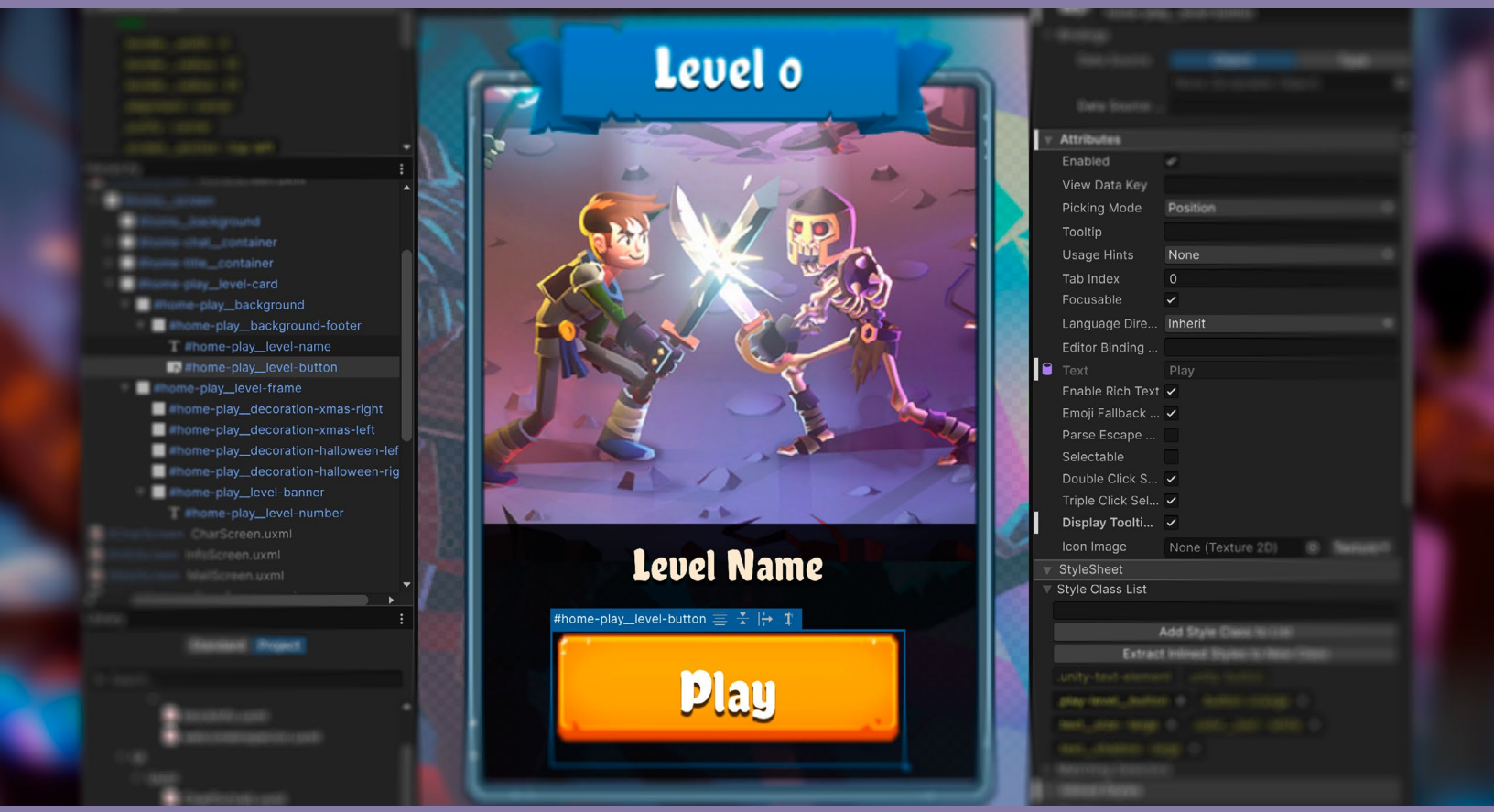
→ ЭЛЕКТРОННАЯ КНИГА

UI Toolkit для опытных разработчиков Unity

(издание для Unity 6)



Unity®

Содержание

Введение

Лучший пользовательский интерфейс - тот, которого не замечаешь.

Пользовательский интерфейс (UI) играет важнейшую роль в любой игре. Хорошо реализованный интерфейс незаметен и органично встроен в приложение. Плохо реализованный, напротив, раздражает пользователей и мешает получать удовольствие от игры.

Качественный UI продолжает визуальный стиль игры. Современная аудитория ожидает продуманный, понятный интерфейс, который естественно сочетается с приложением. Он служит игрокам основным источником важной информации - от показателей состояния персонажа до экономики игрового мира.

По мере усложнения интерфейсов растёт и мастерство их создателей. Над дизайном UI главным образом работают специалисты двух направлений:

Художники по UI. Они владеют основами дизайна, теорией цвета, формой, типографикой и компоновкой. Художники создают интерфейс с учётом целевой аудитории и игрового мира, а внимание к деталям помогает им добиваться точности вплоть до каждого пикселя.

UX-дизайнеры. Они изучают поведение пользователей и более широкие потребности конечной аудитории. UX-дизайнеры определяют способы взаимодействия человека с цифровым продуктом и выстраивают навигационные сценарии так, чтобы работа с ним была максимально понятной и приятной.

Эти специалисты тесно взаимодействуют друг с другом, а также с другими 2D- и 3D-художниками и дизайнерами. Именно такое сотрудничество позволяет создавать более цельные и эффективные интерфейсы.

Ещё одна важная роль принадлежит UI-программисту, который тесно сотрудничает с перечисленными специалистами. Он работает с выбранным технологическим стеком, выстраивает процесс или конвейер превращения дизайна UI в работающие интерфейсы, связывает игровой код с UI и передаёт данные из интерфейса обратно в игровые системы.

В предыдущей электронной книге «Проектирование и реализация пользовательского интерфейса в Unity» мы показали, как художники и дизайнеры UI могут создавать интерфейсы в Unity с помощью двух систем: более ранней Unity UI на основе GameObject и современной UI Toolkit. Мы также рассказали, как студии проектируют интерфейс с нуля и импортируют графику в игру. То руководство было подготовлено для Unity 2021 LTS.

В этой электронной книге основное внимание уделено UI Toolkit в Unity 6. Система рассчитана на высокую производительность и повторное использование компонентов, а её рабочие процессы вдохновлены веб-разработкой. Дизайнерам UI с опытом веб-разработки они покажутся знакомыми и понятными, а UI-программисты получат ясное представление о возможностях UI Toolkit при создании игр.

Благодаря модульной структуре разделы руководства можно читать в любом порядке, поэтому книга также удобна как справочник по UI Toolkit. Приступим.

Основной автор и участники

Основной автор этого руководства и создатель двух примеров UI Toolkit - Уилмер Лин, опытный художник по 3D-графике и визуальным эффектам, разработчик и преподаватель.

Значительный вклад в руководство и пример UI Toolkit Sample - Dragon Crashers внёс Эдуардо Орис, старший менеджер по контент-маркетингу Unity и графический дизайнер.

Ещё один ключевой участник, работавший над руководством и примером QuizU, - Томас Крог-Якобсен, старший менеджер по управлению контент-маркетингом Unity.

Другие участники со стороны Unity

Камиль Бузиди, разработчик программного обеспечения

Мартен Коте, старший разработчик графических технологий

Уго Бурре-Демаре, старший разработчик программного обеспечения

Бенуа Дюпюи, старший технический менеджер продукта

Карл Джонс, старший инженер-программист

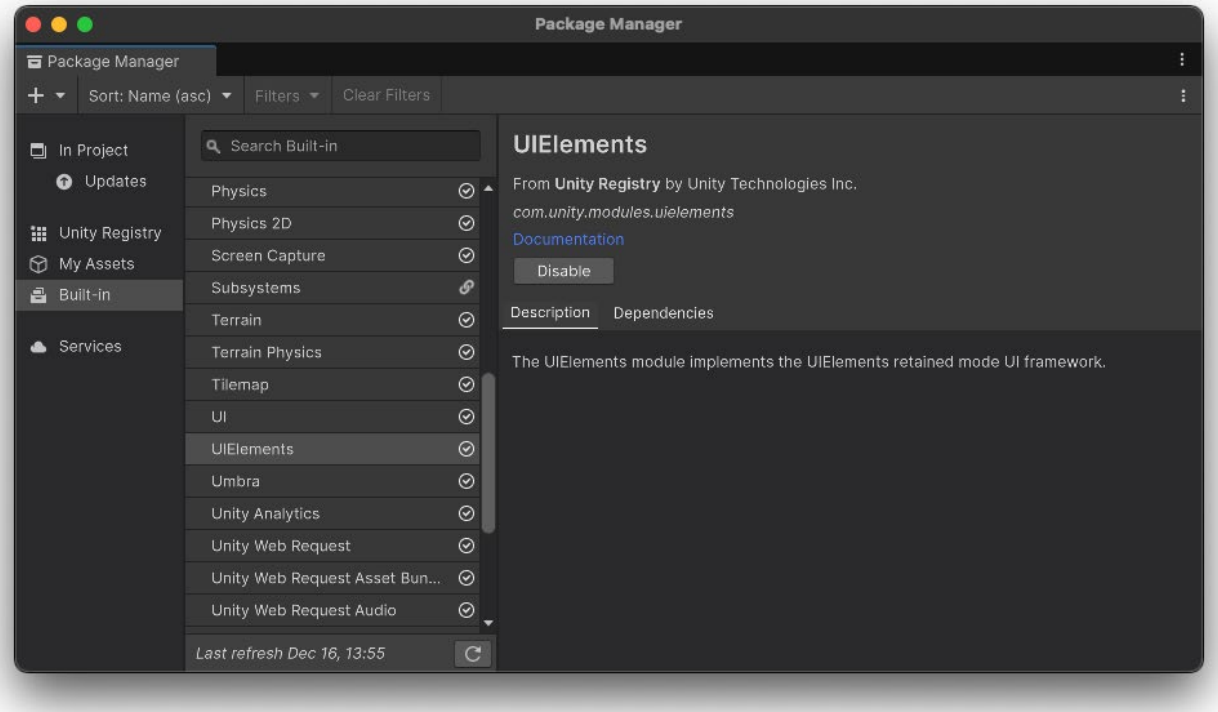
Антуан Лассозе, ведущий разработчик программного обеспечения

Мартен Паради, ведущий разработчик программного обеспечения

Стефания Валоросо, менеджер и продуктовый дизайнер

Установка UI Toolkit и примеров проектов

UI Toolkit входит в состав базовой платформы Unity 6, поэтому для Unity 6 и более поздних версий отдельный пакет устанавливать не требуется. Чтобы работать с материалами этого руководства, достаточно создать новый проект на основе одного из доступных шаблонов.



UIElements - пространство имён UI Toolkit, UI Builder и связанных с ними возможностей, которые теперь входят в Unity 6.

Официальные примеры UI Toolkit

В этой электронной книге возможности UI Toolkit в проектах Unity преимущественно демонстрируются на следующих примерах. Каждый из них можно бесплатно загрузить из Unity Asset Store.

Пример UI Toolkit - Dragon Crashers

В этой демонстрации поверх фрагмента двухмерной мини-RPG Dragon Crashers реализован полнофункциональный интерфейс, включая систему главного меню. Для него используется современный рабочий процесс UI Toolkit во время выполнения.

Пример рассчитан не на новичков, а на опытных разработчиков Unity, способных самостоятельно изучить структуру UI, перемещаться по проекту и разбирать конкретные реализации. Изначально демонстрация была выпущена для Unity 2021 LTS, а затем обновлена до Unity 6. В ней можно подробнее познакомиться со следующими темами:

- Структура проекта и соглашения об именовании

- Использование тем для создания вариантов UI с поддержкой книжной и альбомной ориентации

- Сложные элементы: меню с вкладками, инвентари, сообщения и пользовательские элементы управления

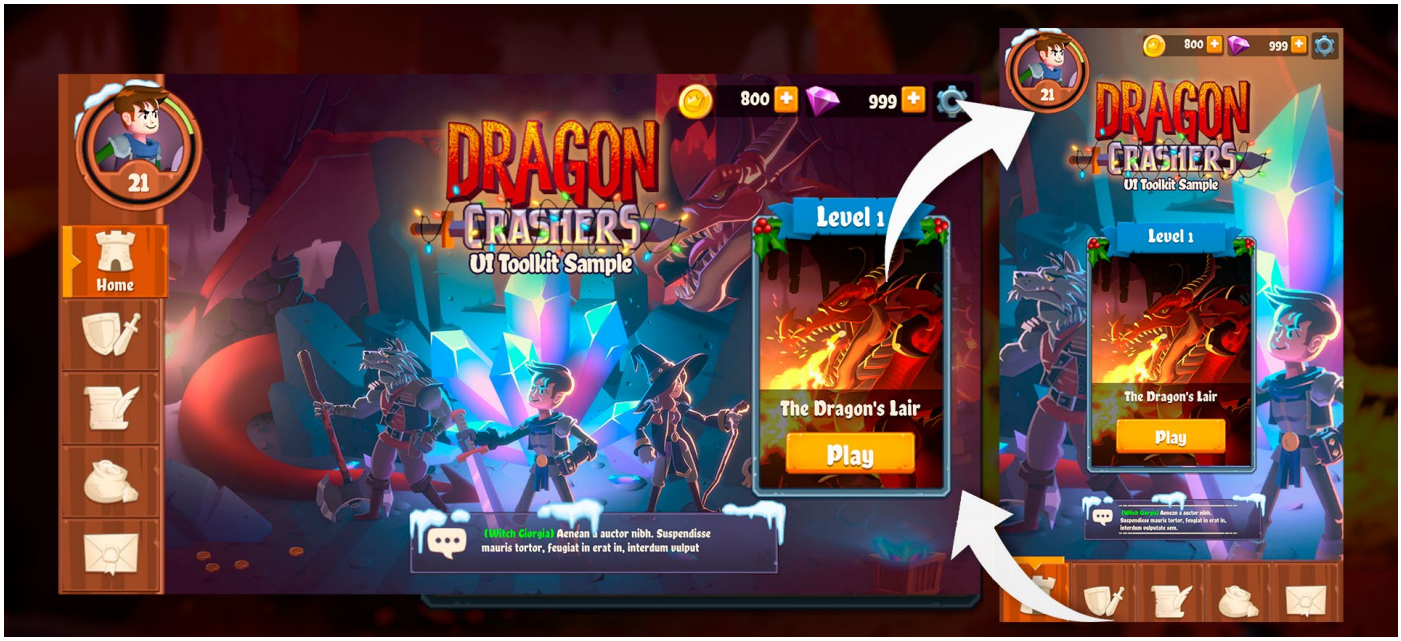
- Использование SafeAreaAPI, чтобы содержимое на мобильных экранах оставалось в пределах безопасной области

- Использование Localization для поддержки нескольких языков

- Привязка данных, упрощающая синхронизацию данных с компонентами UI

- Примеры реализации распространённых интерфейсов казуальных игр

Видеоруководство по примеру и сам проект доступны в Asset Store.

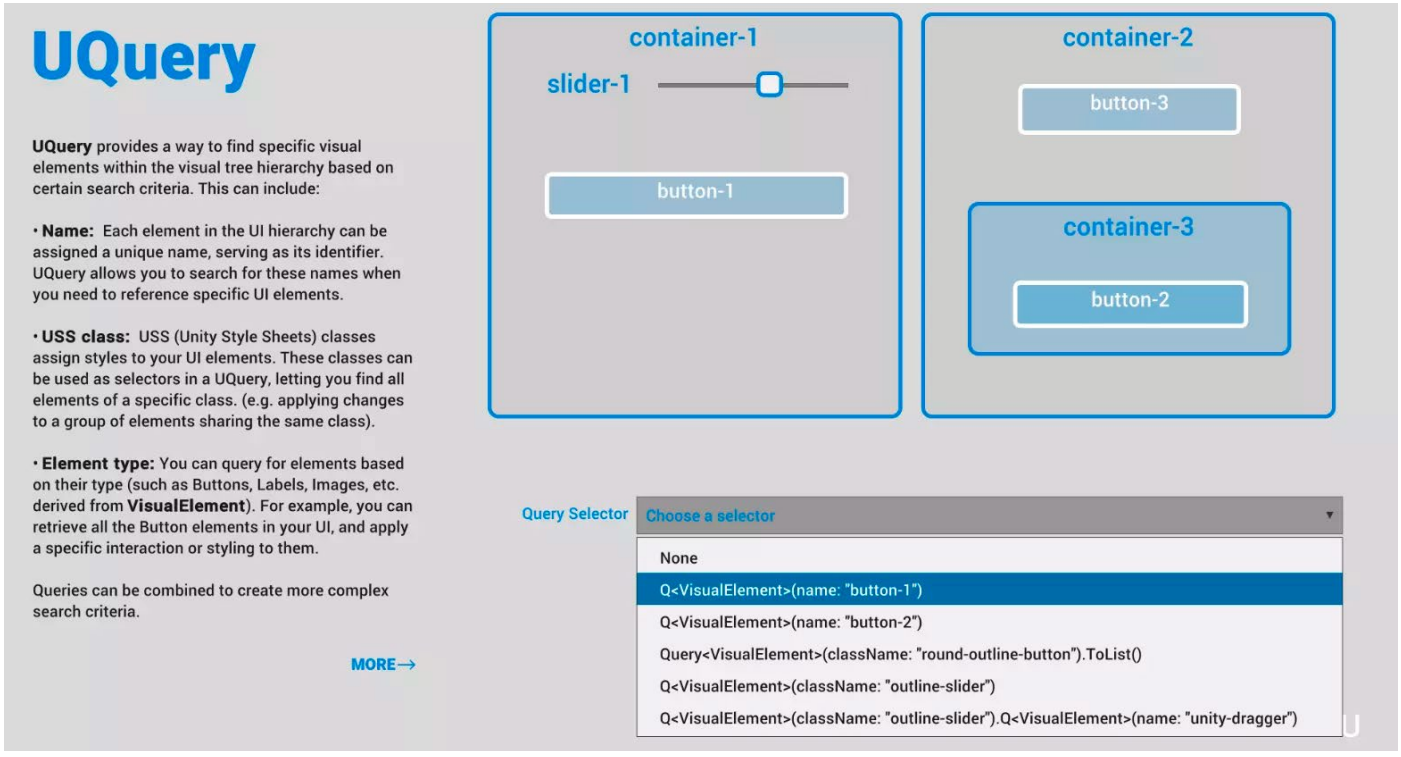


Главный экран в альбомной и книжной ориентации

QuizU

Демонстрационный проект QuizU представляет интерактивную викторину, созданную с помощью UI Toolkit в Unity. Проект рассчитан на разработчиков UI и знакомит с рабочими процессами UI Toolkit, событийной архитектурой и многократно используемыми шаблонами проектирования современных игровых интерфейсов. Демонстрация показывает, как:

- Эффективно организовывать элементы UI с помощью UXML-файлов и вложенных визуальных деревьев
- Применять правила стилей, селекторы USS и псевдоклассы, чтобы интерактивные элементы реагировали на действия пользователя
- Использовать Flexbox для гибких адаптивных макетов UI
- Динамически находить и изменять элементы UI с помощью селекторов
- Инкапсулировать обработку событий в повторно используемых классах, реализуя такие взаимодействия, как перетаскивание и мультитач-жесты
- Использовать диспетчеризацию событий (Event Dispatch), обрабатывать события по фазам и управлять их распространением
- Использовать переходы USS для плавной анимации и эффектов элементов UI, настраивая продолжительность и функцию плавности



Снимок экрана демонстрационного проекта QuizU

Проект QuizU, первоначально выпущенный для Unity 2022 LTS, получил поддержку новых возможностей Unity 6. Теперь в него входят практические примеры создания пользовательских элементов управления, настройки привязки данных и реализации локализации.

Проект можно загрузить из Asset Store.

Знакомство с UI Toolkit

UI Toolkit обладает существенными преимуществами перед традиционной системой Unity UI, также известной как uGUI, и устаревшей системой IMGUI для инструментов редактора. Это более современное, гибкое и производительное решение, которое лучше масштабируется для большинства проектов. UI Toolkit также может охватывать весь производственный конвейер: инструменты редактора, игры и приложения, работающие во время выполнения.

По сравнению с прежними системами UI она предоставляет следующие преимущества:

- Более быстрые итерации. Глобальное управление стилями и редактирование с немедленным отображением результата позволяют работать и вносить изменения быстрее.

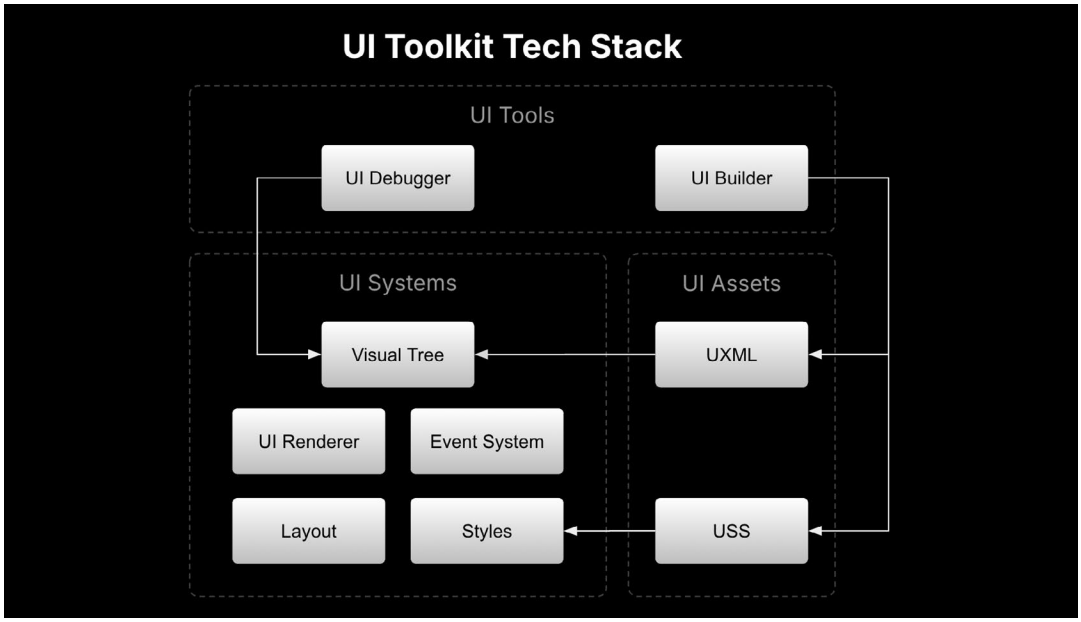
- Производительность рендеринга. Render Hints и динамические атласы текстур дают больше возможностей управлять производительностью игры.

- Более удобная совместная работа. Разделение логики (код C#), структуры UI (документы Unity XML, или UXML) и стилей (Unity Style Sheets, или USS) сокращает число конфликтов и упрощает командную работу.

- Повторное использование. Стили и виджеты можно совместно использовать в одном или нескольких проектах, а также переносить между редактором и средой выполнения.

UI Toolkit основан на подходах веб-технологий, поэтому разработчикам веб-приложений проще освоить эту систему. Для тех, кто впервые сталкивается с языками разметки, такими как HTML/XML, и

каскадными таблицами стилей (CSS), знакомство с UI Toolkit станет отличной возможностью освоить мощный набор отраслевых стандартных инструментов.

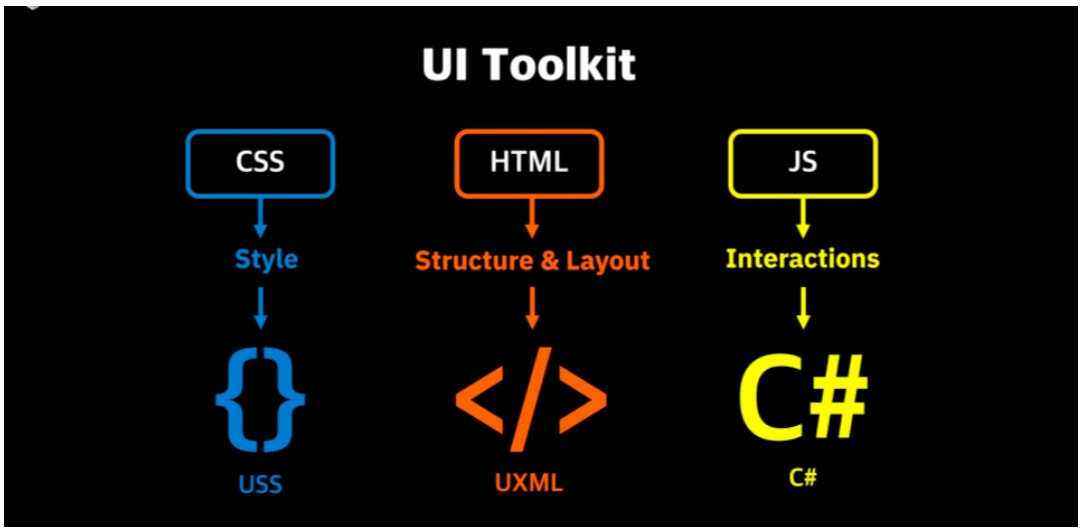


По сути, интерфейсы UI Toolkit строятся из файлов UXML и USS: система UI Toolkit использует их для формирования макета и оформления.

Ресурсы интерфейса

Основными строительными блоками интерфейса служат файлы UXML и USS. UXML (Unity XML) описывает содержимое и структуру UI и напоминает такие языки разметки, как HTML и XML.

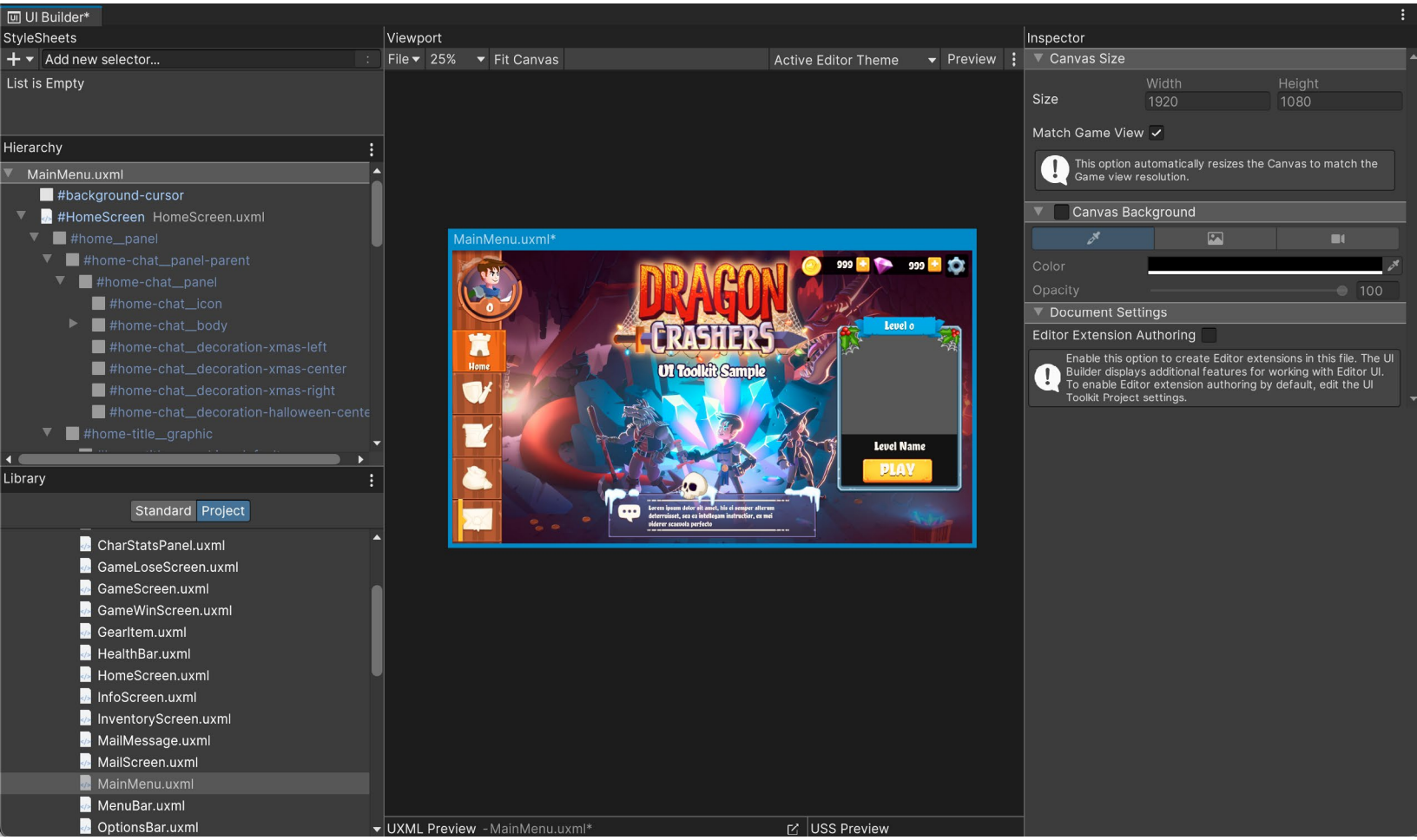
USS, созданный по образцу каскадных таблиц стилей (CSS), определяет внешний вид и стили содержимого UI. В этом руководстве используются и UXML, и USS.



Сходство UI Toolkit с веб-технологиями

UI Builder

Ресурсы интерфейса можно создавать в виде кода в выбранной IDE или визуально, в UI Builder, который входит в UI Toolkit. Интерфейс UI Builder позволяет художникам и дизайнерам редактировать UI и сразу видеть его в процессе создания.



Подготовка графических ресурсов и шрифтов

Значительная часть работы над дизайном UI выполняется за пределами Unity, в приложениях для создания цифрового контента (DCC). В зависимости от стиля и предпочтений UI-художник может проектировать интерфейс в редакторе растровой графики, например Adobe Photoshop, или в векторном редакторе. Обычно каждый графический элемент UI экспортируется в виде растрового изображения с прозрачностью и сжатием без потерь, например PNG, а затем объединяется с другими элементами в атлас текстур ради эффективности во время выполнения.

Если вы работаете в векторном DCC-приложении, для использования графики в UI Toolkit её потребуется экспортировать в растровый формат. Подробнее см. раздел «Векторные изображения».

Растровые изображения

Unity поддерживает большинство распространённых форматов изображений: PNG, BMP, TIF, TGA, JPG и PSD. После добавления таких файлов в папку Assets Unity импортирует их как ресурсы Texture 2D для трёхмерных проектов или как спрайты для двухмерных. После импорта тип можно изменить в поле Texture Type инспектора. UI Toolkit поддерживает оба формата растровой графики интерфейса.

Текстура почти не содержит сведений помимо размеров и формата изображения, тогда как спрайт обладает дополнительными свойствами, которые использует UI Toolkit.

Спрайты

Спрайты - это текстуры, подготовленные для двухмерных игр и предназначенные для компонента Sprite Renderer. В Unity двухмерные спрайты можно замостить, снабдить скелетом и скиннингом для анимации, задать им пользовательскую геометрию или дополнительные карты для двухмерного освещения. В этом разделе рассматриваются только параметры, относящиеся к UI Toolkit. Подробнее о двухмерной графике можно будет прочитать в издании электронной книги «2D-графика, анимация и освещение» для Unity 6 на странице https://unity.com/resources.

Большинство графических ресурсов UI выводится в экранном пространстве, а не в масштабе мира Unity, где одна единица соответствует кубическому метру трёхмерного пространства. UI Toolkit управляет масштабом такой графики, однако значение PPU (Pixels Per Unit, пикселей на единицу) влияет на размер спрайта в интерфейсе. Например, если на одну единицу сетки должно приходиться 128 пикселей изображения, задайте PPU равным 128.

Sprite Editor предоставляет инструменты редактирования графики: например, обрезку синими маркерами и нарезку зелёными. С их помощью изображение можно подготовить к мозаичному заполнению или применить технику девятисегментного масштабирования (9-slice), широко используемую для создания масштабируемых элементов.

Спрайты представляют собой двухмерные текстуры, наложенные на плоские прямоугольные трёхмерные сетки. При импорте по умолчанию используется параметр Mesh Type: Tight. Он подгоняет сетку под контур непрозрачных пикселей спрайта. Это повышает производительность за счёт уменьшения избыточной отрисовки, когда GPU в одном кадре несколько раз рисует один и тот же пиксель из-за перекрывающихся прозрачных областей. Сетку можно вручную уточнить и оптимизировать в разделе Outline окна Sprite Editor.

Полезный параметр Sprite Mode выбирается в инспекторе ресурса спрайта и поддерживает следующие режимы:

- Single. Режим по умолчанию: текстура содержит одно изображение.

- Multiple. Выберите этот режим, если исходная текстура содержит несколько элементов. В Sprite Editor укажите их расположение, чтобы Unity смогла разделить изображение на отдельные дочерние ресурсы. После нарезки каждый графический элемент становится самостоятельным спрайтом, который можно отдельно использовать в UI Toolkit.

- Polygon. Лучше всего подходит для круглых изображений и правильных многоугольников. Режим позволяет задать контур, точно повторяющий форму изображения, и получить более аккуратный результат.

Ресурс Render Texture

Render Texture - это текстура со снимком изображения камеры, которая обновляется в каждом кадре. Такой ресурс можно создать через меню Assets > Create > Rendering, а затем указать в меню Output компонента Camera. В UI Toolkit эти текстуры позволяют отображать мини-карты, экраны выбора персонажа и другую внутриигровую графику, которую требуется встроить в интерфейс.

В UI Toolkit Sample: Dragon Crashers Render Texture используется, например, для предпросмотра персонажа на индикаторе уровня и эффектов частиц поверх кнопок UI.

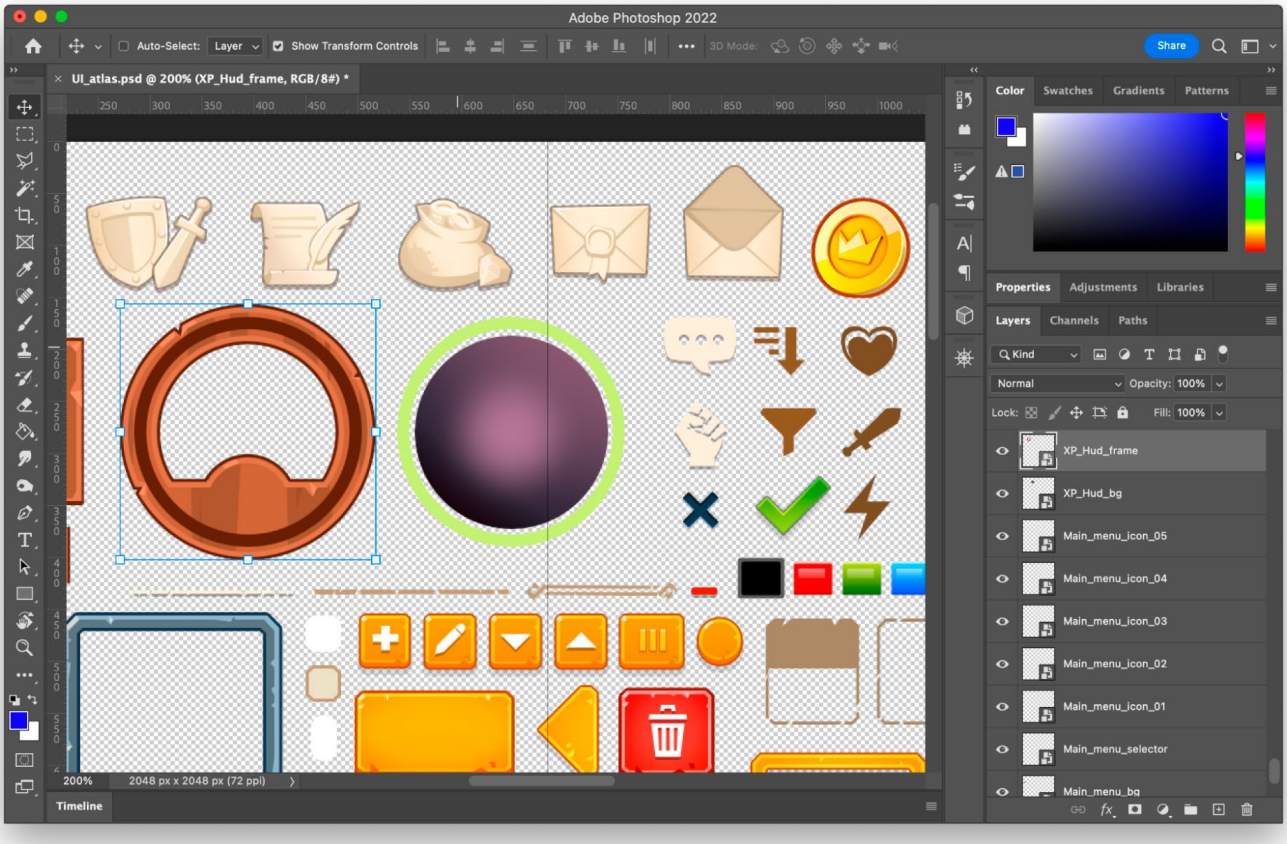
Возможен и обратный сценарий, когда интерфейс нужно показать внутри игрового объекта. Представьте трёхмерную модель компьютера с работающим интерфейсом, созданным в UI Toolkit. Интерфейс UI Toolkit можно отрисовать в Render Texture, назначить её в Panel Settings и Camera, а затем применить к материалу трёхмерной модели.

Параметры Render Texture и простые тесты

Помните, что Render Texture требует значительных ресурсов. Используйте такие текстуры умеренно и обязательно профилируйте проект, чтобы оптимизировать производительность. В полноэкранных интерфейсах, где одновременно не выполняются другие активные элементы игрового процесса, дополнительные эффекты такого рода, скорее всего, не вызовут серьёзных проблем с производительностью.

2D PSD Importer

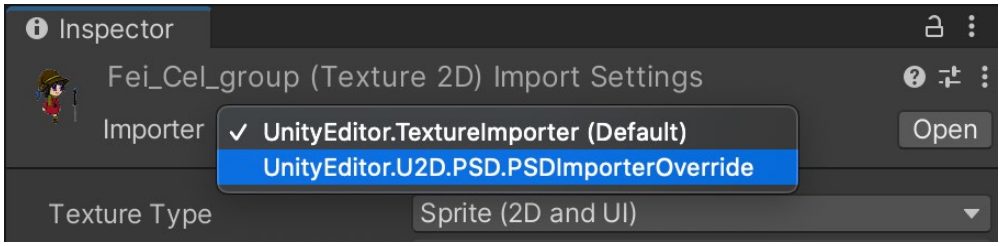
По умолчанию Unity импортирует PSD-файлы Adobe Photoshop как сведённые текстуры. Чтобы сохранить структуру слоёв, в проекте должен быть установлен пакет 2D PSD Importer. Формат PSD обычно используют для хранения нескольких изображений на разных слоях в одном файле; экспорт в него поддерживает большинство DCC-инструментов.



Создание ресурсов UI в Photoshop. Обычно каждый элемент размещают на отдельном слое, в отдельной группе или оформляют как смарт-объект. Смарт-объекты позволяют редактировать элементы независимо и сохраняют их исходное разрешение, даже если позднее изменить размер в основном документе.

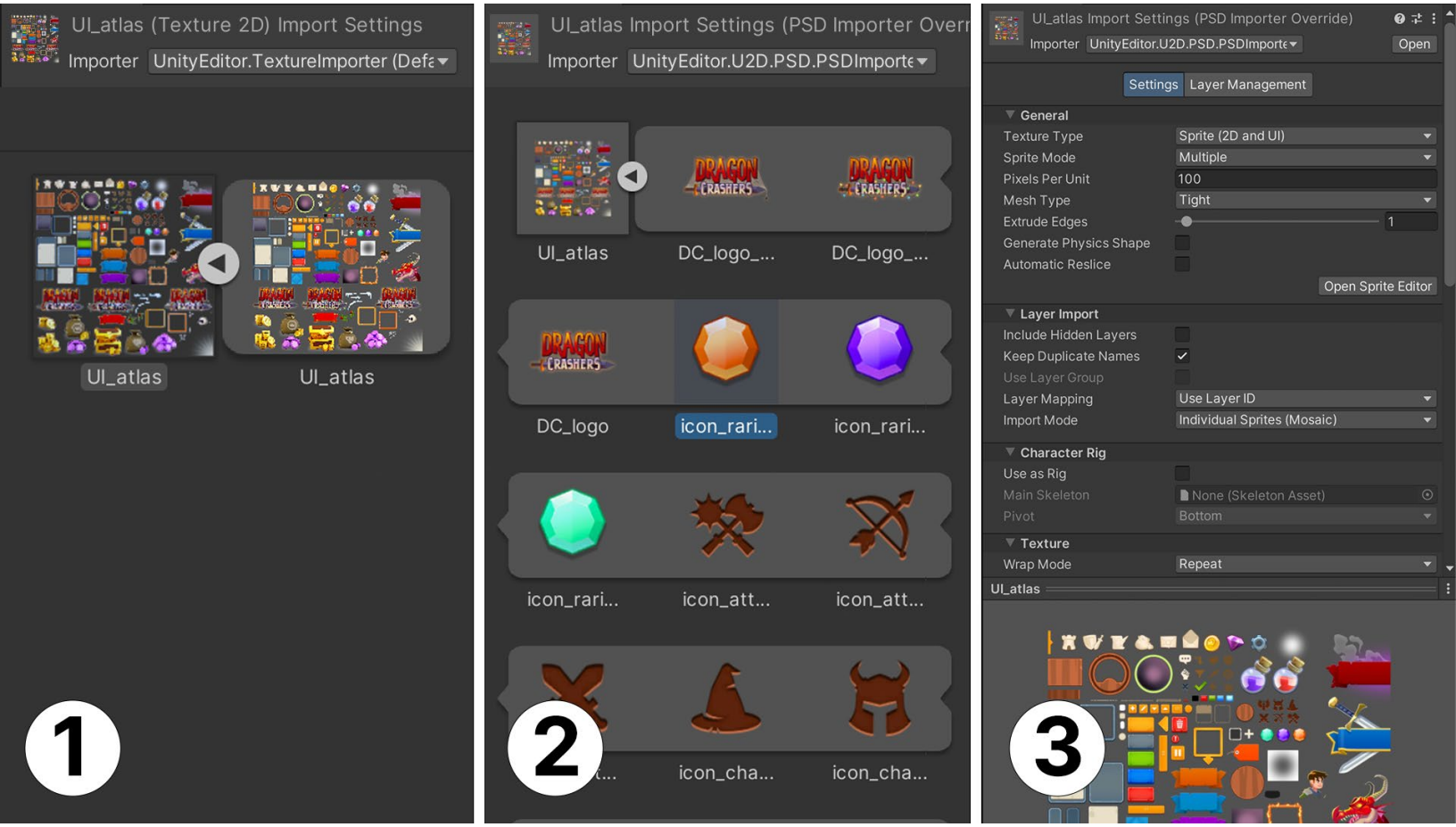
PSD-файлы упрощают рабочий процесс: их можно импортировать непосредственно в Unity, не экспортируя каждый слой в отдельный файл и не повторяя эту операцию после каждого изменения.

Установите пакет 2D PSD Importer через Package Manager, затем проверьте параметры импорта PSD-файла в инспекторе.



Выберите PSD Importer в инспекторе, чтобы просмотреть параметры обработки файла.

При работе с ресурсами UI отключите в инспекторе параметр Use as Rig в разделе Character Rig. Он нужен только для скелетной анимации двухмерных персонажей и не требуется элементам UI. Здесь же доступны параметры импорта слоёв: например, исключение скрытых слоёв и группировка объектов по слоям.



Переключитесь на PSD Importer, чтобы получить дополнительные параметры импорта.

Созданные из PSD спрайты в окне Project можно использовать как обычные. В Sprite Editor их можно нарезать, изменить контур или настроить Pixels Per Unit (PPU) так же, как у стандартных ресурсов Sprite.

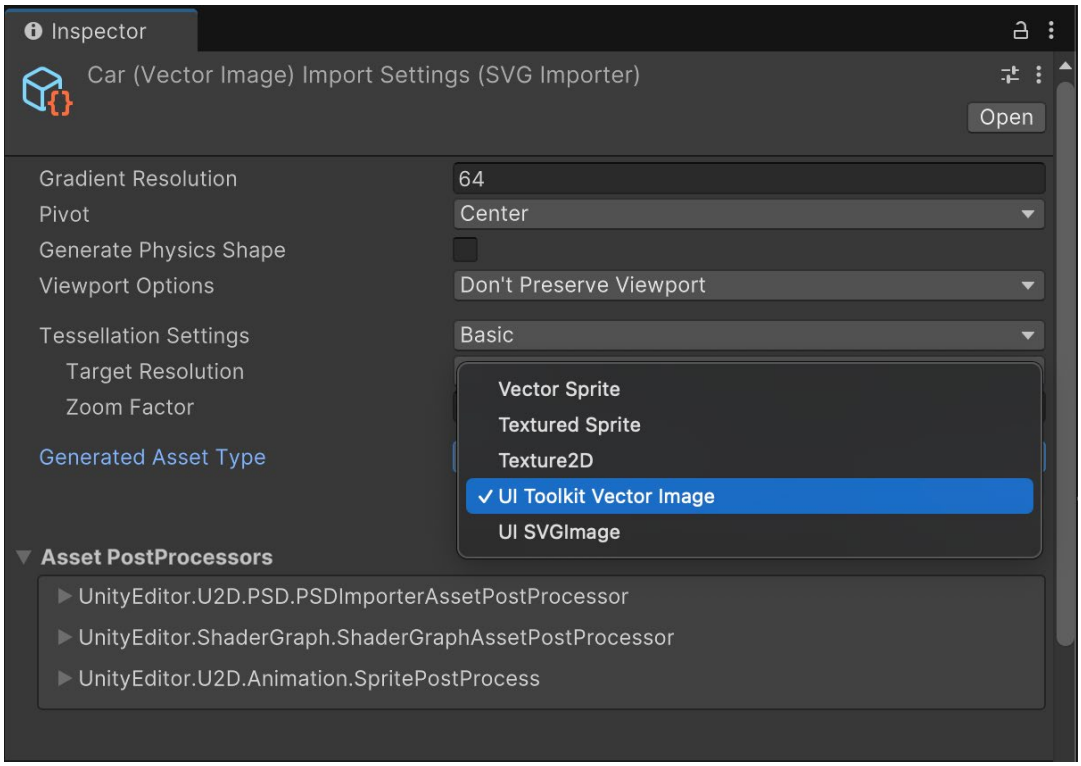
Совет: итеративное проектирование

Unity автоматически обновляет спрайты из PSD-файла при каждом его сохранении. Благодаря этому можно быстро создать временный вариант, дорабатывать его и сразу наблюдать изменения в окне Game. Просмотр результата в контексте без замены файлов и без помощи другого разработчика заметно экономит время и помогает повысить качество работы.

Векторные изображения

На момент написания книги поддержка векторного формата всё ещё разрабатывалась, однако в UI Builder его уже можно было использовать для фоновых изображений. Тем не менее для UI Toolkit пока рекомендуются растровые изображения - спрайты и текстуры.

Чтобы опробовать векторную графику, установите экспериментальный пакет Vector Graphics, который по умолчанию скрыт в Package Manager. Порядок установки приведён в документации. В пакете можно настроить уровень тесселяции при преобразовании векторной графики в полигоны. Чтобы использовать полученный ресурс в UI Toolkit, для Generated Asset Type выберите UI Toolkit Vector Image.



Пакет Vector Graphics позволяет в ограниченном объёме испытать SVG-изображения в игре или интерфейсе.

Сейчас при рендеринге SVG-файлы разбиваются на полигоны, что ограничивает преимущества векторной графики. При сильном увеличении вместо характерных для вектора гладких кривых могут проявиться многоугольные грани. На момент написания книги сглаживание в UI Toolkit ещё не поддерживалось.

Ожидается, что завершённая реализация будет непосредственно работать с настоящими векторными формами и отдельный пакет Vector Graphics больше не потребуется.

Шрифты

UI Toolkit поддерживает типы Font и FontAsset:

- Font. Стандартные форматы шрифтов TTF и OTF поддерживаются ради обратной совместимости, но в фоновом режиме автоматически преобразуются в FontAsset.

- FontAsset. Рекомендуемый формат, позволяющий точно настраивать кернинг, базовую линию и другие характеристики без изменения исходного файла шрифта. Это особенно полезно для выразительных стилизованных шрифтов, часто используемых в играх.

    - FontAsset также позволяет точно управлять созданием атласов: выбирать набор символов, разрешение и способ заполнения. Эти параметры помогают сократить потребление памяти, особенно при работе с Unicode-шрифтами для языков с большим числом символов.

Упаковщики текстур

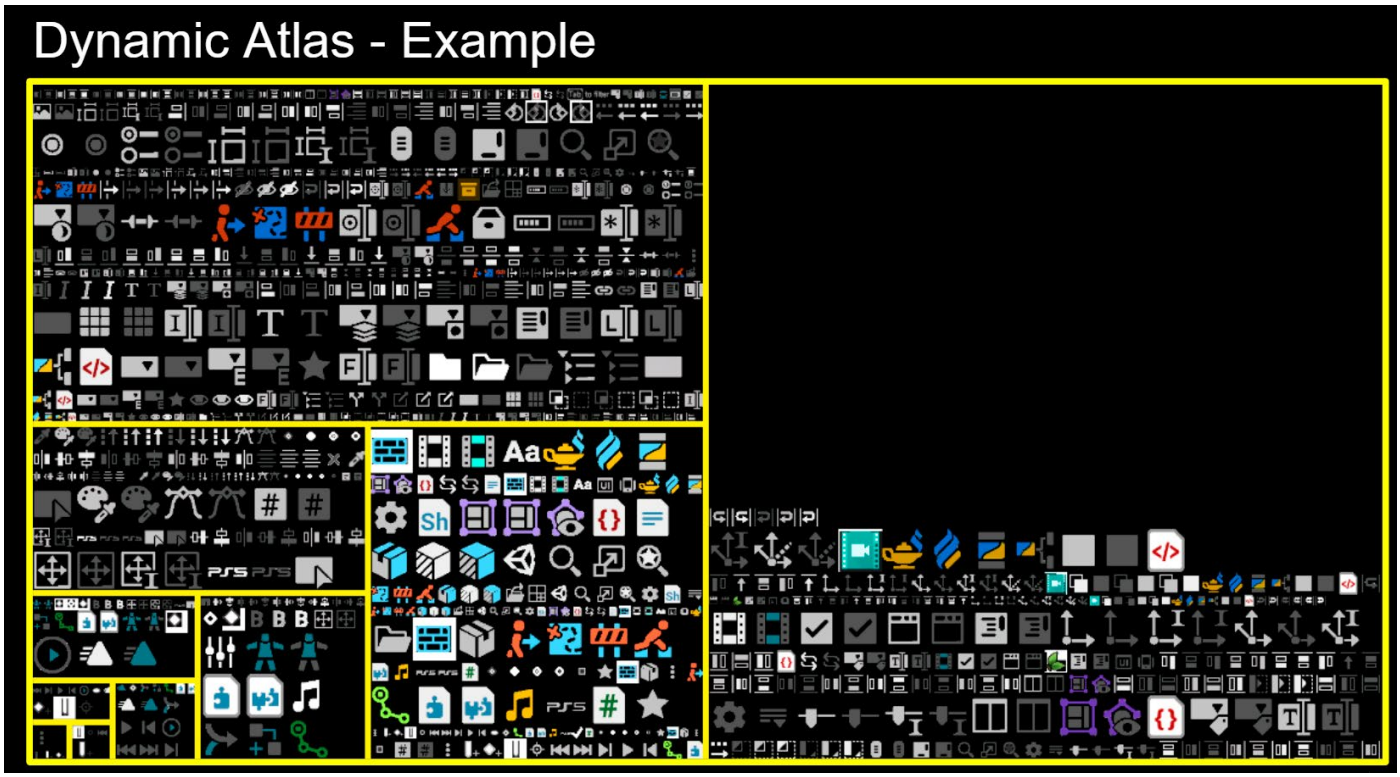
Объединение двухмерной графики в одной текстуре - распространённый способ оптимизации, который уменьшает число вызовов отрисовки и улучшает использование памяти. UI Toolkit поддерживает две системы атласов.

Атлас спрайтов

Типичный атлас игрового интерфейса из UI Toolkit Sample - Dragon Crashers

Sprite Atlas - инструмент Unity для упаковки спрайтов и другой графики двухмерных игр в атлас; его также можно использовать для графики UI. Он автоматически упаковывает ресурсы из одной папки проекта, создавая атласы спрайтов, карт нормалей и масок. Поддерживаются варианты для разных платформ и API для расширенного управления. Sprite Atlas обычно применяют для упаковки ресурсов в редакторе, а не во время выполнения.

Динамический атлас



Динамический атлас, созданный редактором Unity и открытый в Texture Atlas Viewer. Атлас увеличивается по горизонтали и вертикали степенями двойки, не превышая максимально допустимый размер текстуры.

Если графика UI не упакована средствами Sprite Atlas, UI Toolkit автоматически добавляет её в динамический атлас на предварительном проходе.

Изображения, на которые ссылается визуальный элемент, включаются в атлас по критериям, заданным в Panel Settings документа UI. Например, можно указать минимальные и максимальные размеры упаковываемых текстур или фильтровать изображения по другим свойствам. Созданные атласы доступны для просмотра в Texture Atlas Viewer отладчика UI Toolkit.

Динамический атлас работает и во время выполнения, и в редакторе, поэтому удобен для элементов UI, создаваемых динамически, например содержимого инвентаря игрока.

При подготовке графики придерживайтесь следующих рекомендаций:

- Уже на этапе создания макетов задайте в графическом редакторе максимальное целевое разрешение, чтобы впоследствии не переделывать работу. Например, если вы планируете поддерживать графику вплоть до 4K, используйте это разрешение как минимальное рабочее.

- Не увеличивайте растровые изображения после их создания: это приводит к пикселизации и размытию, снижая визуальное качество. Начинайте работу с максимального поддерживаемого разрешения и уменьшайте изображение при экспорте из графического приложения.

- При работе с векторной графикой последующее изменение размера ресурсов не так критично. Тем не менее используйте эталонное разрешение (Reference Resolution), чтобы сохранять правильный относительный масштаб каждого ресурса, например одинаковую толщину контура элементов.

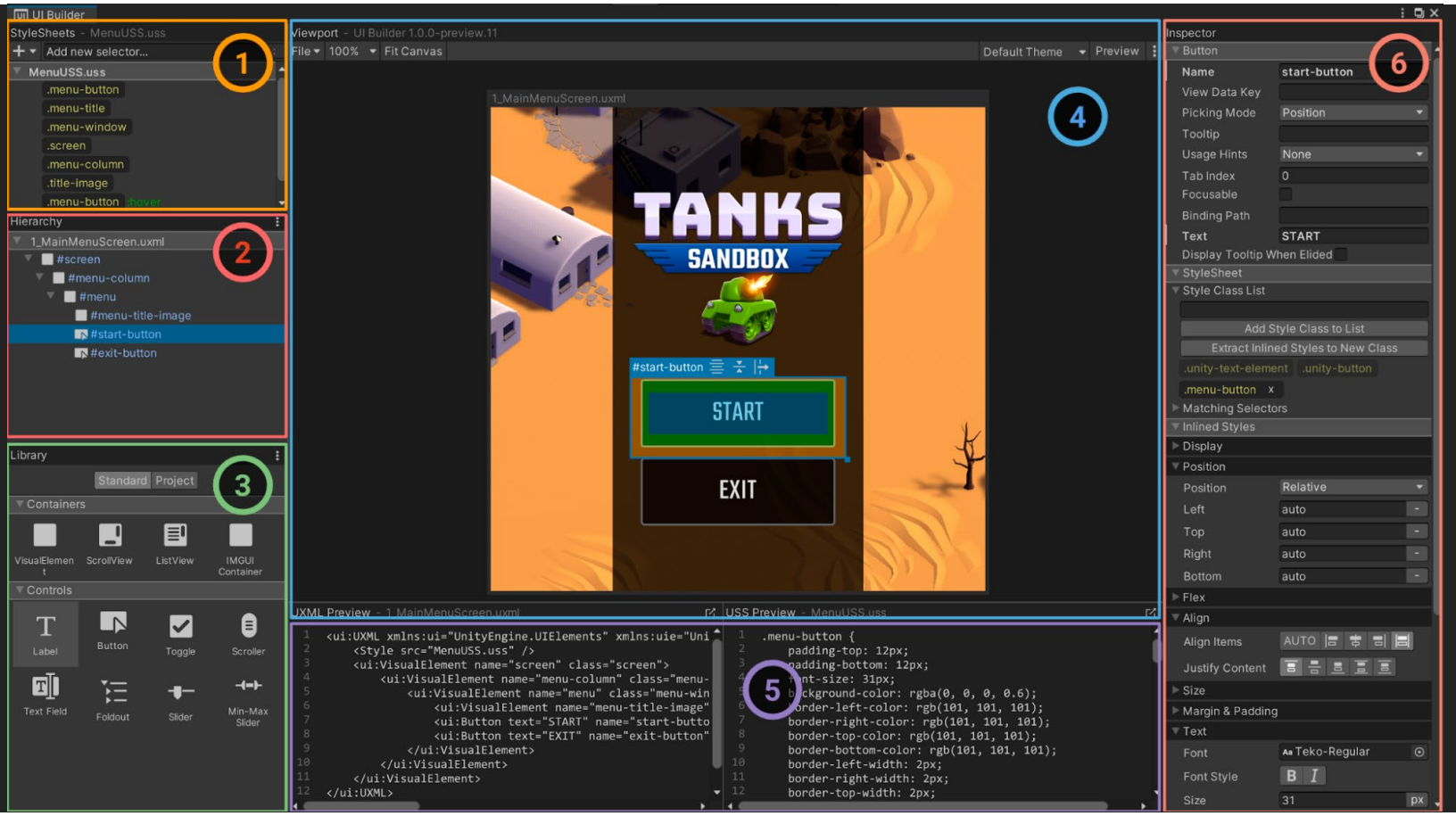
- Если разрешение графических ресурсов ниже требуемого, попробуйте 2D Enhancers и функцию масштабирования на основе ИИ в Sprite Editor.

- Используйте возможности 2D PSD Importer, импортируя PSD-файлы прямо в Unity. После сохранения PSD все изменения слоёв автоматически отразятся в Unity. Кроме того, находящийся в проекте PSD-файл можно хранить в системе контроля версий.

- Автоматизируйте импорт. Не изменяйте параметры вручную при каждом добавлении графического ресурса. С помощью Preset можно сохранить настройки одного ресурса и автоматически применять их ко всем ресурсам того же типа в заданной папке.

- Для более глубокой автоматизации, например проверки ресурсов или массового изменения их параметров, используйте API AssetPostprocessor.

UI Builder



UI Builder открывается через меню Window > UI Toolkit > UI Builder.

UI Builder позволяет создавать, просматривать и изменять файлы UXML и USS в визуальном интерфейсе, встроенном в основной редактор. Рассмотрим ключевые области UI Builder:

1. StyleSheets. Здесь находятся правила компоновки и оформления, также называемые селекторами USS. Они позволяют совместно использовать стили в разных UXML-документах и элементах UI.

2. Hierarchy. Как и окно Hierarchy сцены, эта область показывает иерархию визуальных элементов UXML-документа.

3. Library. Здесь собраны готовые стандартные и пользовательские элементы управления: кнопки, подписи, ползунки и другие компоненты, которые можно добавить в иерархию. Отсюда же в текущий UXML можно вставить другой UXML-документ как шаблон.

4. Viewport. Эта область показывает внешний вид интерфейса. Элементы на холсте можно редактировать непосредственно с помощью экранных манипуляторов.

5. Code Previews. Здесь отображается код, который UI Builder формирует в фоновом режиме для документа UI (UXML) и таблиц стилей (USS). Чтобы увидеть код целиком, возможно, потребуется изменить размер окна.

6. Inspector. Позволяет изменять атрибуты и свойства стиля выбранного элемента или селектора USS.

Совет: сохранение ресурсов UI

В UI Builder сохраняйте изменения через меню области Viewport: File > Save. Команда сохраняет все открытые файлы UXML и USS.

В отличие от Unity UI, при работе с UI Toolkit игра может выполняться в редакторе одновременно с внесением изменений. Звёздочка * рядом с именем файла в заголовке холста UI Builder означает, что изменения ещё не сохранены.

Фон холста

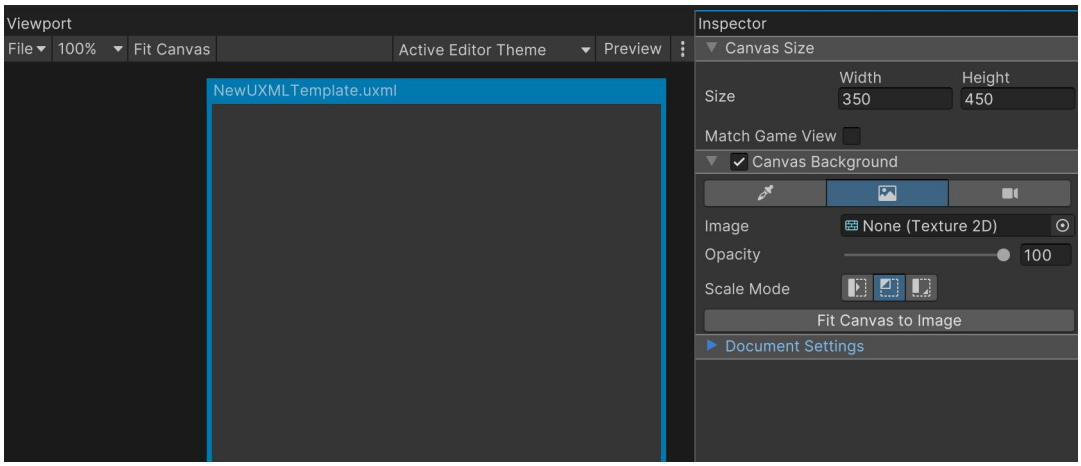
Включив фон холста, можно оценить оформление элементов на заданном цвете или фоновом изображении. Выберите UXML-файл в области Hierarchy, затем настройте фон холста так, чтобы он напоминал итоговый интерфейс: это позволит оценивать изменения стиля в контексте.

Для фона Canvas доступно несколько вариантов:

- Background Color. Задаёт оттенок или цвет игрового окружения.

- Image. Позволяет выбрать спрайт или текстуру в качестве фона, что удобно для воспроизведения макетов экранов или референсной графики.

- Camera. Показывает текущий игровой процесс на фоне, позволяя оценивать UI непосредственно в контексте игры.



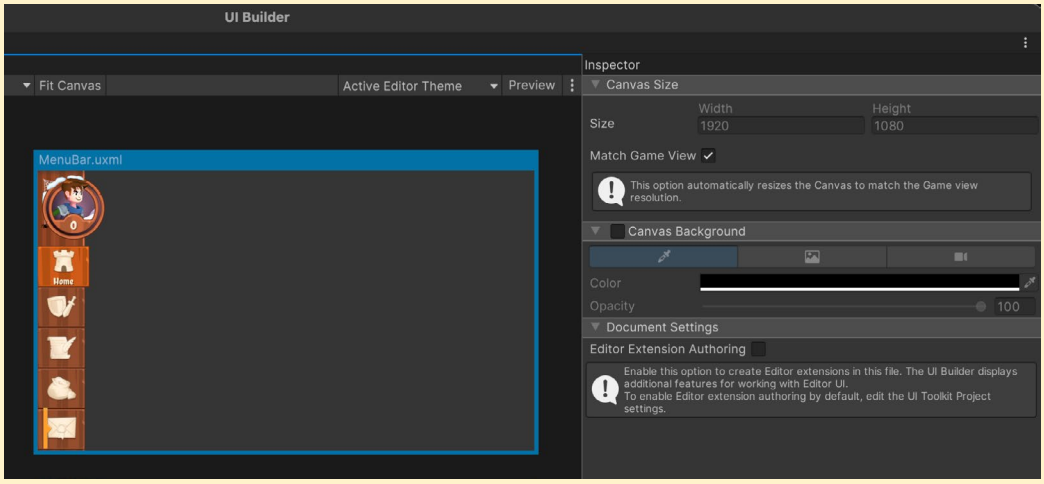
Холст нового UXML-документа. Его внешний вид настраивается с помощью параметров Color и Image.

Настройки области просмотра

Для навигации по рабочей области изменяйте масштаб в пределах 25-500% или выберите Fit Canvas, чтобы он автоматически подстроился под доступное место на экране.

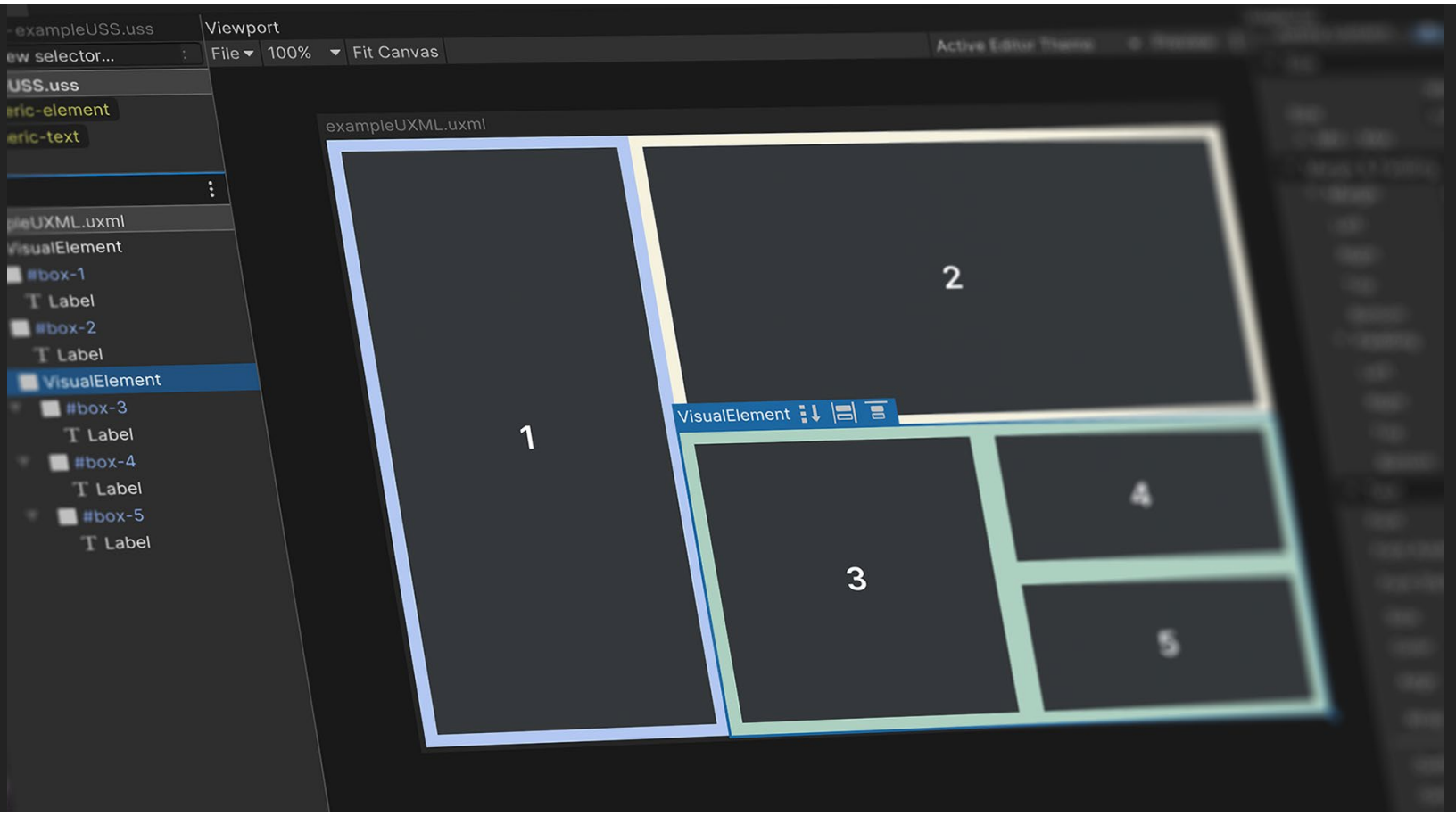
Режим Preview позволяет просматривать UI, не рискуя случайно изменить выбранные элементы. В активной области Viewport также можно отображать стили для определённых событий мыши, например наведения и получения фокуса.

Совет: соответствие окну Game и предварительный просмотр тем



Чтобы приблизить отображение к UI во время выполнения, выберите загруженный UI Document (UXML) в Hierarchy и включите Match Game View. Размер области Viewport будет приведён к эталонному разрешению проекта. Изменение этого параметра влияет только на предварительный просмотр, но не на сами файлы UI. В UI Builder также можно заранее просматривать различные темы проекта; подробнее эта возможность рассматривается далее.

Макеты



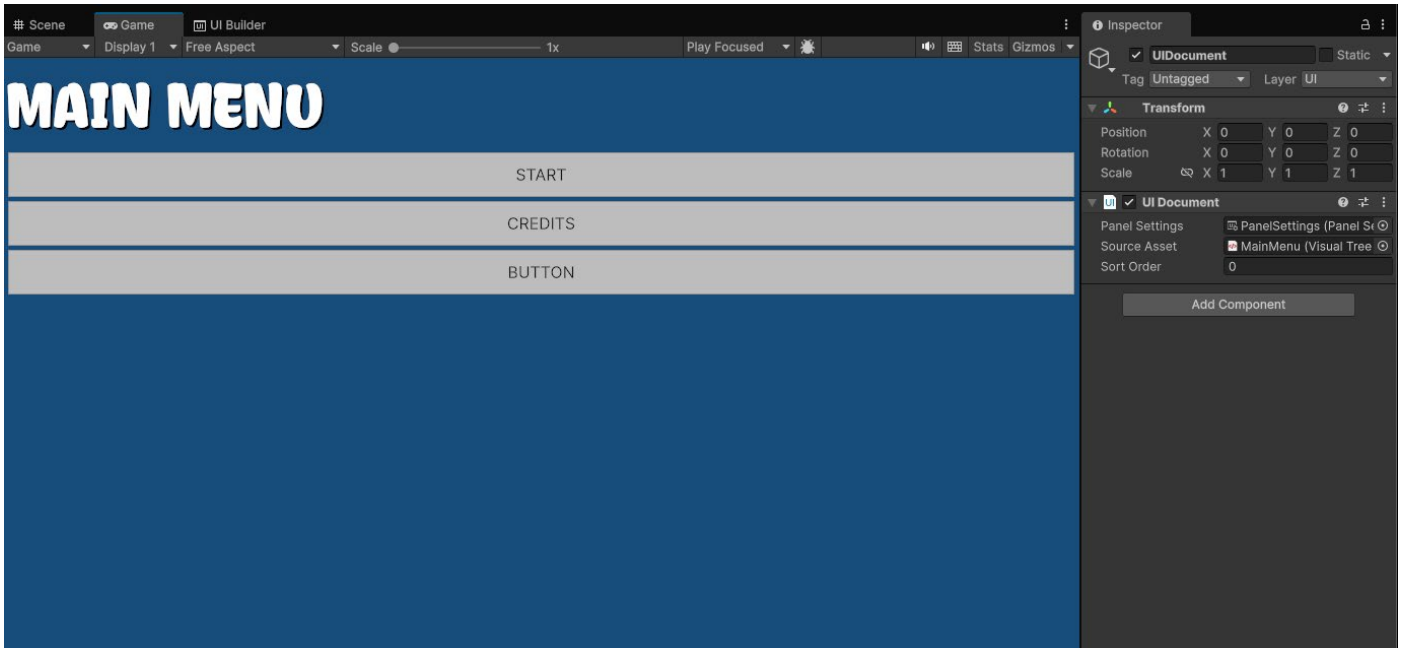
UI Builder предоставляет все инструменты, необходимые для создания адаптивного макета.

В этом разделе рассматриваются основные этапы создания макетов в UI Builder.

UI Builder - наглядный WYSIWYG-инструмент, удобный для дизайнеров и позволяющий эффективно создавать файлы UXML и USS без ручного написания кода. Некоторые команды предпочитают формировать UI непосредственно в коде, однако UI Builder даёт художникам творческий контроль и заметно совершенствует рабочий процесс. При внесении изменений UI Builder генерирует код автоматически: всё созданное в нём можно напрямую представить кодом UXML и USS.

Одно из главных преимуществ UI Toolkit и UI Builder - эффективное создание адаптивных макетов. Они необходимы любой игре, предназначенной для нескольких платформ с разными разрешениями и соотношениями сторон экрана. В этом разделе рассматриваются основные этапы создания таких макетов в UI Builder.

Ниже показан UXML-файл и его код в панели UXML Preview окна UI Builder. Чтобы создать ресурс в UI Builder, выберите File > New, а затем Save As.



Щёлкните значок в правом верхнем углу окна Code Preview, чтобы открыть код в своей IDE.

В этом примере кода UXML визуальные элементы представлены разметкой, напоминающей HTML, с открывающими и закрывающими угловыми скобками. Например, так выглядит синтаксис кнопки Start:

```
<engine:Button text="START" name="start-game__button" />
```

Основные компоненты времени выполнения

Элементы UI Toolkit не отображаются в окне Scene. Создаваемый интерфейс виден в UI Builder, но более точный предварительный просмотр с целевым разрешением даёт окно Game. Чтобы вывести UI в окне Game, на GameObject должен находиться компонент UI Document с ресурсами Panel Settings и Visual Tree (UXML), как показано на снимке:

Компонент UI Document определяет, какой UXML будет отображаться, и по умолчанию содержит ресурс Panel Settings. Поле Sort Order задаёт порядок этого документа относительно других UI Document, использующих те же Panel Settings.

Добавьте компонент на GameObject через меню Add Component в инспекторе или щёлкните правой кнопкой мыши в Hierarchy и выберите UI Toolkit > UI Document - ресурс Panel Settings будет назначен автоматически.

Ресурс Panel Settings определяет, как компонент UI Document создаётся и отображается во время выполнения. В проекте может быть несколько Panel Settings для разных вариантов UI. Например, HUD и мини-карта могут использовать собственные Panel Settings.

Создайте ресурс через Assets > Create > UI Toolkit > Panel Settings Asset. Он появится в корневой папке проекта, после чего его можно назначить компоненту UI Document на GameObject.

Адаптивные макеты: Flexbox

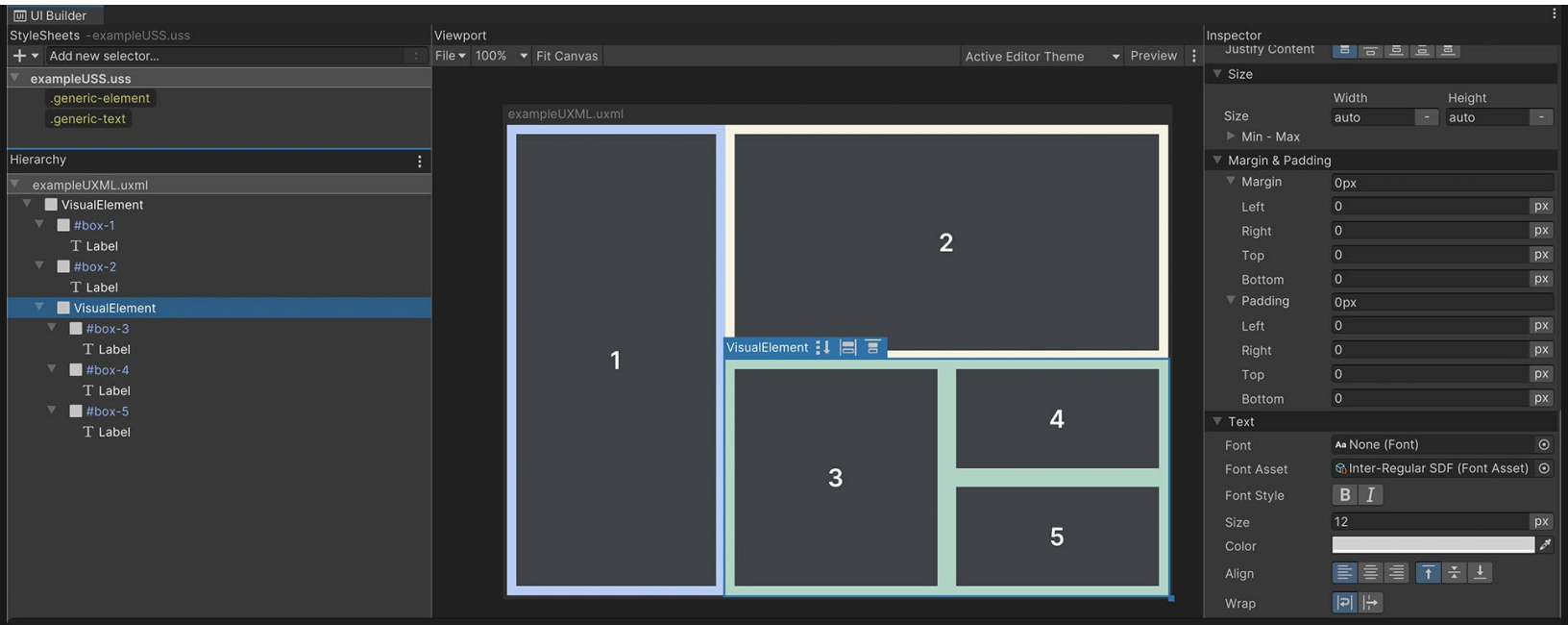
UI Toolkit позиционирует визуальные элементы с помощью Yoga - механизма компоновки HTML/CSS, реализующего подмножество Flexbox. Если вы ещё не знакомы с Yoga и Flexbox, эта глава поможет освоить принципы, лежащие в основе системы компоновки UI Toolkit.

Flexbox, или Flexible Box Layout, - способ размещать элементы по строкам или столбцам. Эта система отлично подходит для сложных, хорошо организованных макетов. Рассмотрим несколько её преимуществ:

- Адаптивный UI. Flexbox организует содержимое в сеть блоков или контейнеров. Элементы можно вкладывать друг в друга как родительские и дочерние и размещать на экране по простым правилам. Дочерние элементы автоматически реагируют на изменения родительских контейнеров. Адаптивный макет подстраивается под разные размеры и разрешения экрана, упрощая поддержку нескольких платформ.

- Упорядоченная сложность. Стили задают простые правила, управляющие внешним видом визуального элемента. Один стиль можно одновременно применить к сотням элементов, а изменения сразу отразятся во всём UI. Благодаря этому дизайн интерфейса строится на единообразных повторно используемых стилях, а не на ручном оформлении каждого элемента.

- Разделение логики и дизайна. Макеты и стили UI отделены от кода. Дизайнеры и разработчики могут работать параллельно, не нарушая зависимости, и каждый сосредотачивается на своей специализации.



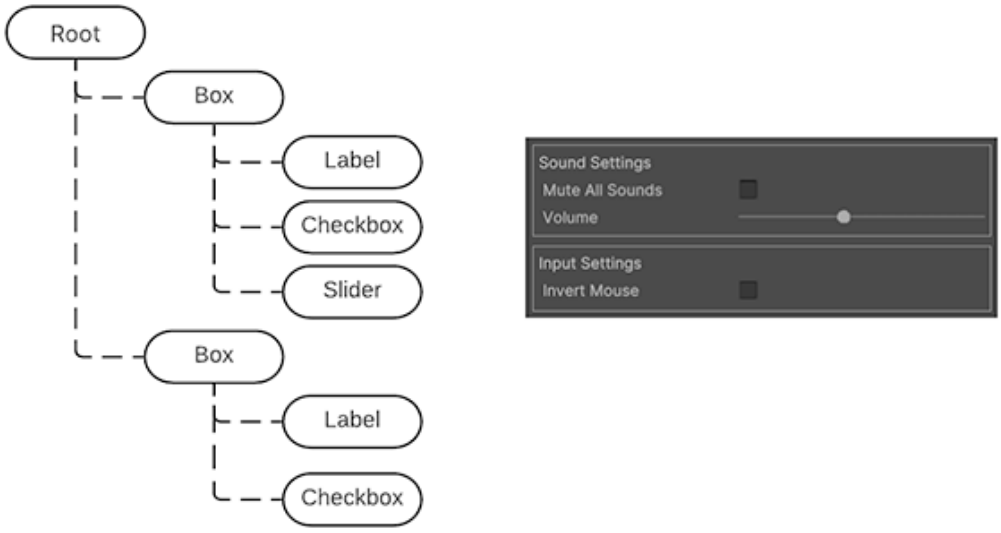

```
//Add logic that interacts with the UI controls in the `OnEnable` methods
private void OnEnable()
{
    // The UXML is already instantiated by the UIDocument component
    var uiDocument = GetComponent<UIDocument>();
    _box1 = uiDocument.rootVisualElement.Q<VisualElement>("box-1");
    _label = uiDocument.rootVisualElement.Q<Label>("Label");
```

Разделение логики и дизайна. Программисты связывают визуальные элементы с игровой логикой, а дизайнеры сосредотачиваются на определении их стилей.

Визуальные элементы

В UI Toolkit основными строительными блоками каждого интерфейса служат визуальные элементы. VisualElement - базовый класс всех элементов UI Toolkit: кнопок, изображений, текста и других компонентов. Их можно считать аналогом GameObject в UI Toolkit.

Иерархия UI из одного или нескольких визуальных элементов называется визуальным деревом (Visual Tree).



Упрощённая иерархия визуального дерева и соответствующий ей интерфейс справа

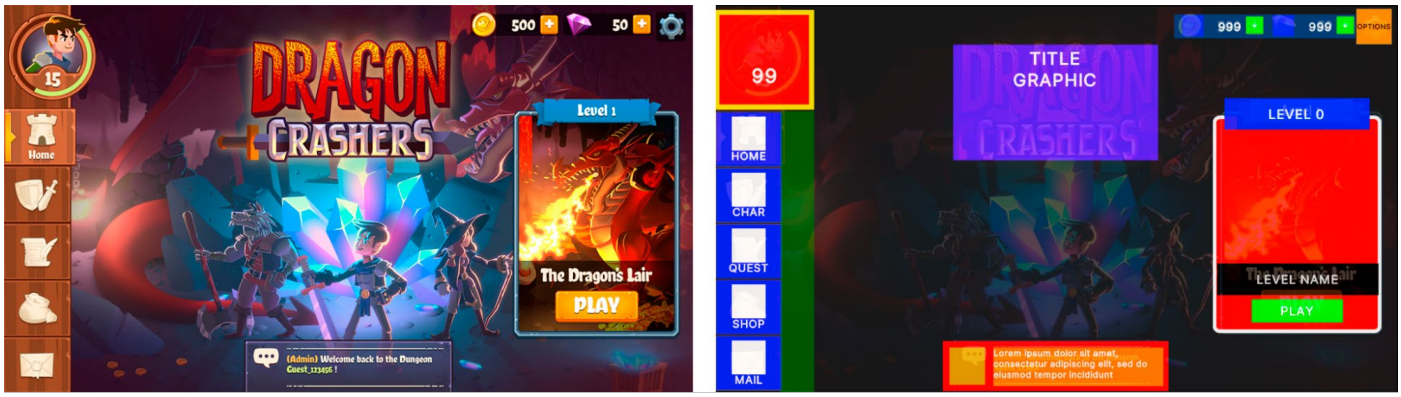
Комбинации визуальных элементов хранятся в UXML-файлах. Они содержат сведения об иерархии, макете элементов и их стилях, если те не вынесены в StyleSheet или USS.

Прежде чем углубляться в UI Toolkit, необходимо освоить основы компоновки Flexbox. Их удобно изучать на простых визуальных элементах в UI Builder.

Позиционирование визуальных элементов

При создании макета UI рассматривайте каждый экран как отдельную группу визуальных элементов. Продумайте, на какие блоки его разбить, как расположить эти блоки по горизонтали или вертикали и нужны ли вложенные блоки для дальнейшей организации информации.

В примере ниже один большой визуальный элемент может служить контейнером для панели меню и её содержимого слева. Каждую кнопку следует представить отдельным дочерним визуальным элементом.



Рекомендуется подготовить подробный макет или каркас (слева), а затем выделить и разметить элементы, из которых этот дизайн будет собран в UI Toolkit (справа).

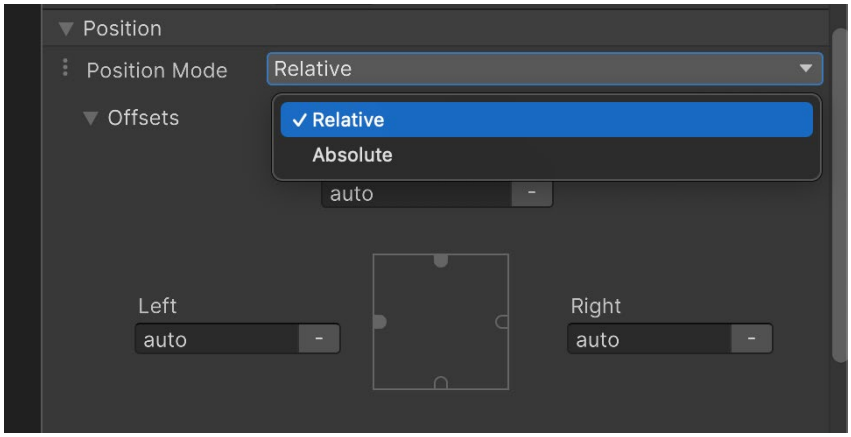
UI Builder поддерживает два способа позиционирования визуальных элементов:

- Относительное позиционирование. Используется по умолчанию для новых визуальных элементов. Дочерние элементы подчиняются правилам Flexbox родительского контейнера. Например, если Direction родителя задан как Row, дочерние визуальные элементы располагаются слева направо. При относительном позиционировании размеры и положение элементов динамически зависят от следующих факторов:

  - Размер и правила стиля родительского элемента. Если изменить параметры родителя, например Padding или Align > Justify Content, дочерние элементы соответствующим образом перестроятся.

  - Собственные размеры и правила стиля дочернего элемента. Если для дочернего визуального элемента заданы минимальные или максимальные размеры, система компоновки также старается их соблюдать.

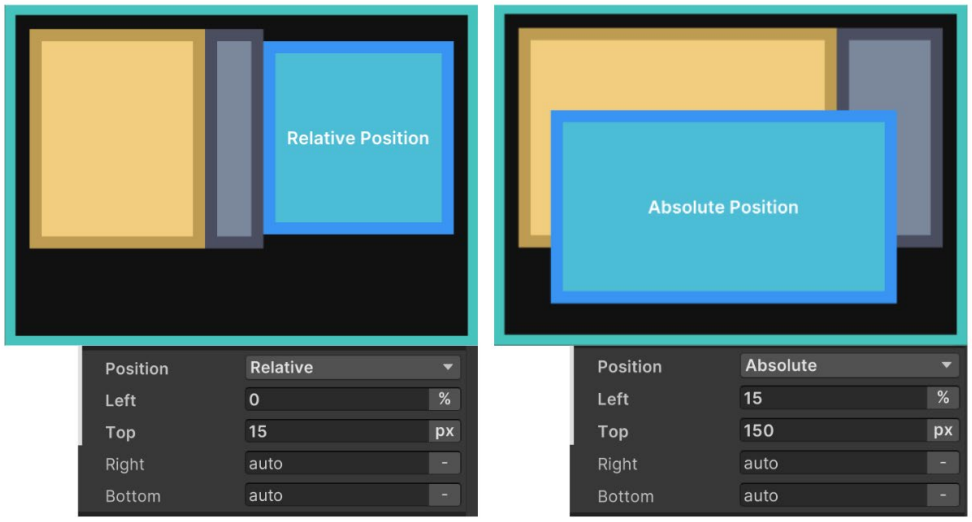
UI Toolkit разрешает возникающие противоречия между параметрами родительского и дочернего элементов. Например, если минимальная ширина дочернего элемента больше ширины контейнера, содержимое выйдет за его границы.



Режимы позиционирования, доступные для любого визуального элемента

- Абсолютное позиционирование. Положение визуального элемента привязывается к родительскому контейнеру, как при работе Unity UI с Canvas. Правила размера и параметры, влияющие на дочерние элементы, продолжают действовать, но сам элемент накладывается поверх родительского контейнера и игнорирует такие параметры Flex, как Grow, Shrink и Margins.

    Для привязки абсолютно позиционированного элемента можно использовать Left, Top, Right и Bottom. Например, нулевые значения Right и Bottom закрепят Button в правом нижнем углу родительского контейнера.



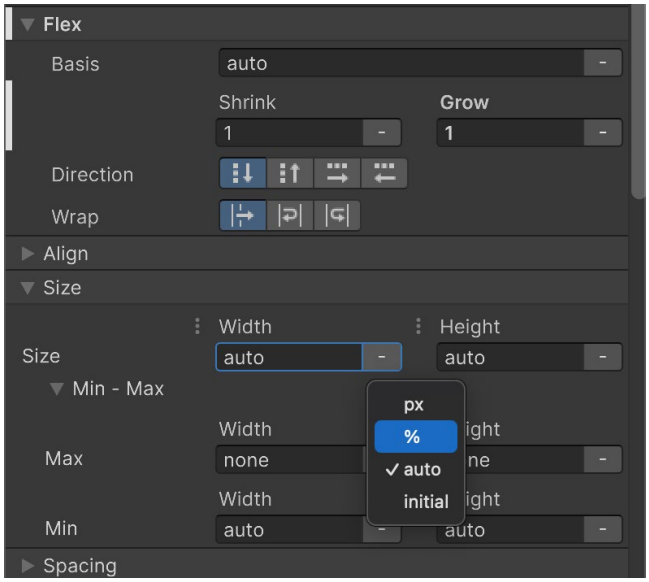
Слева синий визуальный элемент использует Relative, а у его родителя для Flex задано Direction: Row. Справа синий визуальный элемент использует Absolute и игнорирует правила Flexbox родителя.

Относительное позиционирование обычно подходит для постоянно видимых элементов, сложных групп и контейнеров с множеством дочерних элементов.

Абсолютное удобно для временных интерфейсов, например всплывающих окон, декоративных элементов, не влияющих на общую компоновку, и UI, следующего за игровыми объектами, например индикатора здоровья персонажа.

Параметры размера

Помните, что визуальные элементы представляют собой контейнеры. В Unity 6 значение Grow по умолчанию равно 1, поэтому элемент занимает всё свободное место в контейнере. В иных случаях он не занимает пространства, если только внутри нет дочерних элементов с заданным размером или самому элементу не назначены определённые Width и Height.



Параметры размера визуального элемента

Поля Width и Height определяют размер элемента. Max Width и Max Height ограничивают его расширение, а Min Width и Min Height - сжатие. Вместо значения auto по умолчанию размеры можно задать в пикселях или процентах. Эти параметры влияют на то, как настройки Flex, описанные ниже, изменяют размер элементов в зависимости от свободного пространства.

Параметры Flex

При относительном позиционировании параметры Flex могут изменять размер элемента. Рекомендуется поэкспериментировать с ними, чтобы на практике понять их действие.

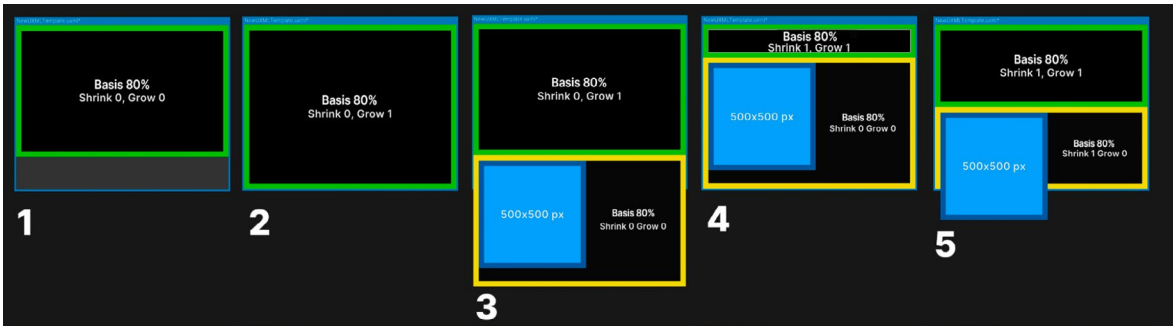
Basis задаёт исходные Width и Height до применения коэффициентов Grow и Shrink:

- Если Grow равен 1, элемент займёт всё доступное вертикальное или горизонтальное пространство родителя. При значении 0,5 он займёт половину свободного пространства.

- Если Grow равен 0, элемент не увеличивается сверх текущего Basis, то есть исходного размера.

- Если Shrink равен 1, элемент сжимается настолько, насколько требуется, чтобы поместиться в свободном пространстве родителя.

- Если Shrink равен 0, элемент не сжимается и при необходимости выходит за границы контейнера.



Параметры Basis, Grow и Shrink

Пример выше показывает совместную работу Basis, Grow и Shrink:

1. Зелёный элемент с Basis 80% занимает 80% доступного пространства.

2. При Grow 1 зелёный элемент расширяется на всё свободное пространство.

3. После добавления жёлтого элемента содержимое выходит за границы контейнера, а зелёный элемент вновь занимает 80% пространства.

4. При Shrink 1 зелёный элемент сжимается, освобождая место для жёлтого.

5. Здесь Shrink обоих элементов равен 1, поэтому они сжимаются в равной степени и вместе помещаются в доступном пространстве.

Как видно из примера, элементы с фиксированным размером в пикселях - синий блок на шагах 3-5 - не реагируют на параметры Basis, Grow и Shrink.

Совет: расчёт размера визуального элемента

При относительном позиционировании система компоновки объединяет параметры Size и Flexbox, чтобы определить итоговый размер каждого элемента. Расчёт выполняется следующим образом:

1. Система компоновки рассчитывает размер элемента по свойствам Width и Height.

2. Затем она проверяет, осталось ли в родительском контейнере свободное место или дочерние элементы уже выходят за доступные границы.

3. Если свободное место есть, система находит элементы с ненулевым Flex > Grow и распределяет между ними дополнительное пространство пропорционально этому коэффициенту, увеличивая дочерние элементы.

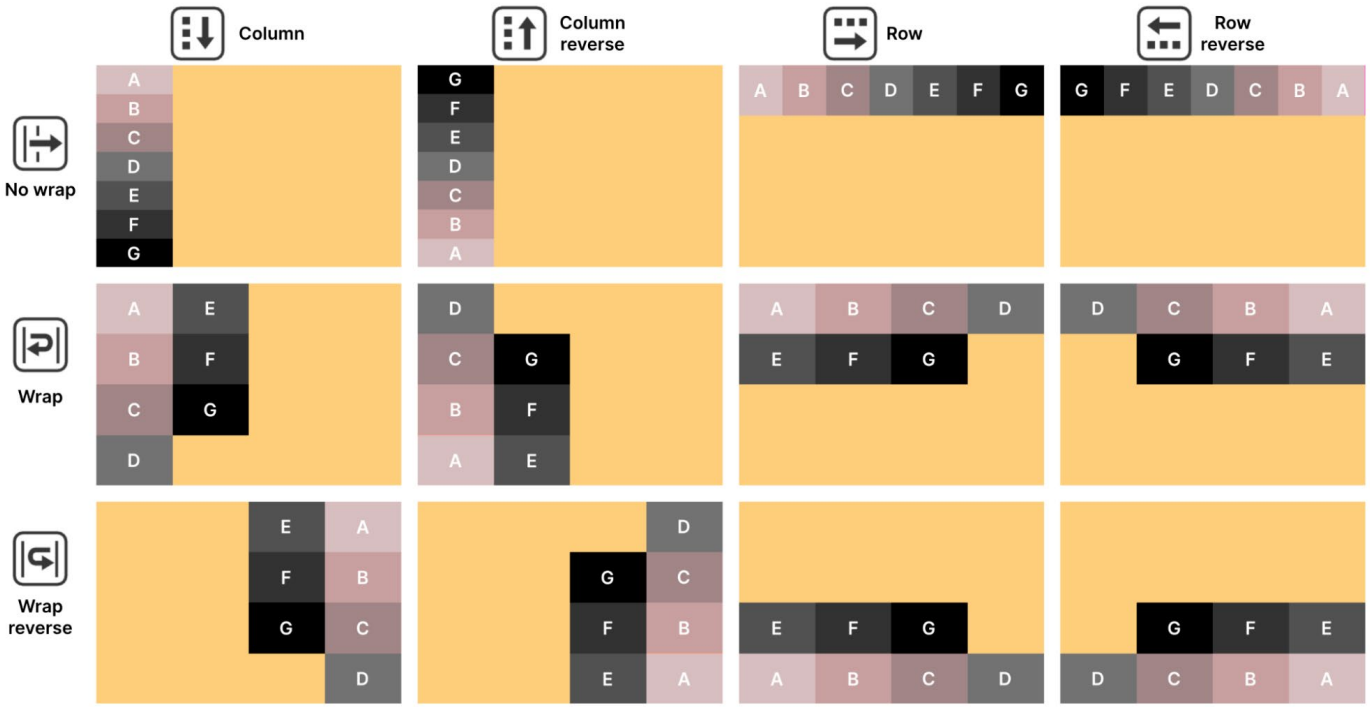
4. Если дочерние элементы не помещаются, элементы с ненулевым Flex > Shrink соответствующим образом уменьшаются.

5. Затем учитываются прочие свойства, влияющие на итоговый размер элемента, например Min Width и Flex Basis.

6. Система компоновки применяет рассчитанный итоговый размер.

Параметр Direction определяет порядок дочерних элементов внутри родителя. Элементы, расположенные выше в Hierarchy, выводятся первыми, а находящиеся в конце - последними.

Параметр Wrap указывает, следует ли пытаться уместить элементы в одной строке или одном столбце (No Wrap) либо переносить их в следующую строку или столбец (Wrap или Wrap Reverse).



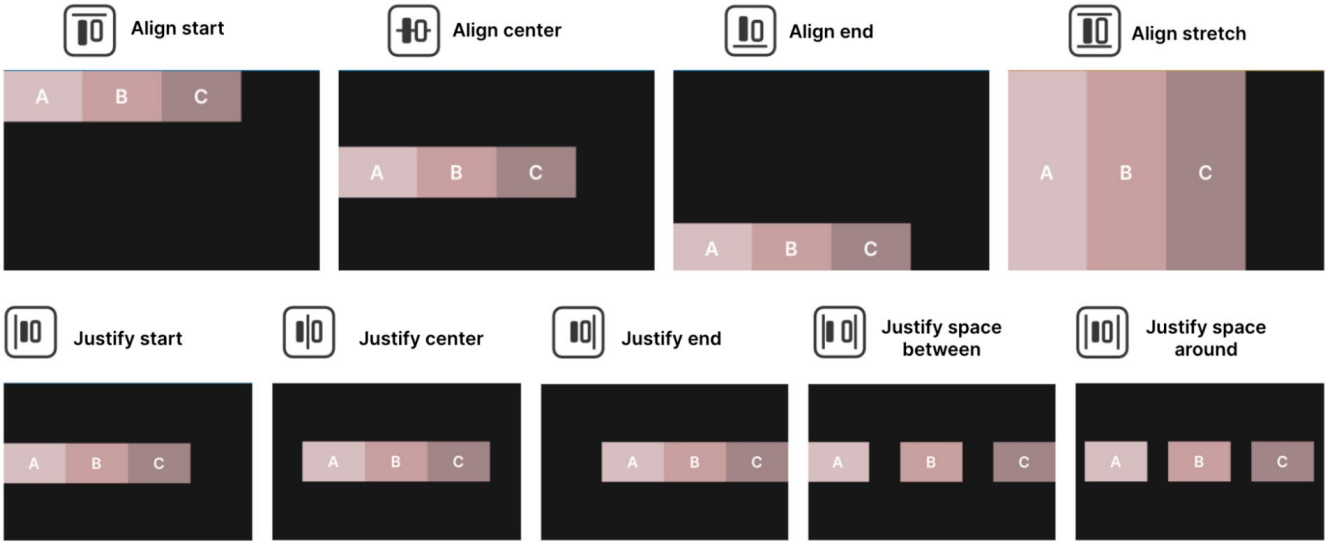
Родительские и дочерние визуальные элементы в UI Builder с относительным позиционированием и разными сочетаниями Direction и Wrap

Параметры выравнивания

Параметры Align определяют выравнивание дочерних элементов относительно родителя. В родительском элементе задайте Align > Align Items, чтобы прижать дочерние элементы к началу, центру или концу. Эти варианты действуют вдоль поперечной оси, перпендикулярной строке или столбцу из Flex > Direction. Stretch также работает по поперечной оси, но его эффект могут ограничивать минимальные и максимальные размеры; этот вариант используется по умолчанию.

Значение Auto позволяет системе компоновки автоматически выбрать один из остальных вариантов с учётом других параметров. Для более точного управления макетом рекомендуется явно выбирать нужный режим, оставляя Auto для особых случаев.

Параметр Align > Justify Content определяет распределение дочерних элементов внутри родителя. Они могут располагаться вплотную друг к другу либо распределяться по доступному пространству. На результат также влияют Flex > Grow и Flex > Shrink.

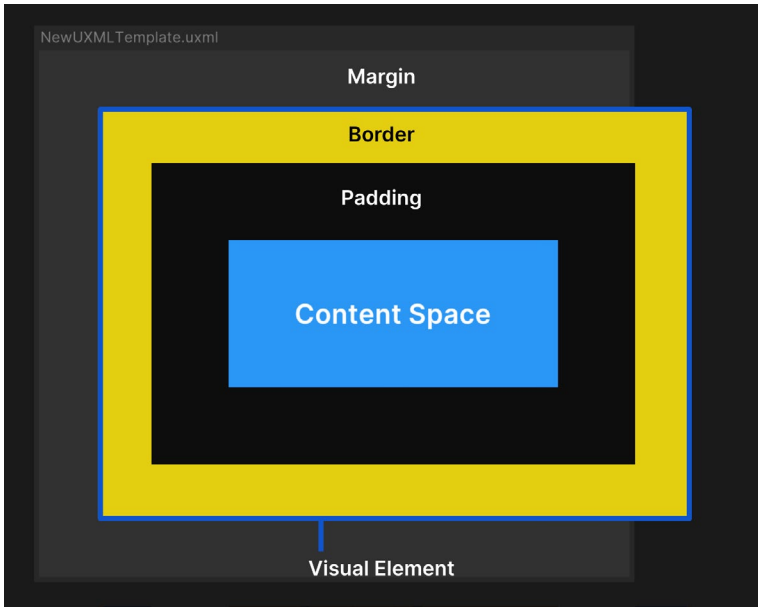


Параметры Align и Justify применены к родительскому элементу с Direction: Row. Обратите внимание: на итоговый результат могут влиять и другие параметры положения и размера.

Параметр Align Self позволяет самому контейнеру выровняться по центру, началу или концу Flex-макета.

Внешние и внутренние отступы

Параметры Margin и Padding задают свободное пространство вокруг визуальных элементов и их содержимого. Unity использует разновидность стандартной блочной модели CSS, представленную на схеме ниже.



Визуальный элемент в UI Builder с заданными Size, Margin, Border и Padding. Элементы с фиксированными Width или Height могут выходить за границы доступного пространства.

- Content Space содержит основные визуальные компоненты: текст, изображения, элементы управления и т. д.

- Padding задаёт пустую область вокруг Content Space, но внутри Border.

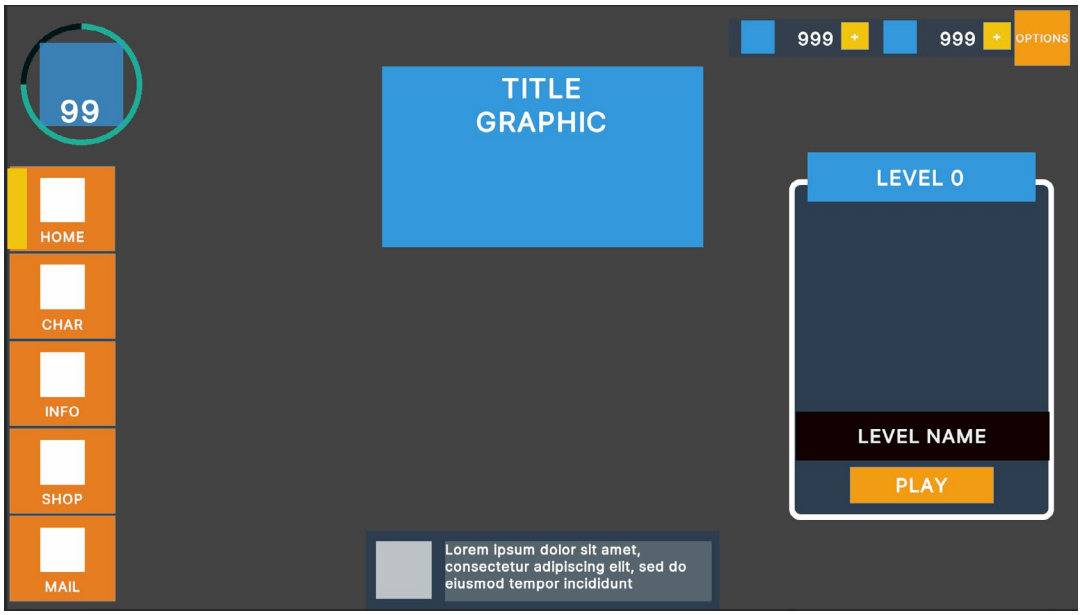
- Border образует границу между Padding и Margin. Ей можно назначить цвет и скругление. При заданной толщине Border расширяется внутрь.

- Margin, как и Padding, задаёт свободную область, но уже снаружи Border. Для элементов с Position: Absolute параметры Margin не действуют; внешнее пространство относительно точки привязки можно добавить параметрами Position.

Фон и изображения

В UI Toolkit любой визуальный элемент может выводить изображение на экран. Достаточно задать свойство background и назначить текстуру или спрайт.

Цвет или изображение фона меняет внешний вид элемента и особенно удобно при создании каркаса. Яркие контрастные цвета помогают увидеть, как соседние элементы сочетаются и реагируют на изменения контейнеров.



На этапе создания каркаса используйте контрастные цвета.

Относительные и фиксированные единицы измерения

В UI Builder расстояния и размеры элементов задаются четырьмя вариантами:

- Auto. Значение по умолчанию для размера и положения. Система компоновки рассчитывает параметры элемента по сведениям о родителе и дочерних элементах.

- Percentage. Единица равна процентной доле контейнера и динамически меняется вместе с Width и Height родителя. Проценты помогают масштабировать макет для экранов разных форматов.

- Pixels. Подходит для фиксированного размера. Например, небольшим элементам можно задать минимальный размер в пикселях, чтобы они всегда оставались читаемыми.

- Initial. Возвращает свойство в исходное состояние, заданное стандартными правилами стилей Unity, и игнорирует текущие значения оформления.

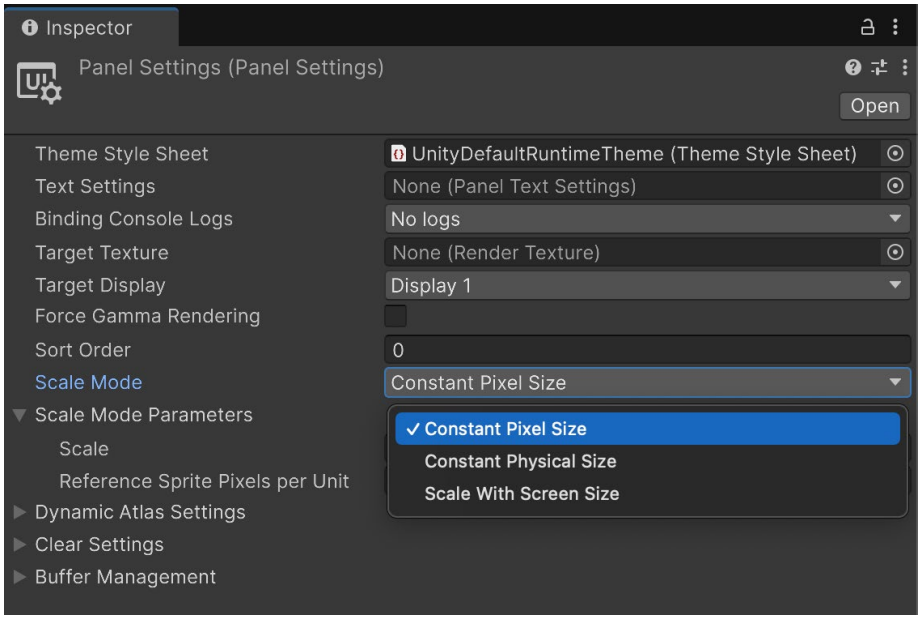
Примеры стандартных параметров Size и размеров, заданных в пикселях и процентах

Чтобы применить единое правило масштабирования ко всему UI, используйте параметры Scale Mode в Panel Settings:

- Constant Pixel Size. Элементы сохраняют фиксированный размер в пикселях независимо от размеров экрана. Для пропорционального изменения всех размеров можно задать Scale Factor.

- Constant Physical Size. Элементы сохраняют одинаковый физический размер на разных экранах. Система масштабирует UI относительно Reference DPI, корректируя размер при отличии фактического DPI экрана.

- Scale with Screen Size. Размеры элементов динамически меняются в зависимости от разрешения. Screen Match Mode определяет приоритет ширины, высоты или их сочетания, а Reference Resolution задаёт базовый размер UI. Если в Screen Match Mode выбран Match Width or Height, значение Match определяет, будет ли UI подстраиваться под ширину экрана, его высоту или промежуточное соотношение.

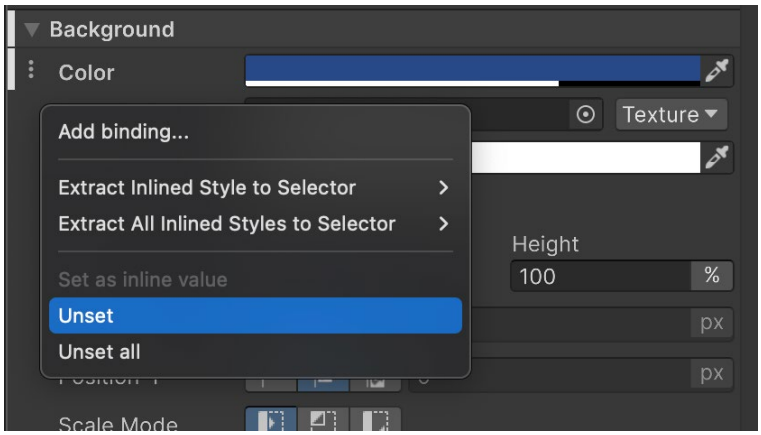


В Panel Settings системы UI Toolkit доступны параметры масштабирования, похожие на параметры Unity UI.

Переопределённые свойства в UI Builder

Изменённые свойства выделяются в инспекторе жирным шрифтом и белой линией, как на снимке ниже. Это означает, что они переопределяют стандартные значения или значения селектора из таблицы стилей USS выбранного UXML-файла. Такое оформление также называют встроенным стилем (inline styling).

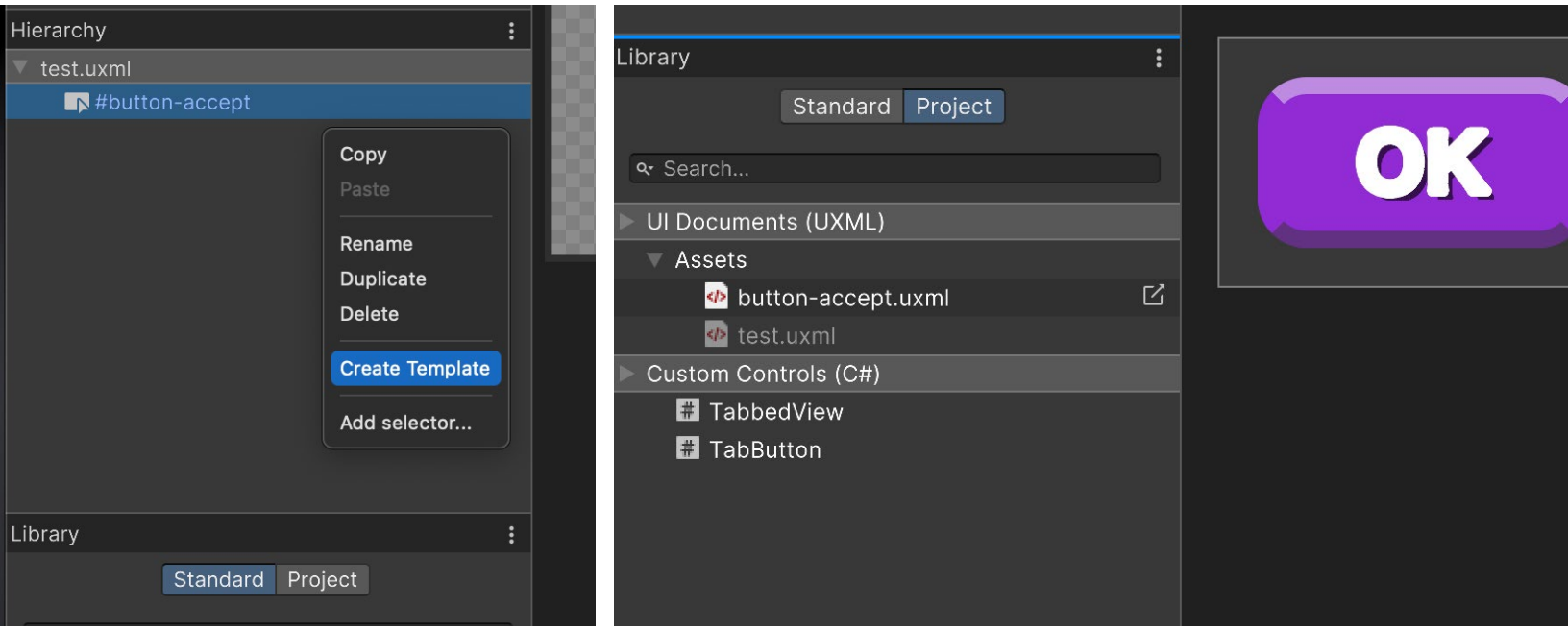
Если значение менять не требуется, лучше оставить его в исходном состоянии: так переопределения легче находить и контролировать. Чтобы сбросить свойство, воспользуйтесь соответствующей командой в меню с вертикальным многоточием (⋮) рядом с разделом свойств.



В этом меню изменённое значение можно вернуть к стандартному или к исходному значению селектора.

UXML как шаблоны

UXML-файлы можно использовать подобно префабам. Например, в проекте может быть UXML-макет со значком предмета и счётчиком количества, который нужно многократно создавать в инвентаре. Щёлкнув правой кнопкой мыши любой UXML, можно создать Template. Затем его можно добавить к другому визуальному элементу в области Hierarchy или создать экземпляр из кода. Готовый шаблон появится в Library и окне Project.



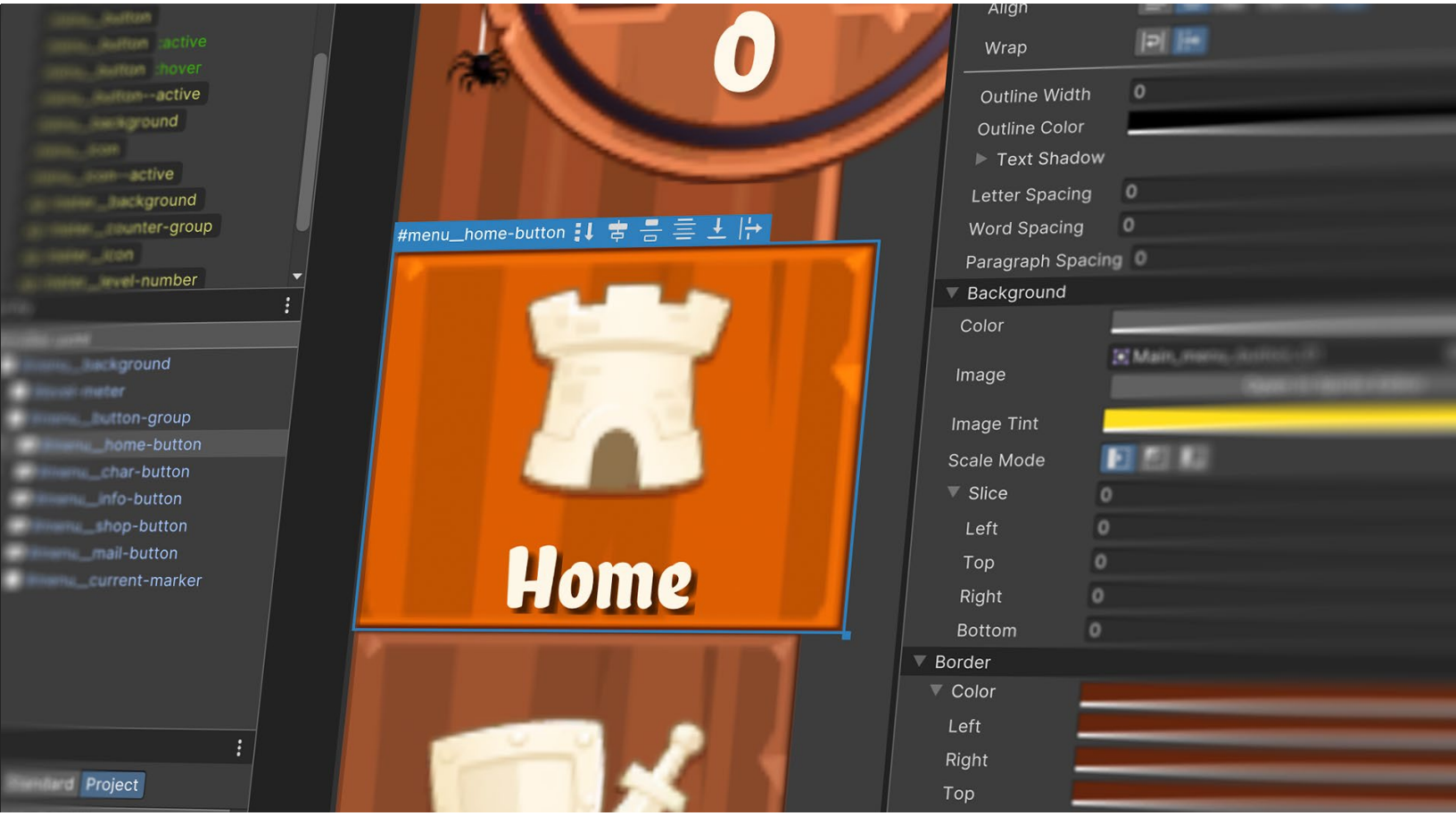
Шаблоны представляют повторно используемые UXML и доступны на вкладке Project области Library.

Дополнительные ресурсы

Подробнее о системе компоновки Flexbox можно узнать из следующих материалов. Flexbox и Yoga - признанные стандарты веб-разработки и разработки приложений, поэтому в интернете доступно множество дополнительных источников.

- UI Toolkit во время выполнения: подробный разбор
- Официальная документация Yoga
- Руководство CSS-Tricks по Flexbox

Стилизация



Изменение свойств стиля непосредственно в UI Builder

Создав каркасные макеты из визуальных элементов, можно перейти к оформлению или сохранить параметры форматирования в виде повторно используемых стилей. Именно в работе со стилями UI Toolkit раскрывает свои основные возможности.

Предпочтительно оформлять визуальные элементы через файлы Unity Style Sheet (USS), создаваемые командой Assets > Create > UI Toolkit > StyleSheet. Это аналог веб-файлов CSS с похожим форматом правил. Таблицы USS делают дизайн гибче, упрощают повторное использование и обеспечивают единообразие стилей во всём проекте.

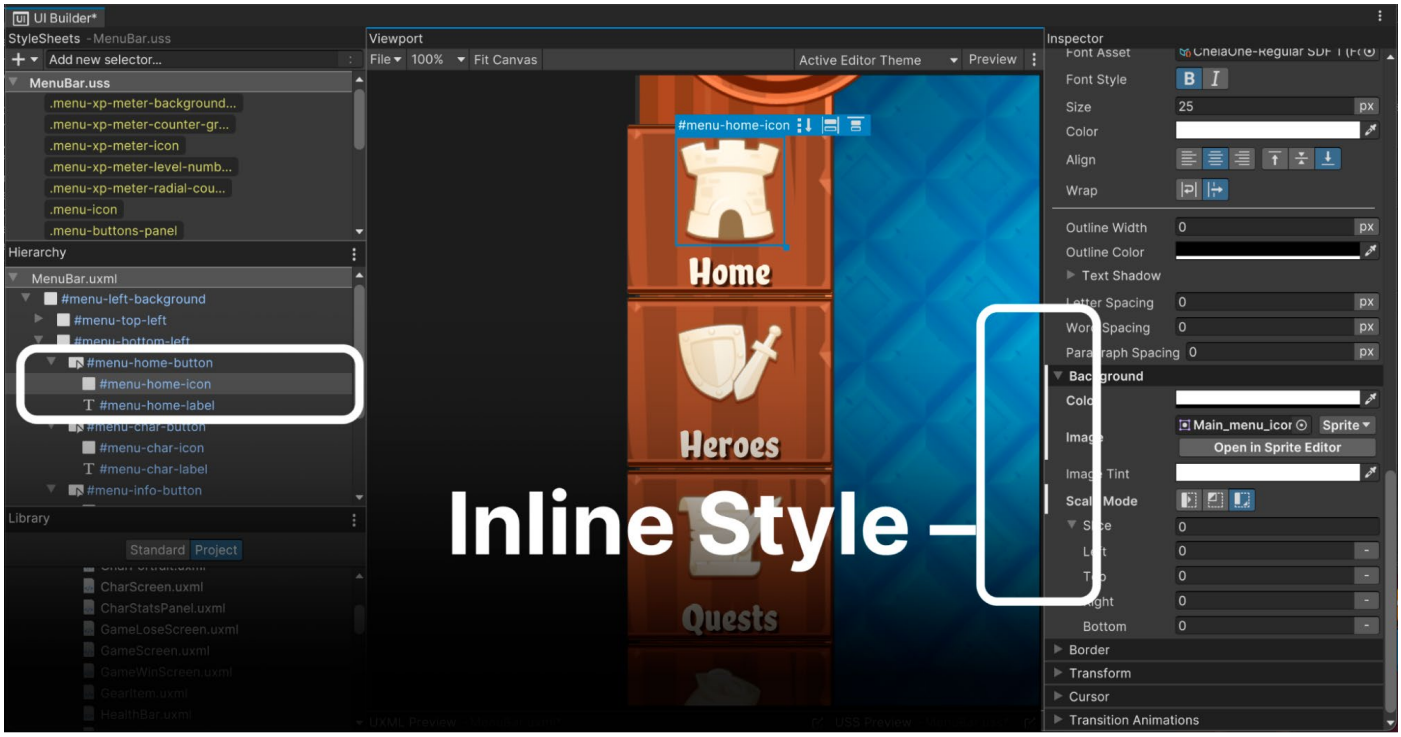
Файлы USS определяют размеры, цвета, шрифты, интервалы, границы и положение элементов.

Селекторы USS

Пока файл USS не создан, все изменения оформления сохраняются непосредственно в ресурсе UXML как встроенные стили. Они меняют вид только конкретного визуального элемента и не могут повторно использоваться в проекте. Если, например, в проекте сотни кнопок, обновлять стиль каждой по отдельности долго и неэффективно.

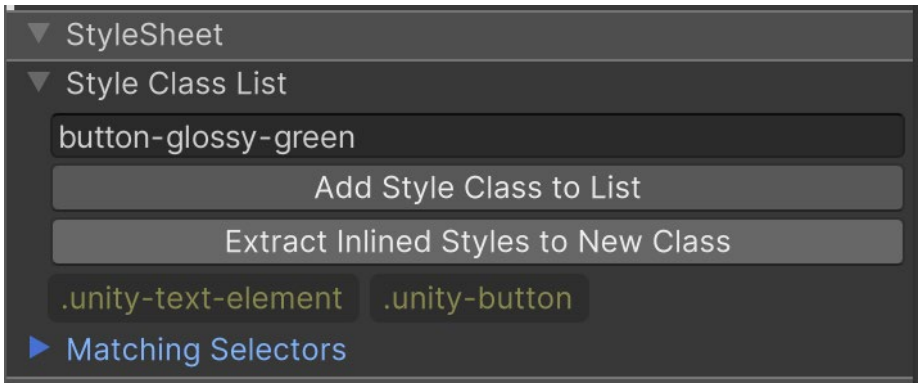
Вместо этого определите селектор USS. Селекторы позволяют таблицам стилей UI Builder совместно использовать и применять оформление ко множеству элементов в UXML-ресурсах.

Преобразование встроенных стилей в селекторы



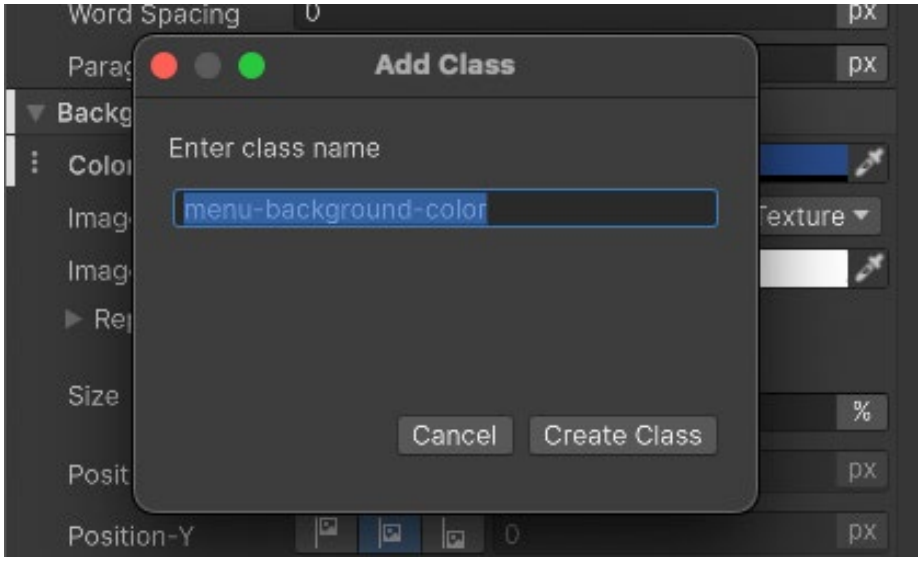
Встроенные стили переопределяют остальные стили.

Нажмите Add Style Class to List, введите имя и преобразуйте все встроенные стили элемента в селектор, начинающийся с жёлтой точки «.». Теперь свойства оформления сосредоточены в одном селекторе, который можно применять к другим кнопкам и элементам во всём проекте. Любое изменение селектора автоматически отразится на всех связанных элементах, поэтому такой подход легко масштабировать и сопровождать.



Извлечение всех свойств Inline Style в селектор

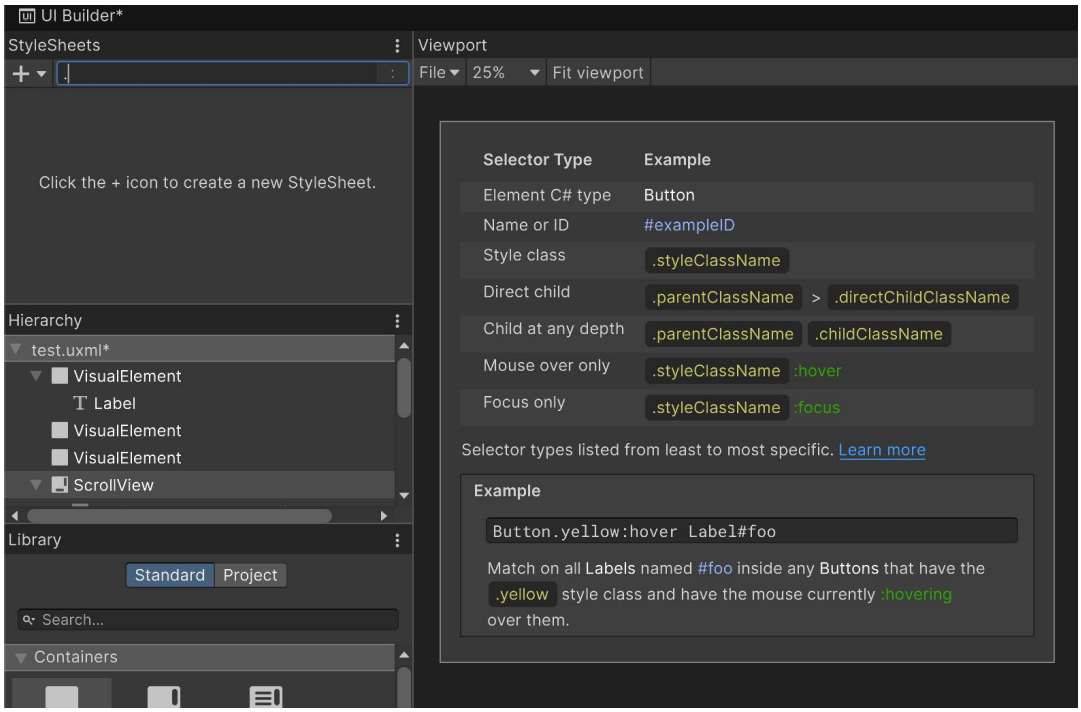
Чтобы вынести в новый селектор отдельное встроенное свойство, щёлкните вертикальное многоточие (⋮) рядом с ним и выберите Extract Inlined Style to Selector / Add Class. Свойство будет преобразовано в селектор.



Извлечение Inline Style отдельного свойства в селектор

Создание новых селекторов

Селекторы находят в визуальном дереве элементы, соответствующие заданным условиям, после чего UI Toolkit применяет к ним изменения стиля. Чтобы добавить селектор, щёлкните поле Add new selector... в левом верхнем углу UI Builder:



Справка по селекторам USS при создании нового селектора

Селекторы USS могут находить визуальные элементы по следующим признакам:

- Тип C# элемента. Такие селекторы работают по типу: Button, Label, ListView и т. д. Используются стандартные имена Type из области Library без специальных символов перед именем; селекторы типов отображаются белым. Например, Button применит стиль ко всем элементам типа Button.

- Имя или ID. Такой селектор применяет оформление ко всем элементам с одинаковым именем. Перед именем ставится знак решётки #, а сам селектор отображается синим. Например, #title применит стиль ко всем элементам Hierarchy с именем title.

> Примечание. В отличие от ID в HTML, атрибуты name в UXML не обязаны быть уникальными: UI Toolkit поддерживает UXML-шаблоны и повторно используемые компоненты, поэтому несколько элементов могут иметь одинаковое имя и стиль.

- Класс стиля. Селектор Style Class задаёт повторно используемый стиль, который можно применить к любому визуальному элементу, добавив имя класса в его свойство Class List. Перед именем ставится точка, а селектор отображается жёлтым. Например, чтобы применить класс .smallFont к элементу, добавьте smallFont в его Class List.

- Прямой дочерний элемент. Если после первого условия поставить >, второму условию будут сопоставляться только непосредственные дочерние элементы. Например, селектор #title > Label применит стиль к элементам Label, которые являются прямыми дочерними элементами #title. Элементы Label за пределами этого родителя или глубже в иерархии затронуты не будут.

- Потомок на любой глубине. Этот селектор похож на предыдущий, но второе условие применяется к любому потомку независимо от его глубины в иерархии родителя.

> Примечание. По возможности избегайте чрезмерно широких селекторов, особенно заканчивающихся на * или нацеленных на общие классы Unity вроде .unity-button. Глубокие селекторы потомков могут снижать производительность, если проверяют значительную часть визуального дерева.

- Псевдокласс. Псевдоклассы задают отдельные стили для разных состояний визуального элемента, например при наведении указателя или получении фокуса. Они обозначаются двоеточием : и дополняют существующие селекторы. Например, Button:focus применит заданный стиль ко всем элементам Button в фокусе.

> Псевдоклассы удобны для визуальной обратной связи: эффектов наведения, индикаторов фокуса и других состояний. В сочетании с анимацией USS они позволяют создавать плавное движение и динамические переходы, улучшая взаимодействие с интерфейсом.

> Список доступных псевдоклассов приведён по ссылке.

Если визуальный элемент соответствует нескольким селекторам, действует селектор с наибольшей специфичностью.

В USS приоритет определяется в следующем порядке:

1. Inline Styles. Стили, назначенные непосредственно элементу, например в UXML или коде, имеют наивысший приоритет и переопределяют все селекторы USS.

2. Селекторы ID (#id). Это наиболее специфичные селекторы USS; они применяются к элементам с соответствующим свойством name.

3. Селекторы классов (.className). Применяются к элементам, у которых соответствующий класс добавлен в Class List.

4. Селекторы типов C#, например Button или Label. Они применяются ко всем элементам указанного типа.

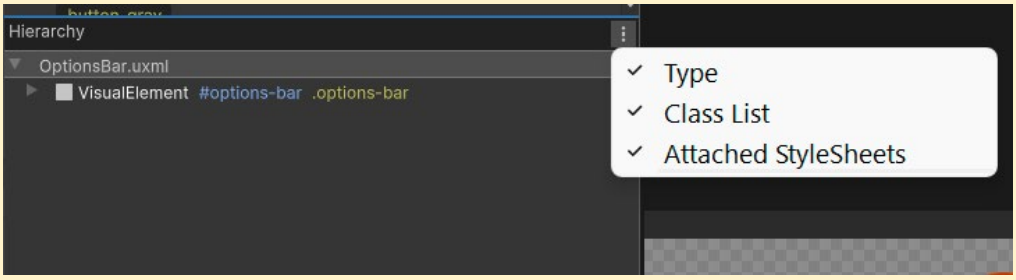
Если элемент имеет встроенный стиль и одновременно соответствует ID-селектору #title, встроенный стиль переопределяет ID-селектор. Если элемент соответствует селектору класса и селектору типа, приоритет имеет селектор класса.

Если несколько селекторов одинаковой специфичности переопределяют одно свойство, решающим становится их порядок в таблице USS: преимущество имеет селектор, расположенный ниже по списку.

Подробнее о приоритетах селекторов см. в документации. .

Совет: дополнительные сведения в Hierarchy

Щёлкните вертикальное многоточие (⋮) в заголовке Hierarchy, чтобы вывести дополнительные сведения об элементах UI.



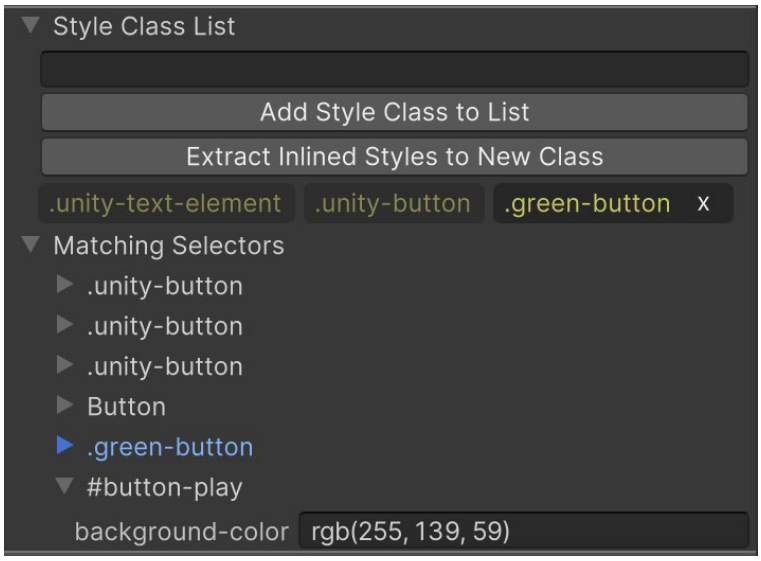
Фильтрация разных селекторов в Hierarchy

В области Hierarchy рядом с Type элемента можно отобразить дополнительные сведения. При включении соответствующих параметров появляются селектор имени #options-bar и селектор класса стиля .options-bar.

Некоторые селекторы начинаются с префикса .unity-. Это стандартные стили, применяемые ко всем элементам. Определённые пользователем селекторы переопределяют их значения.

Селекторы, назначенные элементам

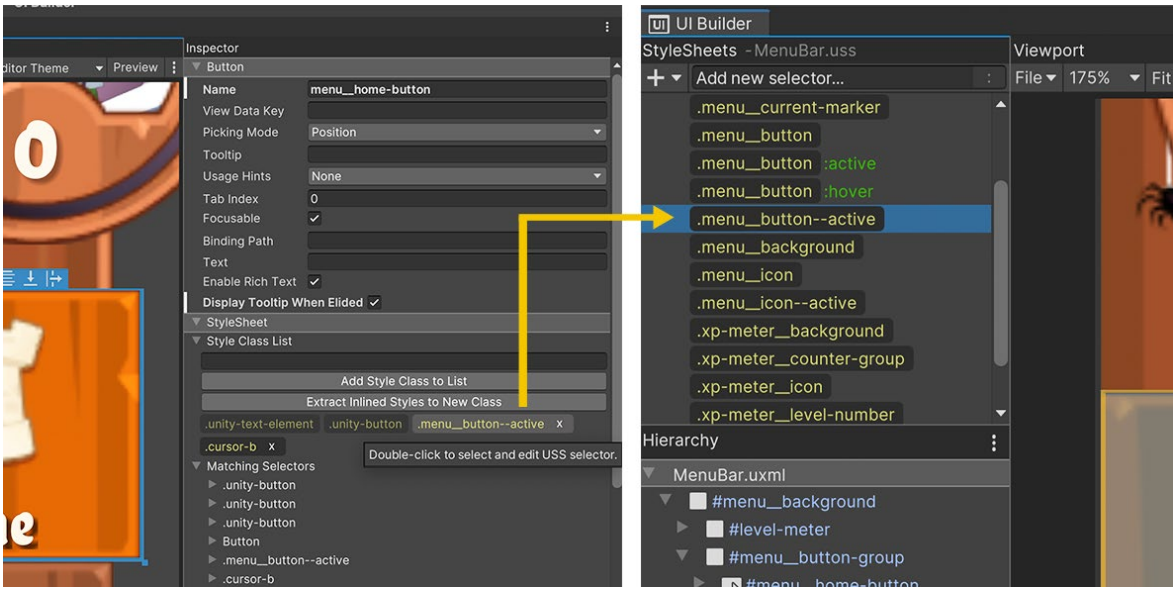
В инспекторе можно увидеть селекторы, которым соответствует выбранный в Hierarchy элемент. Приоритет имеет селектор в нижней части списка. Разверните подробности, чтобы узнать, какие параметры стиля он изменяет.



Для выбранного визуального элемента в инспекторе показаны соответствующие селекторы

Редактирование селекторов

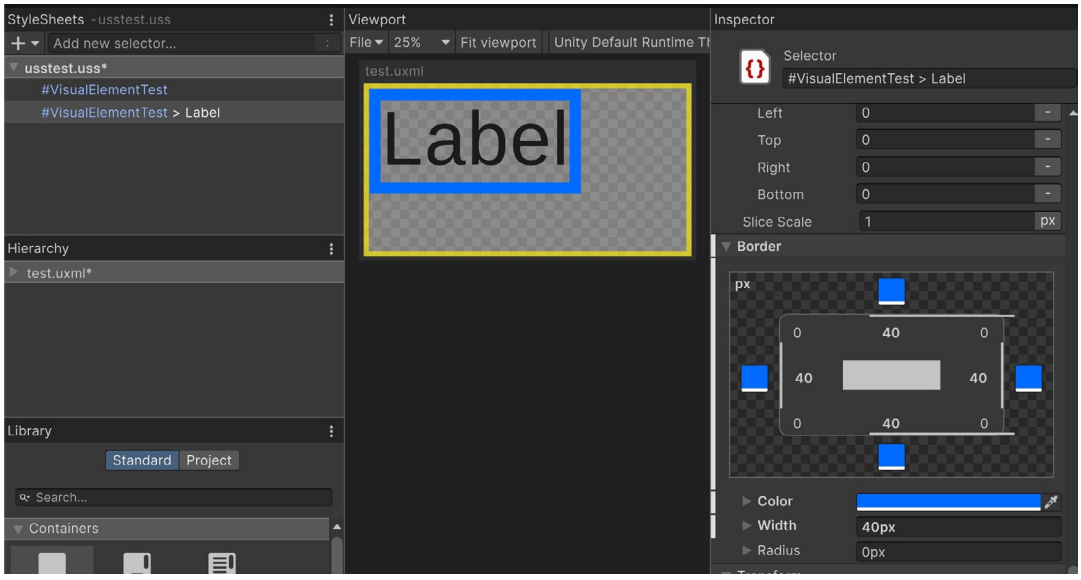
При изменении селектора Style выберите Style Class в области Style Sheet, а не визуальный элемент в Hierarchy. Иначе вы измените встроенный стиль конкретного элемента, но не сам Style Class.



Дважды щёлкните Style Class в инспекторе, чтобы убедиться, что выбран именно он.

Двойной щелчок по Style Class в инспекторе снимает выделение с элемента и выбирает селектор Style. Параметры селектора редактируются так же, как встроенные стили в UXML: выберите селектор в области StyleSheets и задайте переопределения. Изменённые значения выделяются жирным шрифтом и белой линией.

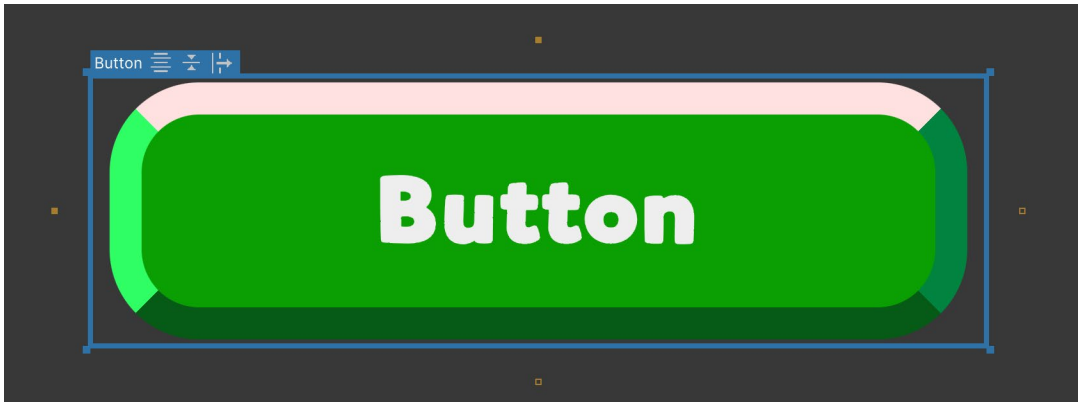
Чтобы сбросить значение, воспользуйтесь меню с вертикальным многоточием (⋮) рядом со свойством.



Редактирование селектора USS

Многочисленные параметры форматирования позволяют менять основной вид элементов и шрифтов. Расширенные возможности UI Toolkit сокращают потребность в специально подготовленных спрайтах.

В UI Builder к элементам можно добавлять контуры, скруглённые углы, настройки изображения и цвета границы. Среди прочих возможностей - эффекты фаски и замена изображения курсора.



UI Toolkit предлагает несколько эффектов оформления, не требующих дополнительных текстур.

Переопределение стилей

Правила существуют, чтобы их нарушать. Для любого класса стиля UI найдутся исключения. Например, если сотни элементов Button отличаются только значками, создавать отдельный селектор для каждой кнопки не требуется: это лишило бы повторно используемые стили их главного преимущества.

Примените ко всем кнопкам один стиль, а затем переопределите только уникальные части. Например, каждый Button может задать собственный значок через Background > Image.

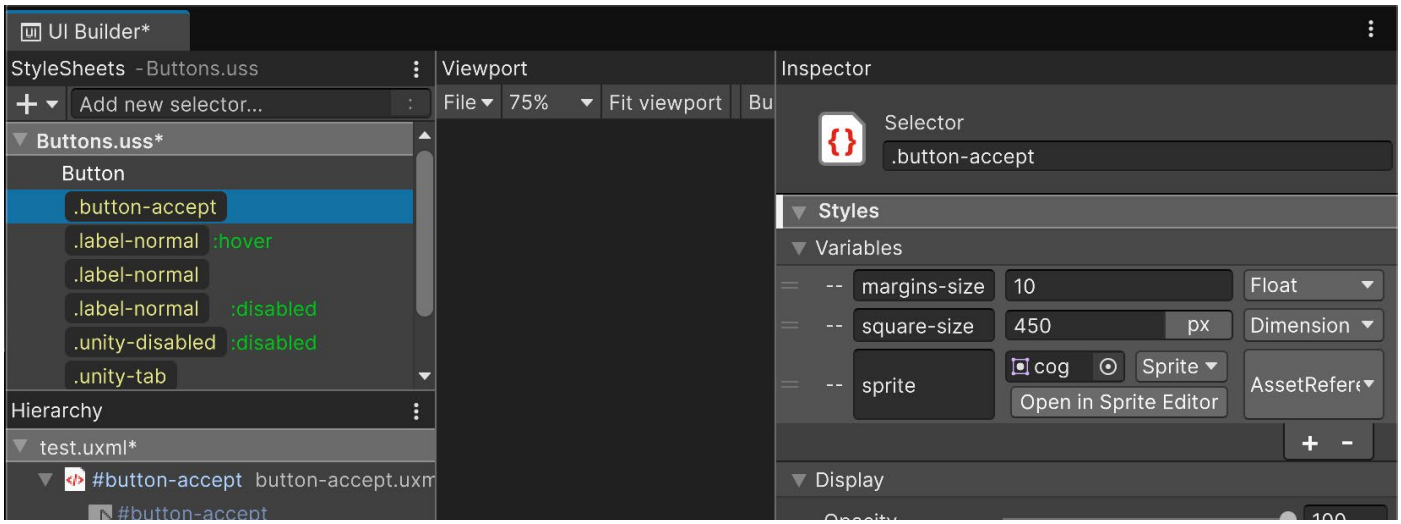
Такие переопределения представляют собой свойства встроенного стиля (Inline Style).

Совет: встроенные стили имеют приоритет над селекторами

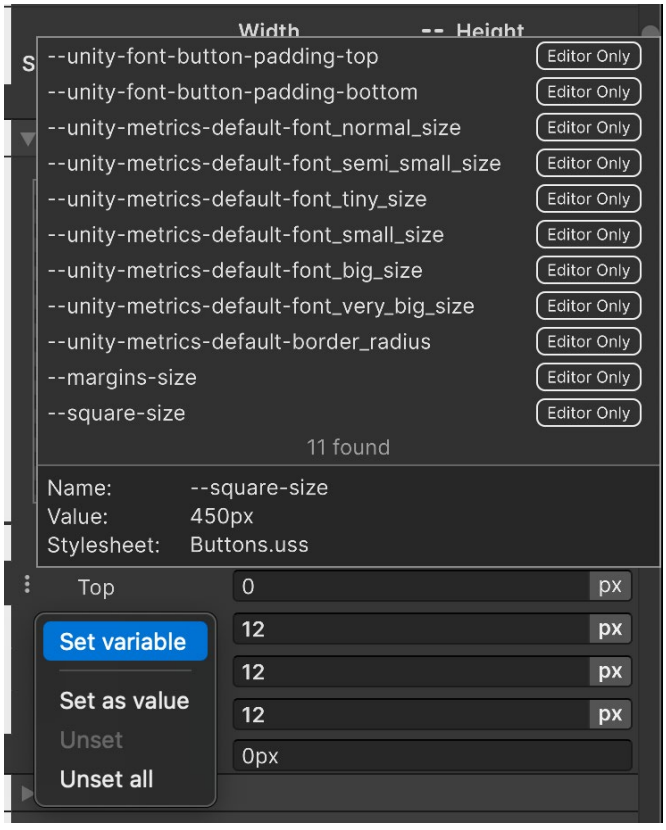
Встроенный стиль всегда имеет приоритет над селекторами. Если после применения селектора оформление не меняется, проверьте, нет ли у элемента переопределений.

Переменные USS

Переменные USS экономят время, когда одно значение требуется вручную задавать в разных свойствах. При изменении переменной обновляются все использующие её свойства USS. Начиная с Unity 6.1 такие переменные можно настраивать и в редакторе UI Builder.



В Unity 6.1 переменные селекторов USS можно создавать и редактировать в UI Builder



Назначение свойству переменной вместо непосредственного значения

Можно создавать переменные следующих типов: float, color, string, ссылка на ресурс для фоновых изображений, размерность в пикселях, градусах, процентах и других единицах, а также enum. Область видимости переменной ограничена её селектором: использовать переменные из других селекторов нельзя, однако сам селектор можно применить к любому числу элементов.

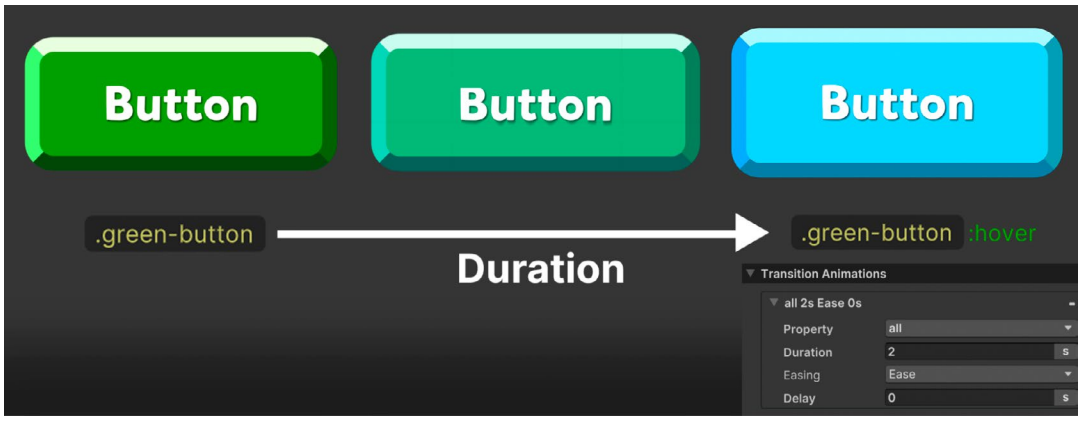
Анимация переходов USS

Переходы между экранами меню заметно улучшают визуальное восприятие интерфейса и удобство работы с ним. В UI Toolkit их довольно легко настроить с помощью свойства Transition Animations в Inspector.

Для анимации можно задать Property, Duration, Easing и Delay. После настройки переход будет автоматически воспроизводиться при изменении соответствующих стилей во время выполнения.

Например, рассмотрим переход между состояниями кнопки: псевдокласс :hover для селектора класса .green-button. У каждого состояния заданы свои размер и цвет.

Чтобы определить переход при наведении указателя, настройте Transition Animations для селектора .green-button:hover в нижней части Inspector. В результате кнопка будет плавно реагировать на перемещения указателя.



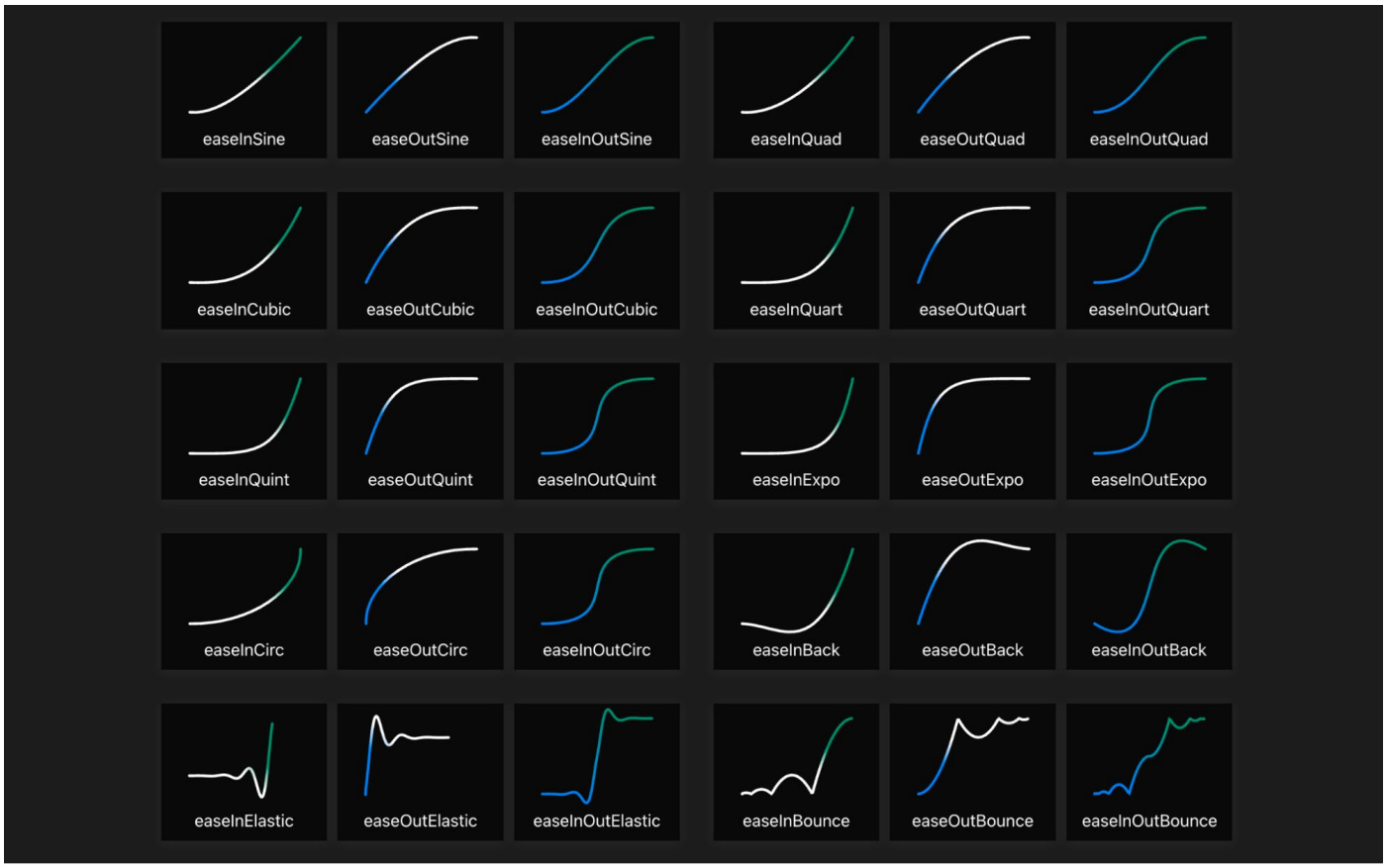
С помощью Transition Animations можно плавно переходить от одного стиля к другому.

Transition Animations выполняет интерполяцию между стилями со следующими параметрами:

- Property: определяет, какое свойство интерполировать. По умолчанию выбрано значение all, но в раскрывающемся списке можно указать отдельное свойство. В приведённом выше примере состояние :hover изменяет только свойства Color и Transform. Полный перечень доступен в списке анимируемых свойств.

- Duration: длительность перехода в секундах или миллисекундах. Чтобы переход был виден, значение Duration должно быть больше 0.

- Easing Function: функция сглаживания определяет, как анимация развивается во времени, и позволяет имитировать естественное движение - например, ускорение, замедление или упругость. Благодаря сглаживанию переход выглядит плавнее и естественнее, чем при простой линейной интерполяции с постоянной скоростью.



Эта памятка поможет наглядно представить доступные функции (визуализация предоставлена сайтом https://easings.net/).

- Delay: задержка в секундах или миллисекундах перед началом перехода.

- Add Transition: каждое свойство нового состояния можно анимировать отдельно, задав для него собственные длительность, задержку и функцию сглаживания.

> Нажмите Add Transition, чтобы добавить в цепочку ещё один анимированный переход. Так можно одновременно запускать несколько перекрывающихся переходов, делая движение более естественным и менее механическим.

Совет: события переходов

Для анимируемых визуальных элементов можно регистрировать обратные вызовы событий перехода. Они необходимы для более сложных сценариев, например последовательного или циклического воспроизведения.

Ниже перечислены распространённые события и моменты их отправки:

- TransitionRunEvent: отправляется при создании перехода.

- TransitionStartEvent: отправляется после завершения фазы задержки, когда начинается переход.

- TransitionEndEvent: отправляется после завершения перехода.

- TransitionCancelEvent: отправляется при отмене перехода.

Подробнее о переходах USS см. в документации .

Для анимации визуальных элементов дополнительный код не требуется: псевдоклассы (:active, :inactive, :hover и другие) могут иметь собственные селекторы. Когда псевдокласс изменяет стиль, все заданные переходы автоматически анимируют это изменение. Например, кнопка может увеличиваться или уменьшаться при наведении (:hover) либо нажатии (:active), а элементы - постепенно исчезать или становиться невидимыми в зависимости от действий пользователя и других событий.

Набор псевдоклассов предопределён; создавать собственные псевдоклассы нельзя.

Смена стилей по требованию

При любых других событиях игры оформление также можно изменять из кода с помощью API элементов UI. Например, чтобы выбирать стиль в зависимости от редкости персонажа, используйте методы RemoveFromClassList и AddToClassList.

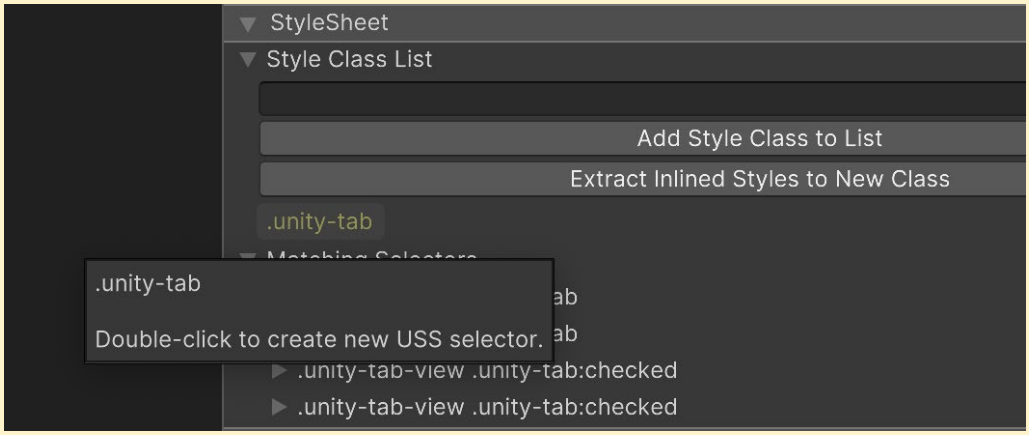
```
if (character.rarity == RarityType.Legendary)
{
   visualElement.RemoveFromClassList("common");
   visualElement.AddToClassList("legendary");
}
```

Кроме того, можно задействовать псевдоклассы :active и :inactive, зависящие от состояния активности визуального элемента, и применять переходы USS при смене этого состояния. Так они могут представлять состояния «до» и «после».

Кнопки панели меню в примере UI Toolkit Sample - Dragon Crashers используют PointerEventClick для запуска некоторых переходов вручную.

Совет: переопределение стандартных селекторов Unity

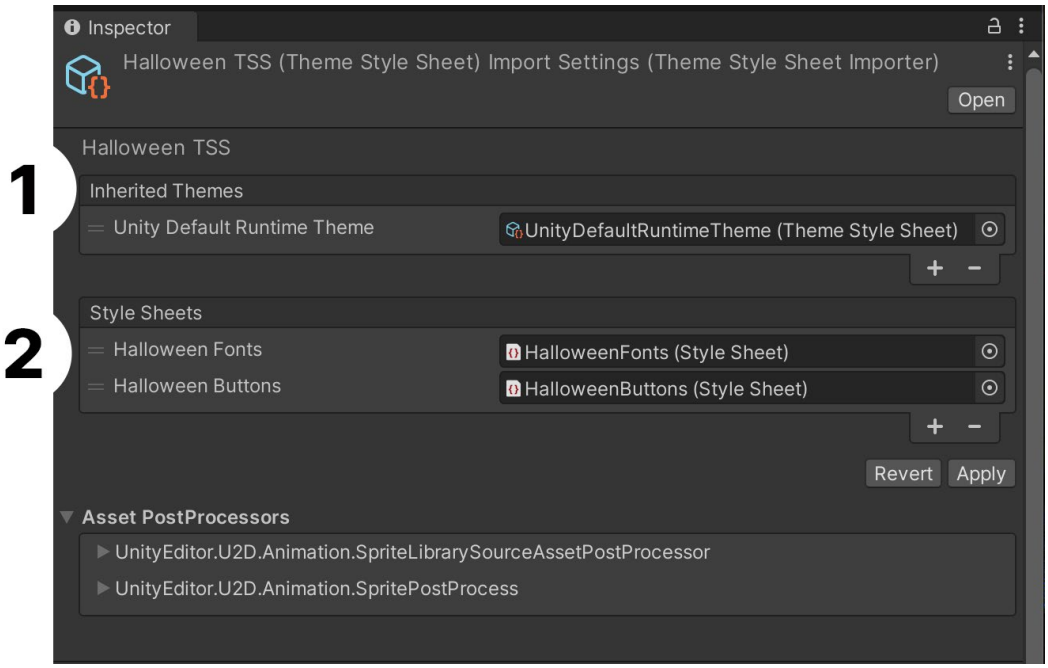
Более сложные визуальные элементы, например представление Tab, состоят из родительского элемента и заранее определённых системой дочерних элементов. При добавлении содержимого такие элементы ведут себя особым образом, а применённые стили могут выглядеть недоступными для редактирования, поскольку созданы Unity. Любой стандартный селектор можно переопределить: дважды щёлкните используемый селектор и создайте его копию для редактирования в своей таблице стилей USS.



Темы

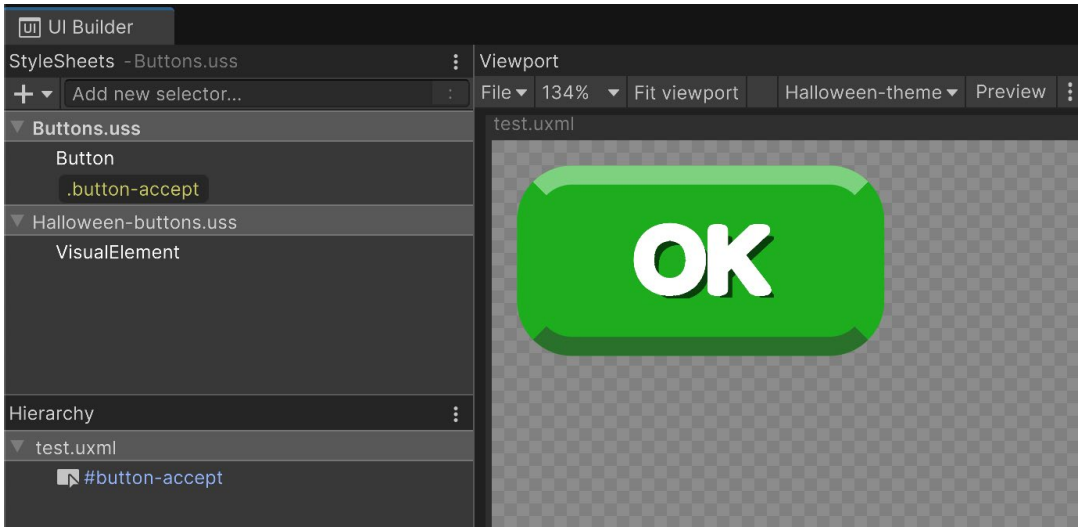
Если вы хотите создать сезонную версию интерфейса или предложить несколько цветовых схем, упростить работу помогут таблицы стилей темы Theme Style Sheets (TSS). Чтобы создать файл TSS, выберите Create > UI Toolkit > TSS theme file.

Файлы TSS - это ресурсы, работающие аналогично обычным файлам USS. Они служат основой для пользовательской темы, состоящей из селекторов USS, а также настроек свойств и переменных.



В этом примере элементов интерфейса в стиле Хэллоуина тема Halloween TSS сначала наследуется от Unity Default Runtime TSS, а затем добавляет тематические таблицы стилей для шрифтов и кнопок.

При наследовании тем, если в новой теме отсутствуют таблицы стилей с некоторыми селекторами исходной темы, для них применяется оформление исходной темы. Это упрощает настройку. Например, можно создать тему, которая изменяет только шрифты, а остальные элементы интерфейса - цвета, отступы и границы - продолжат использовать стили исходной темы. Такой подход удобен для светлой и тёмной тем, индивидуального оформления интерфейса для разных персонажей и тематических внутриигровых событий.



Показанная на снимке экрана тема Halloween TSS использует файл Halloween-buttons.uss. Однако в нём нет селектора .button-accept, применённого к кнопке, поэтому для неё используется соответствующий селектор исходной темы.

Создать новую тему на основе существующей можно следующим образом:

1. Создайте файл TSS и добавьте в него тему, от которой нужно наследоваться, а также новый файл USS для этой темы.

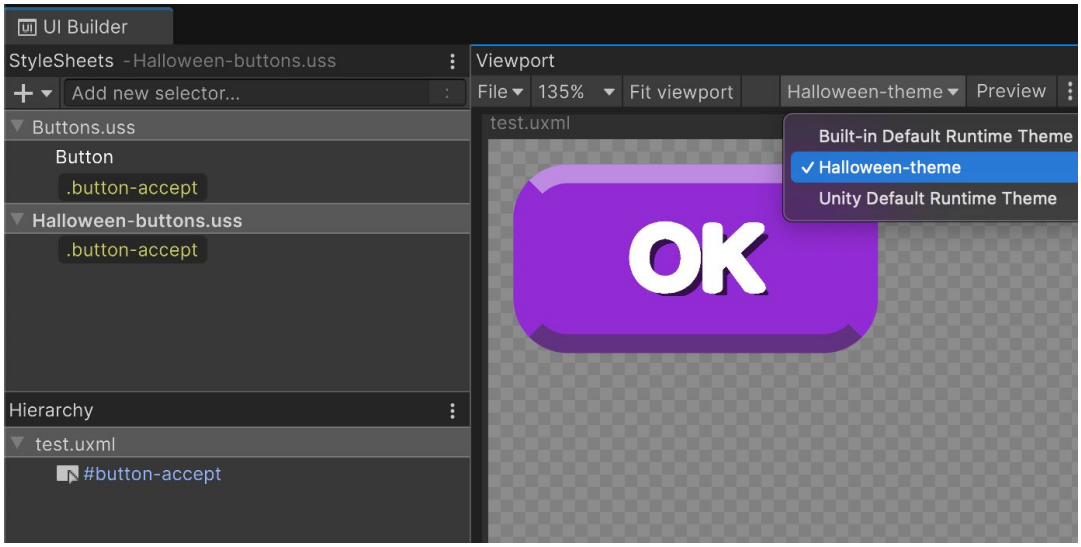
2. В UI Builder в разделе StyleSheets нажмите Add Existing USS и выберите файл, который будет использовать новая тема.

3. Скопируйте селектор, который должна переопределить новая тема.

4. Вставьте его в файл USS новой темы, затем щёлкните правой кнопкой мыши и выберите Set as Active USS.

5. Отредактируйте селектор в новом файле USS.

6. В раскрывающемся списке UI Builder можно посмотреть стиль, применяемый той или иной темой.



Выберите тему, которую нужно применить в области просмотра UI Builder.

Для использования во время выполнения укажите новую тему в поле Theme Style Sheet инспектора Panel Settings.

Соглашения об именовании

В UI Toolkit визуальные элементы и стили USS приходится находить по строковым идентификаторам, поэтому единые правила именования сокращают число ошибок и делают код понятнее.

Поскольку все участники команды обращаются к одним и тем же ресурсам UXML и USS, из которых состоит интерфейс, важно стандартизировать имена как визуальных элементов, так и таблиц стилей. Соглашения об именовании помогают поддерживать порядок в иерархии UI Builder, устраняют неоднозначность в оформлении кода и обеспечивают единообразие кодовой базы.

```
root.Query<Button>("foo").First();
```

Имена визуальных элементов используются для хранения ссылок на них в коде.

Универсального руководства по стилю не существует: выбирайте правила, которые лучше всего подходят вашей команде и проекту. Тем не менее обычно стоит придерживаться общепринятых отраслевых стандартов. Поэтому для визуальных элементов и таблиц стилей мы рекомендуем соглашение Block Element Modifier (BEM - «блок, элемент, модификатор»). BEM широко применяется в CSS и современной веб-разработке, которые послужили источником идей для UI Toolkit.

По имени элемента, составленному в стиле BEM, можно сразу понять его назначение, расположение и связь с соседними элементами. В соглашении BEM используются три основных компонента:

```
block-name__element-name--modifier-name
```

Пример:

```
navbar-menu__shop-button--small
```

Каждая часть имени может содержать латинские буквы, цифры и дефисы. Части имени соединяются двойным подчёркиванием __ или двойным дефисом --. Рассмотрим все три компонента подробнее:

- Имя блока (block-name) обозначает высокоуровневый компонент: например, navbar-menu, панель характеристик персонажа или любой другой самостоятельный и значимый компонент интерфейса в макете. Для универсальной кнопки, не относящейся к конкретному блоку, имя блока можно опустить, например: button--small.

- Элемент (element-name) является дочерней частью блока и поэтому семантически с ним связан. Иными словами, контекст элемента задаётся блоком, и без него элемент не существует. Например, shop-button оформлен иначе, чем другие кнопки блока navbar-menu; полное имя может выглядеть как navbar-menu__shop-button.

> Если новый элемент создаёт дочерние элементы в конструкторе, назначьте им соответствующие классы. Например: my-block__first-child и my-block__other-child.

- Модификатор обозначает вариант или состояние блока либо элемента. Это может быть нажатая кнопка, выбранный и подсвеченный элемент текстового поля или, как в нашем примере, уменьшенный вариант кнопки магазина. Модификаторы позволяют адаптировать компонент к разным сценариям без дублирования кода.

Дополнительные примеры имён в стиле BEM:

— `menu__button-home`

— `menu__button-shop`

— `navbar-menu__shop-button--small`

— `navbar-menu__shop-button--large`

Имена классов BEM описывают сами себя: разработчикам проще понять структуру и назначение компонентов и поддерживать ясную иерархию стилей по мере роста проекта. Общее правило - отдавайте предпочтение читаемости, а не краткости. Ясность важнее нескольких секунд, сэкономленных удалением пары гласных.

В этих примерах части имени разделяются дефисами (так называемый kebab-case) - это распространённый стиль именования в CSS. Команде следует в начале проекта выбрать подходящую схему именования и придерживаться её на всём протяжении разработки.

Подробнее о правилах именования CSS можно прочитать в этой статье и в документации UI Toolkit.

Советы: соглашения об именовании в UI Toolkit

Ниже приведены рекомендации по эффективному именованию:

- Выбирайте короткие, понятные и однозначные имена. Они должны быть лаконичными, но достаточно содержательными, чтобы передавать назначение и роль элемента в интерфейсе.

- Подчёркивайте в именах роли и связи: например, используйте inventory__slot--equipped вместо inventory__button--equipped. Опускайте названия типов вроде Button и Label, если они не делают имя понятнее.

- Избегайте имён и модификаторов, смысл которых может измениться. Например, пока цветовая схема не утверждена, используйте button--quit вместо button--red. Семантические имена предпочтительнее имён, описывающих внешний вид: они остаются актуальными после изменения оформления.

- Распространите эти правила на графические ресурсы интерфейса UI Toolkit, например спрайты и текстуры. Единообразное именование в коде и ресурсах помогает сохранить понятные связи и порядок во всём проекте.

- Если элемент планируется использовать в других проектах, добавьте к классам префикс, чтобы избежать конфликтов с существующими пользовательскими именами. Пространства имён и префиксы предотвращают коллизии при интеграции с другими проектами и библиотеками.

- Вызывайте AddToClassList() в конструкторе, чтобы назначать экземплярам элемента соответствующие классы USS. Метод добавляет необходимые классы при создании экземпляра, обеспечивая правильное применение стилей, единообразие и понятность кода интерфейса.

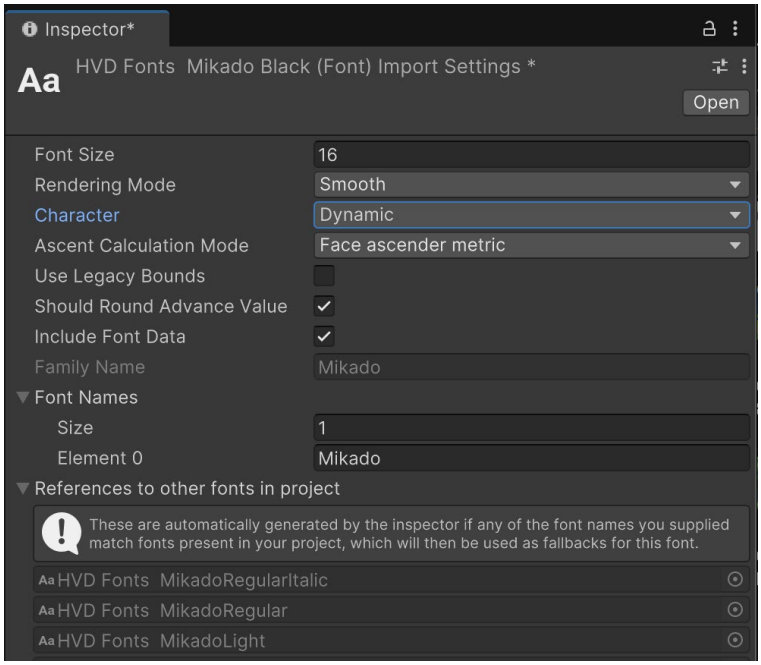
Текст

UI Toolkit использует TextCore - технологию рендеринга шрифтов, первоначально созданную на основе TextMesh Pro, который применяется в прежней системе интерфейса Unity UI. TextCore предоставляет расширенные возможности оформления и чётко отображает текст при разных кеглях и разрешениях. Технология использует шрифты на основе знакового поля расстояний Signed Distance Field (SDF), поэтому ресурсы шрифтов сохраняют резкость при преобразовании и увеличении. Подробнее о режимах рендеринга TextCore см. в документации.

Теперь рассмотрим типы ресурсов шрифтов и их назначение.

Исходный файл шрифта

Наиболее распространённые форматы шрифтов, TTF и OTF, необходимо преобразовать в ресурсы шрифтов, прежде чем использовать их в проекте Unity. Ресурс шрифта - это специальный ресурс Unity, содержащий данные для отображения шрифта: глифы символов, метрики и параметры рендеринга, включая размер, насыщенность и начертание. Для импортированного исходного файла отображаются сведения о каждом семействе шрифтов и доступных вариантах рендеринга.



Многие из этих параметров импорта остались от прежней текстовой системы Unity UI и будут удалены в будущих выпусках. Параметры Rendering Mode, Character и Include Font Data используются при создании ресурса шрифта.

Чтобы создать соответствующий ресурс шрифта, выберите исходный файл шрифта, щёлкните правой кнопкой мыши в окне Assets и выберите Create > Text Core > Font Asset > SDF, если SDF - предпочтительный режим рендеринга.

В разных системах интерфейса используются разные ресурсы шрифтов.

Параметры ресурса шрифта

После преобразования исходного файла выберите ресурс шрифта. В Inspector отобразятся все параметры, позволяющие полностью управлять созданием шрифта. Рассмотрим основные из них; дополнительные сведения о ресурсах шрифтов приведены в документации:

- Face Info: параметры интервалов и масштаба шрифта. С их помощью можно скорректировать исходный шрифт, если стандартные значения не подходят приложению.

- Generation Settings: основные параметры, включая исходный шрифт, начертание для ресурса шрифта, если исходный файл содержит несколько начертаний, режим заполнения атласа и режимы рендеринга.

- Atlas and Material: созданные материал и текстура; тип атласа - статический или динамический; режим рендеринга - растровый или SDF. Здесь также можно управлять размером создаваемого атласа, что особенно важно для языков с большими наборами символов.

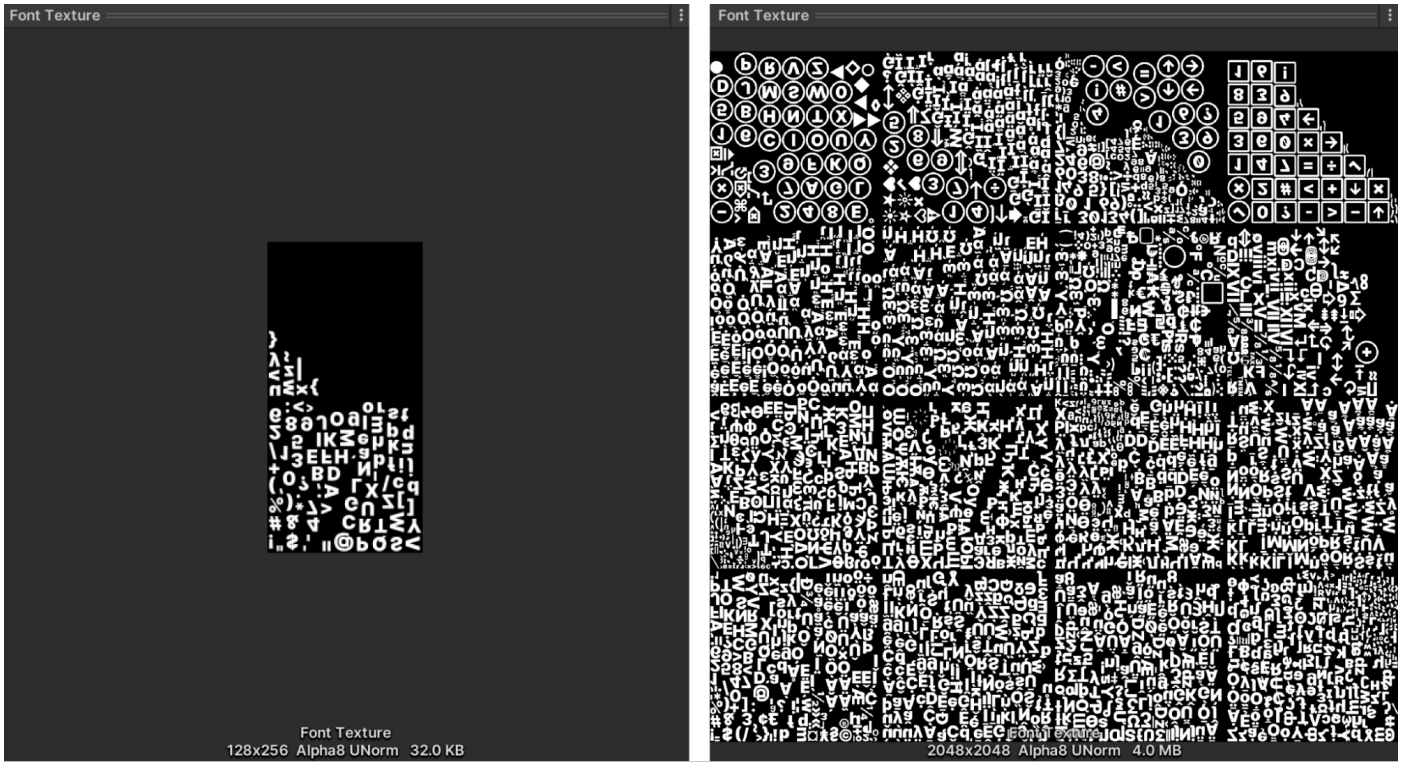
- Font Weights: имитация разных вариантов насыщенности, если их нет в исходном ресурсе шрифта.

- Fallback Font Asset: резервный шрифт для символов или глифов, отсутствующих в текущем ресурсе шрифта.

- Character and Glyph Tables: подробный список всех символов и глифов, включённых в ресурс шрифта.

- Ligature table: добавление глифа, который используется для сочетания двух символов и улучшает читаемость и зрительную плавность текста.

- Glyph Adjustments: переопределения для отдельных символов или глифов.



Исходные шрифты и атласы могут увеличивать размер сборки: слева показан атлас с символами ASCII, справа - атлас с полным набором символов Unicode.

Если выбрать ресурс шрифта, в верхней части Inspector под кнопкой Update Atlas Texture будет доступен Font Asset Creator. Он позволяет полностью управлять заполнением атласа и его свойствами.

Совет: поля и разрешение атласа

Между символами в текстуре шрифта необходимо оставить поля заданного размера в пикселях, чтобы каждый символ можно было отрисовывать отдельно. Поля также создают пространство для градиента SDF. Чем они шире, тем плавнее переход, что позволяет получать качественное изображение и применять эффекты вроде толстой обводки.

Если используются только символы ASCII, для большинства шрифтов достаточно атласа 512 x 512 с полем 5 пикселей. Для шрифтов с большим числом символов может потребоваться более высокое разрешение или несколько атласов. Как правило, размер поля должен составлять примерно одну десятую размера выборки.

Вариант ресурса шрифта

Чтобы изменить метрики шрифта без создания нового атласа, выберите Create > Text Core > Font Asset Variant. Такой вариант может содержать альтернативные метрики строк.

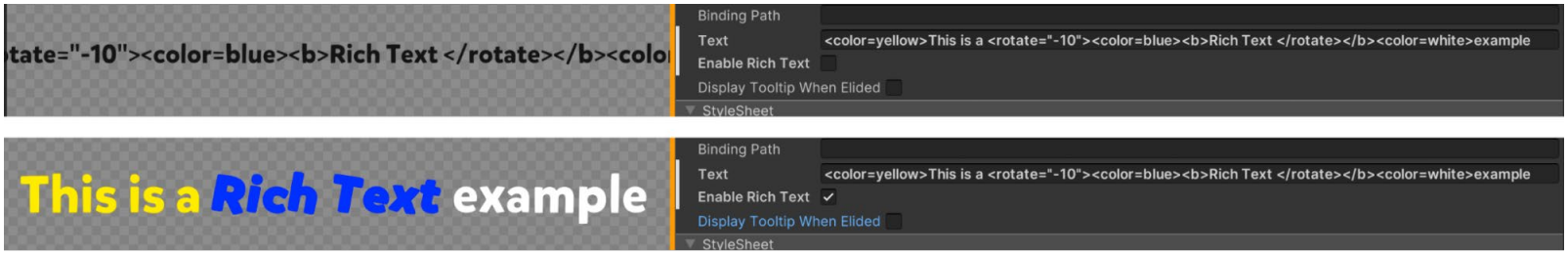
Он хранит собственные параметры Face Info - например, высоту строки и положение нижнего индекса, - но по-прежнему ссылается на исходный атлас. Благодаря этому вариант может иметь оформление, отличное от исходного ресурса шрифта, не занимая дополнительное место под текстуры.

Форматированный текст

Теги форматированного текста изменяют вид и компоновку текста с помощью дополнительных тегов в текстовом поле. Их можно использовать для визуальных элементов как в коде, так и в UI Builder. Теги позволяют форматировать текст во время выполнения - например, менять оформление имени пользователя.

С их помощью можно изменять цвет или выравнивание текста, не меняя свойства и стили элемента. Применяйте их для оформления диалогов или визуального акцента на важных сообщениях.

Чтобы включить форматированный текст в UI Builder, откройте Extra Settings. После этого текст вместе с тегами будет отображаться правильно. Например, текст между тегами и станет полужирным.

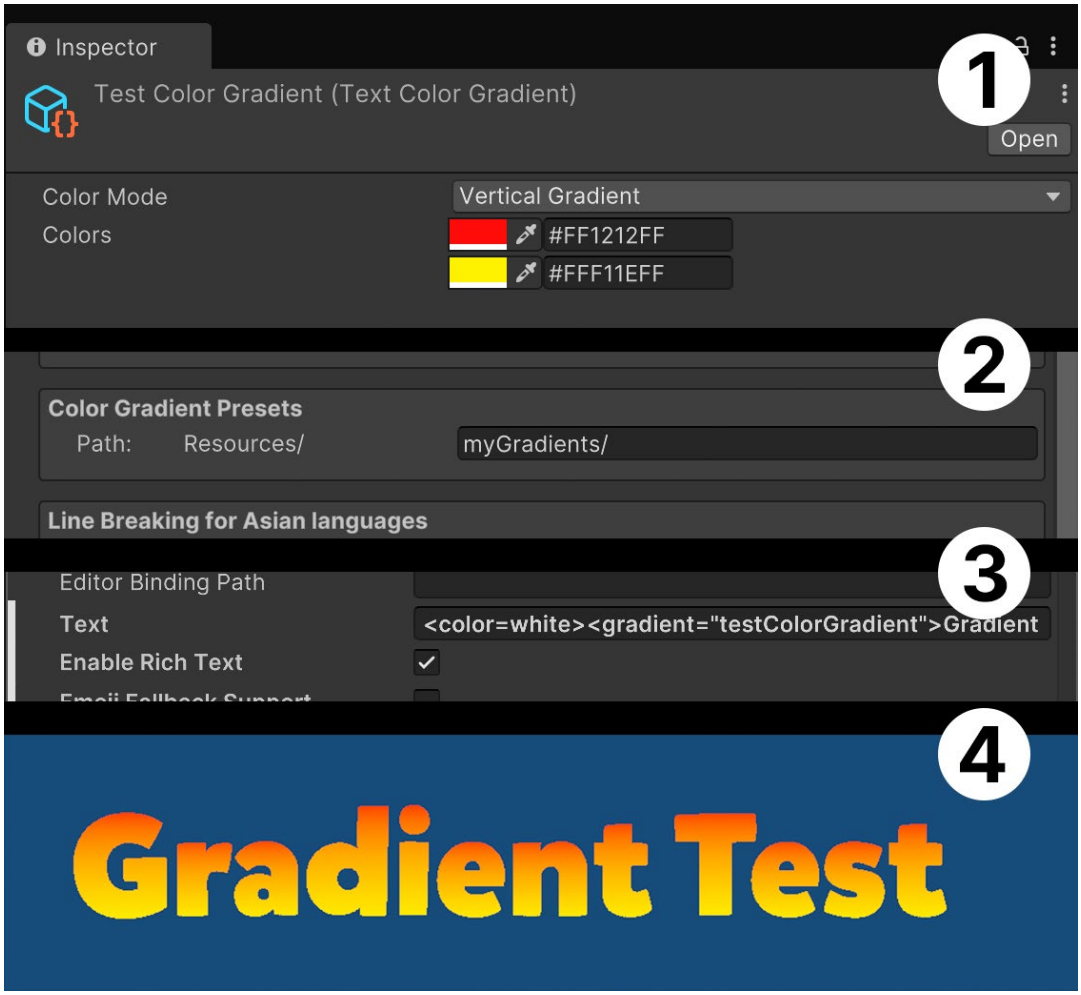


Включите поддержку тегов форматированного текста в UI Builder, чтобы они изменяли визуальное оформление текста.

Полный список доступных тегов форматированного текста и их параметров приведён по ссылке.

Градиенты

Градиенты помогают оформить интерфейс. В UI Toolkit их можно применять с помощью тега <gradient>. Чтобы создать простой градиент, выполните следующие действия:



1. Создайте ресурс цветового градиента командой Create > Text Core > Gradient Color. Поместите файл в папку Assets/Resources или в любую её вложенную папку.

2. Создайте ресурс Text Settings, на который будет ссылаться Panel Settings. В ресурсе найдите параметр Color Gradient Presets и укажите папку или вложенную папку внутри Resources, где находится градиент.

3. Добавьте в UI Builder следующие теги форматированного текста: `<color=white><gradient="testColorGradient">Gradient Test</gradient></color>`.

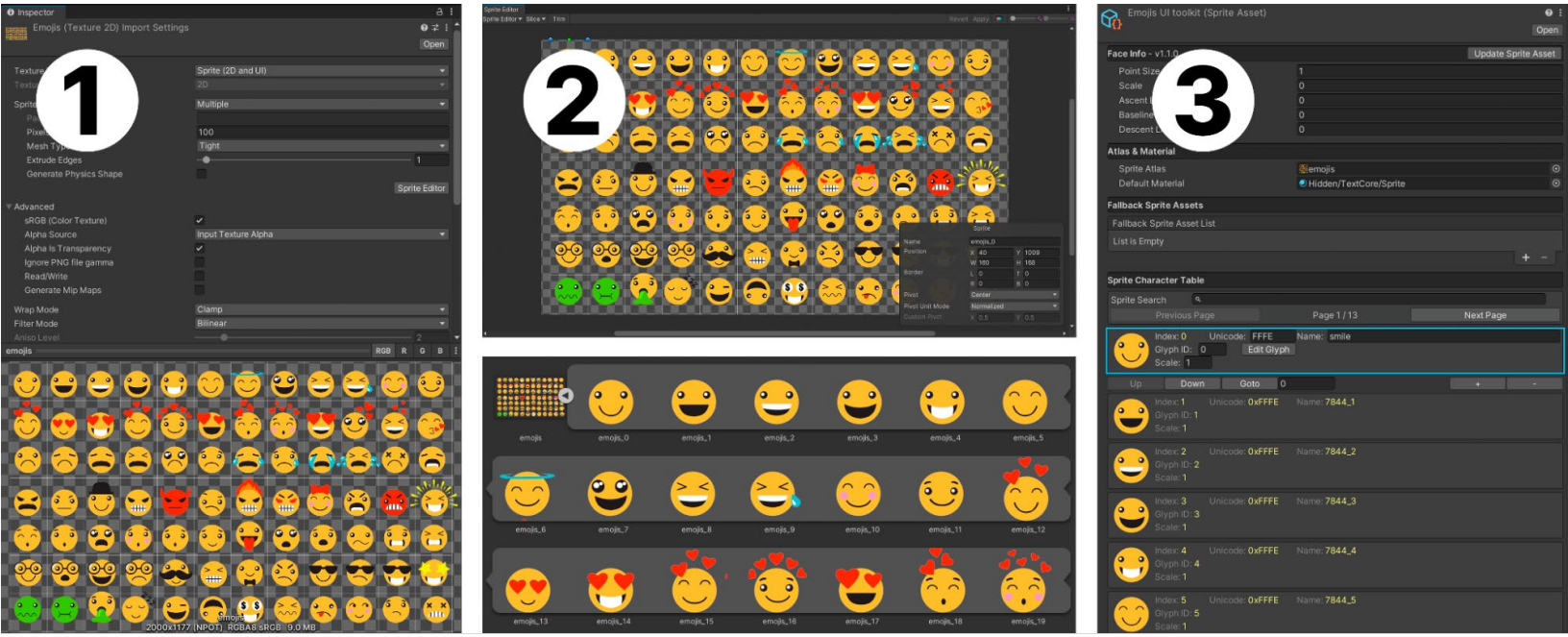
4. Тег color возвращает белый цвет шрифта, чтобы градиент выглядел правильно. Имя в теге gradient должно совпадать с именем ресурса, созданного на шаге 1. Убедитесь, что параметр Rich Text включён.

5. Результат можно увидеть непосредственно в UI Builder или в окне Game.

Ресурсы спрайтов и эмодзи

С помощью тегов форматированного текста в текст можно добавлять спрайты, например эмодзи. Для этого понадобится ресурс Sprite Asset, аналогичный ресурсу градиента.

Если импортируется несколько спрайтов, объедините их в один атлас, чтобы сократить число вызовов отрисовки. Убедитесь, что разрешение атласа спрайтов подходит для целевой платформы. Подробнее о разрешении спрайтов см. в разделе подготовки графики.



Распространённый сценарий применения Sprite Asset - добавление эмодзи и значков непосредственно в текстовые строки.

Чтобы импортировать спрайты для этой цели, выполните следующие действия:

1. Импортируйте файл спрайта или PSD с эмодзи либо значками.

2. Разделите изображение на отдельные спрайты. Если используется PSD-файл, описанный в разделе подготовки графических ресурсов и шрифтов, выполнять разрезание не нужно. Создайте Sprite Asset из файла: выберите его, затем воспользуйтесь командой Create > Text Core > Sprite Asset. Поместите ресурс в Assets/Resources или вложенную папку.

3. В новом Sprite Asset можно изменить Face Info, а также внешний вид и имена каждого «глифа». Эти параметры заменяют стандартные Face Settings из ресурса шрифта.

Примечание: в данном случае команда Update Sprite Asset синхронизирует Sprite Asset с последними изменениями в Sprite Editor.

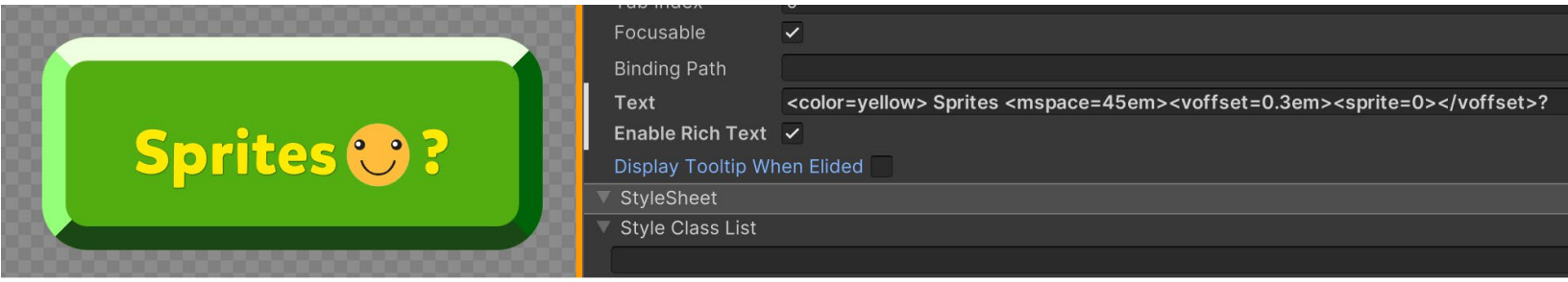
Чтобы использовать этот ресурс в UI Toolkit, выполните те же действия, что и для градиентов:

1. Выберите Panel Settings в UI Document.

2. Откройте ресурс Text Settings или создайте его, если он отсутствует.

3. В файле Text Settings с помощью окна выбора файла укажите Sprite Asset. Сохраните изменения и войдите в режим Play, чтобы новые параметры вступили в силу.

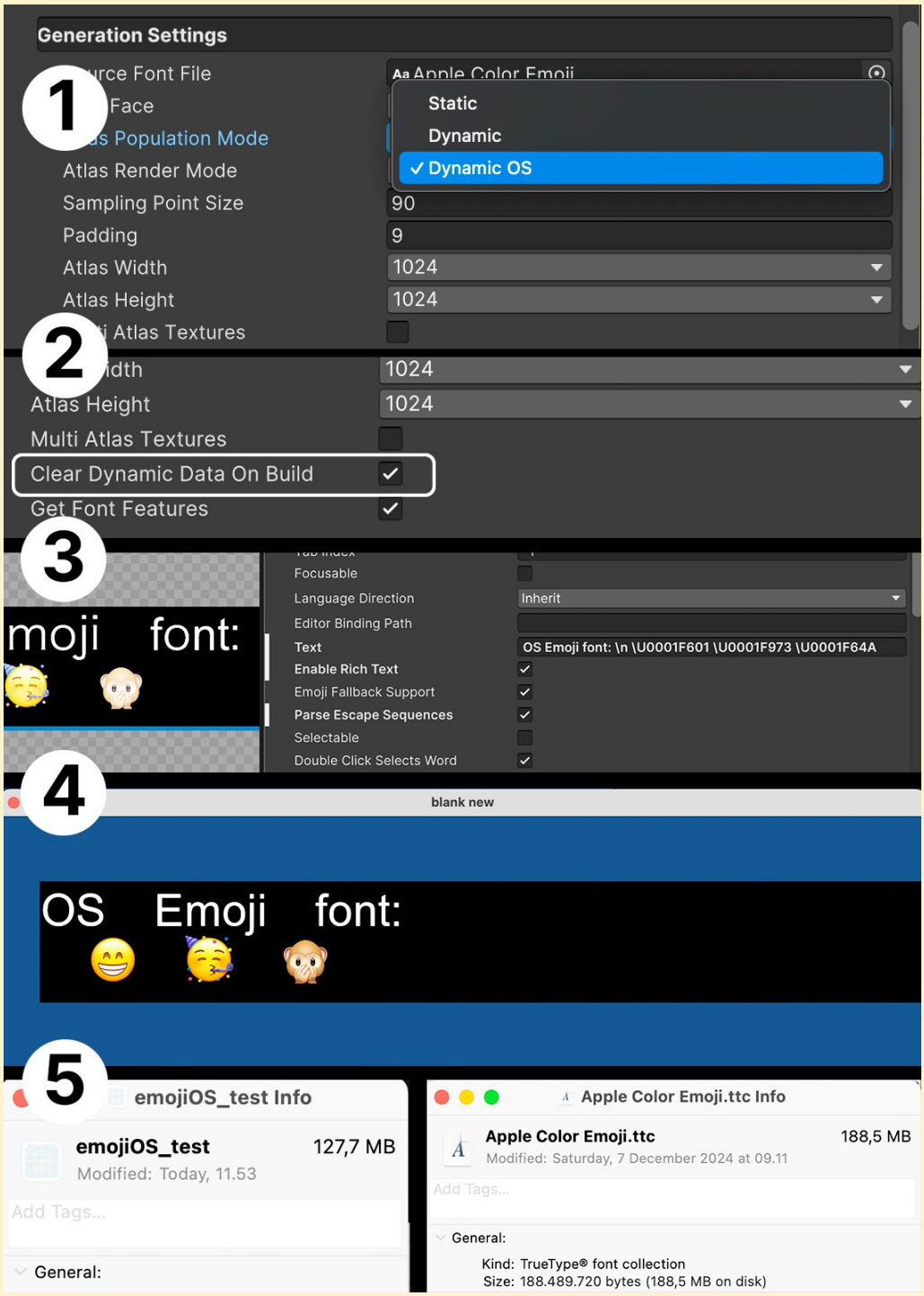
4. Добавьте спрайт с помощью тега форматированного текста: <sprite index=0> или <sprite name="name">. Встроенный спрайт также учитывает остальные текстовые теги.



Добавление Sprite Asset в текстовое поле UI Toolkit с помощью тегов форматированного текста. Убедитесь, что флажок Enable Rich Text установлен (вверху).

Совет: использование эмодзи операционной системы

Если приложение предназначено для конкретной платформы, например iOS или Android, вместо включения исходного шрифта в проект можно использовать встроенный системный шрифт эмодзи. Это экономит память и избавляет от необходимости поставлять с приложением большую коллекцию эмодзи. Такие шрифты также часто хорошо подходят для списка Global Fallbacks в Text Settings.



Чтобы использовать эмодзи операционной системы в проекте, выполните следующие действия:

1. Создайте ресурс шрифта на основе шрифта целевой системы. В iOS он называется Apple Emoji и используется в этом примере, а в Android - Noto Color Emoji; в настоящее время поддерживается только COLRv0. Выберите для Font Asset тип Color, затем установите режим заполнения атласа Dynamic OS. Он не требует включать исходный шрифт в ресурс и тем самым экономит место.

2. В Font Asset установите флажок Clean Dynamic Data On Build.

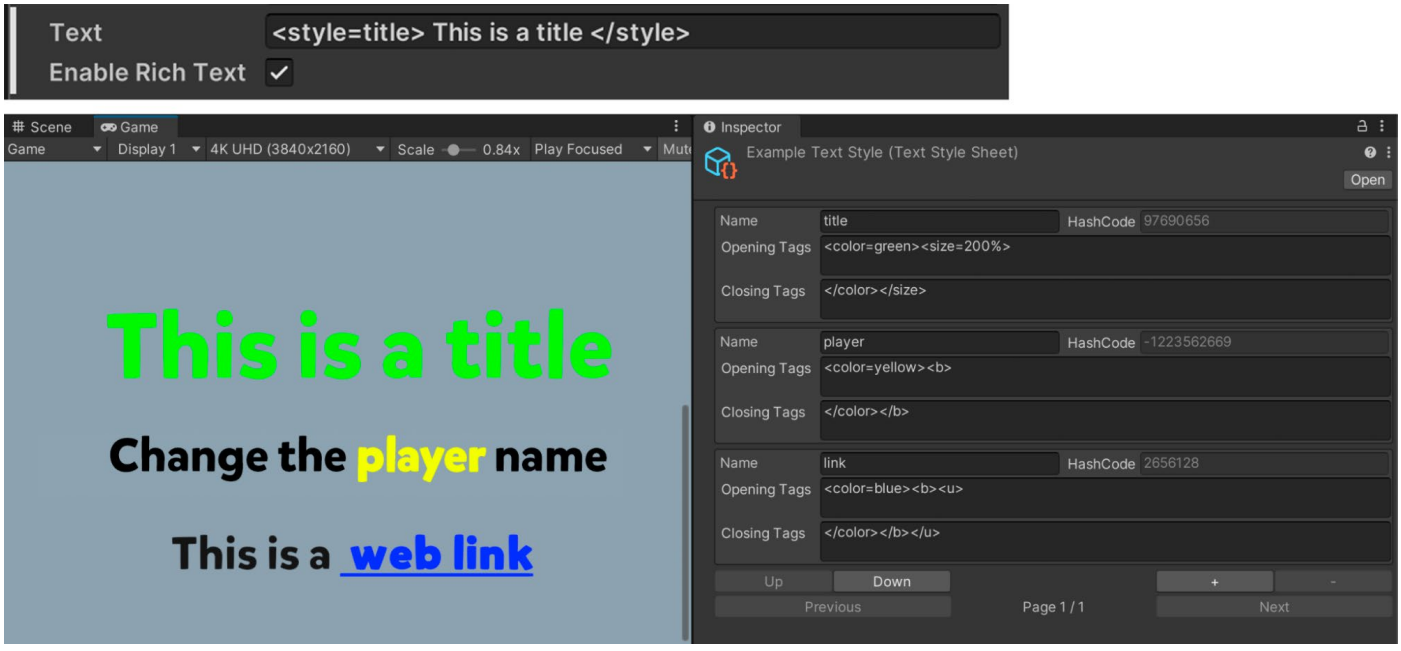
3. В UI Builder включите Parse Escape Sequences и введите нужные эмодзи с помощью панели эмодзи macOS или Windows либо в формате UTF. Например, улыбающееся лицо можно задать как \U0001F601. Код UTF каждого эмодзи можно посмотреть в Character Table ресурса шрифта.

4. Сборка, запущенная в macOS, отображает эмодзи с помощью системного шрифта.

5. В нашем тесте размер сборки оказался меньше размера отдельного шрифта эмодзи. Это подтверждает, что шрифт не был включён в проект, хотя применялся для отображения соответствующих эмодзи.

Таблицы стилей текста

Если в приложении много текста, для управления его оформлением стоит создать таблицу стилей текста. Она позволяет определять пользовательские стили, применяемые тегом форматированного текста <style>. Создать таблицу можно командой Assets > Text Core > Text Stylesheet в меню Create.



Повторно используемая таблица стилей текста

Таблицы стилей текста обладают следующими преимуществами:

- Пользовательский стиль может содержать открывающие и закрывающие теги форматированного текста, а также текст до и после них.

- Таблицу стилей текста удобно обновлять - намного удобнее, чем изменять форматирование непосредственно в тексте.

- Пользовательские стили сокращают число тегов форматирования. Достаточно одного тега <style=name>, который применяет всё необходимое оформление.

- Один тег форматированного текста проще изменить в таблице стилей, и вероятность ошибки ниже, чем при ручном редактировании множества тегов <style>.

Привязка данных

Пользовательский интерфейс связывает игроков с данными, управляющими приложением. Именно через него они видят внутреннее состояние и логику игры и взаимодействуют с ними.

Игроки не изучают необработанные значения характеристик - вместо этого они видят шкалу здоровья. Они не читают списки предметов напрямую, а пользуются инвентарём с перетаскиванием объектов. Такое взаимодействие интерфейса с данными влияет на структуру проекта.

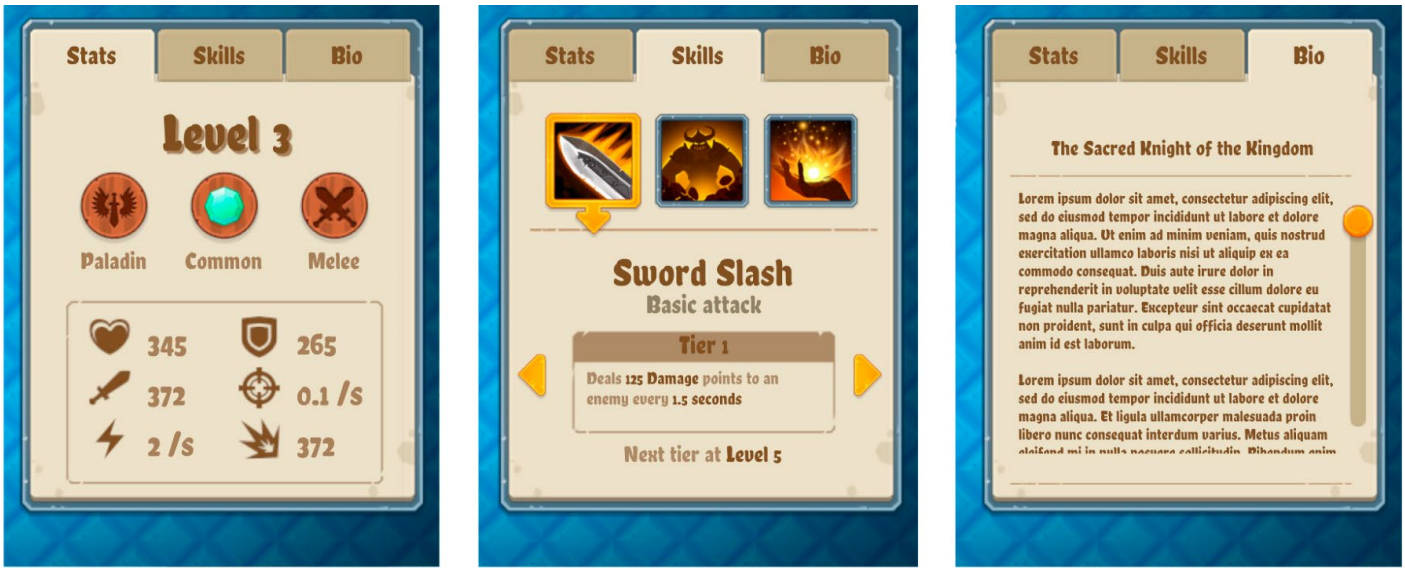
Интерфейс, отражающий игровые данные

Ниже показано окно характеристик персонажа из UI Toolkit Sample - Dragon Crashers. Этот интерфейс отображает основные параметры игры в жанре RPG.

Представление - это собственно интерфейс, с которым взаимодействует игрок. Контейнеры с вкладками упорядочивают способности персонажа и упрощают навигацию.

Окно характеристик персонажа отображает игровые данные.

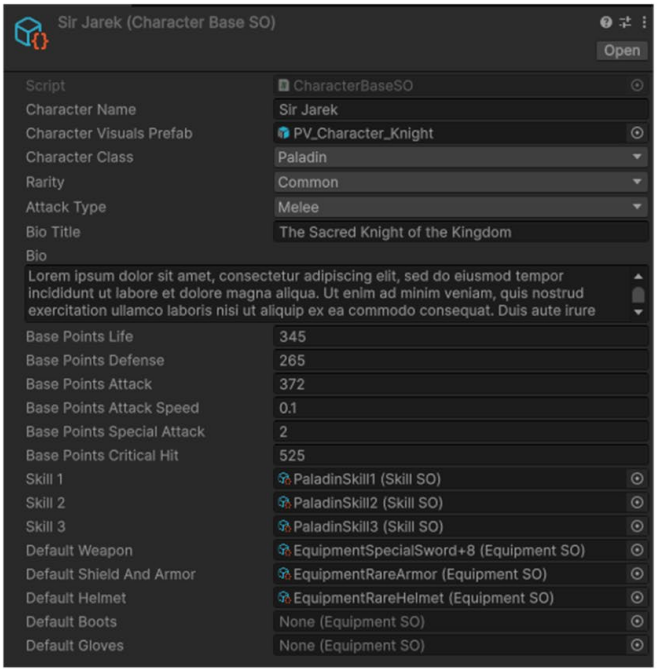
За интерфейсом стоит модель с данными - например, ScriptableObject, в котором хранятся характеристики каждого персонажа.

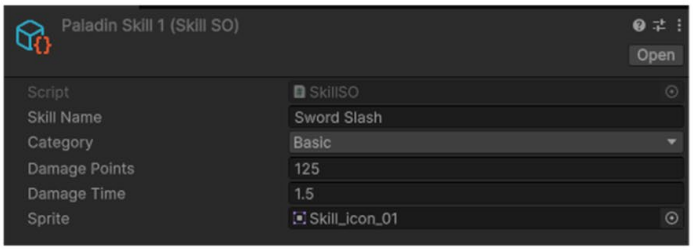


View (user interface)

↑

Model (data)





Ресурс ScriptableObject содержит данные персонажа.

Разделение представления и модели - один из основных принципов архитектуры пользовательских интерфейсов. Отделив визуальный интерфейс от данных, вы сделаете код гибче, пригоднее для повторного использования и удобнее в сопровождении.

Но после разделения модели и представления их необходимо синхронизировать. Традиционно для этого данные обновляют напрямую или используют систему событий: при изменении данных наблюдатели обновляют интерфейс. Такой подход работает, но приводит к повторяющемуся шаблонному коду.

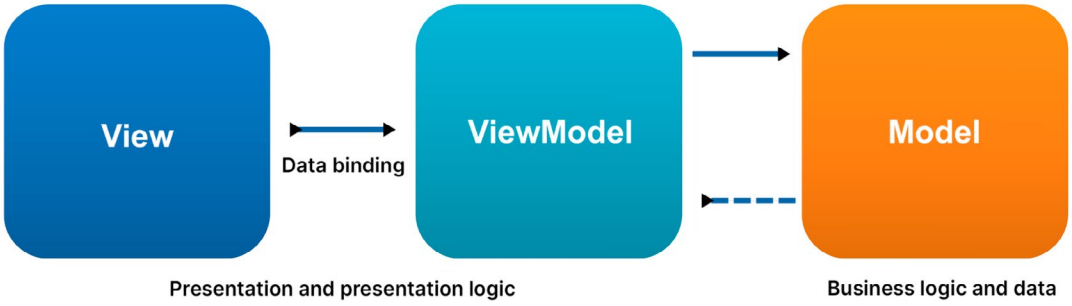
По мере роста проекта управлять системой становится всё сложнее: новые элементы и зависимости требуют дополнительной логики обновления или обработчиков событий. В результате сценарии разрастаются, их труднее читать и поддерживать.

Привязка данных во время выполнения

Привязка данных во время выполнения в Unity 6 предлагает более простой способ решить эту задачу. Данные приложения напрямую связываются с элементами интерфейса, поэтому изменения одной стороны автоматически отражаются на другой.

Архитектура Model-View-ViewModel (MVVM - «модель - представление - модель представления») добавляет между представлением и моделью слой логики отображения. ViewModel выступает посредником и предоставляет данные модели в формате, подходящем представлению.

Подробнее об MVVM и других шаблонах проектирования читайте в электронной книге Unity «Совершенствуйте код с помощью шаблонов проектирования и SOLID».



Архитектура MVVM (источник: Wikipedia)

Например, шкала здоровья может автоматически отображать запас здоровья игрока, а надпись со счётом - обновляться в реальном времени без дополнительной логики в сценариях и ручной обработки событий. Чем меньше кода синхронизации, тем проще масштабировать проект.

Рассмотрим примеры привязки данных UI Toolkit во время выполнения и способы её применения.

Основные понятия привязки данных

В Unity 6 появилась система привязки данных во время выполнения, которая обеспечивает структурированный способ связывать элементы интерфейса с данными приложения. Чтобы привязать свойство визуального элемента к источнику данных, создайте экземпляр DataBinding.

Ниже перечислены основные понятия:

- Data source: объект, содержащий данные для привязок интерфейса.

- Data source path: свойство или поле источника данных, с которым связывается элемент интерфейса.

- Binding mode: определяет направление передачи данных между источником и интерфейсом; привязка может быть односторонней или двусторонней.

Вместе эти части образуют привязку данных. Рассмотрим их подробнее.

Подготовка источника данных

Источник данных - это объект, содержащий данные для привязок интерфейса. Им может быть любой объект C#, включая ScriptableObject, MonoBehaviour или пользовательский объект C#. Структуры в роли источников данных способны повысить производительность благодаря небольшому объёму выделяемой памяти и меньшему числу сборок мусора. Привязку можно настроить как из кода, так и в Inspector.

В демонстрационном проекте источниками данных служат ScriptableObject, поскольку содержащиеся в них данные удобно сериализовать через Inspector Unity.

Использование атрибута CreateProperty

Чтобы свойства можно было использовать в привязках, UI Toolkit опирается на контейнеры свойств, создаваемые модулем Unity Properties. Они определяют, какие свойства источника данных доступны системе привязки.

Пометьте нужные свойства атрибутом CreateProperty, чтобы явно предоставить их системе. Ниже показан распространённый вариант настройки:

```
[SerializeField, DontCreateProperty]
int m_Value;

[CreateProperty]
public int Value
{
   get => m_Value;
   set => m_Value = value;
}
```

В этом примере поле m_Value помечено атрибутом SerializeField для сериализации, но исключено из привязки атрибутом DontCreateProperty.

Свойство Value, напротив, помечено CreateProperty и поэтому доступно системе привязки. Такое явное разделение упрощает управление потоком данных между моделью и интерфейсом.

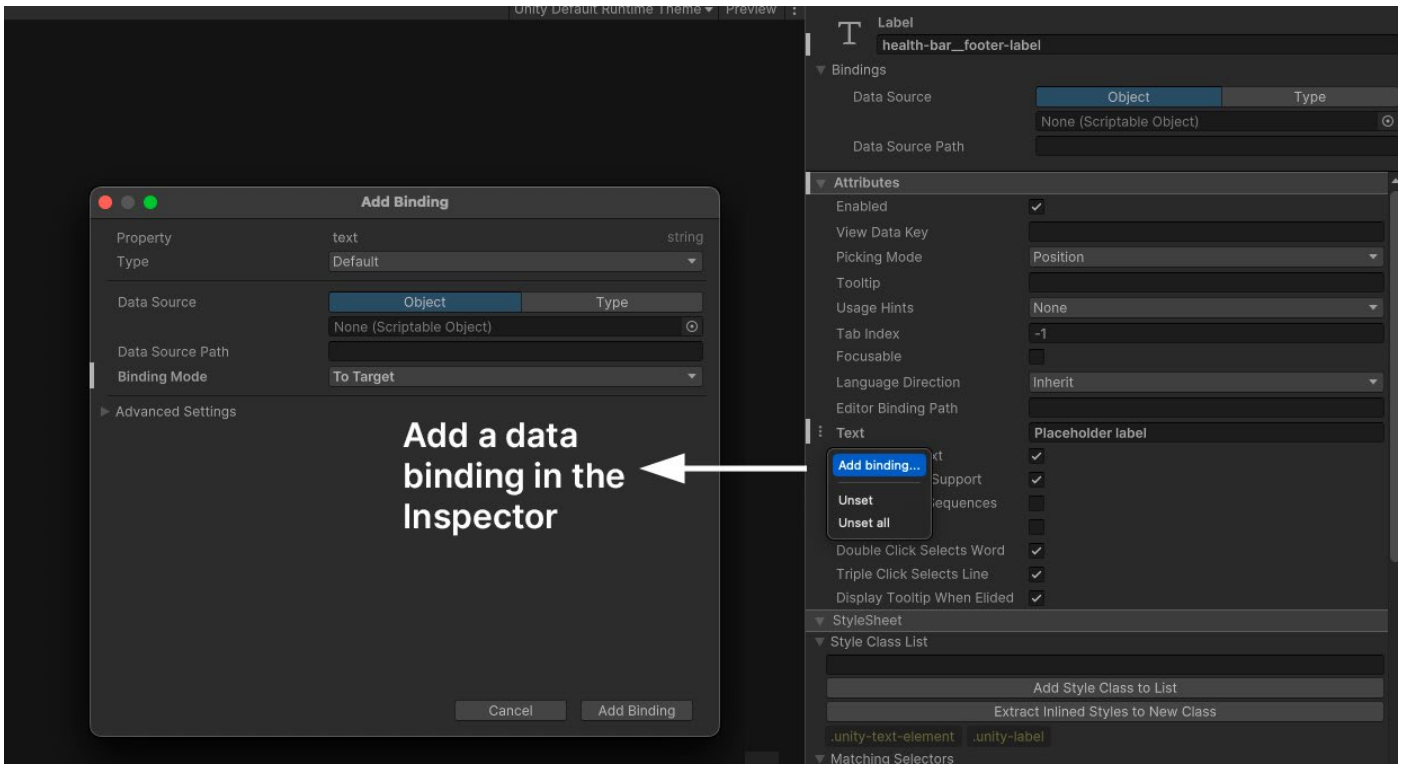
Привязки данных во время выполнения используют контейнеры свойств для эффективного обхода и изменения данных типа. По умолчанию Unity создаёт такой контейнер с помощью рефлексии при первом обращении к типу, что вызывает небольшие накладные расходы во время выполнения.

Чтобы избежать этого, помечайте объявляемые свойства атрибутом CreateProperty. Тогда код привязки создаётся во время компиляции, рефлексия во время выполнения не требуется, а накладные расходы снижаются.

Источники данных и пути

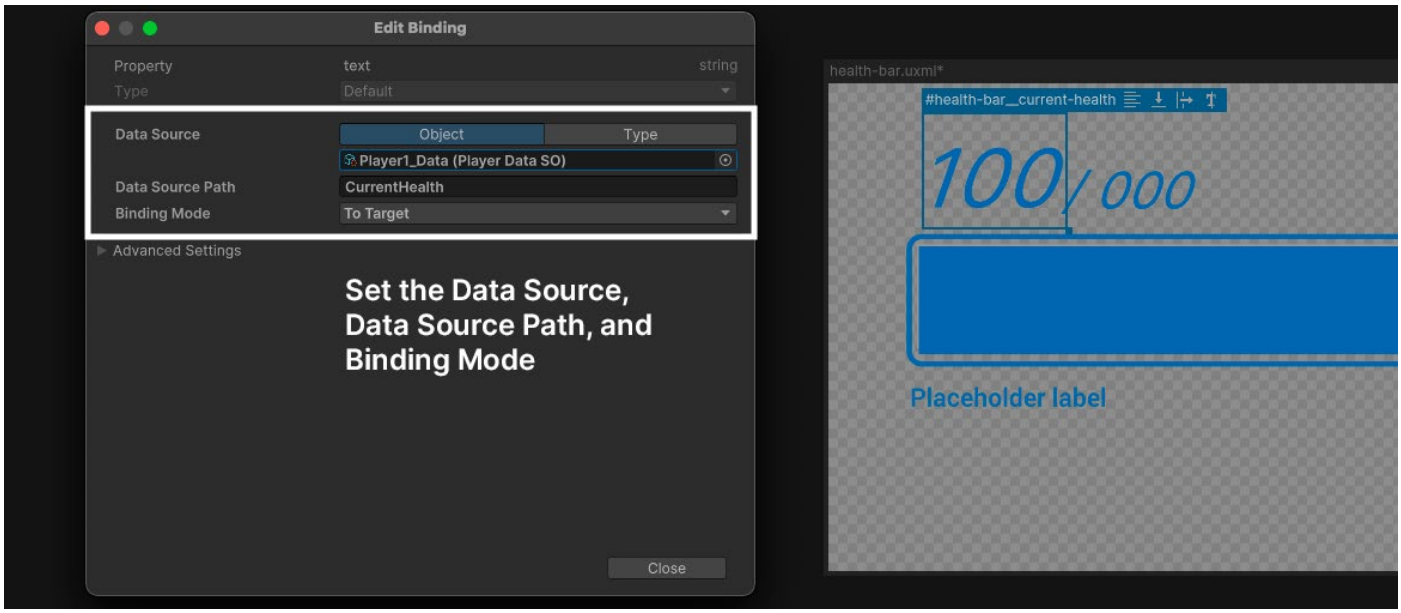
Когда источник данных готов, его можно связать с интерфейсом. Путь к источнику данных указывает свойство или поле, которое нужно связать с элементом интерфейса. Например, если источник содержит свойство health, путь будет указывать непосредственно на него - в UXML или в настройке привязки C#. Рассмотрим практический пример.

В UI Builder выберите элемент в Hierarchy, откройте Inspector и в меню параметров (⋮) выберите Add Binding.



Добавление привязки в Inspector.

Затем назначьте Data Source - например, ScriptableObject PlayerDataSO - и укажите Data Source Path, например CurrentHealth.



Настройка Data Source и Data Source Path в UI Builder.

В UXML: когда привязка данных настраивается в UI Builder, соответствующий код UXML создаётся автоматически. Путь к источнику данных также можно добавить или изменить вручную в текстовом редакторе. Ниже показан блок кода, создающий привязку:

```
<Bindings>
   <ui:DataBinding property="text" data-source-path="Health"/>
</Bindings>
```

В C#: создайте экземпляр объекта источника данных или получите ссылку на него в сценарии - например, на ScriptableObject. Присвойте объект свойству dataSource корневого элемента. Точное привязываемое свойство задайте через dataSourcePath.

В следующем фрагменте показано, как установить свойства dataSource и dataSourcePath из сценария. Подробнее этот способ рассматривается ниже, в разделе о настройке привязки данных на C#.

```
var label = new Label();
var parentData = ScriptableObject.CreateInstance<PlayerDataSO>();
playerData.Health = 100;

label.SetBinding("text", new DataBinding()
{
   dataSource = playerData,
   dataSourcePath = new PropertyPath(nameof(PlayerDataSO.Health)),
});
```

Примечание: если определить несколько привязок для одного элемента интерфейса, может возникнуть конфликт. Чтобы избежать путаницы:

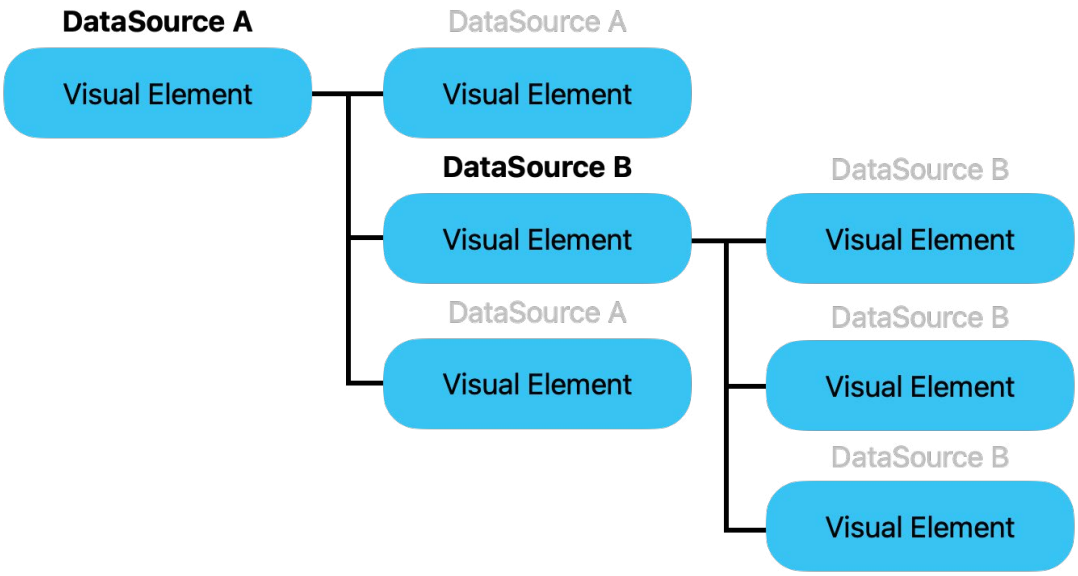
- Используйте привязки UI Builder/UXML для статических или стандартных конфигураций данных, которые не требуется изменять во время выполнения.

- Используйте привязки C# для динамических обновлений и случаев, когда источник данных должен меняться во время игры.

Можно также частично настроить привязку в UI Builder/UXML, а завершить её во время выполнения. Дополнительные сведения приведены ниже, в разделе «Работа с неразрешёнными привязками данных».

Наследование источников данных

Визуальные элементы автоматически наследуют источник данных родительского элемента, если им явно не назначен другой. Например, когда источник данных задан корневому элементу, все его дочерние элементы по умолчанию используют тот же источник. Это поведение показано на схеме:



Дочерний элемент может переопределить источник данных родительского элемента.

Если у родительского элемента есть источник данных, дочерние элементы автоматически его наследуют. В UI Builder поле Data Source дочернего элемента заранее заполняется источником родителя, но при необходимости значение можно переопределить.

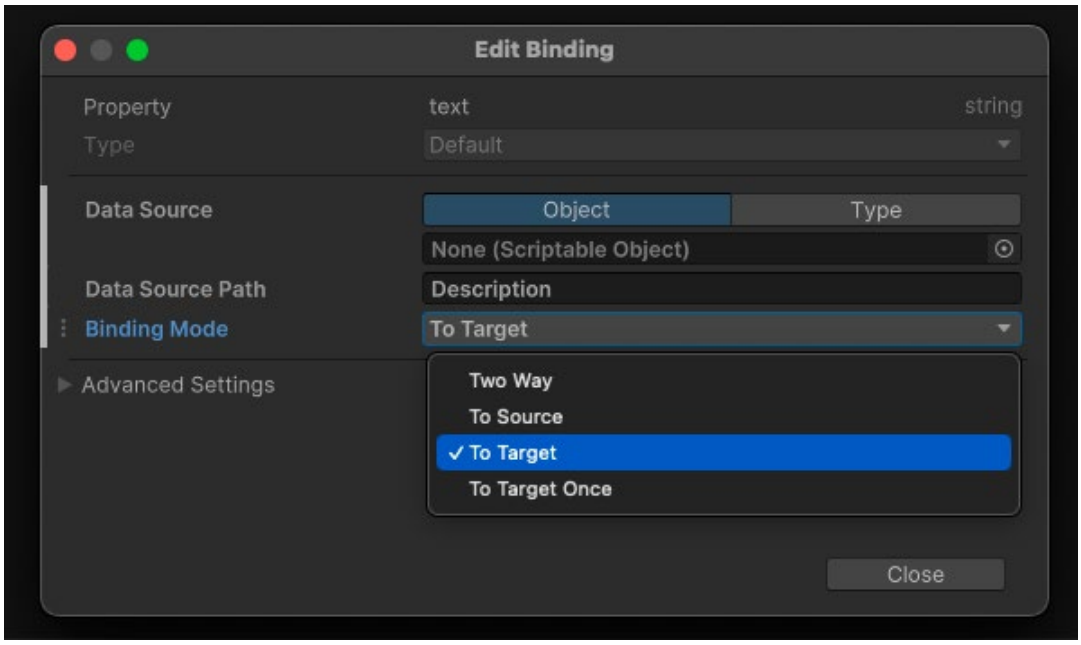
При работе с C# действует тот же принцип наследования, как показано в следующем примере:

```
var root = new VisualElement();
var parentData = ScriptableObject.CreateInstance<PlayerDataSO>();
parentData.Health = 100;

// Assign a data source to the root element
root.dataSource = parentData;

var child = new VisualElement();
var childData = ScriptableObject.CreateInstance<PlayerDataSO>();
childData.Health = 50;

// Override the inherited data source for the child
child.dataSource = childData;

root.Add(child);
```

Здесь дочерний элемент переопределяет значение родителя и получает независимый источник данных.

Режимы привязки

Режим привязки определяет направление передачи данных между источником и интерфейсом.



Режимы привязки в UI Builder позволяют управлять потоком данных между источником данных и интерфейсом.

В UI Builder и API C# доступны следующие режимы:

- TwoWay (по умолчанию): изменения передаются в обе стороны - из источника данных в интерфейс и из интерфейса в источник. Используйте этот режим для интерактивных элементов, через которые пользователь может менять данные, например ползунков и текстовых полей.

- ToTarget: данные передаются только из источника в интерфейс. Подходит для элементов интерфейса, доступных только для чтения.

- ToSource: данные передаются только из интерфейса в источник. Полезно для полей ввода, в которых не требуется изначально отображать текущее значение.

- ToTargetOnce: данные передаются из источника в интерфейс один раз; последующие изменения источника не отслеживаются.

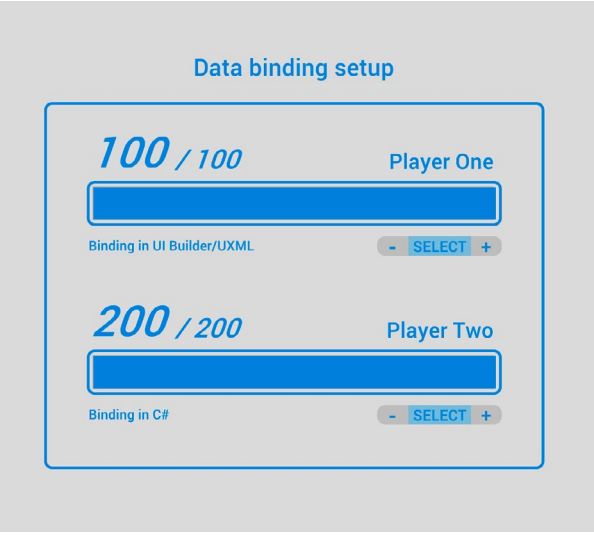
Пример: привязка данных шкалы здоровья

Рассмотрим практический пример базовой привязки данных в UI Toolkit. В демонстрационной сцене используется простая шкала здоровья, которая динамически обновляется в соответствии с запасом здоровья игрока.

Демонстрационная сцена. Следующие примеры входят в демонстрацию Data Binding проекта QuizU.

Чтобы открыть её во время выполнения, в главном меню выберите Demos > Data Binding. Также можно отключить загрузчик командой Quiz > Don't Load Bootstrap Scene on Play и загрузить сцену DataBindingDemo напрямую.

В демонстрации есть две шкалы здоровья: привязки одной созданы в UXML с помощью UI Builder, а привязки другой - в C#.



Шкала здоровья отображает данные игрока.

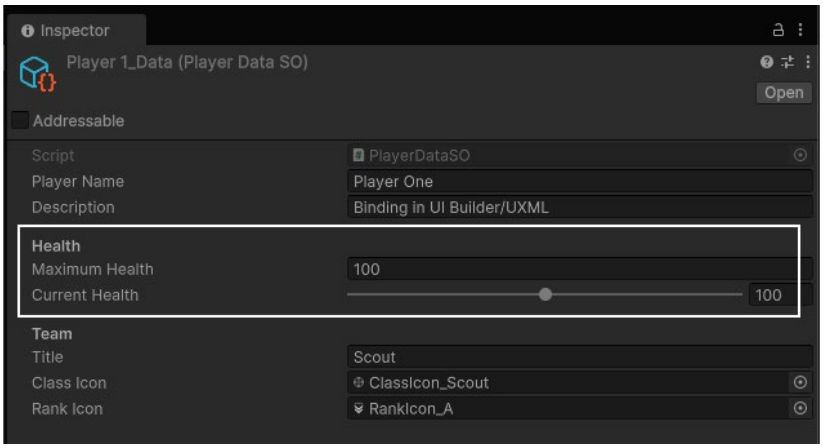
Подготовка источника данных

Сведения и характеристики игрока в примере хранятся в ScriptableObject PlayerDataSO. Нужные свойства PlayerDataSO помечены атрибутом CreateProperty, поэтому их можно использовать в привязках.

Каждая шкала здоровья представляет лишь часть данных PlayerDataSO: имя игрока и значения здоровья. Во фрагменте класса показаны некоторые свойства и связанные с ними поля:

```
using System;
using Unity.Properties;
using UnityEngine;
using UnityEngine.UIElements;

[CreateAssetMenu(fileName = "PlayerDataSO", menuName = "Demos/Player_Data")]
public class PlayerDataSO : ScriptableObject
{

    [CreateProperty] public string PlayerName => m_PlayerName;
    [CreateProperty] public int CurrentHealth => Mathf.Clamp(m_CurrentHealth, 0,
m_MaximumHealth);
    [CreateProperty] public int MaximumHealth => m_MaximumHealth;

    [SerializeField] string m_PlayerName;
    [SerializeField] int m_MaximumHealth = 100;
    [SerializeField] [Range(0, k_MaxHealthRange)]
    int m_CurrentHealth = 100;

    const int k_MaxHealthRange = 200;
```

Для отображения этих сведений на экране интерфейс использует конкретные пути к данным, например PlayerName, CurrentHealth и MaximumHealth.



Источник данных содержит сведения о здоровье из ScriptableObject PlayerDataSO.

Привязка данных в UI Builder/UXML

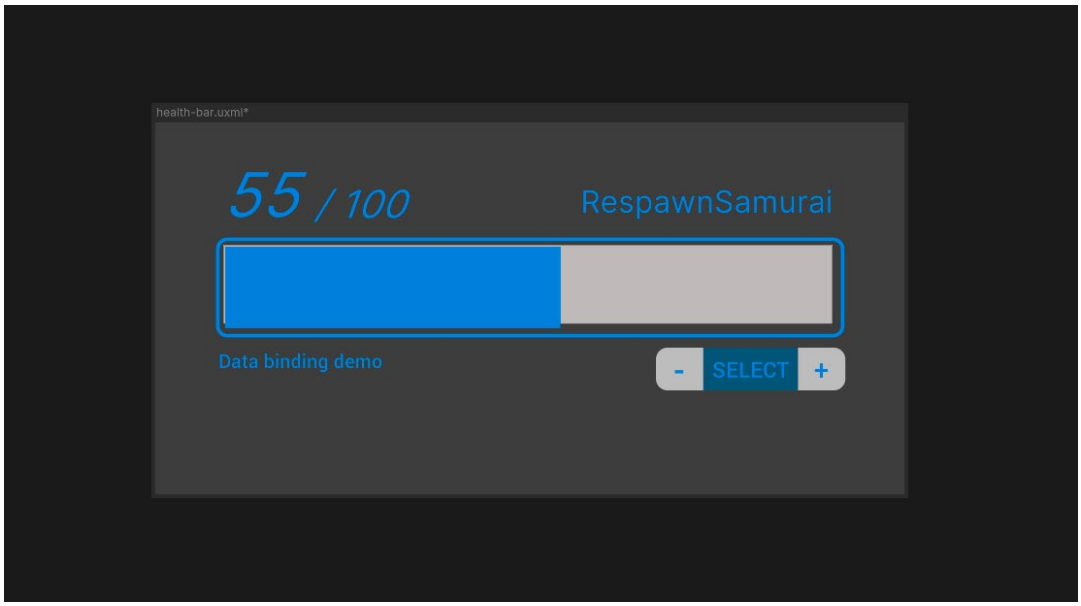
UI Builder предлагает наглядный интерактивный способ связывать элементы интерфейса с данными. Он удобен художникам интерфейсов, предпочитающим работать с дизайном визуально, и разработчикам, которым важна мгновенная обратная связь во время настройки. Кроме того, это полезный учебный инструмент для тех, кто только знакомится с привязкой данных.

В демонстрационной сцене все привязки шкалы здоровья Player One настроены в UI Builder. Для этого необходимо:

- Выбрать корневой элемент: в иерархии выберите корневой элемент, содержащий шкалу здоровья. В данном примере самый верхний контейнер - элемент demo_container-uxml.

- Назначить источник данных: в разделе Data Binding окна Inspector укажите ресурс ScriptableObject как источник. Он будет назначен выбранному элементу и унаследован всеми дочерними элементами.

- Задать пути к источнику данных: укажите пути, связывающие отдельные элементы интерфейса с соответствующими свойствами ScriptableObject, например PlayerDataSO.PlayerName.



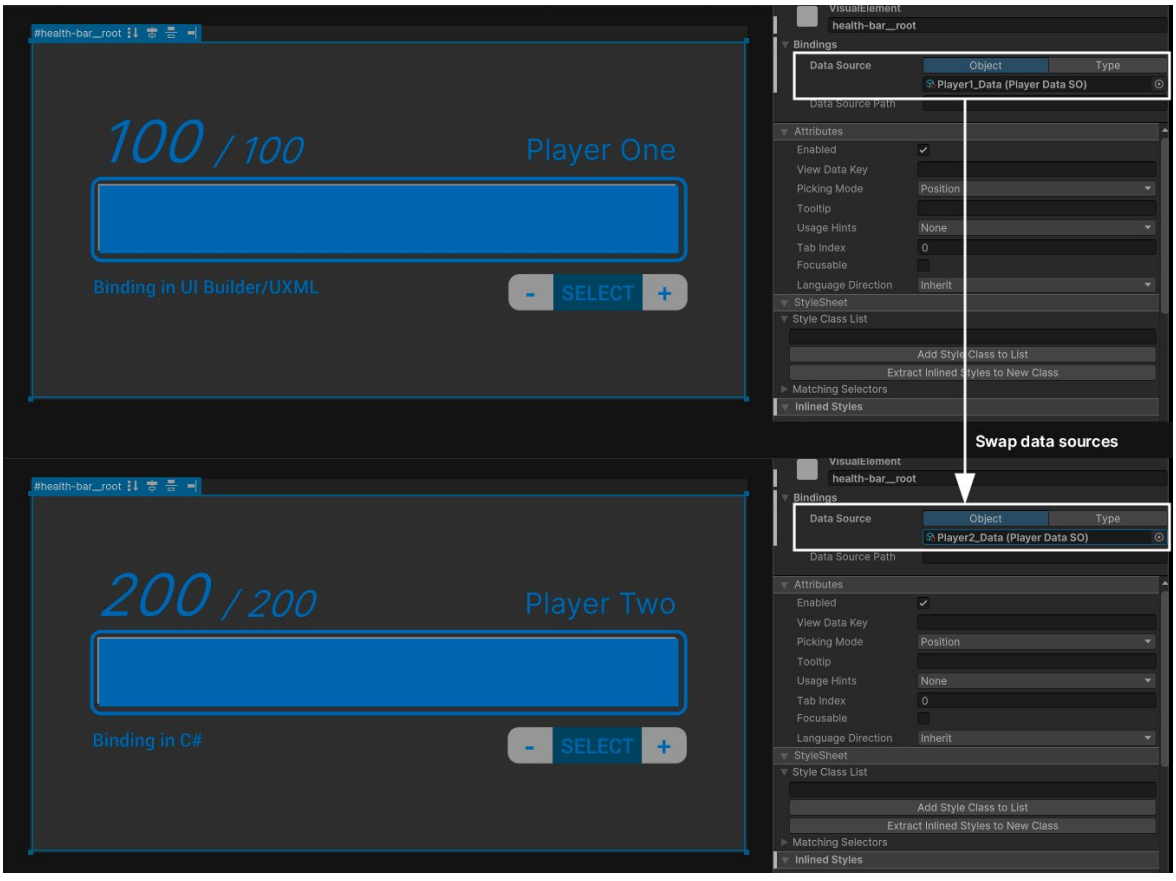
Базовая шкала здоровья

После назначения источника данных корневому элементу он должен появиться как источник по умолчанию у дочерних элементов. Остаётся лишь указать правильный путь к данным. В таблице показано, как привязки соединяют свойства элементов интерфейса со свойствами ScriptableObject:

Элемент UI	Свойство элемента UI	Привязанное свойство	Примечание
health-bar__player-name	text	PlayerName	Имя игрока
health-bar__current-health	text	CurrentHealth	Текущее здоровье
health-bar__max-health	text	MaximumHealth	Максимальное здоровье
health-bar__progress	style.width	Progress	Динамическая ширина шкалы

После завершения настройки шкала здоровья обновляется в реальном времени, отображая значения и полосу текущего здоровья игрока.

Источник данных легко заменить: назначьте другой ресурс ScriptableObject, и интерфейс автоматически покажет новые значения, сохранив прежние привязки.



При смене источника данных привязки обновляются.

Привязки, настроенные в UI Builder, добавляются непосредственно в файл UXML: для каждого привязанного элемента создаётся блок <Bindings>.

Ниже приведён фрагмент созданного UXML, в котором свойство text элемента health-bar__player-name связано со свойством PlayerName. Для удобства чтения некоторые атрибуты опущены:

```
<ui:Label text="Placeholder" name="health-bar__player-name" class="health-bar__player-name">
   <Bindings>
       <ui:DataBinding property="text" data-source-path="PlayerName" binding-mode="ToTarget"
/>
   </Bindings>
</ui:Label>
```

Опытные пользователи могут создавать такие привязки непосредственно в UXML. При большом числе привязок ручное редактирование кода может обеспечить более точный контроль и ускорить работу. Написанный вручную UXML также формирует более понятные различия в системе контроля версий, поэтому конфликты слияния проще разрешать, а изменения - отслеживать.

Настройка привязки данных в C#

UI Builder отлично подходит для прототипирования со статическими данными, например заранее созданными ресурсами ScriptableObject. Однако динамические данные во время выполнения часто удобнее обрабатывать в C#. В следующем примере кода показано, как в демонстрационной сцене работает шкала здоровья Player Two:

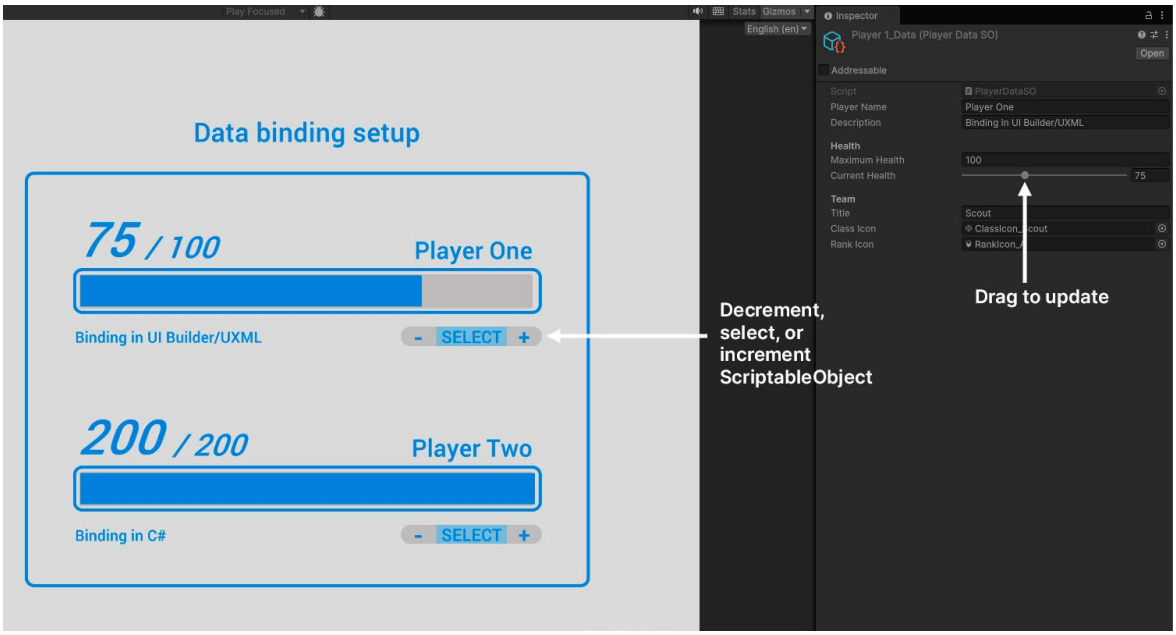
```
using UnityEngine;
using UnityEngine.UIElements;
using Unity.Properties;

public class HealthBar : MonoBehaviour
{
       [SerializeField] PlayerDataSO m_HealthData;

       public void Initialize(VisualElement root)
       {
           var m_PlayerName = root.Q<Label>("health-bar__player-name");

           root.dataSource = m_HealthData;

           m_PlayerName.SetBinding("text", new DataBinding()
           {
               dataSourcePath = new PropertyPath(nameof(PlayerDataSO.PlayerName)),
               bindingMode = BindingMode.ToTarget
           });
       }
}
```

Сценарий HealthBar выполняет настройку в методе Initialize, который основной управляющий сценарий вызывает из OnEnable.

- Сначала выполняется запрос элемента health-bar__player-name. Затем данные ScriptableObject назначаются ему как источник.

- После этого метод SetBinding связывает свойство text с новым экземпляром DataBinding и задаёт параметры dataSourcePath и bindingMode.

Все четыре привязки из приведённой выше таблицы настраиваются аналогично. Изменяйте CurrentHealth ползунком ScriptableObject или пользовательским Property Drawer редактора. Демонстрация содержит элементы управления для тестирования в режиме Play: +, - и Select позволяют увеличивать или уменьшать значение либо выбирать ScriptableObject. Шкала здоровья динамически обновляется при каждом изменении.

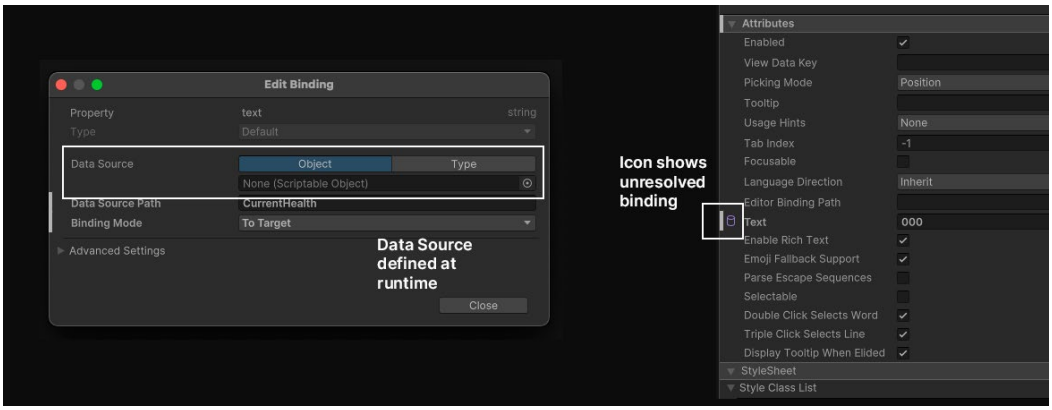


HealthBar синхронизируется со значением CurrentHealth.

Работа с неразрешёнными привязками данных

Unity 6 также поддерживает гибридный рабочий процесс, сочетающий визуальную настройку в UI Builder с гибкостью сценариев.

Вместо жёсткого задания источника данных в UXML можно указать Data Source Type, оставив сам источник неразрешённым. UI Builder отмечает такие незавершённые привязки полым значком. Это означает, что пути и типы уже заданы, но источник данных ещё не назначен.



Неразрешённая привязка данных отмечается полым значком.

Во время выполнения источник данных можно назначить одной строкой кода. Например:

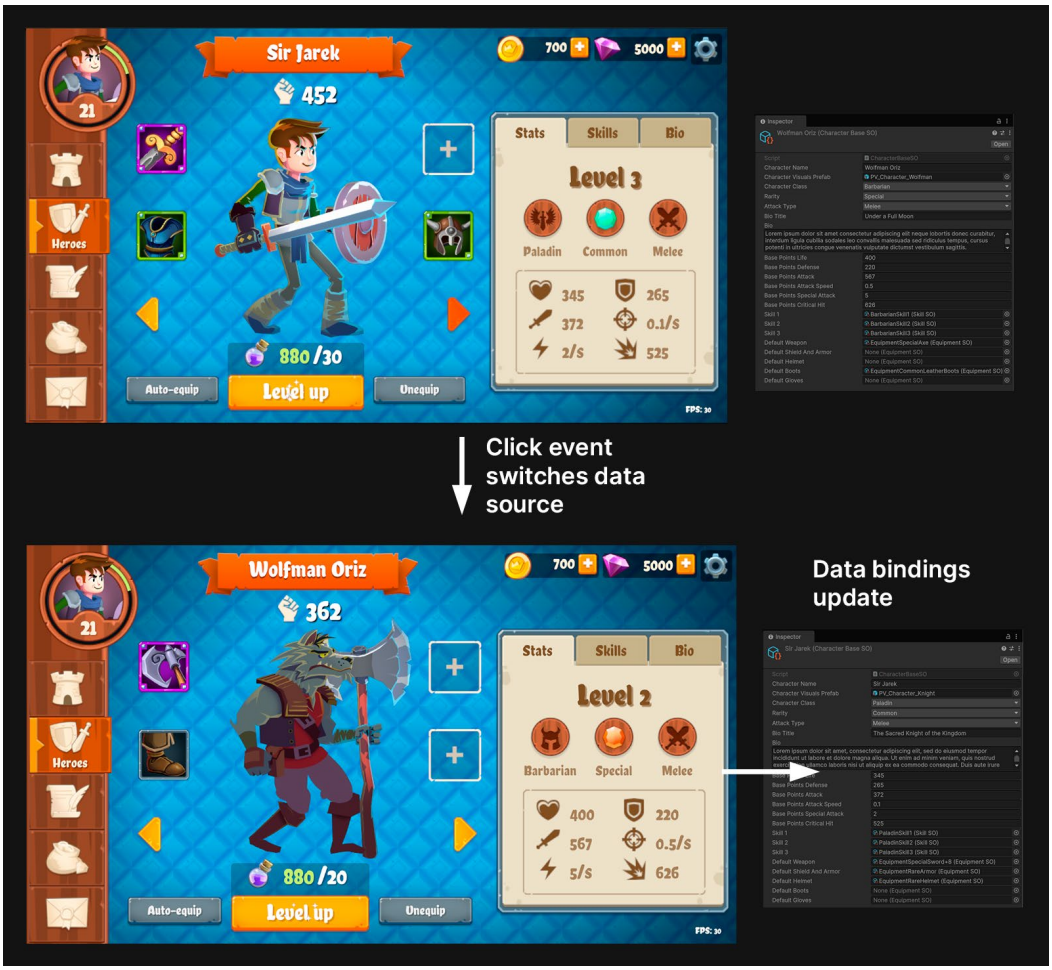
```
myElement.dataSource = myNewDataSource;
```

Здесь назначение myNewDataSource элементу myElement разрешает привязки-заполнители, определённые в UXML, после чего интерфейс обновляется автоматически. Это устраняет повторяющиеся вызовы SetBinding и сохраняет гибкость UXML.

Например, в проекте Dragon Crashers пути к данным заранее определены в UXML, а фактические источники назначаются во время выполнения.

При нажатии кнопок перехода к следующему и последнему персонажу выбранный персонаж становится текущим источником данных. Для смены источника изменять UXML не требуется.

После назначения нового источника неразрешённые привязки отображают правильные характеристики персонажа.



Обновление источника данных в примере Dragon Crashers

Примечание: если в UXML указан конкретный источник данных, например data-source="PlayerDataSO.asset", привязка становится фиксированной, и изменить её во время выполнения нельзя. Чтобы разрешить изменение, оставьте атрибут data-source пустым или используйте data-source-type.

Пример такого гибридного процесса приведён в разделе о привязке списка к ListView.

Конвертеры типов

Конвертеры типов в Unity 6 преобразуют исходные данные в более удобный для пользователя формат отображения. Они работают как посредники между источником данных и интерфейсом, представляя значения в интуитивно понятном виде.

Например, конвертер может переводить радианы в градусы или преобразовывать числовой запас здоровья в цвет шкалы. Так интерфейс показывает информацию ясно и наглядно, а писать большой объём логики преобразования вручную не приходится.

Unity 6 поддерживает две категории конвертеров типов:

- Глобальные конвертеры: применяются к любым привязкам, которым требуется определённое преобразование типа. Например, глобальный конвертер может преобразовывать любое значение здоровья типа float в цвет или объекты Color в значения StyleColor, обеспечивая единообразное поведение во всём интерфейсе.

- Конвертеры отдельных привязок: применяются к конкретным привязкам данных и обеспечивают более точное управление.

Пример: преобразование значения в цвет

Шкала, меняющая цвет в зависимости от здоровья игрока, наглядно демонстрирует привязку данных с конвертером типа. Текущее здоровье сопоставляется с цветовым градиентом: например, зелёный означает высокий запас здоровья, жёлтый - низкий, красный - критический. Благодаря этому игрок быстро оценивает своё состояние во время игры.

Пример можно увидеть в сцене DataBindingDemo проекта QuizU.

Настройка HealthDataConverter

В сцене DataBindingDemo класс HealthBarWithConverter использует функции статического класса HealthDataConverter, чтобы зарегистрировать несколько DataConverter:

- Процент здоровья управляет цветовым градиентом шкалы: от зелёного при полном здоровье до красного при критическом.

- Одна надпись представляет числовое значение строкой с процентом, например «75%».

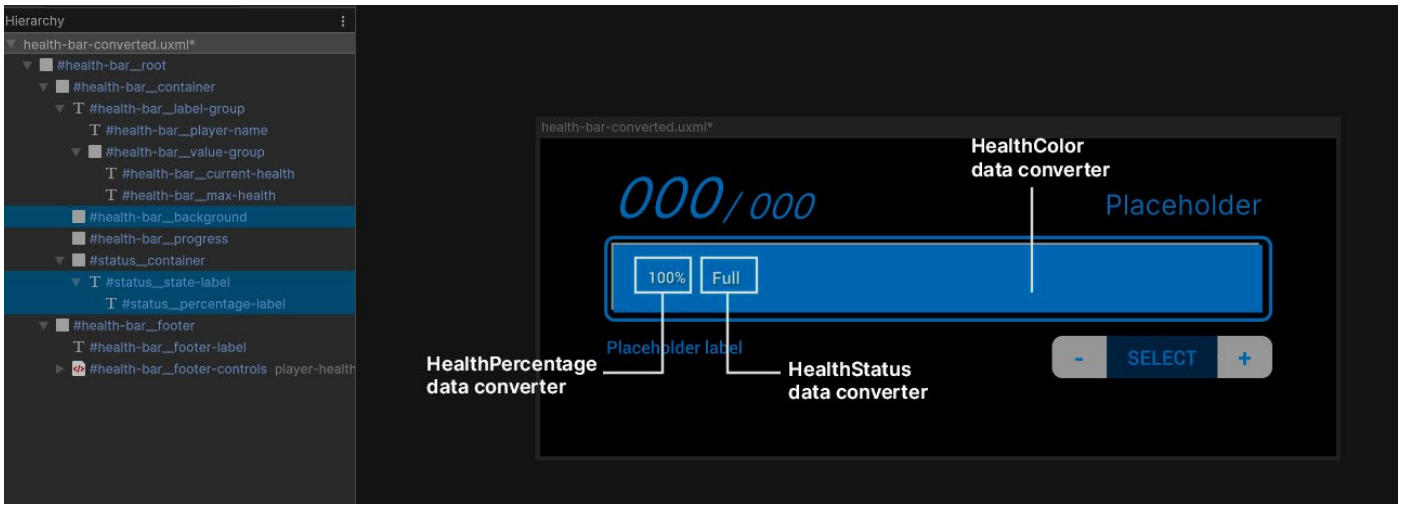
- Другая сопоставляет тот же процент здоровья текстовому состоянию, например Full, Mid или Critical.

Ниже приведён фрагмент класса HealthDataConverter:

```
public static class HealthDataConverter
{
   static readonly Color s_FullColor = new Color(0.2f, 1f, 0.2f);
   static readonly Color s_MidColor = Color.yellow;
   static readonly Color s_LowColor = new Color(1f, 0.3f, 0f);
   static readonly Color s_CriticalColor = Color.red;

   public static void Register()
   {
       RegisterHealthColorConverter();
       // …
   }

   static void RegisterHealthColorConverter()
   {
       var colorConverter = new ConverterGroup("HealthColor");

       colorConverter.AddConverter((ref float healthPercentage) =>
       {
           if (healthPercentage > 0.5f)
           {
               return new StyleColor(Color.Lerp(s_MidColor, s_FullColor,
(healthPercentage - 0.5f) * 2f));
           }
           else if (healthPercentage > 0.25f)
           {
               return new StyleColor(Color.Lerp(s_LowColor, s_MidColor,
(healthPercentage - 0.25f) * 4f));
           }
           else
           {
               return new StyleColor(Color.Lerp(s_CriticalColor, s_LowColor,
healthPercentage * 4f));
           }
       });

       ConverterGroups.RegisterConverterGroup(colorConverter);
   }

   // …

}
```

Приведённая выше логика создаёт ConverterGroup HealthColor, который преобразует долю здоровья типа float в диапазоне от 0 до 1 в соответствующее значение StyleColor между красным при низком здоровье и зелёным при полном.

Класс HealthDataConverter также содержит конвертеры для двух надписей. Они представляют свойство HealthPercentage объекта PlayerDataSO в виде форматированных строк. Несколько конвертеров можно объединить в одну ConverterGroup, однако для удобства чтения в демонстрации они разделены по разным группам.



Использование конвертеров типов в UI Builder.

Использование HealthBarWithConverter

Обратите внимание: вся фактическая логика находится в классе HealthDataConverter. Класс HealthBarWithConverter лишь выполняет следующее:

```
public class HealthBarWithConverter : HealthBar
{
#if UNITY_EDITOR
    [UnityEditor.InitializeOnLoadMethod]
        #else
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.SubsystemRegistration)]
        #endif
        public static void RegisterConverters()
    {
        HealthDataConverter.Register();
    }
}
```

Обратите внимание на следующее:

- UnityEditor.InitializeOnLoadMethod регистрирует ConverterGroup и делает её доступной в UI Builder, чтобы группу можно было увидеть и применить в редакторе.

- RuntimeInitializeOnLoadMethod обеспечивает доступность ConverterGroup во время выполнения игры.

Директива препроцессора #if UNITY_EDITOR гарантирует вызов нужного метода в зависимости от того, выполняется код в редакторе или во время игры.

Применение DataConverter в UI Builder

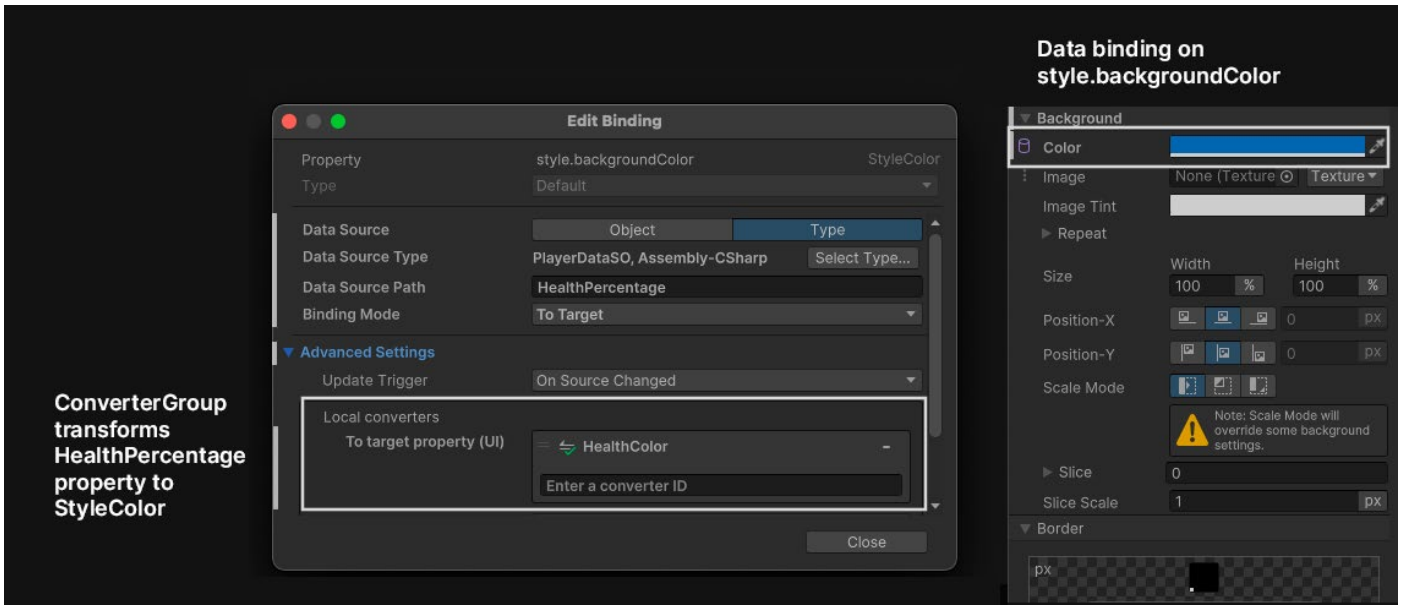
После регистрации DataConverter можно применить к любой привязке, которой требуется это преобразование. Чтобы использовать его непосредственно в UI Builder:

1. Откройте файл UXML и выберите элемент шкалы. В проекте QuizU пример настройки можно посмотреть в файле RuntimeDataBinding.uxml.

2. Назначьте ScriptableObject PlayerDataSO в качестве Data Source.

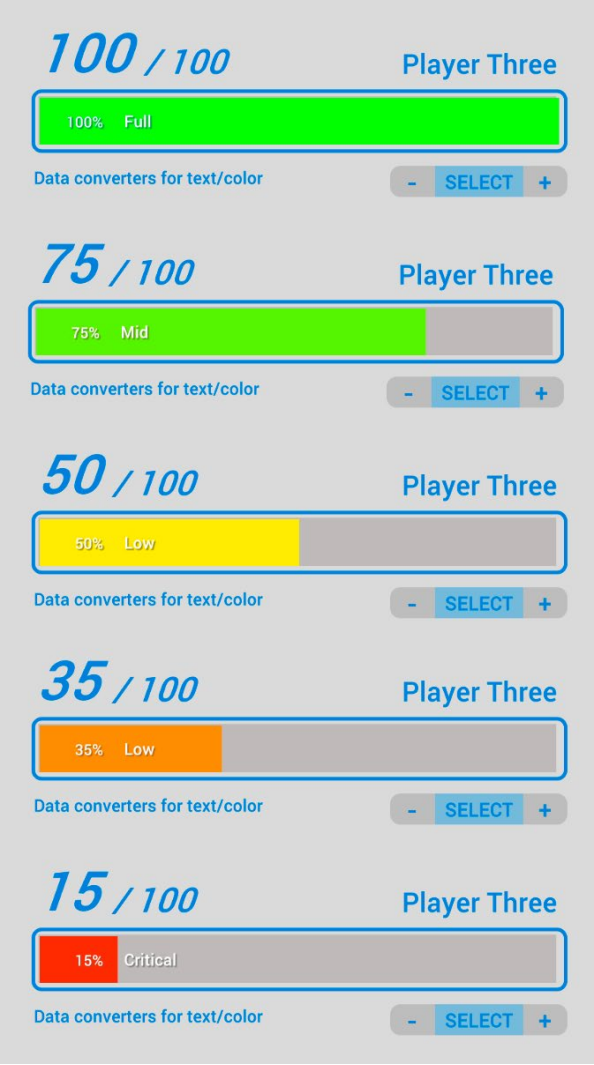
3. Свяжите стилевое свойство backgroundColor шкалы с путём данных HealthPercentage.

4. Используйте ConverterGroup HealthColor, чтобы преобразовать процент здоровья в цвет фона шкалы.



Настройка привязки данных для шкалы здоровья.

5. Теперь при перетаскивании ползунка CurrentHealth в ScriptableObject PlayerDataSO цвет шкалы здоровья обновляется. Градиент плавно интерполируется от зелёного при полном здоровье через жёлтый при среднем и оранжевый при низком к красному при критическом.



Конвертер HealthColor изменяет цвет шкалы.

Теперь этот глобальный DataConverter доступен во всём приложении, где требуется преобразовать значение float в такой цветовой градиент.

Рекомендации

При работе с конвертерами типов придерживайтесь следующих рекомендаций:

- Сведите выделения памяти к минимуму: делегаты преобразования должны оставаться легковесными, особенно при частых вызовах, чтобы избежать лишних накладных расходов.

- Не усложняйте: создавайте простые специализированные конвертеры для быстрых преобразований. Не помещайте в них сложную или ресурсоёмкую логику.

- Выполняйте преобразование в источнике данных: базовые преобразования обрабатывайте непосредственно в источнике - например, заранее форматируйте процент здоровья в свойстве ScriptableObject. Оставляйте DataConverter для преобразований, относящихся именно к привязкам интерфейса.

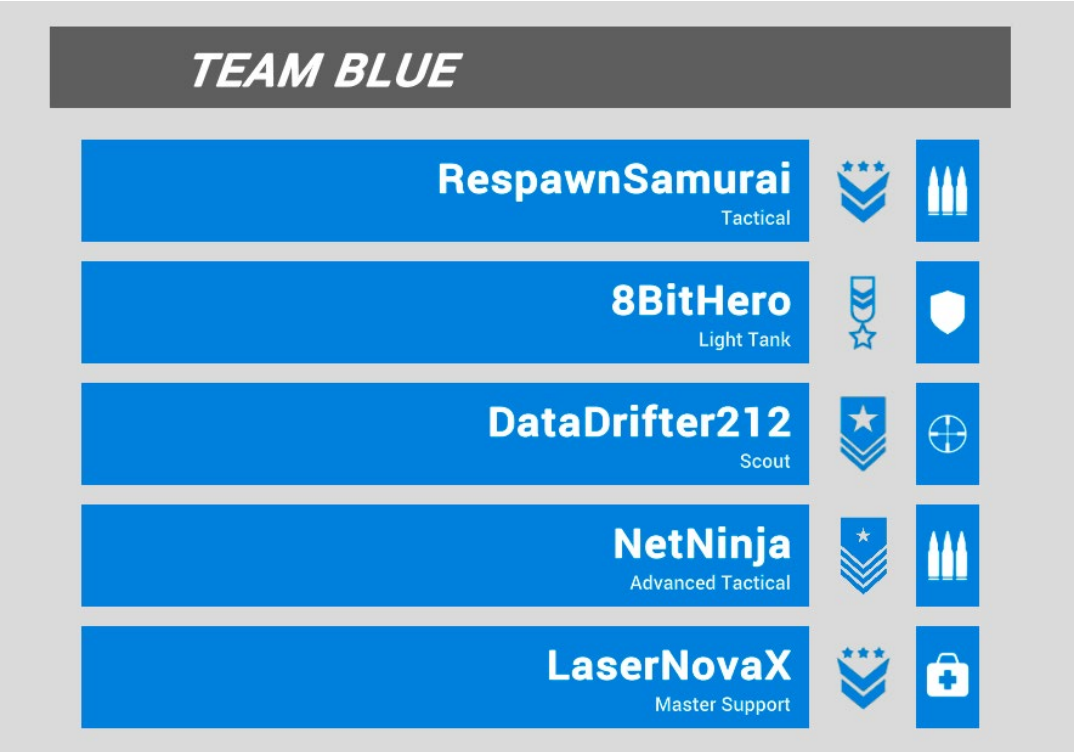
Пример: привязка списка к ListView

В зависимости от интерфейса игры приложению может потребоваться отображать разные коллекции данных: инвентарь собранных предметов, журнал заданий с целями, рейтинг игроков и так далее.

ListView предоставляет аккуратный прокручиваемый интерфейс, в котором удобно представлять такие сведения и управлять ими. В Unity 6 привязка данных во время выполнения упрощает процесс: при изменении данных не нужно вручную обновлять интерфейс или писать для этого специальные сценарии.

В предыдущих версиях Unity для заполнения ListView и обработки изменений приходилось создавать собственный код. В Unity 6 ListView можно связать с источником данных напрямую, и изменения будут автоматически отслеживаться и отображаться в интерфейсе.

В демонстрационной сцене простой ListView связан со списком ScriptableObject PlayerDataSO. Так можно создать интерфейс, похожий на лобби многопользовательской игры или таблицу рекордов.



TeamList связывает список с ListView.

С помощью привязки во время выполнения ListView можно напрямую связать с источником данных, например ScriptableObject. ListView автоматически отслеживает изменения, упрощая настройку и сопровождение.

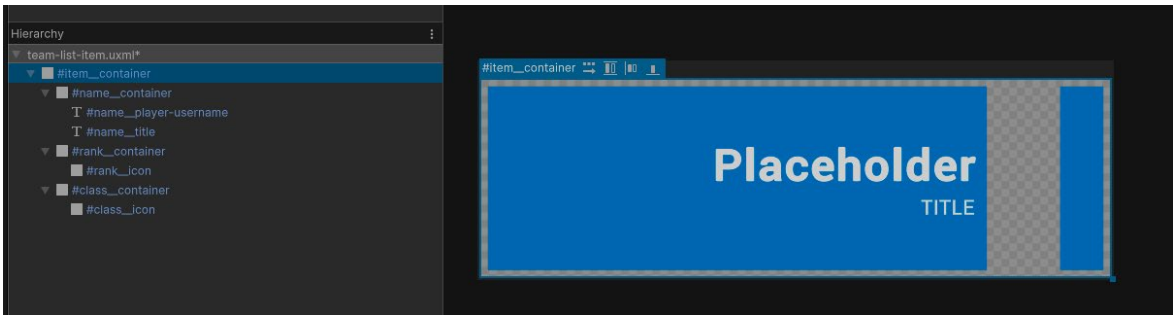
Чтобы привязать ListView к списку, сначала создайте несколько неразрешённых привязок, а затем завершите их настройку во время выполнения.

Настройка списка и шаблонов

Чтобы подготовить ListView к привязке данных, выполните следующие действия:

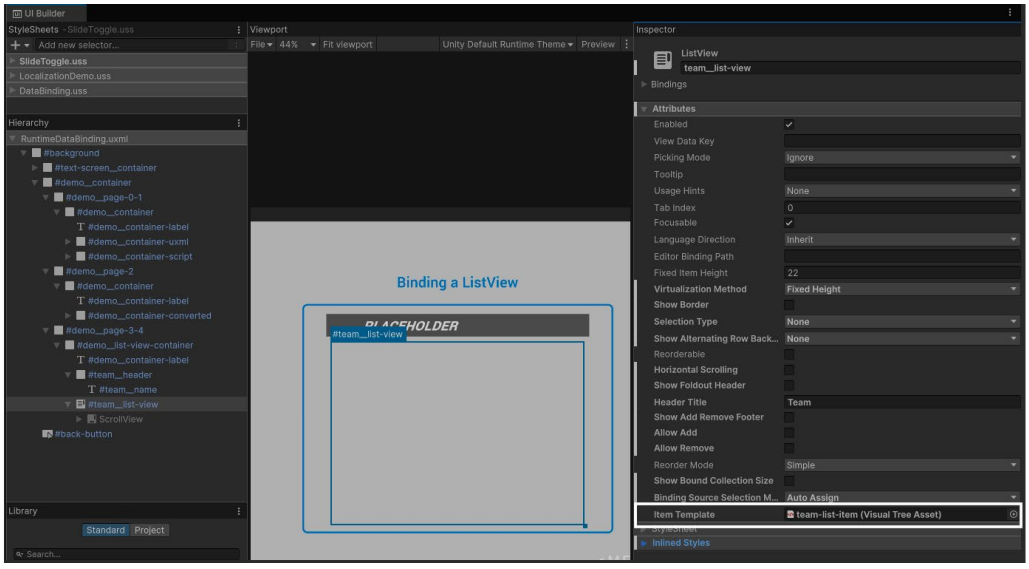
1. Определите источник данных. ListView необходим список данных. В этой демонстрации ScriptableObject TeamSO содержит список объектов PlayerDataSO. Каждый элемент списка соответствует одной строке ListView.

2. Создайте шаблон элемента UXML. В UI Builder разработайте шаблон UXML, то есть VisualTreeAsset, который определяет вид одного элемента списка. Например, шаблон team-list-item из демонстрации содержит имя игрока и несколько свойств Texture2D. Не указывайте источник данных напрямую: задайте в UI Builder параметры Data Source Type и Data Source Path. Привязка останется неразрешённой, и её можно будет завершить позднее во время выполнения.



Разработка ресурса визуального дерева в UI Builder.

3. Добавьте ListView в основной интерфейс. В другом файле UXML добавьте элемент ListView, который будет отображать весь список игроков. Назначьте созданный шаблон свойству Item Template списка. Теперь ListView знает, как должна выглядеть каждая строка, но конкретный источник данных ещё не определён.



Add VisualTreeAsset as Item Template

Добавление шаблона в ListView.

В ListView демонстрационной сцены используются лишь несколько базовых параметров, показанных выше. Расширенные возможности описаны в официальной документации ListView.

Завершение привязки во время выполнения

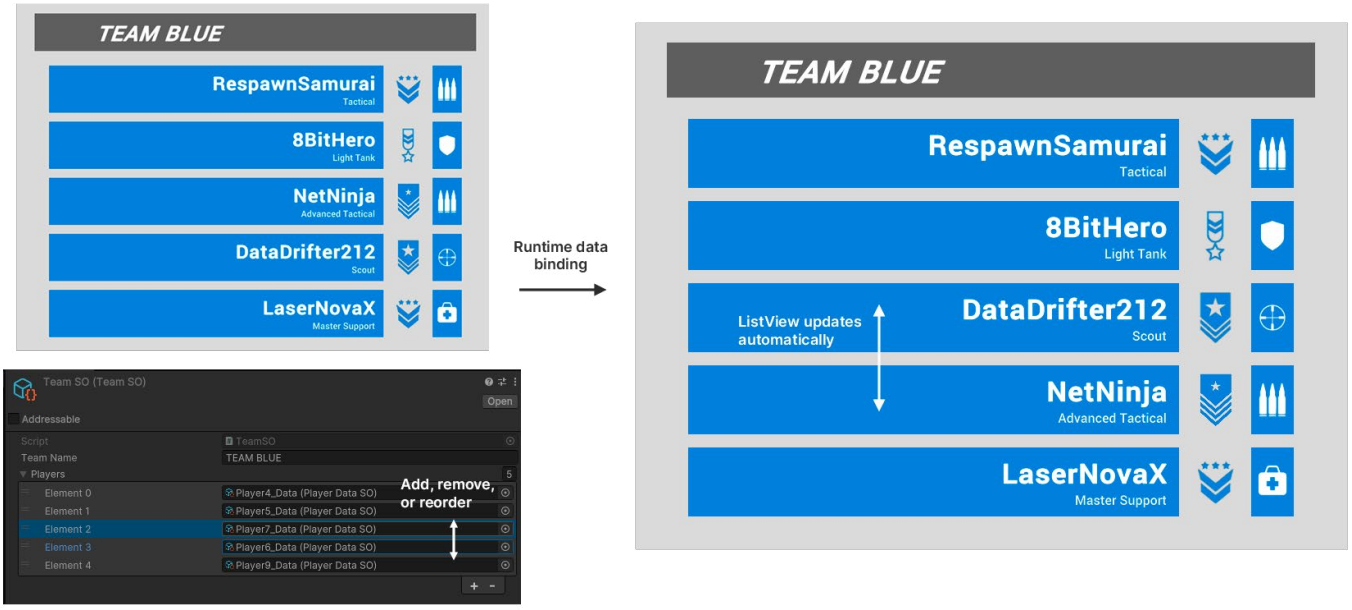
Во время выполнения простой сценарий TeamList завершает привязку, предоставляя фактический источник данных. Следующие строки разрешают ранее незавершённые привязки:

```
// Set the data source
m_ListView.dataSource = m_TeamData;

// Bind the "itemsSource" to the Players list
m_ListView.SetBinding("itemsSource", new DataBinding
{
    dataSourcePath = new PropertyPath("Players")
});
```

Здесь m_TeamData, экземпляр TeamSO, назначается списку ListView. Однократный вызов SetBinding связывает свойство Players с itemsSource, после чего ListView заполняет строки интерфейса.

Поскольку до начала выполнения эти привязки в UXML остаются неразрешёнными, подключать каждый элемент списка отдельно не требуется. UI Toolkit самостоятельно разрешает привязки и заполняет данными все элементы.

Любые изменения списка в источнике - добавление, удаление или перестановка игроков - сразу появляются в интерфейсе без дополнительного кода.



Интерфейс отражает изменения списка Player.

Помните: гибридный подход к привязке данных позволяет значительно сократить объём повторяющегося шаблонного кода. Задав в UXML пути-заполнители, можно отложить назначение фактического источника до времени выполнения.

Если модель данных изменится, переписывать всю логику привязки не придётся: достаточно одного обновления при запуске, чтобы переключить интерфейс на новый источник.

Полное руководство по привязке ListView к списку приведено на этой странице документации.

Оптимизация привязки данных

Эффективные привязки помогают поддерживать высокую производительность интерфейса. Избыточные и дублирующиеся привязки перегружают систему, вызывают лишние обновления и снижают производительность. Это особенно важно для сложных или ресурсоёмких интерфейсов.

По умолчанию система привязки во время выполнения обновляет элементы интерфейса каждый кадр. Для небольшого приложения это обеспечивает хорошую отзывчивость, но при большом числе привязок может стать узким местом.

В этом разделе рассматриваются способы повысить эффективность привязки данных в крупных проектах.

Работа с типами-значениями

Если источник данных использует типы-значения, например int, float или struct, учитывайте затраты на упаковку. Свойство dataSource имеет тип object, поэтому частое преобразование типов-значений создаёт дополнительные расходы.

Чтобы снизить их, избегайте лишних привязок и повторных обновлений свойств типов-значений.

Сокращение накладных расходов

Сначала найдите привязки, которые несколько раз обновляют одни и те же элементы или отслеживают редко меняющиеся данные. Объедините либо удалите их, чтобы сократить лишнюю работу. По возможности используйте плоские простые структуры данных вместо сложных иерархий - это помогает избежать задержек из-за частого поиска значений.

Ресурсоёмкие вычисления стоит выполнять заранее или кэшировать их результаты. Привязка к заранее рассчитанным значениям уменьшает вычислительную нагрузку и исключает повторные расчёты.

Часто обновляемые привязки оставляйте только у элементов, которым это действительно необходимо. Если постоянная синхронизация не нужна, удалите привязку и назначайте значение напрямую либо обновляйте его по событию.

Использование условий обновления

Привязки обновляются в соответствии с условиями, которые определяют частоту синхронизации интерфейса с источником данных. Это позволяет найти баланс между производительностью и отзывчивостью. Доступны следующие варианты:

- Каждый кадр: непрерывное обновление. Используйте для элементов, которым требуется постоянная синхронизация, например для шкал здоровья из примера.

- При обнаружении изменений: обновление выполняется, когда меняется источник данных, либо каждый кадр, если обнаружить изменение невозможно. Подходит, например, для панелей характеристик и списков инвентаря, основанных на наблюдаемых данных.

- При пометке как изменённой: если обновления происходят редко, явный вызов MarkDirty позволяет избежать лишних циклов обновления. Такой вариант подходит, например, для меню настроек, которые меняются лишь в определённых ситуациях.

Подбирая условие обновления под потребности каждого элемента интерфейса, можно сохранить и отзывчивость, и эффективность.

Версионирование и отслеживание изменений

Чтобы сократить число лишних обновлений, добавьте в источники данных версионирование и отслеживание изменений.

Повысить эффективность привязки помогают два интерфейса:

- IDataSourceViewHashProvider: отслеживает общие изменения с помощью хеша версии и запускает обновление только при изменении источника. Полезен для статических и полустатических данных, которые обновляются редко.

- INotifyBindablePropertyChanged: отслеживает изменения отдельных свойств и обновляет только затронутые привязки, обеспечивая более точное управление.

Добавьте эти интерфейсы в источник данных. Их можно использовать по отдельности или вместе для более точного управления обновлениями. Примеры и рекомендации приведены на этой странице документации.

Совет: дополнительные рекомендации по оптимизации UI Toolkit

В докладе Unite 2024 об оптимизации UI Toolkit рассматриваются цепочки вызовов отрисовки и влияние размеров буферов, рекомендации по динамическим атласам, а также работа с ограничениями, включая пользовательские шейдеры и трёхмерные интерфейсы.

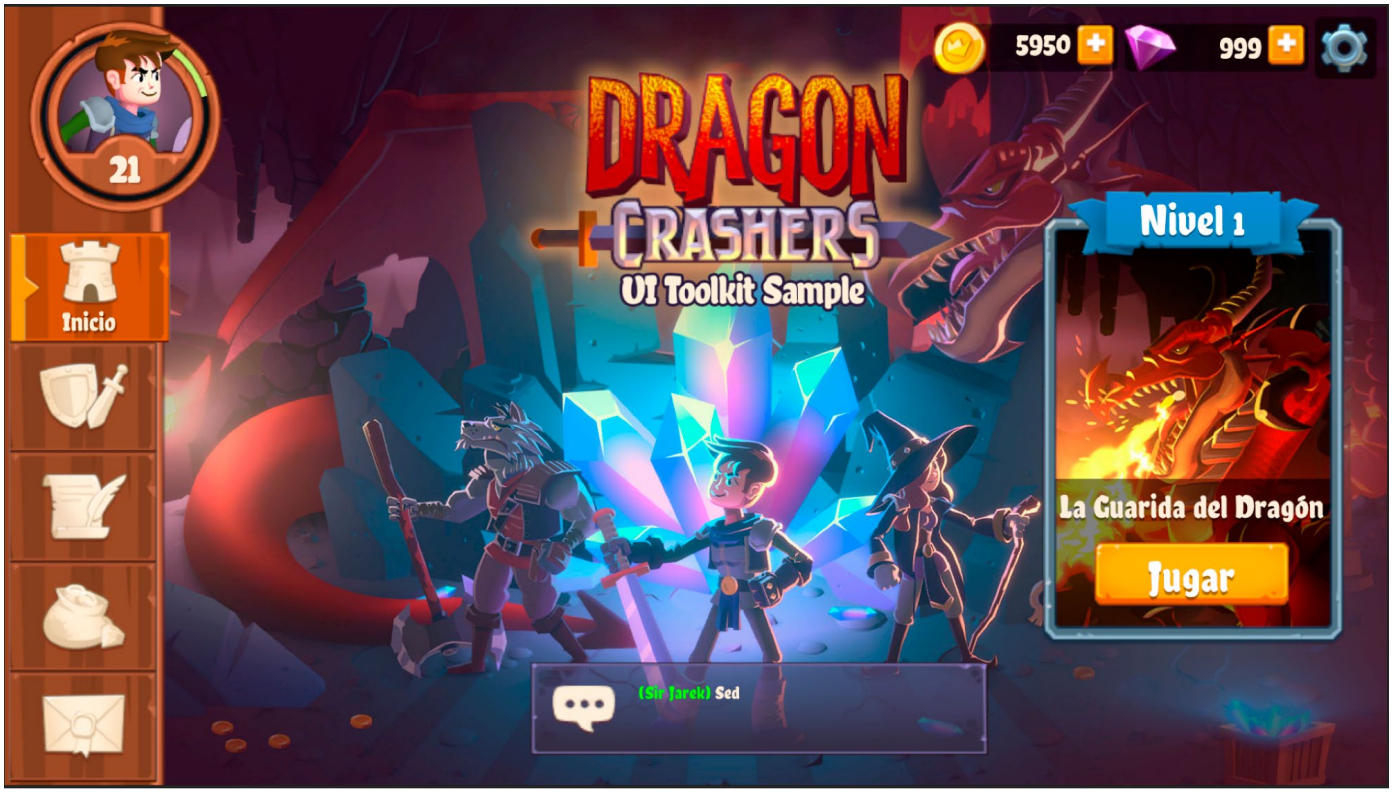
Локализация

Локализация интерфейса помогает игре выйти на мировую аудиторию и делает приложение понятным и привычным на любом языке.

Unity 6 упрощает процесс благодаря прямой интеграции пакета Localization с UI Toolkit. Она позволяет предлагать игрокам содержимое с учётом их региона, где бы они ни находились.

Ключевую роль в локализации Unity играет класс Locale: он представляет конкретный язык и управляет региональными параметрами, например форматами валют и чисел.

Рассмотрим простой пример настройки локализации в UI Toolkit. После такой настройки приложение сможет динамически менять содержимое в соответствии с выбранным Locale.



Пример локализации на испанский язык в UI Toolkit Sample - Dragon Crashers

Как это работает

Основные возможности пакета Localization:

- Локализация строк: класс LocalizedString позволяет управлять строками, которые автоматически обновляются при смене Locale во время выполнения. Smart Strings поддерживают заполнители, формы множественного числа и другие языковые особенности.

- Локализация ресурсов: текстуры и другие ресурсы можно заменять в зависимости от Locale, создавая региональные варианты не только текста, но и другого содержимого.

- Привязка данных: пакет Localization интегрирован с привязкой данных UI Toolkit во время выполнения и связывает элементы интерфейса с String Table и Asset Table. Изменение данных, Locale или состояния загрузки автоматически обновляет интерфейс.

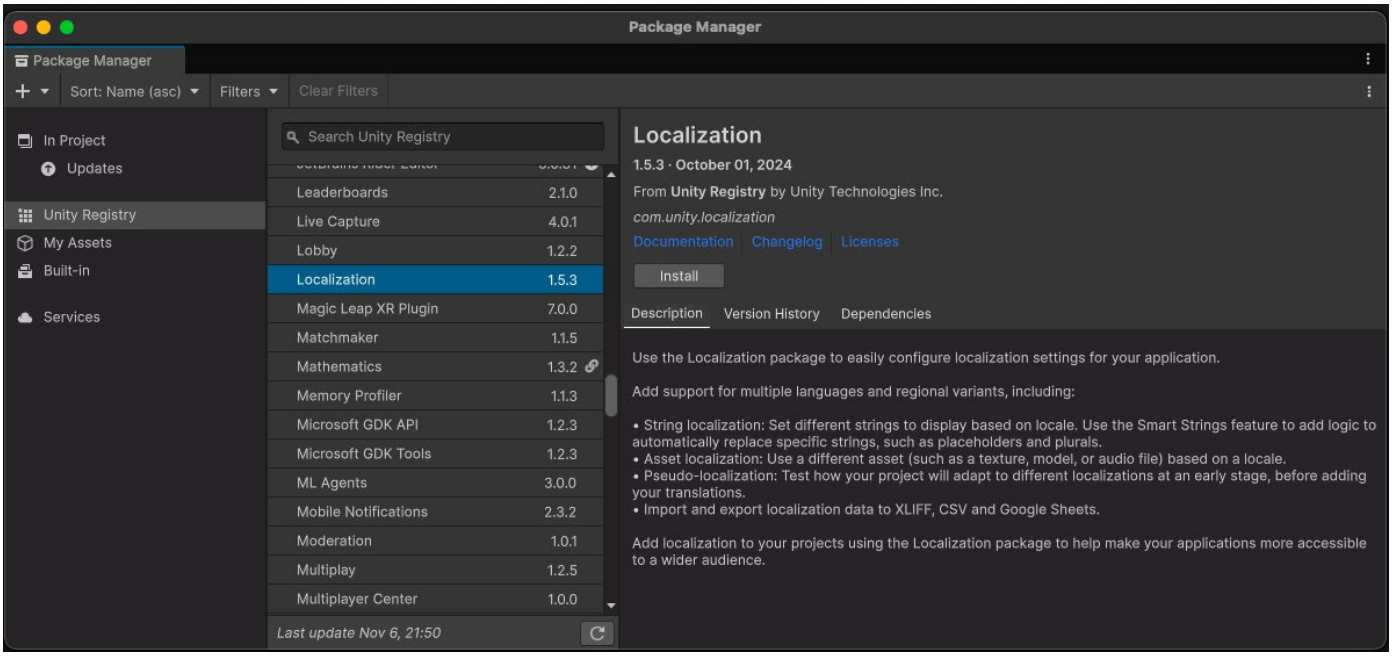
- Управление String Table и Asset Table: таблицы хранят пары «ключ - значение» для перевода текста или замены других ресурсов вариантами, соответствующими Locale. Централизованный интерфейс даёт общее представление обо всех локализованных строках и ресурсах проекта.

- Переключение Locale: язык можно менять в реальном времени без перезапуска приложения. Выберите новый Locale во время выполнения, и интерфейс сразу обновится.

При адаптации к разной длине и оформлению текста используйте FlexBox-контейнеры и элементы UI Toolkit с автоматическим размером. Это помогает создавать интерфейсы, которые лучше приспосабливаются к разным языкам.

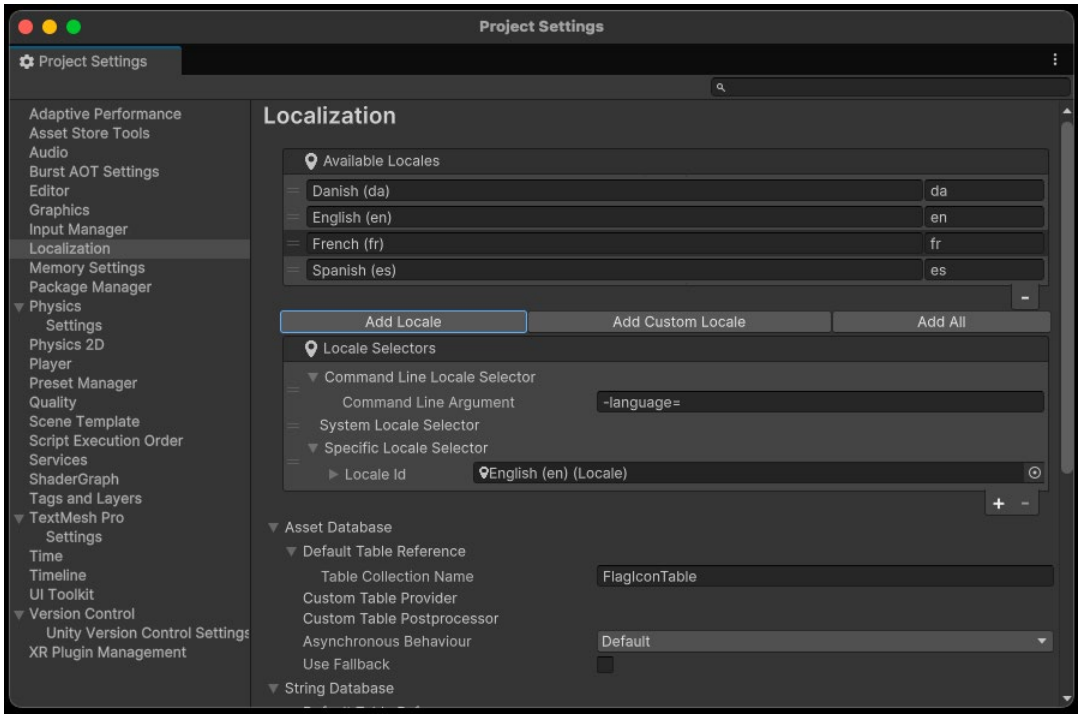
Настройка локализации

Чтобы начать использовать пакет Localization Unity с UI Toolkit, выполните следующие основные действия: подготовьте локализованное содержимое и свяжите его с элементами интерфейса в UI Builder.



Установите пакет Localization через Package Manager.

1. Настройте Localization Settings. Установите пакет Localization через Package Manager, затем откройте Project Settings > Localization и создайте ресурс Localization Settings. В нём настраиваются все локализованные ресурсы проекта.

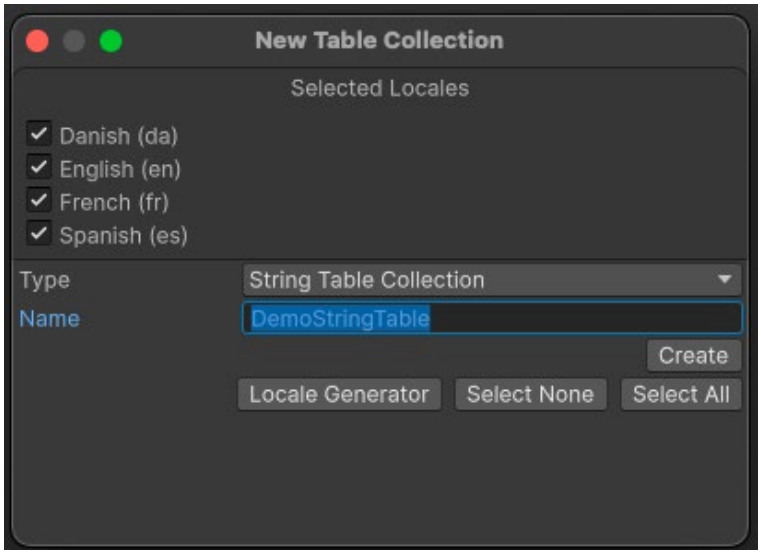


Создание и настройка Localization Settings в Project Settings.

2. Создайте Locale. С помощью Locale Generator определите поддерживаемые языки и регионы. Для каждого Locale будет создан ресурс с уникальным двухбуквенным кодом: например, en для английского, fr для французского и es для испанского. Укажите Locale по умолчанию, который будет использоваться при запуске приложения.

Добавление Locale.

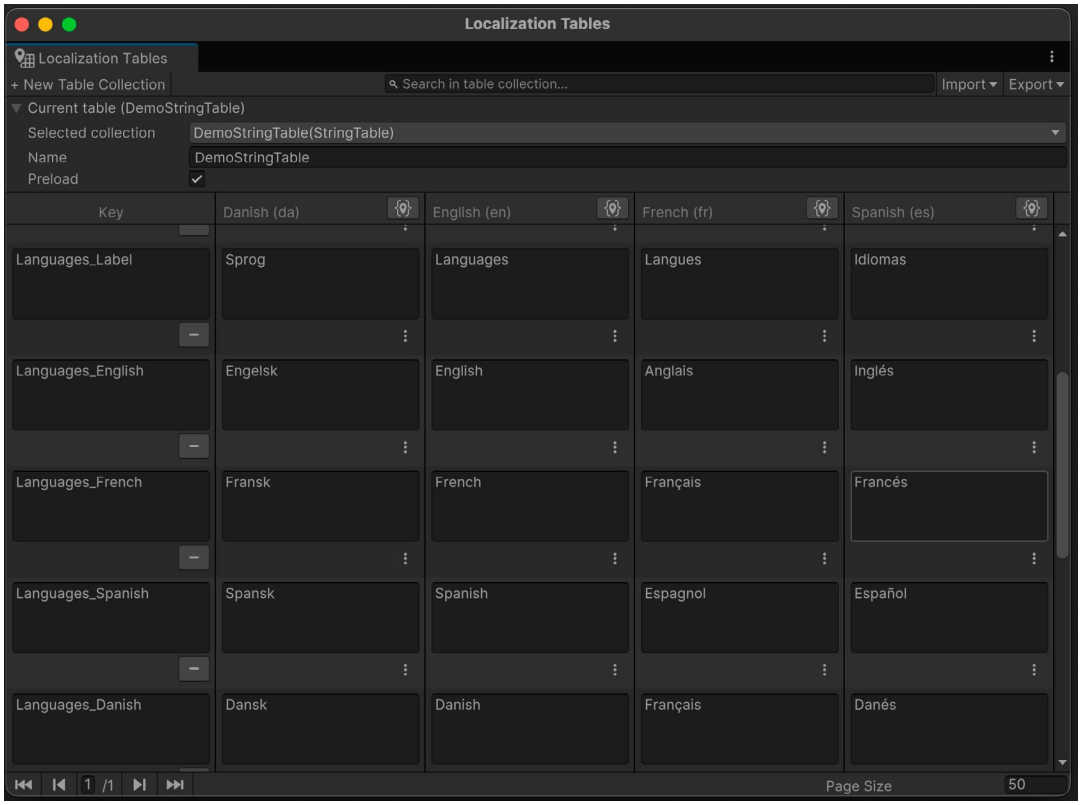
3. Создайте String Table и Asset Table. Используйте String Table для хранения текстовых записей каждого Locale. Добавьте строки элементов интерфейса: надписей, кнопок и вариантов раскрывающихся списков.



Создание String Table и Asset Table.

4. В окне Localization Tables (Window > Asset Management > Localization Tables) добавьте пары «ключ - значение» для каждого текстового элемента интерфейса. Ключ обозначает конкретный текстовый элемент, например метку или кнопку, а значение содержит перевод этого элемента для каждой локали.

5. Храните региональные ресурсы, например изображения или GameObject, в таблицах ресурсов (Asset Tables). В частности, туда можно добавить спрайты или текстуры значков для каждой локали. Свяжите каждый ключ с ресурсами, предназначенными для соответствующей локали и учитывающими региональные или культурные особенности.

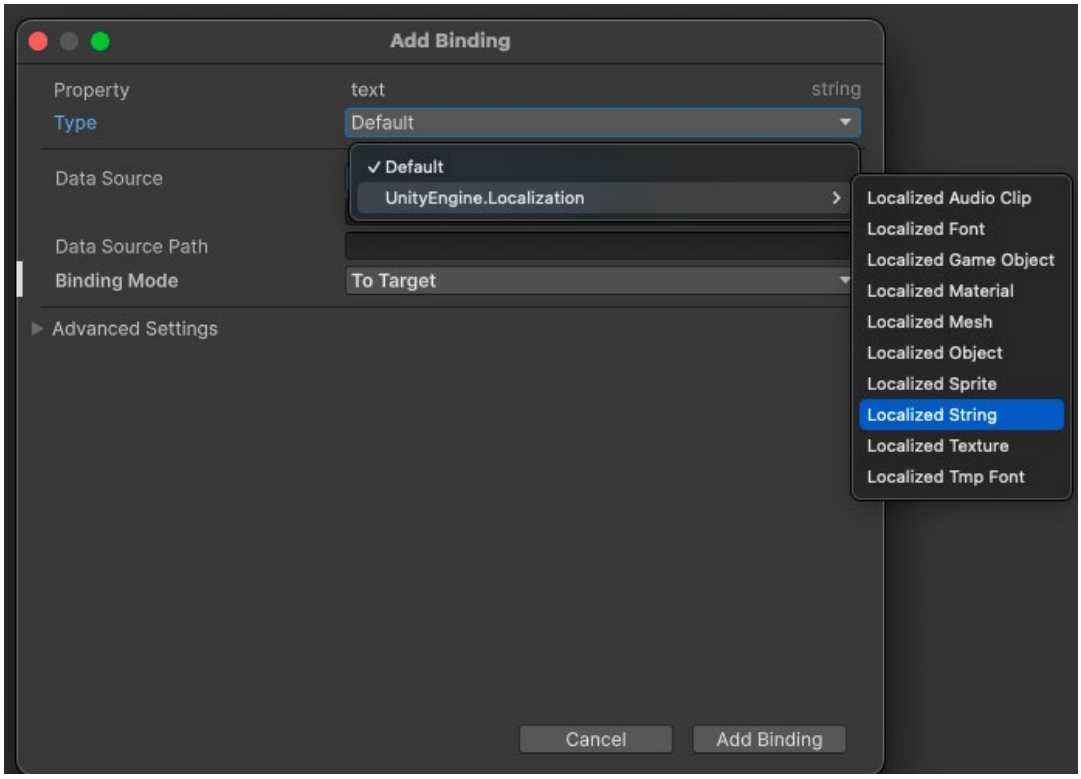


Каждый локализуемый элемент представлен парой «ключ - значение».

6. Создайте интерфейс в UXML. В UI Builder создайте UXML-файл с такими элементами, как Button, DropdownField и Label. В демонстрационной сцене этот UXML содержит несколько элементов, подготовленных к локализации. Для текстовых полей используйте запись из таблицы строк (String Table), а для нетекстовых, например текстур, - запись из таблицы ресурсов (Asset Table).

Демонстрационная сцена

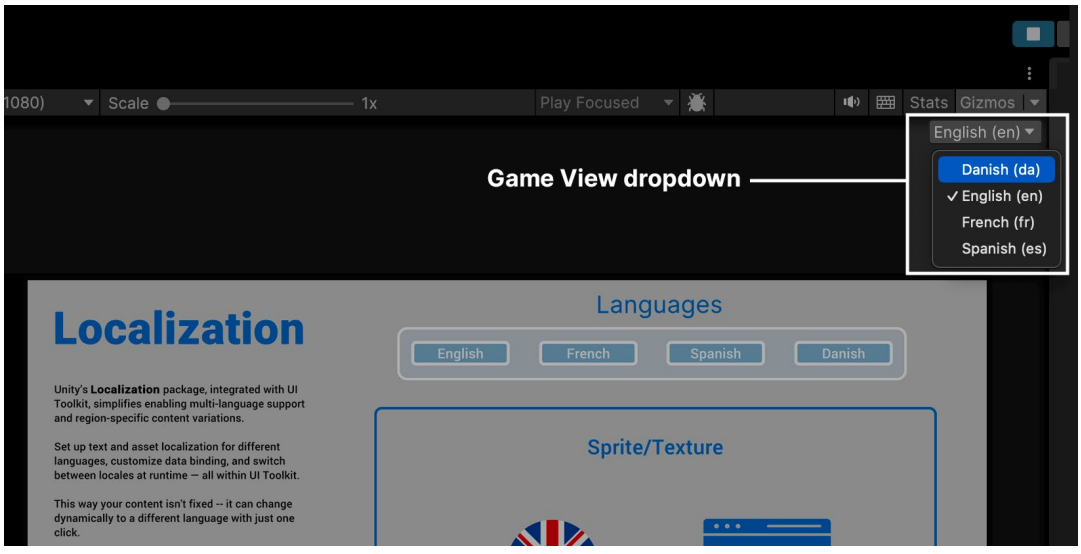
Пример реализации локализации находится в сцене LocalizationDemo проекта QuizU. Чтобы открыть его во время выполнения, перейдите в главное меню и выберите Demos > Localization. Также можно отключить загрузчик (Quiz > Don’t Load Bootstrap Scene on Play) и загрузить сцену LocalizationDemo напрямую.



Добавьте в UI Builder привязки данных к локализованным строкам и ресурсам.

7. Привяжите данные к элементам интерфейса в UI Builder. Здесь раскрываются возможности системы привязки данных UI Toolkit во время выполнения. Выберите в UI Builder элемент, который требуется локализовать. Откройте панель Inspector и в поле содержимого выберите Add Binding: например, для Label это поле text, а для изображения - backgroundImage.

> В качестве типа привязки выберите LocalizedString или другой локализованный ресурс и укажите соответствующую запись в таблице строк или ресурсов. По мере локализации новых элементов добавляйте записи в эти таблицы.



Для предварительного просмотра локализации используйте раскрывающийся список локалей в окне Game View.

8. Для проверки выберите нужные языки в раскрывающемся списке локалей окна Game View и убедитесь, что элементы интерфейса правильно отображаются в каждой локали.

Базовая настройка завершена! Теперь в этом примере текстовые свойства кнопок и меток можно переключать на любой настроенный язык. Чтобы локализовать весь интерфейс, создайте в таблице строк отдельную запись для каждого фрагмента текста.

Пакет Localization позволяет гибко организовать содержимое: крупный проект можно разделить на удобные разделы с помощью нескольких таблиц строк или сгруппировать записи по категориям. Для локализации текстур и других нетекстовых ресурсов используйте таблицы ресурсов.

После добавления привязок локализации в UI Builder они сохраняются непосредственно в элементах UXML-файла. В результате получается такой блок кода, где каждое локализованное свойство связано с конкретной записью в таблице строк или ресурсов:

```
<Bindings>
   <UnityEngine.Localization.LocalizedString property="text"
table="GUID:6aaa262cde38a4024bc3fc7f5ce6d50d"
entry="Id(104135776002048)" />
</Bindings>
```

Этот фрагмент UXML создаёт привязку данных, которая связывает свойство text элемента интерфейса с записью в таблице строк или ресурсов.

При каждом обновлении таблиц локализации связанные элементы интерфейса автоматически получают актуальное локализованное содержимое.

Примечание. UI Builder упрощает создание привязок данных, однако опытные пользователи могут редактировать UXML напрямую, чтобы точнее управлять локализованным содержимым.

Использование API локализации

Раскрывающийся список локалей в окне Game View удобен для проверки разных языков в редакторе, но в сборке приложения его не будет. Чтобы пользователи могли менять язык в готовом приложении, создайте собственный интерфейс переключения локалей.

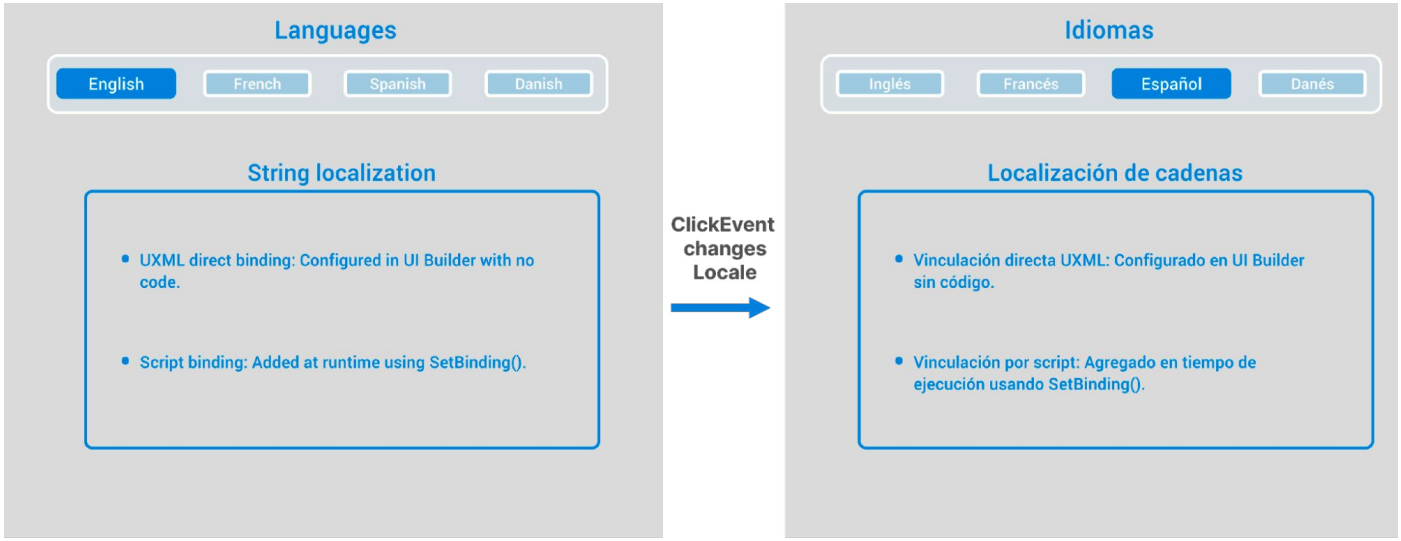
Выбор локали

Если известен двухбуквенный идентификатор локали, активную локаль можно задать через LocalizationSettings. Затем свяжите это действие с кнопками: используйте манипулятор clicked каждой кнопки или метод RegisterCallback<ClickEvent>.

Один из вариантов реализации показан в скрипте LocalizationDemo из примера проекта:

```
void SelectLocale(string localeCode)
{
    Locale locale = LocalizationSettings.AvailableLocales.GetLocale(localeCode);
    LocalizationSettings.SelectedLocale = locale;
}


void RegisterCallbacks()
{
    m_ButtonDanish.clicked += () => SelectLocale("da");
    m_ButtonEnglish.clicked += () => SelectLocale("en");
    m_ButtonSpanish.clicked += () => SelectLocale("es");
    m_ButtonFrench.clicked += () => SelectLocale("fr");
}
```

Теперь каждая кнопка может переключать интерфейс на указанную для неё локаль. При нажатии кнопки English, French, Spanish или Danish текст интерфейса меняется во время выполнения.



Кнопки позволяют переключать локали.

Использование SetBinding

Настраивать привязки данных в UI Builder удобно и наглядно. Однако иногда привязку требуется создать скриптом во время выполнения - например, если элементы интерфейса создаются динамически или данные для привязки становятся доступны только во время игры.

Чтобы настроить привязку данных в C#, вызовите метод SetBinding у визуального элемента. Ниже показано, как связать свойство text элемента Label с записью LocalizedString в StringTable:

```
using UnityEngine;
using UnityEngine.Localization;
using UnityEngine.UIElements;


public class LocalizationDemo : MonoBehaviour
{
    // Set in Inspector
    [SerializeField] LocalizedString m_LocalizedText;

    Label m_LocalizedLabel;
    UIDocument m_UIDocument;

    void Start()
    {
        m_LocalizedLabel = m_UIDocument.rootVisualElement.Q<Label>("text__label");
        m_LocalizedLabel.SetBinding("text", m_LocalizedText);
    }
}
```

В этой конфигурации полю m_LocalizedText через Inspector назначается запись из DemoStringTable. Код связывает свойство text элемента m_LocalizedLabel с указанным объектом LocalizedString, поэтому при смене локали текст обновляется автоматически.

Отслеживание смены локали

Иногда при смене локали нужно не только обновить локализованные строки, но и выполнить дополнительные действия. Если требуется запускать определённую логику при каждом изменении локали, подпишитесь на событие SelectedLocaleChanged API LocalizationSettings.

Пример:

```
using UnityEngine;
using UnityEngine.Localization;
using UnityEngine.Localization.Settings;

public class LocalizationExample: MonoBehaviour
{
   void OnEnable()
   {
       LocalizationSettings.SelectedLocaleChanged += OnLocaleChanged;
   }

   void OnDisable()
   {
       LocalizationSettings.SelectedLocaleChanged -= OnLocaleChanged;
   }

   void OnLocaleChanged(Locale newLocale)
   {
        // Perform actions when the Locale changes, like updating UI elements
        Debug.Log($"Locale changed to: {newLocale.Identifier.Code}");
    }
}
```

В данном случае метод OnLocaleChanged вызывается при каждой смене локали, позволяя обновить другие элементы или выполнить пользовательскую логику. С помощью этого обработчика можно корректировать свойства и стили интерфейса, особенно если переведённый текст не помещается в текущую компоновку.

Работа с таблицами строк

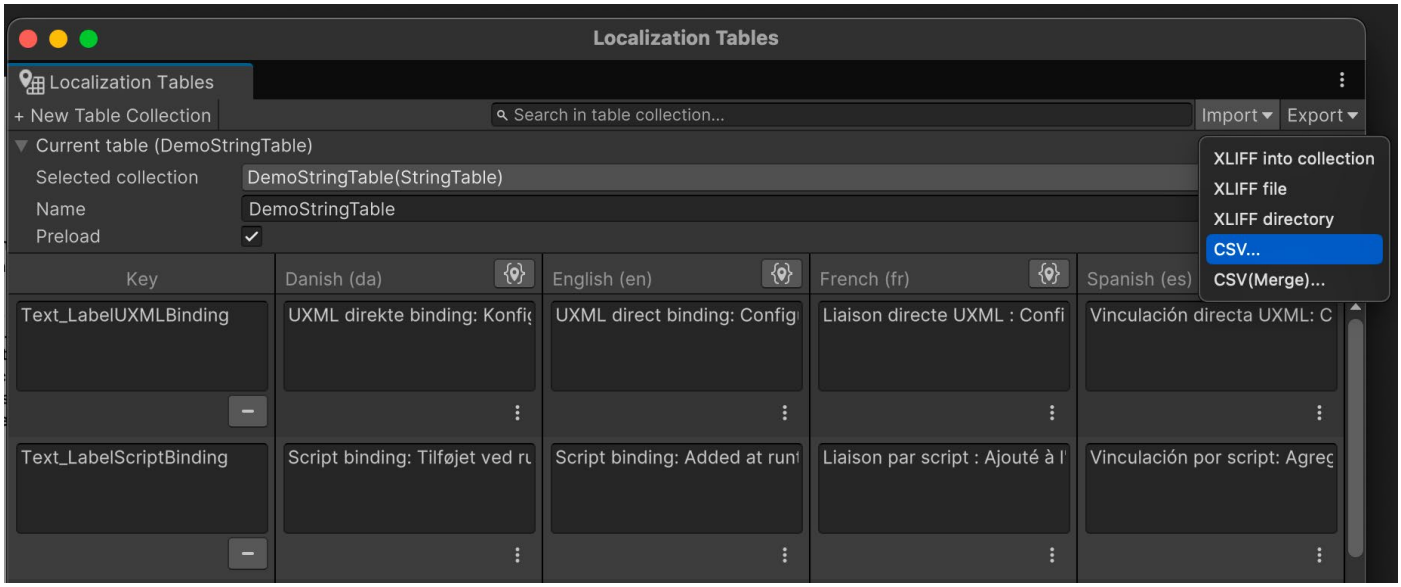
Основная часть локализации выполняется с помощью таблиц строк (String Tables), в которых хранятся все текстовые переводы для интерфейса и меток.

Откройте окно Localization Tables (Window > Asset Management > Localization Tables) и создайте или выберите коллекцию таблиц строк (String Table Collection). Здесь можно добавлять записи, задавать для них уникальные ключи и вводить перевод для каждой локали.

Импорт и экспорт строковых данных

CSV-файлы

Таблицу строк можно заполнить, импортировав данные из CSV-файла (значения, разделённые запятыми), поэтому дизайнеры смогут подготовить текст во внешнем редакторе. Чтобы редактировать существующие записи в обычном текстовом формате, экспортируйте таблицу строк в CSV.



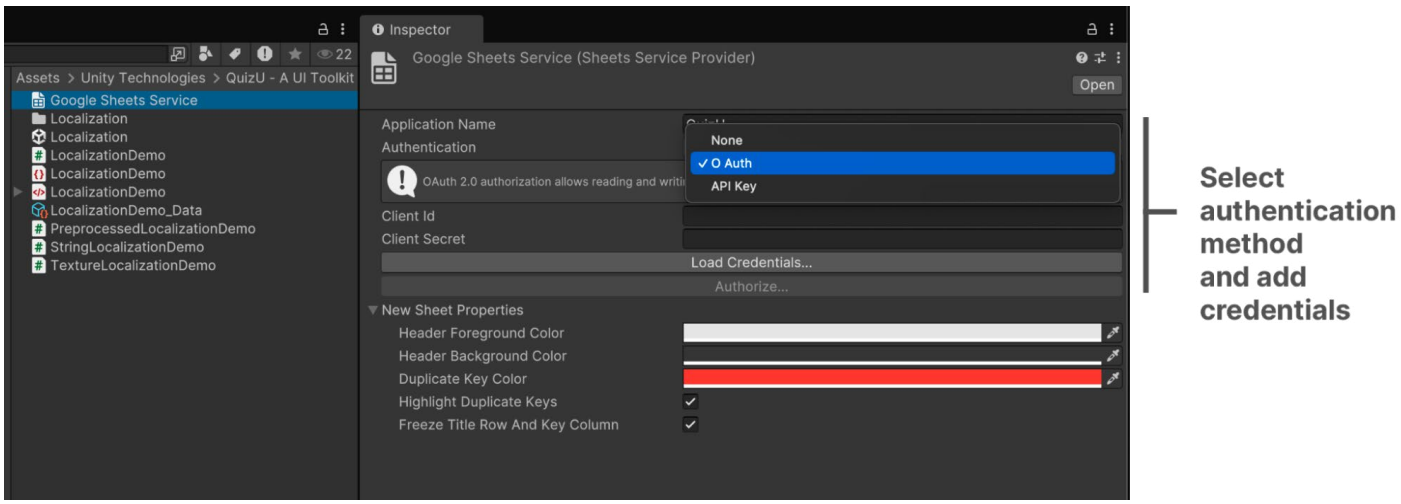
Импортируйте или экспортируйте CSV-файлы.

После редактирования импортируйте файл обратно в Unity: записи автоматически обновятся по своим ключам.

Синхронизация с Google Таблицами

Чтобы подключить проект к Google Таблицам, потребуется ресурс Sheets Service Provider. Он управляет аутентификацией и позволяет создавать таблицы непосредственно в редакторе.

Для его создания выберите Assets > Create > Localization > Google Sheets Service.

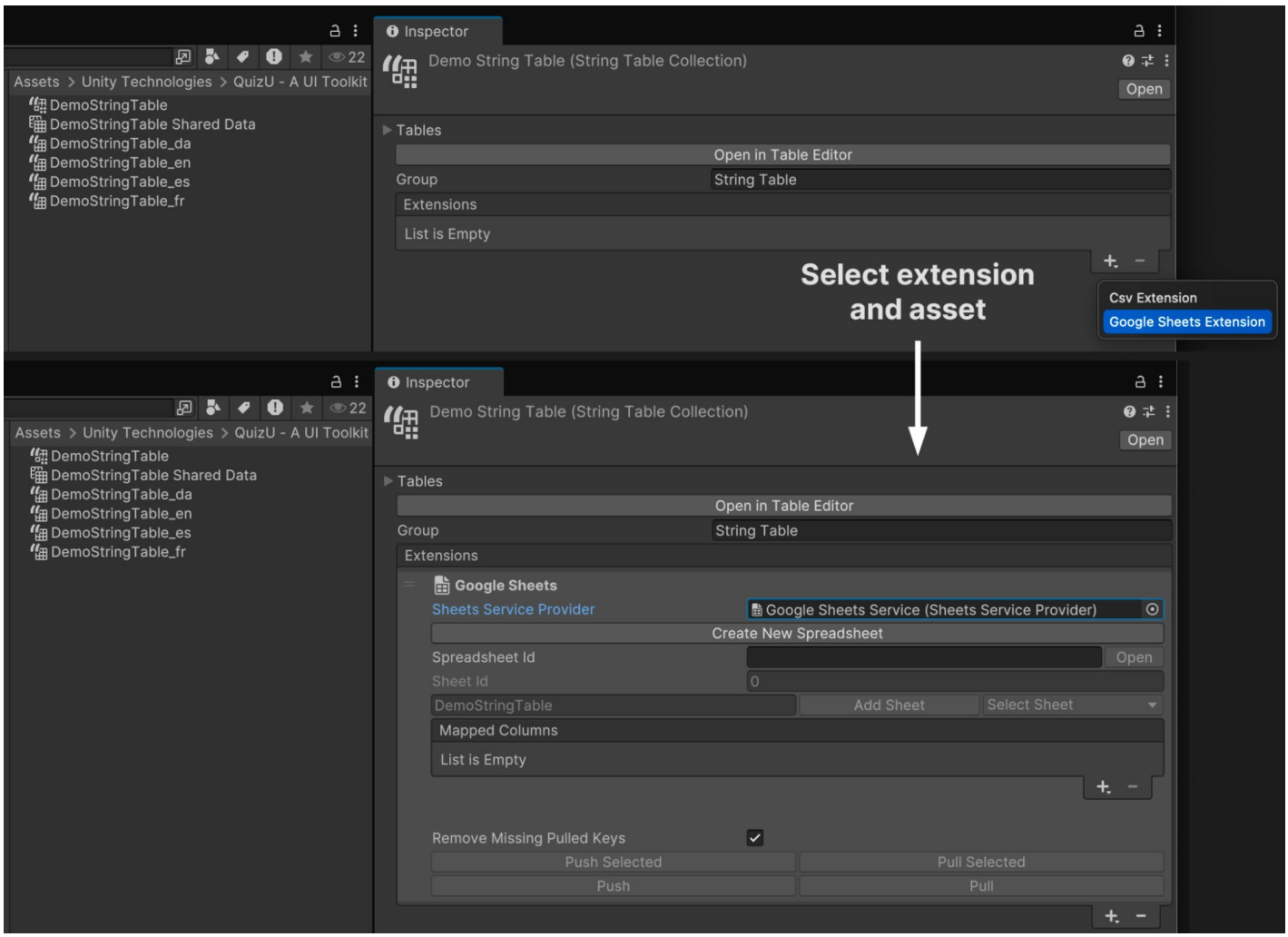


Создайте сервис Google Таблиц и выполните авторизацию.

Сервис Google Таблиц поддерживает два способа авторизации: OAuth и API Key. Используйте OAuth, если нужен доступ к закрытым таблицам для чтения и записи. Ключ API подходит, если требуется только читать общедоступные таблицы. Для полного доступа на чтение и запись необходимо получить разрешение Google. Подробнее см. в документации Google Таблиц в разделе Authorizing Requests.

Чтобы связать коллекцию таблиц строк с Google Таблицей, добавьте расширение Google Sheet в список Extensions этой коллекции. Выберите ресурс String Table, затем в Inspector нажмите кнопку Add (+) рядом с разделом Extensions. В одну коллекцию таблиц строк можно добавить несколько расширений и при необходимости назначить каждой локали отдельную таблицу.

Для синхронизации таблицы строк с Google Таблицей подключите её к ресурсу Sheets Service Provider. Инструкции по созданию и настройке ресурса см. в разделе Sheets Service Provider.



Добавьте сервис Google Таблиц в список расширений StringTable.

После настройки синхронизации дизайнеры и другие участники команды, не занимающиеся разработкой, смогут редактировать записи локализации непосредственно в Google Таблицах.

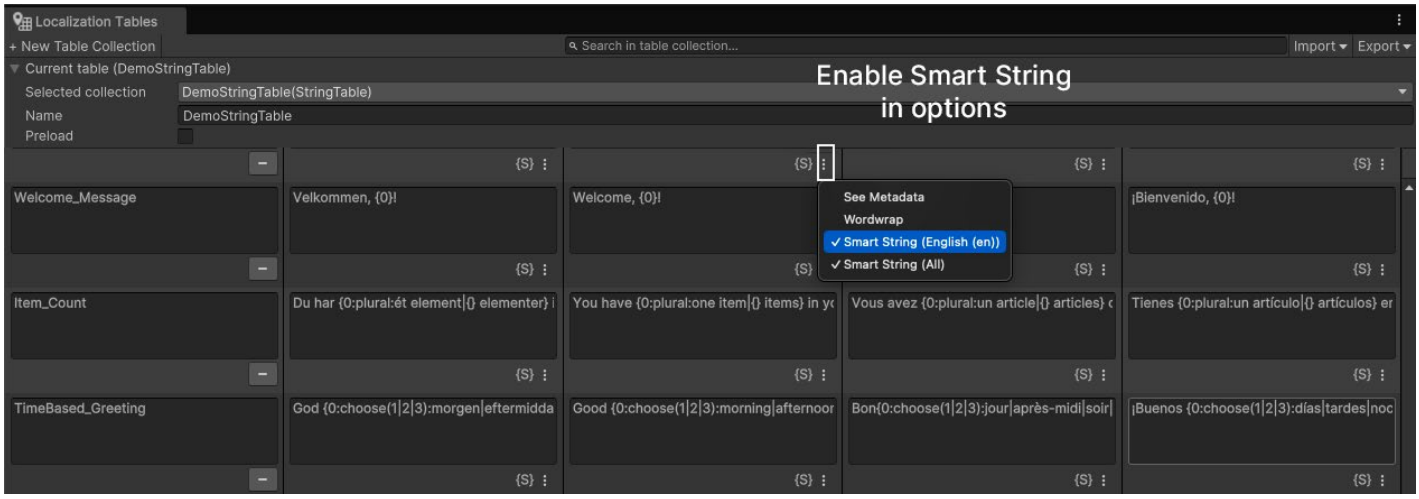
Использование Smart Strings

Smart Strings - мощная альтернатива String.Format для создания динамических строк. Они позволяют использовать шаблоны на основе данных с поддержкой форм множественного числа, условного форматирования, списков и других языковых правил. Благодаря этому настройка локализации становится проще.

Чтобы использовать Smart String, пометьте строку как интеллектуальную в окне Localization Tables.

Откройте это окно, затем в меню (⋮) выберите Smart Format и убедитесь, что рядом с записью появился значок {S}.

Другой способ - включить параметр Smart в редакторе Localized String в Inspector.



Включить Smart Strings можно в окне StringTable или в Inspector.

Настройка Smart String в скрипте

Чтобы управлять SmartString из скрипта:

- Настройте локализованную строку с заполнителями. Smart String состоит из обычного текста и заполнителей в фигурных скобках {}, как в String.Format, но обладает более гибкими возможностями. Создайте в таблице строк запись с заполнителем, например "Welcome, {0}!". Здесь {0} будет заменён данными во время выполнения.

- Используйте LocalizedString и аргументы. Создайте в скрипте объект LocalizedString для этой записи и передайте данные времени выполнения через свойство Arguments. Например, следующий фрагмент из SmartStringDemo показывает, как заменить один заполнитель:

```
//  Replaces placeholder with player name (e.g.,
//       "Welcome, {0}!" => "Welcome, Player One!")

m_PlaceholderLabel = root.Q<Label>("welcome__label");

m_PlaceholderMessage.Arguments = new object[] { m_PlayerName };

m_PlaceholderLabel.SetBinding("text", m_PlaceholderMessage);
```

Этот код связывает LocalizedString со свойством text метки и во время выполнения подставляет имя игрока. Исходная запись "Welcome, {0}!" может отобразиться на экране как "Welcome, Player One!".

Принцип работы заполнителей

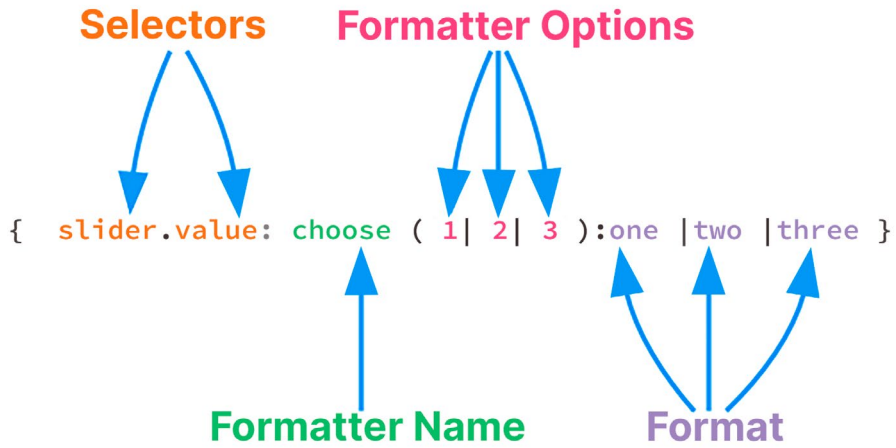
Заполнители Smart Strings не ограничиваются простой подстановкой {0}. Они могут иметь более сложную структуру и предназначены для расширенных сценариев. Заполнитель может состоять из нескольких частей, включая:

- Селектор: определяет, какие данные использовать. Например, {player.name} выбирает свойство name объекта player.

- Имя форматтера: задаёт применяемый форматтер, например plural для выбора формы числа.

- Параметры форматтера: настраивают его поведение, например задают формы единственного и множественного числа.

- Формат: определяет представление результата, например преобразование числа в слово нужной формы, форматирование даты или времени либо выбор фразы в зависимости от входных данных.



Заполнитель может состоять из нескольких частей.

Селекторы отличаются гибкостью и позволяют динамически получать данные во время выполнения. Они могут обращаться к свойствам и полям объектов. Например, селектор {gameObject.name} возвращает свойство name объекта GameObject, а {slider.value} - свойство value ползунка.

Форматтеры преобразуют полученные данные в итоговую строку. С их помощью можно форматировать даты, время и списки, выбирать формы множественного числа и даже применять условную логику.

После получения данных форматтер преобразует их в итоговый результат. У каждого форматтера есть собственные параметры и правила форматирования. В примере проекта используются два форматтера:

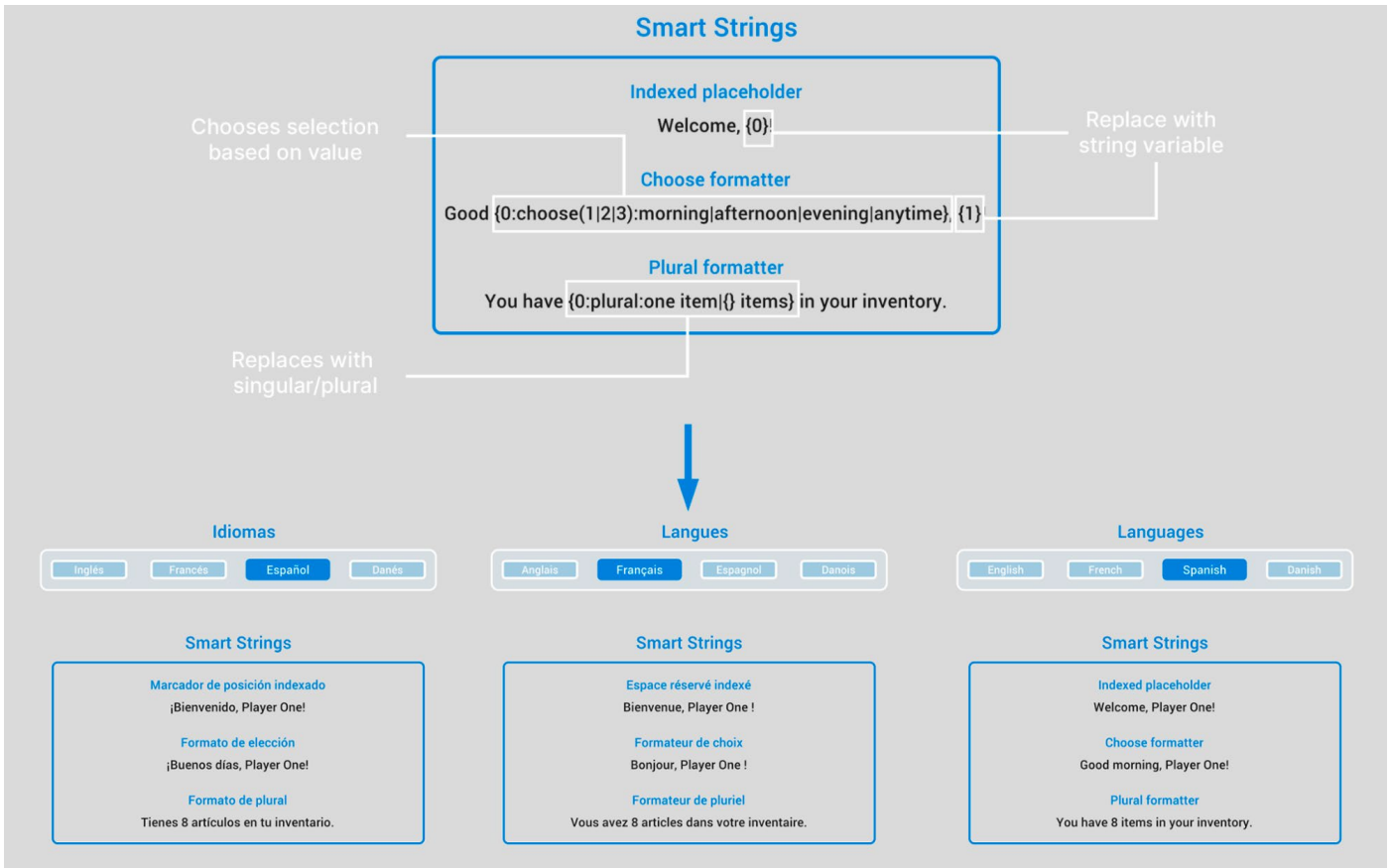
- Форматтер Choose:

```
{0:choose(1|2|3):morning|afternoon|evening|anytime} выбирает значение "morning", "afternoon" или "evening" в зависимости от входных данных.
```

- Форматтер Plural:

{0:plural:one item|{} items} выбирает форму единственного или множественного числа.

Измените значения в Inspector и перейдите в режим Play, чтобы увидеть полученный текст.



Smart Strings - альтернатива String.Format.

Smart Strings предоставляют ряд встроенных форматтеров, которые расширяют возможности локализации и позволяют адаптировать текст к состоянию игры или контексту:

- Форматтер Choose: применяет условную логику на основе числового значения.

- Форматтер Plural: автоматически выбирает форму числа в зависимости от количества.

- Форматтер Time: отображает дату и время.

- Форматтер Conditional: реализует логику, подобную if/else.

- Форматтер List: форматирует массивы и списки.

- Форматтер Is Match: выбирает отображаемый текст по шаблону регулярного выражения.

С помощью API также можно создавать собственные форматтеры. Дополнительные сведения см. в документации по Smart Strings.

Предварительная обработка строк

В некоторых случаях прямая привязка элементов интерфейса к LocalizedString неудобна. Например, перед отображением локализованного текста может потребоваться дополнительное форматирование или преобразование. В таких ситуациях LocalizedString можно предварительно обработать до вывода в интерфейсе.

GetLocalizedString

Для этого подходит метод GetLocalizedString: во время выполнения он преобразует LocalizedString в обычную строку. До передачи обработанной строки в интерфейс к ней можно применить собственное форматирование - например, добавить префикс или объединить несколько строк. Пример:

```
[SerializeField] int m_PlayerLevel = 1;
LocalizedString m_LevelMessage = new LocalizedString("My_Table", "My_Entry");


// A property that retrieves the localized string and replaces the placeholder {0} with the player's level
[CreateProperty] public string LevelMessage => m_LevelMessage.GetLocalizedString(m_PlayerLevel);
```

В этом примере свойство LevelMessage заменяет заполнитель {0} в LocalizedString текущим уровнем игрока. Атрибут [CreateProperty] позволяет использовать это свойство в системе привязки данных во время выполнения и напрямую связывать его с элементами интерфейса.

В простых случаях подобные свойства, как показанное выше LevelMessage, могут инкапсулировать логику форматирования без дополнительных обработчиков событий.

Использование события StringChanged

Событие StringChanged объекта LocalizedString удобно использовать для такой предварительной обработки. Оно срабатывает при каждом обновлении LocalizedString, например при смене локали, и позволяет изменить текст перед отрисовкой.

Для этого подпишите обработчик на StringChanged. Следующий фрагмент создаёт объект LocalizedString из записи My_Entry таблицы My_Table:

```
LocalizedString localizedString = new LocalizedString
   ("My_Table", "My_Entry");

localizedString.StringChanged += OnLocalizedStringChanged;
```

Обратите внимание: обработчик OnLocalizedStringChanged автоматически получает результат как обычную строку. Затем перед отображением текста к нему можно применить собственную логику:

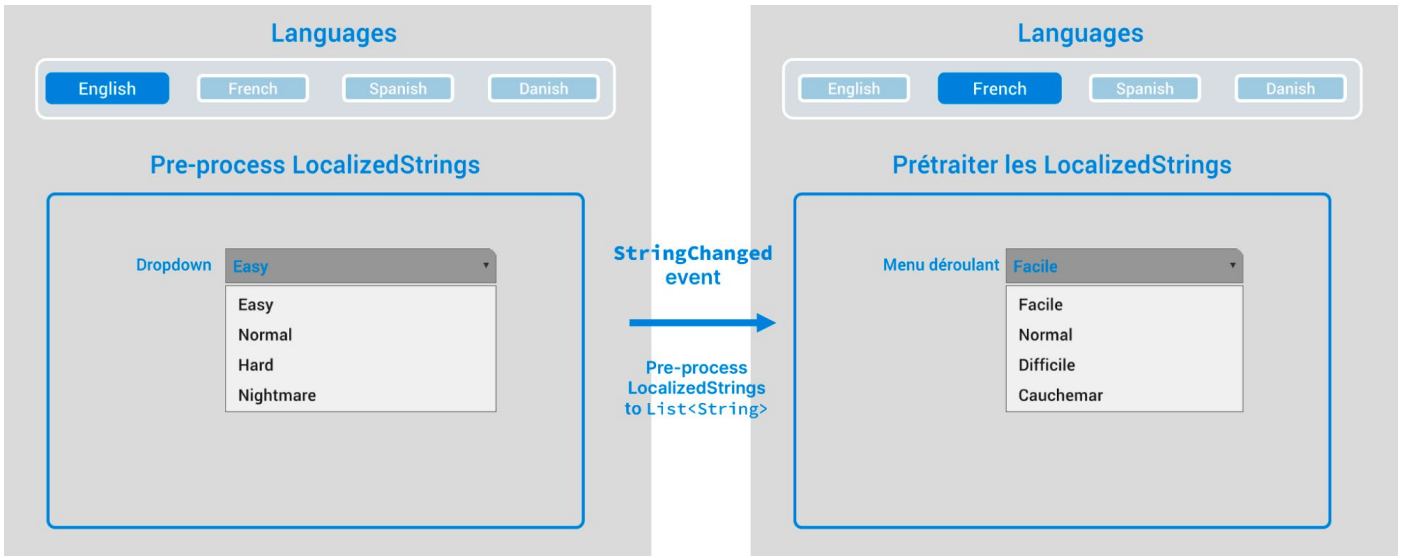
```
void OnLocalizedStringChanged(string value)
{
   // Example: Add a prefix based on certain conditions
   string processedString = $"[Prefix] {value}";

   // Update the UI element with the processed string
   m_TextLabel.text = processedString;
}
```

Динамические элементы управления интерфейса

Разумеется, предварительная обработка LocalizedString не ограничивается простыми текстовыми полями. Она особенно полезна для сложных свойств и структур интерфейса, когда одной Smart String недостаточно для требуемой логики или форматирования.

Например, свойство choices элемента DropdownField содержит список строк. Предварительная обработка позволяет динамически локализовать этот список вариантов в соответствии с активной локалью.

В данном случае скрипт PreprocessDemo локализует варианты DropdownField и обновляет их каждый раз, когда игрок выбирает другой язык. Логика перестроения списка снова запускается в ответ на событие StringChanged.



Предварительная обработка LocalizedString с помощью события StringChanged.

Ниже приведён фрагмент скрипта PreprocessDemo, демонстрирующий этот механизм:

```
[SerializeField] LocalizedString m_Choice1LocalizedString;
[SerializeField] LocalizedString m_Choice2LocalizedString;
[SerializeField] LocalizedString m_Choice3LocalizedString;
[SerializeField] LocalizedString m_Choice4LocalizedString;

public void Initialize(VisualElement root)
{

   m_DropdownField = root.Q<DropdownField>("dropdown__field");

   m_Choice1LocalizedString.StringChanged += UpdateDropdownChoices;
   m_Choice2LocalizedString.StringChanged += UpdateDropdownChoices;

   // … register other choices

   // Initial population
   UpdateDropdownChoices(null);
}
```

При срабатывании события StringChanged список вариантов раскрывающегося поля перестраивается, а текущее выбранное значение сохраняется:

```
void UpdateDropdownChoices(string value)
{
    if (m_DropdownField == null)
        return;

    // Save the current selection
    int selection = m_DropdownField.index;

    // Remove previous choices
    m_DropdownField.choices.Clear();

    // Add current localized values

    m_DropdownField.choices.Add(m_Choice1LocalizedString.GetLocalizedString());

    m_DropdownField.choices.Add(m_Choice2LocalizedString.GetLocalizedString());

    // … Add other choices

    // Restore selected index and value
    m_DropdownField.index = selection;

    m_DropdownField.SetValueWithoutNotify(m_DropdownField.choices[selection]);
}
```

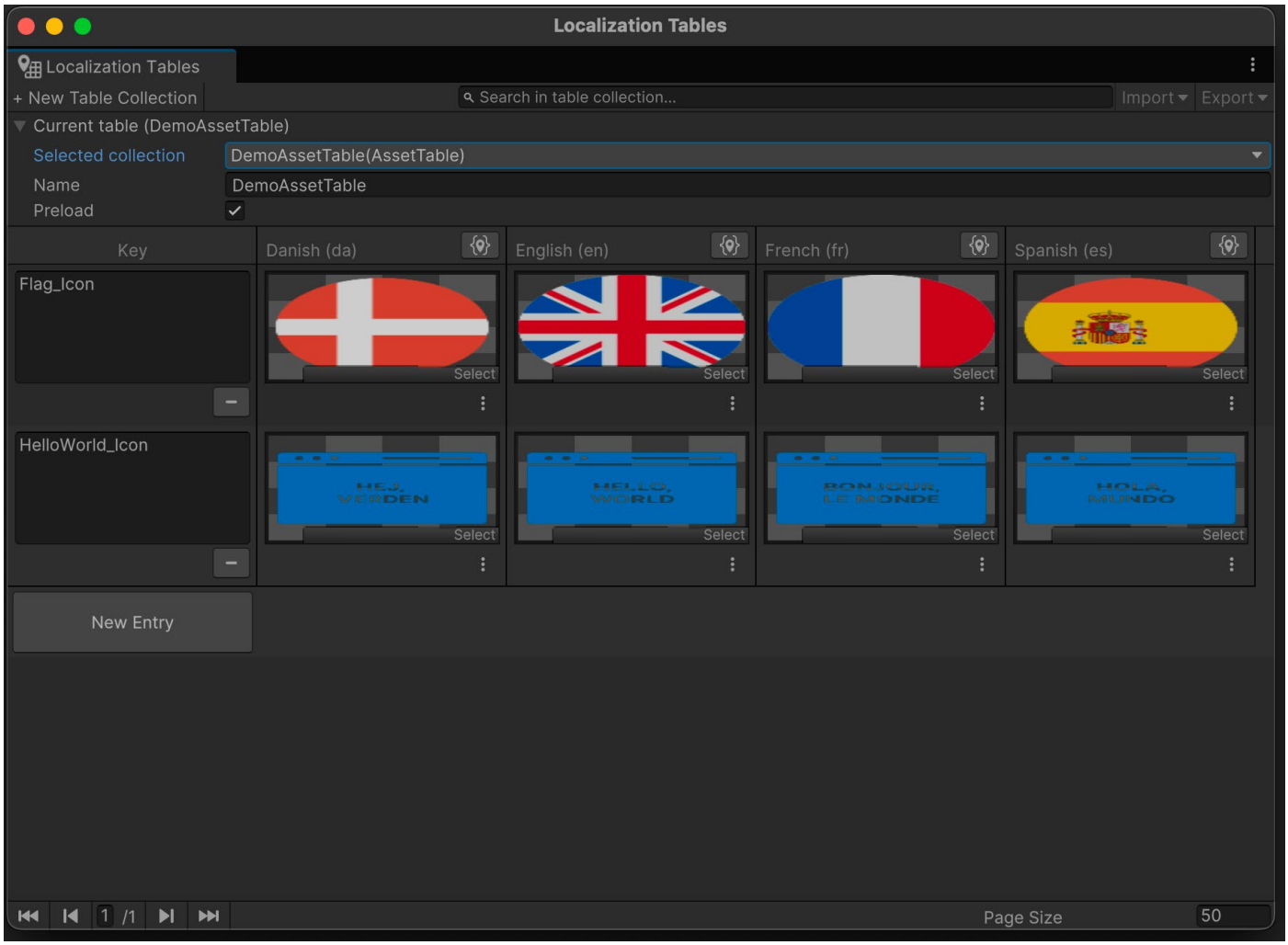
Метод SetValueWithoutNotify обновляет отображаемое значение раскрывающегося поля, не вызывая ChangeEvent. Это предотвращает рекурсивные обновления и сохраняет выбор пользователя при изменении списка вариантов.

В примере проекта DropdownField динамически обновляет свойство choices на основе значений LocalizedString. При выборе новой локали язык обновляется и в вариантах раскрывающегося списка.

Таким образом, предварительная обработка служит дополнительным инструментом для создания локализованных интерфейсов, учитывающих контекст. Smart Strings решают многие задачи локализации, включая заполнители и формы множественного числа, а предварительная обработка обеспечивает дополнительную гибкость и форматирование в случаях, с которыми Smart Strings самостоятельно не справляются.

Локализация ресурсов

Хотя основная часть локализации связана со строками, помимо текста иногда требуется локализовать и ресурсы. Например, в демонстрационном проекте предусмотрены значки, соответствующие настроенным локалям.



Каждую локаль обозначают флаг и значок «Hello, world».

Настройка локализации ресурсов

Локализация ресурсов устроена почти так же, как локализация строк. Для локализованного текста используются таблицы строк, а для локализованных ресурсов - таблицы ресурсов (Asset Tables). Рабочий процесс у них схож: в таблицы добавляют записи, на которые затем ссылаются из скриптов или UXML-файлов.

Локализованные ресурсы можно привязать к элементам интерфейса в UI Builder или в скрипте C#. Например, свойство style.backgroundImage визуального элемента можно связать с локализованным спрайтом или текстурой.

В примере проекта:

- Привязка данных одного элемента задана в UXML с помощью UI Builder.

- Привязка другого элемента настроена в скрипте C#.

Теперь при выборе локали во время выполнения значки обновляются вместе с текстовыми метками, наглядно показывая активную локаль.

Объекты LocalizedTexture обновляются при каждой смене локали.

Таблицы ресурсов и таблицы строк

Работа с таблицами ресурсов во многом повторяет работу с таблицами строк. Оба типа позволяют задавать записи для каждой локали и получать их во время выполнения. Однако между ними есть различия:

- Обработка событий: таблицы ресурсов уведомляют об изменении локализованных ресурсов событием AssetChanged, тогда как для строк используется StringChanged.

- Методы привязки: и строки, и ресурсы можно привязывать методом SetBinding, но целевое свойство зависит от типа данных - например, text для строк и style.backgroundImage для текстур.

В следующем фрагменте демонстрационный код получает LocalizedTexture из таблицы ресурсов по имени, а затем привязывает её к свойству style.backgroundImage:

```
m_LocalizedTexture = new LocalizedTexture()
{
   TableReference = "DemoAssetTable",
   TableEntryReference = "HelloWorld_Icon"
};

m_IconElement = root.Q<VisualElement>("icon__hello-world");
m_IconElement.SetBinding("style.backgroundImage", m_LocalizedTexture);
```

Распространённые локализуемые ресурсы в UI Toolkit

Локализуемые ресурсы бывают разных типов. При работе с UI Toolkit чаще всего встречаются следующие:

- Локализованные текстуры: идеально подходят для значков, фонов и других декоративных изображений. Их можно напрямую привязать к свойствам визуального элемента, например style.backgroundImage.

- Локализованные спрайты: встречаются реже, но полезны для пользовательских компонентов и графики на основе спрайтов.

- Локализованные шрифты: позволяют переключать шрифты, чтобы поддерживать письменность или особенности типографики разных языков.

- Локализованные объекты: подходят для ссылок на сложные ресурсы, например префабы или ресурсы на основе данных, которые должны меняться в зависимости от локали.

Используя эти ресурсы вместе с таблицами ресурсов, можно динамически адаптировать к активной локали не только текст интерфейса, но и его визуальное оформление.

Локализация в примере Dragon Crashers

Демонстрационный проект UI Toolkit Sample - Dragon Crashers показывает несколько приёмов локализации в действии. На экране Settings можно выбрать один из поддерживаемых языков в раскрывающемся меню. После выбора нового языка система LocalizationSettings обнаруживает изменение и обновляет интерфейс в реальном времени.

Выберите локаль в раскрывающемся меню Language.

Самостоятельно изучая проект, обратите внимание на следующие аспекты:

- Выбор локали в SettingsScreen: экран настроек позволяет выбрать локаль из раскрывающегося меню. Интерфейс отслеживает изменения LocalizationSettings, обнаруживает выбор новой локали и обновляется в реальном времени.

- Приёмы привязки данных: в интерфейсе сочетаются разные способы локализации. Статические свойства привязываются непосредственно в UI Builder и сохраняются в UXML.

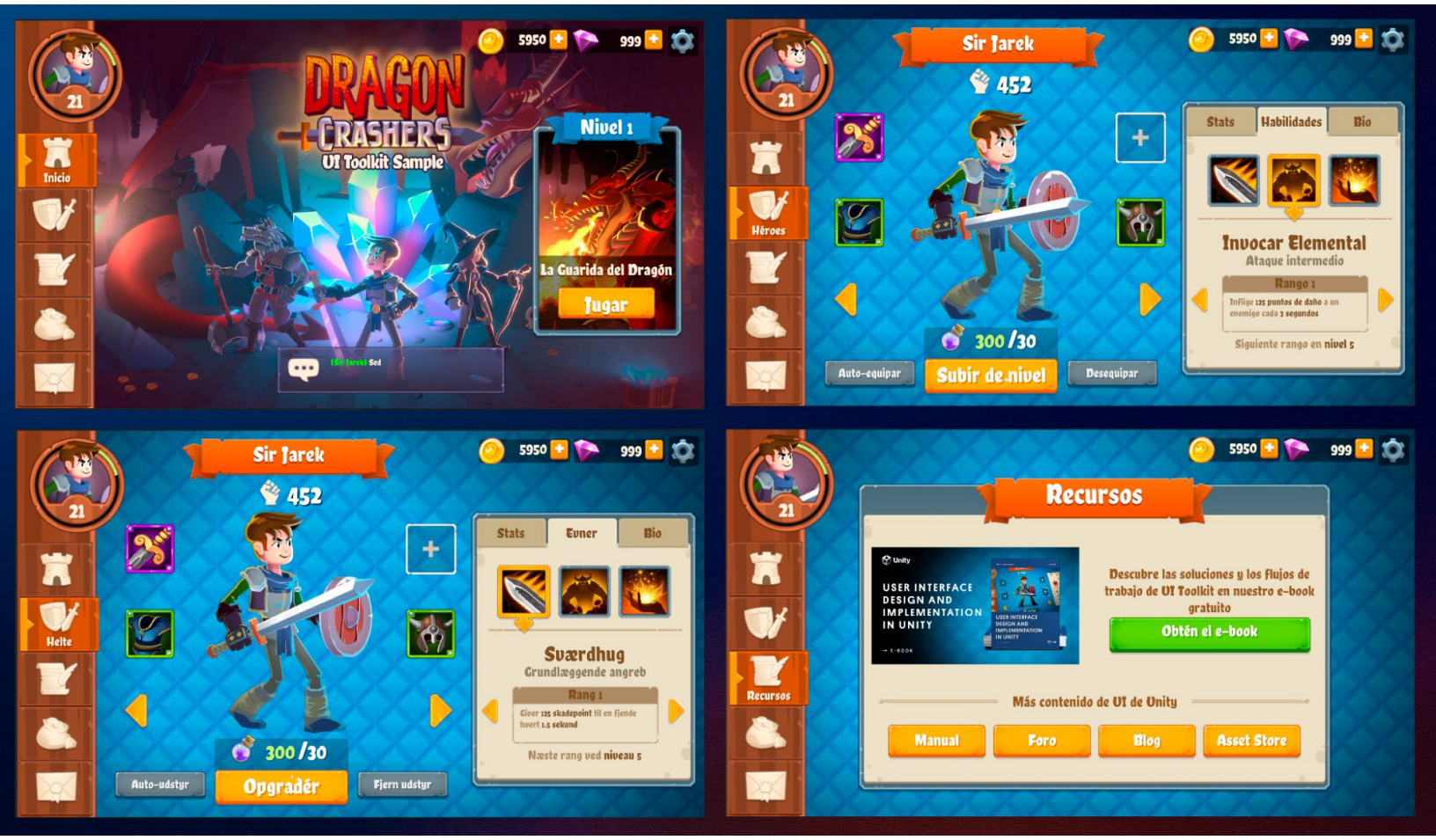
    Для динамически заполняемых полей привязку данных выполняют скрипты во время выполнения. Метод SetBinding связывает текстовые свойства с объектами LocalizedString, поэтому интерфейс соответствует выбранной локали.

- Предварительно отформатированные объекты LocalizedString в ScriptableObject: некоторые ресурсы ScriptableObject содержат заранее отформатированные свойства LocalizedString. Например, на экране Settings раскрывающиеся поля Theme и Language динамически перестраивают списки из локализованных значений, переводя доступные варианты.

    Другие элементы, такие как RadioButtonGroup и пользовательский SlideToggle, также предварительно обрабатывают LocalizedString с помощью события StringChanged.

Независимо от способа локализации - привязки данных в UXML или настройки из скрипта C# - интерфейс реагирует на смену локали в реальном времени.

Используйте решения из демонстрационного проекта как основу для локализованных интерфейсов в собственных проектах Unity. Сочетание привязки данных и UI Toolkit позволяет создать гибкий многоязычный интерфейс для игроков со всего мира.



Дополнительные примеры локализации представлены в проекте UI Toolkit Sample - Dragon Crashers.

Пользовательские элементы управления

UI Toolkit предоставляет стандартный набор элементов для построения интерфейсов, но вы также можете создавать пользовательские элементы управления под задачи своего приложения.

Например, индикатор здоровья может менять цвет в зависимости от текущего значения: по мере его снижения плавно переходить от зелёного к жёлтому и красному. Такой элемент можно без дополнительной настройки использовать для разных персонажей или приспособить для других показателей, например маны или силы. Этот самодостаточный элемент управления станет более наглядной заменой стандартному ползунку из библиотеки UI Toolkit.

Пользовательские элементы управления позволяют заключить функциональность в самостоятельные компоненты и повторно использовать их в разных частях интерфейса. Хорошо спроектированный элемент не зависит от конкретного контекста, самодостаточен и способствует повторному использованию кода, упрощая сопровождение проекта. Не стоит создавать пользовательский элемент управления для части интерфейса, которая жёстко связана с конкретным компонентом и не имеет самостоятельной функции, например для игрового меню целиком.

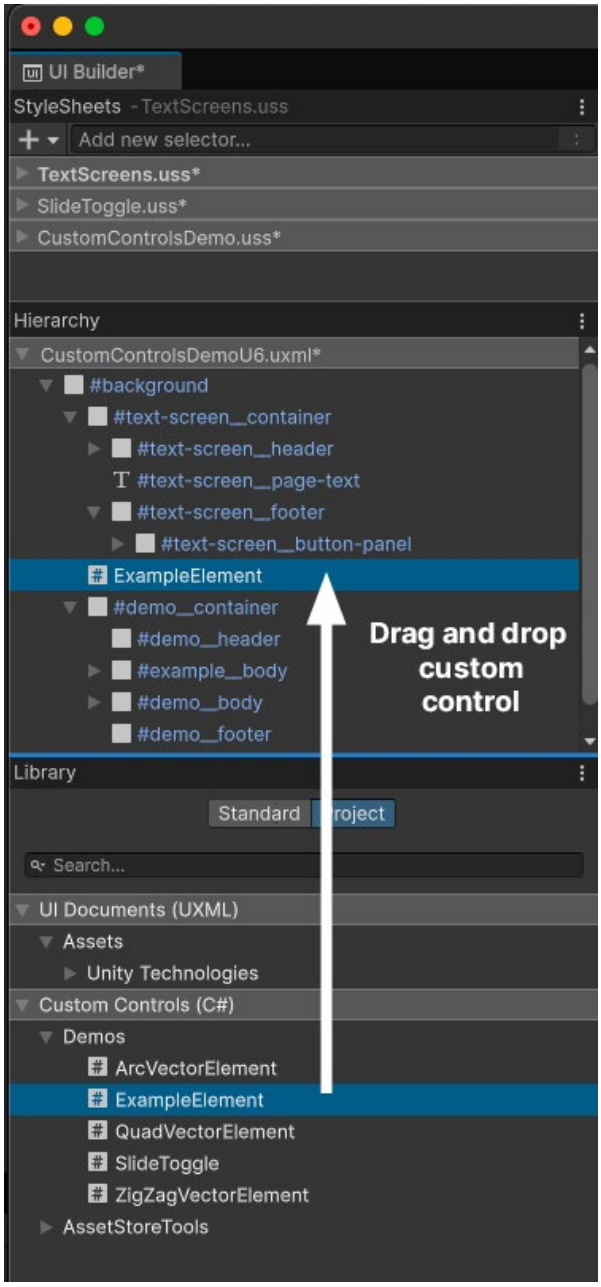
Атрибут UxmlElement

Чтобы создать пользовательский элемент управления, определите новый C#-скрипт, класс которого наследуется от VisualElement или от более подходящего производного класса. Нужен элемент, похожий на кнопку? Унаследуйте его от Button.

Чтобы пользовательский элемент управления стал доступен в UXML и UI Builder, добавьте к его классу атрибут UxmlElement. Сам класс должен быть объявлен как public partial:

```
[UxmlElement]
public partial class ExampleElement: VisualElement
{


}
```

После этого пользовательский элемент управления появится в разделе Library UI Builder, в категории Custom Controls (C#). Оттуда его можно перетащить в окно Hierarchy.



Пользовательские элементы управления отображаются в библиотеке UI Builder.

Визуальные элементы не являются GameObject, поэтому у них нет привычных событий жизненного цикла Awake, OnEnable, OnDisable и OnDestroy. Пользовательский элемент управления инициализируется в конструкторе.

```
[UxmlElement]
public partial class ExampleElement: VisualElement
{

   // Constructor
   public ExampleElement()
   {
      // Initialization
   }
}
```

Инициализацию также можно отложить до добавления пользовательского элемента в интерфейс. Для этого зарегистрируйте обратный вызов AttachToPanelEvent.

Чтобы определить, когда элемент удалён из интерфейса, используйте обратный вызов DetachFromPanelEvent.

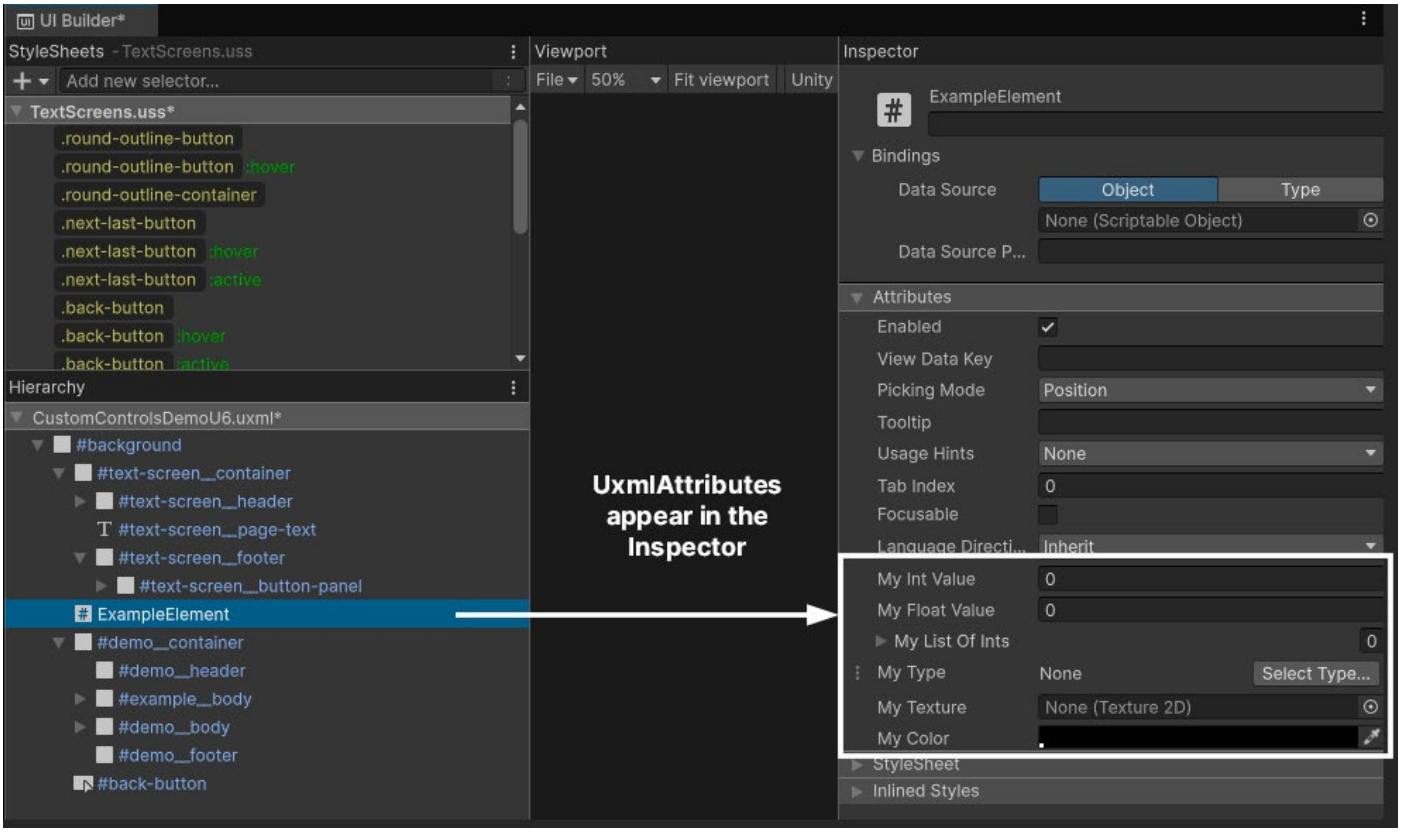
Атрибут UxmlAttribute

Если добавить к свойству атрибут UxmlAttribute, оно появится в окне Inspector UI Builder и начальное значение можно будет задать интерактивно. Это особенно удобно при совместной работе с дизайнером: изменения в Inspector не требуют правки кода.

Примените UxmlAttribute к каждому свойству, которое требуется открыть для настройки. Имя атрибута можно изменить аргументом name.

При выборе элемента управления в Hierarchy его пользовательские атрибуты отображаются в Inspector, где их можно настроить напрямую.

Атрибуты-декораторы изменяют представление полей пользовательских атрибутов примерно так же, как при работе с MonoBehaviour. Среди полезных декораторов - TextArea, Tooltip, Range, Header, Min, Multiline, Space и Delayed. Например, Range добавляет ползунок для выбора значения в заданном диапазоне.



Пользовательские атрибуты отображаются в Inspector UI Builder.

Ниже приведён простой пример пользовательского элемента управления с атрибутом UxmlElement и двумя открытыми свойствами, отмеченными UxmlAttribute.

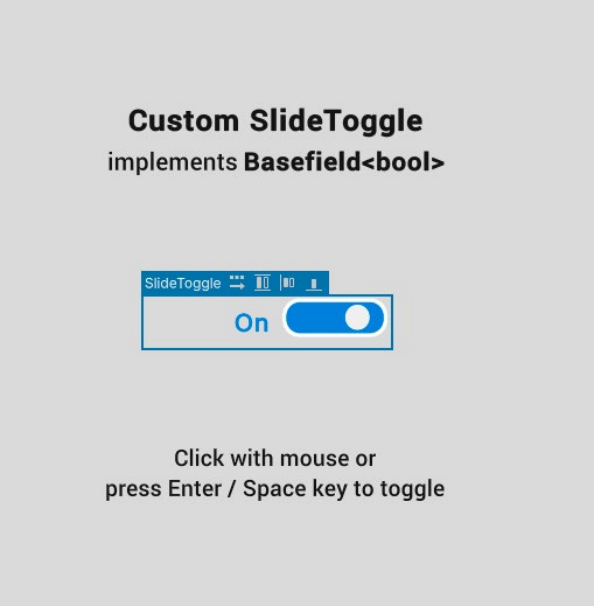
```
[UxmlElement]
public partial class ExampleElement: VisualElement
{
   [UxmlAttribute(name:"my-text")]
   public string myStringValue { get; set; }

   [UxmlAttribute]
   public int myIntValue { get; set; }

}
```

В этом примере благодаря параметру name свойство MyStringValue отображается в Inspector под именем "My Text". Свойства MyStringValue и MyIntValue можно редактировать в Inspector, когда в Hierarchy выбран экземпляр ExampleElement.

До Unity 6 для создания пользовательских элементов управления требовалось реализовывать классы UxmlTraits и UxmlFactory, которые отвечали за регистрацию атрибутов и создание экземпляров пользовательских элементов.

В Unity 6 появились атрибуты UxmlElement и UxmlAttribute, упрощающие этот процесс. Они напрямую делают пользовательские элементы управления и их свойства доступными в UXML и UI Builder. Новый рабочий процесс требует меньше шаблонного кода и ускоряет настройку элементов интерфейса.

Пример: пользовательский переключатель-ползунок



Пользовательский переключатель-ползунок представляет логическое значение.

Простой пример пользовательского элемента управления - переключатель-ползунок, представляющий логическое значение.

Он может быть привлекательнее стандартного переключателя. Дополнительная визуальная обратная связь - анимация ползунка, смена цвета и динамический текст - делает интерфейс понятнее.

Определение пользовательского элемента управления

Простая реализация такого элемента находится в сцене CustomControlsDemo проекта QuizU. Чтобы разобраться в её работе, откройте скрипт SlideToggle.cs; его фрагменты приведены ниже.

Пользовательский элемент SlideToggle наследуется от наиболее подходящего базового класса - в данном случае BaseField<bool>. Атрибут UxmlElement делает элемент доступным в UXML и UI Builder, благодаря чему его можно использовать повторно.

```
[UxmlElement]
public partial class SlideToggle : BaseField<bool>
{
    // …
```

```
    [UxmlAttribute]
    public string EnabledText { get; set; } = "Enabled";

    [UxmlAttribute]
    public string DisabledText { get; set; } = "Disabled";

    [UxmlAttribute]
    public Color EnabledBackgroundColor { get; set; } = new Color(0f, 0.5f, 0.85f,1f);

    [UxmlAttribute]
    public Color DisabledBackgroundColor { get; set; } = Color.gray;
```

Визуальная структура состоит из фона (m_Input) и ползунка (m_Knob), а внешний вид задаётся классами USS.

```
public SlideToggle(string label) : base(label, new VisualElement())
{
    AddToClassList(ussClassName);

    m_Input = this.Q(className: BaseField<bool>.inputUssClassName);
    m_Input.AddToClassList(inputUssClassName);
    m_Input.name = "input";

    m_Knob = new();
    m_Knob.AddToClassList(inputKnobUssClassName);
    m_Knob.name = "knob";
    m_Input.Add(m_Knob);

    labelElement.name = "label";
    labelElement.text = (value) ? "enabled" : "disabled";
```

Обработчики реагируют на щелчки, нажатия клавиш и события навигации, поэтому состояние элемента можно менять несколькими способами.

```
  // …
   RegisterCallback<ClickEvent>(evt => OnClick(evt));
   RegisterCallback<KeyDownEvent>(evt => OnKeydownEvent(evt));

   // …
}

static void OnClick(ClickEvent evt)
{
   var slideToggle = evt.currentTarget as SlideToggle;
   slideToggle.ToggleValue();
   evt.StopPropagation();
}

static void OnKeydownEvent(KeyDownEvent evt)
{
   var slideToggle = evt.currentTarget as SlideToggle;

   if (slideToggle.panel?.contextType == ContextType.Player)
       return;

   if (evt.keyCode == KeyCode.KeypadEnter || evt.keyCode == KeyCode.Return || evt.keyCode ==
KeyCode.Space)
   {
       slideToggle.ToggleValue();
       evt.StopPropagation();
   }
}
```

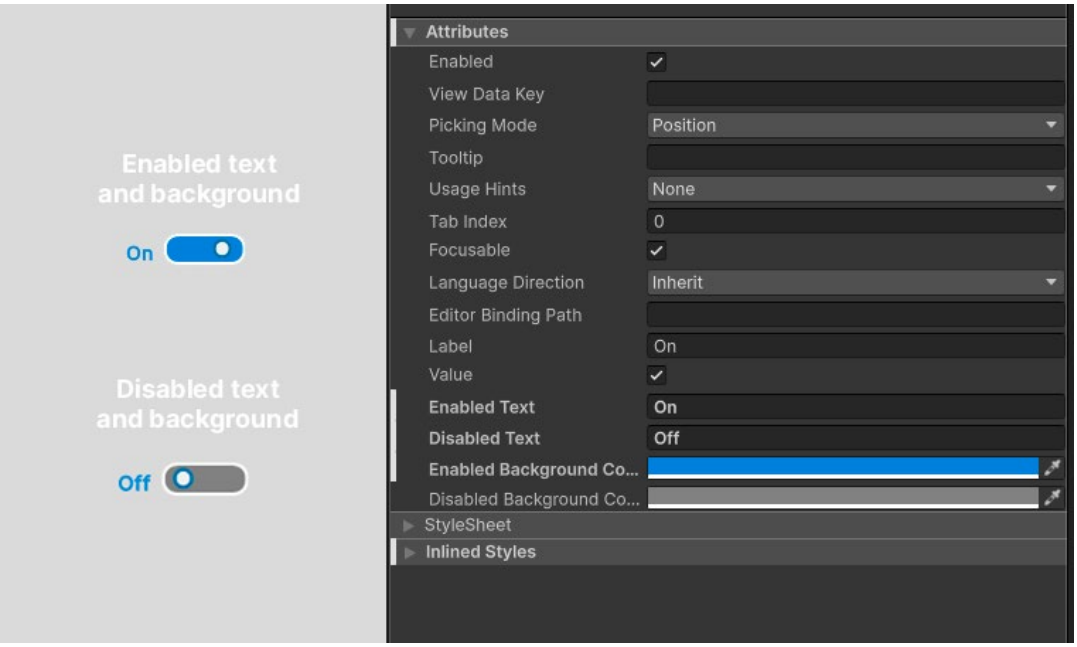
При переключении элемента надпись и цвет фона автоматически обновляются, обеспечивая визуальную обратную связь.

Здесь SetValueWithoutNotify обновляет визуальное состояние переключателя, не вызывая ChangeEvent. Этот метод вызывается внутри элемента при изменении значения, поэтому интерфейс обновляется корректно и не попадает в бесконечный цикл обновлений.

```
public override void SetValueWithoutNotify(bool newValue)
{
   base.SetValueWithoutNotify(newValue);

   m_Input.EnableInClassList(inputCheckedUssClassName, newValue);

   m_Input.style.backgroundColor = newValue ? EnabledBackgroundColor :
DisabledBackgroundColor;
   labelElement.text = (value) ? EnabledText : DisabledText;
}
```

Изучите пример реализации в сцене CustomControlsDemo. Чтобы переключить активное состояние элемента, щёлкните по нему мышью или нажмите Enter либо пробел. В этом примере при переключении динамически меняются надпись и цвет фона, а короткая анимация обеспечивает визуальную обратную связь.

В Inspector задайте надписи и цвета фона для включённого и выключенного состояний.



Настройте текст и цвета SlideToggle.

Использование SlideToggle

После компиляции SlideToggle можно встраивать в любую часть интерфейса. Используйте пользовательский класс SlideToggle так же, как любой другой визуальный элемент. Ниже приведён пример, в котором SlideToggle включает и отключает звук:

```
public class MuteAudioToggle : MonoBehaviour
{
   [SerializeField] AudioSettingsSO m_AudioSettingsSO;
   [SerializeField] UIDocument m_Document;

   void OnEnable()
   {
       var root = m_Document.rootVisualElement;
       SlideToggle slideToggle = root.Q<SlideToggle>("master-audio-toggle");
```

```
        if (slideToggle != null)
        {
            slideToggle.value = !m_AudioSettingsSO.IsMasterMuted;

            slideToggle.RegisterValueChangedCallback(evt => m_AudioSettingsSO.IsMasterMuted =
!evt.newValue);
        }
    }
}
```

Здесь SlideToggle входит в существующий UXML-документ. MonoBehaviour находит его по имени в визуальном дереве, а затем методом RegisterValueChangedCallback связывает состояние переключателя с настройками звука.

Поскольку SlideToggle - самостоятельный пользовательский элемент, его можно применять для любых переключателей в интерфейсе. Например, в проекте UI Toolkit Sample - Dragon Crashers похожий SlideToggle включает и отключает счётчик кадров в секунду.

Стилизованный переключатель из проекта UI Toolkit Sample - Dragon Crashers

Адаптируйте SlideToggle к требованиям приложения: он отлично подходит для настроек графики, звука и игрового процесса. Создайте его один раз и используйте везде, где нестандартный переключатель улучшит взаимодействие с пользователем.

Полную реализацию см. в скрипте SlideToggle.cs проекта QuizU.

Создание других пользовательских элементов управления

Если нужного элемента нет в стандартной библиотеке UI Toolkit, его можно создать самостоятельно. Вот несколько примеров применения пользовательских элементов управления в играх:

- Индикаторы здоровья и прогресса. Такие игровые показатели, как здоровье, мана и сила, сильно зависят от игрового процесса, поэтому хорошо подходят для пользовательских элементов управления. Откройте через UxmlAttribute максимальное и текущее значения, а также цвета состояний, чтобы можно было настраивать цветовые градиенты.

- Рейтинговые звёзды. Этот элемент работает как сегментированный индикатор прогресса и представляет целое значение, например количество звёзд за прохождение уровня. Создайте визуальный элемент с несколькими дочерними элементами, которые переключаются между заполненным и пустым состояниями. Откройте в Inspector целочисленное значение и его максимум, а изображения спрайтов позвольте настраивать через UxmlAttribute.

- Представление с вкладками. Вкладки часто используются для переключения между представлениями или разделами одного окна. Создайте пользовательский элемент со строкой вкладок и областью содержимого. Каждая вкладка может быть визуальным элементом, похожим на кнопку; предусмотрите возможность динамически добавлять и удалять вкладки.

Не забывайте, что в большинстве случаев визуальный эффект можно усилить анимированными переходами USS. Пользовательские элементы управления позволяют игрокам использовать жесты, щёлкать, прокручивать и переключать элементы уникального интерфейса вашей игры.

Нам не терпится увидеть, что вы создадите.

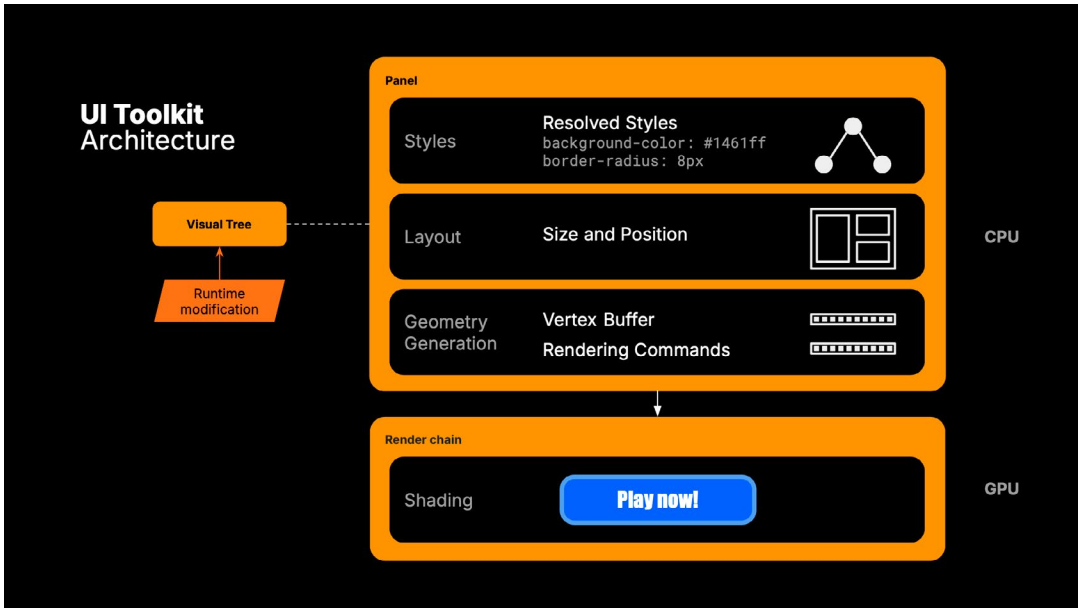
Оптимизация производительности

Сложный игровой интерфейс часто включает обширную иерархию экранных элементов. Когда их сотни, возникают технические трудности: даже небольшие просчёты в производительности накапливаются и приводят к рывкам или кратковременным зависаниям во время выполнения, ухудшая впечатления игрока.

К счастью, большинство таких проблем решается оптимизацией. В Unity 6 UI Toolkit заметно усовершенствован и уже в стандартной конфигурации работает быстрее, однако по-настоящему эффективный интерфейс всё равно требует усилий разработчика.

Во многом задача сводится к устранению лишних накладных расходов и сокращению количества вызовов отрисовки. Рассмотрим несколько способов оптимизации UI Toolkit в Unity 6.

Механизмы обновления



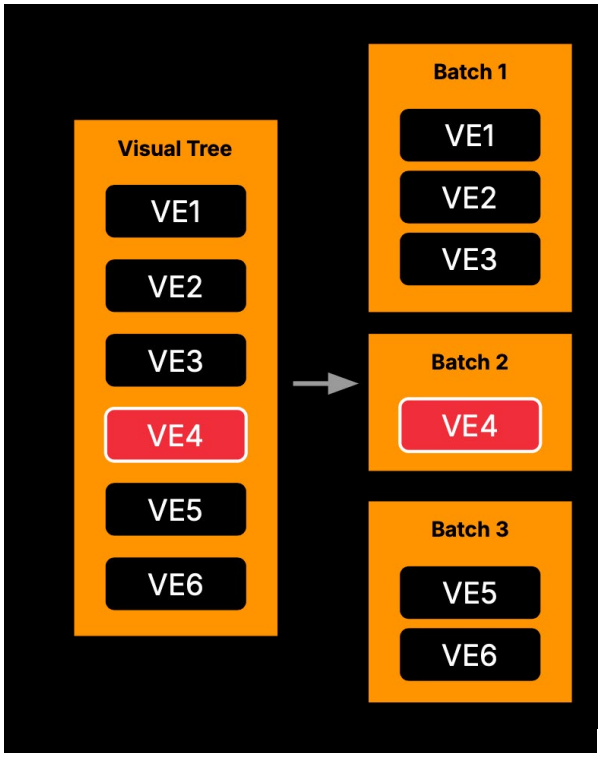
Визуальное дерево использует несколько механизмов обновления.

Визуальное дерево содержит несколько механизмов обновления, которые реагируют на изменения стилей, компоновки и содержимого во время выполнения. Каждый из них может влиять на производительность. В таблице кратко описано, когда срабатывают эти механизмы и как они отражаются на производительности.

Механизм	Описание	Когда срабатывает	Влияние
Разрешение стилей	Определяет окончательный вид элементов, применяя селекторы и стили USS.	При изменении классов или стилей, например при добавлении класса стиля или смене цвета.	В больших или глубоко вложенных иерархиях операция обходится дорого. Избегайте частых изменений.
Перерасчёт компоновки	Корректирует размер и положение элементов в иерархии интерфейса.	При изменении размера, положения или выравнивания элемента, например размера панели или положения элементов.	Частые обновления компоновки требуют значительных ресурсов. Для анимации используйте преобразования.
Обновление буфера вершин	Обновляет геометрию элементов интерфейса: прямоугольники, скруглённые углы и другие формы.	При изменении геометрии элемента, например при добавлении скруглений или изменении границ.	Обновление буферов вершин ресурсоёмко. Избегайте частых изменений геометрии.
Изменение состояния рендеринга	Меняет необходимые для отрисовки состояния, например текстуры и режимы смешивания.	При использовании масок, уникальных текстур и других возможностей, нарушающих пакетную обработку.	Частая смена состояний повышает нагрузку на CPU. Применяйте пакетную обработку и ограничивайте уникальные текстуры и маски.

Разумеется, стоимость этих операций зависит от частоты и масштаба изменений элементов интерфейса.

Пакетная обработка элементов



Разрыв пакетов снижает производительность.

При отрисовке интерфейса для каждого визуального элемента необходимо отправить инструкции на GPU. UI Toolkit оптимизирует эти вызовы отрисовки с помощью пакетной обработки: элементы с одинаковыми требованиями к GPU объединяются и обрабатываются вместе. Это значительно сокращает затраты на обмен данными с GPU, подобно пакетной обработке вызовов отрисовки для GameObject.

Для эффективного объединения элементы должны использовать одно и то же состояние GPU: одинаковые шейдеры, текстуры, данные сетки и другие параметры. Например, последовательность текстовых элементов с одним шрифтом и стилем можно обработать одним пакетом. Если вставить между ними изображение, потребуются другие параметры GPU и будет создан новый пакет.

Каждый такой разрыв пакета немного снижает эффективность. Для высокой производительности структуру интерфейса следует организовать так, чтобы свести число разрывов к минимуму.

Поскольку каждый пакет может породить один или несколько вызовов отрисовки на GPU, меньшее число пакетов обычно означает меньшие накладные расходы и более высокую производительность.

Далее рассмотрим способы сократить количество пакетов и добиться стабильной работы интерфейса.

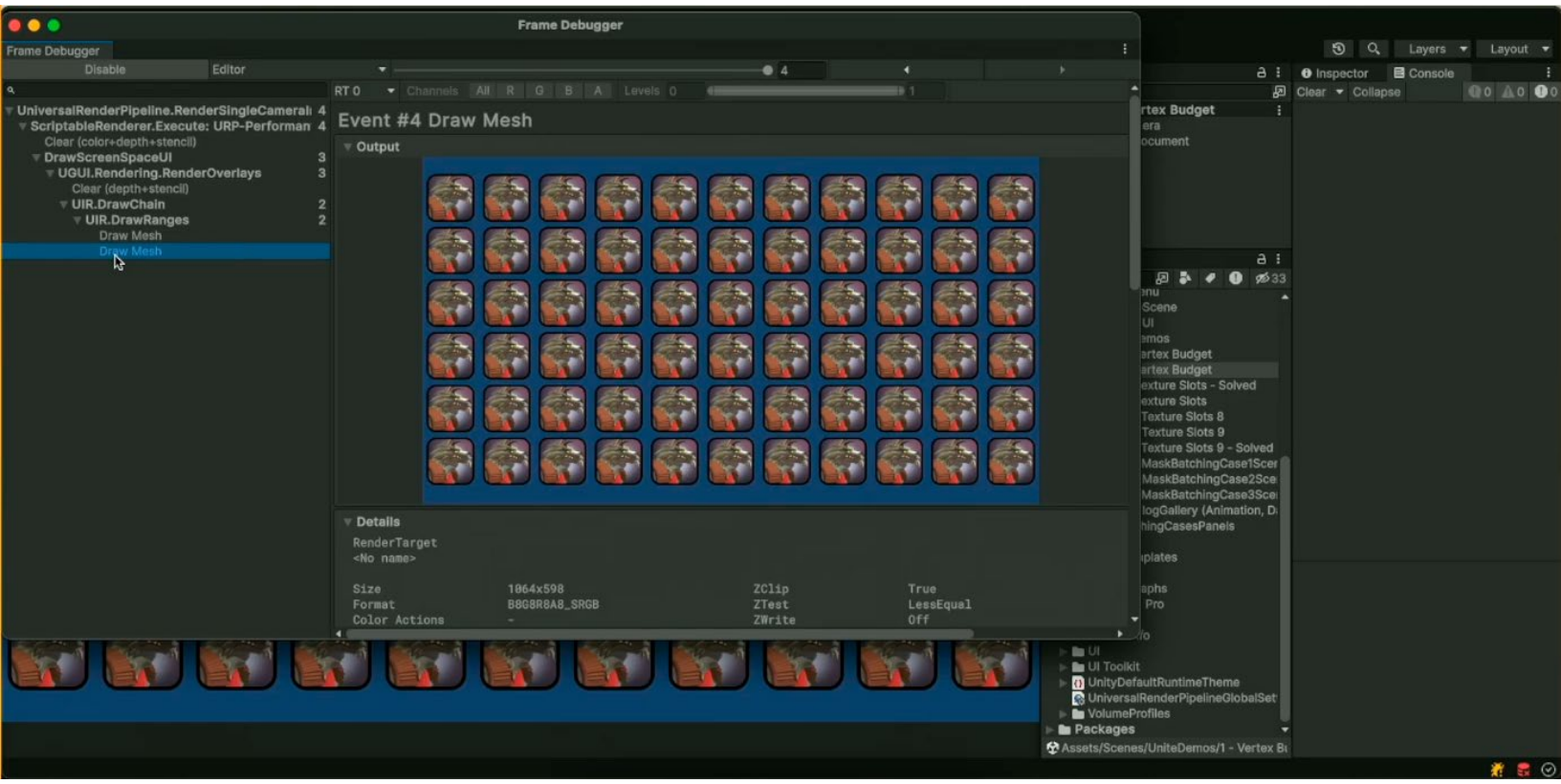
Буферы вершин

В UI Toolkit буферы вершин хранят геометрию, необходимую для отрисовки интерфейса. Когда UIDocument создаёт Panel во время выполнения, для визуальных элементов заранее выделяется один буфер вершин. Его можно представить как распределитель памяти в куче: по мере добавления элементов в интерфейс он динамически выделяет им память.

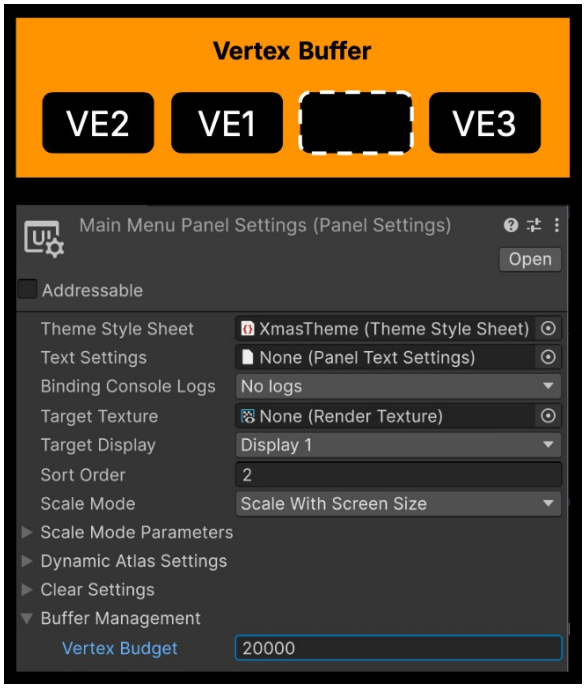
Если интерфейс превышает ёмкость буфера, создаются дополнительные буферы. Это может раздробить пакеты, увеличить число вызовов отрисовки и в итоге снизить производительность.

Начальный размер буфера настраивается параметром Vertex Budget в Panel Settings. Значение по умолчанию 0 позволяет Unity определить размер автоматически. Однако для сложных интерфейсов ручное увеличение этого значения иногда повышает производительность за счёт сокращения числа вызовов отрисовки.

Рассмотрим пример. Интерфейс содержит множество элементов, которые не помещаются в один буфер вершин. Frame Debugger показывает, что в результате вместо одного вызова отрисовки выполняются два.

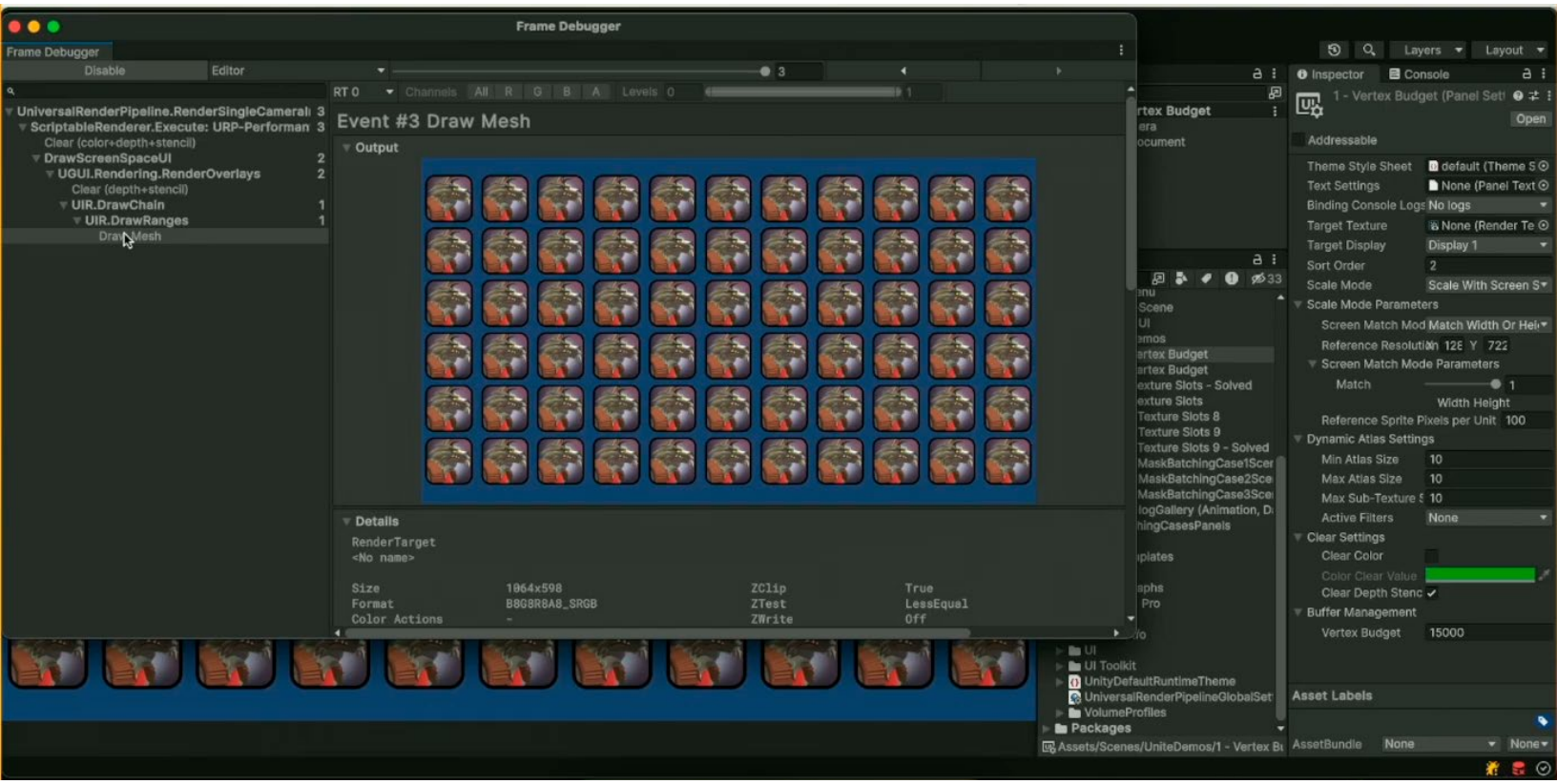


Для этого интерфейса требуется несколько вызовов отрисовки.



Если, например, увеличить Vertex Budget до 20 000 вершин, элементы интерфейса могут поместиться в одном буфере и отрисоваться за один вызов. Таким образом, изменение единственного параметра повышает эффективность интерфейса в нашем примере.

Увеличение Vertex Budget может сократить число вызовов отрисовки.



Настройка Vertex Budget восстанавливает отрисовку одним вызовом.

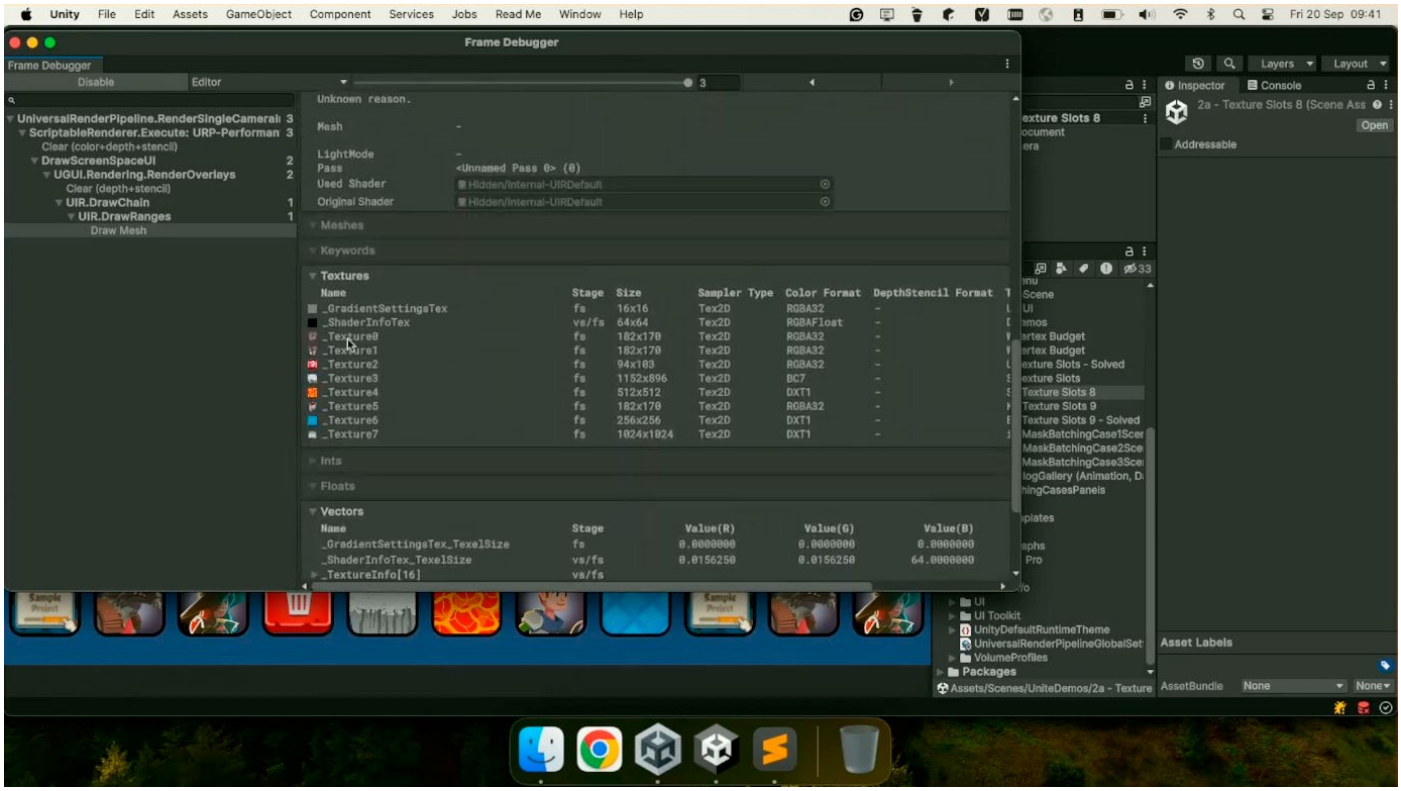
Для сложных интерфейсов ручное увеличение этого значения может повысить производительность благодаря сокращению числа вызовов отрисовки, но избегайте избыточного выделения памяти. С помощью Frame Debugger и Unity Profiler найдите оптимальный баланс между объёмом памяти и количеством вызовов отрисовки.

Универсальный шейдер и ограничение в восемь текстур

UI Toolkit объединяет всю функциональность отрисовки интерфейса в одном универсальном uber-шейдере. Вместо нескольких вариантов шейдера он использует динамическое ветвление, выбирая нужный путь рендеринга во время выполнения. Благодаря меньшему числу переключений шейдеров снижается нагрузка на CPU, хотя логика ветвления несколько увеличивает стоимость обработки на GPU.

Одна из важных возможностей этого шейдера - поддержка до восьми текстур в одном пакете. Поэтому элементы с разными текстурами можно отрисовать одним вызовом. На следующем изображении показан интерфейс, состоящий из восьми разных текстур:

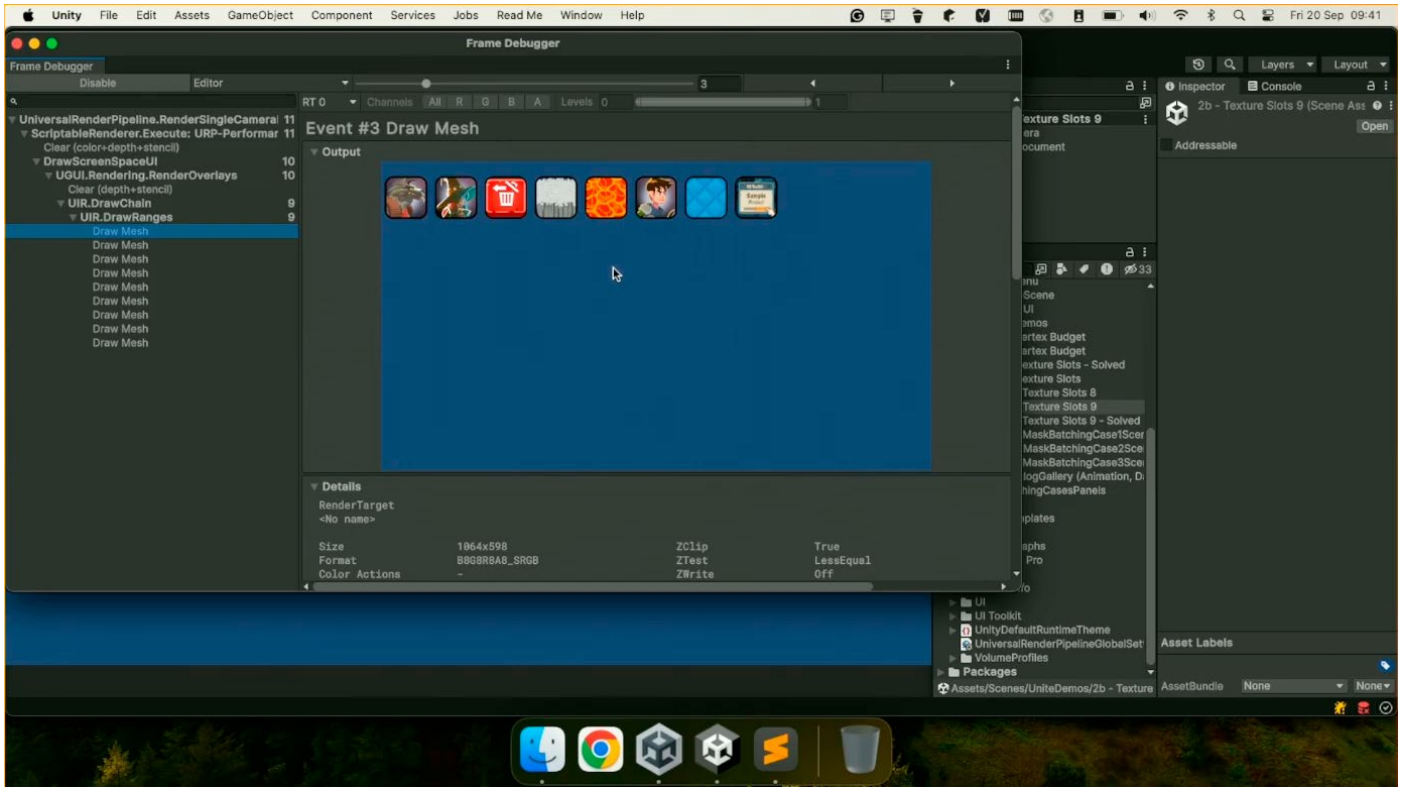
Интерфейс в этом примере содержит восемь текстур.



Универсальный uber-шейдер отрисовывает их одним вызовом.

Как видно в Frame Debugger, Unity отрисовывает пример интерфейса с восемью текстурами одним вызовом. Однако при превышении этого ограничения система вынуждена разделять элементы на отдельные пакеты, увеличивая накладные расходы.

Ниже показано, что происходит при использовании более восьми текстур: вместо одного вызова отрисовки выполняется сразу несколько.



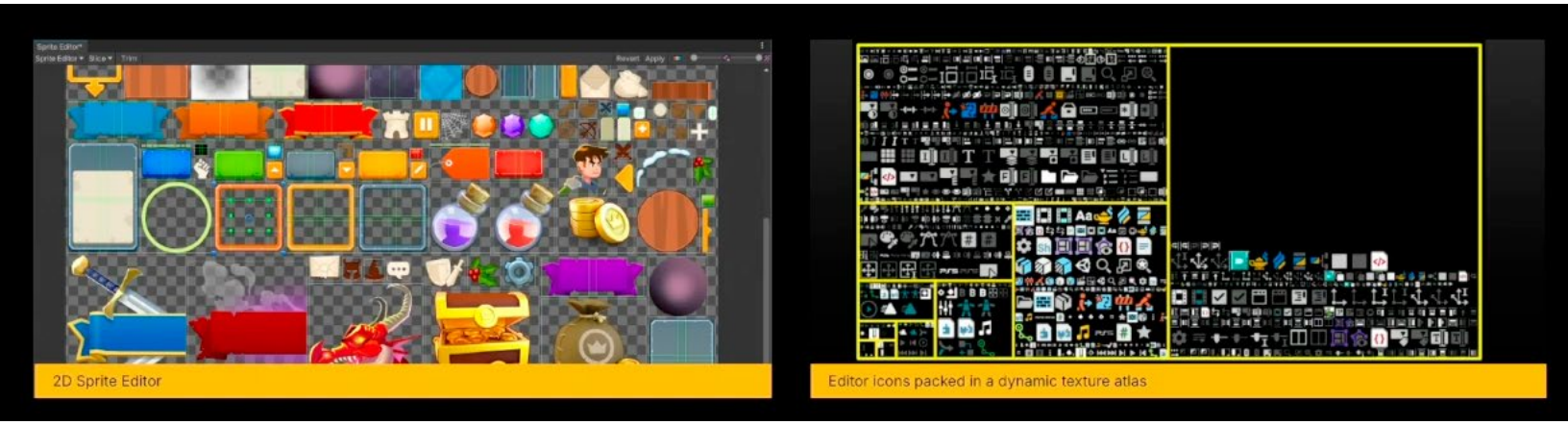
Избыток текстур приводит к разрыву пакетов.

Чтобы обойти это ограничение, UI Toolkit предоставляет инструменты оптимизации текстур. Например, объединение текстур в атласы позволяет удержать их число в поддерживаемых пределах, сохранить эффективность пакетной обработки и сократить количество вызовов отрисовки.

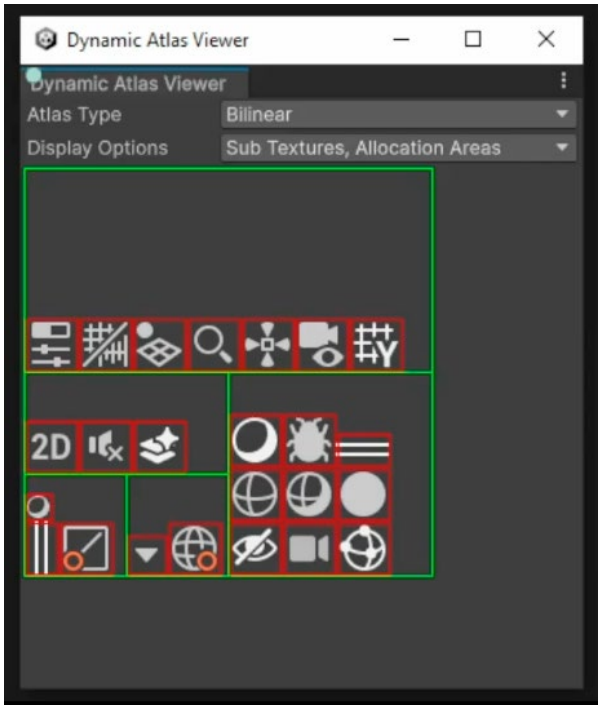
Динамические атласы текстур

Переключение между множеством текстур может вынудить UI Toolkit разбивать пакеты, увеличивая число вызовов отрисовки и снижая производительность. Распространённое решение - атлас текстур, объединяющий несколько небольших текстур в одну крупную. Если вы знакомы с 2D Sprite Atlas, то уже знаете эффективный способ повысить производительность. Когда несколько спрайтов упакованы в атлас, Unity обрабатывает их как одну текстуру, что сокращает разрывы пакетов и вызовы отрисовки.

2D Sprite Atlas без труда интегрируется с UI Toolkit и отлично подходит для статического или заранее подготовленного содержимого. Но у него есть ограничения: например, он не поддерживает текстуры, создаваемые во время выполнения. Кроме того, атлас требует предварительной настройки и компоновки спрайтов, что может занять немало времени.



В примере Dragon Crashers используется 2D Sprite Atlas.



Используйте Dynamic Atlas Viewer в отладчике UI Toolkit.

Динамический атлас текстур UI Toolkit эффективно объединяет несколько изображений в одну текстуру, сокращая число изменений состояния. Параметры атласа настраиваются в Panel Settings, а его компоновку можно просмотреть в Dynamic Atlas Viewer, доступном в окне UI Toolkit Debugger.

Если интерфейс значительно меняется со временем, например в него часто добавляются и из него удаляются текстуры, атлас может фрагментироваться. В таком случае API ResetDynamicAtlas восстановит его исходное состояние.

В UI Toolkit можно одновременно использовать 2D Sprite Atlas и динамические атласы текстур. Атласы спрайтов лучше всего подходят для статического, заранее подготовленного содержимого, а динамические атласы - для интерфейса, содержимое которого формируется во время выполнения.

Маскирование

UI Toolkit создаёт маски - области, показывающие или скрывающие части элементов интерфейса, - с помощью буфера трафарета. Поскольку этот буфер входит в состояние GPU, изменение параметров маски может привести к разрыву пакетов.

Иерархическое наложение маскированных элементов дополнительно усложняет обработку: на каждом уровне вложенности буфер трафарета должен отслеживать ещё одно состояние, что повышает нагрузку на GPU.

UI Toolkit поддерживает два типа маскирования:

- Прямоугольное маскирование. Прямоугольные маски используют операции на уровне шейдера, сохраняя целостность пакетов без изменения состояния GPU. Буфер трафарета при этом не задействуется, поэтому глубина вложенности прямоугольных масок не ограничена.

- Скруглённые углы и сложные маски (буфер трафарета). Для скруглённых углов и других сложных форм требуются операции с буфером трафарета, которые могут разрывать пакет на каждом уровне маскирования. Поддерживается до семи уровней вложенности.

Прямоугольные маски и маски со скруглёнными углами

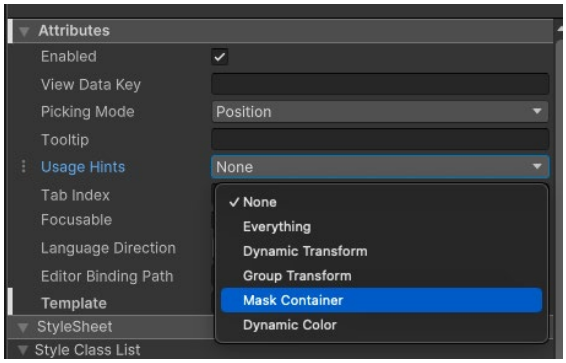
Чтобы повысить производительность маскированных элементов:

- По возможности используйте прямоугольные маски, чтобы избежать операций с буфером трафарета.

- Сведите глубину вложенности масок к минимуму. Плоская иерархия требует меньше перерасчётов буфера трафарета.

- По возможности применяйте одну маску к родительскому элементу вместо нескольких масок у дочерних элементов.

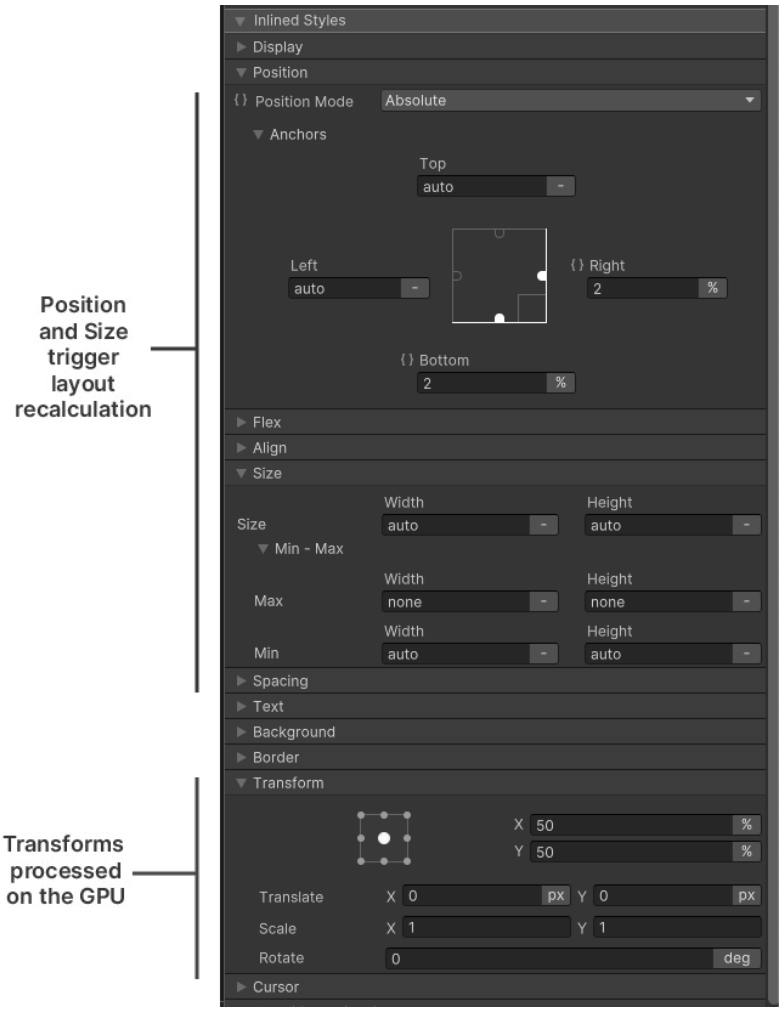
- Если без нескольких уровней маскирования не обойтись, примените указание по использованию Mask Container, чтобы оптимизировать настройку состояния буфера трафарета. Используйте его умеренно, поскольку оно может привести к разрыву пакетов.



Используйте указание по использованию Mask Container.

Наконец, оцените результат оптимизации в Frame Debugger и убедитесь, что отрисовка и пакетная обработка выполняются эффективно.

Анимации и переходы



Анимируйте преобразования, а не свойства компоновки.

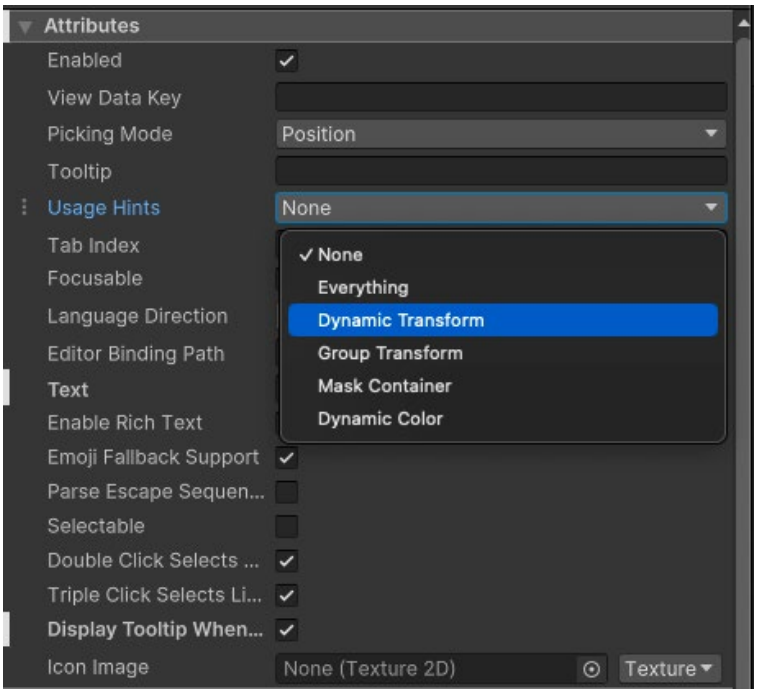
Переходы USS в UI Toolkit позволяют создавать простую анимацию свойств, однако изменение параметров компоновки, например размера или положения, может запустить дорогостоящий перерасчёт. Снизить нагрузку можно несколькими способами.

Прежде всего используйте анимацию преобразований вместо изменения свойств компоновки. Не анимируйте width, height, top или left - применяйте преобразования translate, scale и rotate. Эти операции обрабатываются непосредственно на GPU и не требуют перерасчёта компоновки, поэтому анимация получается плавнее.

Для любого анимируемого визуального элемента можно также включить указания по использованию.

Указание DynamicTransform предписывает UI Toolkit обрабатывать изменения положения и преобразований на GPU, не выполняя дорогостоящий перерасчёт данных вершин.

Для родительских контейнеров с несколькими анимируемыми дочерними элементами указание GroupTransform может значительно сократить накладные расходы. Оно применяет одно преобразование к родительскому элементу, а GPU эффективно распространяет его на всех потомков, оптимизируя анимацию больших групп.



Указания по использованию доступны для каждого визуального элемента.

Кроме того, во время анимации по возможности не переключайте классы для изменения стилей в крупных иерархиях. Смена класса может вызвать масштабный перерасчёт стилей, особенно в сложных структурах интерфейса. Чтобы снизить вычислительные затраты, изменяйте стили напрямую через встроенные свойства.

Наконец, контролируйте производительность анимации с помощью Unity Frame Debugger: этот инструмент поможет проверить, что оптимизация даёт ожидаемый результат.

Привязка данных во время выполнения

Привязка данных во время выполнения в Unity упрощает обновление элементов интерфейса: они автоматически отражают изменения исходных данных. Ручные обновления больше не требуются, поэтому разработка интерфейса становится эффективнее, а сопровождение - проще.

Этот процесс можно дополнительно оптимизировать следующими способами.

Контейнеры свойств и генерация исходного кода

Контейнер свойств (property bag) - вспомогательный объект для эффективного обхода и изменения данных определённого типа. По умолчанию Unity создаёт контейнер с помощью рефлексии при первом обращении к типу. Такой подход удобен, но вносит небольшие накладные расходы во время выполнения: генерация выполняется отложенно, если контейнер ещё не зарегистрирован.

Для повышения производительности можно включить генерацию кода контейнеров свойств. Пометьте тип атрибутом [Unity.Properties.GeneratePropertyBag] и убедитесь, что для сборки также включена генерация кода. Тогда Unity создаст и зарегистрирует контейнер на этапе компиляции, исключив рефлексию во время выполнения. Подробнее см. в документации по контейнерам свойств.

Атрибут GeneratePropertyBag оптимизирует тип целиком, а CreateProperty у отдельных свойств позволяет Unity сгенерировать код привязки на этапе компиляции. Системе не приходится искать и связывать свойства через рефлексию во время выполнения, поэтому привязка данных работает быстрее и эффективнее.

Во многих случаях для её оптимизации достаточно одного [CreateProperty]. Если тип нуждается и в других оптимизациях, например в эффективной сериализации или частом обходе всех свойств, сочетание [CreateProperty] и [GeneratePropertyBag] обеспечивает наилучшую общую производительность.

Отслеживание изменений

Привязка данных во время выполнения включает два интерфейса, которые оптимизируют частоту обновлений:

- IDataSourceViewHashProvider: выполняет проверку равенства по хешу и обновляет привязки только при существенном изменении данных.

- INotifyBindablePropertyChanged: запускает обновление только при изменении значения конкретного свойства.

Эти интерфейсы особенно полезны в сложных интерфейсах, поскольку предотвращают лишние обновления, когда данные фактически не изменились.

Ниже показан пример ScriptableObject, реализующего эти оптимизации:

```
[CreateAssetMenu(fileName = "CarData", menuName = "Scriptable Objects/CarData"),
GeneratePropertyBag]
public class CarData : ScriptableObject, INotifyBindablePropertyChanged,
IDataSourceViewHashProvider
{
   private long _version;

   [SerializeField, DontCreateProperty] string _name;
   public event EventHandler<BindablePropertyChangedEventArgs> propertyChanged;

   [CreateProperty]
   public string Name
   {
       get => _name;
       set
       {
           _name = value;
           _version++;
           Notify();
       }
   }

   void Notify([CallerMemberName] string property = "")
   {
       propertyChanged?.Invoke(this,
           new BindablePropertyChangedEventArgs(property));
   }

   public long GetViewHashCode() => _version;
}
```

Класс из примера использует [GeneratePropertyBag], чтобы создать контейнер свойств на этапе компиляции, и [CreateProperty], чтобы оптимизировать привязку свойства Name во время выполнения.

Для отслеживания изменений класс реализует INotifyBindablePropertyChanged. Метод Notify вызывает событие propertyChanged при каждом обновлении Name, сообщая об изменении интерфейсу и всем подписчикам.

Класс также реализует IDataSourceViewHashProvider. Метод GetViewHashCode возвращает версионный хеш, который увеличивается при каждом изменении Name, поэтому обновления легко обнаружить.

Отображение и скрытие элементов

Для производительности не всегда достаточно просто сделать элемент интерфейса прозрачным или переместить его за пределы экрана. Даже скрытые таким образом элементы участвуют в расчёте компоновки, обновлении стилей и привязке данных.

UI Toolkit предлагает несколько способов скрыть элемент, каждый со своими компромиссами. Краткое сравнение приведено в таблице.

Метод	Обновление привязок	Обновление компоновки	Стоимость рендера	Расчёт стилей	Стоимость смены	Память стилей	Память мешей
Opacity: 0	Да	Да	Высокая	Да	Низкая	Да	Полная
За экраном	Да	Да	Средняя	Да	Низкая	Да	Полная
Visibility: Hidden	Да	Да	Средняя	Да	Средняя	Да	Трафарет
Display: None	Да	Нет	Нет	Нет	Средняя	Да	Полная
Удаление из иерархии	Нет	Нет	Нет	Нет	Высокая	Нет	Нет

Переключайте видимость элементов разными способами.

Если установить opacity в 0 или переместить элемент за пределы экрана, он по-прежнему обрабатывается GPU и системой компоновки, поэтому стоимость отрисовки остаётся средней или высокой. Эти способы подходят для переходов, но не сокращают расход памяти и затраты на компоновку.

Если присвоить свойству visible значение false, элемент не отрисовывается, но остаётся частью компоновки. Это компромиссный вариант для временного скрытия элемента, при котором сохраняются связанные с ним затраты памяти.

Более эффективный способ - установить style.display в DisplayStyle.None: и отрисовка, и обновление компоновки полностью прекращаются. Однако при повторном отображении элемента компоновку придётся пересчитать.

Редко появляющиеся элементы, например диалоговые окна и панели настроек, можно удалить из иерархии методом RemoveFromHierarchy, чтобы исключить постоянные накладные расходы. Учтите, что при повторном добавлении возникнет более заметный всплеск нагрузки, поскольку компоновку придётся построить заново.

Выбирайте способ с учётом частоты переключения элементов, балансируя краткосрочные требования к отрисовке и долгосрочную производительность.

Избыточная отрисовка

UI Toolkit отрисовывает элементы с прозрачностью, поэтому при их перекрытии может возникнуть значительная избыточная отрисовка: один пиксель обрабатывается несколько раз. При использовании универсального шейдера UI Toolkit это особенно затратно, поскольку каждый слой перекрывающихся элементов усложняет вычисления. Наложение нескольких прозрачных или полупрозрачных слоёв дополнительно снижает производительность.

Снизить влияние избыточной отрисовки на производительность помогут следующие приёмы:

- Чтобы полностью скрыть элемент, используйте style.display = DisplayStyle.None вместо style.opacity = 0, при котором прозрачный элемент всё равно отрисовывается.

- Не накладывайте без необходимости несколько элементов друг на друга: удаляйте или скрывайте те, которые полностью перекрыты.

- Для прокручиваемого содержимого реализуйте виртуализацию с помощью ListView. Этот элемент эффективно отрисовывает только видимые на экране строки.

- Также можно задать style.overflow = Overflow.Hidden, чтобы обрезать содержимое по границам заданной области и не отрисовывать невидимые фрагменты.

Управление памятью

Файлы USS и UXML содержат прямые ссылки на шрифты, текстуры и другие ресурсы. При загрузке таких файлов в память загружаются и все ресурсы по ссылкам, что может увеличить её расход. Ниже показаны ресурсы, на которые ссылается пример USS:

USS содержит ссылки на ресурсы.

После импорта эти ресурсы сразу занимают память, даже если не используются. Без грамотного управления это приводит к неэффективному расходу памяти.

Для оптимизации использования ресурсов применяйте следующие стратегии:

- Используйте AssetBundle или Addressables. По возможности загружайте только те документы интерфейса и таблицы стилей, которые нужны в текущей сцене или контексте. Это помогает контролировать расход памяти.

- Выгружайте ненужные ресурсы. Если элемент или документ интерфейса больше не используется, удалите его из иерархии методом RemoveFromHierarchy, а затем выгрузите через Addressables.Release или AssetBundle.Unload(true), освободив память для других операций.

- Загружайте части сложного интерфейса выборочно. Разделите крупные UXML- или USS-файлы на небольшие модульные шаблоны VisualTreeAsset и загружайте их динамически по мере необходимости. Если в память попадают только ресурсы видимых элементов, её расход остаётся низким.

Инструменты профилирования

Unity предоставляет несколько инструментов для поиска и устранения проблем с производительностью интерфейса.

Unity Profiler, UI Toolkit Debugger и Frame Debugger незаменимы при диагностике: с их помощью можно анализировать вызовы отрисовки, пакеты и дорогостоящие операции - перерасчёт компоновки, обновление стилей и изменение буфера вершин.

Для более подробного отслеживания изменений интерфейса используйте метод SetPanelChangeReceiver из Panel Settings. Он позволяет получать уведомления об изменениях и определять их источник. Метод доступен только в редакторе и отладочных сборках, но помогает выявить конкретное поведение интерфейса, вызывающее замедление.

Ниже приведён скрипт, который записывает в журнал каждое изменение интерфейса:

```
using UnityEngine;
using UnityEngine.UIElements;

public class PanelChangeReceiver : MonoBehaviour, IDebugPanelChangeReceiver
{
   [SerializeField] PanelSettings m_PanelSettings;

   void Awake()
   {
       m_PanelSettings.SetPanelChangeReceiver(this);
   }

   void OnDestroy()
   {
       m_PanelSettings.SetPanelChangeReceiver(null);
   }
```

```
    public void OnVisualElementChange(VisualElement element, VersionChangeType changeType)
    {
        Debug.Log($"{element.name} {changeType}");
    }
}
```

Добавьте этот компонент к GameObject и задайте PanelSettings в Inspector. Метод OnVisualElementChange срабатывает при любом изменении визуального элемента, например его компоновки, стиля или преобразования, и выводит сообщение в консоль. Так можно понять, какая часть интерфейса изменяется в данный момент.

Улучшения производительности в Unity 6

В Unity 6 реализован широкий набор оптимизаций, обеспечивающих плавную и отзывчивую работу как в редакторе, так и во время выполнения:

- Распространение событий. Правила распространения событий упрощены: теперь их легче понять, а обработка выполняется вдвое быстрее.

- Улучшенная генерация сеток. Геометрия стандартных элементов теперь создаётся в системе заданий, векторный API переведён на нативную реализацию, а генерация текста распараллелена.

- API пользовательской геометрии. Новый общедоступный API позволяет разработчикам создавать собственную геометрию с тем же уровнем производительности и строить высокооптимизированные компоненты интерфейса.

- Производительность компоновки в глубоких иерархиях. Улучшенное кэширование вычислений компоновки заметно ускоряет работу глубоких иерархий и делает взаимодействие плавнее.

- Оптимизированный TreeView для больших наборов данных. Элемент TreeView, ранее неэффективный на больших объёмах данных, получил новый высокопроизводительный внутренний механизм, специально предназначенный для Entities.

Дополнительные материалы по оптимизации производительности

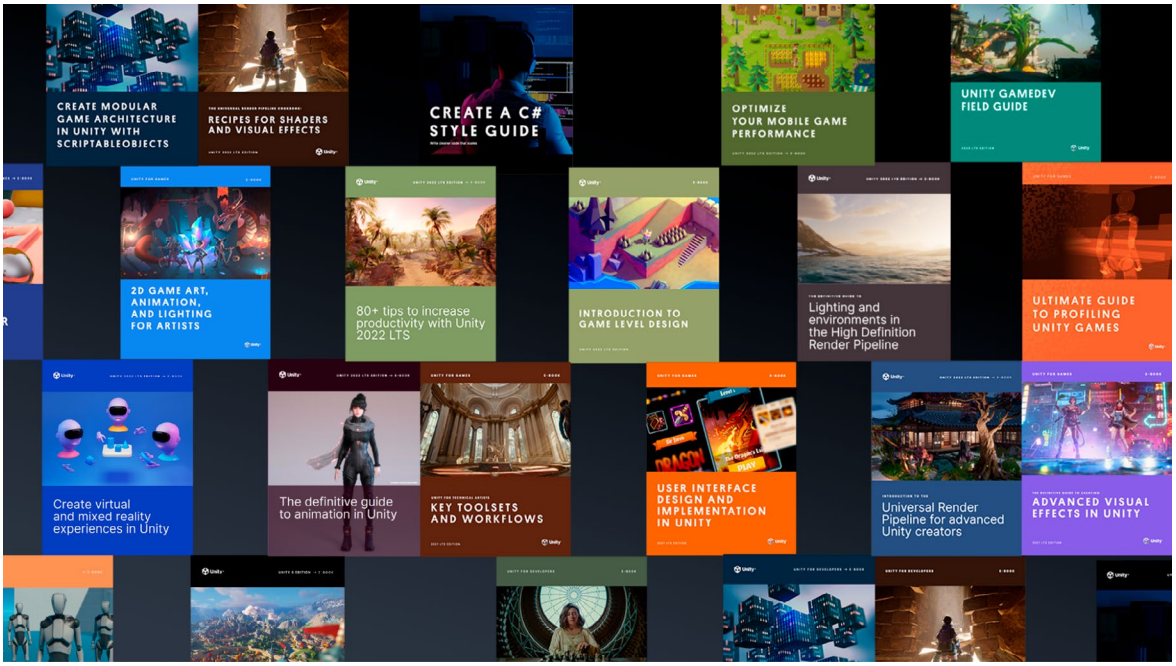
- Unite 2024: «Максимальная производительность с UI Toolkit».

- Электронная книга: «Полное руководство по профилированию игр Unity».

- Электронная книга: «Оптимизация производительности игр для мобильных устройств, XR и веб-платформ в Unity».

- Электронная книга: «Оптимизация производительности игр для консолей и ПК в Unity».

Ресурсы для опытных разработчиков и художников



В центре рекомендаций Unity можно скачать ещё больше электронных книг для опытных разработчиков и авторов контента. На выбор доступно свыше 30 руководств от отраслевых экспертов, инженеров и технических художников Unity с рекомендациями по эффективной работе с инструментами и системами Unity.

Советы, передовые практики и новости также публикуются в Unity Blog, Unity Discussions, Unity Learn и по хештегу #unitytips.


Unity®