        m_UnregisterActions += () =>
visualElement.UnregisterCallback(eventCallback);
    }
    // Unregisters all callbacks by invoking the m_UnregisterActions delegate, then
sets it to null.
    public void Dispose()
    {
        m_UnregisterActions?.Invoke();
        m_UnregisterActions = null;
    }
}
```

The **Dispose** method invokes the m_UnregisterActions delegate and then sets it back to null. This unregisters the callbacks all at once, ensuring that all callbacks in the EventRegistry are properly disposed of.

This is necessary when we want to clean up before deactivating or destroying a part of our user interface. One call to Dispose will unregister all of the event handlers.

This approach means that we don't have to manually manage each event callback's registration and deregistration. The EventRegistry class can do that for us, but we'll need to adjust how to handle the registration.

Using the Event Registry

The EventRegistry class now lets us register callbacks for easier disposal later. This can prevent the need to unregister each callback individually. Even if you have many event handlers in your UI, disposing of them just requires one method call.

To use it, we need to adjust our workflow. In the above SomeClass example, we instantiate the EventRegistry class and use that instance to register each callback for its corresponding visual element:

```
EventRegistry m_EventRegistry = new EventRegistry();
m_EventRegistry.RegisterCallback<ClickEvent>(m_Button, ButtonClickHandler);

void ButtonClickHandler(ClickEvent evt)
{
```

```
    Debug.Log("Button clicked!");
}
```

When we no longer need the callbacks (e.g. when the specific UI is no longer in use), we can call `Dispose` from a reference to the EventRegistry.

This is public so it can happen within the original SomeClass script or from another controller script managing that one. In the QuizU project, the GameScreen class manages several other smaller UIs. It can dispose of all UI elements under its care when destroyed or disabled.

Extending the Registry

The EventRegistry works for UI Toolkit events that derive from EventBase.

We can make some adjustments to the EventRegistry to handle other events as well:

```
public class EventRegistry : IDisposable
{
    Action m_UnregisterActions;


    public void RegisterCallback<TEvent>(VisualElement visualElement, Action<TEvent>
callback) where TEvent : EventBase<TEvent>, new()
    {
        EventCallback<TEvent> eventCallback = new EventCallback<TEvent>(callback);
        visualElement.RegisterCallback(eventCallback);

        m_UnregisterActions += () =>
visualElement.UnregisterCallback(eventCallback);
    }

    public void RegisterCallback<TEvent>(VisualElement visualElement, Action
callback) where TEvent : EventBase<TEvent>, new()
    {
        EventCallback<TEvent> eventCallback = new EventCallback<TEvent>((evt) =>
callback());
        visualElement.RegisterCallback(eventCallback);

        m_UnregisterActions += () =>
visualElement.UnregisterCallback(eventCallback);
    }

    public void RegisterValueChangedCallback<T>(BindableElement bindableElement,
Action<T> callback) where T : struct
```

```
    {
        EventCallback<ChangeEvent<T>> eventCallback = new
EventCallback<ChangeEvent<T>>(evt => callback(evt.newValue));
        bindableElement.RegisterCallback(eventCallback);

        m_UnregisterActions += () =>
bindableElement.UnregisterCallback(eventCallback);
    }

    public void Dispose()
    {
        m_UnregisterActions?.Invoke();
        m_UnregisterActions = null;
    }
}
```

- The overloaded **RegisterCallback<TEvent>(VisualElement, Action)** registers callbacks that don't require event data. This can be handy if you want to add simple actions to a set of buttons. This allows you to use any kind of System.Action, even a lambda, as a callback. The EventRegistry will unregister those automatically in Dispose.

- The **RegisterValueChangedCallback<T>** method is used with elements that have changeable values, such as sliders or text fields. When the value of these elements changes, the event passes a new value as a parameter.

- The **Dispose** method invokes the m_UnregisterActions delegate and then sets it back to null just like before. This unregisters the callbacks all at once, including callbacks from other types of events. One call to Dispose is still all we need to unregister all of the event handlers.

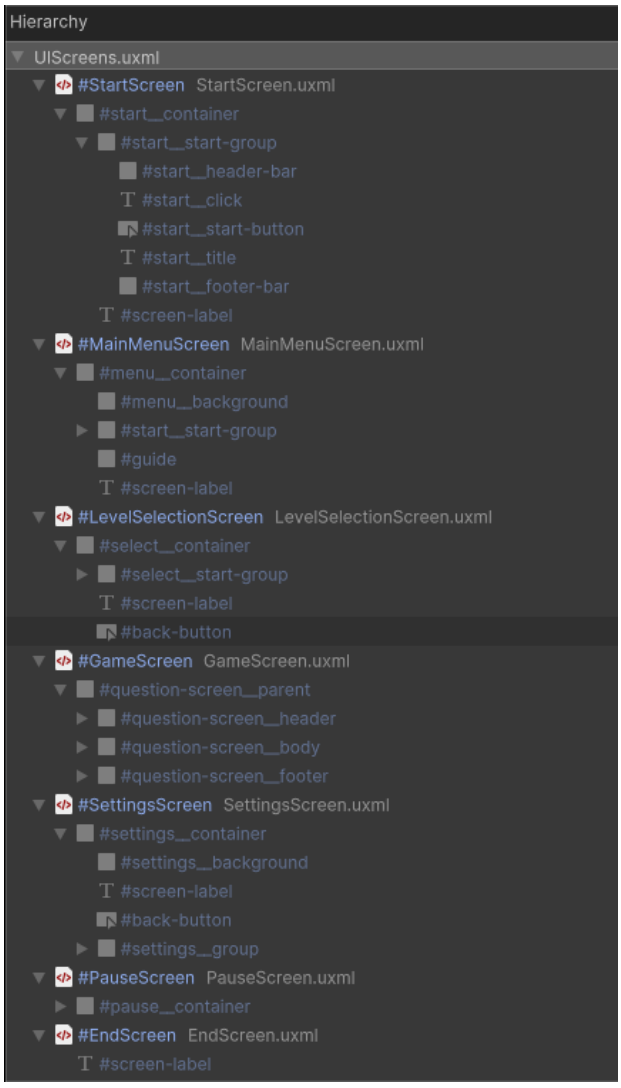
You can find this version of the **EventRegistry** in the QuizU sample and examples of how to use it throughout the project. This gives you a convenient way to manage your callbacks and dispose of them when they’re no longer needed. From simple button clicks to intricate UI interactions, consider using an EventRegistry as a flexible way to streamline callback management in your UI logic.

Then, you can worry less about how to track your event handlers, and instead spend your time on what's important,creating engaging UI experiences for your players.

UI Toolkit performance tips

Building complex UIs with UI Toolkit can enhance performance, but you'll still want to be mindful of best practices when deploying it.

Consider these tips and techniques for getting the most out of UI Toolkit.



Pre-create hierarchies rather than instantiating at runtime.

Hierarchy and elements

Pre-create hierarchies: Instantiating templates at runtime can slow your application. Pre-create your hierarchies in the Editor or during loading or non-peak periods to optimize performance.

Keep the visible element count low. More visible elements require more processing. Try to keep your UI lean and only display what is necessary.

Use DisplayStyle.None for hidden/offscreen elements. Instantiating a complex hierarchy of elements is a relatively slow operation. Pre-creating UI elements and setting them to DisplayStyle.None can improve performance, but balance this with memory usage.

Pool recurring elements. For elements that frequently appear and disappear, utilizing a pooling system can be beneficial. Rather than constantly creating and destroying these elements, you maintain a set of them in a pool. When needed, you simply reuse and recycle.

Use ListView for lists. ListView provides built-in pooling functionality, which can significantly improve performance for list-type UIs. ListView only renders the visible elements and recycles them as you scroll through the list, making it an efficient choice for long lists.

ListView provides built-in pooling for performance.

Implement custom culling. For non-ListView elements, you can implement custom culling using GeometryChangedEvent and VisualElement.layout property to manage the visibility and size of child elements.

Listen for the GeometryChangedEvent to know when a visual element's layout (size/position) property changes. If so, then check the VisualElement.layout against the

container displaying it. If the element is outside of this visible area, you can cull the child element by making it invisible.

Leverage UsageHints: Use the usageHints property to inform the system about how an element will be used at runtime. UsageHints can help optimize data storage and performance.

For instance, if you know that a particular element is going to be frequently updated (e.g., a score counter), you might set its UsageHints to DynamicTransform or DynamicMaterial, which tells the system to optimize the element for frequent updates.

Set the usageHints property before the VisualElement is part of a Panel. Otherwise, it is read-only.

Asset loading

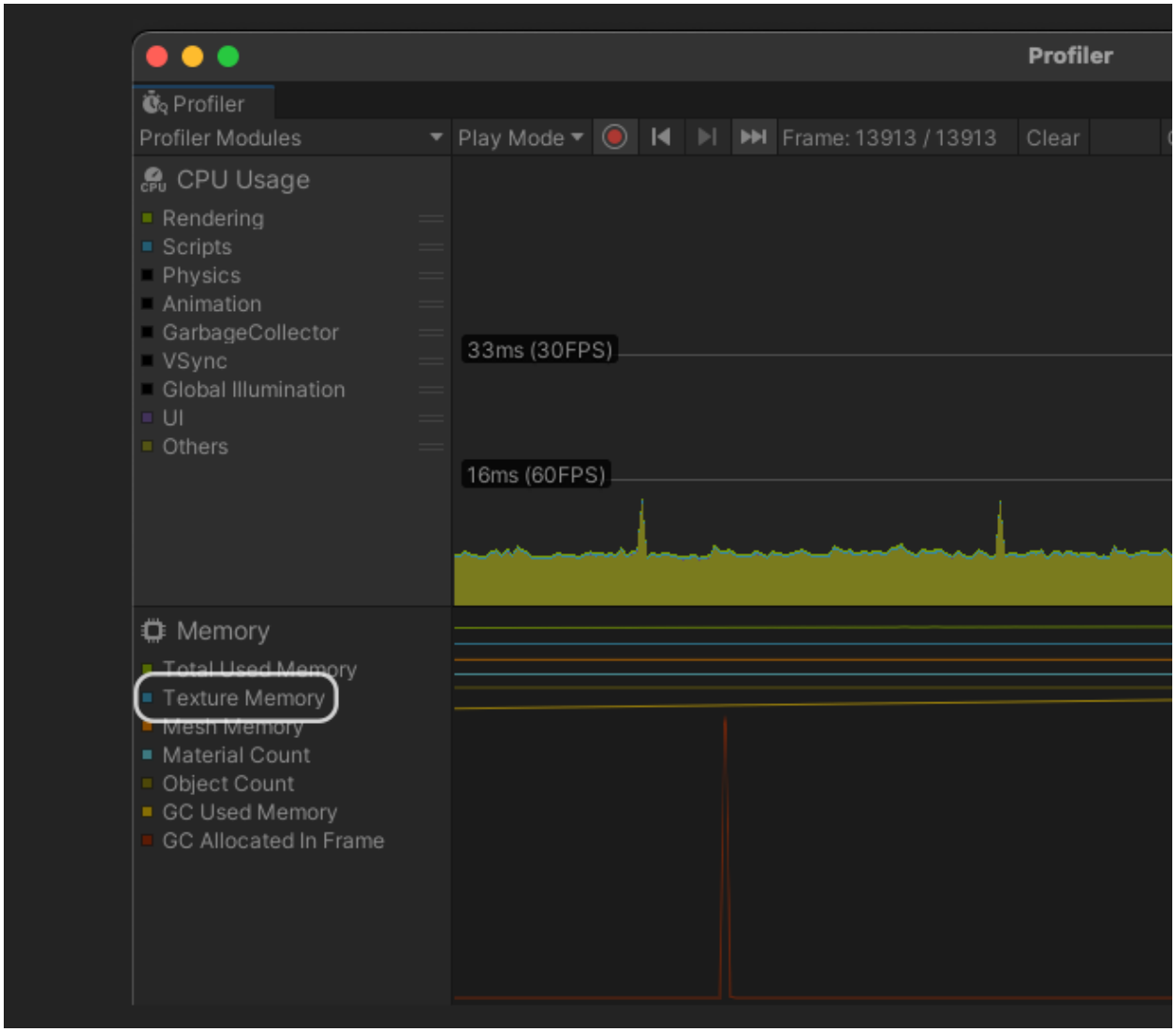
Optimize asset loading: Load assets or scenes during opportune moments, such as loading screens or transitions, to minimize disruptions to gameplay.

Understand UXML/USS dependencies: UXML/USS assets preload their dependencies, such as images referenced from the UI Builder. This is generally a good thing, but it can lead to inefficiencies if not managed correctly. Be aware of these dependencies to avoid unnecessary redundancy.

Mind your textures. Excessive textures can lead to an increase in draw calls. Be smart about texture usage and batching to maintain smooth rendering performance.

Utilize dynamic atlas systems. Small UI textures like icons can be added to an in-memory texture atlas. Misconfigured textures, however, can lead to unnecessary batch breaks. Doublecheck texture import settings, texture formats, mipmaps, non-power of two sizes, and compression settings when setting up a dynamic atlas.

Group textures with Sprite Atlases. You can use Sprite Atlases to group together textures used at the same time, which might be preferable to an auto-managed atlas.



Check texture memory in the Profiler.

UQuery

Cache Query results: UQuery requires traversing the hierarchy, which can be computationally expensive. Cache your query results for reuse to reduce overhead.

Leverage UQueryBuilder and the UQueryState struct. Avoid creating lists and optimize your queries by using UQueryBuilder and UQueryState. UQueryBuilder is a class used to construct queries to select specific UI elements based on various constraints like name, type, or class. UQueryState represents the result of an executed query. It contains the matching UI elements and allows iteration and manipulation of these elements.

For large UIs, querying for elements using UQuery can be more efficient than iterating

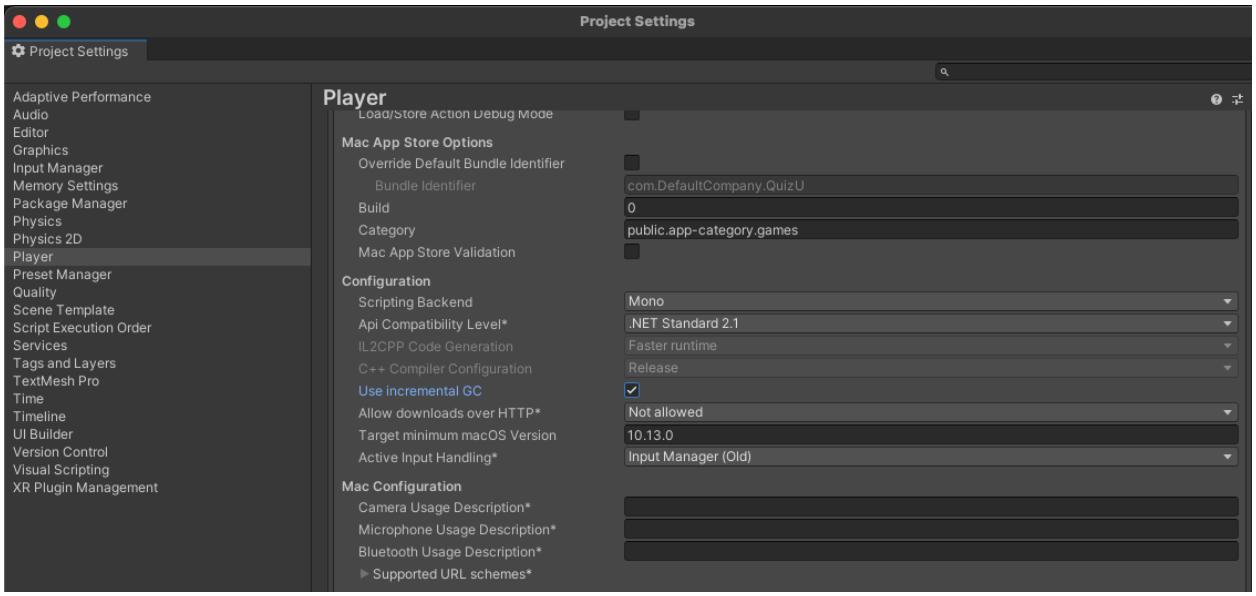
over a list, especially when the query results in a small subset of elements. Unlike a list, the UQuery system does not require a full iteration.

Garbage collection

Be mindful of garbage collection: Unity's garbage collection can feel like a black box, but try not to retain references to visual elements in classes that outlive the UIDocument or Window where they originated.

Be cautious about closures and lambda functions, which might accidentally retain a reference to a visual element; that visual element won't be garbage collected as long as the closure exists. This can accidentally lead to memory leaks and performance issues if not managed carefully.

Incremental GC can be beneficial: With high memory pressure, GC spikes can occur. In the Project Settings window, find the Configuration section, and check the box next to **Enable Incremental GC** to help alleviate this.



Enabling Incremental Garbage Collection can help reduce memory pressure.

Rendering

Understand UI Toolkit's shader approach. UI Toolkit uses a single shader approach, so each draw call isn't as costly as a full material change. Understand the trade-offs to optimize your UI (e.g. you might not be able to incorporate certain visual or shader effects into your UI if they require a different shader).

Avoid expensive tessellation: Any time the size (width/height) of an element changes, its geometry re-builds. This can be computationally expensive.

Instead of frequently changing the size of UI elements, you can use transforms for animations or adjustments. Transforms are less performance-intensive as they don't trigger tessellation.

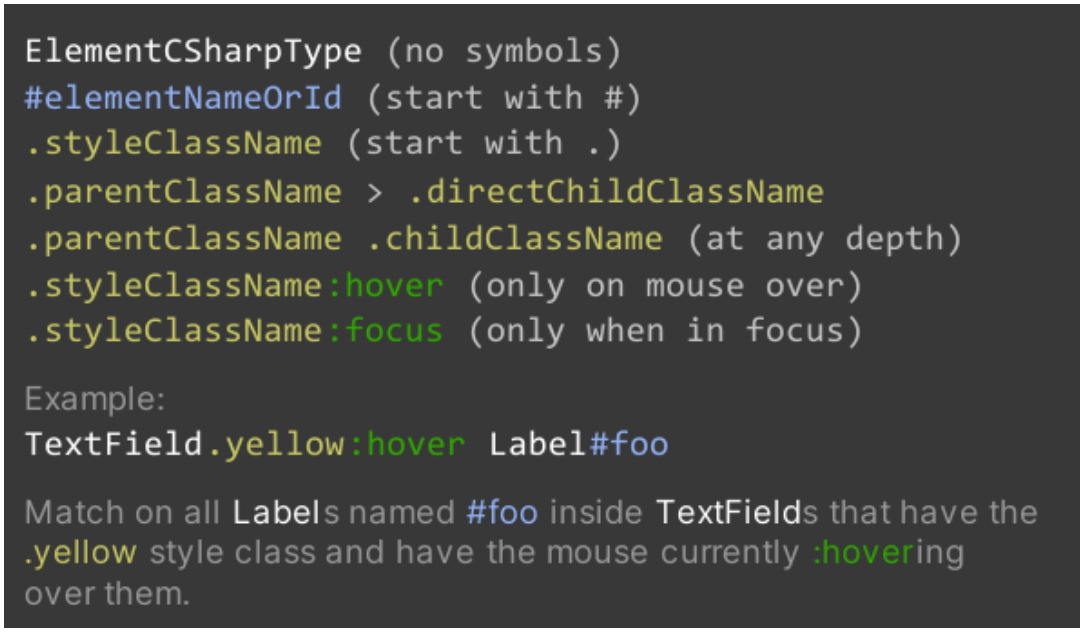
Similarly, using textures or 9-sliced sprites can be a better choice for rounded corners or borders. These alternatives do not trigger tessellation and can be more efficiently reused across different UI elements.

Choose the right font population mode. Use Static Font Assets for known text, such as labels and titles. They are fast and efficient, as they pre-bake characters into the atlas texture and don't require the source font file at build time. Dynamic OS Font Assets reduce memory overhead since the font source files don't need to be included in the build. Dynamic Font Assets incur additional performance overhead because the source font files must be included in the build.

Choose the right atlas mode for fonts. In terms of rendering, use Bitmap for pixel-perfect alignment, ideal for pixel art. For fonts that might undergo transformations or need special effects like outlines, opt for Signed Distance Field (SDF) rendering for a crisper and more adaptable presentation.

Styles and selectors

Simplify USS selectors. Complex hierarchical selectors can negatively impact performance. Use the Block-Element-Modifier convention to reduce complexity, and avoid the "*" selector as it requires *every* potential element to be tested against the selector.



USS selector cheat sheet (avoid the * selector).

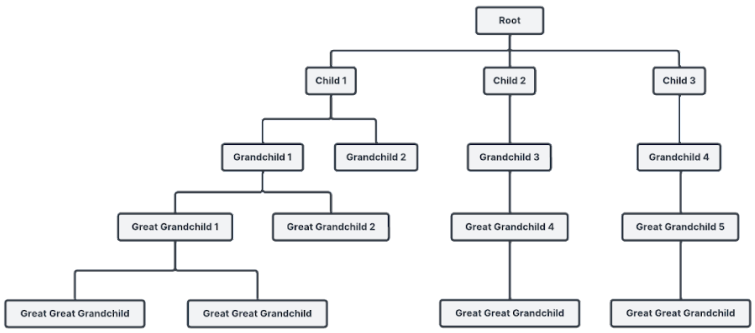
Use :hover sparingly. While **:hover** can enhance interactivity, it can trigger a redrawing of the hierarchy of targeted elements and their children. This could lead to significant performance overhead, especially if the hierarchy is complex or the changes are frequent.

Avoid inline styles. Inline styles have per-element memory overhead and require more processing to resolve the final style of an element. Favor using USS rules for efficient memory usage and better performance.

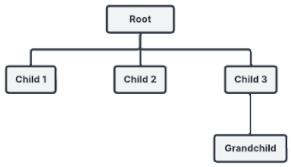
Balancing Performance and Usability

Of course, remember that optimization is both an art and a science. Some of these suggestions may lead to increased code complexity or a change of workflow, so factor that in before attempting a specific technique.

For more general tips about performance optimization in UI Toolkit, be sure to see the manual page in the documentation.



Avoid complex hierarchies with :hover



Use simple hierarchies

Be aware of complex hierarchies when using the :hover pseudo-state.

Remember to profile your UI to identify potential performance bottlenecks and optimize based on your specific project needs. Optimization often involves a trade-off between performance and usability. Always strive to strike the right balance for your specific use case.

In most cases, you will need to profile to see how these techniques can tangibly improve your specific project. Then, decide as a team which ones to implement.