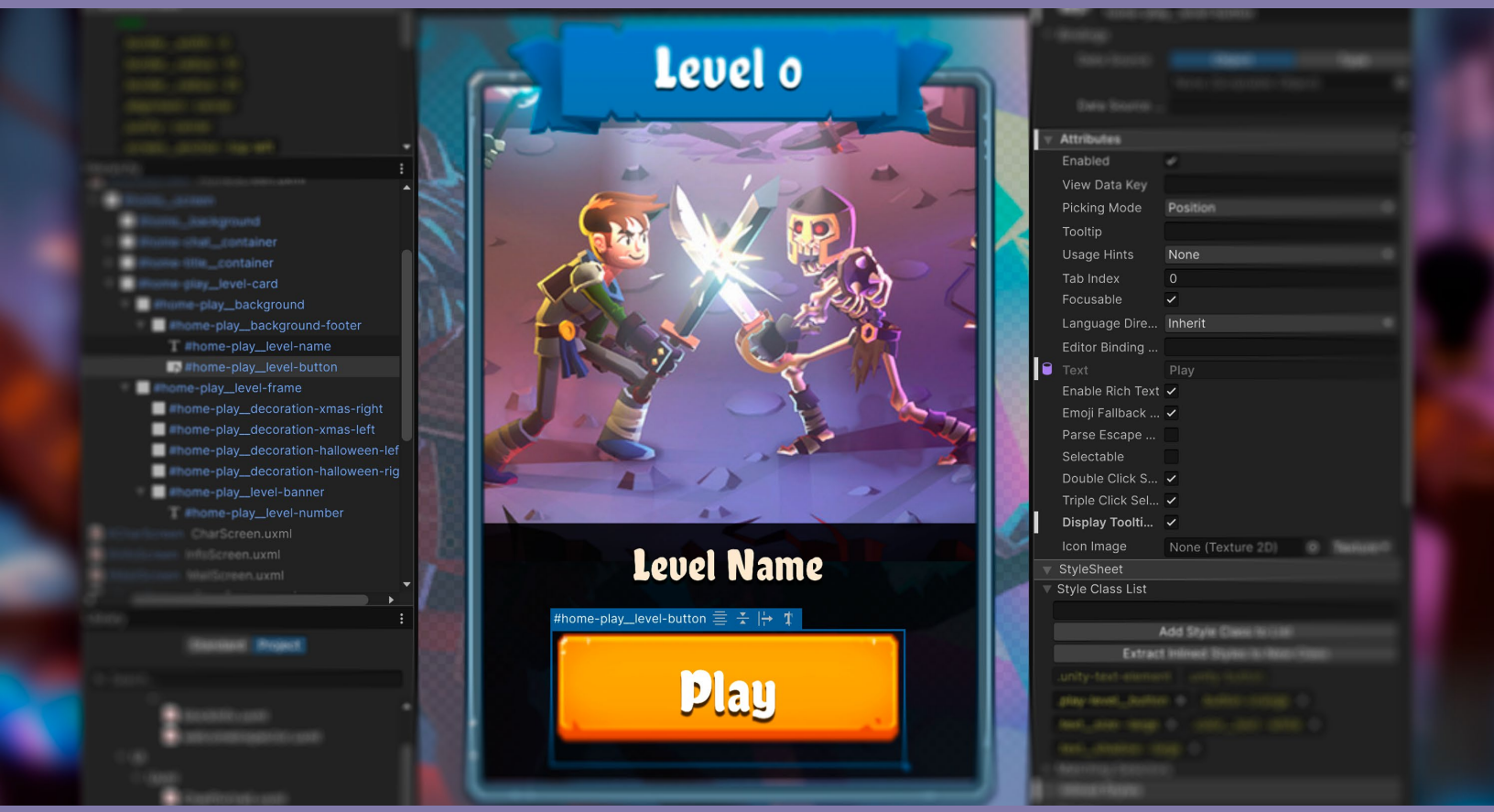
전 자 책

고급 Unity 개발자를 위한 UI 툴킷

(Unity 6 에디션)



Unity®

목차

들어가는 말

최고의 사용자 인터페이스는 바로 '눈에 띄지 않는' 사용자 인터페이스입니다.

UI(사용자 인터페이스)는 어떤 게임에서든 가장 중요한 요소입니다. 잘 만들어진 UI는 눈에 띄지 않고 애플리케이션에 자연스럽게 어우러집니다. 하지만 잘못 만들어진 UI는 사용자를 방해하고 게임플레이 경험을 저해합니다.

탄탄한 UI는 게임의 시각적 정체성을 강화해 줍니다. 오늘날의 게이머들은 애플리케이션에 자연스럽게 통합되면서도 세련되고 직관적인 UI를 원합니다. 플레이어는 인터페이스라는 관문을 통해 캐릭터의 주요 스탯이나 게임 월드의 경제 관련 정보를 비롯한 핵심 정보에 접근할 수 있습니다.

UI가 더 정교해지면서 UI 구현에 필요한 아트의 수준도 높아지고 있습니다. UI 디자인을 결정하는 전문가는 주로 두 가지 유형입니다.

UI 아티스트: 디자인, 컬러, 셰이프, 타이포그래피, 레이아웃의 기본 사항을 전담합니다. UI 아티스트는 게임 월드의 타겟 플레이어를 위한 디자인을 제작하며, 디테일을 중시하기 때문에 '픽셀 단위의 완벽성을 갖춘' UI를 구현합니다.

UX 디자이너: 사용자 동작과 최종 사용자의 폭넓은 요구 사항을 조사합니다. UX 디자이너는 사용자가 디지털 제품과 상호 작용하는 방식을 제어하며, 최대한 직관적이고 재미있는 경험을 구현한다는 목표로 내비게이션 플로를 구축합니다.

UX 디자이너는 다른 2D 또는 3D 아티스트 및 디자이너와 긴밀하게 협력하며, 이러한 협업을 통해 강력하고 효과적인 UI가 탄생합니다.

또 다른 핵심적인 역할인 **UI 프로그래머**는 앞에서 언급한 유형의 전문가들과 긴밀하게 협력합니다. 조직에서 사용하는 기술 스택을 바탕으로 모든 UI 디자인을 기능적인 인터페이스에 적용하고, 게임플레이 코드를 UI에 연결하고, 다시 UI를 통해 게임 시스템에 데이터를 제공하는 프로세스 또는 파이프라인을 확립합니다.

이전에 공개된 Unity의 사용자 인터페이스 디자인 및 구현 전자책에서는 UI 아티스트와 디자이너들이 Unity의 두 가지 UI 시스템, 즉 기존의 게임 오브젝트 기반 시스템인 Unity UI와 최신 UI 툴킷을 사용하여 Unity에서 인터페이스를 제작하는 방법을 살펴보았습니다. 게임 스튜디오가 UI를 처음부터 새로 디자인하고 아트워크를 게임으로 임포트하는 방법도 알아보았습니다. 해당 가이드는 Unity 2021 LTS를 기반으로 작성되었습니다.

새로 선보이는 이번 전자책에서는 웹 기반의 워크플로를 사용하여 최대한의 성능과 재사용성을 제공할 수 있도록 구축된 Unity 6의 UI 툴킷을 중점적으로 살펴봅니다. 본 전자책은 웹 개발 경험이 있는 UI 디자이너라면 직관적으로 이해할 수 있을 것이며, UI 프로그래머에게는 게임 제작에 UI 툴킷을 사용하는 방법에 관해 더 명확한 설명을 제시할 것입니다. 본 전자책은 모듈식으로 구성되어 있어 섹션을 어떤 순서로 읽어도 이해하는 데 문제가 없으며, UI 툴킷 학습 시 참고 자료로 유용하게 활용할 수 있습니다.

그럼 이제 시작하겠습니다.

주요 저자 및 도움을 주신 분들

본 가이드와 두 개의 UI 툴킷 샘플을 만든 주요 저자는 3D 및 시각 효과 아티스트, 개발자 겸 교육자인 윌머 린입니다.

또한 유니티의 시니어 콘텐츠 마케팅 매니저이자 그래픽 디자이너인 에두아르도 오리즈가 본 가이드 작성에 큰 도움을 제공하고 샘플 UI 툴킷인 드래곤 크래셔(Dragon Crashers)를 공유해 주었습니다.

유니티의 시니어 콘텐츠 마케팅 매니저인 토마스 크로그 야콥센도 본 가이드 작성에 큰 도움을 주고 샘플 QuizU를 제공하였습니다.

도움을 주신 유니티 직원분들

카밀 부지디, 소프트웨어 디벨로퍼

마틴 코테, 시니어 그래픽 디벨로퍼

휴고 부레 데마레, 시니어 소프트웨어 디벨로퍼

브누아 뒤푸아, 시니어 테크니컬 프로덕트 매니저

칼 존스, 시니어 소프트웨어 엔지니어

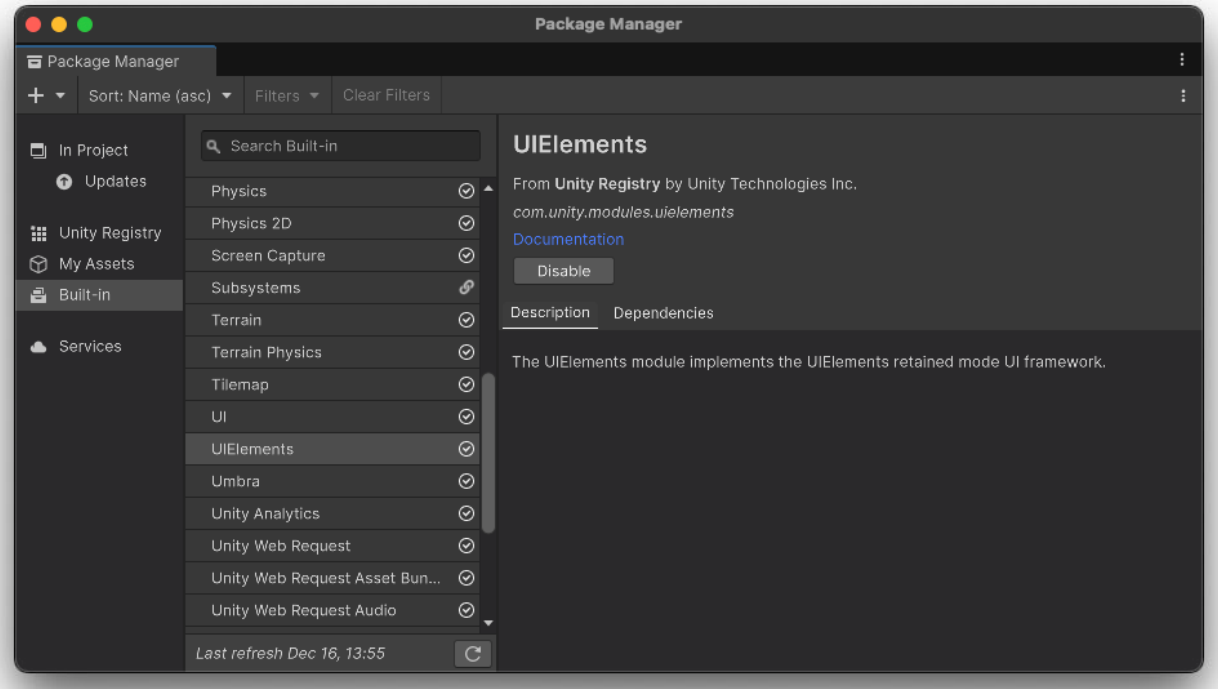
앙투안 라소제, 스태프 소프트웨어 디벨로퍼

마틴 파라디스, 스태프 소프트웨어 디벨로퍼

스테파니 발로로소, 매니저, 프로덕트 디자이너

UI 툴킷 및 샘플 프로젝트 설치

UI 툴킷은 Unity 6 코어 플랫폼에 통합되어 있습니다. 따라서 Unity 6 및 이후 버전에서 사용하기 위해 별도의 패키지를 설치할 필요가 없습니다. 제공되는 템플릿으로 새로운 프로젝트를 시작하는 것만으로도 본 가이드의 내용을 충분히 따라할 수 있습니다.



UIElements는 UI 툴킷, UI 빌더 및 해당 기능의 네임스페이스이며, 이제 Unity 6에 이러한 모든 요소가 포함됩니다.

공식 UI 툴킷 샘플

본 전자책에서는 주로 다음과 같은 샘플을 사용하여 Unity 프로젝트의 UI 툴킷 기능을 제시하고 설명합니다. 각 샘플은 Unity 에셋 스토어에서 무료로 다운로드할 수 있습니다.

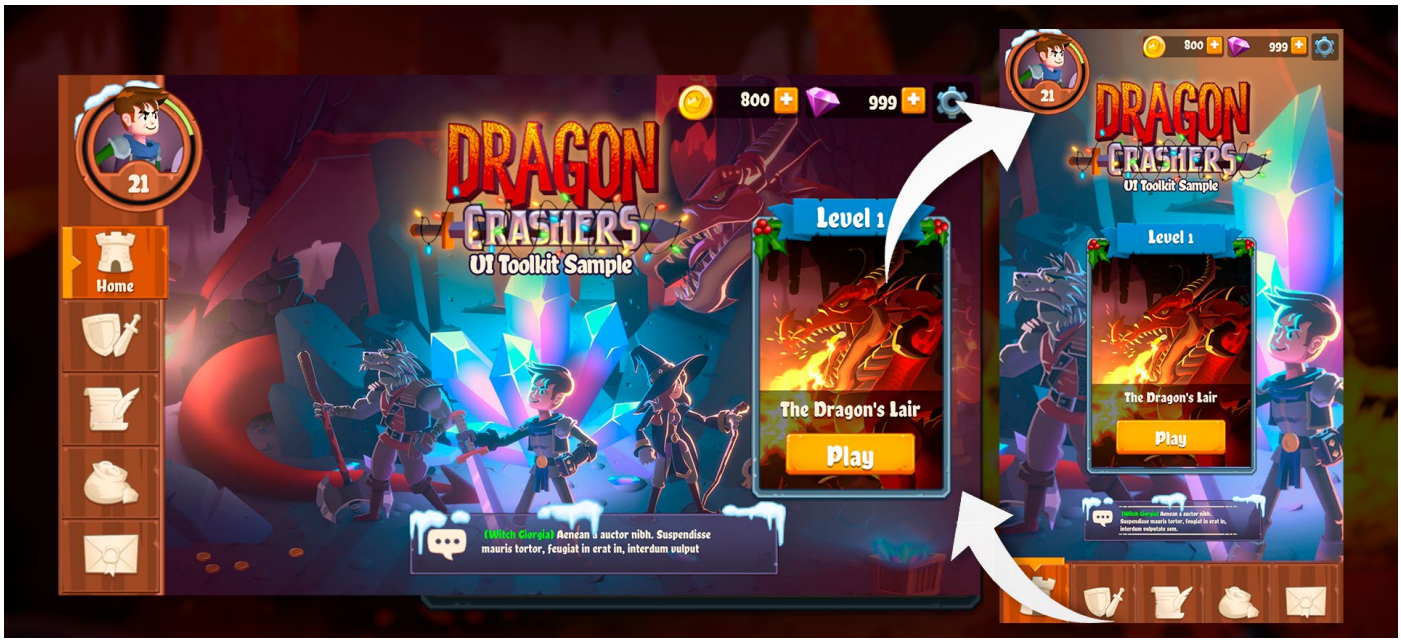
UI 툴킷 샘플 – 드래곤 크래셔

이 데모는 런타임에 최신 UI 툴킷 워크플로를 사용하여, 프론트엔드 메뉴 시스템을 포함한 모든 기능을 갖춘 인터페이스가 적용된 상태로 미니 RPG 게임인 2D 프로젝트 드래곤 크래셔의 일부를 구동합니다.

이 데모는 초보자용이 아닙니다. UI 구조를 들여다 보고, 데모를 진행하며 구체적인 구현 내용을 살펴볼 수 있는 숙련된 Unity 개발자를 대상으로 제작된 데모입니다. 이 데모는 Unity 2021 LTS용으로 최초 출시되었으며, 이후 Unity 6로 업데이트되었습니다. 이 데모를 통해 살펴볼 수 있는 주제는 다음과 같습니다.

- 프로젝트 구조와 명명 규칙
- 테마를 사용하여 UI 배리에이션을 만들고 세로 및 가로 방향 지원을 추가하는 방법
- 탭 메뉴, 인벤토리, 메시지, 커스텀 컨트롤과 같은 복잡한 요소
- 모바일 화면의 안전 표시 영역 내에 콘텐츠를 표시하도록 **SafeAreaAPI**를 사용하는 방법
- 다중 언어 지원에 필요한 현지화 사용 방법
- 데이터와 UI 컴포넌트의 동기화를 간소화하는 데이터 바인딩
- 일반적인 캐주얼 게임 인터페이스를 구현하는 방법 예시

샘플을 소개하는 동영상을 시청하고 에셋 스토어에서 샘플을 다운로드할 수 있습니다.

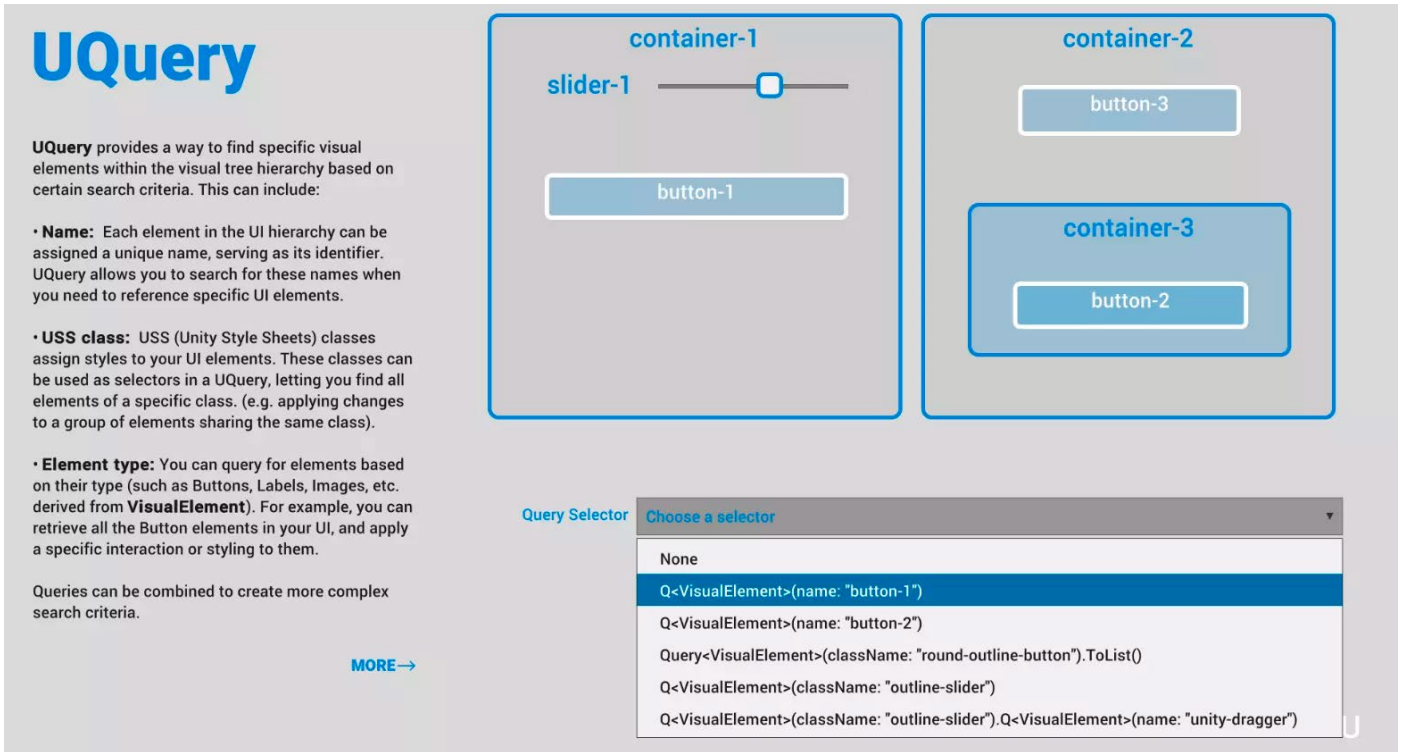


홈 화면은 가로 방향 또는 세로 방향으로 표시할 수 있습니다.

QuizU

QuizU 데모는 Unity의 UI 툴킷으로 제작된 인터랙티브 퀴즈 게임입니다. UI 개발자를 대상으로 하는 이 프로젝트는 최신 게임 사용자 인터페이스를 제작하는 데 필요한 UI 툴킷 워크플로, 이벤트 기반 아키텍처, 재사용 가능한 디자인 패턴을 소개합니다. 이 데모를 통해 다음과 같은 내용을 살펴볼 수 있습니다.

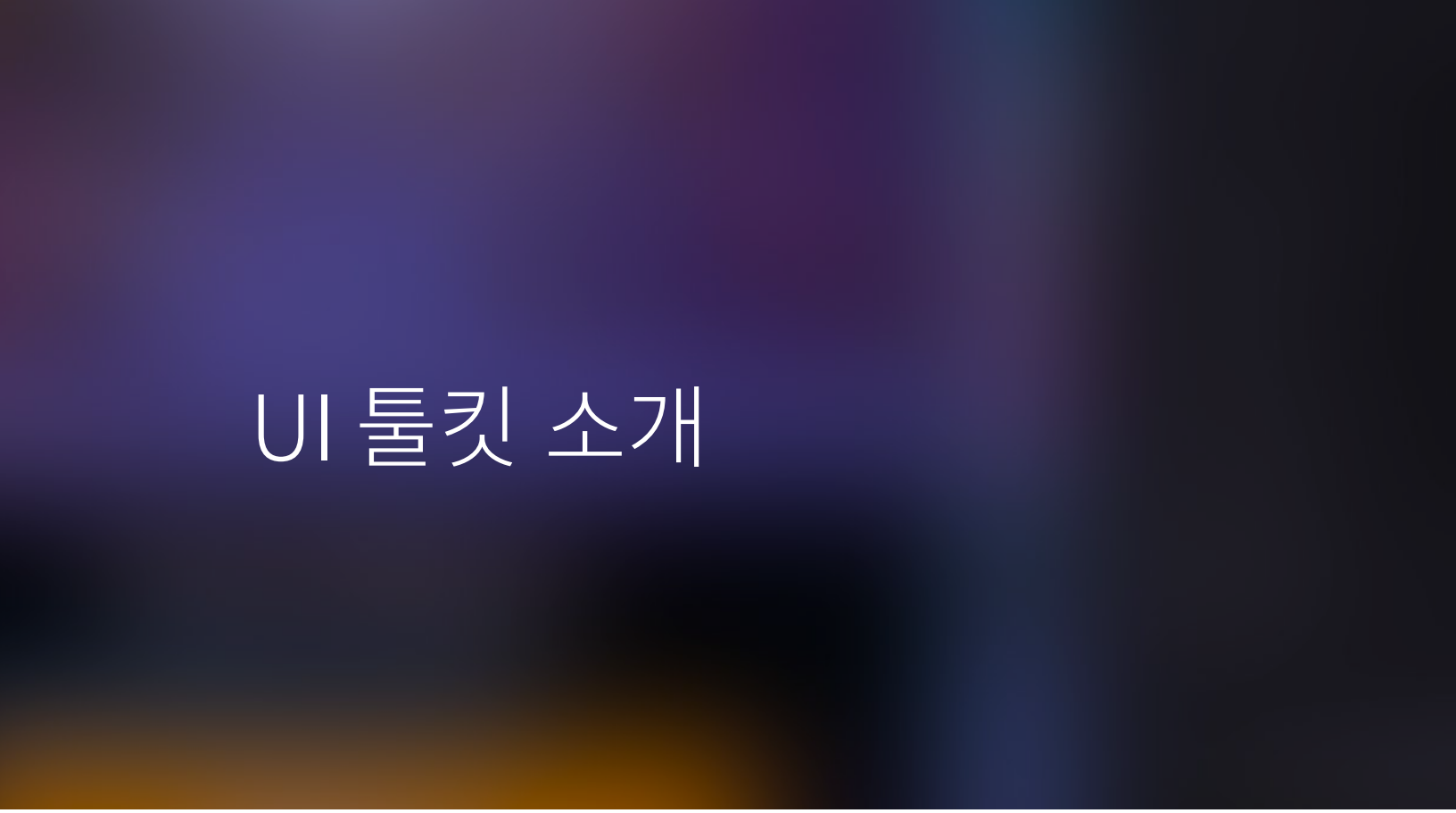
- UXML 파일과 중첩된 시각적 트리를 사용하여 효율적으로 UI 요소를 구성합니다.
- 인터랙티브 요소가 사용자 입력에 반응하도록 USS 선택자와 유사 클래스를 사용하여 스타일 규칙을 적용합니다.
- 반응형 UI 동작을 위해 FlexBox 기능을 사용하여 Flex 기반 레이아웃을 구현합니다.
- 선택자를 사용하여 동적으로 UI 요소를 쿼리하고 수정합니다.
- 재사용 가능한 클래스에서 이벤트 처리를 캡슐화하여 드래그, 멀티 터치 제스처와 같은 커스텀 상호 작용을 구현합니다.
- 이벤트 디스패치를 사용하여 이벤트를 단계별로 처리하고 이벤트가 전파되는 방식을 관리합니다.
- USS 전환을 사용하여 Duration, Easing과 같은 프로퍼티로 UI 요소에 매끄러운 애니메이션과 효과를 추가합니다.



QuizU UI 툴킷 데모의 스크린샷

QuizU는 Unity 2022 LTS용으로 최초 출시되었으며, Unity 6의 새로운 기능으로 업데이트되었습니다. 이 프로젝트는 이제 커스텀 컨트롤 제작, 데이터 바인딩 설정, 현지화 구현 등의 방법을 소개하는 데모를 포함합니다.

에셋 스토어에서 이 프로젝트를 다운로드할 수 있습니다.



UI 툴킷 소개

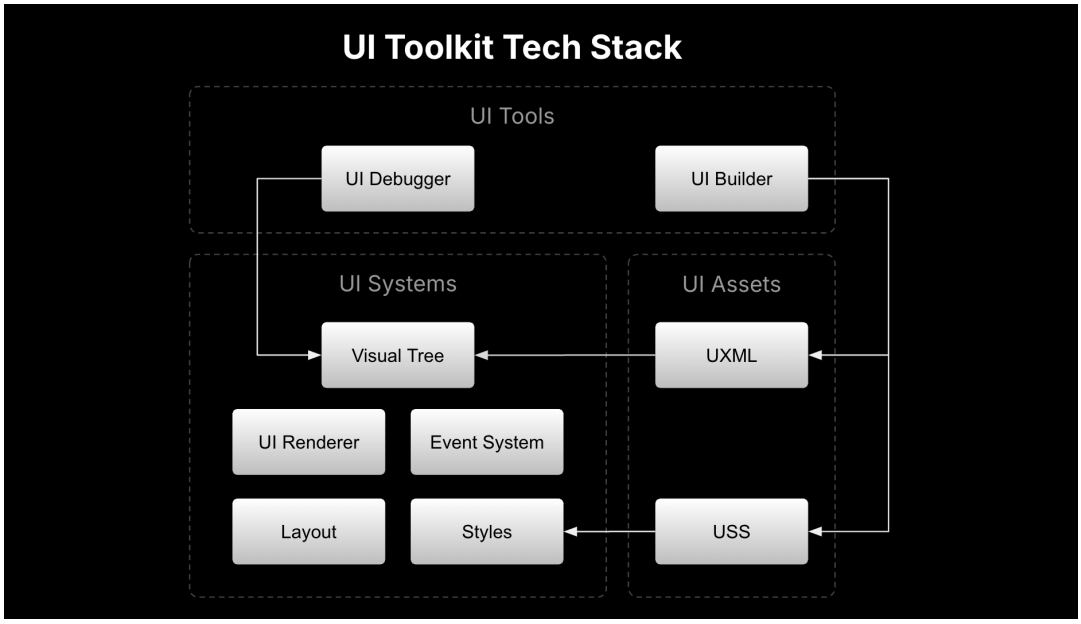
UI 툴킷은 기존 Unity UI(또는 uGUI) 및 레거시 IMGUI(에디터 툴용) 시스템에 비해 상당한 이점을 제공합니다. UI 툴킷은 대부분의 프로젝트에서 더 효율적으로 규모를 조절할 수 있는 유연한 성능 중심의 최신 솔루션입니다. 또한 제작 파이프라인 전체를 지원하므로, 에디터 툴과 런타임 게임 또는 애플리케이션을 모두 처리할 수 있습니다.

레거시 UI 시스템에 비해 UI 툴킷의 장점은 다음과 같습니다.

- **더 빠른 반복 작업:** 글로벌 스타일 관리와 실시간 저작 기능으로 더 빠르게 작업하고 반복 수정(iteration)할 수 있습니다.
- **렌더링 성능:** 렌더링 힌트와 동적 텍스처 아틀라스를 사용하여 게임의 성능을 더욱 효과적으로 관리할 수 있습니다.
- **협업 향상:** 로직(C# 코드), UI 구조(UXML(Unity XML) 문서), 스타일(USS(Unity 스타일 시트) 사용)을 서로 분리해 충돌을 줄이고 팀 작업 효율을 높일 수 있습니다.
- **재사용성:** 프로젝트 내, 여러 프로젝트 사이, 그리고 에디터와 런타임 사이에 스타일과 위젯을 공유하고 재사용할 수 있습니다.

UI 툴킷은 웹 기술을 기반으로 만들어졌기 때문에 웹 애플리케이션에 익숙한 개발자들이 손쉽게 사용할 수 있습니다. HTML/XML, CSS(캐스케이딩 스타일 시트)와 같은 마크업 언어에 익숙하지 않은 개발자에게는

업계 표준 툴의 강력한 기능을 살펴보는 좋은 기회가 될 수 있습니다.

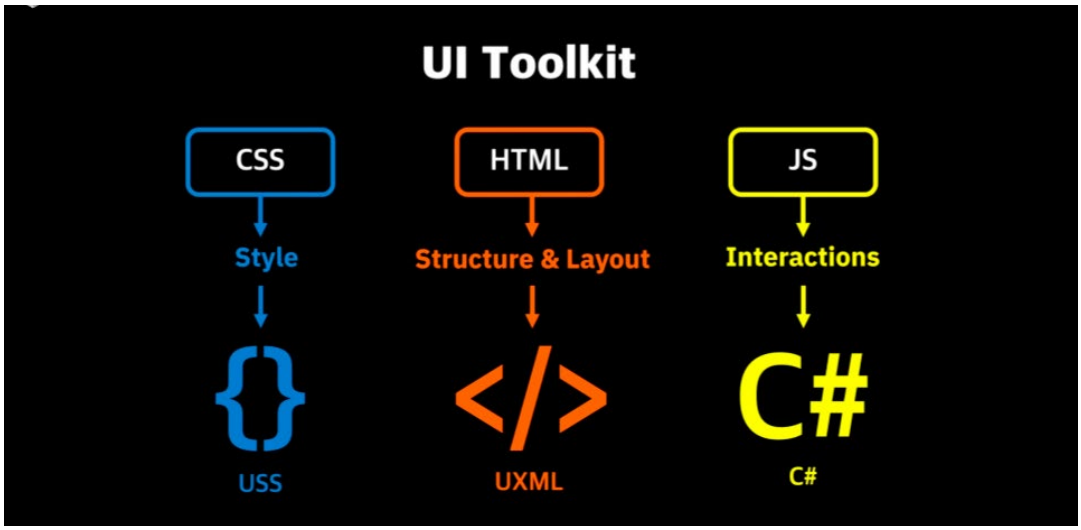


본질적으로 UI 툴킷 인터페이스는 UI 툴킷 시스템으로 레이아웃과 스타일을 생성하기 위한 UXML 파일과 USS 파일로 구성됩니다.

UI 에셋

UI 제작의 기본 구성 요소인 UI 에셋은 UXML 파일과 USS 파일로 구성됩니다. UXML(Unity XML)은 UI의 내용과 구조를 나타내며, HTML이나 XML과 같은 마크업 언어와 유사합니다.

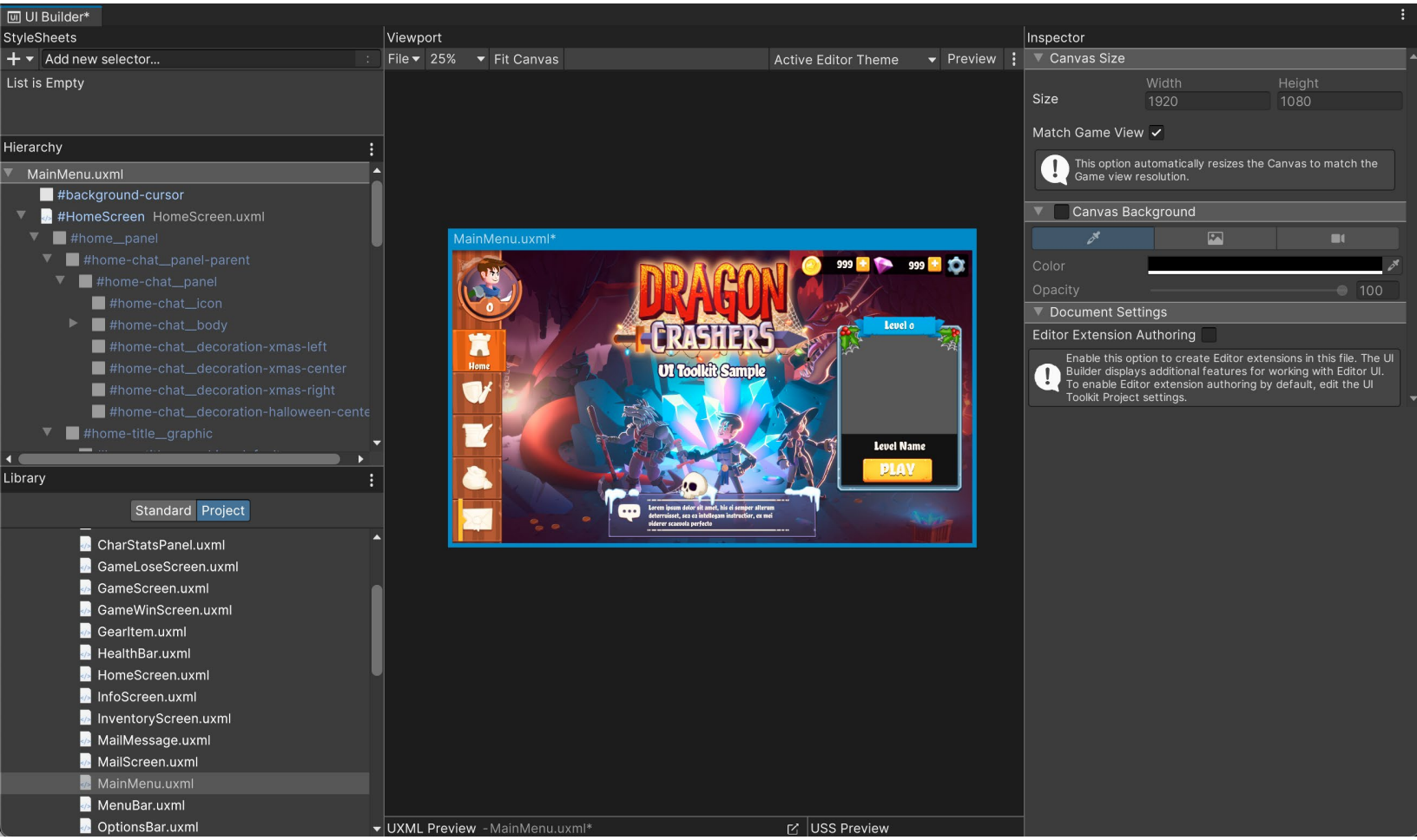
CSS(캐스케이딩 스타일 시트) 기반으로 개발된 USS는 UI 콘텐츠의 외형과 스타일을 정의하는 데 사용됩니다. 이러한 UXML과 USS는 본 가이드 전반에서 사용됩니다.



UI 툴킷과 웹 기술의 유사점

UI 빌더

UI 에셋은 선호하는 IDE를 사용하여 코드로 작성할 수도 있고, UI 툴킷의 일부인 UI 빌더를 사용하여 시각적으로 작성할 수도 있습니다. 아티스트와 디자이너는 UI 빌더 인터페이스를 사용하여 제작 중인 UI를 편집하고 시각화할 수 있습니다.



그래픽 및 폰트 에셋 준비

UI 디자인은 대부분 Unity 외부에 있는 DCC(디지털 콘텐츠 제작) 애플리케이션에서 이루어집니다. UI 아티스트의 스타일과 선호도에 따라 UI 디자인 작업은 Adobe Photoshop과 같은 래스터 드로잉 애플리케이션이나 벡터 기반 툴에서 진행됩니다. 일반적으로 모든 UI 그래픽 요소는 PNG처럼 투명도가 있는 무손실 비트맵 이미지로 익스포트되며, 런타임 효율을 위해 다른 UI 요소들과 함께 하나의 텍스처 아틀라스에 결합됩니다.

벡터 기반 DCC 애플리케이션을 사용한다면, 벡터 그래픽스를 래스터 포맷으로 익스포트해야 UI 툴킷에서 작업할 수 있습니다. 자세한 내용은 벡터 이미지 섹션을 참조하세요.

비트맵 이미지

Unity는 PNG, BMP, TIF, TGA, JPG, PSD 등 가장 일반적인 이미지 파일 포맷을 지원합니다. 이러한 포맷의 파일을 Assets 폴더에 추가하면 Unity는 해당 파일을 텍스처 2D 에셋(3D 프로젝트) 또는 스프라이트(2D 프로젝트) 포맷으로 임포트합니다. 임포트된 파일의 포맷은 인스펙터(Inspector)의 Texture Type 필드에서 변경할 수 있습니다. UI 툴킷은 이러한 두 가지 포맷의 UI 비트맵 그래픽스를 모두 지원합니다.

텍스처에는 이미지의 크기와 포맷 외에는 별다른 정보가 포함되지 않지만, 스프라이트에는 UI 툴킷에서 사용되는 추가 프로퍼티가 있습니다.

스프라이트

스프라이트는 2D 게임 개발에 필요한 텍스처로, Sprite Renderer 컴포넌트에서 사용됩니다. Unity의 2D Sprite는 타일화, 리깅, 스키닝을 거쳐 애니메이션에 활용할 수 있고, 커스텀 지오메트리를 적용하거나 2D 조명에 필요한 추가 맵을 포함할 수 있습니다. 이 섹션에서는 UI 툴킷과 관련된 설정만 살펴봅니다. 2D 그래픽스에 대해 자세히 알아보려면 https://unity.com/kr/resources 페이지에 공개될 예정인 Unity 6의 2D 아트, 애니메이션 및 조명 전자책을 참조하세요.

대부분의 UI 그래픽 에셋은 Unity의 월드 스케일(1유닛이 3D 공간의 1세제곱미터를 나타냄)을 따르는 대신 스크린 공간에 렌더링됩니다. UI 툴킷은 이러한 그래픽스의 스케일을 관리하지만, 스프라이트의 **PPU(단위당 픽셀)**는 UI에서 스프라이트의 크기에 영향을 줍니다. 예를 들어 스프라이트의 해상도가 그리드 단위당 128 픽셀이어야 한다면 PPU를 128로 설정하세요.

스프라이트 에디터는 그래픽스 수정에 필요한 툴을 제공합니다(예: 파란색 핸들을 사용하여 자르기, 녹색 핸들을 사용하여 분할하기 등). 이러한 툴을 사용하여 그래픽을 타일화하거나, 스케일링 가능한 요소를 만드는 일반적인 방법인 9슬라이싱 기법을 사용할 수 있습니다.

스프라이트는 평평한 직사각형 3D 메시에 매핑된 2D 텍스처입니다. 임포트된 스프라이트에는 기본적으로 **Mesh Type**: **Tight** 설정이 적용됩니다. 이 설정은 스프라이트의 불투명한 픽셀 윤곽선을 그대로 따라가도록 메시를 조정합니다. 이렇게 하면 중첩된 투명한 영역을 표현하기 위해 GPU가 한 프레임 내에서 동일한 픽셀을 두 번 이상 그리는 경우 발생하는 오버드로우가 감소하여 성능이 향상됩니다. 스프라이트 에디터의 Outline 섹션에서 이 메시를 수동으로 조정하고 최적화할 수 있습니다.

Sprite Mode는 스프라이트 에셋의 인스펙터에서 선택할 수 있는 유용한 기능으로, 다음과 같은 모드를 제공합니다.

- **Single**: 기본 모드로, 이미지에 하나의 이미지 요소만 포함됩니다.
- **Multiple**: 텍스처 소스 파일에서 같은 이미지 안에 요소가 여러 개 있는 경우 이 값을 선택합니다. 그런 다음 이미지를 여러 개의 하위 에셋으로 분할할 방법을 Unity에서 인지할 수 있도록 스프라이트 에디터에서 요소의 위치를 정의합니다. 분할된 각 그래픽은 UI 툴킷에서 따로 사용할 수 있는 개별 스프라이트가 됩니다.
- **Polygon:** 원형 또는 정다각형 폴리곤 이미지에 가장 적합한 모드로, 이미지 형태와 가장 일치하는 윤곽선을 설정하여 깔끔한 윤곽선을 얻을 수 있습니다.

렌더 텍스처 에셋

렌더 텍스처는 텍스처에서 프레임마다 업데이트되는 카메라 뷰의 스냅샷입니다. 렌더 텍스처는 **Assets > Create > Rendering**을 통해 생성하고, Output 메뉴의 Camera 컴포넌트에서 참조할 수 있습니다. UI 툴킷에서 이 텍스처를 사용해 미니 맵, 캐릭터 선택 화면 또는 그 외에 UI에 통합해야 하는 게임 내 비주얼과 같은 요소를 표시할 수 있습니다.

UI 툴킷 샘플: 드래곤 크래셔에는 레벨 미터가 포함된 캐릭터 프리뷰, UI 버튼 위에 렌더링된 파티클 효과와 같은 렌더 텍스처 예시가 포함되어 있습니다.

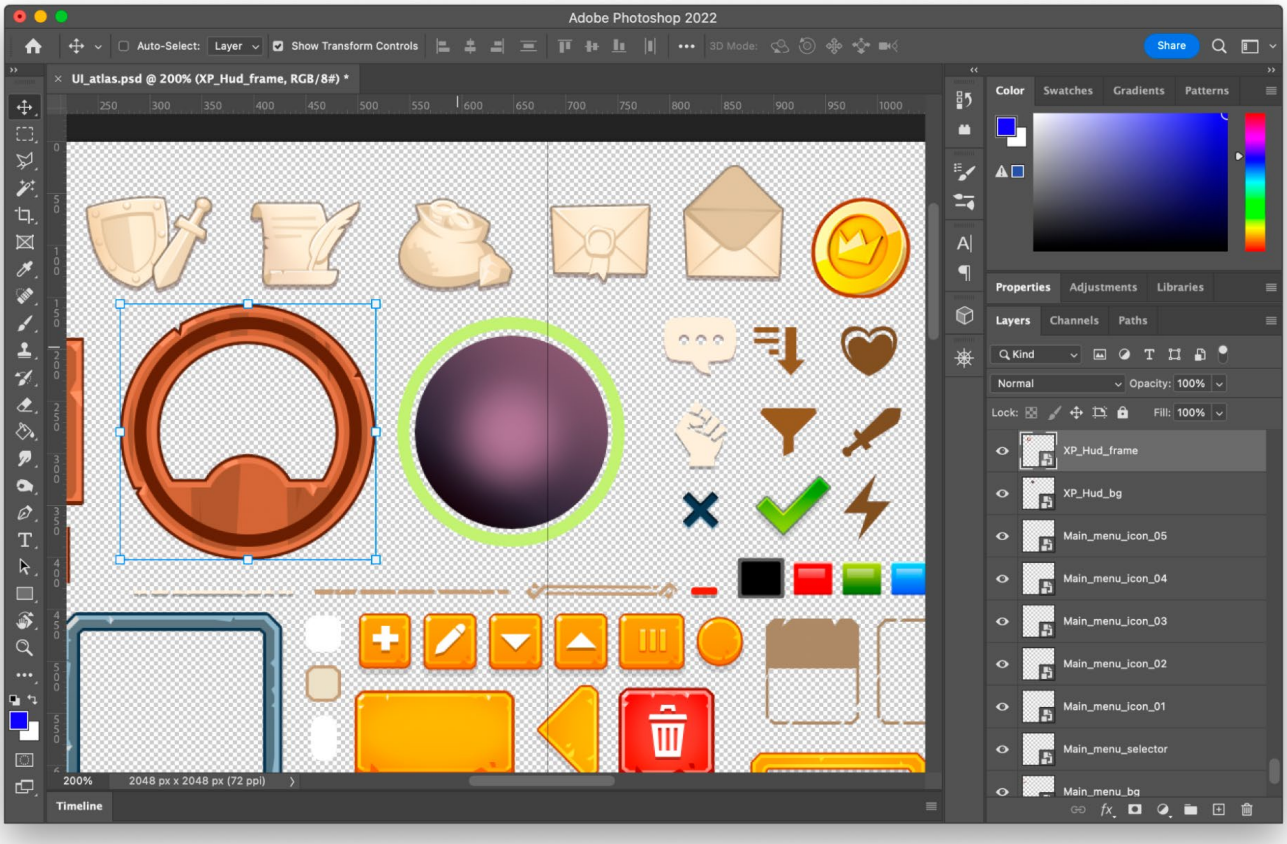
게임 요소 내에 UI를 표시하는 반대의 사용 사례도 가능합니다. 애플리케이션에서 UI 툴킷으로 만든 기능적인 인터페이스를 표시하는 3D 컴퓨터 모델을 예로 들겠습니다. UI 툴킷 인터페이스를 렌더 텍스처로 렌더링하고 Panel Settings 및 Camera에서 할당한 다음 3D 모델의 머티리얼에 적용할 수 있습니다.

렌더 텍스처 설정과 간단한 테스트

렌더 텍스처는 리소스를 많이 소모한다는 점을 기억해 두세요. 필요한 경우에만 사용하고 프로젝트의 특성을 파악하여 성능을 최적화해야 합니다. 활성화된 다른 게임플레이 요소가 없는 전체 화면 인터페이스에서는 이러한 방식으로 추가 효과를 적용해도 성능에 큰 문제가 발생하지 않습니다.

2D PSD Importer

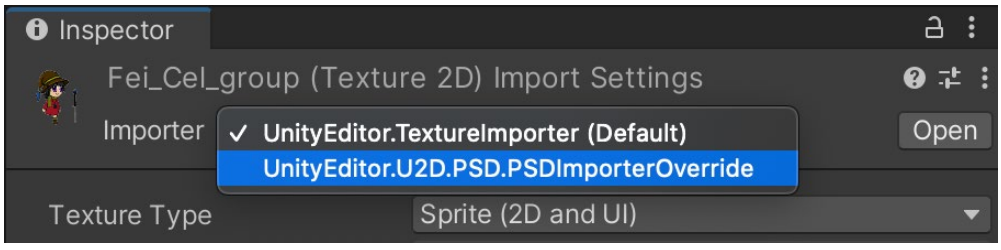
프로젝트에 2D PSD Importer 패키지가 설치되어 있지 않으면 Unity는 PSD(Adobe Photoshop 파일)를 평면 텍스처 포맷으로 임포트합니다. PSD 파일은 주로 하나의 파일에서 여러 개의 이미지를 레이어로 저장하는 용도로 사용됩니다. 대부분의 DCC 툴에서 이러한 포맷의 익스포트를 지원합니다.



Photoshop에서 UI 에셋 생성: 각 요소는 보통 자체 레이어 또는 그룹을 가지거나 스마트 오브젝트입니다. 스마트 오브젝트를 사용하면 각 요소를 개별적으로 작업할 수 있으며, 기본 문서에서 크기를 조절하더라도 요소의 원래 해상도를 유지할 수 있습니다.

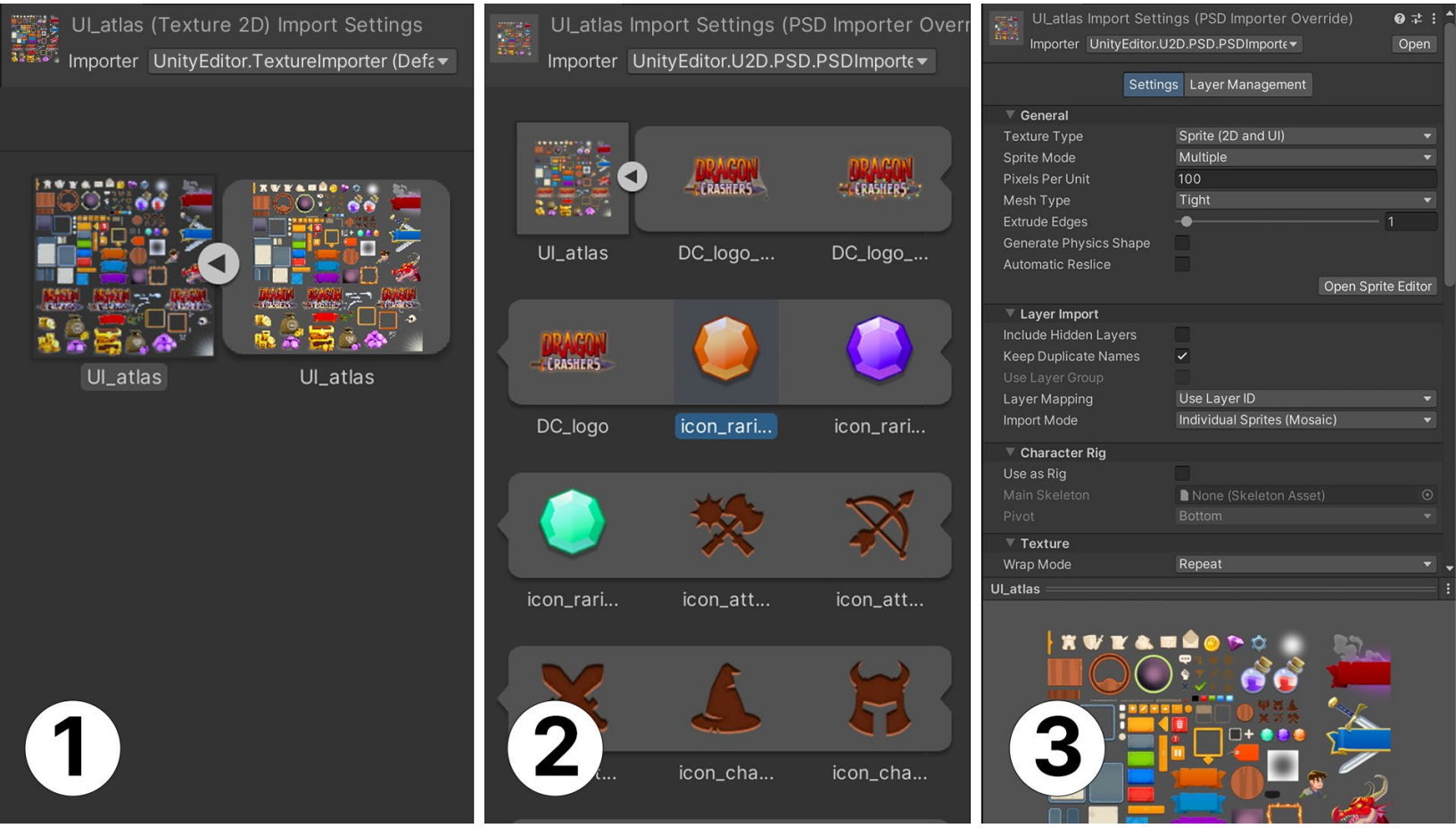
PSD 파일을 사용하면 각 레이어를 개별 파일로 익스포트한 후 변경이 필요할 때마다 이 프로세스를 반복하는 번거로움 없이 Unity에 바로 임포트할 수 있으므로 워크플로가 간소화됩니다.

패키지 관리자에서 2D PSD Importer 패키지를 설치하고, 인스펙터를 통해 PSD 파일을 임포트하세요.



인스펙터에서 PSD Importer를 선택하면 파일 처리 옵션을 볼 수 있습니다.

UI 에셋 작업을 진행할 때는 인스펙터에 있는 **Character Rig**에서 **Use as Rig** 옵션을 선택 해제해야 합니다. 2D 캐릭터의 골격 애니메이션과 관련된 설정으로, UI 요소에는 필요하지 않습니다. 숨겨진 레이어 폐기, 레이어별 오브젝트 그룹화 등 레이어 임포트와 관련된 옵션도 볼 수 있습니다.



PSD Importer로 전환하면 더 많은 임포트 옵션을 이용할 수 있습니다.

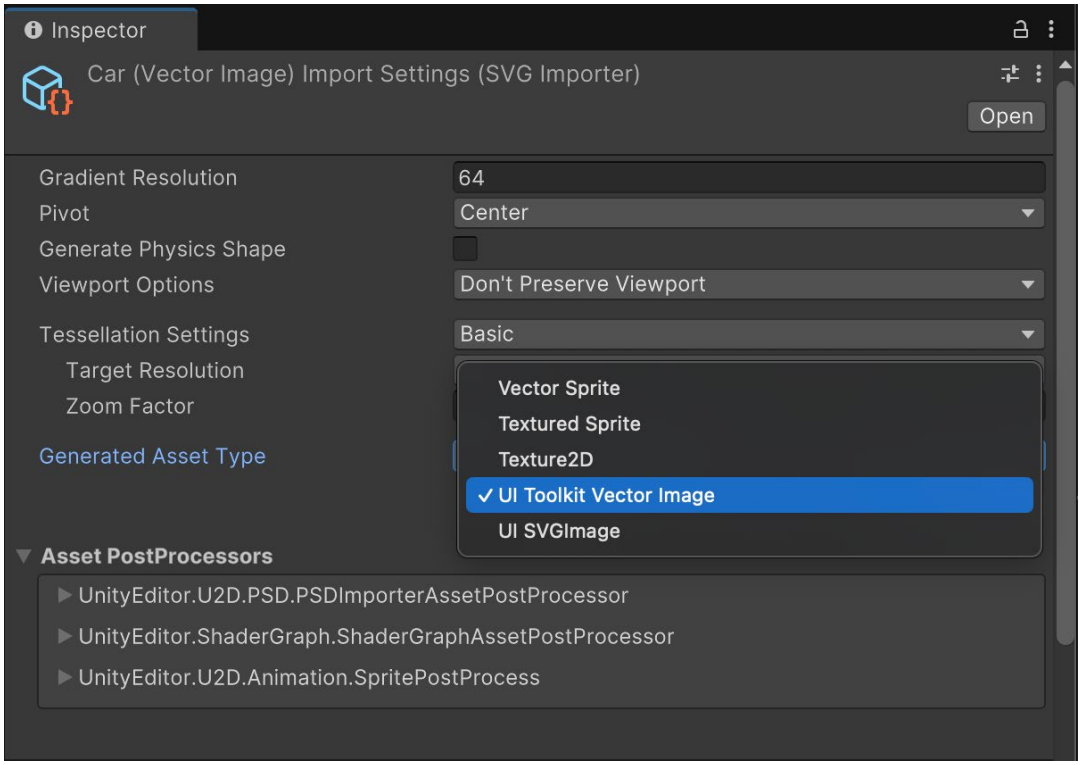
PSD에서 생성된 스프라이트는 프로젝트(Project) 뷰에서 일반 스프라이트로 사용할 수 있습니다. 일반 스프라이트 에셋과 마찬가지로 스프라이트 에디터에서 스프라이트를 분할하고, 윤곽선을 변경하고, PPU(단위당 픽셀)를 수정할 수 있습니다.

팁: 디자인 반복 수정
사용자가 PSD 파일을 저장할 때마다 Unity는 PSD 파일에 포함된 스프라이트를 자동으로 새로고침합니다. 따라서 간단한 플레이스홀더를 만들고 게임(Game) 뷰에서 변경 사항을 확인하면서 디자인을 반복 수정할 수 있습니다. 이렇게 하면 파일을 바꾸거나 다른 개발자 팀원의 지원을 받지 않고도 컨텍스트 내에서 작업물을 확인하며 시간을 단축하고 품질을 높일 수 있습니다.

벡터 이미지

벡터 포맷 지원은 본 가이드 작성 시점 기준으로 아직 개발 단계에 있지만, UI 빌더에서 배경 이미지 옵션으로 사용할 수 있습니다. 하지만 현재 UI 툴킷에는 래스터 이미지(스프라이트와 텍스처) 포맷이 권장됩니다.

이 기능을 테스트해 보려면 Vector Graphics 패키지가 필요합니다. 이 패키지는 아직 프리뷰 단계이며 패키지 관리자에서는 기본적으로 숨겨져 있습니다. 이 패키지를 설치하는 방법은 기술 자료를 참고하세요. 이 패키지에는 벡터 그래픽스를 폴리곤으로 전환할 때 테셀레이션 레벨을 정의하는 설정이 포함되어 있습니다. UI 툴킷에서 사용하려면 **Generated Asset Type** 설정으로 **UI Toolkit Vector Image**를 선택해야 합니다.



Vector Graphics 패키지를 사용하면 게임 또는 UI에서 SVG 이미지를 제한적으로 사용해 볼 수 있습니다.

현재 SVG 파일을 렌더링하면 폴리곤으로 테셀레이션되기 때문에 벡터 이미지가 갖는 이점이 제한됩니다. 스케일을 키우면 에지가 벡터 이미지에서 흔히 볼 수 있는 부드러운 곡선이 아니라 폴리곤으로 표현되는 것을 볼 수 있습니다. 본 가이드 작성 시점에는 UI 툴킷에 대해 안티앨리어싱이 활성화되어 있지 않습니다.

벡터 지원의 최종 버전에서는 별도의 Vector Graphics 패키지를 사용하지 않고도 기본적으로 실제 벡터 형태를 지원할 수 있을 것입니다.

폰트

UI 툴킷은 Font 및 FontAsset을 지원합니다.

- **Font:** 이전 버전과의 호환성을 위해 TTF, OTF와 같은 표준 폰트 포맷이 지원됩니다. 단, 백그라운드에서는 자동으로 FontAsset으로 전환됩니다.
- **FontAsset**: 권장되는 포맷이며, 이 포맷을 사용하면 원래의 폰트 에셋을 수정하지 않고 글자 간격, 기준선과 같은 항목을 세부적으로 조정할 수 있습니다. 이는 게임에서 흔히 사용되는 스타일이 적용된 폰트에 유용합니다.
  - FontAsset을 사용하면 문자 세트, 해상도, Atlas Population 옵션 등 아틀라스가 생성되는 방식을 정밀하게 제어할 수 있습니다. 이러한 설정은 특히 문자 수가 많은 언어를 지원하는 유니코드 폰트로 작업할 때 메모리 사용량을 줄이는 데 도움이 됩니다.

텍스처 패커

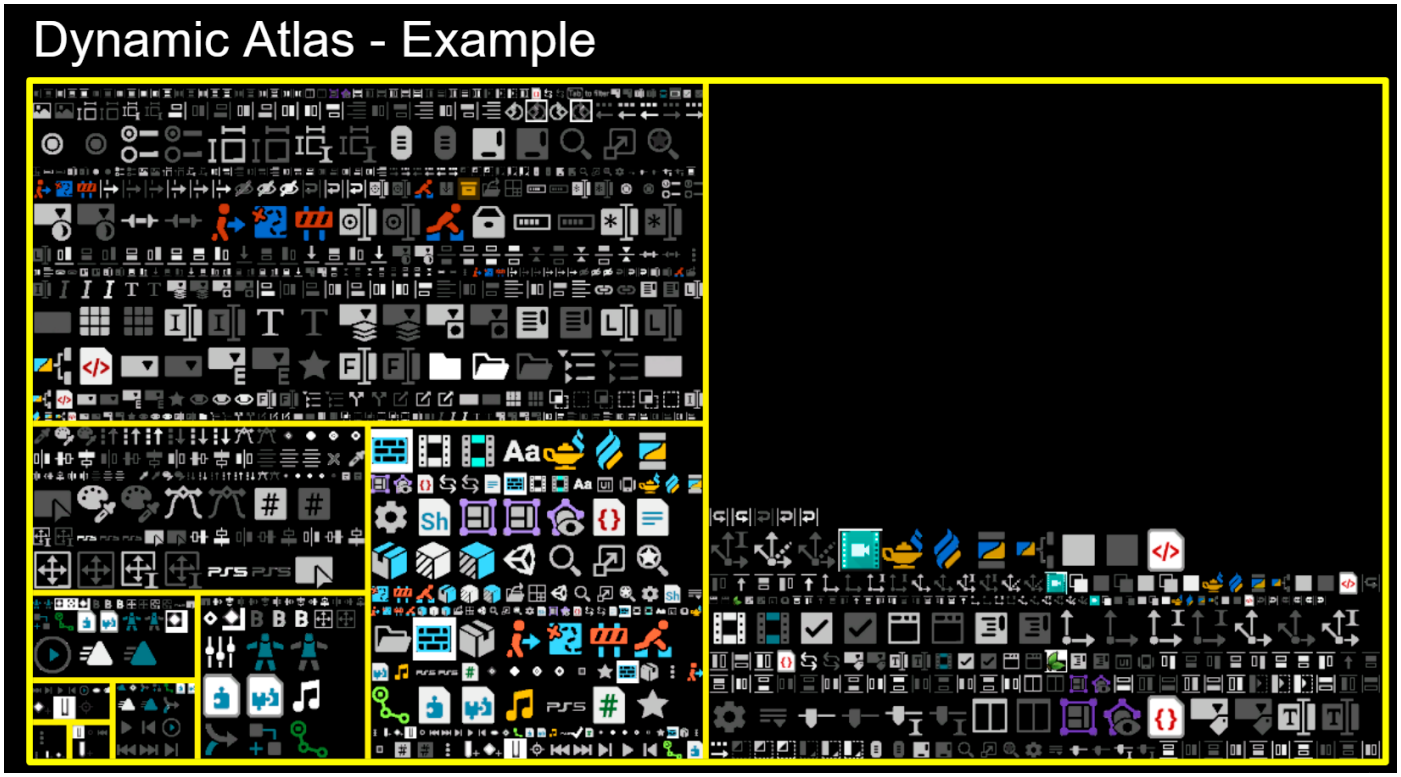
여러 개의 2D 그래픽스를 하나의 텍스처로 결합하는 것은 드로우 콜을 줄이고 메모리 사용량을 개선하기 위해 흔히 사용되는 최적화 기법입니다. UI 툴킷은 두 가지 최신 아틀라스 시스템을 지원합니다.

스프라이트 아틀라스

UI 툴킷 샘플 – 드래곤 크래셔의 일반적인 게임 UI 아틀라스

스프라이트 아틀라스는 2D 게임 개발 및 스프라이트를 위한 Unity 아틀라스 툴이지만, UI 그래픽스 용도로 사용할 수도 있습니다. 스프라이트 아틀라스는 여러 개의 에셋을 자동으로 같은 프로젝트 폴더에 패킹하여 스프라이트용 아틀라스와 노멀 맵 및 마스크 맵을 생성합니다. 또한 플랫폼별 배리언트를 지원하며, 고급 제어에 필요한 API를 제공합니다. 스프라이트 아틀라스는 에디터에서 에셋을 패킹하는 용도로 흔히 사용되지만 런타임에는 작동하지 않습니다.

동적 아틀라스



동적 아틀라스는 Unity 에디터에서 생성되고 Texture Atlas Viewer에 표시되며, 허용되는 최대 텍스처 크기까지 가로 및 세로 방향으로 2의 배수 단위로 커집니다.

스프라이트 아틀라스로 패킹되지 않은 UI 그래픽스는 프리패스 중에 UI 툴킷의 동적 아틀라스 기능을 통해 자동으로 패킹됩니다.

시각적 요소 내에서 참조된 이미지는 UI 문서의 Panel Settings에 정의된 기준에 따라 아틀라스 처리됩니다. 예를 들어 패킹할 최소 또는 최대 텍스처 크기를 정의하거나 다른 프로퍼티를 기준으로 이미지를 필터링할 수 있습니다. 생성된 아틀라스는 UI Toolkit Debugger의 Texture Atlas Viewer에서 미리 볼 수 있습니다.

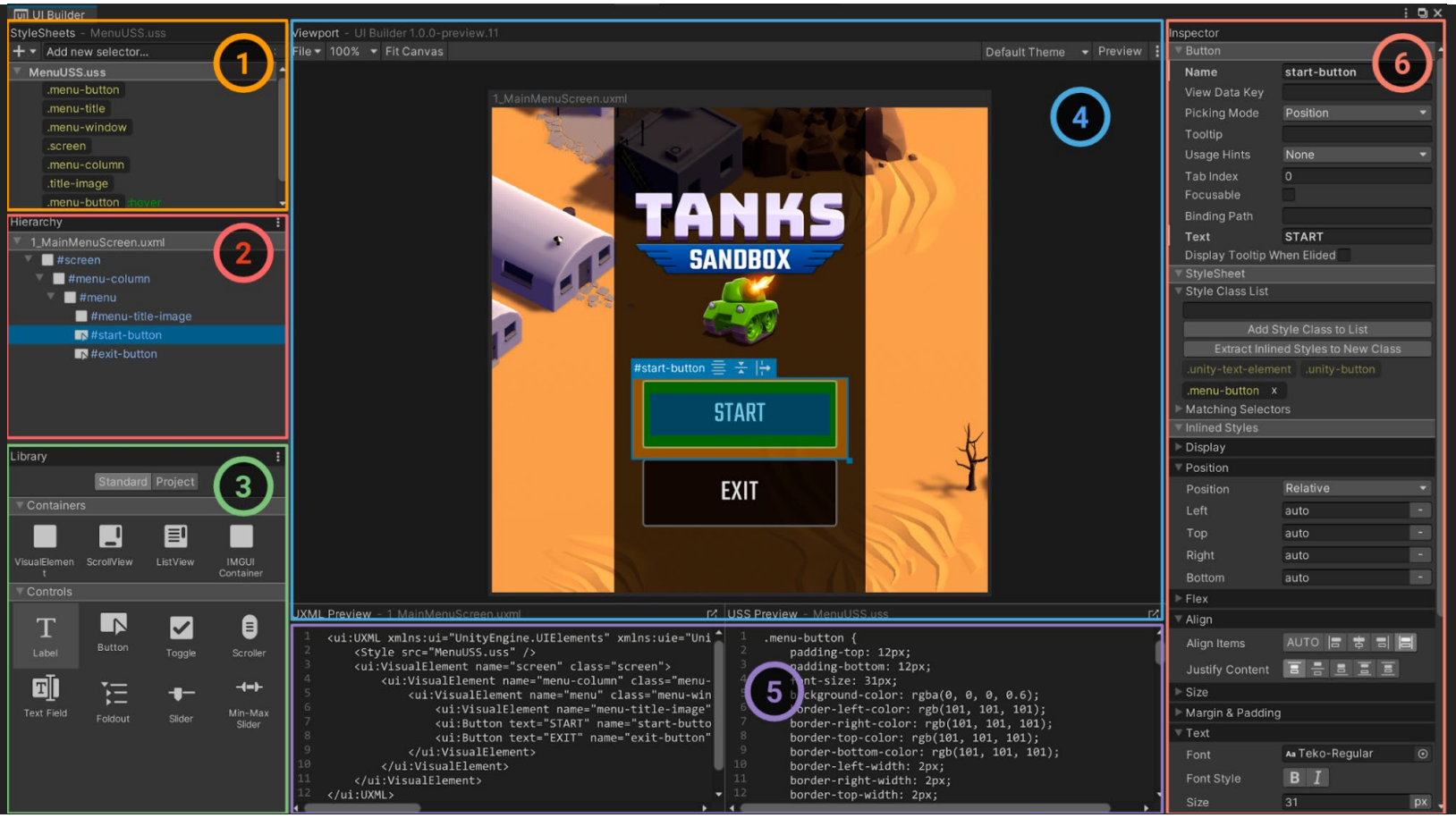
동적 아틀라스 툴은 에디터에서는 물론 런타임에도 작동하기 때문에 플레이어의 인벤토리와 같이 동적으로 생성되는 UI 요소에 유용합니다.

일반적인 그래픽스 모범 사례는 다음과 같습니다.

- 목업 화면을 제작하려면 먼저 드로잉 소프트웨어에서 최대 타겟 해상도를 설정해야 나중에 작업을 다시 해야 하는 상황을 피할 수 있습니다. 예를 들어 최대 4K 그래픽스까지 지원할 계획이라면 최소 작업 해상도를 4K로 맞춰야 합니다.
- 래스터 이미지를 만든 후에는 크기를 키우지 않는 것이 좋습니다. 래스터 이미지의 크기를 키우면 픽셀이 깨지고 블러 효과를 일으켜서 시각적 품질이 저하됩니다. 처음부터 지원되는 최고 해상도로 제작하고, 그래픽스 애플리케이션에서 익스포트할 때 아트의 크기를 줄이는 것이 좋습니다.

- 벡터 그래픽스로 디자인하는 경우, 나중에 에셋의 크기를 조정하는 것은 별문제가 되지 않습니다. 하지만 Reference Resolution을 기준으로 작업해야 모든 에셋에 올바른 상대 스케일을 적용(예: 요소의 윤곽선 두께를 일정하게 유지)할 수 있습니다.
- 그래픽스 에셋의 해상도가 필요한 해상도보다 낮다면 2D Enhancers와 스프라이트 에디터의 AI 기반 업스케일 기능을 사용해 보세요.
- 2D PSD Importer를 사용하여 PSD를 Unity에 바로 임포트합니다. 레이어의 변경 사항은 PSD 파일을 저장하면 Unity에 반영됩니다. Unity로 임포트한 PSD 파일에는 Version Control을 적용할 수도 있습니다.
- 임포트 프로세스를 자동화합니다. 그래픽 에셋을 추가할 때마다 에셋 설정을 수동으로 변경하지 않도록 하세요. 프리셋 기능을 사용하면 하나의 에셋에 적용된 설정을 저장하여 특정 폴더에 있는 동일한 유형의 모든 에셋에 자동으로 적용할 수 있습니다.
- 프로세스를 한층 더 자동화해야 하는 경우(예: 에셋을 검사하거나 여러 에셋의 설정을 일괄 변경해야 하는 경우) Asset PostProcessor API를 사용할 수 있습니다.

UI 빌더



UI 빌더는 Window/UI Toolkit/UI Builder 메뉴에서 액세스할 수 있습니다.

UI 빌더를 사용하면 메인 에디터에 통합된 시각적 인터페이스에서 UXML 및 USS 파일을 생성하고, 시각화하고, 수정할 수 있습니다. UI 빌더의 주요 기능을 살펴보겠습니다.

1. **StyleSheets:** UXML 문서와 UI 요소에서 스타일을 공유할 수 있도록 레이아웃 및 스타일 포맷 지정 규칙(USS 선택자)을 관리하는 곳입니다.
2. **계층 구조:** 씬(Scene) 뷰와 유사하게 UXML 문서에 포함된 시각적 요소의 계층 구조(Hierarchy)를 보여줍니다.
3. **Library:** 계층 구조에 추가할 수 있는 사전 정의된 컨트롤 또는 커스텀 컨트롤(버튼, 레이블, 슬라이더 등)이 포함되어 있습니다. 여기에서 현재 UXML(템플릿)에 다른 UXML을 추가할 수도 있습니다.
4. **뷰포트:** 인터페이스가 실제로 어떻게 보이는지 확인하고, 기즈모를 사용하여 캔버스 내에서 요소를 바로 편집할 수 있습니다.
5. **코드 프리뷰:** UI 문서(UXML) 및 StyleSheets(USS)에 대해 UI 빌더가 백그라운드에서 생성 중인 코드를 볼 수 있습니다. 더 많은 코드를 한눈에 보려면 창의 크기를 조정해야 할 수 있습니다.
6. **인스펙터:** 선택한 요소 또는 USS 선택자의 속성과 스타일 프로퍼티를 변경합니다.

팁: UI 에셋 저장

UI 빌더에서는 **뷰포트** 메뉴(**File > Save**)를 통해 변경 사항을 저장하세요. 이렇게 하면 열려 있는 모든 UXML 파일과 USS 파일이 저장됩니다.

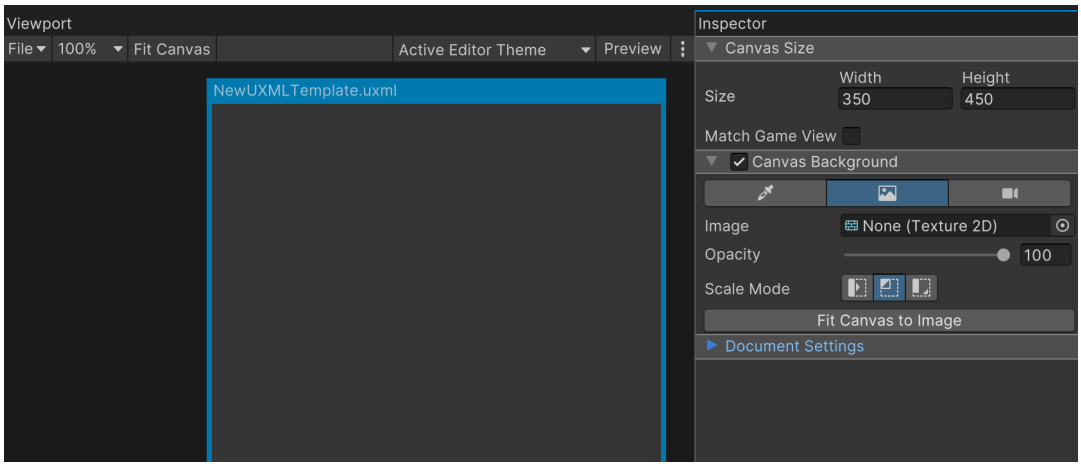
Unity UI와 달리, UI 툴킷에서 에셋을 변경하는 동안에도 에디터에서 게임이 계속 실행되도록 할 수 있습니다. UI 빌더의 캔버스 헤더에서 파일 이름 옆에 있는 별표(*)는 파일에 저장되지 않은 변경 사항이 있음을 나타냅니다.

캔버스 배경

캔버스 배경을 사용하면 특정 컬러나 배경 이미지 위에서 요소의 스타일을 시각화할 수 있습니다. 계층 구조 창에서 UXML 파일을 선택하고 최종 UI 인터페이스와 비슷한 캔버스 배경을 선택하면 컨텍스트에 맞는 스타일 변화를 확인할 수 있습니다.

캔버스 배경에 사용할 수 있는 몇 가지 옵션은 다음과 같습니다.

- **Background Color**: 게임 환경의 특정 음영이나 색조를 나타냅니다.
- **Image**: 스프라이트나 텍스처를 배경으로 선택할 수 있으며, 목업 화면이나 참조 아트를 모사할 때 유용합니다.
- **Camera**: 배경에 현재 게임플레이가 표시됩니다. 실제 게임의 컨텍스트에서 UI를 볼 수 있습니다.



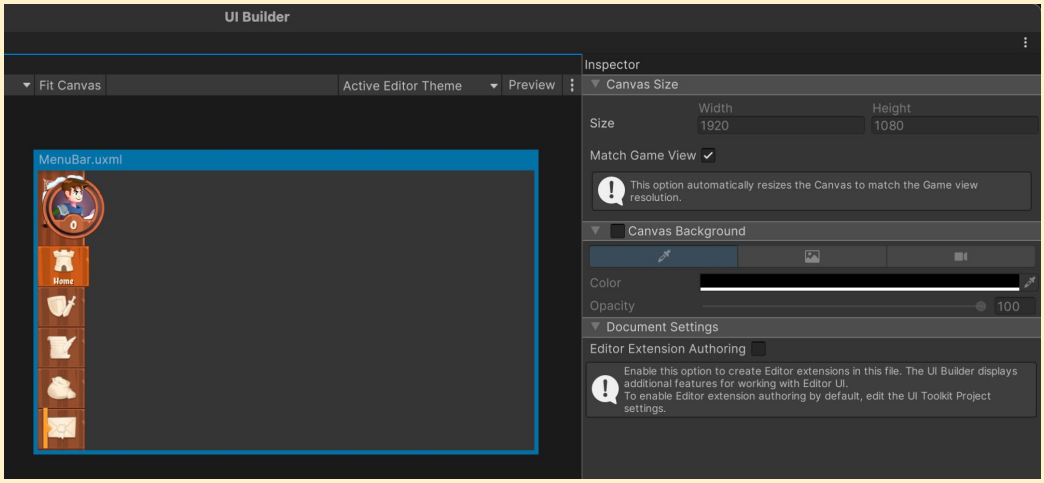
새 UXML 문서의 캔버스: Color 및 Image 옵션을 사용하여 외관을 조정합니다.

뷰포트 설정

작업 영역을 탐색하려면 줌 레벨을 조정(25%~500%)하거나, 현재 화면 공간에 따라 자동으로 줌을 조정하는 **Fit Canvas** 옵션을 선택합니다.

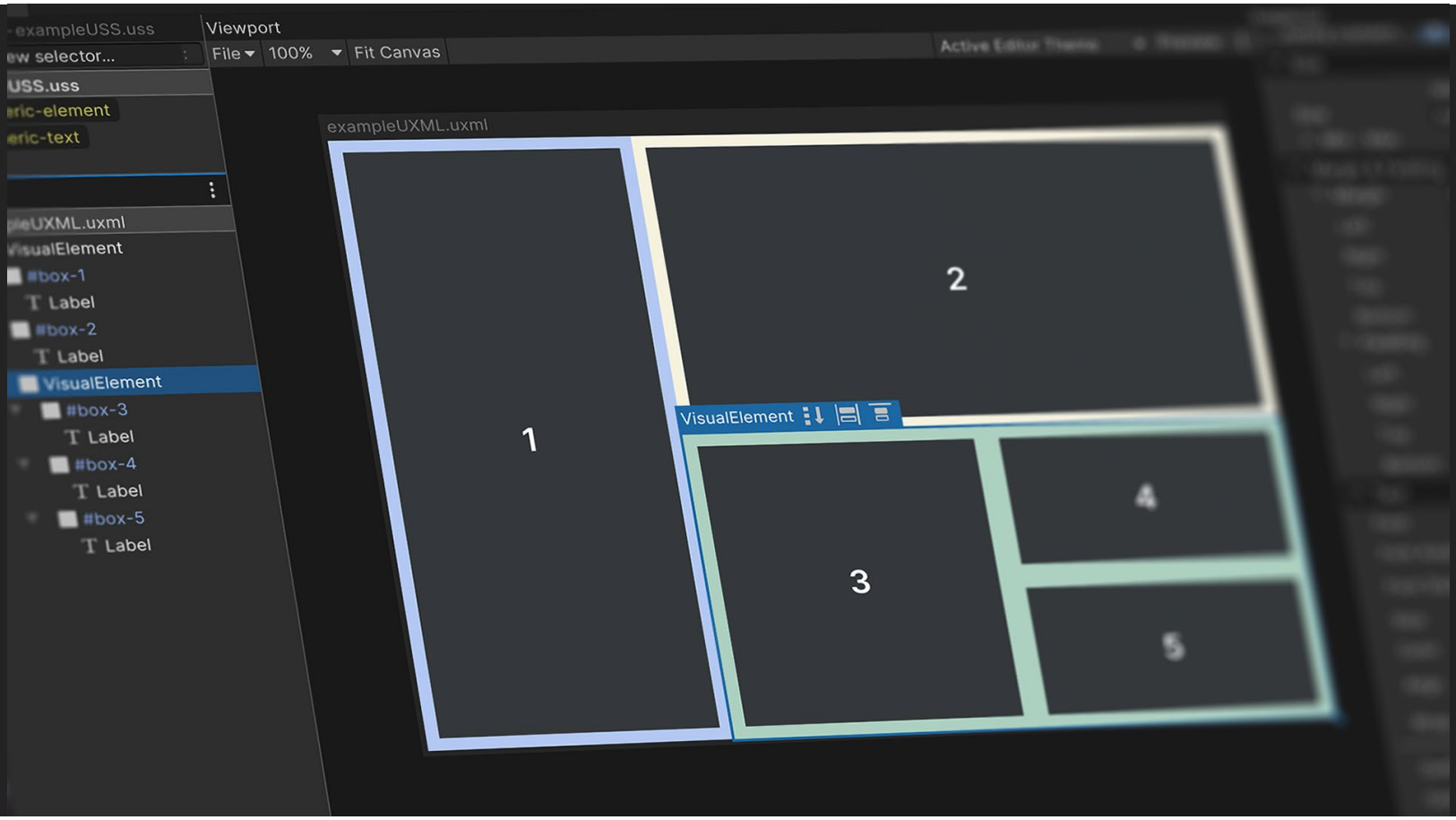
Preview를 사용하면 선택한 요소를 실수로 편집하는 일 없이 UI를 시각화할 수 있습니다. 활성화되면 뷰포트에서 특정 마우스 이벤트(예: 마우스 커서 올리기, 초점 맞추기 등)에 적용된 스타일도 확인할 수 있습니다.

팁: Match Game View와 테마



런타임 UI와 비슷하게 만들려면, 계층 구조에서 현재 로드된 UI 문서(UXML)를 선택하고 **Match Game View**를 선택합니다. 이 옵션을 선택하면 뷰포트의 크기가 프로젝트의 Reference Resolution 값에 맞게 조정됩니다. 이 파라미터를 수정해도 UI 파일 자체에는 영향을 주지 않으며, 시각화에만 영향을 니다. UI 빌더에서 프로젝트에 사용된 각종 테마를 사전 시각화할 수도 있습니다. 이 기능은 본 가이드의 뒷부분에서 다룹니다.

레이아웃



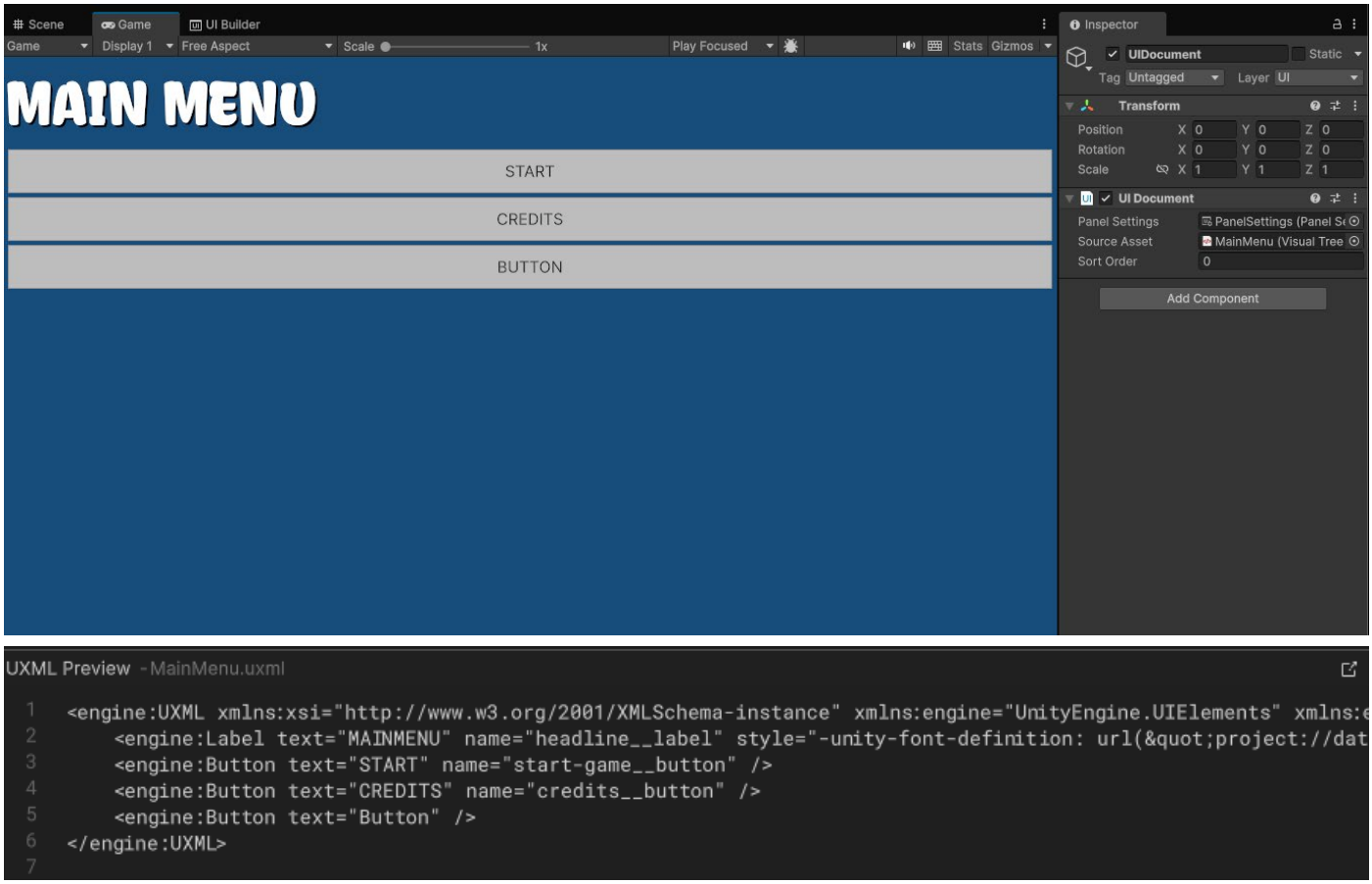
UI 빌더는 반응형 레이아웃을 디자인하는 데 필요한 모든 툴을 제공합니다.

이 섹션에서는 UI 빌더에서 레이아웃을 만드는 기본적인 단계를 설명합니다.

UI 빌더는 코드 작성 없이 UXML 및 USS 파일을 효율적으로 생성하도록 지원하는 디자이너 친화적인 WYSIWYG 방식의 툴입니다. 코드를 통해 직접 UI를 생성하는 방식을 선호하는 팀도 있을 수 있지만, UI 빌더는 아티스트가 창작물을 제어하도록 지원하여 워크플로를 크게 향상합니다. UI 빌더에서 내용을 변경하면 코드가 자동으로 생성되며, UI 빌더에서 생성한 모든 내용은 UXML 및 USS에서 바로 코드로 구현될 수 있습니다.

UI 툴킷과 UI 빌더 사용 시 얻을 수 있는 주요 장점은 반응형 레이아웃의 탁월한 효율성입니다. 반응형 레이아웃은 화면 해상도와 비율이 서로 다른 여러 플랫폼을 타게팅하는 게임에 필요한 기능입니다. 이 섹션에서는 UI 빌더에서 레이아웃을 만드는 기본적인 단계를 설명합니다.

아래 이미지에서는 UI 빌더의 UXML Preview 패널에 코드가 표시된 UXML 파일을 볼 수 있습니다. UI 빌더에서 에셋을 생성하려면 File > New를 선택하고 Save As를 선택합니다.



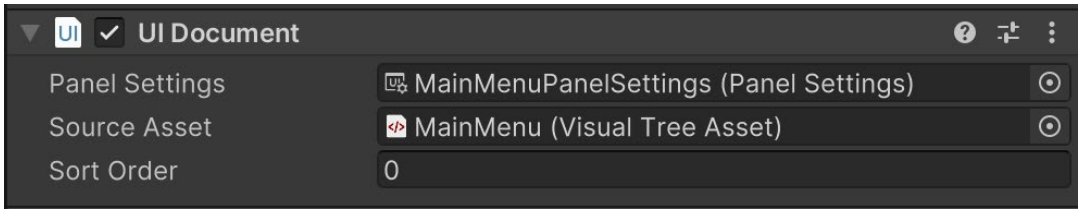
코드 프리뷰 창 오른쪽 상단에 있는 아이콘을 클릭하면 프리뷰가 IDE에서 열립니다.

이 UXML 코드 예시에서는 시각적 요소가 HTML과 유사한(여는 괄호로 시작하고 닫는 괄호로 끝나는) 마크업 언어로 표현된 것을 볼 수 있습니다. 예를 들어 다음은 START 버튼의 구문입니다.

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

핵심 런타임 컴포넌트

UI 툴킷 요소는 씬 뷰에 표시되지 않습니다. UI 빌더에서 제작 중인 인터페이스를 보며 작업할 수 있지만, 게임 뷰에서는 더 정확한 프리뷰를 타겟 해상도로 볼 수 있습니다. UI를 게임 뷰에 렌더링하려면 다음 스크린샷처럼 게임 오브젝트에 **Panel Settings** 에셋과 **Visual Tree Asset**(UXML)이 있는 **UI Document** 컴포넌트가 있어야 합니다.



UI Document 컴포넌트는 표시될 UXML을 정의하며 기본 Panel Settings 에셋을 제공합니다. Sort Order 필드는 이 문서가 동일한 Panel Settings를 사용하는 다른 UI Document와 관련하여 표시되는 방식을 결정합니다.

인스펙터에서 **Add Component** 메뉴를 사용하여 이 컴포넌트를 게임 오브젝트에 추가할 수 있습니다. 또는 계층 구조에서 오른쪽 클릭하고 **UI Toolkit > UI Document**를 선택해서 추가할 수도 있으며, 그러면 자동으로 Panel Settings 에셋이 할당됩니다.

Panel Settings 에셋은 런타임에 UI Document 컴포넌트가 인스턴스화되고 시각화되는 방식을 정의합니다. 여러 Panel Settings 에셋을 사용하여 UI에 다양한 스타일을 적용할 수 있습니다. 이를테면 게임에 HUD나 미니맵이 있는 경우, 이러한 특수 UI에 각각 고유한 Panel Settings를 사용할 수 있습니다.

Assets > Create > UI Toolkit > Panel Settings Asset을 통해 에셋을 생성합니다. 그러면 루트 프로젝트 폴더에 추가되고, 이를 게임 오브젝트의 UI Document 컴포넌트에 적용할 수 있습니다.

반응형 레이아웃: Flexbox

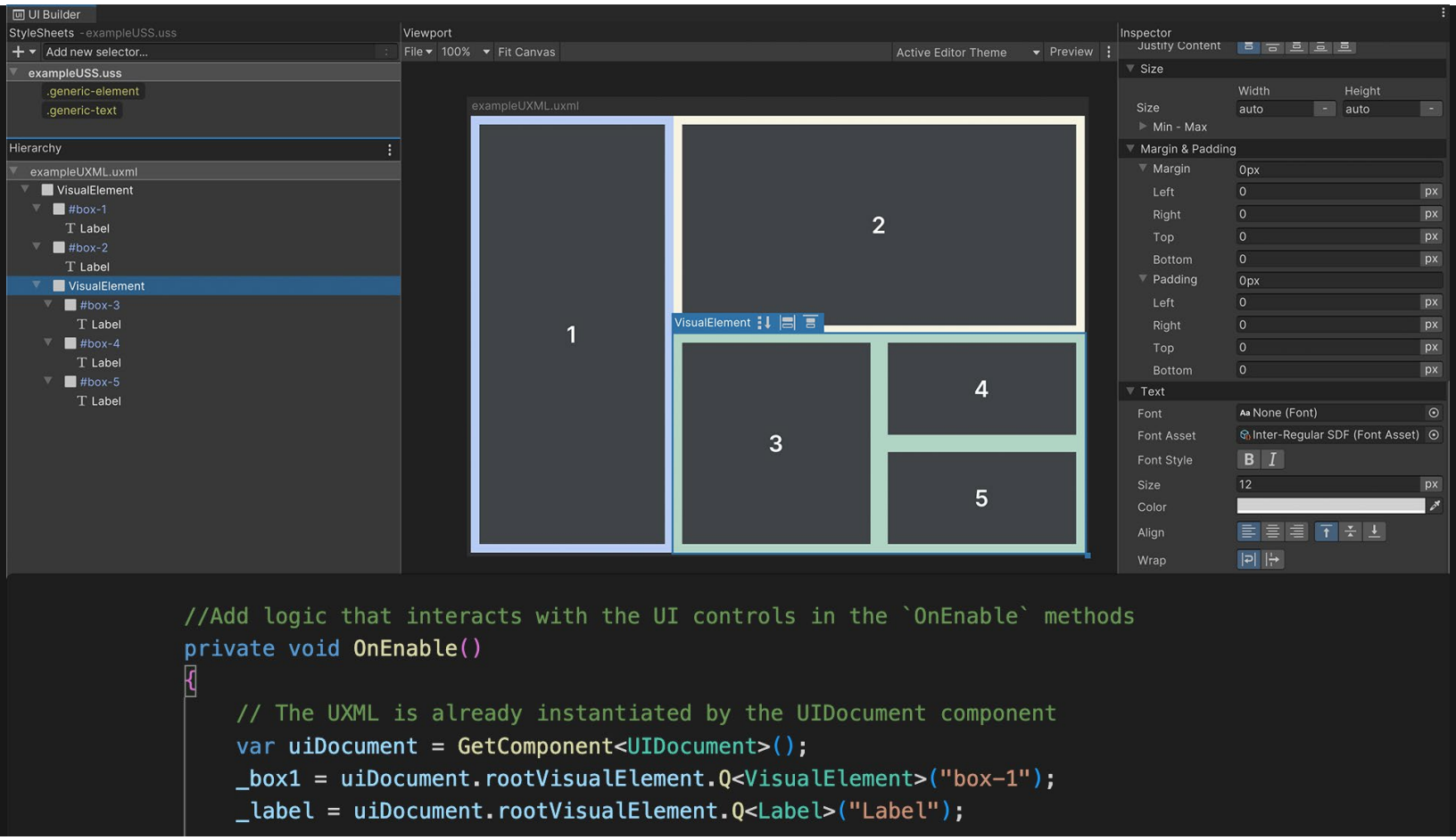
UI 툴킷은 Flexbox의 하위 집합을 구현하는 HTML/CSS 레이아웃 엔진인 Yoga 를 기반으로 시각적 요소의 위치를 지정합니다. Yoga나 Flexbox에 익숙하지 않더라도, 이 챕터에서 UI 툴킷의 레이아웃 엔진이 작동하는 방식을 빠르게 학습할 수 있습니다.

Flexbox(Flexible Box Layout)는 항목을 행 또는 열로 정렬하는 메서드입니다. Flexbox 아키텍처는 복잡하고 체계적인 레이아웃을 개발하는 데 적합합니다. Flexbox의 몇 가지 장점은 다음과 같습니다.

— **반응형 UI**: Flexbox는 모든 요소를 상자 또는 컨테이너 네트워크로 구성합니다. 이러한 요소를 부모-자식 관계로 중첩하고 간단한 규칙을 사용하여 화면에서 공간 효율적으로 정렬할 수 있습니다. 자식은 부모 컨테이너가 변경되면 자동으로 반응합니다. 반응형 레이아웃은 다양한 화면 해상도와 크기에 맞게 조정되므로, 더 쉽게 여러 플랫폼을 타게팅할 수 있습니다.

— **체계적인 복잡도**: 스타일은 시각적 요소의 미적 가치를 제어하는 간단한 규칙을 정의합니다. 스타일 하나를 동시에 수백 개의 요소에 적용할 수 있으며, 변경 사항은 전체 UI에 즉시 반영됩니다. 따라서 개별적인 요소의 외관을 작업하는 것이 아니라, 일관되고 재사용 가능한 스타일 위주로 UI를 디자인할 수 있습니다.

— **로직 및 디자인 분리**: UI 레이아웃과 스타일이 코드와 분리됩니다. 따라서 디자이너와 개발자가 종속성을 해치지 않고 원활하게 병렬로 작업할 수 있습니다. 각 담당자가 가장 잘하는 일에 집중할 수 있게 하세요.



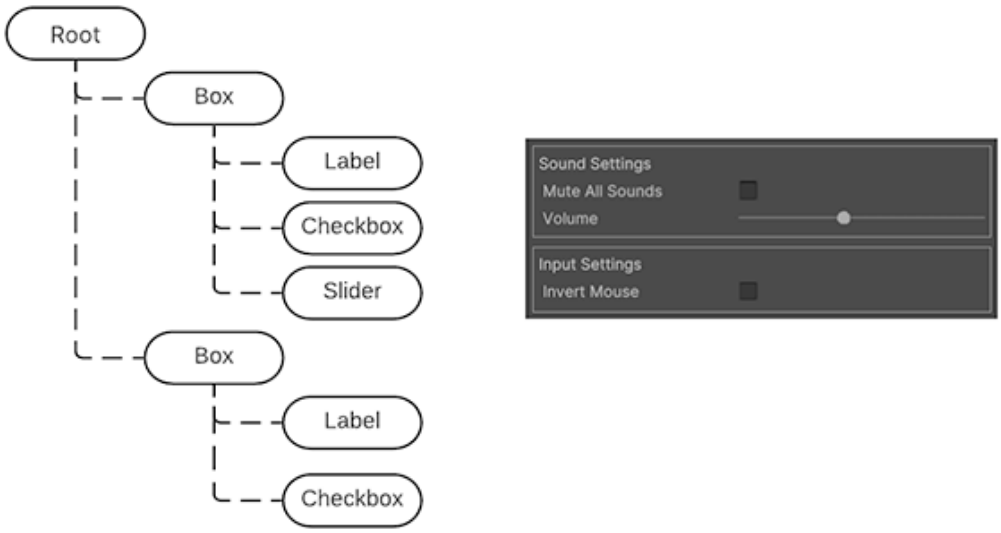

```
//Add logic that interacts with the UI controls in the `OnEnable` methods
private void OnEnable()
{
    // The UXML is already instantiated by the UIDocument component
    var uiDocument = GetComponent<UIDocument>();
    _box1 = uiDocument.rootVisualElement.Q<VisualElement>("box-1");
    _label =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

로직 및 디자인 분리: 프로그래머는 시각적 요소를 실제 게임 로직에 연결하고 디자이너는 해당 요소의 스타일을 정의하는 데 집중할 수 있습니다.

시각적 요소

UI 툴킷에서 각 인터페이스의 기본 구성 요소는 시각적 요소입니다. 시각적 요소는 모든 UI 툴킷 요소(버튼, 이미지, 텍스트 등)의 기본 클래스입니다. 시각적 요소는 UI 툴킷에서 사용하는 게임 오브젝트라 할 수 있습니다.

하나 이상의 시각적 요소로 구성된 **UI 계층 구조**를 **시각적 트리**라고 합니다.



시각적 트리의 간소화된 UI 계층 구조(왼쪽) 및 실제 보이는 모습(오른쪽)

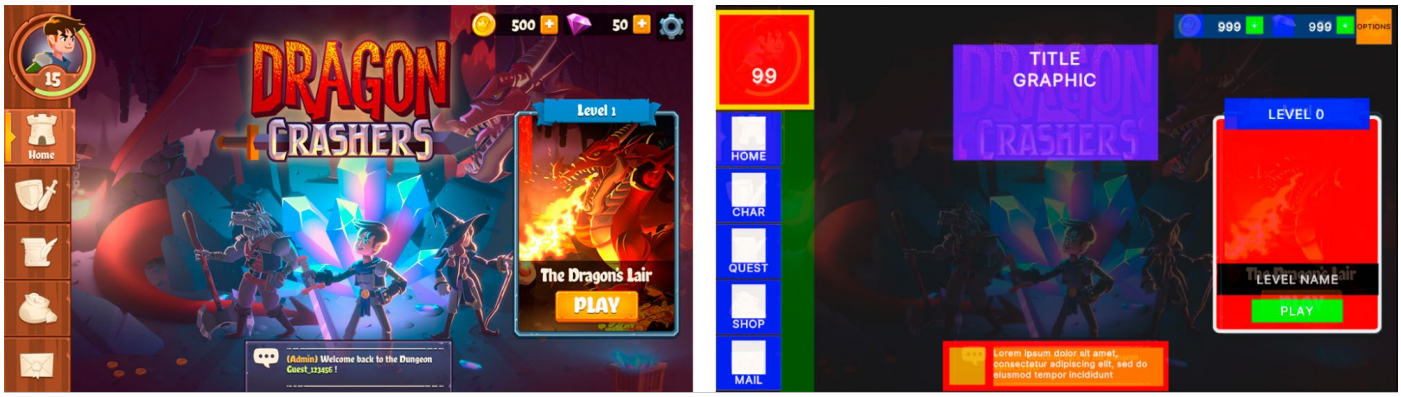
여러 시각적 요소의 조합은 UXML 파일에 저장됩니다. 이 파일에는 계층 구조와 관련된 정보에 더해 시각적 요소의 스타일(StyleSheet 또는 USS를 사용하지 않는 경우) 및 레이아웃이 포함되어 있습니다.

UI 툴킷에 대해 더 자세히 알아보기 전에 **Flexbox 레이아웃**의 기초를 이해해야 하며, UI 빌더의 기본 시각 요소로 시연해 볼 수 있습니다.

시각적 요소의 위치 지정

UI 목업을 만들 때는 각 화면을 개별적인 시각적 요소 그룹으로 보고 접근하세요. 화면을 가로 또는 세로 방향으로 쌓인 여러 상자로 분할하는 방법을 생각해 보고, 정보를 체계적으로 구성할 수 있도록 자식 상자가 필요한지 확인해야 합니다.

아래 예시에서는 하나의 커다란 시각적 요소 메뉴 하나가 왼쪽의 메뉴 바와 그 요소로 구성된 하나의 컨테이너일 수 있습니다. 각 버튼을 나타내는 자식 시각 요소가 분리되어 있습니다.



권장되는 방식은 상세한 목업 또는 와이어프레임(왼쪽)을 만들어 두고 여기에서 개별 요소를 식별하고 대략적으로 배치하여 UI 툴킷에서 디자인을 재현(오른쪽)하는 것입니다.

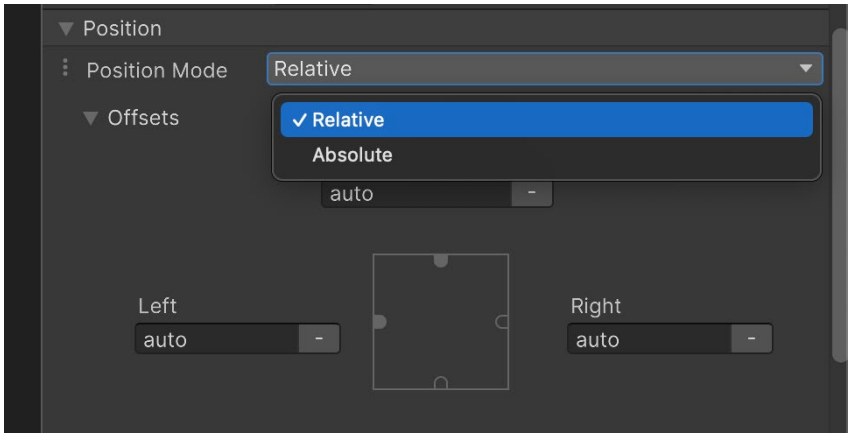
UI 빌더는 시각적 요소에 대해 다음과 같은 두 가지 위치 지정 옵션을 제공합니다.

— **Relative Position**: 이 옵션은 새 시각적 요소의 기본 설정입니다. 자식 요소는 부모 컨테이너의 Flexbox 규칙을 따릅니다. 예를 들어 부모 요소의 **Direction**이 **Row**로 설정되면, 자식 시각 요소는 왼쪽에서 오른쪽으로 정렬됩니다. Relative Position은 다음을 기준으로 요소의 크기를 동적으로 조정하고 옮깁니다.

  — **부모 요소의 크기 또는 스타일 규칙**: **Padding or Align > Justify Content**를 통해 부모 요소의 설정을 수정하면 해당 변경 사항에 따라 자식 요소도 조정됩니다.

  — **자식 요소의 자체 크기 및 스타일 규칙**: 자식 시각적 요소에 자체적인 최소 또는 최대 크기 설정이 있는 경우, 레이아웃 엔진은 해당 설정을 최대한 반영합니다.

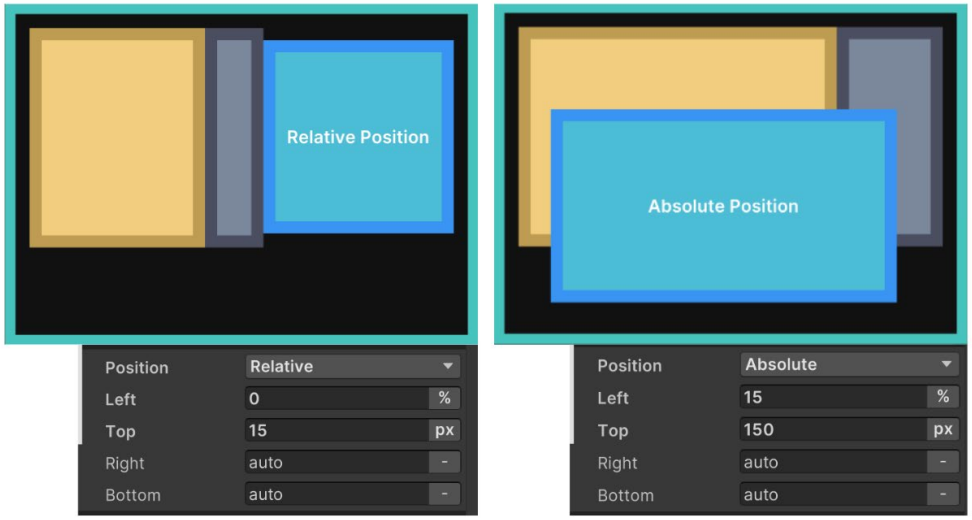
  UI 툴킷은 부모 요소와 자식 요소 간의 충돌을 처리합니다. 예를 들어 컨테이너에 포함된 자식 요소의 최소 너비가 컨테이너보다 넓다면 오버플로가 발생합니다.



모든 시각적 요소에서 사용할 수 있는 Position Mode

— **Absolute Position**: 여기서 시각적 요소의 위치는 캔버스에서 Unity UI가 작동하는 방식과 유사하게 부모 컨테이너에 고정됩니다. 크기 규칙 또는 자식 요소에 영향을 주는 규칙은 계속 적용되지만, 요소 자체는 **Grow**, **Shrink**, **Margins**와 같은 Flex 설정을 무시하고 부모 컨테이너 위에 오버레이됩니다.

  절대적으로 위치가 지정된 요소는 **Left**, **Top**, **Right**, **Bottom** 설정을 앵커로 사용할 수 있습니다. 예를 들어 Right 및 Bottom 값이 0이면 버튼이 부모 컨테이너의 오른쪽 하단에 고정됩니다.



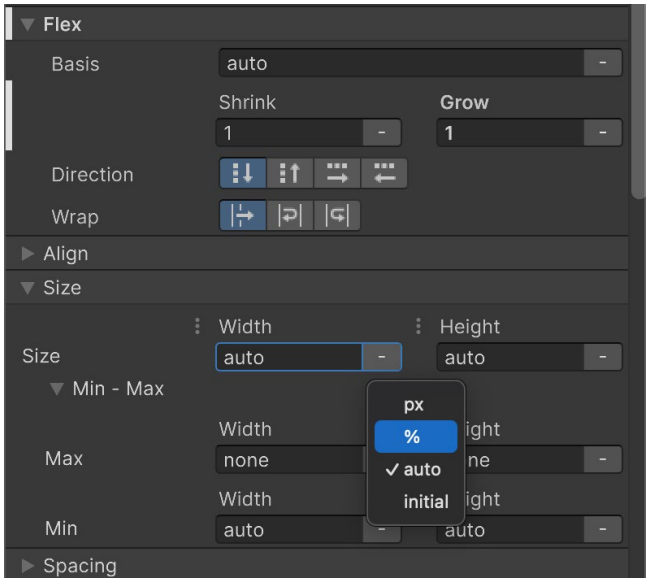
왼쪽의 파란색 시각적 요소는 Relative Position을 사용하며, 부모 요소는 Flex 설정으로 **Direction: Row**를 사용합니다. 오른쪽의 파란색 시각적 요소는 Absolute Position을 사용하며, 부모 요소의 Flexbox 규칙을 무시합니다.

영구적으로 표시해야 하거나, 그룹화가 복잡하거나, 여러 요소가 포함된 요소에는 Relative Position을 사용하는 것이 좋습니다.

Absolute Position은 팝업 창 같은 임시 UI, 레이아웃 구도에 방해가 되지 않는 장식적인 요소, 캐릭터의 체력 바 등 다른 게임 내 요소의 위치를 따르는 요소에 유용합니다.

크기 설정

시각적 요소는 단순한 컨테이너에 불과합니다. Unity 6에서 시각적 요소의 기본 Grow 설정은 1로 지정됩니다(컨테이너에서 사용 가능한 모든 공간을 차지). 이 설정이 1이 아닌 경우, 이미 구체적인 크기 값이 있는 다른 자식 요소로 채워져 있거나 특정 **Width** 및 **Height** 값을 설정하지 않는 한 시각적 요소는 공간을 차지하지 않습니다.



시각적 요소의 크기 설정

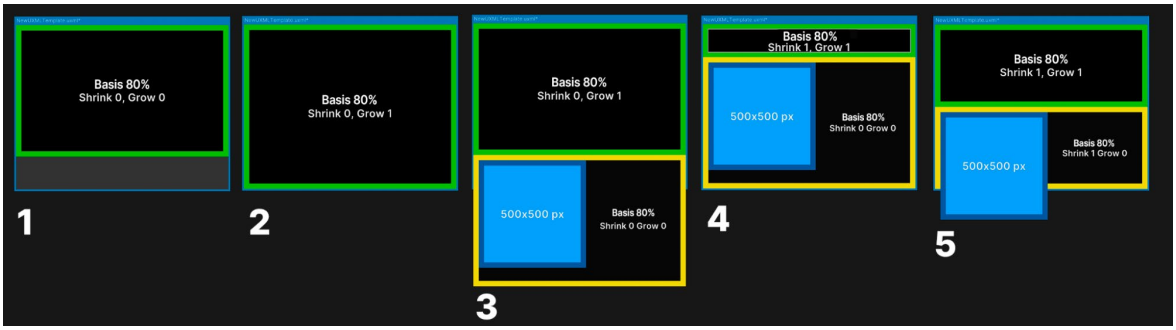
Width 필드와 Height 필드는 요소의 크기를 정의합니다. **Max Width**와 **Max Height**는 요소가 확장될 수 있는 최대 크기를 제한합니다. 마찬가지로, **Min Width**와 **Min Height**는 요소가 축소될 수 있는 최소 크기를 제한합니다. 크기를 픽셀 단위 또는 백분율로 정의하여 기본 설정을 오버라이드할 수 있습니다. 이는 사용 가능한 공간에 따라 요소의 크기를 조절하는 Flex 설정(아래)에 영향을 줍니다.

Flex 설정

Flex 설정은 Relative Position을 사용할 때 요소의 크기에 영향을 줄 수 있습니다. 요소의 동작을 이해하려면 직접 실험해 보는 것이 좋습니다.

Basis는 항목에 Grow 또는 Shrink 비율 연산이 수행되기 전의 기본 Width 및 Height를 나타냅니다.

- Grow를 1로 설정하면 이 요소는 부모 요소에서 사용할 수 있는 모든 세로 또는 가로 공간을 차지합니다. Grow를 0.5로 설정하면 사용 가능한 모든 공간의 절반을 차지합니다.
- Grow를 0으로 설정하면 요소는 현재 Basis(또는 크기)를 초과하여 커지지 않습니다.
- Shrink를 1로 설정하면 요소는 부모 요소의 사용 가능한 공간에 맞도록 필요한 만큼 축소됩니다.
- Shrink를 0으로 설정하면 요소가 축소되지 않으며, 필요한 경우 오버플로됩니다.



Basis, Grow, Shrink 설정

위 예시는 Basis가 Grow 및 Shrink 옵션과 어떻게 함께 작동하는지 보여 줍니다.

1. Basis가 80%인 초록색 요소는 사용 가능한 공간의 80%를 차지합니다.
2. Grow를 1로 설정하면 초록색 요소가 전체 공간으로 확장됩니다.
3. 노란색 요소가 추가되면 요소가 공간을 오버플로하게 됩니다. 초록색 요소는 다시 공간의 80%를 차지하는 상태로 돌아갑니다.
4. Shrink를 1로 설정하면 노란색 요소가 공간 안에 들어오도록 초록색 요소가 축소됩니다.
5. 두 요소 모두 Shrink 값을 1로 설정하면 사용 가능한 공간에 맞도록 똑같이 축소됩니다.

보시다시피 픽셀 단위로 크기가 고정된 요소(3~5번의 파란색 상자)는 Basis, Grow, Shrink 설정에 반응하지 않습니다.

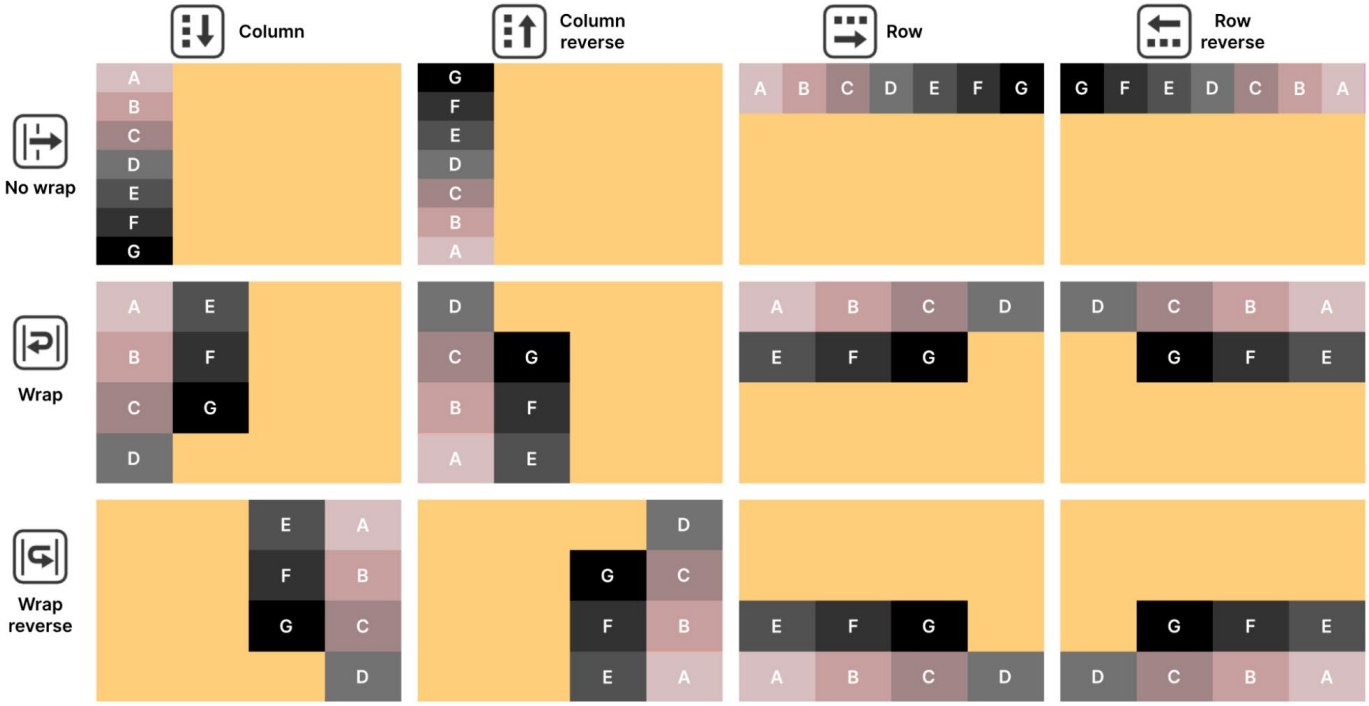
팁: 시각적 요소 크기 계산

Relative Position을 사용할 때 레이아웃 엔진은 Size 설정과 Flexbox 설정을 함께 사용하여 각 요소를 얼마나 크게 표시할 것인지 결정합니다. 시각적 요소의 크기는 다음 단계에 따라 계산됩니다.

1. 레이아웃 시스템이 Width 및 Height 프로퍼티를 기반으로 요소의 크기를 계산합니다.
2. 레이아웃 엔진이 부모 컨테이너에 사용 가능한 공간이 더 있는지, 또는 자식 요소가 이미 사용 가능한 공간을 오버플로하는지 확인합니다.
3. 사용 가능한 공간이 더 있다면 레이아웃 시스템이 Flex/Grow 설정값이 0이 아닌 요소를 찾습니다. 시스템은 해당 인자에 따라 추가 공간을 분배하여 자식 요소를 확장합니다.
4. 자식 요소가 가용 공간을 오버플로하는 경우, Flex/Shrink 값이 0이 아닌 요소는 그에 따라 크기가 줄어듭니다.
5. 요소의 결과적인 크기에 영향을 주는 다른 모든 프로퍼티(Min-Width, Flex-Basis 등)는 그다음에 고려됩니다.
6. 레이아웃 엔진이 최종 결정된 크기를 적용합니다.

Direction 설정은 부모 내에서 자식 요소가 정렬되는 방식을 정의합니다. 계층 구조 메뉴에서 위쪽에 있는 자식 요소일수록 먼저 표시되고, 계층 구조의 끝에 있는 요소가 마지막에 표시됩니다.

Wrap 설정은 레이아웃 시스템에서 요소를 하나의 열이나 행에 맞출 것인지(No Wrap), 아니면 다음 행이나 열에 표시할 것인지(Wrap 또는 Wrap reverse) 결정합니다.



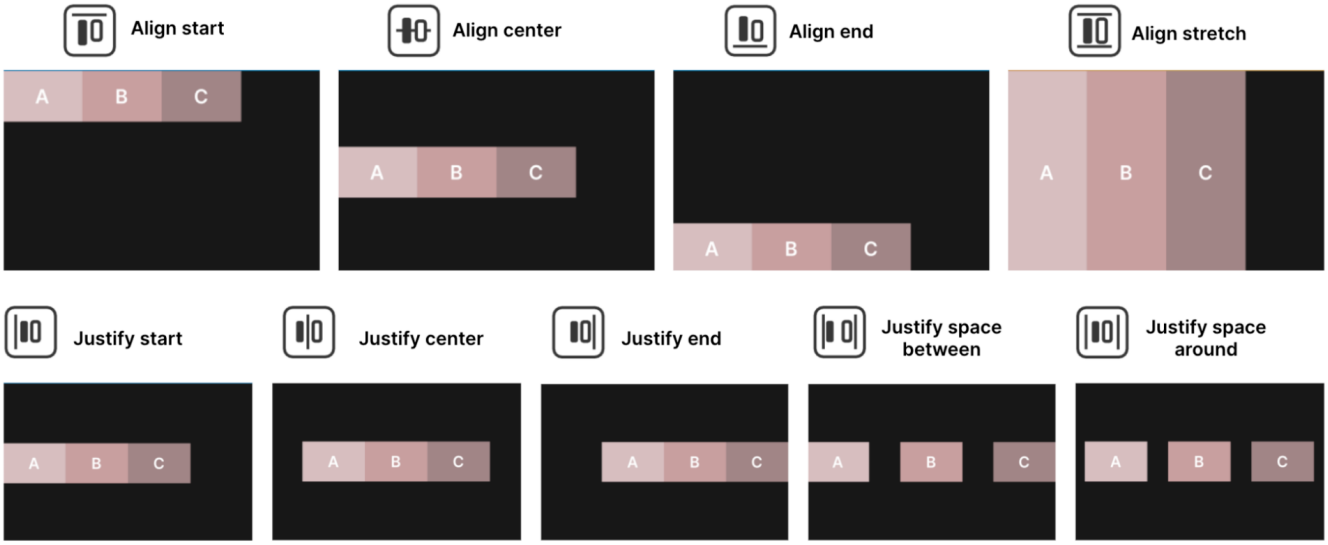
Relative Position과 서로 다른 Direction 및 Wrap 설정을 사용하는 UI 빌더의 부모와 자식 시각적 요소

정렬 설정

Align 설정은 자식 요소가 부모 요소에 정렬되는 방식을 결정합니다. 부모의 **Align > Align Items**를 설정하여 자식 요소를 시작, 가운데, 끝 중 하나로 정렬합니다. 이 옵션은 교차 축(**Flex > Direction**의 행 또는 열에 수직인 축)에 영향을 줍니다.

Stretch 옵션도 교차 축에 따라 작동하지만, 크기 설정의 Min 또는 Max 값으로 인해 제한될 수 있습니다(기본값). 한편 **Auto** 옵션은 레이아웃 엔진이 다른 파라미터에 따라 다른 옵션 중 하나를 자동으로 선택할 수 있음을 나타냅니다. 레이아웃을 더 세밀하게 제어하려면 옵션 중 하나를 선택하고, 특별한 사용 사례에는 Auto 옵션을 주로 사용하는 것이 좋습니다.

Align > Justify Content로 이동하여 레이아웃 엔진이 부모 내에서 자식 요소의 간격을 어떻게 정할 것인지 정의할 수 있습니다. 요소를 일렬로 정렬할 수도 있고, 서로 인접하게 정렬할 수도 있고, 사용 가능한 공간 내에서 펼쳐 놓을 수도 있습니다. **Flex > Grow** 설정과 **Flex > Shrink** 설정은 결과 레이아웃에 영향을 줍니다.

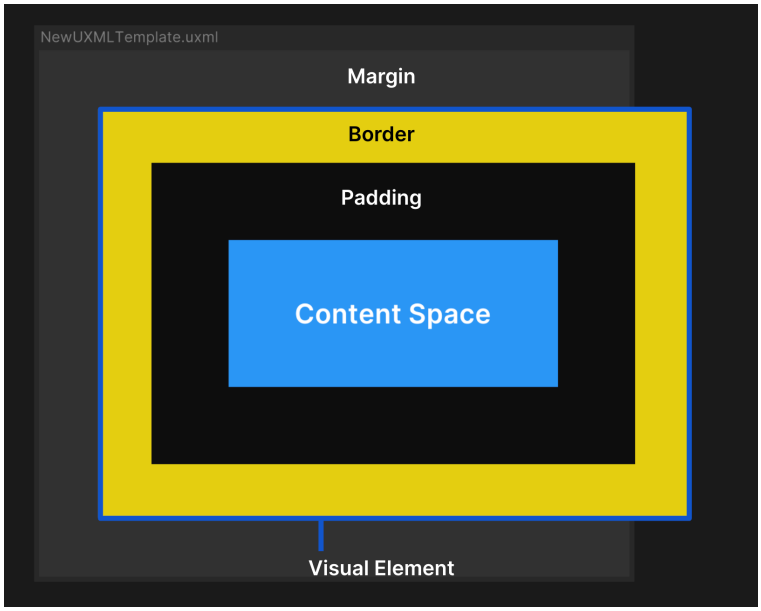


Direction을 Row로 설정한 부모 요소에 Align 및 Justify 설정을 적용한 모습. 다른 위치 및 크기 조정 옵션이 최종 결과물에 영향을 줄 수 있습니다.

Align Self 옵션을 선택하면 컨테이너가 Flex 레이아웃의 중앙, 끝 또는 시작 위치에 정렬됩니다.

Margin 및 Padding

Margin 및 Padding 설정을 사용하여 시각적 요소와 그 콘텐츠 주변의 빈 공간을 정의할 수 있습니다. Unity는 아래 다이어그램과 비슷한 표준 CSS 박스 모델의 배리에이션을 사용합니다.



Size, Margin, Border, Padding 설정이 정의된 UI 빌더의 시각적 요소. Width 또는 Height가 고정된 요소는 공간을 오버플로할 수 있습니다.

— **Content Space**에는 텍스트, 이미지, 컨트롤 등 주요 시각적 요소가 있습니다.

— **Padding**은 Border 내 Content Space 주변의 빈 영역을 정의합니다.

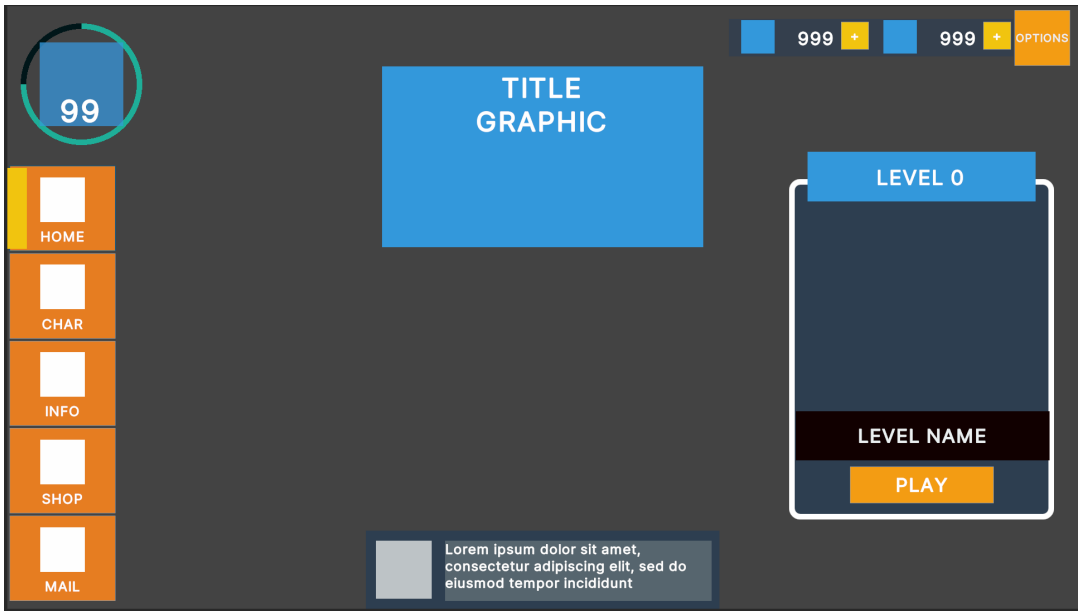
— **Border**는 Padding과 Margin 사이의 경계를 정의합니다. 테두리에 컬러를 정의하거나 테두리를 둥글게 만들 수 있습니다. 두께를 지정하면 Border는 안쪽으로 확장됩니다.

— **Margin**은 Padding과 비슷하지만 Border 바깥 영역을 정의합니다. Absolute Position을 사용하는 요소의 경우 Margin 설정은 영향을 주지 않지만, Position 설정을 사용하여 앵커 포인트를 기준으로 바깥 공간을 추가할 수 있습니다.

배경 및 이미지

UI 툴킷에서 모든 시각적 요소는 화면 이미지를 표시하는 데 사용될 수 있습니다. 배경 프로퍼티를 설정하여 텍스처나 스프라이트를 표시할 수 있습니다.

컬러나 이미지를 채워서 요소의 외관을 변경할 수도 있습니다. 이는 와이어프레임에 유용합니다. 콘트라스트가 높고 밝은 컬러를 사용하면 서로 다른 요소가 서로의 옆에서 어떻게 보이는지, 컨테이너 변경 사항에 어떻게 반응하는지 보여 줄 수 있습니다.



와이어프레임 작업 중에는 대비되는 컬러를 사용하세요.

가변 또는 고정 측정 단위

UI 빌더에는 요소의 거리와 크기를 정의하는 네 개의 파라미터가 있습니다.

— **Auto**: 크기 및 위치에 대한 기본 옵션입니다. 레이아웃 시스템이 부모와 자식 요소의 정보를 기반으로 요소의 값을 계산합니다.

— **Percentage**: 단위는 요소 컨테이너의 1퍼센트와 동일하며, 부모의 Width 및 Height에 따라 동적으로 변합니다. 백분율을 사용하면 여러 포맷 크기를 다루어야 할 때 확장성을 확보할 수 있습니다.

— **Pixels**: 이 옵션은 요소에 고정된 크기를 적용하려는 경우에 유용합니다. 크기가 작은 요소가 언제나 텍스트가 잘리지 않을 만큼의 최소 크기를 픽셀 단위로 유지해야 하는 경우를 예로 들 수 있습니다.

— **Initial**: 현재 스타일 값을 무시하고 프로퍼티를 기본 상태(Unity의 자체 기본 스타일 규칙)로 되돌립니다.

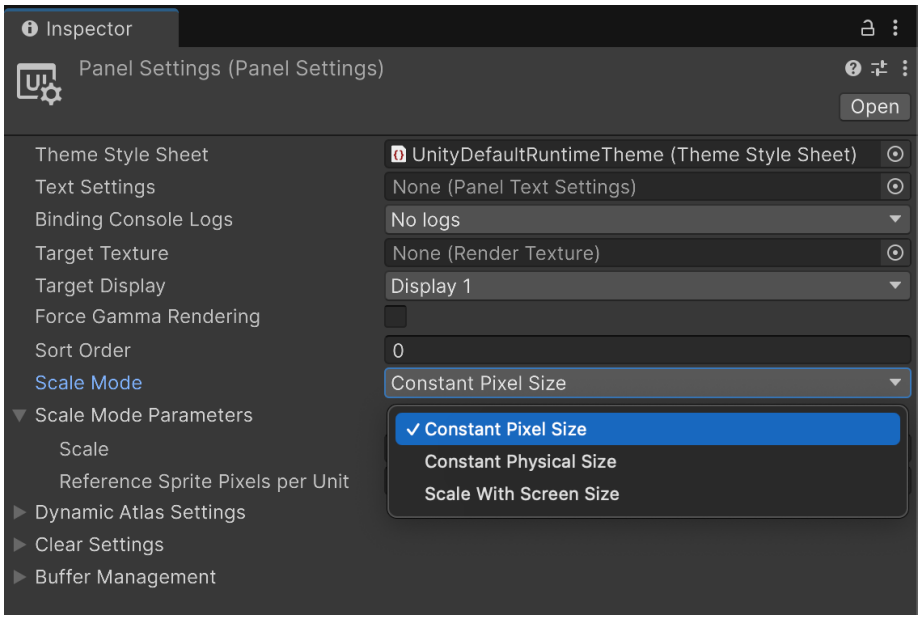
기본 Size 설정 및 픽셀 및 백분율로 정의한 Size 설정 예시

전체 UI에 동시에 스케일링 규칙을 적용하려면 Panel Settings에서 **Scale Mode** 파라미터를 조정하면 됩니다.

— **Constant Pixel Size:** 이 스케일 모드는 요소를 화면 크기와 무관하게 고정된 픽셀 크기로 유지합니다. 요소 크기에 곱할 값으로 Scale Factor를 지정할 수 있습니다.

— **Constant Physical Size:** 이 모드는 모든 화면에서 요소를 동일한 물리적 크기로 유지합니다. 실제 화면 DPI가 다를 경우 시스템이 Reference DPI를 기준으로 UI를 스케일링하여 크기를 조정합니다.

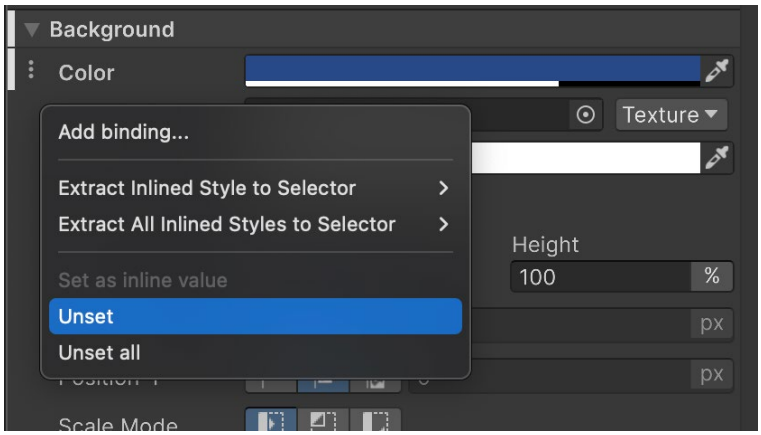
— **Scale with Screen Size:** 이 옵션은 해상도를 기준으로 요소의 크기를 동적으로 조정합니다. **Screen Match Mode**에 따라 너비, 높이, 너비와 높이 둘 모두 중에서 스케일링 시 우선적으로 고려하는 기준이 정해집니다. **Reference Resolution**은 UI의 기본 크기를 설정합니다. Screen Match Mode가 **Match Width or Height**로 설정된 경우, **Match** 값은 UI 시스템이 화면 너비, 화면 높이, 화면 너비와 높이 둘 모두 중 어느 것에 맞게 UI를 스케일링할지를 제어합니다.



UI 툴킷의 Panel Settings에서는 Unity UI의 옵션과 비슷한 스케일링 옵션을 사용할 수 있습니다.

UI 빌더의 오버라이드된 프로퍼티

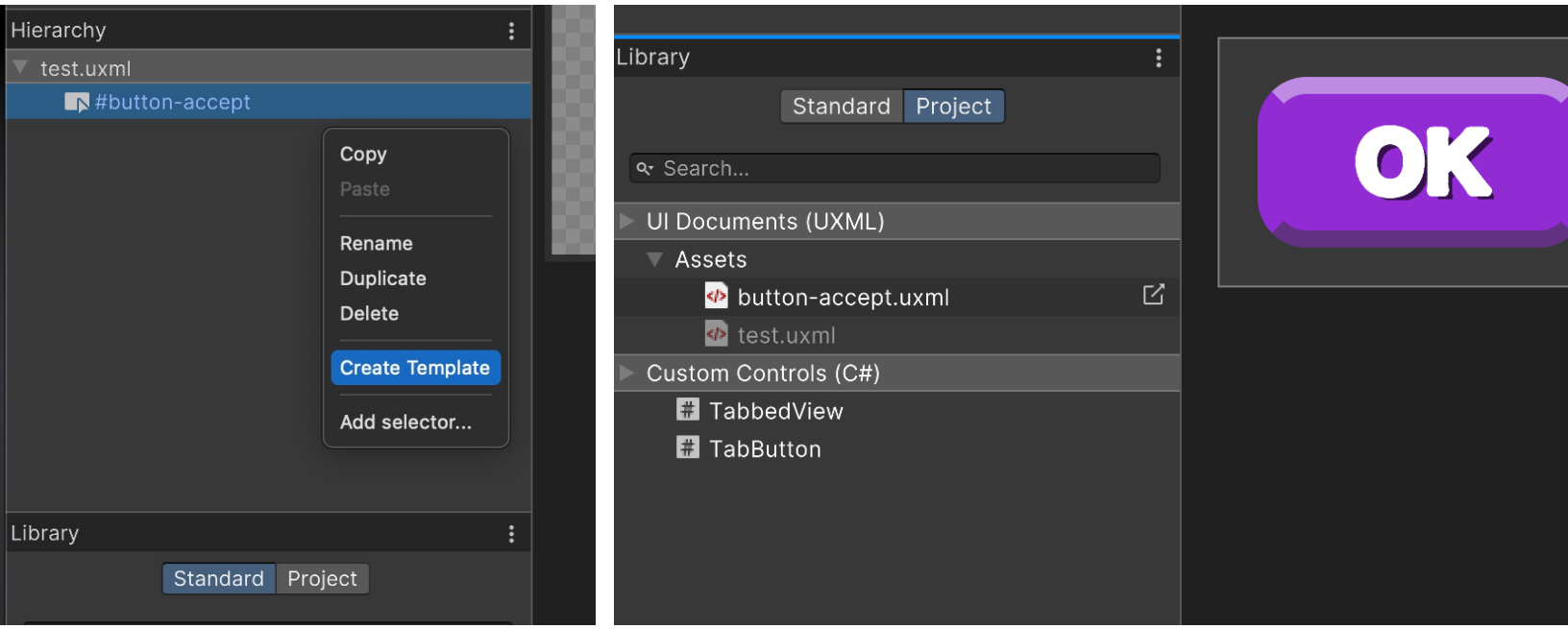
수정된 프로퍼티는 아래 스크린샷처럼 인스펙터에서 굵은 폰트와 왼쪽의 흰색 선으로 강조 표시됩니다. 이는 선택된 UXML 파일의 스타일 시트(USS)에 정의된 선택자 값이나 기본값이 오버라이드되었음을 나타냅니다. 이와 같은 동작을 '인라인 스타일링'이라고 합니다. 값을 수정할 필요가 없는 경우에는 변경 사항을 쉽게 찾고 관리할 수 있도록 기본값으로 두는 것이 좋습니다. 프로퍼티를 기본값으로 재설정하려면 프로퍼티 섹션 옆의 점(:) 메뉴에 있는 옵션을 사용할 수 있습니다.



이 메뉴에서 수정된 값을 기본값이나 선택자에 원래 정의되었던 값으로 복원할 수 있습니다.

UXML을 템플릿으로 사용

UXML 파일은 프리팹처럼 사용할 수 있습니다. 예를 들어 프로젝트에 하나의 항목 아이콘과 카운트 수를 갖는 UXML 레이아웃이 있고, 인벤토리 내에서 여러 번 생성해야 한다고 가정해 보겠습니다. UXML을 오른쪽 클릭하면 Create Template 옵션이 나타납니다. 이렇게 생성한 템플릿은 계층 구조 패널에 있는 시각적 요소에 추가하거나 코드를 통해 인스턴스화할 수 있습니다. 생성한 템플릿은 Library의 Project 탭에서 볼 수 있습니다.



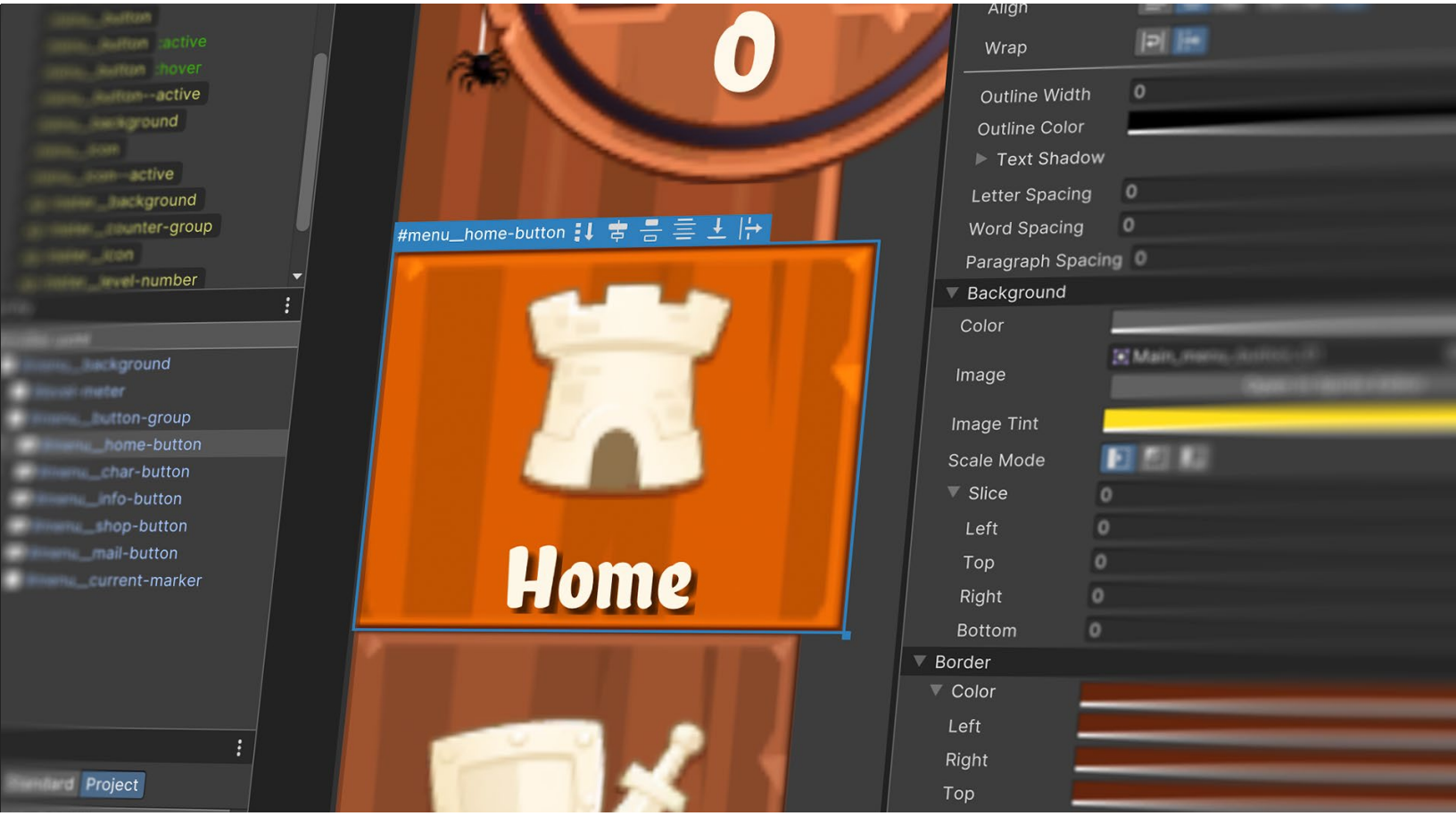
템플릿은 재사용 가능한 UXML로, Library 패널의 Project 탭에서 볼 수 있습니다.

더 많은 자료

Flexbox 레이아웃 엔진에 대해 자세히 알아보려면 다음 자료를 참고하세요. Flexbox와 Yoga는 기존의 웹 및 앱 개발 표준이므로 온라인에서 다양한 자료를 찾아볼 수 있습니다.

- 런타임 UI 제작을 위한 UI 툴킷 요약
- Yoga 공식 기술 자료
- CSS-Tricks의 Flexbox 가이드

스타일링



UI 빌더에서 직접 스타일 프로퍼티 변경하기

시각적 요소로 와이어프레임 레이아웃 목업을 만들었다면 스타일을 지정하거나 포맷 지정 프로퍼티를 재사용 가능한 스타일로 저장할 수 있습니다. 스타일을 지정할 때 UI 툴킷의 진가를 확인할 수 있습니다.

시각적 요소에 스타일을 추가할 때는 USS(Unity 스타일 시트) 파일(**Assets > Create > UI 툴킷 > StyleSheet**)을 사용하는 것이 좋습니다. 이러한 파일은 웹 CSS 파일에 해당하는 Unity 파일이며, 유사한 규칙 기반 포맷을 사용합니다. 또한 디자인 프로세스에 유연성을 더해 주기 때문에 손쉽게 프로젝트 전반에서 재사용하고 스타일의 일관성을 유지할 수 있습니다.

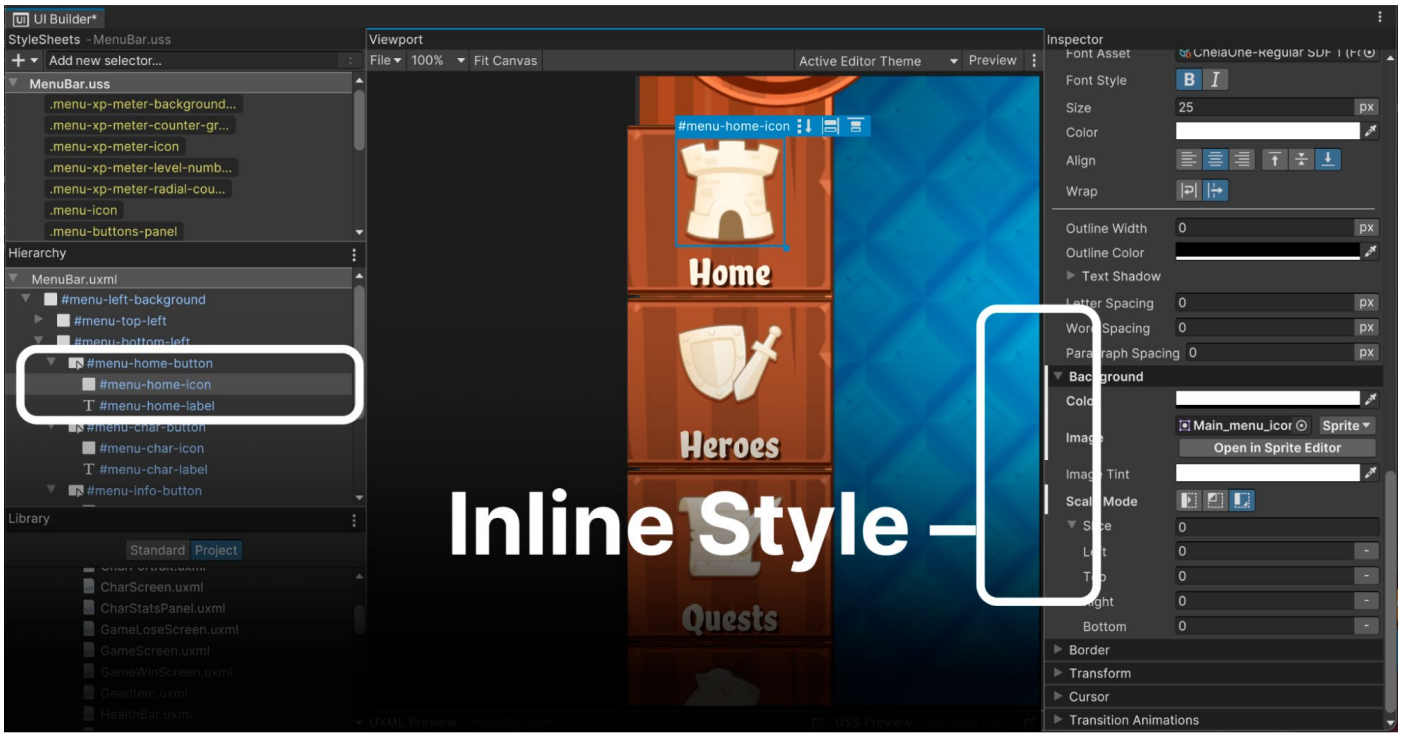
USS 파일은 요소의 크기, 컬러, 폰트, 간격, 테두리 또는 위치를 정의합니다.

USS 선택자

아직 USS 파일을 생성하지 않았다면, 앞으로 적용할 모든 스타일 변경 사항은 UXML 에셋에 인라인 스타일로 포함됩니다. 이러한 인라인 스타일은 연결된 시각적 요소의 외형에 영향을 주지만, 프로젝트 전반에서 재사용할 수는 없습니다.

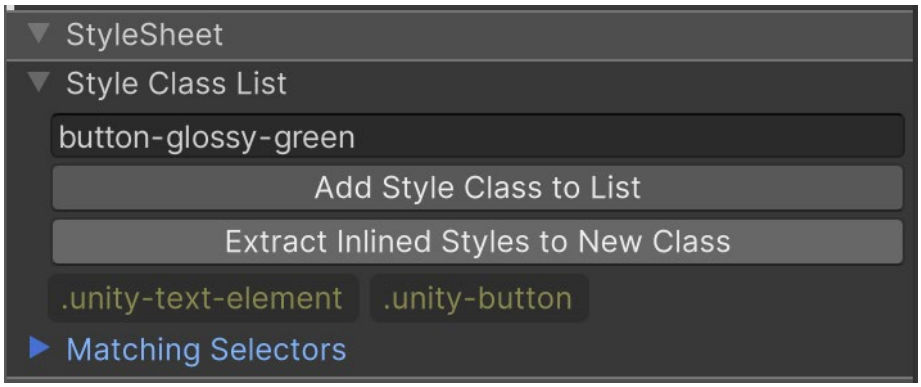
예를 들어 프로젝트에서 버튼이 수백 개인 경우, 개별 버튼의 스타일을 각각 업데이트하면 시간이 많이 소모되고 효율성도 떨어질 것입니다. 이렇게 하는 대신 USS에서 선택자를 정의할 수 있습니다. USS 선택자를 사용하면 UI 빌더의 스타일 시트로 UXML 에셋의 여러 요소 간에 스타일을 공유하고 적용할 수 있습니다.

기존 인라인 스타일을 선택자로 전환하기



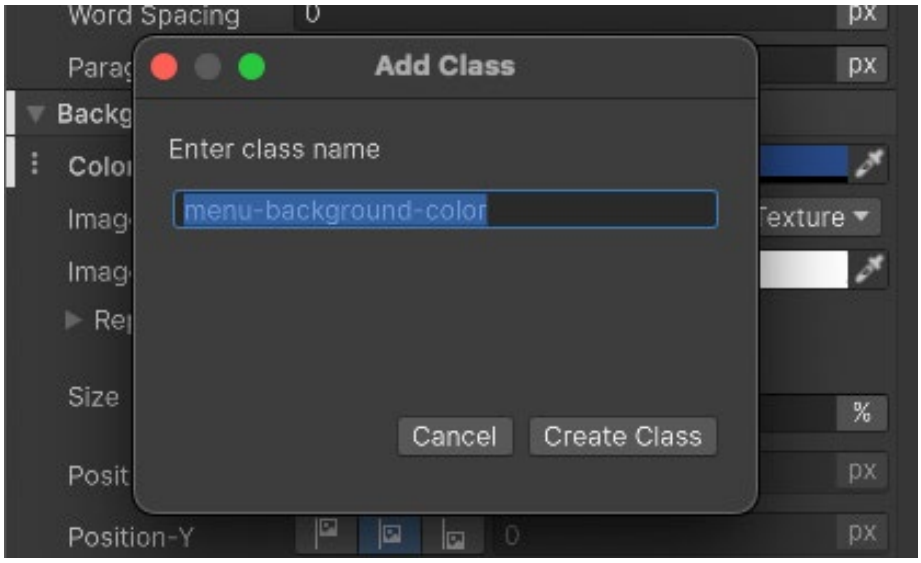
인라인 스타일은 오버라이드입니다.

Add Style Class to List 버튼을 사용하여 요소의 모든 인라인 스타일을 입력하고 선택자로 전환합니다('.'로 시작하는 노란색 텍스트). 이렇게 하면 스타일 프로퍼티가 이 선택자로 일원화되기 때문에 프로젝트 전반에서 다른 버튼(또는 요소)에 일관된 스타일을 적용할 수 있습니다. 선택자에 업데이트를 적용하면 자동으로 모든 연결된 요소에도 반영되므로 프로세스의 확장성이 향상되고 관리가 쉬워집니다.



모든 인라인 스타일 프로퍼티를 선택자로 추출하기

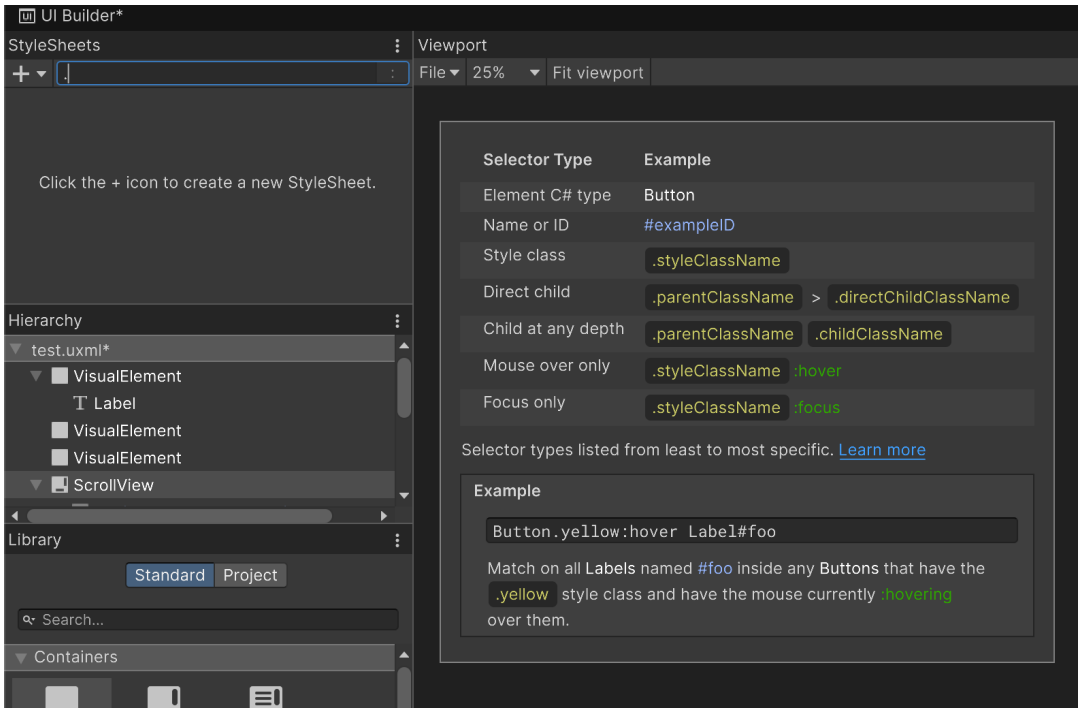
특정 인라인 스타일을 새 선택자로 추출하려면 프로퍼티 옆의 세로 줄임표( )를 클릭하고 **Extract Inlined Style to Selector/Add Class**를 선택합니다. 그러면 해당 프로퍼티가 선택자로 변환됩니다.



프로퍼티의 인라인 스타일을 선택자로 추출하기

새 선택자 생성

선택자는 시각적 트리에서 지정된 검색 조건과 일치하는 모든 요소를 쿼리합니다. 그러면 UI 툴킷은 일치하는 모든 요소에 스타일 변경 사항을 적용합니다. UI 빌더 왼쪽 상단의 **Add new selector...** 필드를 클릭하여 새 선택자를 추가할 수 있습니다.



새 선택자 생성 시의 USS 선택자 레퍼런스

USS 선택자는 다음을 기준으로 시각적 요소를 일치시킵니다.

— **Element C# type**: 이 선택자는 유형(Button, Label, ListView 등)을 기준으로 작동합니다. 선택자는 Library 패널의 기본 유형 이름에 해당합니다. 이름 앞에는 특수 문자를 포함하지 않습니다. 클래스 선택자는 흰색으로 표시됩니다. 예를 들어 **Button**은 유형이 Button인 모든 요소에 스타일을 적용합니다.

— **Name or ID**: 이 선택자는 이름이 같은 모든 요소에 스타일을 적용할 수 있습니다. Name 선택자는 앞에 '#' 해시 기호가 붙으며 파란색으로 표시됩니다. 예를 들어 **#title**은 계층 구조에서 이름이 title인 모든 요소에 스타일을 적용합니다.

참고: UXML 이름 속성은 HTML ID와 달리 고유할 필요가 없습니다. UI 툴킷은 UXML 템플릿 및 재사용 가능한 컴포넌트를 지원하므로 여러 요소가 동일한 이름과 스타일을 공유할 수 있기 때문입니다.

— **Style class**: Style class 선택자는 요소의 Class List 프로퍼티에 해당하는 클래스 이름을 추가하여 모든 시각적 요소에 적용할 수 있는 재사용 가능한 스타일입니다. Style class 선택자는 앞에 마침표 '.'가 붙으며 노란색으로 표시됩니다. 예를 들어 **.smallFont**는 Class List에 smallFont를 추가하여 모든 요소에 특정 스타일을 적용하는 용도로 사용될 수 있습니다.

— **Direct child:** 일치하는 기준 뒤에 **>** 기호를 추가하면 > 기호 뒤에 오는 두 번째 기준과 일치하는 직속 자식 요소만 영향을 받습니다. 예를 들어 **#title > Label** 선택자는 이름이 **#title**인 요소 내의 모든 **Label** 유형에만 스타일을 적용합니다. 해당 부모를 갖지 않는 모든 Label과 계층 구조에서 더 아래쪽에 종속된 모든 Label은 영향을 받지 않습니다.

— **Child at any depth:** 직전 선택자와 같되, 이 경우에는 두 번째 일치 기준이 부모 계층 구조에서의 뎁스와 관계없이 모든 자식 요소에 적용됩니다.

참고: 가능하다면 지나치게 광범위한 선택자는 사용하지 마세요(특히 ***로 끝나는 선택자 또는 .unity-button과 같이 제네릭 Unity 클래스를 타게팅하는 선택자). 계층 구조에서 더 아래쪽에 종속된 자식 선택자는 시각적 트리의 많은 부분에 대응할 때 성능을 저하할 수 있습니다.

— **유사 클래스:** 유사 클래스를 사용하면 시각적 요소의 상태가 변경된 경우(예: 요소 위에 마우스 커서를 올린 경우, 요소에 포커스가 있는 경우 등)에만 적용되는 스타일을 정의할 수 있습니다. 유사 클래스는 콜론 '**:**'으로 표시되며 기존 선택자를 수정합니다.

예를 들어 Button **:focus** 선택자는 모든 Button 요소에 포커스가 있는 경우에만 특정 스타일을 적용합니다. 유사 클래스는 마우스 커서 올리기 효과나 포커스 표시기 같은 시각적 피드백을 추가하려는 경우에 유용합니다. 유사 클래스를 USS 애니메이션과 함께 사용하면 매끄러운 모션과 동적인 전환을 적용하여 사용자 경험을 향상할 수 있습니다.

사용 가능한 유사 클래스에 대한 자세한 내용은 여기를 참조하세요.

하나의 시각적 요소가 여러 선택자와 일치하는 경우, 특이성이 가장 두드러지는 선택자가 우선합니다.

USS의 특이성 계층 구조는 다음과 같습니다.

1. **인라인 스타일**: 요소에 직접 적용된 스타일(예: UXML에서 또는 코드를 통해)은 우선순위가 가장 높으며 모든 USS 선택자를 오버라이드합니다.
2. **ID 선택자**(#id): 가장 구체적인 USS 선택자로, 고유한 이름 프로퍼티를 갖는 요소에 적용됩니다.
3. **클래스 선택자**(.className): 해당하는 클래스가 **Class List**에 추가된 요소에 적용됩니다.
4. **C# 유형 선택자**(예: Button, Label): 지정된 유형의 모든 요소에 적용됩니다.

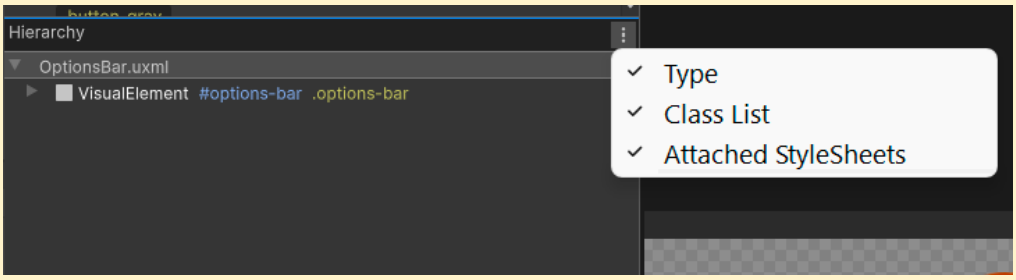
예를 들어 한 요소가 인라인 스타일이 있는 동시에 #title ID 선택자와 일치한다면, 인라인 스타일이 ID 선택자를 오버라이드합니다. 마찬가지로, 요소가 클래스 선택자 및 유형 선택자 모두와 일치한다면, 클래스 선택자가 우선합니다.

우선순위가 같은 경우(여러 선택자가 동일한 프로퍼티에 대한 오버라이드를 시도 중이며 모두 특이성 레벨이 동일하다면) USS 스타일 시트에서 정의되는 순서에 따라 우선순위가 정해집니다. 즉, 목록에서 더 아래쪽에 오는 선택자가 우선합니다.

선택자 우선순위에 관한 내용은 기술 자료에서 자세히 알아볼 수 있습니다.

팁: 계층 구조에 대한 추가 정보

계층 구조 헤더에서 세로 줄임표(:)를 클릭하여 UI 요소를 추가로 시각화할 수 있습니다.



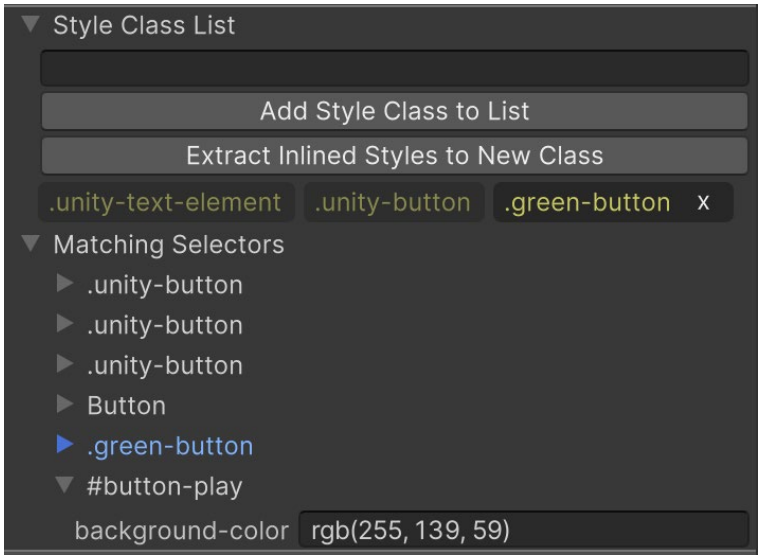
계층 구조에서 다양한 선택자를 필터링할 수 있습니다.

계층 구조 패널에서 요소 유형 옆에 추가 정보가 표시됩니다. 요소를 선택하면 **#options-bar** 이름 선택자와 **.options-bar** 스타일 클래스 선택자가 표시됩니다.

.unity-로 시작하는 선택자도 볼 수 있으며, 이러한 선택자는 모든 요소에 적용되는 기본 스타일을 나타냅니다. 정의된 선택자가 있으면 이 값이 오버라이드됩니다.

요소에 할당된 선택자

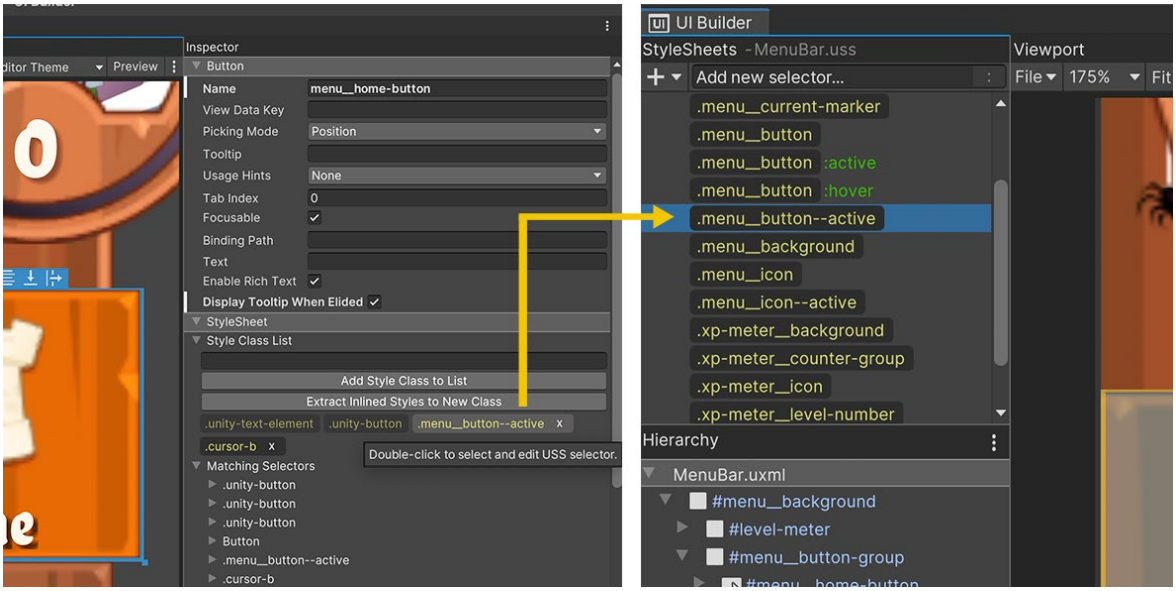
계층 구조에서 선택한 요소와 일치하는 선택자를 인스펙터에서 시각화할 수 있습니다. 목록에서 아래쪽에 있는 선택자가 우선권을 갖습니다. 세부 정보를 펼쳐 어떤 스타일 파라미터가 변경되는지 확인할 수 있습니다.



시각적 요소를 선택하면 인스펙터에서 일치하는 선택자가 표시됩니다.

선택자 편집

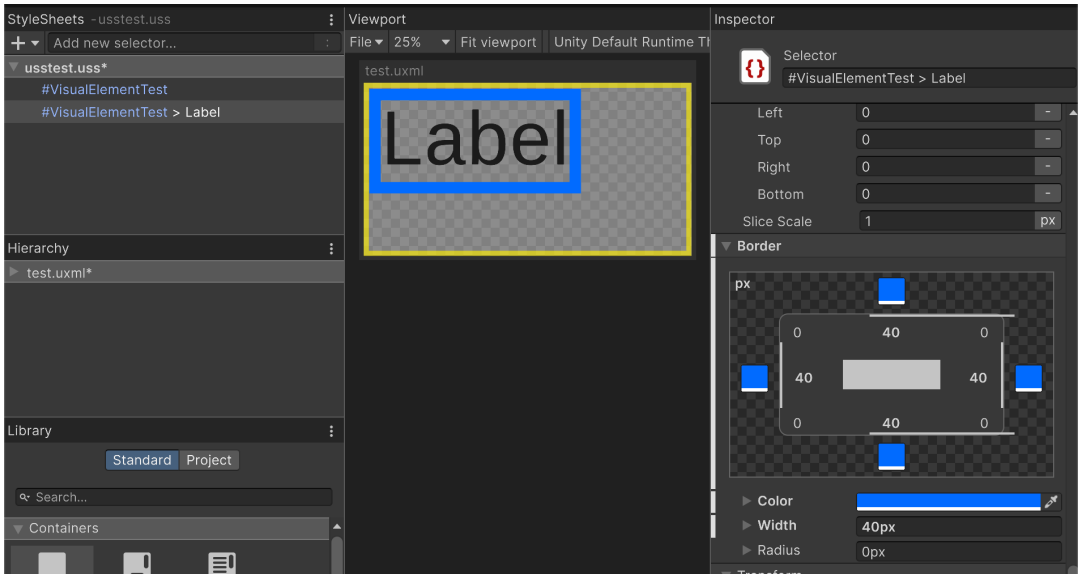
스타일 선택자를 수정할 때는 계층 구조에 있는 시각적 요소가 아니라 **Style Sheet** 패널에서 **스타일 클래스**를 선택해야 합니다. 계층 구조의 시각적 요소를 선택하면 스타일 클래스가 아닌 특정 요소의 인라인 스타일이 변경됩니다.



인스펙터에서 스타일 클래스를 더블 클릭해 활성화되었는지 확인합니다.

인스펙터에서 스타일 클래스를 더블 클릭해 요소를 선택 해제하고 대신 스타일 선택자를 선택할 수 있습니다.

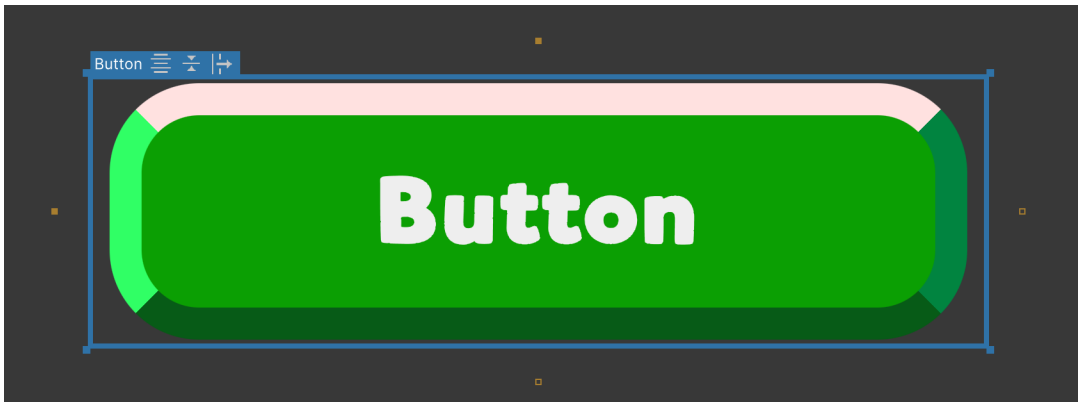
UXML에서 파라미터를 직접 인라인 스타일로 수정할 때처럼, 선택자의 파라미터를 StyleSheets 패널에서 선택하여 오버라이드로 수정할 수 있습니다. 변경 사항은 굵은 폰트와 왼쪽의 흰색 선으로 강조 표시됩니다. 프로퍼티 옆의 세로 줄임표(:) 메뉴에서 값을 설정 해제할 수 있습니다.



USS 선택자 편집하기

수많은 서식 옵션으로 요소와 폰트의 기본적인 외관을 수정할 수 있습니다. UI 툴킷은 커스텀 제작 스프라이트의 수요를 줄일 수 있는 고급 스타일링 기능을 제공합니다.

UI 빌더를 사용하면 요소에 윤곽선, 둥근 모서리, 이미지 조정, 테두리 컬러 등을 간편하게 추가할 수 있습니다. 굴곡 효과와 커서 이미지 변경 기능도 스타일링에 포함됩니다.



UI 툴킷은 추가 텍스처가 필요하지 않은 여러 스타일링 효과를 제공합니다.

스타일 오버라이드

규칙은 깨지기 마련입니다. UI 요소에 대한 스타일 클래스를 정의할 때도 언제나 예외 상황이 발생합니다.

예를 들어 수백 개의 버튼 요소가 있는데 각 요소의 아이콘이 서로 다르다면, 요소별로 선택자를 새로 만들 필요가 없습니다. 각 요소의 선택자를 만들면 스타일을 재사용 가능하게 만드는 목적(편의성)에 어긋납니다.

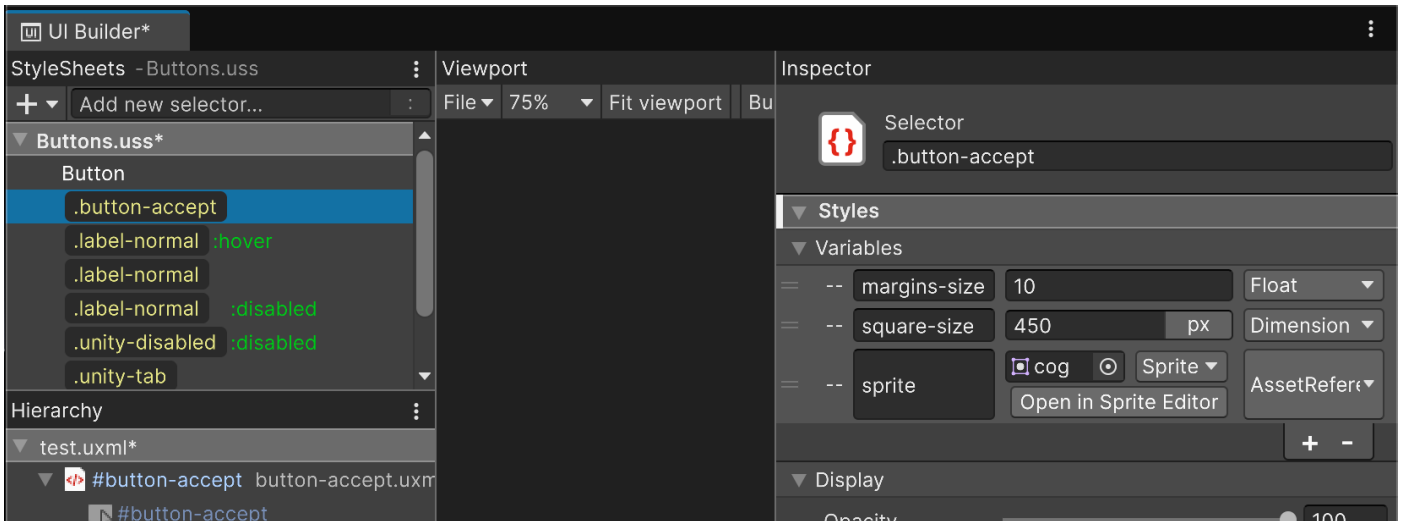
대신 모든 버튼에 동일한 스타일을 적용하고 각 버튼의 고유한 특정 부분을 오버라이드할 수 있습니다. 예를 들어 **Background > Image**에서 각 버튼이 고유한 아이콘을 사용하도록 버튼 요소를 오버라이드할 수 있습니다. 이러한 오버라이드를 **인라인 스타일** 프로퍼티라고 합니다.

팁: 선택자보다 인라인 스타일이 우선

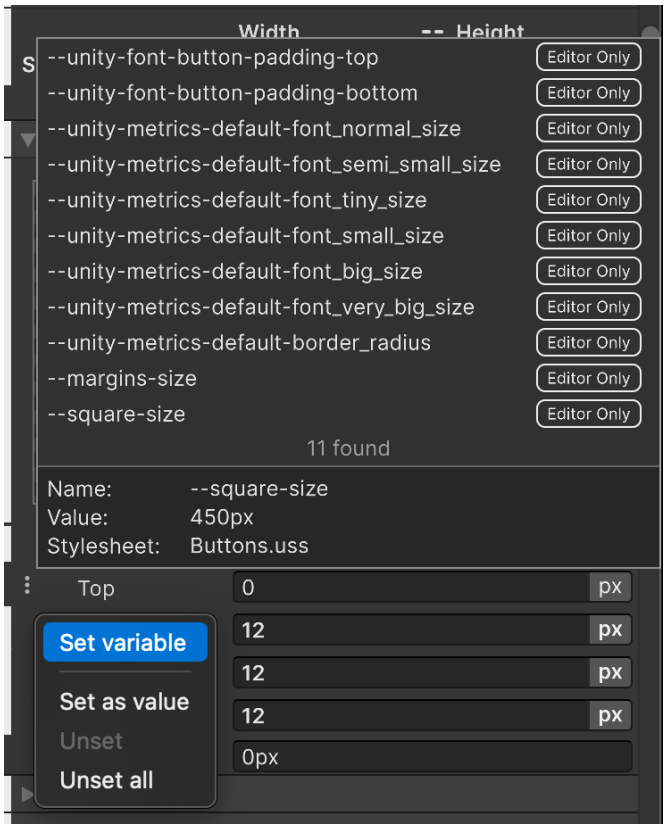
인라인 스타일은 항상 선택자보다 우선합니다. 따라서 선택자를 적용했을 때 스타일이 업데이트되지 않는 이유가 확실치 않다면 요소에 오버라이드가 있는지 확인해 보는 것이 좋습니다.

USS 변수

USS 변수를 생성하면 서로 다른 프로퍼티에서 같은 값을 수동으로 설정하는 시간을 단축할 수 있습니다. USS 변수를 업데이트하면 해당 변수를 사용하는 모든 USS 프로퍼티가 업데이트됩니다. Unity 6.1에서는 이러한 변수를 UI 빌더 에디터에서도 설정할 수 있습니다.



Unity 6.1에서는 UI 빌더에서 USS 선택자의 변수를 생성하고 편집할 수 있습니다.



값을 직접 입력하는 대신 프로퍼티 내에서 변수를 설정합니다.

변수는 float, color, string, asset reference(배경 이미지), dimensions(픽셀, 도, 백분율 등), enum 형식으로 생성할 수 있습니다. 변수는 선택자 레벨의 범위를 갖습니다. 다른 선택자에 있는 변수는 사용할 수 없지만, 선택자 자체는 필요한 만큼 많은 요소에 적용할 수 있습니다.

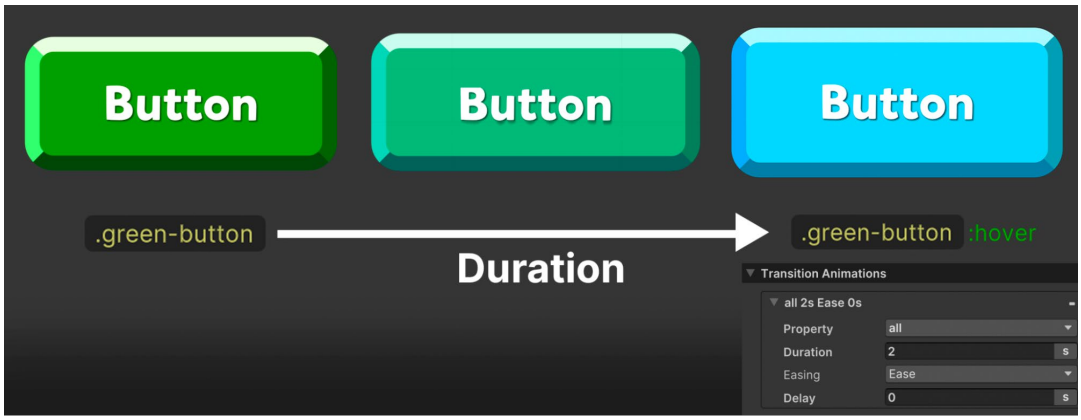
USS 전환 애니메이션

메뉴 화면에 전환을 추가하면 UI를 더 세련된 모습으로 다듬고 사용자 경험을 향상할 수 있습니다. UI 툴킷을 사용하면 인스펙터에서 Transition Animations 프로퍼티를 사용해 비교적 쉽게 전환을 추가할 수 있습니다.

애니메이션은 Property, Duration, Easing, Delay를 구성하여 설정할 수 있습니다. 구성을 마치면 런타임 중에 관련 스타일이 변경될 때 자동으로 전환이 적용됩니다.

.green-button 클래스 선택자에 **:hover** 유사 클래스가 있는 버튼의 유사 클래스 간 전환을 생각해 보세요. 각 스타일마다 고유한 크기와 컬러가 있습니다.

버튼 위에 마우스 커서가 올라간 상태의 전환을 정의하려면 **.green-button:hover** 선택자에서 인스펙터 하단에 있는 Transition Animations를 설정할 수 있습니다. 그렇게 하면 마우스 포인터의 움직임에 따라 애니메이션이 적용된 버튼이 만들어집니다.



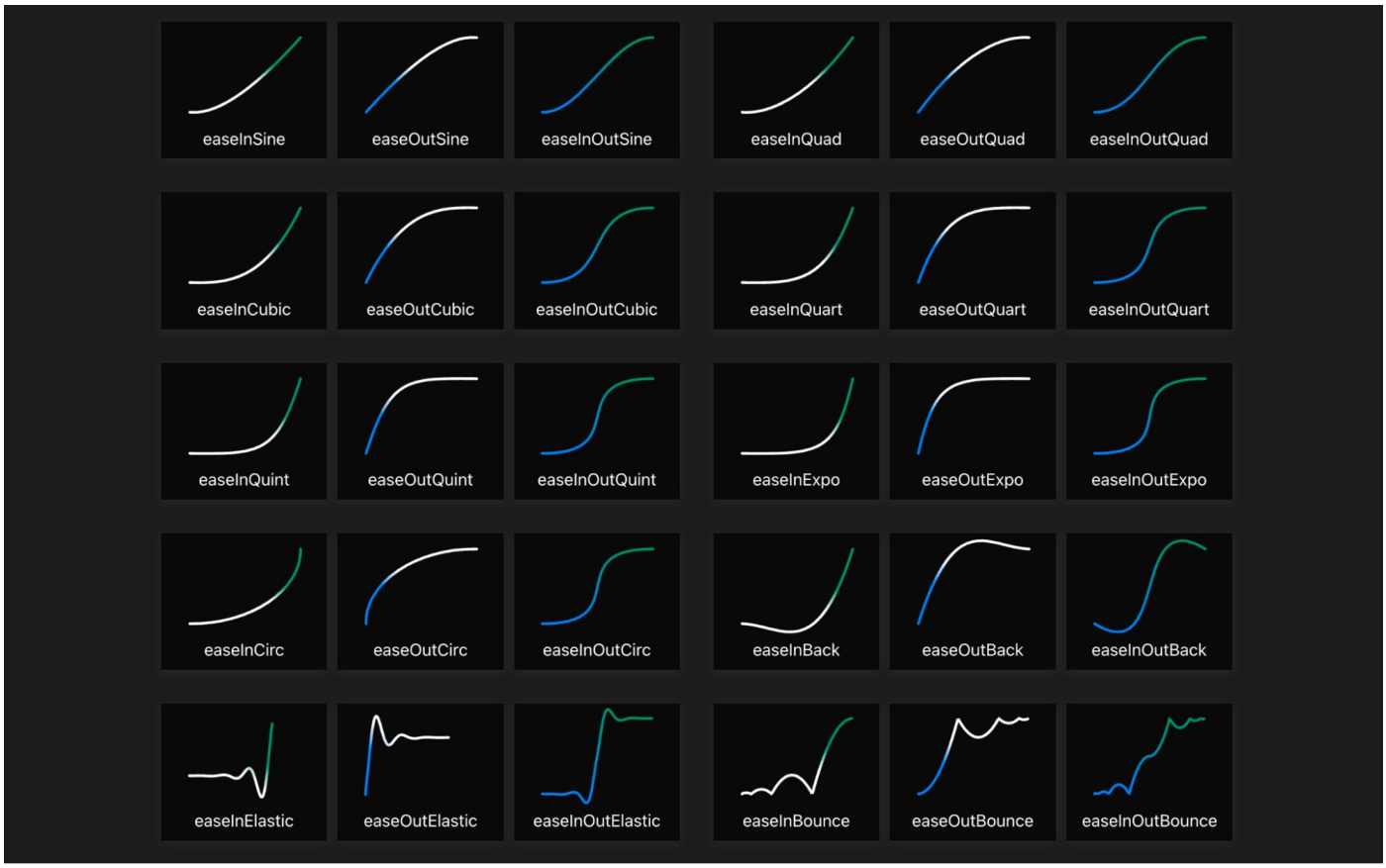
전환 애니메이션을 사용하면 스타일 사이를 보간할 수 있습니다.

전환 애니메이션은 다음 옵션을 사용하여 스타일 사이를 보간합니다.

— **Property**: 보간할 대상을 결정합니다. 기본 설정은 **all**이지만, 드롭다운 목록에서 특정 프로퍼티를 선택할 수 있습니다. 위 예시에서는 **:hover** 상태에 따라 **Color** 프로퍼티와 **Transform** 프로퍼티만 수정됩니다. 애니메이션 적용이 가능한 프로퍼티의 전체 목록을 살펴보세요.

— **Duration**: 전환의 길이를 나타내며 초 또는 밀리초 단위로 표시됩니다. 전환 애니메이션이 보이려면 Duration을 0보다 큰 값으로 설정해야 합니다.

— **Easing**: easing 함수는 시간의 흐름에 따른 애니메이션의 변화를 설정하며, 이를 사용하여 가속, 감속, 탄성과 같은 자연스러운 모션을 시뮬레이션할 수 있습니다. easing 함수를 사용하면 일정한 속도로 움직이는 기본적인 선형 보간에 비해 애니메이션 전환이 좀더 매끄럽고 자연스럽게 보입니다.



이 참고 자료는 사용 가능한 함수를 시각화하는 데 도움이 됩니다(시각화 이미지 제공: https://easings.net/).

— **Delay:** 초 또는 밀리초 단위로 정의되며, 전환을 시작하기 전에 대기할 시간을 지정합니다.

— **Add Transition:** 새로운 상태의 각 프로퍼티가 다양한 재생 시간, 지연, 이징(easing) 효과가 적용된 개별적인 애니메이션이 될 수 있습니다.

Add Transition 버튼을 클릭해 다른 전환 애니메이션을 연결하세요. 그러면 중첩된 전환 여러 개를 동시에 트리거할 수 있으므로 더 자연스럽고 유기적인 전환 애니메이션을 만들 수 있습니다.

팁: 전환 이벤트

애니메이션화할 시각적 요소에 전환 이벤트에 대한 콜백을 추가할 수 있습니다. 콜백은 시퀀싱 또는 루핑 등 고급 워크플로를 지원하는 역할을 합니다.

다음은 몇 가지 일반적인 전환 이벤트와 해당 이벤트가 전송되는 시점에 대한 설명입니다.

- TransitionRunEvent: 전환이 생성되면 전송
- TransitionStartEvent: 전환의 지연 단계가 끝나고 전환이 시작되면 전송
- TransitionEndEvent: 전환이 종료되면 전송
- TransitionCancelEvent: 전환이 취소되면 전송

기술 자료에서 USS 전환에 대해 자세히 알아보세요.

시각적 요소의 경우, 유사 클래스(`:active`, `:inactive`, `:hover` 등)가 자체 선택자를 가질 수 있으므로 추가 코딩 없이 애니메이션을 적용할 수 있습니다. 유사 클래스가 스타일 변경을 트리거할 때마다, 정의된 전환이 자동으로 해당 변경 사항에 애니메이션을 적용합니다. 예를 들면 다음과 같습니다. 마우스 커서를 올리거나(`:hover`) 클릭했을 때(`:active`) 버튼이 커지거나 작아질 수 있고, 사용자 상호 작용이나 그 밖의 이벤트로 인해 요소가 페이드아웃되거나 보이지 않게 될 수 있습니다.

유사 클래스는 사전 정의되며, 자체적으로 만들 수 없습니다.

필요에 따라 스타일 변경

게임의 다른 모든 이벤트에 대해, UI Element API의 메서드를 사용하여 코드 내에서 스타일을 변경할 수 있습니다. 예를 들어 캐릭터 희귀도에 따라 다른 스타일로 변경하려면 `RemoveFromClassList` 메서드와 `AddToClassList` 메서드를 사용할 수 있습니다.

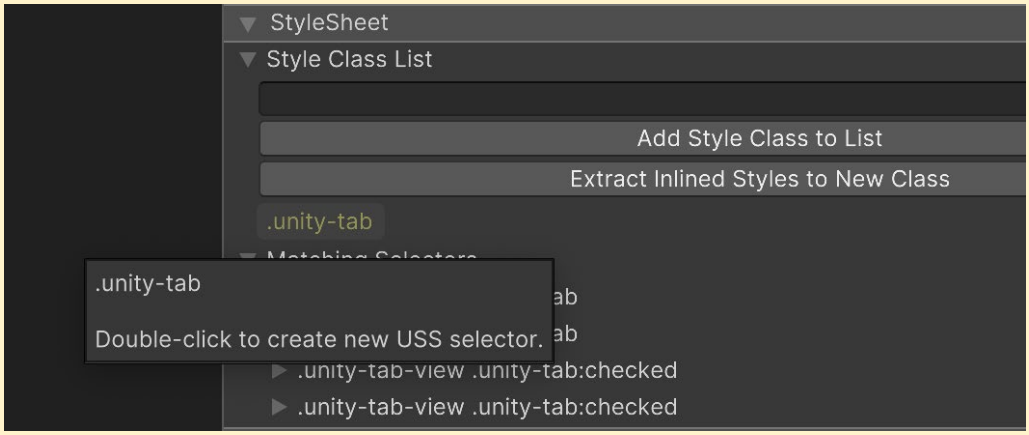
```
if (character.rarity == RarityType.Legendary)
{
   visualElement.RemoveFromClassList("common");
   visualElement.AddToClassList("legendary");
}
```

또한 시각적 요소의 활성화 상태를 기준으로 하는 유사 클래스 `:active` 또는 `:inactive`를 추가로 트리거하여, 상태 변경 시 USS 전환이 적용되도록 할 수 있습니다. 이전 상태와 이후 상태를 표현해 보세요.

UI 툴킷 샘플 – 드래곤 크래셔의 메뉴 바 버튼은 PointerEventClick을 사용하여 수동 전환을 트리거합니다.

팁: Unity 기본 선택자 오버라이드

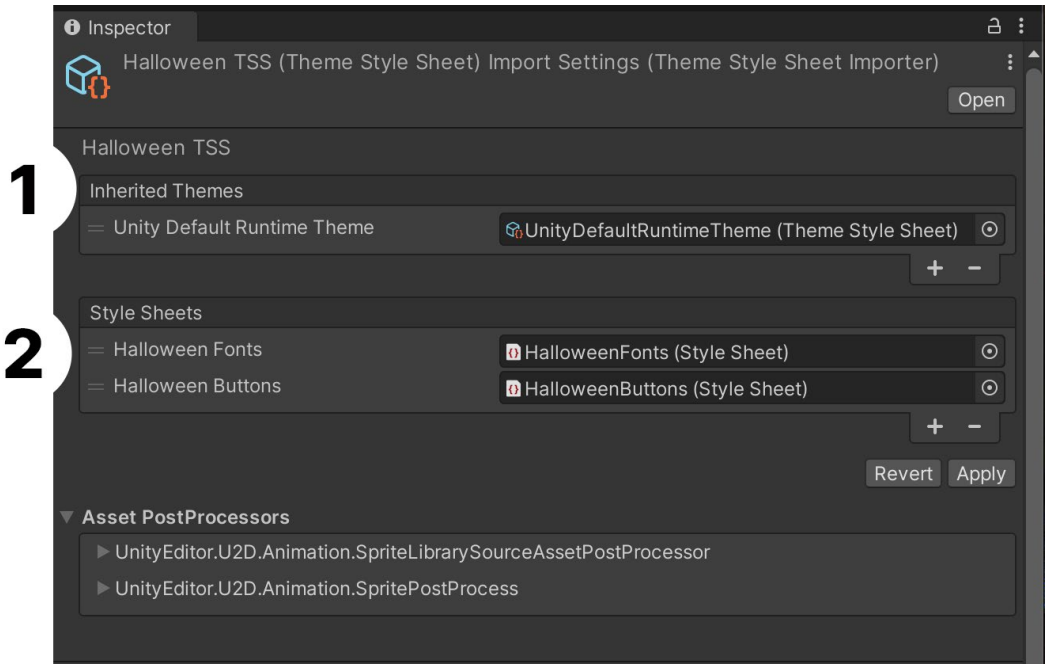
탭 뷰처럼 더욱 복잡한 시각적 요소는 하나의 부모 요소와 시스템에 의해 사전 정의된 다수의 자식 요소로 구성됩니다. 이러한 요소에 콘텐츠를 추가하면 특정한 방식으로 동작하며, 사용된 스타일은 비활성화되고 Unity 스타일이 적용된 것처럼 보입니다. 사용 중인 선택자를 더블 클릭하고 복사하여 스타일 시트(USS)에서 편집하는 방법으로 이러한 기본 선택자를 오버라이드할 수 있습니다.



테마

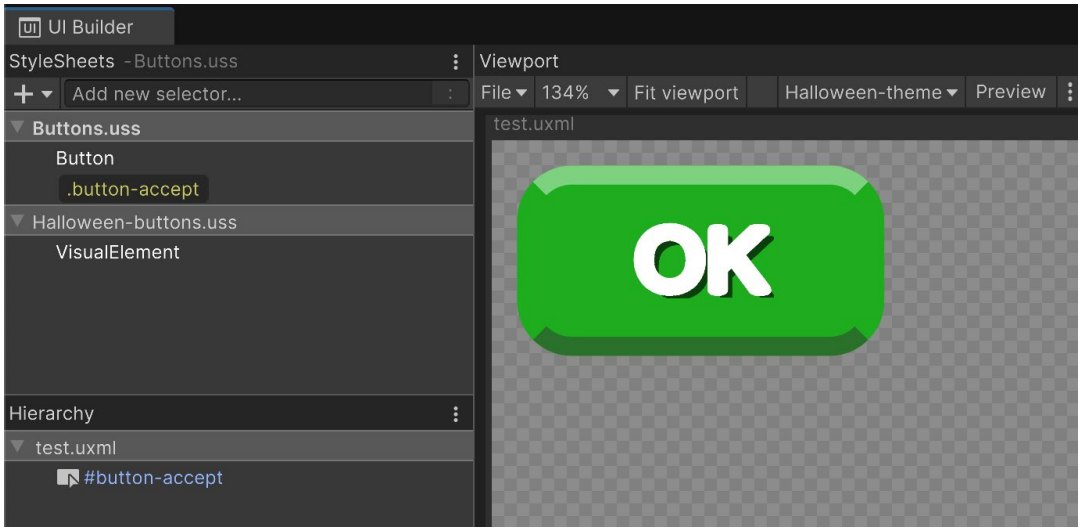
시즌에 따라 바뀌는 UI 버전을 만들거나 다양한 컬러 스타일을 제공하려는 경우, TSS(테마 스타일 시트)를 사용해 프로세스를 간소화할 수 있습니다. TSS는 **Create > UI Toolkit > TSS theme file을 통해 생성합니다.**

TSS 파일은 일반적인 USS 파일처럼 작동하는 에셋 파일입니다. TSS 파일을 사용하면 USS 선택자와 프로퍼티 및 변수 설정으로 구성된 커스텀 테마를 정의할 수 있습니다.



이 할로윈 테마 UI 요소 예시에서 Halloween TSS는 Unity Default Runtime TSS를 상속한 다음, Fonts와 Buttons에 대해 테마별 스타일 시트를 추가합니다.

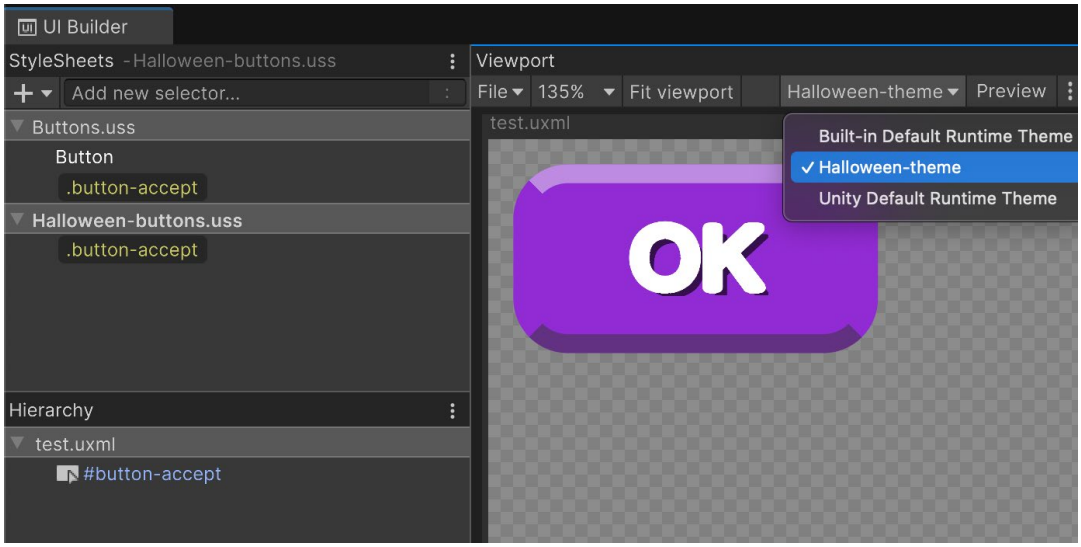
새로운 테마가 상속된 테마라면 원본 테마에 있는 선택자가 스타일 시트에서 누락되어 있어도 원본 테마의 스타일이 적용됩니다. 따라서 커스터마이징이 한결 간편해집니다. 예를 들어 나머지 UI(예: 컬러, 패딩, 테두리) 스타일은 원본 테마를 유지한 채 폰트만 수정된 새로운 테마를 생성할 수도 있습니다. 이 방식은 밝은/어두운 모드, 캐릭터별 UI 커스터마이징을 구현할 때나 게임별 이벤트 테마를 생성할 때 유용합니다.



이 스크린샷에 보이는 할로윈 테마의 TSS는 **Halloween-buttons.uss**를 사용하지만, 버튼이 사용 중인 선택자 **.button-accept**와 일치하는 선택자가 없으므로 원본 테마에 적용된 선택자가 사용됩니다.

기존 테마를 기반으로 새로운 테마를 생성하는 워크플로는 다음과 같습니다.

1. 새 TSS를 생성하고, 상속받을 테마와 이 새 TSS가 사용할 새 USS 파일을 추가합니다.
2. **UI Builder > StyleSheets**에서 **Add Existing USS**를 클릭하고 새 테마가 사용할 USS를 선택합니다.
3. 새 테마가 오버라이드할 선택자를 복사합니다.
4. 새 테마가 사용하는 USS에 붙여 넣은 다음 오른쪽 클릭하고 **Set as Active USS**를 선택합니다.
5. 새 USS에서 선택자를 편집합니다.
6. UI 빌더의 드롭다운 목록에서 각 테마가 사용하는 스타일을 볼 수 있습니다.



UI 빌더 뷰포트에서 어느 테마를 적용할지 선택합니다.

런타임의 경우 **Panel Settings 인스펙터**의 **Theme Style Sheet** 필드에서 새 테마를 참조하세요.

명명 규칙

UI 툴킷을 사용하면 문자열 식별자를 사용하여 시각적 요소와 USS를 쿼리해야 하므로, 표준 방식을 정의해 두면 전반적으로 오류를 줄이고 더 가독성이 뛰어난 코드를 작성할 수 있습니다.

개발 팀은 인터페이스를 구성하는 동일한 UXML 및 USS 에셋을 참조하므로, 시각적 요소와 스타일 시트의 명명 규칙을 표준화하는 것이 중요합니다. 명명 규칙은 UI 빌더에서 계층 구조를 체계적으로 유지하는 데 도움이 됩니다. 그뿐만 아니라 코딩 규칙과 포맷 지정 규칙에서 불확실한 요소가 제거되고, 일관성 있는 코드베이스를 마련할 수 있습니다.

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

시각적 요소의 이름은 코드에서 해당 요소에 대한 참조를 저장하는 데 사용됩니다.

모든 경우에 적합한 만능 스타일 가이드는 존재하지 않습니다. 팀과 프로젝트에 가장 적합한 방식을 찾아보세요. 하지만 가능한 한 업계 표준을 준수하는 것이 좋습니다. 따라서 시각적 요소와 스타일 시트에 **BEM(Block Element Modifier)** 명명 규칙을 사용할 것을 권장합니다. BEM은 UI 툴킷 탄생의 배경이 된 CSS 및 최신 웹 개발 컨텍스트에서 널리 사용됩니다.

요소의 BEM식 이름을 보면 해당 요소의 기능, 사용되는 곳, 주변 다른 요소와의 관계를 한눈에 파악할 수 있습니다. BEM은 다음과 같은 규칙을 기반으로 세 가지 주요 컴포넌트를 사용합니다.

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

다음은 한 가지 예입니다.

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

각 이름 부분은 라틴 문자, 숫자, 대시 기호로 구성될 수 있습니다. 각 이름 부분은 이중 밑줄(__) 또는 이중 대시(--)로 연결됩니다. 세 개의 요소를 하나씩 살펴보겠습니다.

- 블록 이름(`block-name`)은 레이아웃에서 뚜렷하게 구분되는 중요한 UI 컴포넌트인 탐색 메뉴나 캐릭터 스탯과 같은 고수준 컴포넌트를 나타냅니다. 특정 블록 전용으로 사용되지 않는 일반적인 버튼은 명시하지 않아도 됩니다(예: `button--small`).
- 요소(`element-name`)는 블록의 자식 요소이거나 일부이므로, 의미상 해당 블록과 연관되어 있습니다. 다시 말하면, 요소는 블록 없이 존재할 수 없고 블록에 따라 컨텍스트가 정해집니다. 따라서 shop-button 예시에서 이 버튼은 `navbar-menu` 블록에 속하는 다른 버튼과 다른 스타일을 갖는 것을 알 수 있습니다(예: `navbar-menu__shop-button`의 `shop-button`).

  새 요소가 생성자에서 하위 요소를 인스턴스화한다면 하위 요소에 관련 클래스를 할당합니다. (예: `my-block__first-child`, `my-block__other-child`)

- 마지막으로, 한정자는 블록 또는 요소의 배리에이션 또는 상태를 나타냅니다. 버튼이 눌린 경우, 텍스트 상자 항목이 선택되어 강조 표시된 경우, 또는 앞의 예시처럼 shop 버튼의 작은 배리언트인 경우일 수 있습니다. 이 방식을 사용하면 코드를 복제하지 않고도 다양한 시나리오에 쉽게 대응할 수 있습니다.

아래에서 몇 가지 BEM 명명 예시를 살펴보세요.

- `menu__button-home`
- `menu__button-shop`
- `navbar-menu__shop-button--small`
- `navbar-menu__shop-button--large`

BEM 클래스 이름은 이름 자체에 설명을 포함하므로 개발자들이 컴포넌트의 구조와 목적을 쉽게 파악할 수 있습니다. 따라서 프로젝트의 규모가 커지더라도 명확한 계층 구조를 바탕으로 더 쉽게 스타일을 관리하고 업데이트할 수 있습니다. 대체로 간결성보다는 가독성을 우선시하는 편이 바람직합니다. 몇 개의 모음을 생략해 시간을 절약하는 것보다 명확성이 더 중요합니다.

이 예시에서는 CSS 명명에 흔히 사용되는 케밥 표기법(Kebab case)이라고도 하는 하이픈 구분 방식을 사용합니다. 프로젝트 초기에 어느 명명 규칙을 사용할지 정하고, 개발 기간 동안 이를 준수하는 것이 중요합니다.

이 문서와 UI 툴킷 기술 자료에서 CSS 명명 규칙에 관해 자세히 알아보세요.

팁: UI 툴킷의 명명 규칙

다음은 효과적인 명명 규칙을 위한 몇 가지 가이드라인입니다.

— 모호하지 않고 짧으면서 명확한 이름을 지으세요. 해당 요소가 UI에서 갖는 목적과 역할을 충분히 전달하는 간결하면서도 구체적인 이름을 사용하세요.

— 역할과 관계를 강조하는 이름을 사용하세요. 예를 들어 `inventory__button--equipped` 보다 `inventory__slot--equipped`가 바람직합니다. Button, Label 등의 유형 이름은 명확성을 높이는 효과가 없다면 생략하세요.

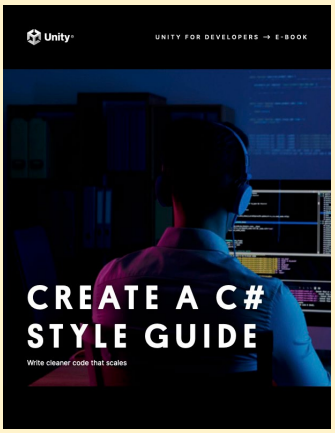
— 바뀔 수 있는 이름/수식어는 피하세요. 예를 들어 컬러 체계가 아직 확정되지 않은 상태라면 'button-red'보다 'button-quit'이 바람직합니다. 표현 중심의 이름 대신 의미 중심의 이름을 사용하면, 스타일 세부 사항이 변경되더라도 이름은 계속 의미에 맞게 유지됩니다.

— 명명 규칙을 UI 툴킷 인터페이스와 관련 스프라이트 및 텍스처 등 아트 에셋으로 확장하세요. 코드와 에셋의 이름을 일관되게 지정하면 명확한 관계를 나타낼 수 있고 프로젝트를 진행하는 동안 더 나은 체계를 유지할 수 있습니다.

— 요소를 다른 프로젝트에서 사용한다면 기존 사용자 클래스 이름과 충돌하지 않도록 클래스에 접두사를 추가하세요. 네임스페이스나 접두사를 추가하면 다른 프로젝트 또는 라이브러리와 통합할 때 충돌을 피할 수 있습니다.

— 요소 인스턴스에 관련 USS 클래스를 추가하려면 생성자에서 AddToClassList()를 사용하세요. 이 메서드를 사용하면 요소가 인스턴스화되는 시점에 필요한 클래스가 추가되어 적절한 스타일이 적용되고 UI 코드에서 일관성과 명확성이 유지됩니다.

C# 스타일 가이드 만들기



팀 차원에서 주요 코드 사용 방식을 개선해 프로젝트의 확장성을 높이고 싶다면, 무료로 제공되는 전자책 **C# 스타일 가이드 만들기: 깔끔하고 확장 가능한 코드를 쓰는 방법**을 확인해 보세요. 이 가이드를 적절히 활용하면 코드 스타일과 명명 규칙을 더 쉽게 표준화할 수 있습니다.

전자책 다운로드

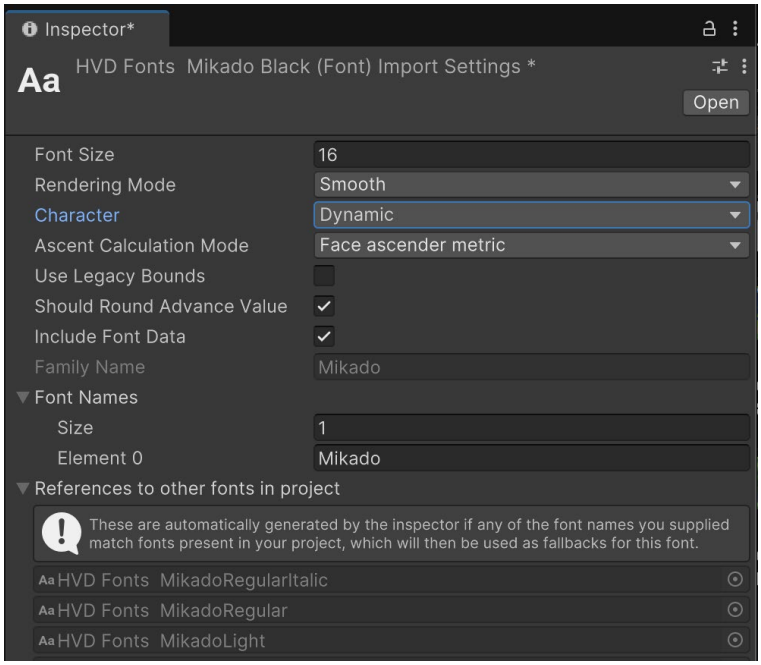
텍스트

UI 툴킷은 레거시 UI 시스템인 Unity UI에서 사용되는 **TextMesh Pro** 기반의 폰트 렌더링 기술인 **TextCore**를 사용합니다. TextCore는 고급 스타일 기능을 제공하며, 다양한 포인트 크기와 해상도에서 텍스트를 깔끔하게 렌더링할 수 있습니다. **SDF(Signed Distance Field)** 폰트 렌더링을 활용하므로, 변환되거나 확대된 경우에도 선명하게 보이는 폰트 에셋을 만들 수 있습니다. TextCore의 여러 렌더링 모드에 대한 자세한 내용은 기술 자료를 참조하세요.

다양한 폰트 에셋 유형과 그 용도에 대해 살펴보겠습니다.

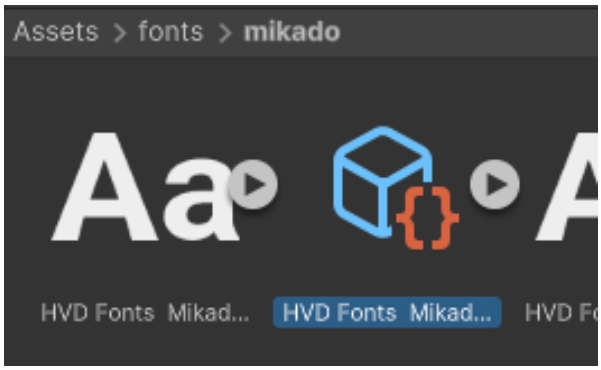
소스 폰트 파일

가장 일반적인 폰트 포맷인 **TTF**와 **OTF** 파일을 Unity 프로젝트에서 사용하려면 먼저 **폰트 에셋**으로 전환해야 합니다. 폰트 에셋은 문자 글리프, 폰트 지표, 렌더링 구성(크기, 두께, 스타일 등)을 비롯해 폰트를 렌더링하는 데 필요한 데이터가 들어 있는 Unity 전용 리소스입니다. 임포트된 소스 파일에서 각 폰트 패밀리와 렌더링 옵션에 대한 정보를 볼 수 있습니다.



이러한 임포트 옵션 중 상당수는 Unity UI의 레거시 텍스트 시스템으로 인해 남아 있는 옵션이며, 추후 릴리스에서 삭제될 예정입니다. Rendering Mode, Character, Include Font Data는 폰트 에셋을 생성하는 데 사용됩니다.

해당하는 폰트 에셋을 생성하려면 소스 폰트 파일을 선택하고 Assets 메뉴를 오른쪽 클릭한 다음 **Create > Text Core > Font Asset > SDF**를 선택합니다(원하는 렌더링 모드가 SDF인 경우).



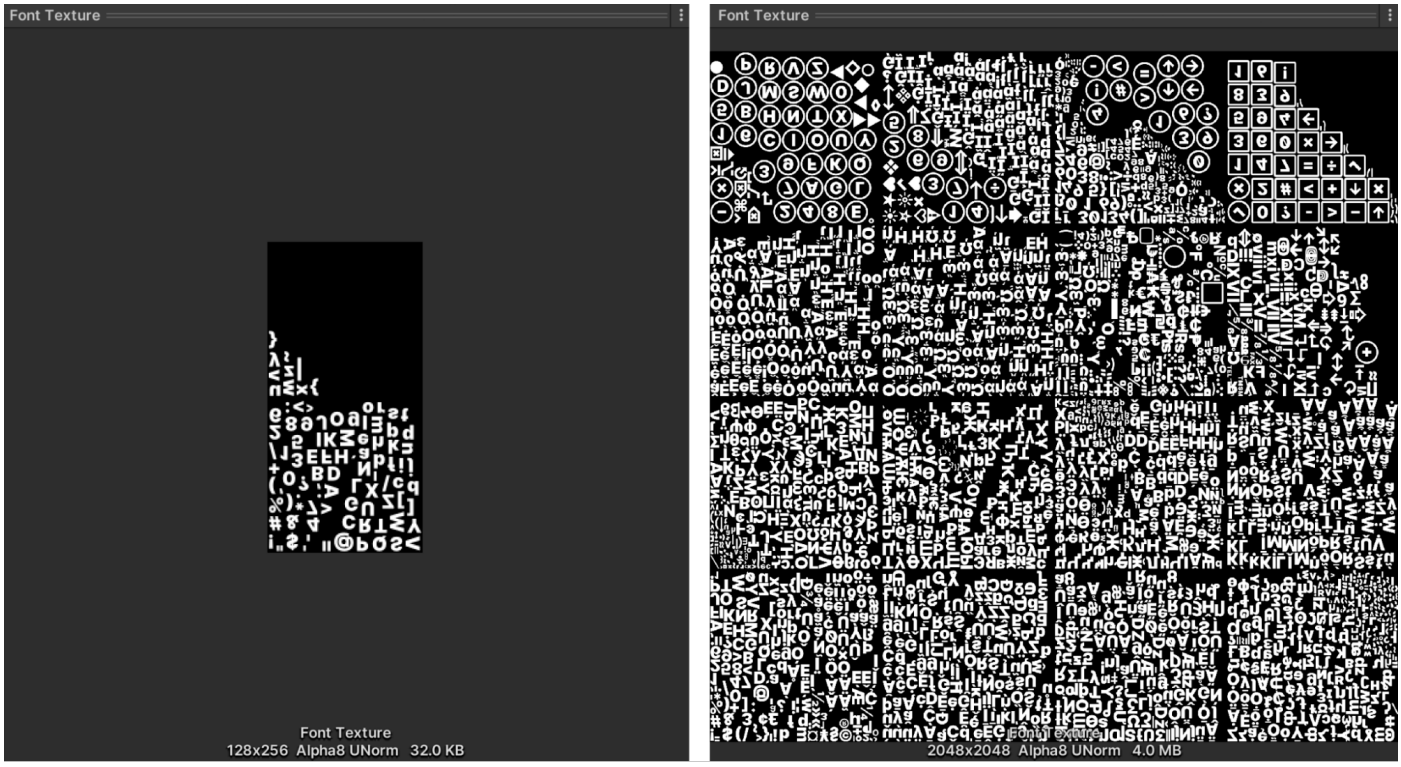
UI 시스템마다 서로 다른 폰트 에셋을 사용합니다.

폰트 에셋 설정

소스 폰트 파일을 전환했으면, 해당 폰트 에셋을 선택하여 폰트 생성을 관리하는 데 필요한 모든 옵션을 볼 수 있습니다. 몇 가지 주요 옵션을 살펴보겠습니다. 폰트 에셋에 대한 자세한 내용은 기술 자료를 참조하세요.

— **Face Info:** 기본 소스 폰트를 조정해야 하는 경우 애플리케이션에 적합하게 파라미터를 수정할 수 있는 폰트 간격 조정 및 스케일링 옵션입니다.

- **Generation Settings**: 소스 폰트, 소스 폰트가 여러 스타일을 포함하는 경우 폰트 에셋에 사용할 폰트 페이스, Atlas Population 모드, 렌더 모드 등의 기본 구성입니다.
- **Atlas and Material:** 생성된 머티리얼과 텍스처로, 아틀라스가 정적인지 동적인지, 렌더링 모드가 비트맵인지 SDF인지에 따라 문자 집합이 큰 언어를 지원할 경우 생성되는 아틀라스의 크기를 제어할 수 있습니다.
- **Font Weights:** 소스 에셋에 대응하는 배리에이션이 없는 경우 여러 폰트 두께를 시뮬레이션합니다.
- **Fallback Font Asset:** 현재 폰트 에셋에 특정 문자 또는 글리프가 없는 경우 사용할 대체 폰트를 제공합니다.
- **Character and Glyph Tables:** 폰트 에셋에 포함된 모든 문자와 글리프로 구성된 상세한 목록입니다.
- **Ligature table:** 두 개의 문자가 연속으로 나타나는 경우 사용할 글리프를 추가합니다(가독성과 시각적 플로 개선).
- **Glyph Adjustments:** 각 문자 또는 글리프의 오버라이드를 정의합니다.



소스 폰트와 아틀라스로 인해 빌드 크기가 커질 수 있습니다. 왼쪽은 ASCII 문자를 사용하는 아틀라스이고, 오른쪽은 완전한 유니코드 문자 집합을 사용하는 아틀라스입니다.

인스펙터 상단에서 폰트 에셋을 선택하면 **Update Atlas Texture** 버튼 아래에서 **Font Asset Creator**를 볼 수 있습니다. 여기에서 아틀라스 프로퍼티를 채우고 정의하는 데 필요한 모든 컨트롤을 찾을 수 있습니다.

팁: 패딩 및 아틀라스 해상도

폰트 텍스처의 문자에는 각 문자를 개별적으로 렌더링하기 위해 픽셀 단위로 지정된 약간의 패딩이 필요합니다. 패딩이 있으면 SDF 그레디언트에 필요한 공간도 마련됩니다. 패딩이 크면 그만큼 전환이 부드러워지므로 고급 렌더링과 두꺼운 윤곽선 같은 효과를 구현할 수 있습니다.

ASCII 문자만 사용하는 경우에는 대부분의 폰트에서 아틀라스 해상도를 512x512로 설정하고 패딩을 5로 지정하면 충분합니다. 더 많은 문자를 사용하는 폰트라면 더 큰 해상도를 사용하거나 여러 아틀라스를 사용해야 할 수 있습니다. 일반적으로 패딩 크기와 샘플링 크기의 비율을 1:10으로 맞추는 편이 좋습니다.

폰트 에셋 배리언트

새로운 폰트 아틀라스를 사용하지 않고 기존 아틀라스를 변경하려면 **Create > Text Core > Font Asset Variant에서 폰트 에셋 배리언트를 만듭니다.** 이 배리언트에는 다른 버전의 폰트 선 지표가 있을 수 있습니다.

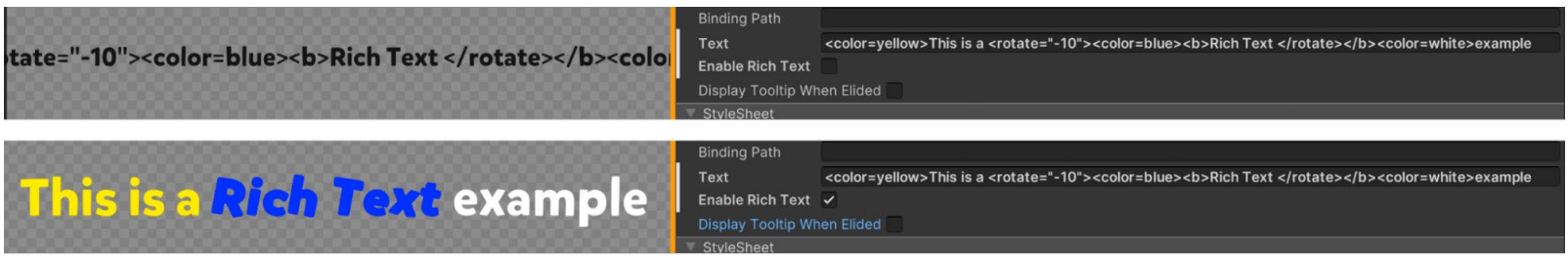
배리언트는 선 높이, 첨자 위치 등 자체 Face Info 설정을 가지면서도 원본 아틀라스를 참조합니다. 따라서 텍스처용 공간을 추가로 사용하지 않고도 원본 폰트 에셋과 구별되는 고유한 스타일을 지정할 수 있습니다.

리치 텍스트

리치 텍스트 태그는 텍스트 필드에 추가 태그를 사용하여 텍스트의 외형과 레이아웃을 변경합니다. 시각적 요소 및 코드 UI 빌더 양쪽 모두에서 리치 텍스트 태그를 사용할 수 있습니다. 태그를 사용하면 런타임에 텍스트의 포맷을 지정할 수 있습니다(예: 사용자 이름의 외형을 커스터마이즈).

리치 텍스트 태그는 텍스트의 프로퍼티나 스타일을 수정하지 않고도 텍스트의 컬러나 얼라인먼트를 변경할 수 있습니다. 리치 텍스트를 사용하여 다이얼로그 시스템의 텍스트 서식을 지정하거나 전달하려는 내용을 시각적으로 강조할 수 있습니다.

Extra Settings로 이동하여 UI 빌더에서 리치 텍스트 기능을 활성화하면 태그를 포함한 텍스트 서식을 적절하게 지정할 수 있습니다. 예를 들어 태그와 닫는 태그 사이의 텍스트는 굵은 글씨로 표시됩니다.

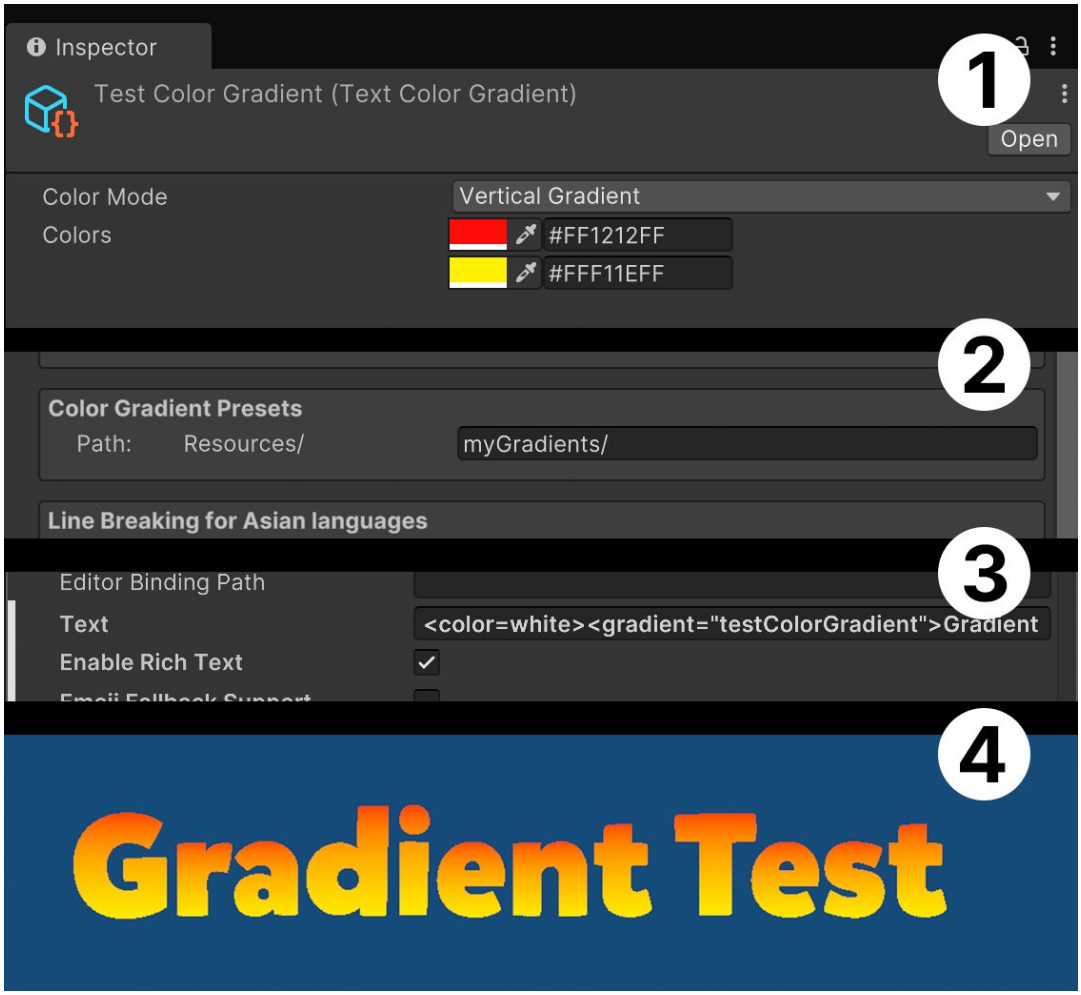


태그로 시각적 텍스트 프로퍼티를 수정할 수 있도록 UI 빌더에서 리치 텍스트 태그를 활성화합니다.

사용 가능한 리치 텍스트 태그와 파라미터의 전체 목록을 확인하세요.

그레디언트

그레디언트는 인터페이스 전반에 스타일을 더하며, UI 툴킷에서 <gradient> 태그를 사용하여 적용할 수 있습니다. 아래의 단계에 따라 간단한 그레디언트를 생성해 보세요.

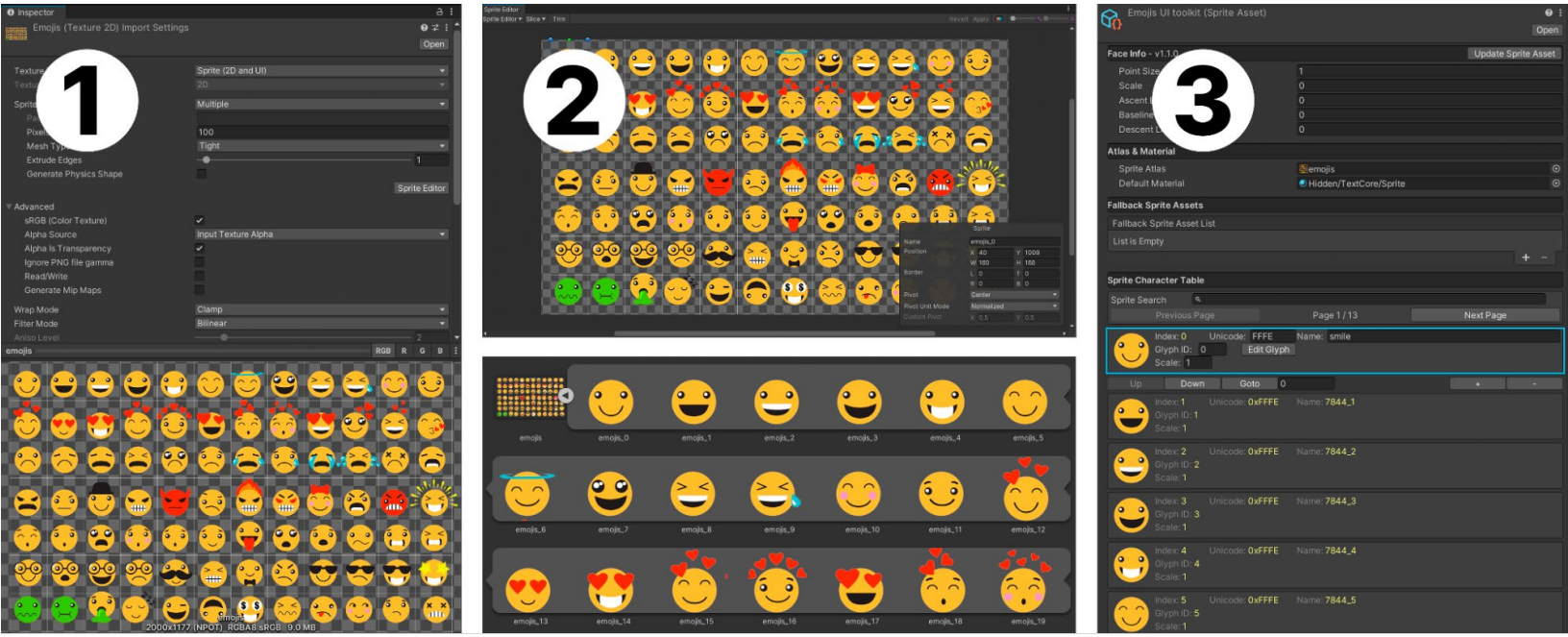


1. **Create > Text Core > Gradient Color**를 통해 그레디언트 컬러 에셋을 생성합니다. 이 파일은 **Assets/Resources** 나 Resources 아래의 하위 폴더에 넣어야 합니다.
2. Panel Settings에서 참조할 Text Settings 에셋을 생성합니다. 해당 에셋에서 Color Gradient Presets를 찾고 에셋이 있는 Resources 아래의 폴더나 하위 폴더를 지정합니다.
3. UI 빌더 내에서 `<color=white><gradient="testColorGradient">Gradient Test</gradient></color>` 리치 텍스트 태그를 추가합니다.
4. 이 컬러 태그는 그레디언트가 의도한 대로 보이도록 폰트 컬러를 흰색으로 복원하며, 이때 참조되는 그레디언트는 1단계에서 생성한 에셋 이름과 일치해야 합니다. **Rich Text**가 활성화되어야 합니다.
5. UI 빌더 또는 게임 뷰에서 변경 사항이 적용된 것을 볼 수 있습니다.

스프라이트 에셋과 이모티콘

리치 텍스트 태그를 통해 텍스트에 이모티콘과 같은 스프라이트를 추가할 수 있습니다. 스프라이트를 사용하려면 그레디언트 에셋과 유사한 스프라이트 에셋을 사용해야 합니다.

여러 스프라이트를 임포트할 때 하나의 아틀라스로 패킹하면 드로우 콜을 줄일 수 있습니다. 스프라이트 아틀라스의 해상도는 반드시 타겟 플랫폼에 적합해야 합니다. 스프라이트 해상도에 관한 자세한 내용은 에셋 준비 섹션을 확인하세요.



스프라이트 에셋은 보통 텍스트 문자열에 통합된 이모티콘이나 아이콘에 사용됩니다.

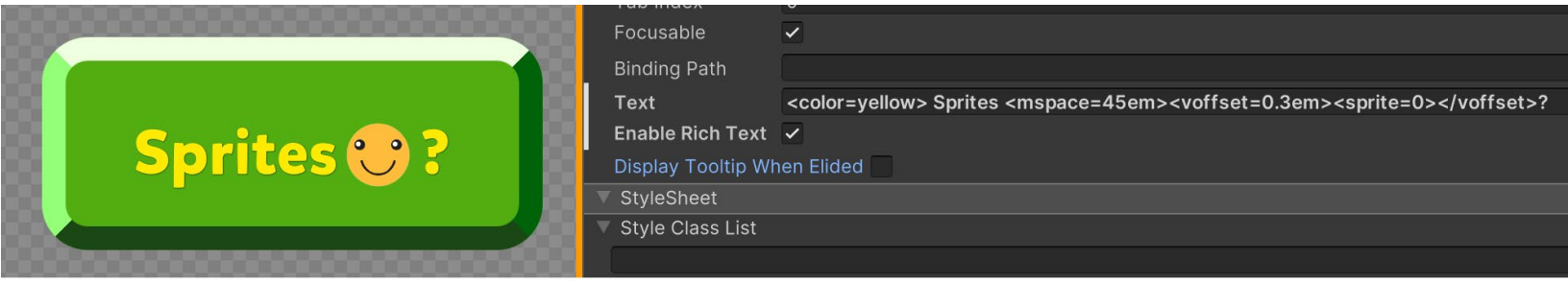
이러한 목적으로 스프라이트를 임포트하는 방법은 다음과 같습니다.

1. 이모티콘이나 아이콘을 포함하는 스프라이트 또는 PSD 파일을 임포트합니다.
2. 이미지를 여러 개의 스프라이트로 분할합니다. 다만 그래픽 및 폰트 에셋 준비 섹션에 설명된 대로 PSD 파일을 사용하는 경우에는 이렇게 분할할 필요가 없습니다. 파일에서 **스프라이트 에셋**을 생성합니다 (**Create > Text Core > Sprite Asset** 메뉴). 에셋은 Assets/Resources 또는 하위 폴더에 넣어야 합니다.
3. Face Info를 조정하고 새로운 스프라이트 에셋의 각 글리프 외관/이름을 커스터마이즈할 수 있습니다. 여기에서 적용한 모든 변경 사항은 폰트 에셋의 기본 Face Settings를 대체합니다.

참고: 여기에서 **Update Sprite Asset**을 선택하면 스프라이트 에셋이 스프라이트 에디터의 최신 변경 사항과 동기화됩니다.

이 에셋을 UI 툴킷에서 사용하려면, 그레디언트 작업에서 한 것과 동일한 단계를 따라야 합니다.

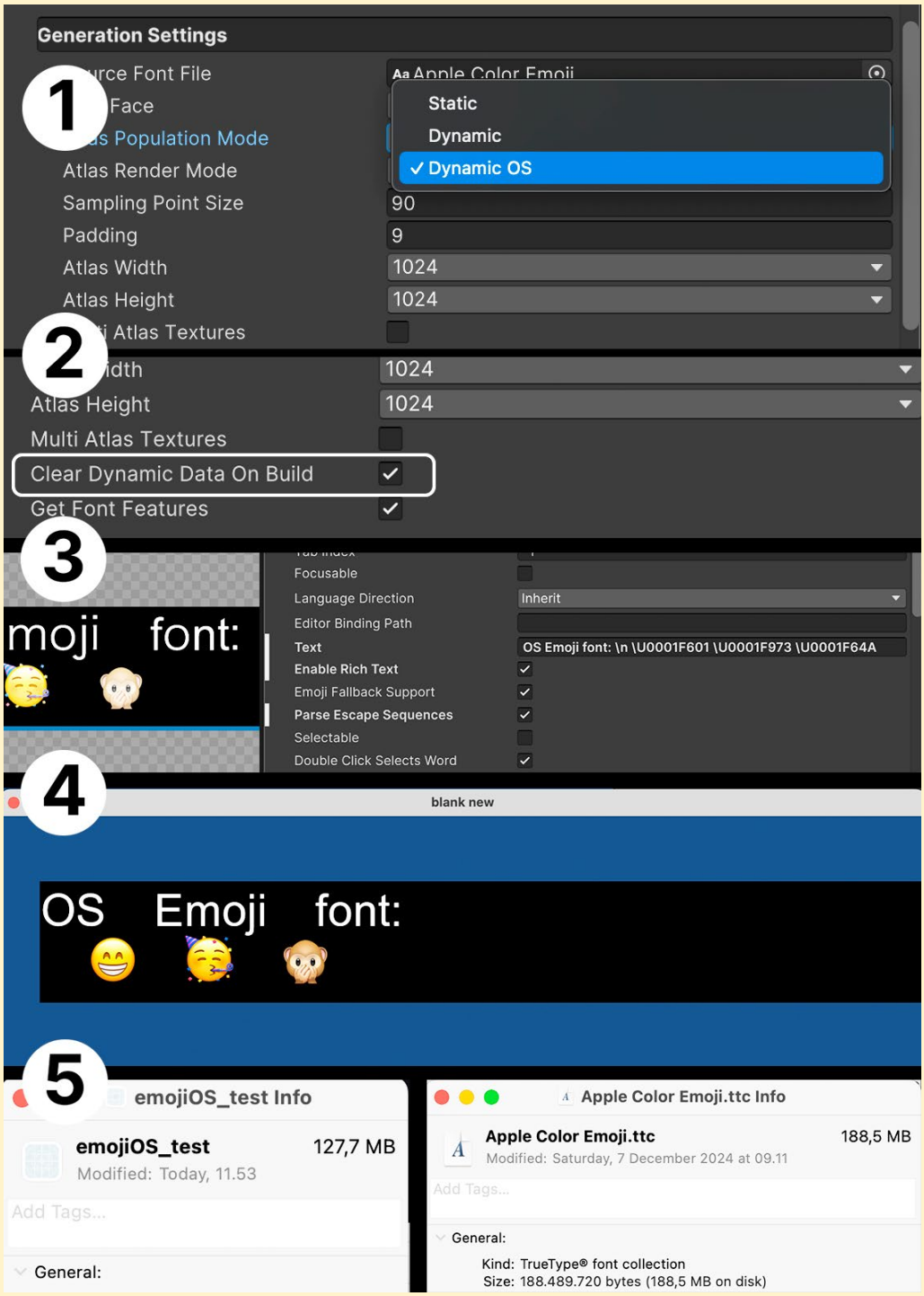
1. **UI Document**에서 **Panel Settings**를 선택합니다.
2. **Text Settings** 에셋을 엽니다. 없으면 새로 만듭니다.
3. Text Settings 파일에서 파일 브라우저를 사용하여 **스프라이트 에셋**에 연결합니다. 저장한 다음 플레이 모드를 실행하여 업데이트된 설정이 적용되는지 확인합니다.
4. 리치 텍스트 태그(`<sprite index=0>` 또는 `<sprite name="name">`)를 사용하여 스프라이트를 추가합니다. 임베드된 스프라이트는 다른 텍스트 태그도 따릅니다.



리치 텍스트 태그를 사용하여 UI 툴킷의 텍스트 필드에 스프라이트 에셋을 추가합니다. 상단의 **Enable Rich Text** 옵션이 선택되어 있어야 합니다.

팁: OS 이모티콘 사용

iOS나 Android와 같은 특정 런타임 플랫폼을 타게팅한다면 프로젝트에 소스 폰트를 포함하는 대신 시스템의 빌트인 이모티콘 폰트를 사용할 수 있습니다. 그러면 메모리를 절감할 수 있으며 대용량 이모티콘 컬렉션을 애플리케이션과 함께 패킹하지 않아도 됩니다. Text Settings에서 글로벌 폴백으로 활용하기에도 적합한 경우가 많습니다.

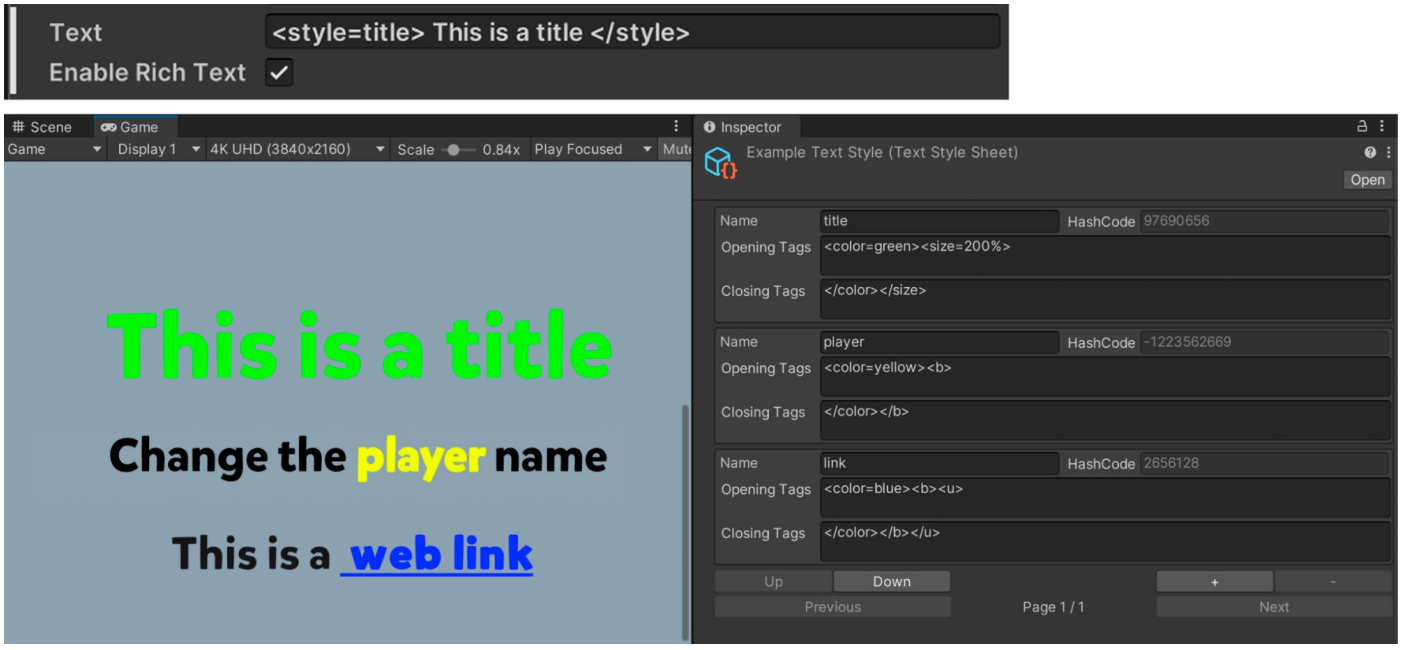


프로젝트에서 OS 이모티콘을 사용하려면 다음 단계를 따르세요.

1. 타겟 시스템이 사용하는 폰트에서 폰트 에셋을 생성합니다. iOS의 폰트는 Apple Emoji이고(이 예시에서 사용), Android의 폰트는 Noto Color Emoji입니다(현재 COLRv0만 지원됨). 폰트 에셋의 유형을 **Color**로 설정하고 Atlas Population 모드를 **Dynamic OS**로 설정해야 합니다. 그러면 에셋에 소스 폰트를 포함할 필요가 없어 공간이 좀더 확보됩니다.
2. 폰트 에셋에서 **Clean Dynamic Data On Build**를 선택합니다.
3. UI 빌더에서 **Parse Escape Sequences**를 선택한 다음, macOS 또는 Windows의 이모티콘 키보드를 사용하거나 UTF 포맷으로 원하는 이모티콘을 입력합니다. 예를 들어 스마일 이모티콘은 `\U0001F601`이며, 폰트 에셋의 Character Table에서 각 이모티콘의 UTF를 확인할 수 있습니다.
4. OS 폰트에 따라 macOS에서 실행 중인 빌드에 이모티콘이 표시됩니다.
5. 테스트에서 보면 스탠드얼론 이모티콘 폰트에 비해 **빌드 크기**가 작은 것을 확인할 수 있습니다. 따라서 프로젝트에 이모티콘 폰트가 포함되지 않았음에도 적합한 이모티콘을 렌더링하는 데 사용되고 있음을 알 수 있습니다.

텍스트 스타일 시트

애플리케이션에서 많은 양의 텍스트를 처리하는 경우, 텍스트 스타일 시트를 만들어서 텍스트 서식을 관리하는 것이 좋습니다. 텍스트 스타일 시트를 만들면 <style> 리치 텍스트 태그를 사용하여 커스텀 텍스트 스타일을 만들 수 있습니다. 텍스트 스타일 시트는 **Assets > Text Core > Text Stylesheet**의 Create 메뉴에서 만들 수 있습니다.



재사용 가능한 텍스트 스타일 시트

텍스트 스타일 시트의 장점은 다음과 같습니다.

- 커스텀 스타일은 열고 닫는 리치 텍스트 태그, 선행 및 후행 텍스트를 포함할 수 있습니다.
- 텍스트 스타일 시트를 편리하게 업데이트할 수 있으며, 이 작업은 특히 리치 텍스트 서식을 직접 변경하는 것보다 간편합니다.
- 커스텀 스타일을 사용하면 리치 텍스트 태그의 양을 줄일 수 있습니다. `<style= name>` 태그 하나만 사용하면 필요한 모든 스타일을 적용할 수 있습니다.
- 따라서 텍스트 스타일 시트에서 하나의 리치 텍스트 태그를 쉽게 변경할 수 있고, 여러 개의 `<style>` 태그를 직접 수정하는 것보다 오류가 발생할 확률이 낮습니다.

데이터 바인딩

사용자 인터페이스의 핵심은 애플리케이션을 구동하는 데이터와 플레이어를 연결하는 것입니다. 플레이어는 사용자 인터페이스를 통해 게임의 내부 상태와 로직을 보고, 만지고, 상호 작용합니다.

플레이어는 원시 데이터 그대로의 스탯이 아닌 체력 바를 보게 되며, 아이템 목록을 직접 읽는 것이 아니라 드래그 앤 드롭 방식의 인벤토리를 사용하게 됩니다. UI와 데이터 사이의 이러한 상호 작용은 프로젝트의 구조를 정하는 방식에 영향을 줍니다.

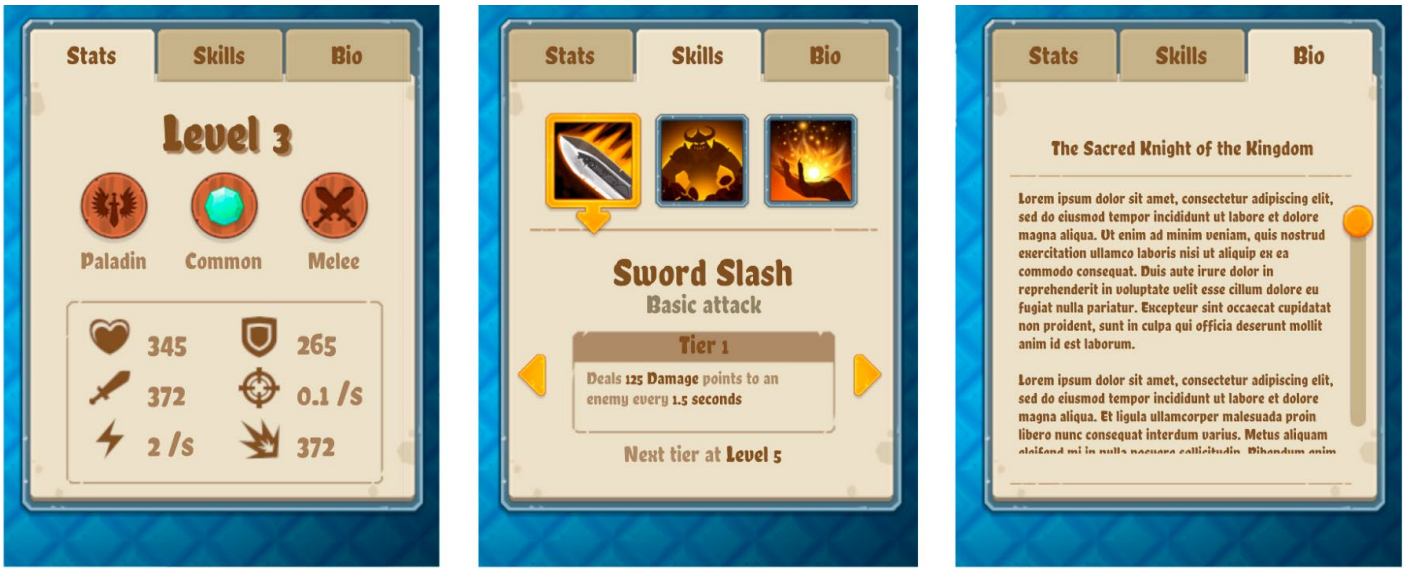
게임 데이터를 반영하는 UI

다음은 UI 툴킷 샘플 – 드래곤 크래셔의 캐릭터 스탯 창입니다. 이 사용자 인터페이스에서는 RPG식 게임의 주요 특성을 엿볼 수 있습니다.

뷰는 플레이어가 상호 작용하는 UI 자체를 나타냅니다. 탭 방식의 컨테이너는 간편하게 탐색할 수 있도록 캐릭터의 능력을 깔끔하게 정리해 보여 줍니다.

캐릭터 스탯 창은 게임 데이터를 나타냅니다.

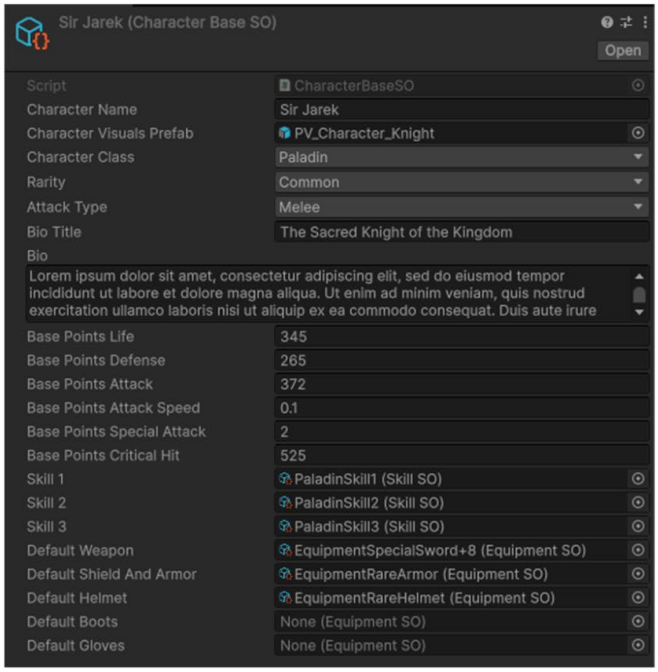
내부적으로 살펴보면, 각 캐릭터의 스탯이 저장되는 ScriptableObject 등의 데이터는 모델 내에 있습니다.

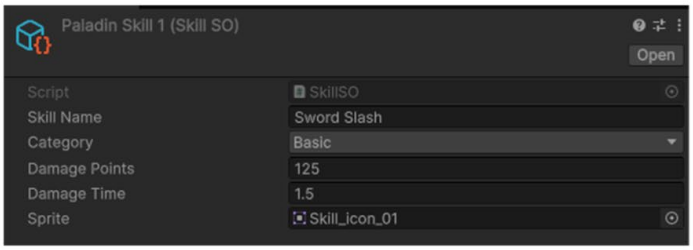


View (user interface)

↑

Model (data)





ScriptableObject 에셋에는 캐릭터의 데이터가 포함됩니다.

뷰와 모델의 이러한 관심사 분리는 UI 아키텍처의 핵심 원칙입니다. 시각적 인터페이스와 기반 데이터를 분리하면 코드의 유연성, 재사용성, 관리 용이성을 높일 수 있습니다.

하지만 일단 분리되면 모델을 뷰에 연결하기 위해 동기화가 필요합니다. 전통적으로 이러한 동기화에는 직접적인 업데이트나 이벤트 기반 시스템이 필요하고, 데이터가 변경되면 관찰자가 UI를 업데이트해야 합니다. 이러한 동기화 방식은 효과적이지만 반복적이고 틀에 박힌 상용구 코드로 이어질 수 있습니다.

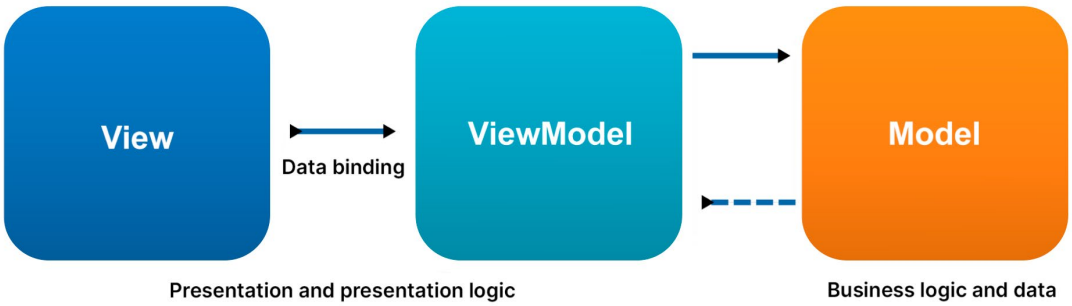
이러한 시스템은 프로젝트의 규모가 성장함에 따라 관리하기가 어려워질 수 있습니다. 새로운 요소나 종속성을 더하려면 업데이트 로직이나 이벤트 핸들러가 추가로 필요한 경우가 많습니다. 그러면 스크립트가 복잡해져 가독성이 떨어지고 유지 관리가 어려워질 수 있습니다.

런타임 데이터 바인딩

Unity 6의 런타임 데이터 바인딩은 이 문제에 대한 간소화된 해결책을 제시합니다. 애플리케이션의 데이터를 UI 요소에 직접 연결하므로, 한 쪽에 적용된 변경 사항이 다른 쪽에도 자동으로 반영됩니다.

이러한 MVVM(모델-뷰-뷰모델) 아키텍처는 뷰와 모델 사이에 프레젠테이션 로직 레이어를 추가합니다. 뷰모델은 모델과 뷰 사이의 중개자 역할을 하며 모델의 데이터에 포맷을 지정하여 뷰로 노출합니다.

Unity 전자책 디자인 패턴 및 SOLID 원칙으로 코딩 스킬 업그레이드에서 MVVM을 비롯한 여러 디자인 패턴에 대해 자세히 알아보세요.



MVVM 아키텍처(출처: Wikipedia)

예를 들어 체력 바에는 플레이어의 체력이 자동으로 표시되고, 점수 레이블은 스크립트 로직이 추가되거나 이벤트가 수동으로 처리되지 않아도 실시간으로 업데이트될 수 있습니다. 관리해야 할 동기화 로직이 줄어들기 때문에 프로젝트를 더 효과적으로 확장할 수 있습니다.

UI 툴킷의 런타임 데이터 바인딩을 프로젝트에서 어떻게 활용하는지 예시를 통해 알아보겠습니다.

데이터 바인딩의 개념

Unity 6에는 UI 요소를 체계적으로 애플리케이션 데이터에 연결하는 방법을 제공하는 런타임 데이터 바인딩 시스템이 추가되었습니다. 시각적 요소의 프로퍼티를 데이터 소스에 바인딩하려면 DataBinding 인스턴스를 생성해야 합니다.

다음은 몇 가지 중요한 개념입니다.

— 데이터 소스: UI 바인딩용 데이터가 저장되는 오브젝트입니다.

— 데이터 소스 경로: 데이터 소스의 프로퍼티 또는 필드로, UI 요소가 여기에 연결됩니다.

— 바인딩 모드: 소스와 UI 간에 데이터가 이동하는 방식을 제어하며, 단방향일 수도 있고 양방향일 수도 있습니다.

이러한 각 요소가 연동하여 데이터 바인딩을 구현합니다. 그럼 각 항목에 대해 자세히 살펴보겠습니다.

데이터 소스 준비

데이터 소스는 UI 바인딩용 데이터가 저장되는 오브젝트입니다. 모든 C# 오브젝트(ScriptableObject, MonoBehaviour, 커스텀 C# 오브젝트 등)는 데이터 소스 역할을 수행할 수 있습니다. 구조체를 데이터 소스로 사용하면 할당되는 메모리가 적어지고 가비지 컬렉션이 줄어들어 성능을 향상할 수 있습니다. 데이터 바인딩은 코드나 인스펙터를 통해 설정할 수 있습니다.

이 데모 프로젝트는 Unity 인스펙터 내에서 데이터를 직렬화하는 편리한 기능이 있는 ScriptableObject를 데이터 소스로 사용합니다.

CreateProperty 속성 사용

UI 툴킷은 Unity Properties 모듈이 생성하는 프로퍼티 백을 사용하여 바인딩용 프로퍼티를 표시합니다. 프로퍼티 백은 데이터 소스의 어느 프로퍼티에서 UI 바인딩에 액세스할 수 있는지 정의합니다.

프로퍼티를 바인딩 가능하게 만들려면 CreateProperty 속성을 사용합니다. 이 속성은 바인딩 시스템을 위한 프로퍼티를 명시적으로 표시합니다. 다음은 자주 사용되는 설정 패턴입니다.

```
[SerializeField, DontCreateProperty]
int m_Value;

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

이 예시에서 `m_Value`는 직렬화되도록 `SerializeField` 속성으로 표시되었으나 `DontCreateProperty` 속성에 의해 바인딩에서 제외되었습니다.

반면에 `Value` 프로퍼티는 `CreateProperty`로 표시되었으므로 바인딩 시스템이 액세스할 수 있습니다. 이처럼 명확한 구분은 모델과 UI 간의 데이터 플로를 관리하는 데 도움이 됩니다.

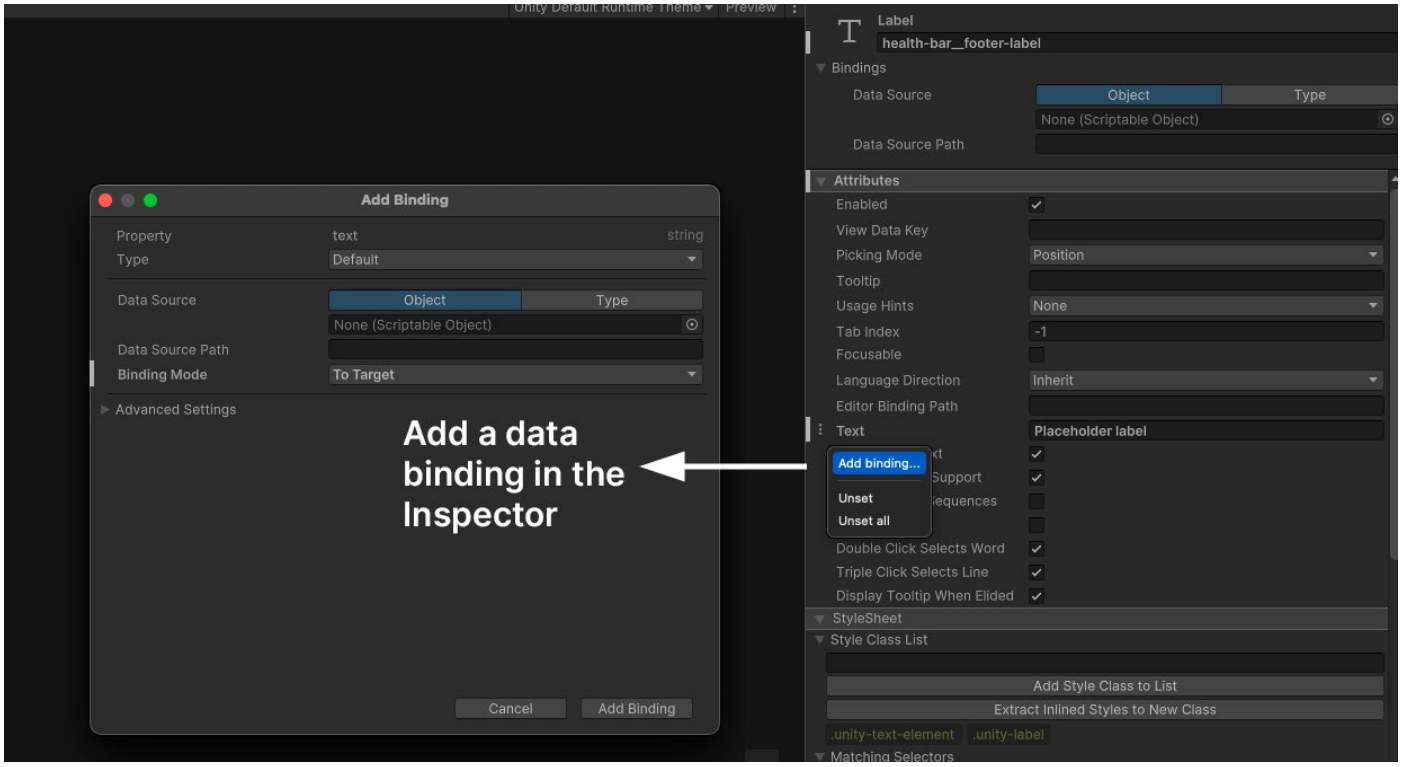
런타임 데이터 바인딩은 프로퍼티 백을 사용하여 특정 유형의 데이터를 효율적으로 통과하고 조작합니다. Unity는 기본적으로 유형이 처음 액세스될 때 리플렉션을 사용하여 프로퍼티 백을 생성하며, 이로 인해 약간의 런타임 오버헤드가 추가로 발생합니다.

이를 방지하려면 프로퍼티를 정의할 때 CreateProperty 속성을 사용하면 됩니다. 그러면 컴파일 시 바인딩 코드가 생성되므로 런타임 리플렉션이 필요하지 않게 되며 성능 오버헤드가 줄어듭니다.

데이터 소스와 경로

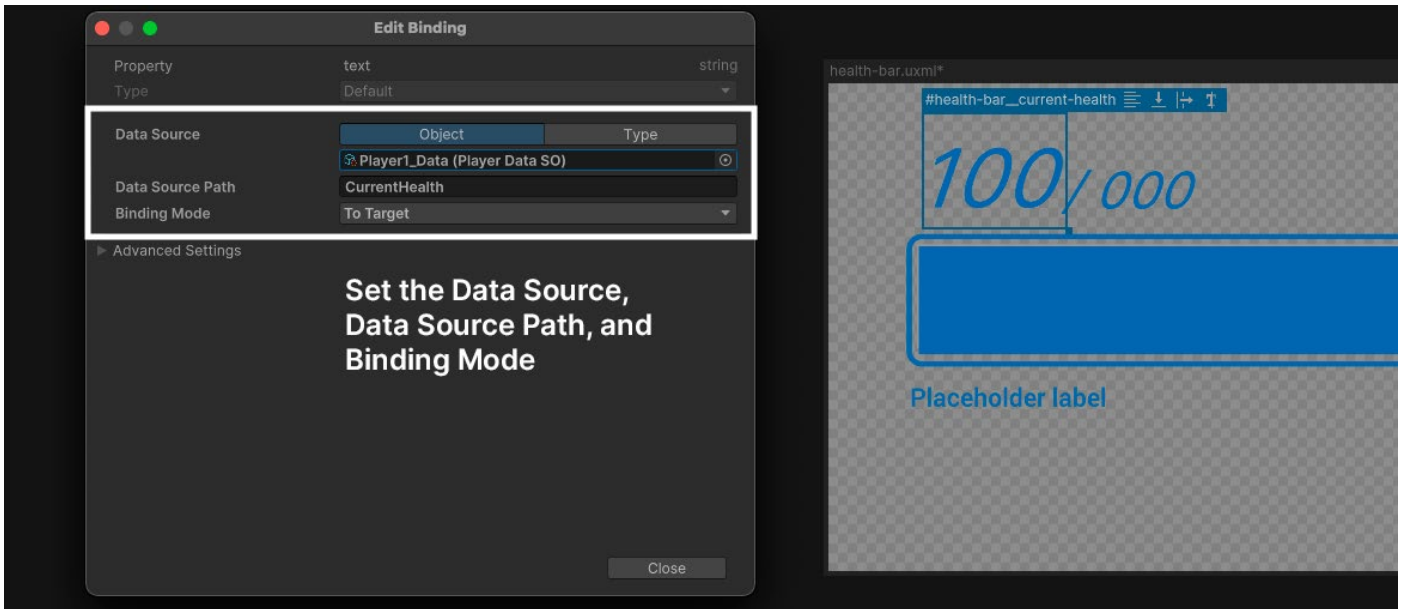
데이터 소스가 준비되면 UI에 바인딩할 수 있습니다. 데이터 소스 경로는 해당 데이터 소스 내에서 UI 요소에 연결할 프로퍼티 또는 필드를 지정합니다. 예를 들어 데이터 소스에 'health' 프로퍼티가 있다면, 해당 경로는 UXML에서 이를 사용하거나 C#의 바인딩 설정을 통해 이 프로퍼티를 직접 가리키게 됩니다. 실제 예시를 살펴보겠습니다.

UI 빌더에서: 계층 구조 요소를 하나 선택하고 인스펙터로 이동한 다음, 옵션(:) 메뉴의 **Add Binding** 옵션을 사용합니다.



인스펙터에서 바인딩을 추가합니다.

그런 다음 **Data Source**에 데이터 소스(이 예시에서는 `PlayerDataSO ScriptableObject`)를 할당하고 **Data Source Path**(이 예시에서는 `CurrentHealth`)를 지정합니다.



UI 빌더에서 Data Source와 Data Source Path를 설정합니다.

UXML에서: UI 빌더에서 데이터 바인딩을 설정하면 그에 상응하는 UXML이 생성됩니다. 텍스트 에디터를 사용하여 수동으로 데이터 소스 경로를 추가하거나 편집해도 됩니다. 다음은 바인딩을 생성하는 코드 블록입니다.

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

C# 사용: 스크립트에서 ScriptableObject와 같은 데이터 소스 오브젝트를 인스턴스화하거나 참조합니다. 해당 오브젝트를 루트 요소의 dataSource 프로퍼티에 할당합니다. dataSourcePath를 사용하여 바인딩할 구체적인 프로퍼티를 지정합니다.

다음 스니핏은 스크립트에서 dataSource 프로퍼티와 dataSourcePath 프로퍼티를 설정하는 방법을 보여 줍니다. 이 내용은 아래의 C#으로 데이터 바인딩 설정 섹션에서 더 자세히 다룹니다.

```
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)),
});
```

참고: 같은 UI 요소에 대해 데이터 바인딩을 정의하는 경우 충돌이 발생할 수 있습니다. 혼동을 피하는 방법은 다음과 같습니다.

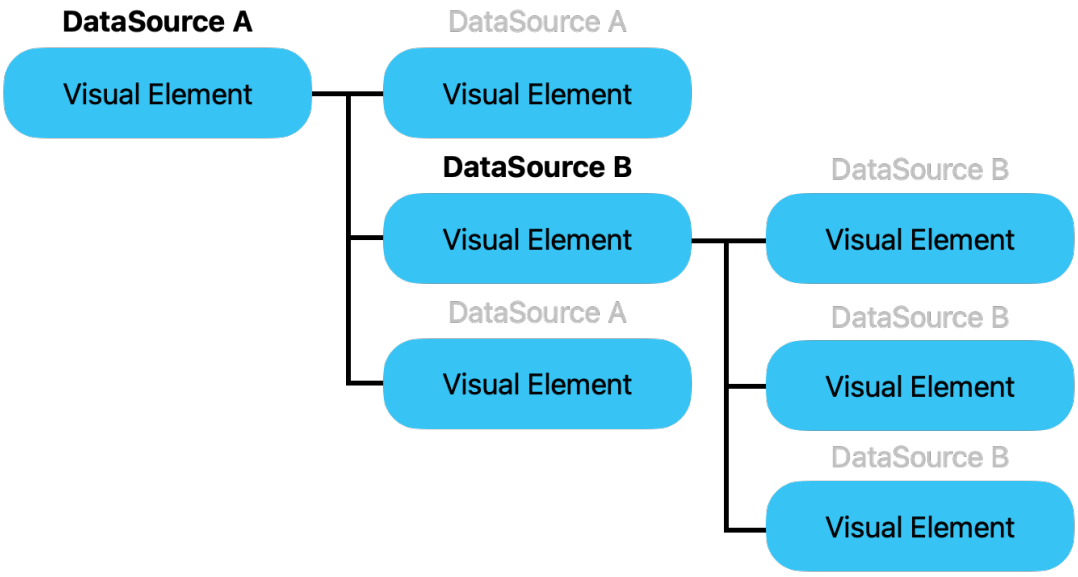
— **UI 빌더/UXML 바인딩**: 런타임에 조정이 필요하지 않은 정적 또는 기본 데이터 구성인 경우에 사용합니다.

— **C# 바인딩**: 동적 업데이트 또는 게임플레이 중에 데이터 소스가 변경되어야 하는 경우에 사용합니다.

바인딩의 일부를 UI 빌더/UXML에서 설정하고 런타임에 바인딩을 완료하는 것도 가능합니다. 자세한 내용은 아래의 미지정 데이터 바인딩 워크플로 섹션을 참조하세요.

데이터 소스 상속

새로운 데이터 소스가 명시적으로 할당되지 않았다면 시각적 요소는 자동으로 부모의 데이터 소스를 상속합니다. 예를 들어 루트 요소에 데이터 소스가 있으면 모든 자식 요소는 기본적으로 이 데이터 소스를 사용합니다. 다음 다이어그램에서 이 동작을 확인할 수 있습니다.



자식 요소는 부모 데이터 소스를 오버라이드할 수 있습니다.

부모 요소에 데이터 소스가 있으면 자식 요소는 이를 자동으로 상속합니다. UI 빌더에서 자식의 Data Source 필드에는 자동으로 부모의 데이터 소스가 입력되지만, 필요한 경우 오버라이드할 수 있습니다.

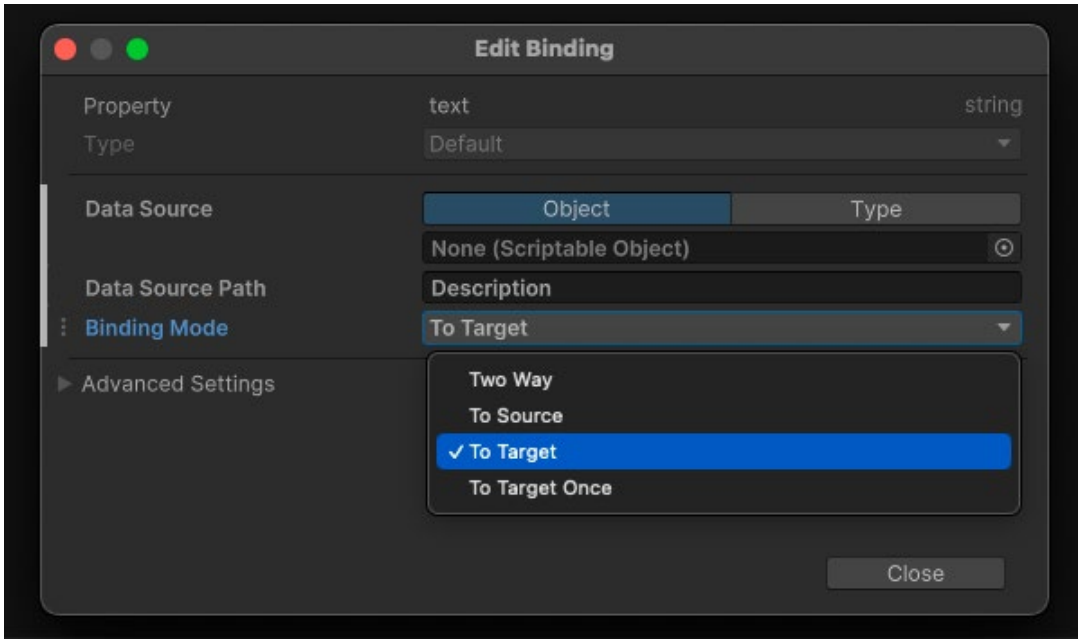
아래 예시에서 볼 수 있듯이 C#을 사용할 때도 동일한 상속 로직이 적용됩니다.

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

// 루트 요소에 데이터 소스 할당
root.dataSource = parentData;

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

// 자식의 상속받은 데이터 소스 오버라이드
child.dataSource = childData;

root.Add(child);
```

자식이 부모를 오버라이드하여 독립적인 데이터 소스를 갖습니다.

바인딩 모드

바인딩 모드는 데이터 소스와 UI 사이의 데이터 플로를 제어합니다.



UI 빌더에서 Binding Mode를 사용하여 데이터 소스와 UI 간의 데이터 플로를 제어할 수 있습니다.

UI 빌더와 C# API에는 다음과 같은 옵션이 있습니다.

- **Two Way(기본값):** 변경 사항이 데이터 소스에서 UI로, 그리고 UI에서 데이터 소스로 전파됩니다. 사용자가 데이터를 변경할 수 있는 슬라이더나 텍스트 필드와 같은 인터랙티브 요소에 사용하세요.
- **To Target:** 데이터 플로가 데이터 소스에서 UI 방향으로만 진행됩니다. 읽기 전용 UI 요소에 사용하세요.
- **To Source:** 데이터 플로가 UI에서 데이터 소스 방향으로만 진행됩니다. 처음부터 현재 값을 표시할 필요가 없는 입력 항목에 적합합니다.
- **To Target Once:** 데이터 플로가 데이터 소스에서 UI로 한 번만 이루어지며, 이후부터는 데이터 소스의 추가 변경 사항이 UI에 반영되지 않습니다.

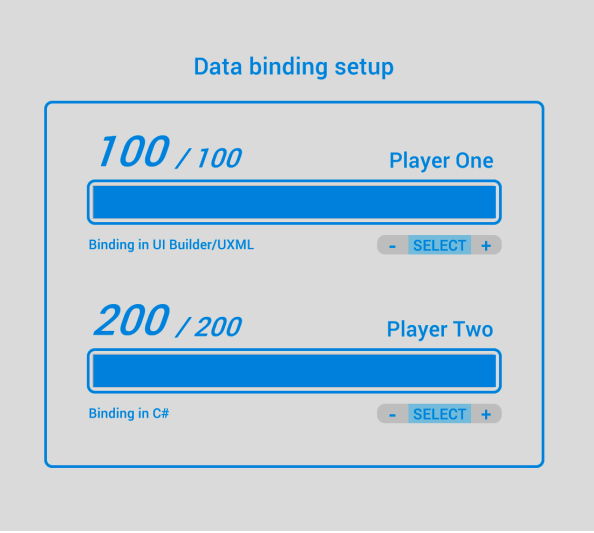
예시: 체력 바 데이터 바인딩

UI 툴킷에서 기본적인 데이터 바인딩을 생성하는 방법을 보여 주는 예제를 살펴보겠습니다. 다음은 데모 씬에서 플레이어의 체력에 따라 동적으로 업데이트되는 간단한 체력 바의 예시입니다.

데모 씬
이어지는 예시는 QuizU 샘플 프로젝트에 포함된 **데이터 바인딩** 사용법 데모에서 볼 수 있습니다.

런타임에 액세스하려면 메인 메뉴로 이동하여 **Demos** > **Data Binding**을 선택하거나, 부트로더를 비활성화한 직후에 **DataBindingDemo**를 로드합니다(**Quiz > Don't Load Bootstrap Scene on Play**).

데모 씬에는 두 개의 체력 바가 있습니다. 하나는 UI 빌더에서 UXML로 바인딩이 생성되었고, 다른 하나는 C#으로 바인딩이 생성된 것입니다.



체력 바는 플레이어 데이터를 나타냅니다.

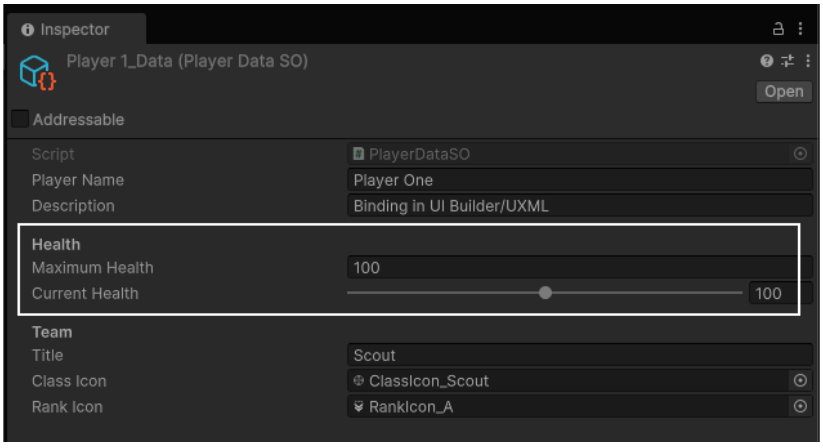
데이터 소스 준비

샘플 프로젝트에는 PlayerDataSO ScriptableObject에 저장된 플레이어 정보와 스탯이 포함되어 있습니다. PlayerDataSO의 관련 프로퍼티는 바인딩에 사용할 수 있도록 CreateProperty 속성으로 표시되어 있습니다.

각 체력 바는 PlayerDataSO에 있는 데이터의 일부 하위 집합만을 나타내며, 여기에는 플레이어 이름과 체력 값이 포함됩니다. 아래의 클래스 스니핏에서 몇 가지 프로퍼티와 관련 필드를 볼 수 있습니다.

```
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;
```

이 UI는 PlayerName, CurrentHealth, MaximumHealth와 같은 특정 데이터 경로를 사용하여 이 정보를 화면에 시각적으로 표시합니다.



데이터 소스는 PlayerDataSO ScriptableObject의 체력 데이터를 포함합니다.

UI 빌더/UXML의 데이터 바인딩

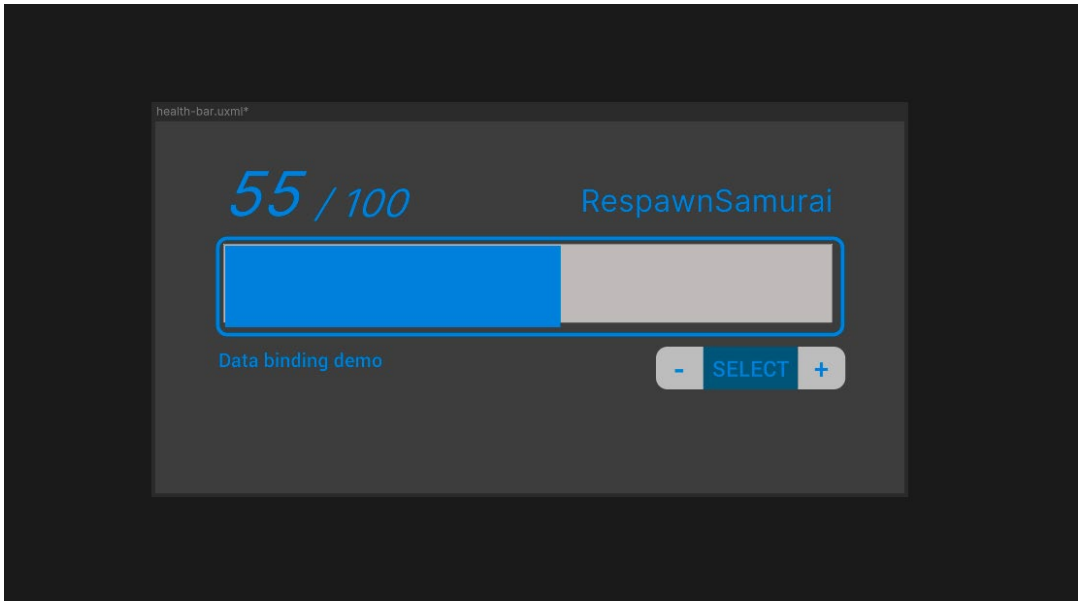
UI 빌더는 UI 요소를 데이터에 바인딩하는 시각적이고 인터랙티브한 방법을 제공합니다. 이 방법은 디자인 중심 워크플로를 선호하는 UI 아티스트 및 설정 과정에서 실시간 피드백이 필요한 개발자에게 적합합니다. 데이터 바인딩을 처음 학습하는 사람들에게도 유용한 학습 툴이 될 수 있습니다.

데모 씬에서 Player One 체력 바의 데이터 바인딩은 UI 빌더만으로 설정되었습니다. 데이터 바인딩을 설정하는 방법은 다음과 같습니다.

— **루트 요소 선택**: 계층 구조에서 체력 바가 있는 루트 요소를 선택합니다. 이 예시에서 최상위 컨테이너는 **demo_container-uxml** 요소입니다.

— **데이터 소스 할당**: 인스펙터의 Data Binding 섹션에서 데이터 소스를 ScriptableObject 에셋으로 설정합니다. 이렇게 하면 데이터 소스가 할당되고 모든 자식 요소로 전파됩니다.

— **데이터 소스 경로 정의**: 개별 UI 요소를 ScriptableObject(이 예시에서는 PlayerDataSO.PlayerName)의 각 프로퍼티에 연결할 데이터 소스 경로를 지정합니다.



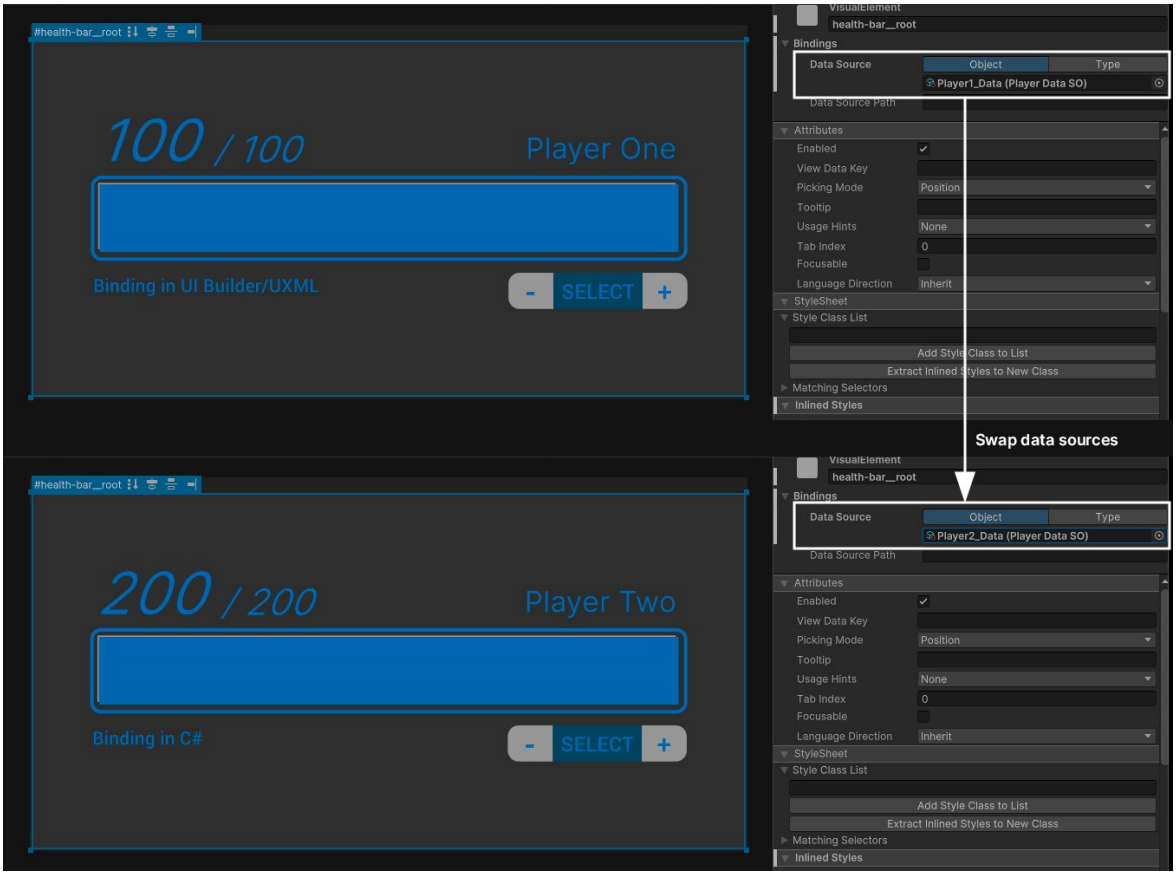
기본 체력 바

루트에서 데이터 소스를 설정하면 이 데이터 소스가 자식 요소들의 기본 데이터 소스로 표시됩니다. 올바른 데이터 소스 경로를 입력하세요. 다음 표에는 UI 요소 프로퍼티를 ScriptableObject에 연결하는 데이터 바인딩이 정리되어 있습니다.

UI 요소	UI 요소 프로퍼티	바운딩 프로퍼티	참고
`health-bar__player-name`	`text`	`PlayerName`	플레이어의 이름 표시
`health-bar__current-health`	`text`	`CurrentHealth`	현재 체력 값 표시
`health-bar__max-health`	`text`	`MaximumHealth`	최대 체력 표시
`health-bar__progress`	`style.width`	`Progress`	동적으로 바 너비 조정

데이터 바인딩이 완료되면 체력 바가 실시간으로 업데이트되어 여러 레이블과 플레이어 체력을 나타내는 진행 표시줄을 볼 수 있습니다.

데이터 소스를 변경하는 방법은 간단합니다. 새로운 ScriptableObject 에셋을 할당하면, UI에서 바인딩은 동일하게 유지된 상태로 자동으로 새로운 값이 반영됩니다.



데이터 소스를 변경하면 데이터 바인딩이 업데이트됩니다.

UI 빌더에서 데이터 바인딩을 설정하면 바인딩이 UXML 파일에 추가되고 각 바인딩 요소별로 `<Bindings>` 블록이 생성됩니다.

다음은 `health-bar__player-name` 요소의 text 프로퍼티를 `PlayerName` 프로퍼티에 바인딩하면 생성되는 UXML의 스니핏입니다(일부 속성은 가독성을 위해 생략됨).

```
<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>
```

숙련된 사용자라면 UXML로 직접 바인딩을 생성할 수 있습니다. 코드를 사용하면 다량의 바인딩을 사용할 때 정밀하게 제어하고 빠르게 편집할 수 있습니다. 직접 작성한 UXML은 버전 관리를 위한 깔끔한 비교 자료를 제공하므로, 병합 충돌을 해결하거나 변경 사항을 추적하기도 쉬워집니다.

C#으로 데이터 바인딩 설정

UI 빌더는 정적 데이터(사전 정의된 ScriptableObject 에셋 등)를 사용하여 프로토타이핑하기에 좋지만, 런타임 데이터는 C#을 사용하여 동적으로 처리해야 하는 경우가 많습니다. 다음 코드 예시는 Player Two의 체력 바가 데모 씬에서 작동하는 방식을 보여 줍니다.

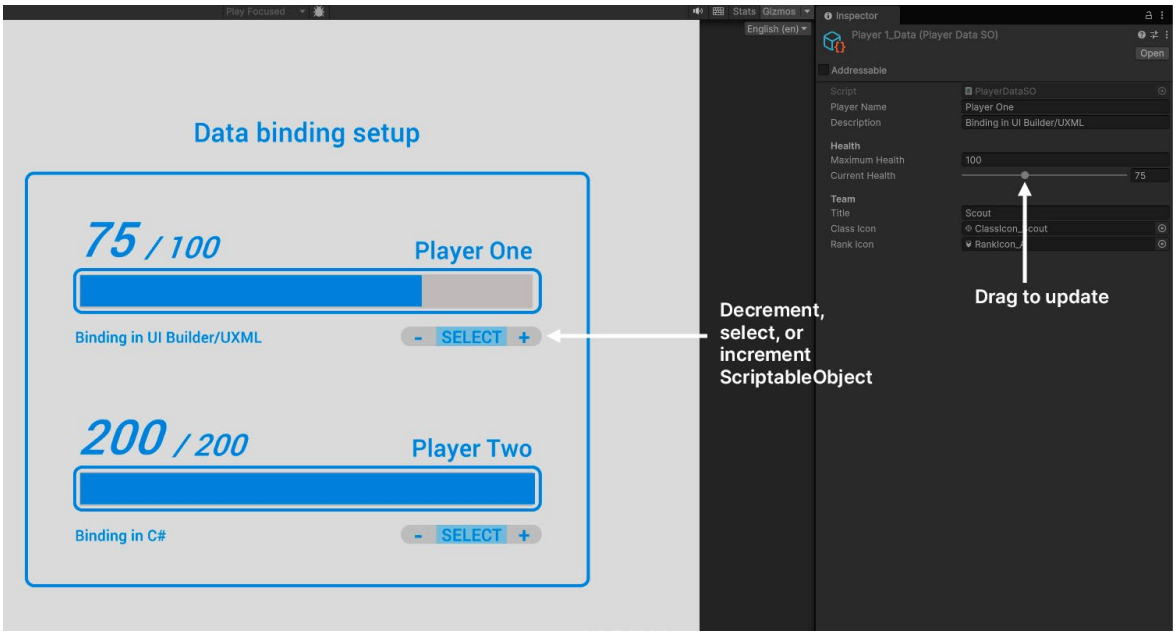
```
using UnityEngine;
using UnityEngine.UIElements;
using Unity.Properties;

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
           });
       }
}
```

`HealthBar` 스크립트는 `OnEnable`의 메인 컨트롤러 스크립트에서 호출되는 `Initialize` 메서드에서 이 작업을 처리합니다 .

— 먼저 `health-bar__player-name` 요소를 쿼리합니다. 그런 다음 데이터 소스로 ScriptableObject를 할당합니다.

— 이어서 `SetBinding` 메서드가 text 프로퍼티를 새 DataBinding 인스턴스에 바인딩하고 `dataSourcePath` 파라미터와 `bindingMode` 파라미터를 설정합니다.

위 표에 나와 있는 네 개의 바인딩은 모두 비슷한 방식으로 설정됩니다. ScriptableObject 슬라이더 또는 커스텀 에디터 프로퍼티 드로어를 사용하여 `CurrentHealth`를 조정합니다. 데모에는 ScriptableObject를 늘리거나 줄이거나 선택하는 데 필요한 플레이 테스트 컨트롤(+, -, Select)이 있습니다. 체력 바는 적용되는 변경 사항에 따라 동적으로 업데이트됩니다.

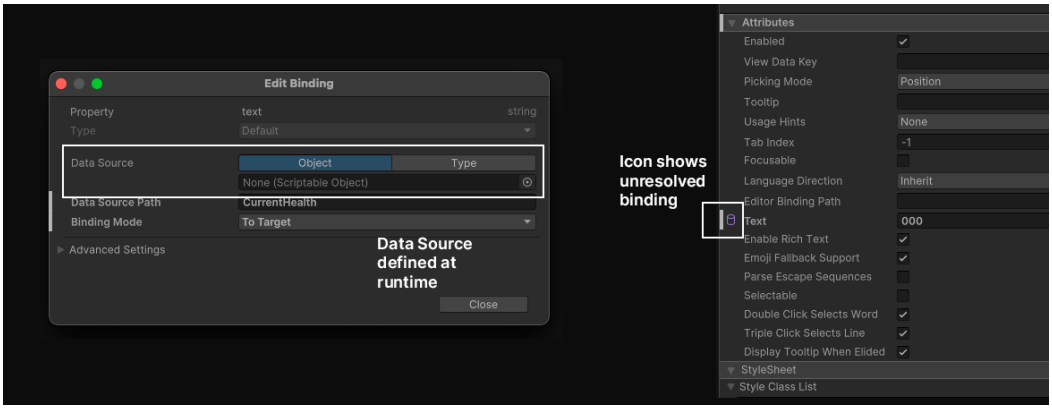


HealthBar는 CurrentHealth 값과 동기화됩니다.

미지정 데이터 바인딩 워크플로

Unity 6는 UI 빌더의 시각적 설정과 스크립팅의 유연성이 결합된 하이브리드 데이터 바인딩 워크플로도 지원합니다.

데이터 소스를 UXML로 하드 코딩하는 대신 Data Source Type을 지정하고 실제 데이터 소스는 미지정 상태로 남겨 둘 수 있습니다. 이처럼 완성되지 않은 바인딩은 UI 빌더에서 비어 있는 아이콘으로 표시됩니다. 경로와 유형은 설정되었으나 데이터 소스가 아직 할당되지 않은 것입니다.



미지정 데이터 바인딩은 비어 있는 아이콘으로 표시됩니다.

런타임에는 한 줄의 코드만 사용하여 데이터 소스를 할당할 수 있습니다. 예를 들면 다음과 같습니다.

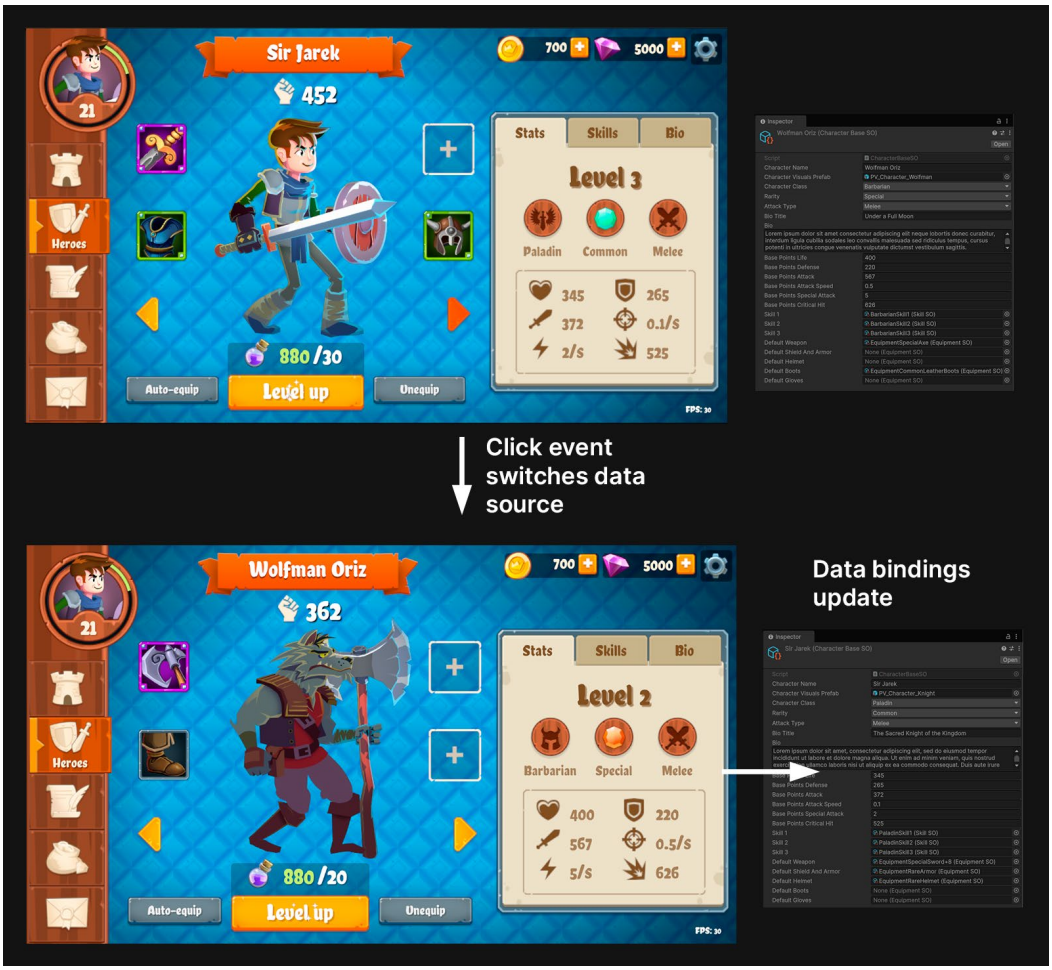
```
myElement.dataSource = myNewDataSource;
```

이처럼 `myNewDataSource`를 `myElement`에 할당하면 UXML로 정의된 플레이스홀더 바인딩이 지정되며, UI가 자동으로 업데이트될 수 있습니다. 이 방법을 사용하면 `SetBinding` 호출을 반복하지 않아도 되며 UXML을 유연하게 유지할 수 있습니다.

드래곤 크래셔 샘플에서도 데이터 경로를 UXML로 사전 정의하고 실제 데이터 소스는 런타임에 설정합니다.

UI에서 이전/다음 버튼을 클릭하면 현재 선택된 캐릭터가 데이터 소스로 설정됩니다. 데이터 소스를 변경하는 작업에는 UXML 수정이 필요하지 않습니다.

미지정 바인딩은 새 데이터 소스가 설정되면 올바른 캐릭터 스탯을 표시합니다.



드래곤 크래셔 샘플에서 데이터 소스 업데이트하기

참고: UXML 파일에 특정 데이터 소스가 설정되면(예: `data-source="PlayerDataSO.asset"`) 바인딩은 고정되고 런타임에 바뀌지 않습니다. 바인딩이 런타임에 바뀌게 하려면 `data-source` 속성을 비워 두거나 `data-source-type`을 사용해야 합니다.

이 하이브리드 데이터 바인딩 워크플로의 예시를 보려면 ListView에 목록 바인딩을 참조하세요.

타입 컨버터

Unity 6의 타입 컨버터를 사용하면 원시 데이터를 UI에 표시하기에 적합한 사용자 친화적인 포맷으로 변환할 수 있습니다. 타입 컨버터는 데이터 소스와 UI 간의 중개자 역할을 하며, 데이터를 사용자에게 더 직관적인 포맷으로 변환합니다.

예를 들어 타입 컨버터는 라디안을 도 단위로 전환하거나 원시 체력 값을 체력 바에 표시할 컬러로 전환할 수 있습니다. 그렇게 하면 명확하고 이해하기 쉬운 포맷으로 정보를 표시하는 UI를 구현할 수 있습니다. 타입 컨버터가 변환을 수행하는 데는 많은 수동 변환 로직이 필요하지 않습니다.

Unity 6는 두 가지 유형의 타입 컨버터를 지원합니다.

— **글로벌 컨버터:** 특정 타입 전환이 필요한 모든 바인딩에 이 컨버터를 적용합니다. 예를 들어 글로벌 컨버터는 모든 float 형식의 체력 백분율 값을 컬러 값으로 전환하거나 `Color` 오브젝트를 `StyleColor` 타입으로 전환하여 UI 전반에서 일관된 동작이 이루어지도록 할 수 있습니다.

— **바인딩별 컨버터:** 더욱 세밀한 제어를 위해 특정 데이터 바인딩에 적용합니다.

예시: 값을 컬러로 전환

플레이어의 체력에 따라 다른 컬러를 표시하는 체력 바는 데이터 바인딩을 타입 컨버터와 함께 사용하는 사례를 보여 줍니다. 플레이어의 현재 체력을 컬러 그레디언트에 매핑하면(예: 초록색 - 체력 높음, 노란색 - 체력 낮음, 빨간색 - 빈사 상태) 플레이어는 게임플레이 중에 자신의 상태를 빠르게 파악할 수 있습니다.

이 동작은 QuizU 프로젝트의 **DataBindingDemo** 씬에서 설정할 수 있습니다.

HealthDataConverter 설정

DataBindingDemo 씬에서 `HealthBarWithConverter` 클래스는 정적 `HealthDataConverter`의 기능을 사용하여 몇 가지 DataConverter를 등록합니다.

— 체력 백분율에 따라 체력 바의 컬러 그레디언트가 초록색(체력 높음)에서 빨간색(빈사 상태)까지 점진적으로 변합니다.

— 한 레이블은 수치 값을 백분율 문자열로 표현합니다(예: ‘75%’).

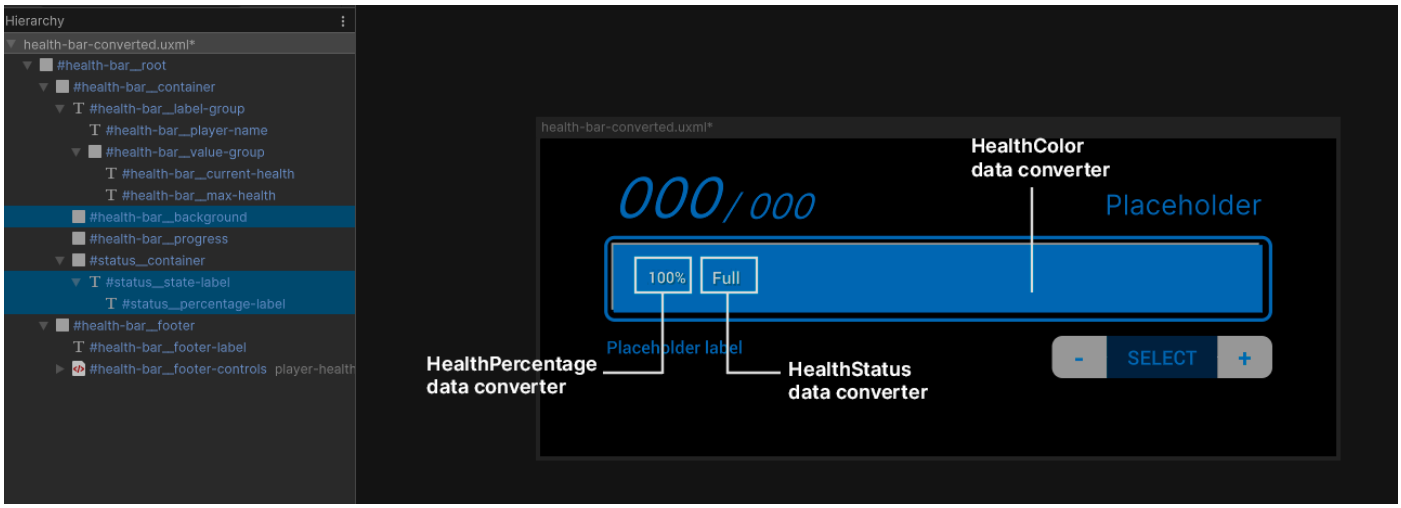
— 또 다른 레이블은 체력 백분율을 ‘Full’, ‘Mid’, ‘Critical’과 같은 상태 레이블로 매핑합니다.

다음은 HealthDataConverter 클래스의 스니핏입니다.

```
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);
    }

    // …

}
```

위 로직은 `float` 체력 백분율(0부터 1)을 빨간색(빈사 상태)과 초록색(체력 높음) 사이의 일치하는 `StyleColor` 값으로 변환하는 `HealthColor` ConverterGroup을 생성합니다.

`HealthDataConverter` 클래스에는 두 레이블을 위한 컨버터도 있습니다. 이를 통해 `PlayerDataSO`의 `HealthPercentage` 프로퍼티를 서식이 지정된 문자열 값으로 나타낼 수 있습니다. 여러 컨버터를 하나의 ConverterGroup으로 묶을 수도 있지만, 이 데모에서는 가독성을 위해 별도의 ConverterGroup으로 분리했습니다.



UI 빌더에서 타입 컨버터를 사용합니다.

HeathBarWithConverter 사용

`HealthDataConverter` 클래스에는 실제 기능이 포함되어 있습니다. `HealthBarWithConverter`는 다음과 같습니다.

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

위 코드에서 다음과 같은 사실을 확인할 수 있습니다.

— `UnityEditor.InitializeOnLoadMethod`는 ConverterGroup이 등록되고 UI 빌더에서 사용할 수 있게 하며, 사용자는 이를 에디터에서 보고 적용할 수 있습니다.

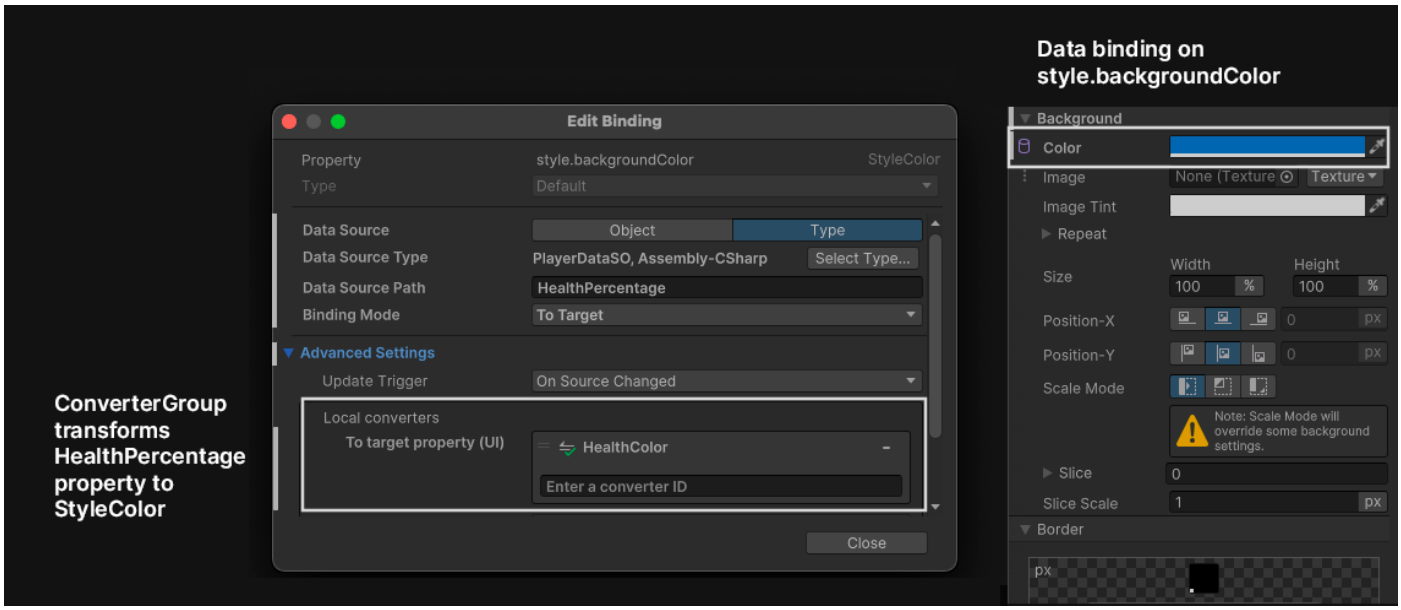
— `RuntimeInitializeOnLoadMethod`를 사용하면 게임이 실행 중일 때 런타임에 ConverterGroup을 사용할 수 있습니다.

`#if UNITY_EDITOR` 프리 프로세서 지시문은 코드가 에디터에서 실행되는지, 또는 게임플레이 중에 실행되는지에 따라 적절한 메서드가 실행되도록 합니다.

UI 빌더에서 DataConverter 적용

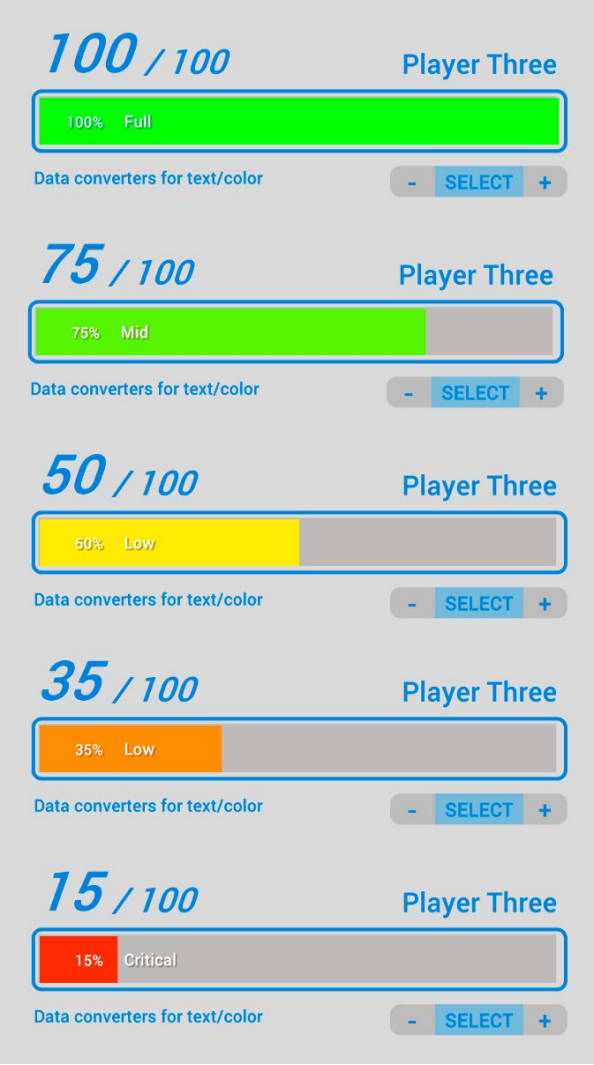
등록된 DataConverter는 이 전환이 필요한 모든 바인딩에 적용할 수 있습니다. UI 빌더에서 직접 사용하는 방법은 다음과 같습니다.

1. UXML 파일을 열고 진행 표시줄 요소를 선택합니다. QuizU 프로젝트에서는 **RuntimeDataBinding.uxml** 파일을 열어서 설정된 방식을 살펴볼 수 있습니다.
2. 데이터 소스를 `PlayerDataSO` ScriptableObject로 설정합니다.
3. 진행 표시줄의 `backgroundColor` 스타일 프로퍼티를 `HealthPercentage` 데이터 경로에 바인딩합니다.
4. `HealthColor` ConverterGroup을 사용하여 체력 백분율 값을 진행 표시줄에 적용할 컬러 배경으로 변환합니다.



체력 바의 데이터 바인딩을 설정합니다.

5. 이제 `PlayerDataSO` ScriptableObject의 `CurrentHealth` 값을 드래그하면 체력 바 컬러가 업데이트됩니다. 그레디언트는 초록색(체력 높음)에서 노란색(체력 중간)을 거쳐 주황색(체력 낮음)과 빨간색(빈사 상태)으로 매끄럽게 선형 보간됩니다.



HealthColor 컨버터로 진행 표시줄 컬러를 변경할 수 있습니다.

이제 float 값을 이 컬러 그레디언트로 전환해야 하는 애플리케이션 내 모든 위치에서 이 글로벌 DataConverter를 사용할 수 있습니다.

베스트 프랙티스

타입 컨버터를 사용할 때 유의할 점은 다음과 같습니다.

- **할당 최소화하기:** 특히 연산이 자주 이루어지는 경우에는 전환 델리게이트를 가볍게 유지해야 불필요한 성능 오버헤드를 줄일 수 있습니다.
- **단순하게 유지하기:** 빠른 변환을 위해 단순하고 필요한 작업에만 집중하는 컨버터를 만듭니다. 복잡하거나 많은 리소스를 사용하는 로직을 포함하지 않습니다.
- **전환을 데이터 소스에 통합하기:** 기본적인 전환은 데이터 소스 자체에서 처리합니다(예: 체력 백분율은 ScriptableObject 프로퍼티에서 사전에 포맷 지정하기). DataConverter는 UI 바인딩에 필요한 전환용으로만 사용합니다.

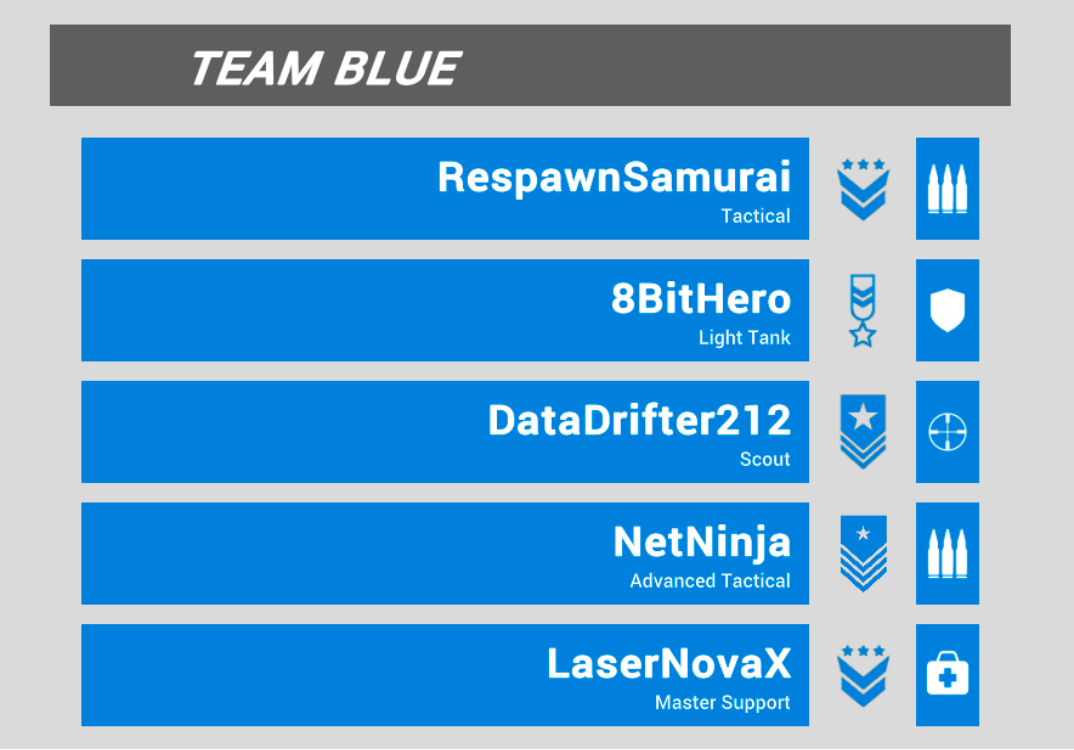
예시: ListView에 목록 바인딩

게임 UI에 따라(예: 수집된 아이템 인벤토리, 목표를 트래킹하는 퀘스트 로그, 플레이어의 순위를 보여 주는 리더보드 등) 애플리케이션에서 화면에 여러 데이터 컬렉션을 표시해야 할 수 있습니다.

ListView는 이 정보를 쉽게 관리하고 표시할 수 있는 깔끔한 스크롤 인터페이스를 제공합니다. Unity 6는 데이터가 변경되었을 때 UI를 새로고침하기 위해 수동으로 업데이트하거나 커스텀 스크립트를 작성할 필요가 없도록 런타임 데이터 바인딩을 사용하여 이 프로세스를 간소화합니다.

이전 버전의 Unity에서 ListView를 설정하려면 목록을 채우고 데이터 변경에 따른 업데이트를 처리하기 위해 커스텀 코드를 작성해야 했습니다. Unity 6에서는 ListView가 데이터 소스에 직접 바인딩되어 자동으로 변경 사항을 트래킹하고 UI에 반영합니다.

이 데모 씬에는 PlayerDataSO ScriptableObject의 목록을 바인딩하는 간단한 ListView가 있습니다. 이 ListView를 사용하여 멀티플레이어 게임 로비나 고득점 리더보드에서 볼 수 있는 인터페이스를 만들 수 있습니다.



TeamList는 ListView를 사용하여 목록을 바인딩합니다.

런타임 데이터 바인딩을 사용하면 ListView를 ScriptableObject와 같은 데이터 소스에 직접 연결할 수 있습니다. ListView가 자동으로 데이터의 변경 사항을 트래킹하여 설정 및 유지 관리를 간소화합니다.

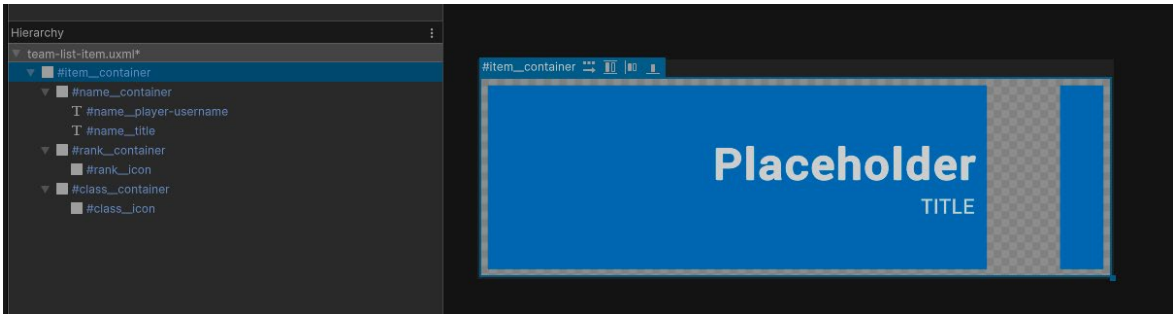
ListView를 목록에 데이터 바인딩 처리하려면 미지정 바인딩을 설정한 다음 런타임에 데이터 바인딩을 완료해야 합니다.

목록과 템플릿 설정

아래 단계에 따라 데이터 바인딩을 위해 ListView를 준비하세요.

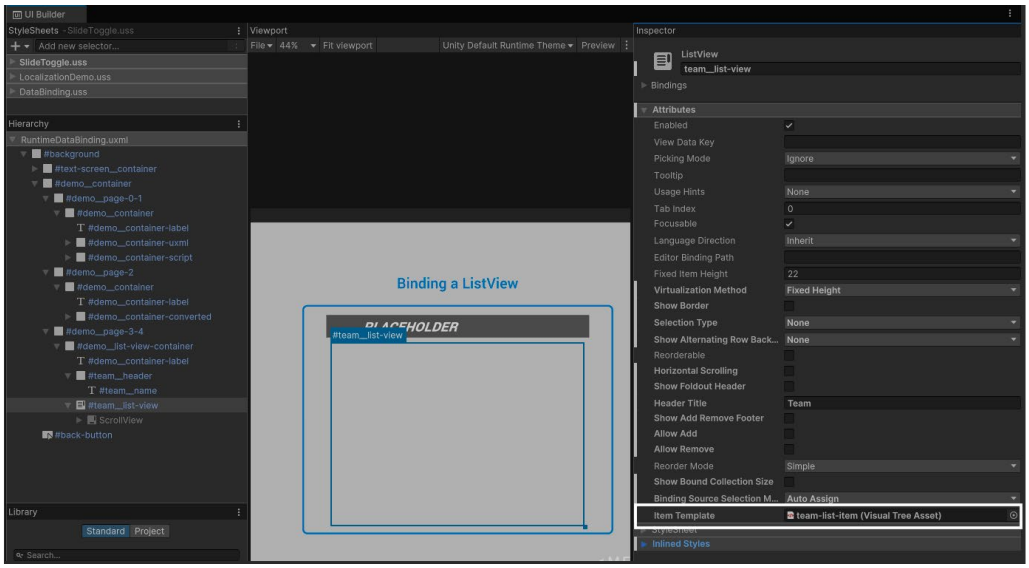
1. **데이터 소스 정의하기**: ListView에는 데이터 목록이 필요합니다. 이 데모에서는 TeamSO ScriptableObject에 PlayerDataSO 오브젝트 목록이 저장됩니다. 이 목록의 각 항목은 ListView의 행 하나에 대응합니다.

2. **UXML 항목 템플릿 생성하기**: UI 빌더에서 단일 목록 항목을 어떤 모습으로 표시할지 정의하는 UXML 템플릿(VisualTreeAsset)을 디자인합니다. 예를 들어 이 데모의 `team-list-item` 템플릿에는 플레이어의 이름과 몇몇 Texture2D 프로퍼티가 있습니다. 데이터 소스를 직접 참조하는 대신, UI 빌더에서 Data Source Type과 Data Source Path를 설정합니다. 이렇게 하면 바인딩이 미지정 상태로 남으며, 이후 런타임에 완성되도록 할 수 있습니다.



UI 빌더에서 시각적 트리 에셋을 디자인합니다.

3. **메인 사용자 인터페이스에 ListView 추가하기**: 또 다른 UXML 파일에 전체 플레이어 목록을 표시할 ListView 요소를 추가합니다. 항목 템플릿을 ListView의 **Item Template**으로 할당합니다. 이 시점에서 ListView는 각 행이 어떻게 보일지는 인지하지만 구체적으로 어떤 데이터 소스를 사용할지는 알지 못합니다.



ListView에 템플릿을 추가합니다.

데모 씬의 ListView는 위에 보이는 몇 가지 기본적인 설정만 사용합니다. 고급 기능을 살펴보려면 공식 ListView 기술 자료를 참조하세요.

런타임에 바인딩 완료

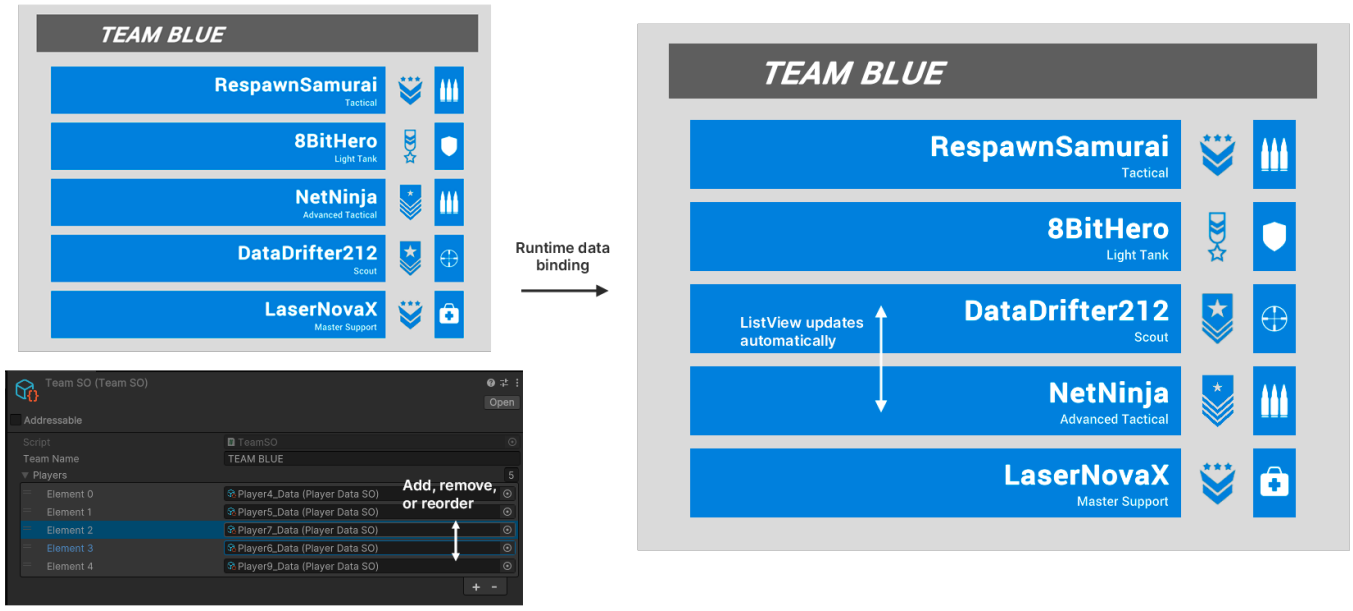
런타임에는 간단한 `TeamList` 스크립트가 실제 데이터 소스를 제공하여 바인딩을 완료합니다. 다음과 같은 코드 줄이 앞에서 지정되지 않은 바인딩을 완료합니다.

```
// 데이터 소스 설정
m_ListView.dataSource = m_TeamData;

// 'itemsSource'를 Players 목록에 바인딩
m_ListView.SetBinding("itemsSource", new DataBinding
{
    dataSourcePath = new PropertyPath("Players")
});
```

여기서 (TeamSO의 인스턴스인) `m_TeamData`가 ListView에 할당되었습니다. `SetBinding`을 한 번 호출하면 `Players` 프로퍼티가 itemsSource에 연결됩니다. 그러면 ListView가 UI의 행에 데이터를 채울 수 있습니다.

UXML의 바인딩은 런타임 전까지 미지정 상태로 남기 때문에 각 목록 요소를 개별적으로 연결할 필요가 없습니다. UI 툴킷이 자동으로 바인딩을 확인하여 모든 목록 항목에 데이터를 채웁니다.

데이터 소스의 목록에 변경 사항이 적용되면(예: 플레이어 추가, 삭제 또는 재정렬) 스크립팅이 추가되는 일 없이 변경 사항이 UI에 즉시 반영됩니다.



UI는 Players 목록의 변경 사항을 반영합니다.

데이터 바인딩에 대한 이 하이브리드 접근 방식은 다량의 반복적인 상용구 코드를 줄일 수 있습니다. UXML에서 플레이스홀더 데이터 경로를 설정하여 실제 데이터 소스가 런타임에 할당되도록 미룰 수 있습니다.

데이터 모델을 변경하더라도 전체 바인딩 로직을 재작성할 필요가 없습니다. 시작 시점에 한 번만 업데이트하면 새로운 소스에 UI를 다시 연결할 수 있습니다.

ListView와 목록 바인딩에 대해 자세히 알아보려면 이 기술 자료 페이지를 참고하세요.

데이터 바인딩 최적화

효율적인 바인딩은 UI의 성능을 보장하는 데 도움이 됩니다. 중복되거나 지나친 바인딩은 시스템 과부하를 유발하며 불필요한 업데이트와 성능 저하로 이어질 수 있습니다. 이러한 점은 인터페이스가 복잡하거나 사용되는 리소스가 많을 때 특히 중요합니다.

런타임 바인딩 시스템은 기본적으로 프레임마다 UI 요소를 업데이트합니다. 이에 따라 규모가 작은 애플리케이션에서는 반응성을 향상할 수 있지만, 바인딩이 추가되면 성능 병목 현상이 발생할 수 있습니다.

이번 섹션에서는 대규모 프로젝트에서 데이터 바인딩의 효율성을 높이는 방법을 알아봅니다.

값 유형 관리

데이터 소스에서 값 유형(int, float, struct 등)을 사용한다면 박싱에 드는 비용에 주의해야 합니다. dataSource 프로퍼티는 오브젝트로 작동하므로 값 유형이 자주 전환되면 오버헤드가 늘어날 수 있습니다.

오버헤드를 줄이려면 값 유형 프로퍼티를 사용할 때 불필요한 바인딩이나 중복된 업데이트를 최소화해야 합니다.

오버헤드 최소화

먼저 같은 요소를 여러 번 업데이트하거나 변경이 드문 데이터를 트래킹하는 바인딩을 식별합니다. 그러한 바인딩을 결합하거나 제거하여 불필요한 작업을 줄이세요. 가능한 한 복잡한 계층 구조 대신 평면적이고 단순한 데이터 구조를 사용합니다. 그러면 데이터를 자주 조회하느라 발생하는 성능 병목 현상을 방지할 수 있습니다.

많은 계산이 필요한 값은 미리 계산하거나 캐시하는 방안을 고려합니다. 이렇게 미리 계산된 값에 바인딩하면 바인딩 시스템의 연산 부하를 줄이고 반복적인 재계산을 방지할 수 있습니다.

자주 업데이트해야 하는 요소만 바인딩합니다. 지속적인 동기화가 필요하지 않은 요소에서는 불필요한 바인딩을 없애고 값을 직접 할당하거나 이벤트에 의해 트리거된 경우에만 업데이트하세요.

업데이트 트리거 사용

바인딩은 업데이트 트리거에 따라 새로고침되며, 이에 따라 UI가 데이터 소스와 동기화되는 빈도가 결정됩니다. 이를 통해 성능과 반응성의 적절한 균형을 유지할 수 있습니다. 바인딩의 업데이트 빈도는 다음 옵션에 따라 결정됩니다.

- **Every frame:** 지속적으로 업데이트합니다. 예시의 체력 바처럼 지속적인 업데이트가 필요한 요소에 사용하세요.
- **On change detection:** 데이터 소스가 변경될 때 업데이트하거나, 변경을 감지할 수 없는 경우 프레임마다 업데이트합니다. 관찰 가능한 데이터를 활용하는 스탯 패널이나 인벤토리 목록 등에 사용하세요.
- **When marked as dirty:** 업데이트가 자주 이루어지지 않는 시나리오에서 MarkDirty를 사용하여 바인딩을 더티로 명시하면 불필요한 새로고침 사이클이 발생하지 않습니다. 이 업데이트 트리거는 특정 컨텍스트에서만 변경되는 설정 메뉴와 같은 요소에 적합합니다.

UI 요소별로 필요에 맞게 업데이트 트리거를 사용하면 반응성과 효율성의 균형을 적절하게 유지할 수 있습니다.

버전 관리와 변경 트래킹

불필요한 업데이트를 줄이기 위해 버전 관리와 변경 트래킹을 데이터 소스에 통합할 수 있습니다.

다음 두 가지 인터페이스가 데이터 바인딩의 효율을 높이는 데 도움이 됩니다.

- IDataSourceViewHashProvider: 버전 해시를 사용하여 전반적인 변경을 트래킹하며, 데이터 소스가 변경된 경우에만 업데이트를 트리거합니다. 업데이트가 자주 이루어지지 않는 정적 또는 반정적 데이터에 적합합니다.
- INotifyBindablePropertyChanged: 프로퍼티 레벨에서 변경을 트래킹하며, 영향을 받는 바인딩이 새로고침되도록 합니다. 이 경우 세밀한 제어가 가능합니다.

위와 같은 인터페이스를 데이터 소스에 추가하세요. 개별적으로 사용할 수도 있고, 함께 사용하여 업데이트를 더욱 세밀하게 제어할 수도 있습니다. 사용법과 베스트 프랙티스는 이 기술 자료 페이지를 참조하세요.

팁: 더 많은 UI 툴킷 최적화 팁

UI 툴킷 최적화를 다루는 이 유나이트 2024 발표에서는 연계된 드로우 콜 구현, 버퍼 크기의 영향, 동적 아틀라스 베스트 프랙티스, 커스텀 셰이더와 3D UI 같은 제약에 대응하는 법 등 여러 주제에 대해 알아볼 수 있습니다.

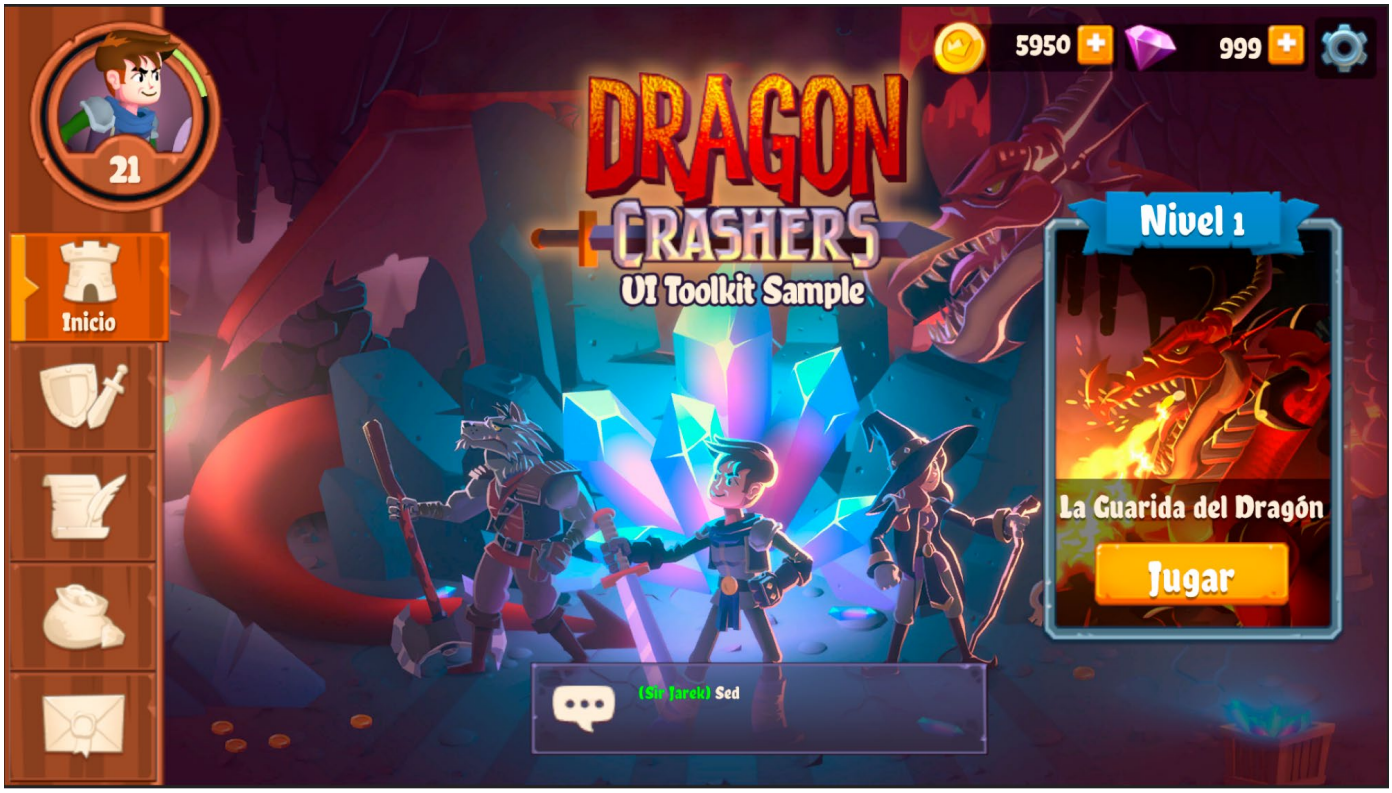
현지화

UI를 현지화하면 더 쉽게 전 세계 사용자를 대상으로 게임을 선보일 수 있으며, 어떤 언어로든 애플리케이션을 직관적이면서도 익숙하게 사용하도록 지원할 수 있습니다.

Unity 6는 UI 툴킷에 Localization 패키지를 직접 통합하여 이 프로세스를 간소화합니다. 이를 통해 개발자는 플레이어가 어디에 있든 상관없이 각자에게 맞는 지역별 콘텐츠를 제공할 수 있습니다.

Unity 현지화의 핵심은 Locale 클래스입니다. Locale 클래스는 특정 언어를 나타내며, 통화, 숫자 포맷 지정과 같은 지역별 세부 사항을 관리합니다.

UI 툴킷에서 현지화를 설정하는 방법을 보여 주는 간단한 예시를 살펴보겠습니다. 이 예시에서는 선택된 로케일에 따라 앱의 콘텐츠가 동적으로 조정됩니다.



UI 툴킷 샘플 - 드래곤 크래셔의 스페인어 현지화 예시

작동 방식

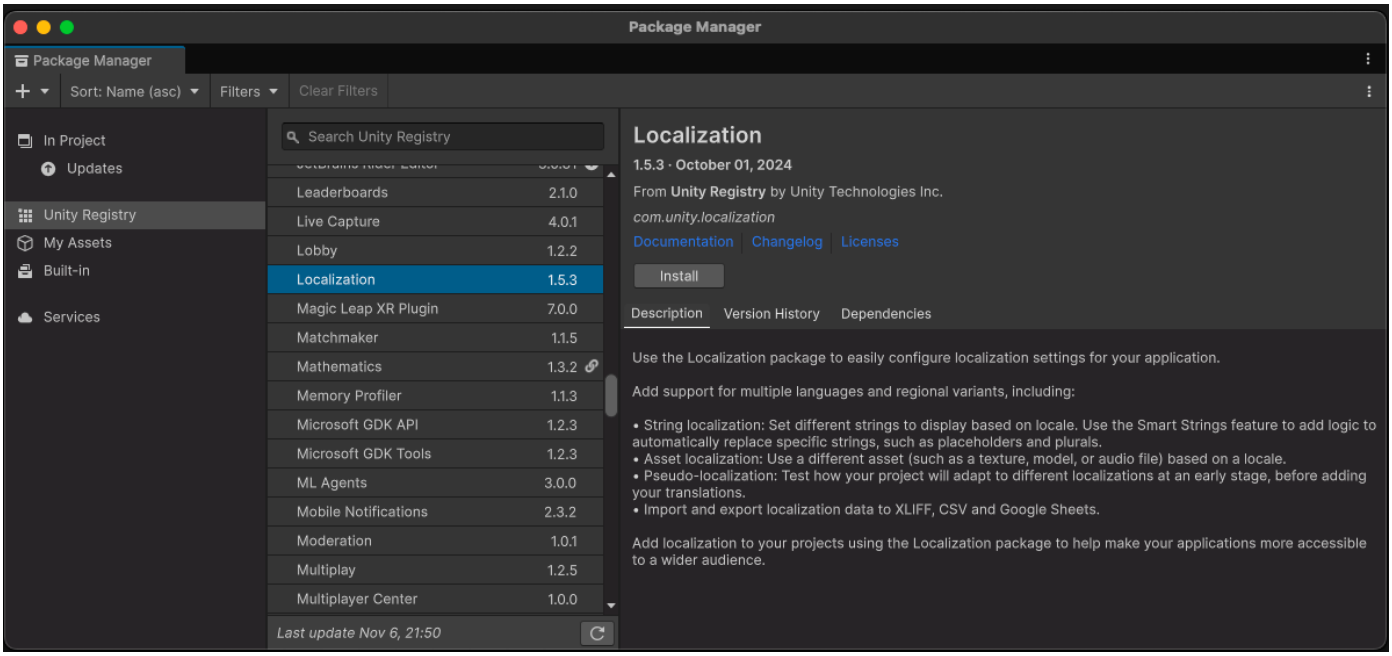
다음은 Localization 패키지의 주요 특징입니다.

- **문자열 현지화:** LocalizedString 클래스를 사용하면 런타임에 로케일이 전환되었을 때 자동으로 업데이트되는 문자열을 관리할 수 있습니다. Smart Strings를 사용하면 플레이스홀더를 추가하고, 복수형을 처리하고, 그 밖의 언어별 뉘앙스에 맞게 조정할 수 있습니다.
- **에셋 현지화:** 로케일에 따라 텍스처와 기타 에셋을 전환하여 단순히 번역된 텍스트 이상의 지역별 콘텐츠를 만들 수 있습니다.
- **데이터 바인딩:** Localization 패키지는 UI 툴킷의 런타임 데이터 바인딩에 통합되어 UI 요소를 String Table 및 Asset Table에 연결합니다. 데이터, 로케일 또는 로드 상태를 변경하면 자동 업데이트가 트리거됩니다.
- **String Table 및 Asset Table 관리:** String Table과 Asset Table에는 텍스트 또는 기타 에셋을 그에 대응하는 로케일별 항목으로 변환하는 데 필요한 키-값 쌍이 저장되어 있습니다. 일원화된 UI 인터페이스로 프로젝트에 포함된 모든 현지화된 텍스트 및 에셋에 대한 대략적인 개요를 확인할 수 있습니다.
- **로케일 전환:** 애플리케이션을 다시 시작할 필요 없이 실시간으로 언어를 전환합니다. 런타임에 새 로케일을 선택하는 즉시 변경 사항이 반영되어 UI가 업데이트됩니다.

텍스트 길이와 포맷의 변경에 대응할 때는 UI 툴킷의 FlexBox 컨테이너와 자동 크기 조정 요소를 활용하세요. 이렇게 하면 여러 언어를 지원할 때 UI 반응성을 향상할 수 있습니다.

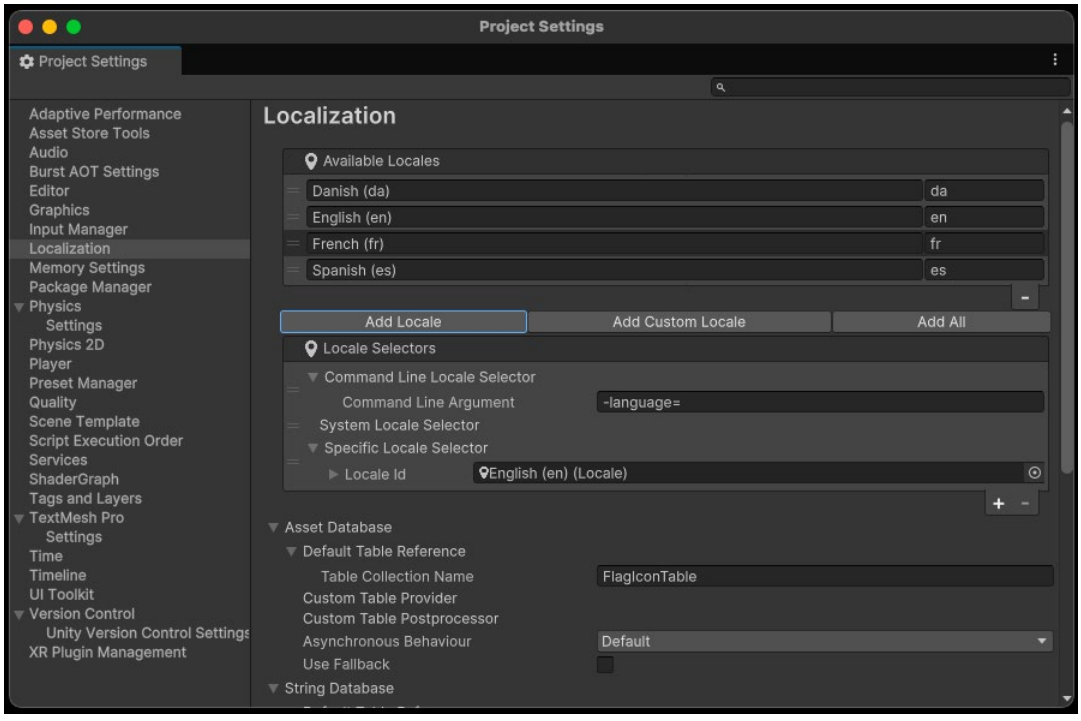
Localization 설정

Unity의 Localization 패키지를 UI 툴킷과 함께 사용하려면 현지화된 콘텐츠를 설정하고 UI 빌더에서 UI 요소에 바인딩하는 기본적인 단계를 따라야 합니다.



패키지 관리자에서 Localization 패키지를 설치합니다.

1. **Localization Settings 조정하기:** 패키지 관리자에서 Localization 패키지를 설치한 다음 **Project Settings > Localization**으로 이동하여 Localization Settings 에셋을 생성하고 구성합니다. 이를 통해 모든 현지화된 에셋을 관리할 수 있습니다.

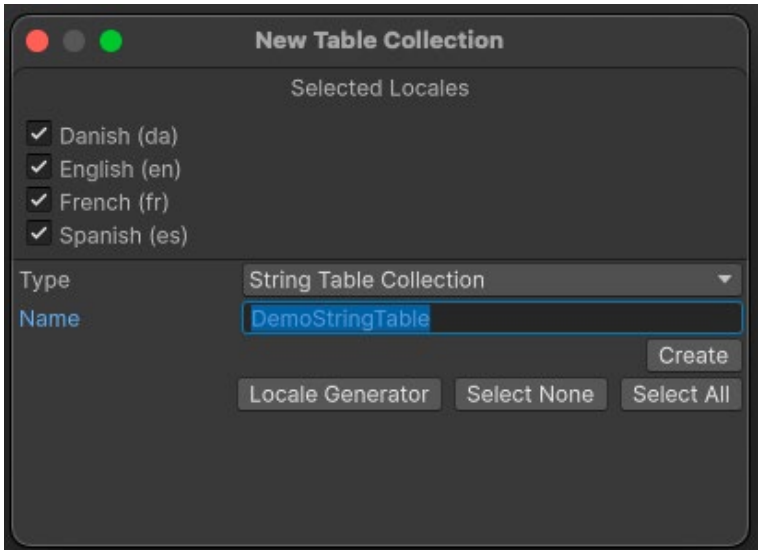


Project Settings에서 Localization Settings를 생성하고 구성합니다.

2. **로케일 생성하기:** Locale Generator를 사용하여 프로젝트에서 지원할 언어와 지역을 정의합니다. 이렇게 하면 로케일별로 고유한 2개 문자 코드로 식별되는 에셋이 생성됩니다(예: 영어는 'en', 프랑스어는 'fr', 스페인어는 'es' 등). 애플리케이션이 시작할 때 사용할 기본 로케일을 설정하세요.

로케일을 추가합니다.

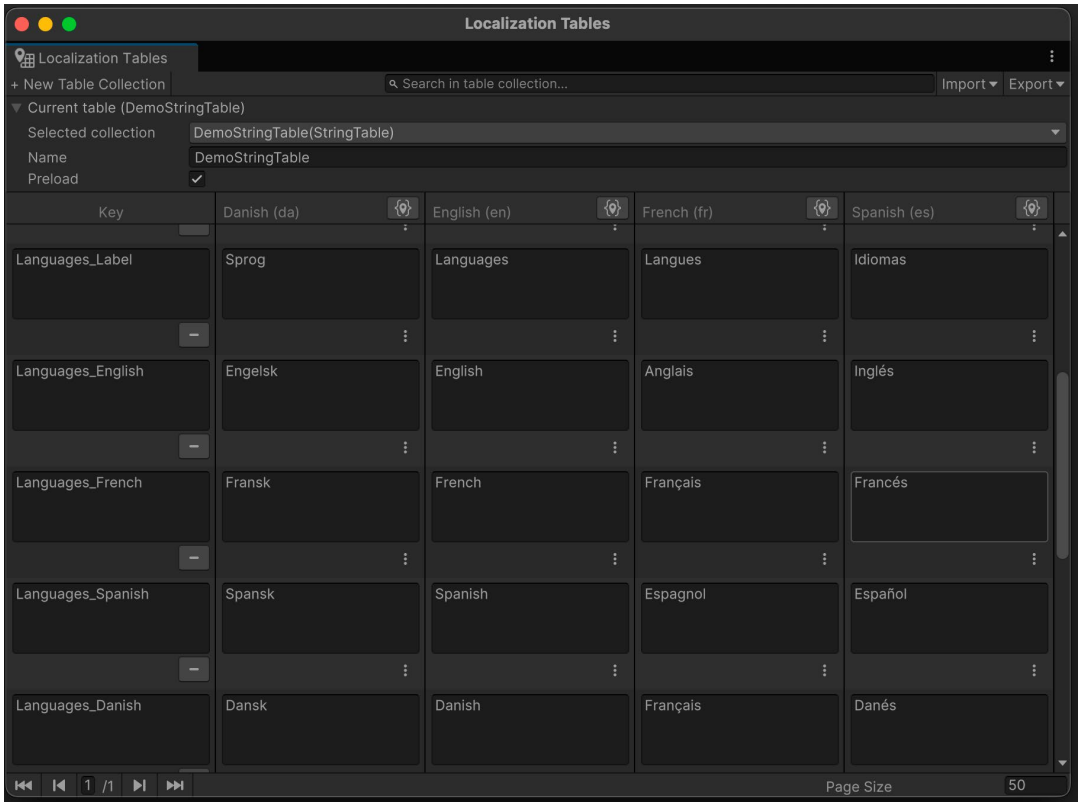
3. **String Table 및 Asset Table 생성하기:** String Table을 사용하여 각 로케일의 텍스트 항목을 저장합니다. 레이블, 버튼, 드롭다운 옵션과 같은 UI 텍스트 요소의 항목을 추가합니다.



String Table과 Asset Table을 생성합니다.

4. Localization Tables 창(**Window > Asset Management > Localization Tables**)에서 UI의 각 텍스트 요소에 대한 키-값 쌍을 추가합니다. 각 키는 특정 텍스트 항목(레이블, 버튼 등)을 나타내고, 각 값은 각 로케일에서 해당 항목이 번역된 텍스트입니다.

5. 이미지나 게임 오브젝트와 같은 영역별 에셋을 Asset Table에 저장합니다. 예를 들어 각 로케일을 나타내는 아이콘으로 스프라이트 또는 텍스처를 추가할 수 있습니다. 각 키에 대해, 지역 또는 문화적 선호도를 반영하는 로케일별 에셋을 연결합니다.

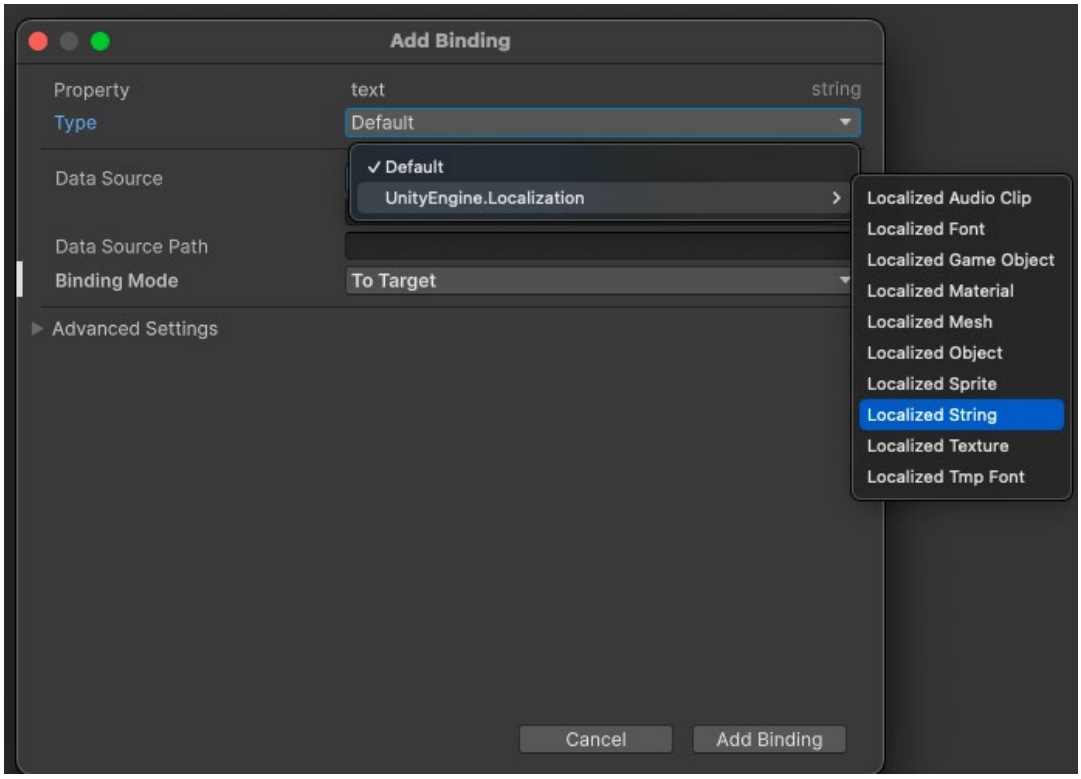


키-값 쌍은 현지화할 각 요소를 나타냅니다.

6. **UXML 인터페이스 정의하기:** UI 빌더를 사용하여 Button, DropdownField, Label 등의 요소가 있는 UXML 파일을 생성합니다. 데모 씬의 이 UXML에는 현지화할 수 있는 몇 가지 요소가 있습니다. 텍스트 필드에는 String Table의 항목을 사용하고, 텍스처와 같은 텍스트가 아닌 필드에는 Asset Table을 사용합니다.

데모 씬
QuizU 프로젝트에 포함된 **LocalizationDemo** 씬에서 샘플 Localization 구현을 살펴볼 수 있습니다.

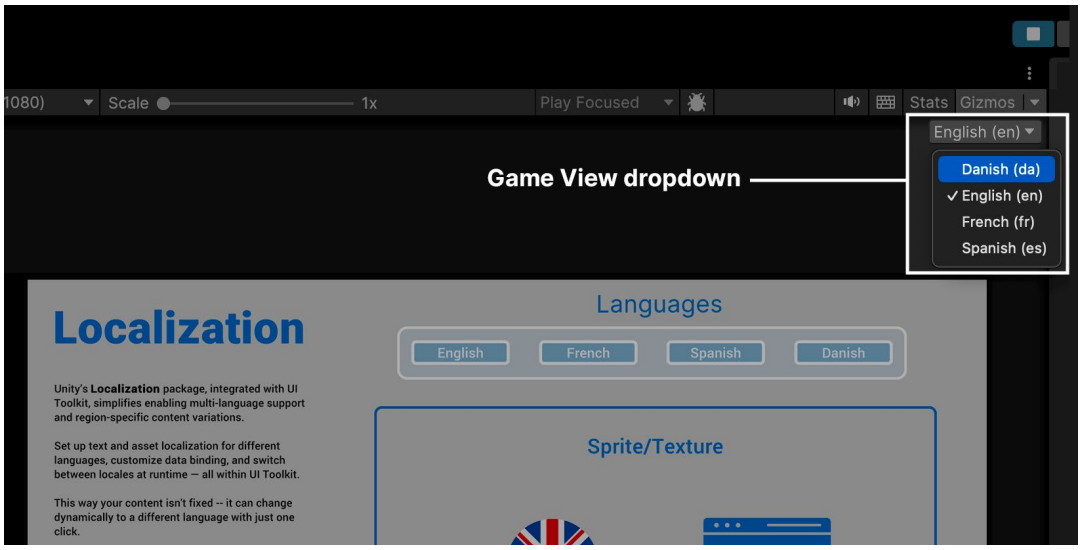
여기에 액세스하려면 런타임에 메인 메뉴로 이동하여 **Demos > Localization**을 선택하거나 부트로더를 비활성화한 직후에 **LocalizationDemo** 씬을 로드합니다(**Quiz > Don't Load Bootstrap Scene on Play**).



UI 빌더에서 현지화된 문자열과 에셋에 데이터 바인딩을 추가합니다.

7. **UI 빌더에서 UI 요소에 데이터 바인딩하기:** 이제 UI 툴킷의 강력한 런타임 데이터 바인딩 시스템을 사용할 차례입니다. UI 빌더에서 현지화하려는 요소를 선택합니다. 인스펙터 패널을 열고 콘텐츠 필드에서 **Add Binding**을 선택합니다(예: 레이블의 경우 `text` 또는 이미지의 경우 `backgroundImage`).

   바인딩 유형으로 **LocalizedString** 또는 그 밖의 현지화된 에셋을 선택하고, String Table 또는 Asset Table에서 이에 대응하는 항목에 연결합니다. 현지화해야 하는 요소 개수만큼 테이블에 항목을 추가합니다.



Game View 로케일 드롭다운 목록을 사용하여 현지화 옵션을 미리 봅니다.

8. 테스트하려면 Game View 로케일 드롭다운 목록을 사용하여 여러 언어로 UI를 미리 보며 각 로케일에서 요소들이 올바르게 표시되는지 확인합니다.

여기까지가 기본적인 설정입니다. 이 예시에서 이제 버튼과 레이블의 text 프로퍼티를 다른 구성된 언어로 전환할 수 있습니다. UI 전체를 현지화하려면 String Table에 모든 텍스트에 대응하는 항목이 있어야 합니다.

Localization 패키지의 콘텐츠는 유연하게 구성할 수 있습니다. String Table을 여러 개 사용하여 대규모 프로젝트를 관리하기 편하도록 작은 섹션으로 분할하거나 다양한 항목으로 분류하세요. 그런 다음 Asset Table을 사용하여 텍스처를 비롯한 텍스트 외 에셋을 현지화하세요.

UI 빌더에서 현지화 바인딩을 추가하면 UXML 파일이 직접 UI 요소에 현지화를 통합합니다. 그러면 다음과 같은 코드 블록이 생성되며, 현지화된 각 프로퍼티가 String Table 또는 Asset Table의 특정 엔트리에 연결됩니다.

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

이 UXML 스니핏은 UI 요소의 text 프로퍼티를 String Table 또는 Asset Table의 항목에 연결하는 데이터 바인딩을 구현합니다.

현지화 테이블을 업데이트할 때마다 연결된 UI 요소가 현지화된 최신 콘텐츠를 자동으로 반영합니다.

참고: UI 빌더로 데이터 바인딩 생성 과정을 간소화할 수는 있으나, 숙련된 사용자라면 현지화된 콘텐츠를 세밀하게 제어해야 할 때 UXML을 직접 편집하는 것이 나을 수 있습니다.

Localizatizon API 사용

에디터의 Game View 로케일 드롭다운 목록은 여러 언어를 테스트하는 용도로 바람직하지만, 애플리케이션 빌드에 포함되지는 않습니다. 최종 애플리케이션에서 사용자가 언어를 변경할 수 있게 하려면 로케일 전환에 필요한 자체 UI를 생성해야 합니다.

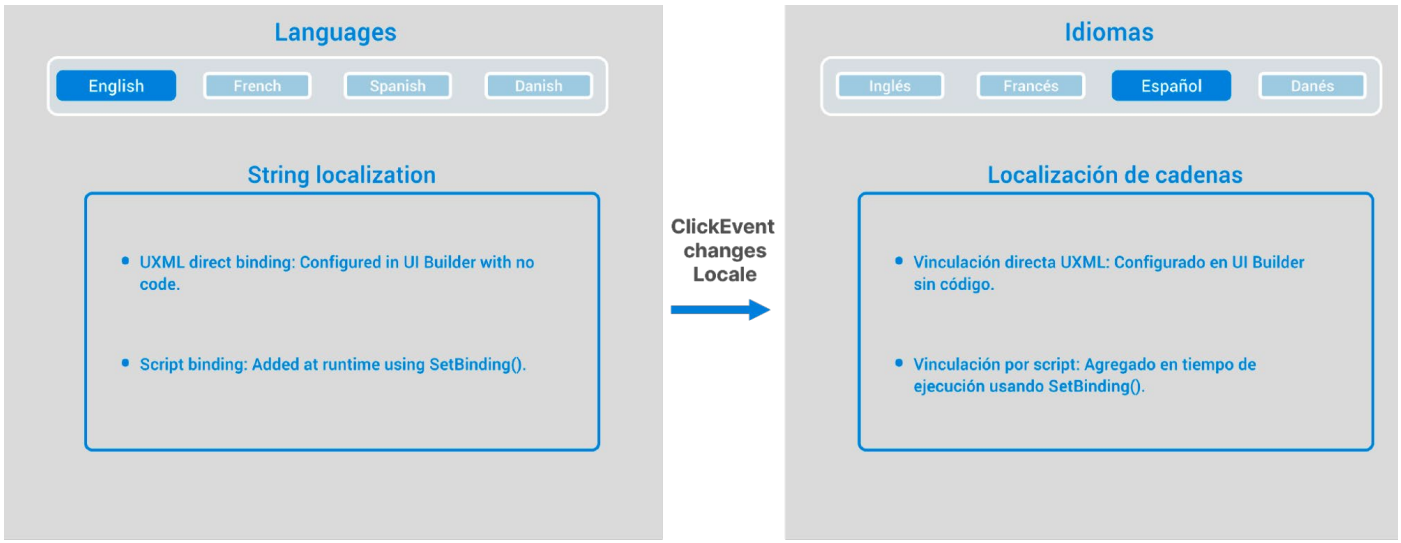
로케일 선택

로케일의 2개 문자 식별자를 알고 있으면 LocalizationSettings에서 활성 로케일을 설정할 수 있습니다. 그런 다음 각 버튼의 clicked 매니퓰레이터 또는 RegisterCallback<ClickEvent> 메서드를 사용하여 이 동작을 버튼에 연결합니다.

샘플 프로젝트의 LocalizationDemo 스크립트에서는 다음과 같은 구현 방식을 확인할 수 있습니다.

```
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");
}
```

각 버튼을 클릭하여 로케일을 지정된 로케일로 변경할 수 있습니다. English, French, Spanish 또는 Danish 라는 이름의 버튼을 누르면 런타임에 UI의 텍스트가 변경됩니다.



버튼을 눌러 로케일을 변경할 수 있습니다.

SetBinding 사용

UI 빌더를 사용하면 데이터 바인딩을 대화식으로 손쉽게 설정할 수 있습니다. 그러나 런타임에 스크립트를 통해 바인딩을 설정해야 하는 경우도 있습니다. 예를 들어 UI 요소를 동적으로 생성하거나, 게임플레이 중에만 사용 가능한 데이터를 활용하는 바인딩이 있을 수 있습니다.

C#으로 데이터 바인딩을 설정하려면 시각적 요소에 대해 SetBinding 메서드를 사용합니다. Label의 text 프로퍼티를 StringTable의 LocalizedString 항목에 바인딩하는 방법은 다음과 같습니다.

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


public class LocalizationDemo : MonoBehaviour
{
    // 인스펙터에서 설정
    [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);
    }
}
```

이 설정의 `m_LocalizedText`는 인스펙터에서 DemoStringTable의 항목에 할당됩니다. 이 코드는 `m_LocalizedLabel`의 text 프로퍼티를 지정된 LocalizedString에 연결합니다. 이에 따라 로케일이 변경되면 텍스트가 자동으로 업데이트됩니다.

로케일 변경 수신 대기

로케일이 변경된 경우 현지화된 문자열을 업데이트하는 것에 더해 추가적인 조치를 취해야 하는 경우가 있습니다. 로케일이 업데이트될 때마다 로직을 실행하려면 LocalizationSettings API의 `SelectedLocaleChanged` 이벤트를 수신 대기하면 됩니다.

예시를 들어 보겠습니다.

```
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)
    {
        // 로케일이 변경되면 작업 수행(예: UI 요소 업데이트)
        Debug.Log($"Locale changed to: {newLocale.Identifier.Code}");
    }
}
```

여기서 로케일이 변경될 때마다 다른 요소를 업데이트하거나 커스텀 로직을 실행할 수 있도록 OnLocaleChanged가 호출됩니다. 이 이벤트 핸들러를 사용하여 UI 프로퍼티나 스타일을 조정하세요. 특히 번역된 텍스트가 현재 레이아웃에 맞지 않을 때 유용합니다.

String Table 사용

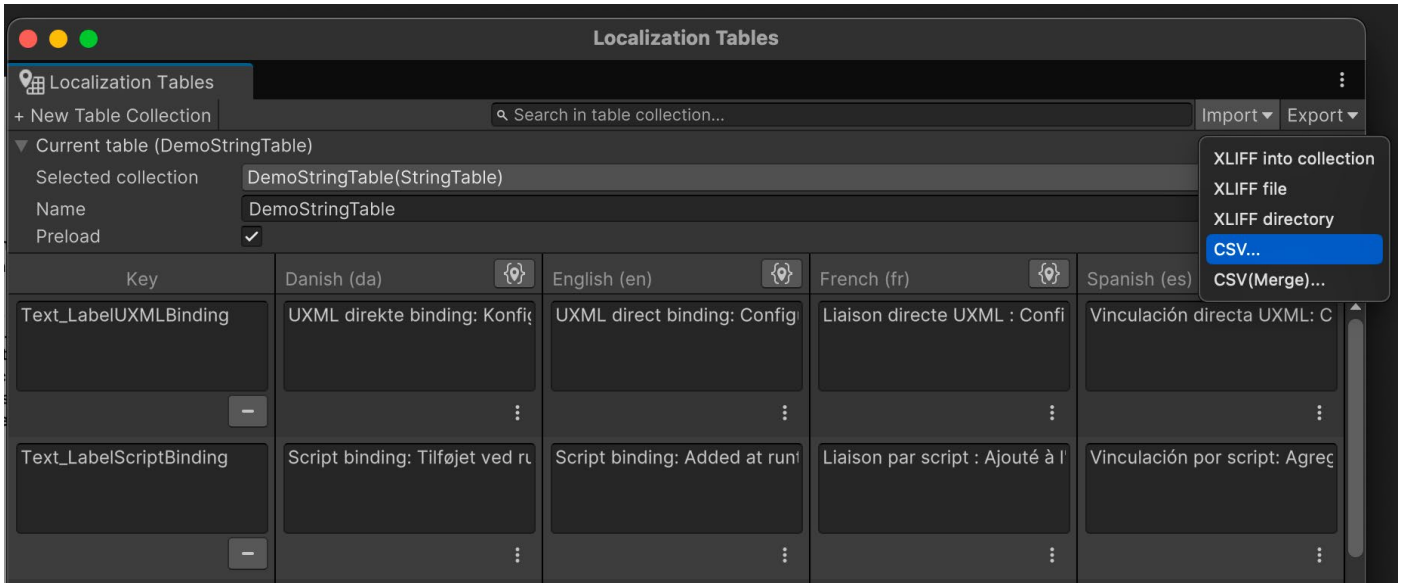
대부분의 현지화 작업에는 UI와 레이블의 모든 텍스트 기반 번역을 처리하는 String Table이 사용됩니다.

Localization Tables 창(**Window > Asset Management > Localization Tables**)을 열고 String Table Collection을 생성하거나 선택합니다. 여기에서 새 항목을 추가하고, 고유 키를 정의하고, 각 로케일별 번역을 입력할 수 있습니다.

문자열 데이터 임포트 및 익스포트

CSV 파일

CSV(쉼표로 구분된 값) 파일에서 데이터를 임포트하여 String Table을 채울 수 있으며, 이렇게 하면 디자이너가 외부에서 텍스트를 설정할 수 있습니다. 일반 텍스트 서식으로 항목을 편집하려면 기존 String Table을 CSV로 익스포트합니다.



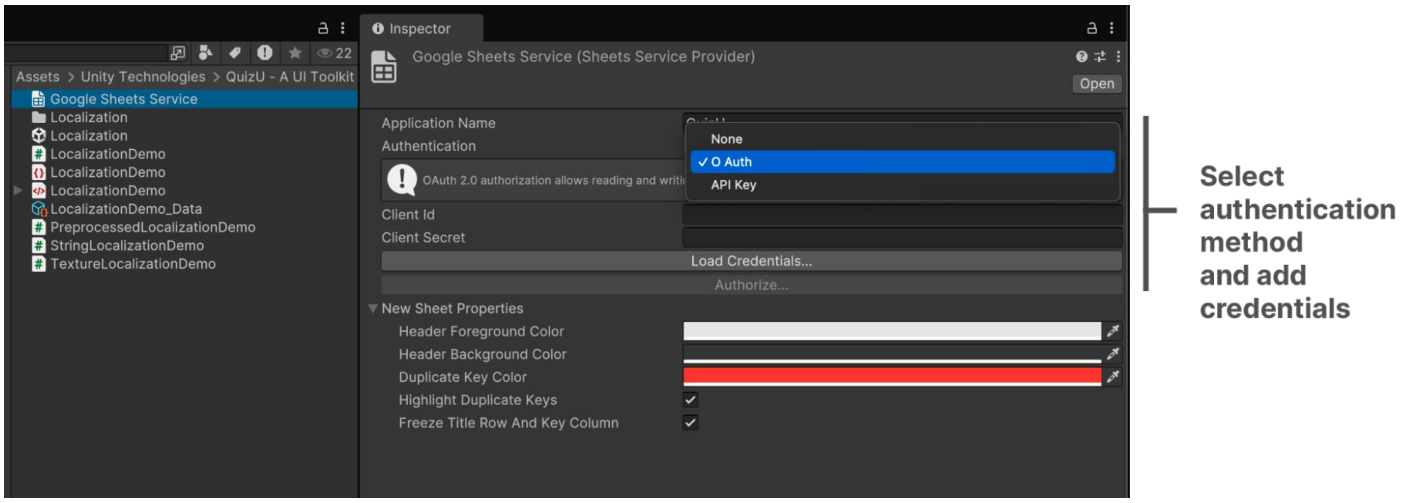
CSV 파일을 임포트 또는 익스포트합니다.

파일을 업데이트한 후에 다시 Unity로 임포트하면 항목이 자동으로 키를 기준으로 업데이트됩니다.

Google Sheets 동기화

프로젝트를 Google Sheets 서비스에 연결하려면 Sheets Service Provider 에셋을 사용해야 합니다. 이 에셋은 인증을 관리하며 사용자가 에디터에서 직접 새 시트를 생성할 수 있도록 지원합니다.

이 에셋을 생성하려면 **Assets > Create > Localization > Google Sheets Service**로 이동합니다.

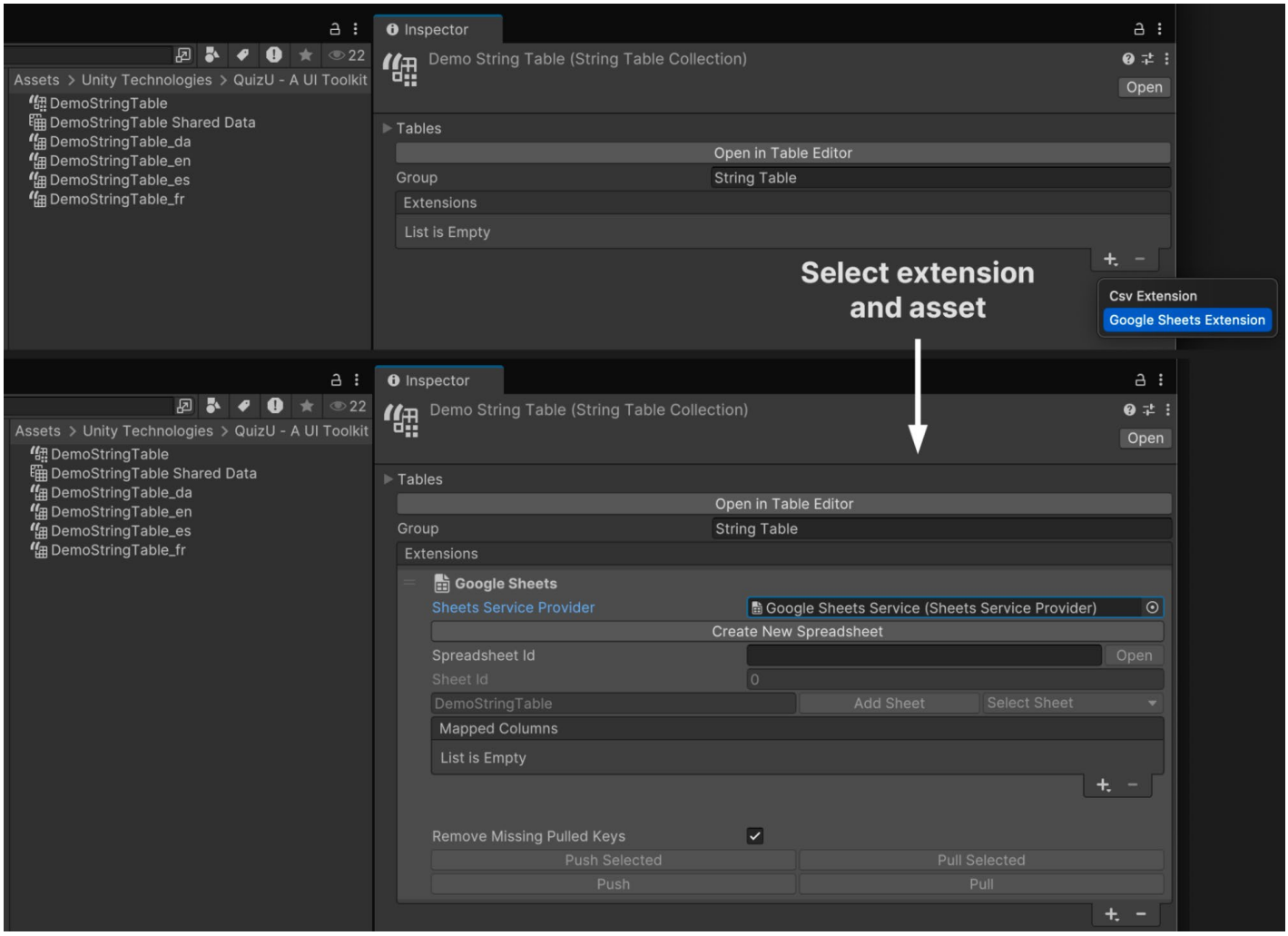


Google Sheets Service를 생성하고 인증합니다.

Google Sheets Service에는 **OAuth**와 **API Key**라는 두 개의 인증 옵션이 있습니다. 비공개 시트에 액세스하여 읽거나 쓰려면 OAuth를 사용하고, 공개 시트에 액세스하여 읽기만 수행해야 하다면 API Key를 사용합니다. 전체 읽기/쓰기 액세스 권한을 얻으려면 Google에 인증을 요청해야 합니다. 자세한 내용은 Google Sheets 기술 자료: 요청 인증하기를 참조하세요.

String Table Collection을 Google Sheet에 연결하려면 컬렉션의 **Extensions** 목록에 **Google Sheet Extension**을 추가합니다. String Table 에셋을 선택한 다음 인스펙터의 **Extensions** 옆에 있는 **더하기(+)** 버튼을 클릭합니다. 여러 확장 프로그램을 String Table Collection 하나에 추가하여, 각 로케일별로 서로 다른 시트를 할당할 수 있습니다.

String Table을 Google Sheet에 동기화하려면 Sheets Service Provider 에셋에 연결합니다. Sheets Service Provider 에셋을 생성하고 구성하는 방법은 Sheets Service Provider를 참조하세요.



StringTable 확장 프로그램에 Google Sheets Service를 추가합니다.

설정을 마치면 디자이너나 비개발자가 이 동기화를 바탕으로 Google Sheets에서 직접 현지화 항목을 편집할 수 있습니다.

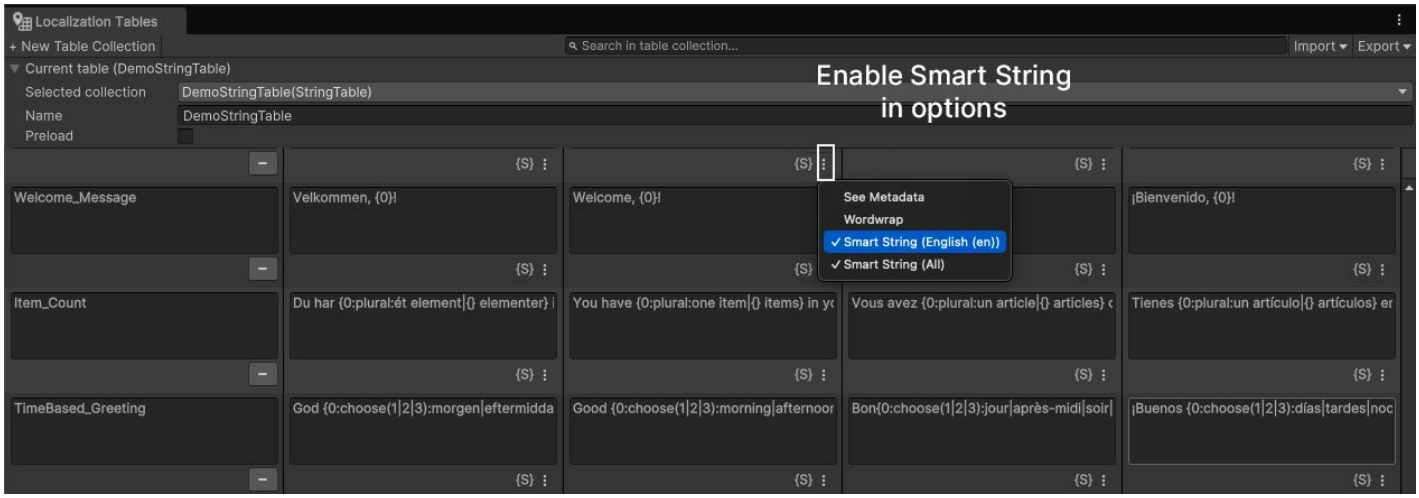
Smart String 사용

Smart String은 동적 문자열을 생성할 때 String.Format 대신 사용할 수 있는 강력한 기능입니다. Smart String을 통해 복수형, 조건부 포맷 지정, 목록 및 기타 언어별 규칙과 같은 기능을 지원하는 데이터 기반 템플릿을 사용할 수 있습니다. 이러한 기능은 현지화 설정을 간소화합니다.

Smart String을 사용하려면 Localization Tables 창에서 문자열을 Smart로 지정합니다.

Localization Tables 창을 엽니다. 메뉴 옵션( )에서 **Smart Format**을 선택합니다. 항목 옆에 **{S}** 아이콘이 표시되는지 확인하세요.

아니면 인스펙터의 **Localized String 에디터** 안에 있는 **Smart** 필드에서 Smart String을 활성화합니다.



StringTable 창 또는 인스펙터에서 Smart String을 활성화합니다.

스크립트에서 Smart String 설정

스크립트에서 SmartString을 관리하는 방법은 다음과 같습니다.

— **플레이스홀더를 사용하여 Localized String 설정하기:** Smart String은 플레이스홀더가 중괄호{}로 묶인 리터럴 텍스트로 구성되며, String.Format과 비슷하지만 훨씬 더 유연합니다. String Table에서 플레이스홀더를 사용하여 항목을 생성합니다(예: “`Welcome, {0}!`”). 여기서 `{0}`은 런타임 데이터를 위한 플레이스홀더입니다.

— **LocalizedString과 Arguments 사용하기:** 스크립트에서 이 항목에 대한 LocalizedString을 생성하고 Arguments 프로퍼티를 사용하여 런타임 데이터를 지정합니다. 예를 들면 SmartStringDemo에서 발췌한 다음 스니핏은 플레이스홀더 하나를 대체하는 방법을 보여 줍니다.

```
//  플레이스홀더를 플레이어 이름으로 대체(예:
//       "반갑네, {0}!" => "반갑네, 플레이어 이름!")

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

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

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

이렇게 하면 LocalizedString이 레이블의 text 프로퍼티에 바인딩되고 런타임에 플레이어의 이름이 삽입됩니다. 원본에 입력한 "반갑네, {0}!"은 화면에 "반갑네, 플레이어 이름!"으로 표시될 수 있습니다.

플레이스홀더 이해

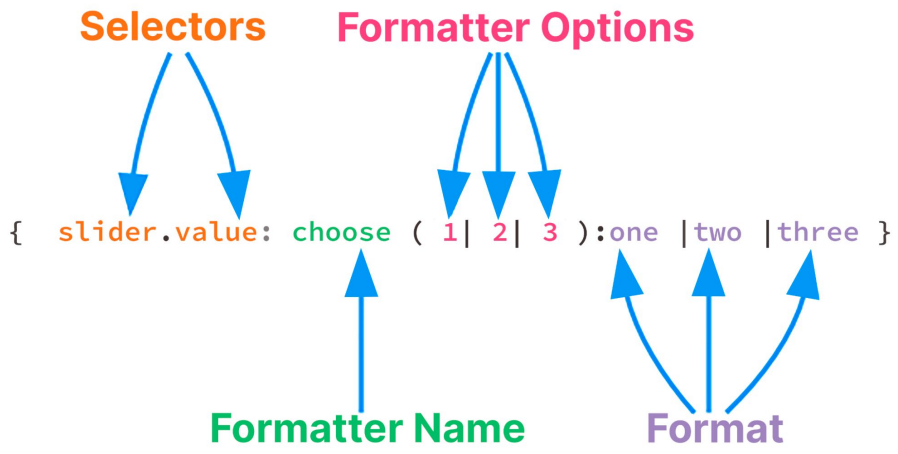
Smart String의 플레이스홀더는 단순히 {0} 치환의 용도로만 사용되지 않습니다. 플레이스홀더는 한층 더 복합적으로 사용할 수 있으며, 고급 시나리오를 처리할 수 있도록 고안되었습니다. 플레이스홀더는 다음과 같은 여러 부분으로 구성될 수 있습니다.

— **선택자**: 어느 데이터를 사용할지 정합니다. 예를 들어 {player.name}은 player 오브젝트의 name 프로퍼티를 선택합니다.

— **포맷터 이름**: 적용할 포맷터를 정의합니다. 예를 들어 복수형 포맷을 지정하려면 plural을 사용합니다.

— **포맷터 옵션**: 포맷터의 동작을 커스터마이즈합니다(예: 단수 형태와 복수 형태 지정).

— **포맷:** 출력이 표시되는 방식을 정합니다. 예를 들어 숫자를 복수형 단어로 전환하거나, 날짜 또는 시간 포맷을 지정하거나, 입력에 따라 문구를 선택합니다.



플레이스홀더는 여러 부분으로 구성될 수 있습니다.

선택자는 유연하게 활용 가능하며, 런타임에 데이터를 동적으로 가져옵니다. 오브젝트의 프로퍼티 또는 필드를 런타임에 쿼리할 수 있습니다. 예를 들어 `{gameObject.name}` 선택자를 사용하면 게임 오브젝트의 name 프로퍼티를 가져올 수 있고, `{slider.value}` 선택자를 사용하면 slider의 value 프로퍼티를 가져올 수 있습니다.

포맷터 는 가져온 데이터를 최종 문자열 포맷으로 전환합니다. 포맷터를 사용하여 날짜, 시간, 목록, 복수형 포맷을 지정하거나 조건부 로직을 적용할 수 있습니다.

포맷터는 가져온 데이터를 최종 출력으로 변환합니다. 각 포맷터는 전용 옵션과 포맷 규칙을 정의합니다. 샘플 프로젝트에는 다음과 같은 두 개의 포맷터가 있습니다.

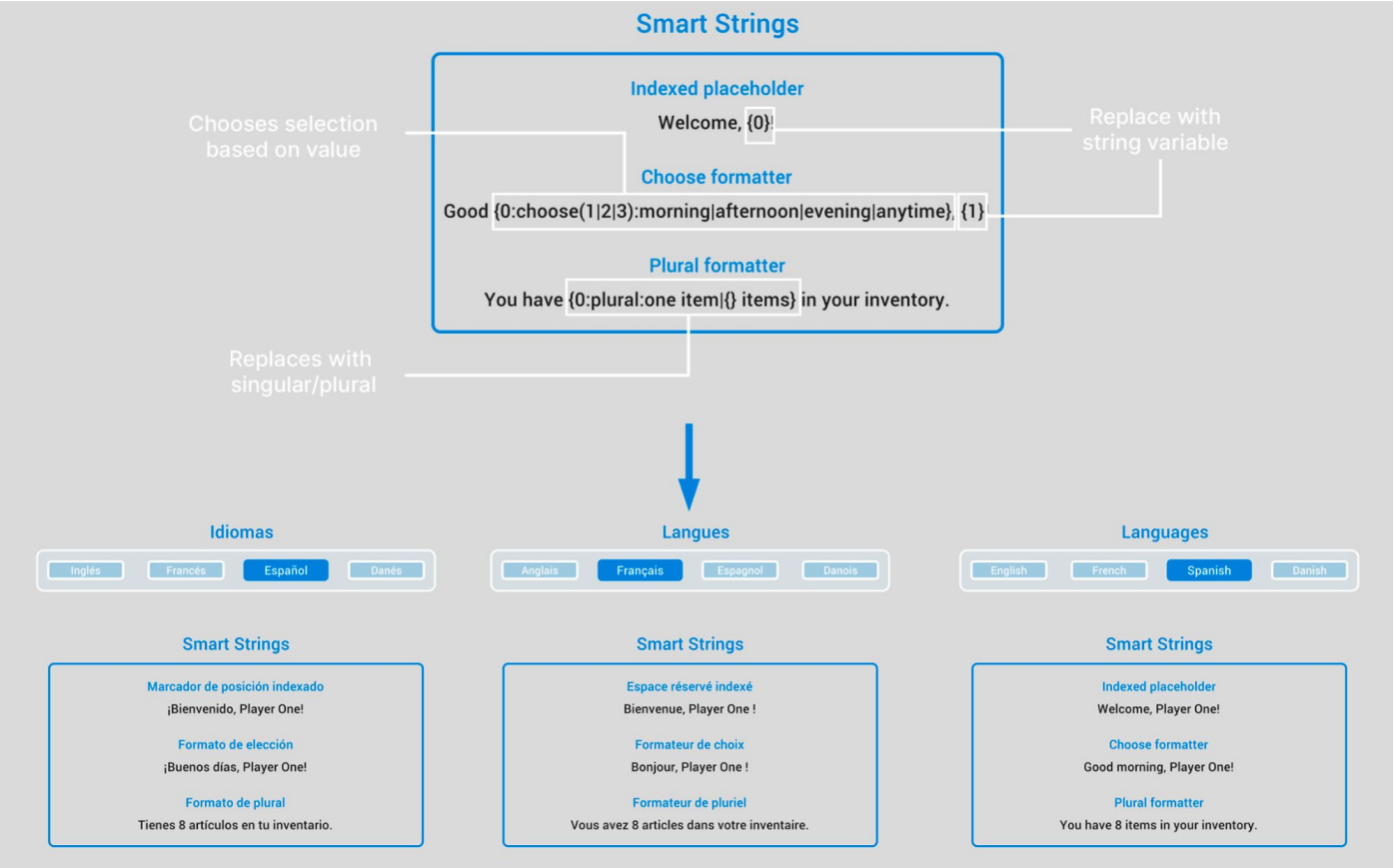
— **Choose 포맷터**:

`{0:choose(1|2|3):morning|afternoon|evening|anytime}`은 입력되는 내용에 따라 'morning', 'afternoon' 또는 'evening'을 선택합니다.

— **Plural 포맷터**:

`{0:plural:one item|{} items}`는 텍스트를 단수형 또는 복수형으로 표시합니다.

인스펙터에서 값을 수정하고 플레이 모드로 전환하여 결과 텍스트를 살펴보세요.



Smart String은 String.Format을 대체하는 기능입니다.

Smart String은 현지화를 향상하는 다양한 빌트인 포맷터를 통해 게임 상태 또는 컨텍스트에 맞게 텍스트를 표시할 수 있도록 지원합니다.

- **Choose 포맷터**: 입력된 수치에 따라 조건부 로직을 적용할 수 있습니다.
- **Plural 포맷터**: 수량을 기준으로 자동으로 복수형 규칙을 적용합니다.
- **Time 포맷터**: 날짜와 시간을 표시합니다.
- **Conditional 포맷터**: if/else와 같은 로직을 적용합니다.
- **List 포맷터**: 배열 또는 목록의 포맷을 지정합니다.
- **Is Match 포맷터**: 정규식 패턴을 기준으로 조건부 텍스트를 표시할 때 사용합니다.

API를 사용하여 커스텀 포맷터를 생성할 수도 있습니다. 포맷터에 대한 자세한 내용은 Smart String 기술 자료를 참조하세요.

문자열 프리 프로세싱

UI 요소를 LocalizedString에 직접 바인딩하기 번거로울 때가 있습니다. 예를 들어 어떤 요소는 현지화된 텍스트를 표시하기 전에 포맷을 추가 지정하거나 수정해야 할 수 있습니다. 이 경우 LocalizedString을 프리 프로세싱한 후에 UI에 표시할 수 있습니다.

GetLocalizedString

`GetLocalizedString` 메서드는 LocalizedString을 런타임에 표준 문자열로 전환합니다. 이 메서드를 사용하면 처리된 문자열을 UI에 표시하기에 앞서 커스텀 포맷(예: 접두사 추가, 문자열 결합 등)을 적용할 수 있습니다. 예시를 들어 보겠습니다.

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

// 현지화된 문자열을 가져와 {0} 플레이스홀더를 플레이어의 레벨로 대체하는 프로퍼티
[CreateProperty] public string LevelMessage =>
m_LevelMessage.GetLocalizedString(m_PlayerLevel);
```

이 예시에서 `LevelMessage` 프로퍼티는 LocalizedString의 `{0}` 플레이스홀더를 플레이어의 현재 레벨로 대체합니다. `[CreateProperty]` 속성을 지정하면 이 프로퍼티를 런타임 데이터 바인딩에 사용할 수 있어, UI 요소에 더 쉽게 직접 바인딩할 수 있습니다.

간단한 사용 사례로, 위에 나온 `LevelMessage`와 같은 프로퍼티가 포맷 지정 로직을 처리하도록 구현하면 추가적인 이벤트 핸들러를 사용할 필요가 없습니다.

StringChanged 이벤트 사용

LocalizedString의 `StringChanged` 이벤트는 이러한 유형의 프리 프로세싱을 구현하는 데 유용합니다. `StringChanged` 이벤트는 LocalizedString이 업데이트될 때마다(즉, 로케일이 변경될 때마다) 트리거되므로 텍스트가 렌더링되기 전에 수정할 수 있습니다.

이 이벤트를 사용하려면 `StringChanged` 이벤트에 핸들러를 연결합니다. 다음은 My_Entry 항목을 사용하여 `My_Table StringTable`에서 새 LocalizedString을 생성하는 코드 스니핏입니다.

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

localizedString.StringChanged += OnLocalizedStringChanged;
```

`OnLocalizedStringChanged` 이벤트 핸들러가 어떻게 표준 문자열로 전환하는 작업을 자동으로 처리하는지 확인하세요. 이제 텍스트가 표시되기에 앞서 수정하는 커스텀 로직을 적용할 수 있습니다.

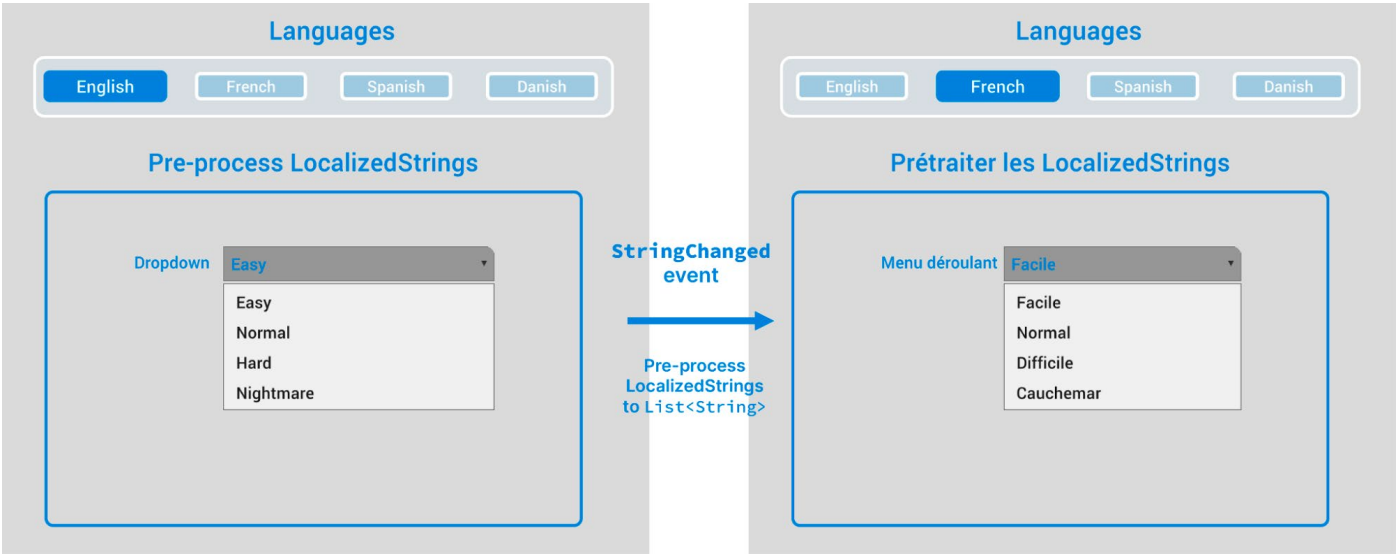
```
void OnLocalizedStringChanged(string value)
{
   // 예시: 특정 조건에 따라 접두사 추가
   string processedString = $"[Prefix] {value}";

   // UI 요소를 처리된 문자열로 업데이트
   m_TextLabel.text = processedString;
}
```

동적 UI 컨트롤

물론 LocalizedString의 프리 프로세싱이 기본적인 텍스트 필드만으로 제한되지는 않습니다. 이는 Smart String만으로 필요한 로직이나 포맷을 처리할 수 없는 경우 복잡한 프로퍼티나 UI 구조를 사용할 때 특히 유용합니다.

예를 들어 DropdownField에는 문자열 목록으로 구성된 choices 프로퍼티가 있습니다. 프리 프로세싱을 통해 활성 로케일을 반영하도록 이 옵션 목록을 동적으로 현지화할 수 있습니다.

여기서 PreprocessDemo 스크립트는 DropdownField choices를 현지화하고, 플레이어가 새 언어를 선택할 때마다 업데이트합니다. 여기서도 목록을 다시 구축하는 로직은 StringChanged 이벤트에 대한 응답으로서 실행됩니다.



StringChanged 이벤트를 사용하여 LocalizedString을 프리 프로세싱합니다.

다음은 PreprocessDemo 스크립트에서 발췌된, 옵션이 다시 구축되는 방식을 확인할 수 있는 스니핏입니다.

```
[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;

    // … 다른 옵션 등록

    //초기 입력
    UpdateDropdownChoices(null);
}
```

StringChanged 이벤트가 트리거되면 드롭다운의 옵션이 다시 구축되고 현재 선택된 옵션은 보존됩니다.

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

    // 현재 선택된 옵션 저장
    int selection = m_DropdownField.index;

    // 이전 옵션 제거
    m_DropdownField.choices.Clear();

    // 현재 현지화된 값 추가

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

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

    // … 다른 옵션 추가

    // 선택된 인덱스와 값 복원
    m_DropdownField.index = selection;

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

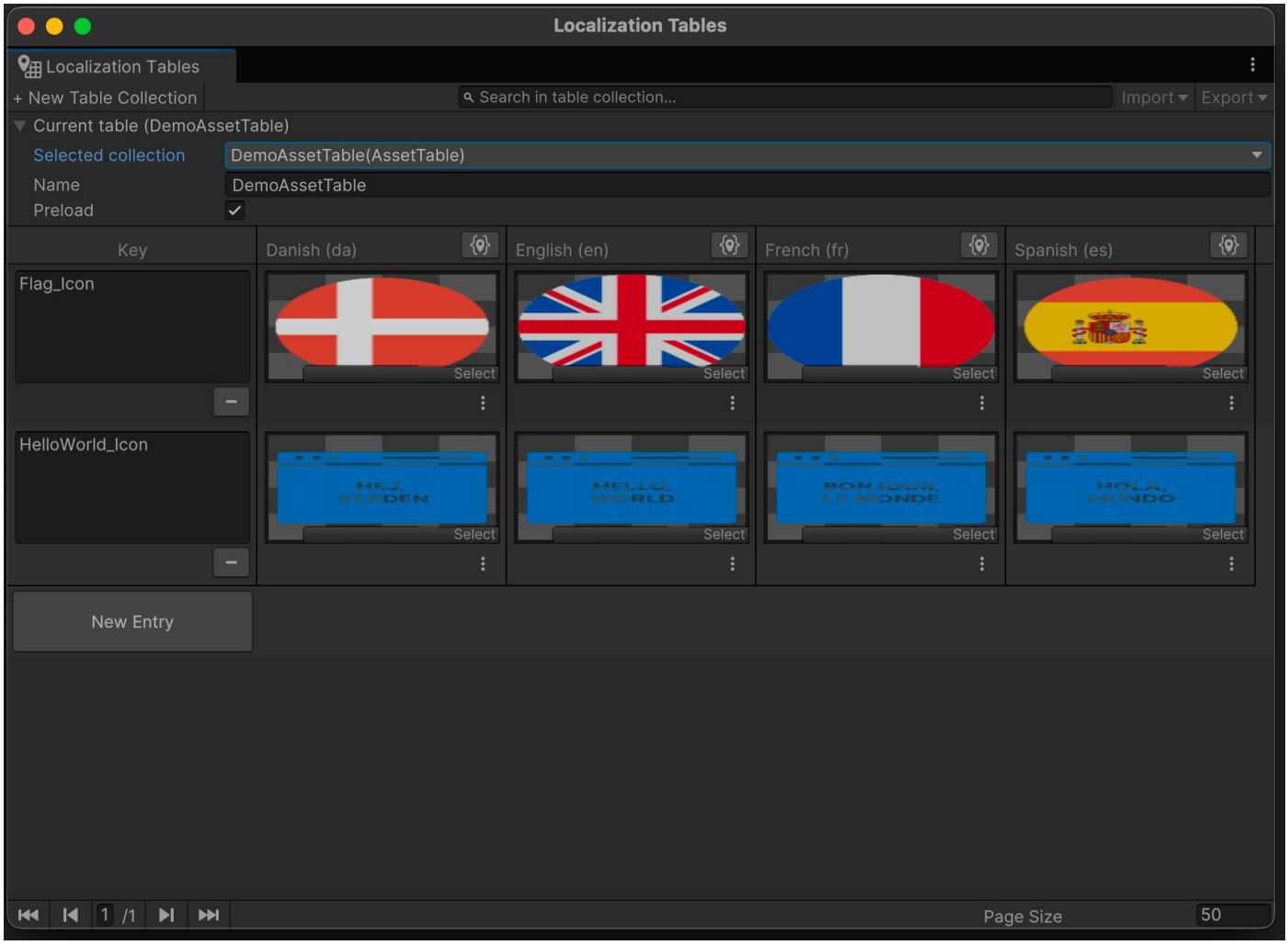
`SetValueWithoutNotify`를 사용하면 ChangeEvent 트리거 없이 드롭다운의 표시가 업데이트됩니다. 이 방식은 재귀적 업데이트를 방지하고 드롭다운 옵션이 변경되었을 때 사용자가 선택한 옵션을 보존합니다.

샘플 프로젝트에서 DropdownField는 LocalizedString 값에 따라 옵션을 동적으로 업데이트합니다. 새 로케일이 선택될 때마다 업데이트된 언어가 드롭다운 옵션으로 전파됩니다.

이 경우 프리 프로세싱은 현지화된 컨텍스트 인식 UI를 생성하는 데 도움이 되는 추가적인 기법이 될 수 있습니다. Smart String은 플레이스홀더, 복수형과 같은 여러 현지화 작업을 처리하지만, 여기에 프리 프로세싱을 더하면 Smart String만으로는 처리할 수 없는 포맷 지정을 적용하고 유연성을 향상할 수 있습니다.

에셋 현지화

현지화에서 가장 큰 비중을 차지하는 것은 문자열이긴 하나, 텍스트 외에도 에셋을 현지화해야 할 수 있습니다. 한 가지 예로, 샘플 프로젝트에는 구성된 로케일에 따라 다르게 적용해야 하는 아이콘이 포함되어 있습니다.



국기와 'Hello, world' 아이콘은 각 로케일을 나타냅니다.

에셋 현지화 설정

에셋 현지화는 문자열 현지화와 거의 비슷합니다. 현지화된 텍스트를 저장하는 데 String Table을 사용하는 것처럼, 현지화된 에셋을 저장하는 데 Asset Table을 사용하면 됩니다. 두 테이블은 항목 추가하기, 스크립트 또는 UXML 파일에서 항목 참조하기 등 비슷한 워크플로를 공유합니다.

현지화된 에셋은 UI 빌더를 통하거나 C# 스크립팅을 이용해 UI 요소에 바인딩할 수 있습니다. 예를 들어 시각적 요소의 style.backgroundImage 프로퍼티를 현지화된 스프라이트 또는 텍스처에 바인딩할 수 있습니다.

샘플 프로젝트를 살펴보면 다음과 같습니다.

— 요소 하나는 데이터 바인딩이 UI 빌더를 통해 UXML로 정의되어 있습니다.

— 또 다른 요소는 바인딩이 C# 스크립트로 설정되어 있습니다.

런타임에 로케일을 선택하면 아이콘과 텍스트 레이블이 업데이트되어 활성 로케일을 빠르게 확인할 수 있습니다.

각 로케일에 따라 LocalizedTexture가 업데이트됩니다.

Asset Table과 String Table 비교

Asset Table을 사용하는 방법은 String Table 사용법과 같습니다. 두 테이블 모두 로케일을 기준으로 항목을 정의하고 런타임에 항목을 가져오는 데 사용됩니다. 단, 다음과 같은 차이점이 있습니다.

— **이벤트 처리**: 현지화된 문자열이 변경되면 StringChanged 이벤트가 트리거된 반면, Asset Table은 AssetChanged를 사용하여 현지화된 에셋의 변경을 알립니다.

— **바인딩 메서드**: SetBinding은 문자열과 에셋 양쪽에 사용되나, 바인딩 프로퍼티는 에셋 유형에 따라 달라집니다(예: 문자열의 경우 text, 텍스처의 경우 style.backgroundImage).

다음 스니핏은 예시 데모에서 어떻게 Asset Table로부터 이름을 기준으로 LocalizedTexture를 가져와 `style.backgroundImage` 프로퍼티에 바인딩하는지를 보여 줍니다.

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

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

UI 툴킷의 자주 사용되는 현지화된 에셋

현지화된 에셋은 다양한 형태로 제공됩니다. UI 툴킷을 사용할 때 볼 수 있는 몇 가지 에셋은 다음과 같습니다.

— **현지화된 텍스처:** 아이콘, 배경 및 기타 장식용 시각적 요소에 적합하며, `style.backgroundImage`와 같이 시각적 요소 프로퍼티에 직접 바인딩될 수 있습니다.

— **현지화된 스프라이트:** 비교적 자주 사용되지 않으나 커스텀 컴포넌트 또는 스프라이트 기반 시각적 요소에 유용합니다.

— **현지화된 폰트:** 폰트를 언어별로 요구되는 특정 스크립트나 서체 스타일로 전환할 수 있습니다.

— **현지화된 오브젝트:** 프리팹이나 데이터 기반 에셋과 같이 로케일에 따라 달라져야 하는 복잡한 리소스를 참조하는 데 유용합니다.

Asset Table로 이러한 에셋을 사용하면 UI의 텍스트뿐 아니라 시각적 요소도 활성 로케일에 맞게 동적으로 조정되도록 할 수 있습니다.

드래곤 크래셔 샘플의 현지화

UI 툴킷 샘플 – 드래곤 크래셔 데모에서는 여러 현지화 기법이 사용되고 있습니다. Settings 뷰에서 드롭다운 메뉴를 사용하여 지원되는 언어 중 하나를 선택할 수 있습니다. 새 언어가 선택되면 LocalizationSettings 시스템이 변경을 감지하고 UI를 실시간으로 업데이트합니다.

Language 드롭다운 메뉴에서 로케일을 선택합니다.

이 프로젝트에서 살펴볼 만한 내용은 다음과 같습니다.

— **SettingsScreen 로케일 선택:** Settings 화면에서는 사용자가 드롭다운 메뉴를 통해 로케일을 선택할 수 있습니다. 이 UI는 `LocalizationSettings`의 변경을 수신 대기하여 새로 선택된 로케일을 감지하고 드롭다운 메뉴가 변경되면 실시간으로 업데이트됩니다.

— **데이터 바인딩 기법:** 이 UI에는 다양한 현지화 기법이 적용되어 있습니다. 정적 프로퍼티는 UI 빌더에서 직접 바인딩되어 UXML에 저장됩니다.

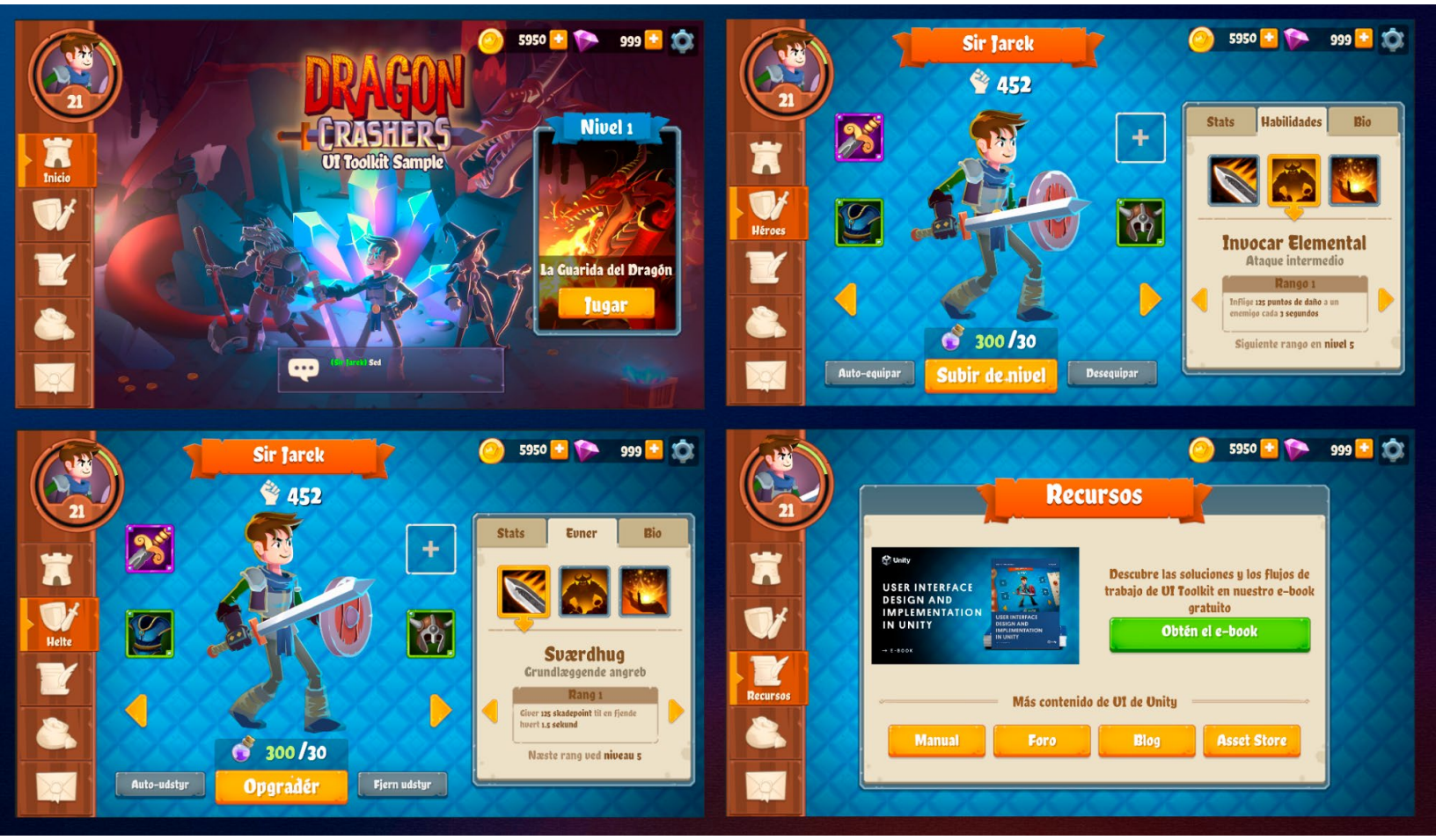
  반면에 동적으로 채워지는 필드는 런타임 스크립트를 통해 데이터가 바인딩됩니다. `SetBinding` 메서드는 UI가 선택된 로케일을 반영하도록 text 프로퍼티를 `LocalizedString` 오브젝트에 연결합니다.

— **ScriptableObject의 포맷이 사전 지정된 LocalizedString:** 포맷이 사전 지정된 `LocalizedString` 프로퍼티가 있는 ScriptableObject 에셋도 있습니다. 예를 들어 Settings 화면에서 Theme 및 Language 드롭다운 필드는 현지화된 값을 가지고 동적으로 목록을 다시 구축하며, 그 과정에서 선택 가능한 옵션을 번역합니다.

그 밖에 RadioButtonGroup이나 커스텀 SlideToggle과 같은 요소도 StringChanged 이벤트를 처리하여 LocalizedString을 프리 프로세싱합니다.

UXML을 통한 데이터 바인딩과 C# 스크립팅을 통한 데이터 바인딩 중 어느 현지화 기법을 사용하든, UI는 로케일 변경에 실시간으로 반응합니다.

이 샘플 프로젝트에 사용된 기법을 참고하여 자체 Unity 프로젝트에서 현지화된 인터페이스를 제작해 보세요. 데이터 바인딩과 UI 툴킷을 함께 사용하면 유연한 다국어 UI를 제작하여 전 세계의 플레이어들에게 제공할 수 있습니다.



UI 툴킷 샘플 – 드래곤 크래셔에서 더 많은 현지화 예시를 살펴보세요.

커스텀 컨트롤

UI 툴킷은 인터페이스를 제작하는 데 필요한 표준 요소를 제공하며, 그 외에도 애플리케이션의 필요에 따라 맞춤화된 커스텀 컨트롤을 생성할 수 있습니다.

예를 들어 체력이 감소하면서 컬러가 초록색에서 노란색을 거쳐 빨간색으로 바뀌는 등, 체력 값에 따라 컬러가 바뀌는 커스텀 체력 바를 만들 수 있습니다. 이렇게 만든 커스텀 체력 바를 추가 설정 없이 모든 캐릭터에 대해 사용할 수도 있고, 마나나 파워와 같은 다른 스탯을 나타내는 데 사용할 수도 있습니다. 이처럼 캡슐화된 컨트롤은 UI 툴킷 표준 라이브러리에서 제공하는 슬라이더에 비해 시각적으로 더 뚜렷한 효과를 낼 수 있습니다.

커스텀 컨트롤을 생성하면 기능을 스탠드얼론 요소로 캡슐화하여 인터페이스의 여러 부분에서 재사용할 수 있습니다. 잘 디자인된 컨트롤은 추상화되어 있고, 독립적이며, 코드 재사용을 지원하기 때문에 프로젝트 유지 관리가 더 쉬워집니다. 커스텀 컨트롤을 구현할 때는 특정 컴포넌트에 연동되어 스탠드얼론 기능을 갖지 않는 요소(예: 게임 메뉴)와 함께 사용하지 않도록 하세요.

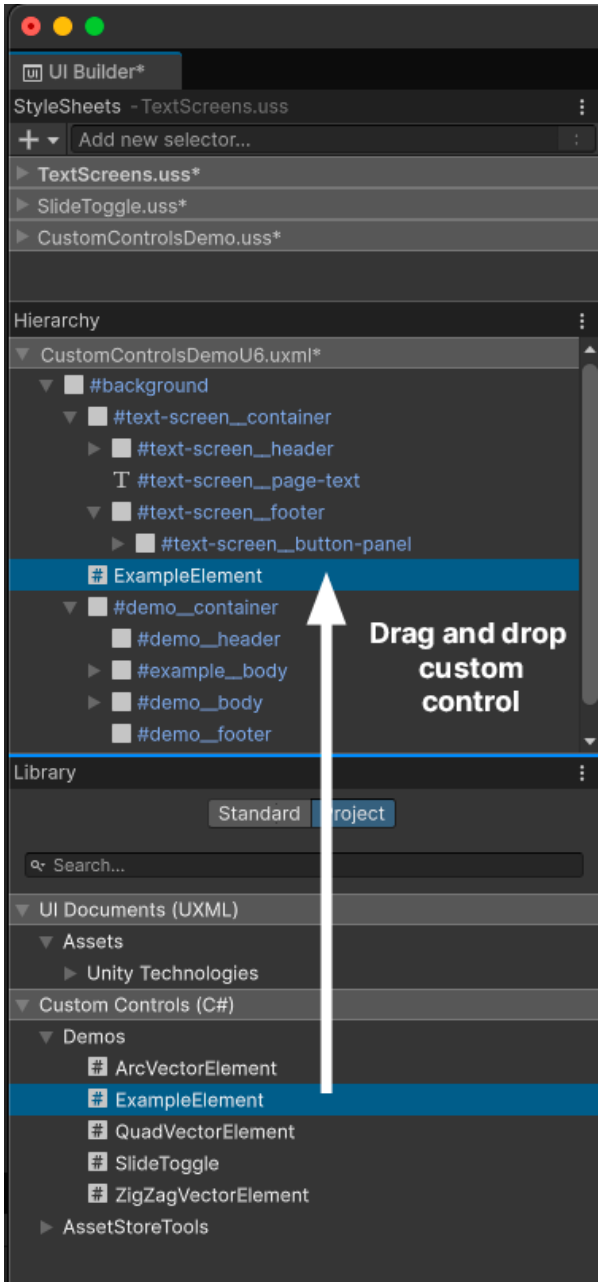
UxmlElement 속성

커스텀 컨트롤을 생성하려면 먼저 VisualElement 클래스 또는 생성하려는 컨트롤과 비슷한 하위 클래스를 상속하는 새 C# 스크립트를 정의합니다. 버튼처럼 작동하는 컨트롤이 필요하다면 Button 클래스를 상속하면 됩니다.

커스텀 컨트롤을 UXML과 UI 빌더에서 사용할 수 있도록 클래스에 `UxmlElement` 속성을 추가합니다. 커스텀 요소는 다음과 같이 `public partial` 클래스로 정의해야 합니다.

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


}
```

그러면 커스텀 컨트롤이 UI 빌더에서 Library 섹션의 **Custom Controls (C#)** 카테고리에 나타납니다. 여기서 UI 빌더의 계층 창으로 커스텀 컨트롤을 드래그하세요.



커스텀 컨트롤은 UI 빌더의 Library에 나타납니다.

시각적 요소는 게임 오브젝트가 아니기 때문에 Awake, OnEnable, OnDisable, OnDestroy와 같은 일반적인 라이프사이클 이벤트가 없습니다. 따라서 커스텀 컨트롤은 생성자를 사용하여 초기화합니다.

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

    // 생성자
    public ExampleElement()
    {
        // 초기화
    }
}
```

커스텀 컨트롤이 UI에 추가되는 시점까지 초기화를 지연할 수도 있습니다. 이렇게 하려면 **AttachToPanelEvent**에 대해 콜백을 등록합니다.

커스텀 컨트롤이 UI에서 제거된 경우를 감지하려면 **DetachFromPanelEvent** 콜백을 사용합니다.

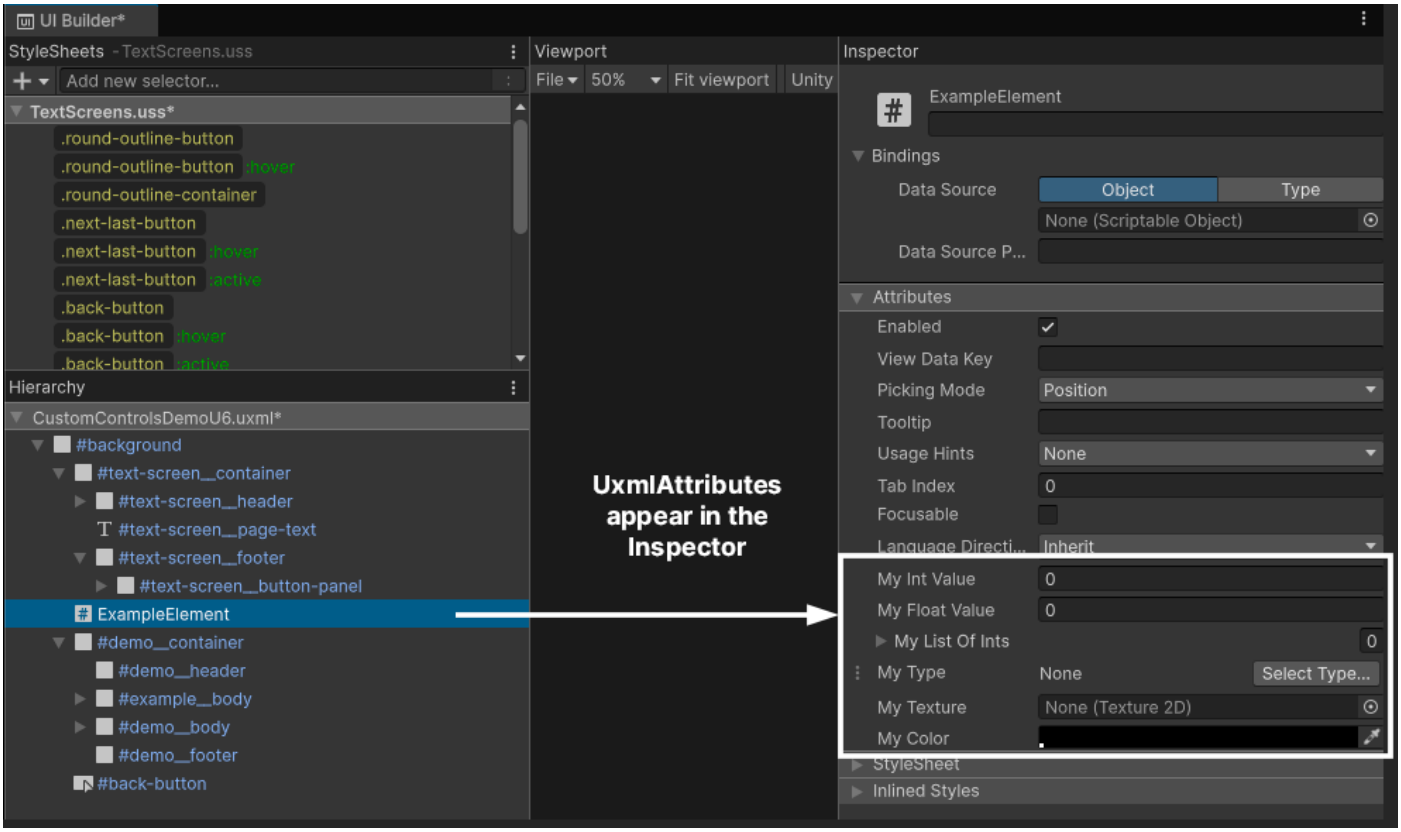
UxmlAttribute 속성

프로퍼티에 **UxmlAttribute** 속성을 추가하면 해당 프로퍼티가 UI 빌더의 인스펙터 창에 나타납니다. 그러면 초기 값을 대화식으로 설정할 수 있습니다. UxmlAttribute를 사용하면 인스펙터에서 프로퍼티를 변경할 때 코드를 수정할 필요가 없기 때문에 디자이너와 함께 작업할 때 유용합니다.

노출하려는 각 프로퍼티에 UxmlAttribute 속성을 적용합니다. **name** 인수를 사용하여 속성 이름을 커스터마이즈할 수도 있습니다.

계층 구조에서 컨트롤을 선택하면 커스텀 속성이 인스펙터 창에 표시되어 속성을 직접 구성할 수 있습니다.

데코레이터 속성은 마치 MonoBehaviour를 사용하는 것처럼 커스텀 속성 필드를 수정할 수 있습니다. 유용한 데코레이터 속성의 예를 들면 TextArea, Tooltip, Range, Header, Min, Multiline, Space, Delayed 등이 있습니다. 예를 들어 Range 속성을 사용하면 일정 범위 내의 값을 선택할 수 있는 슬라이더가 추가됩니다.



커스텀 속성은 UI 빌더의 인스펙터에 나타납니다.

다음은 커스텀 컨트롤에 UxmlElement 속성을 추가하는 기본적인 예시입니다. 여기서는 UxmlAttribute 속성을 사용하여 두 개의 프로퍼티가 노출됩니다.

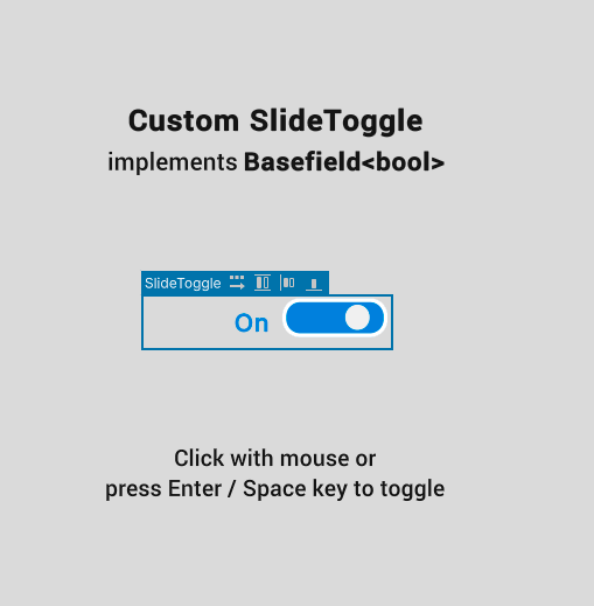
```
[UxmlElement]
public partial class ExampleElement: VisualElement
{
    [UxmlAttribute(name:"my-text")]
    public string myStringValue { get; set; }

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

}
```

이 예시에서 `MyStringValue`는 name 파라미터를 사용하여 인스펙터에 **'My Text'**로 표시됩니다. `MyStringValue`와 `MyIntValue`는 모두 계층 구조에서 ExampleElement의 인스턴스가 선택될 때마다 인스펙터에서 편집할 수 있습니다.

Unity 6 전에는 커스텀 컨트롤을 생성하려면 커스텀 요소의 속성 등록과 오브젝트 인스턴스화를 처리하는 **UxmlTraits** 클래스와 **UxmlFactory** 클래스를 구현해야 했습니다.

Unity 6에서는 UxmlElement 속성과 UxmlAttribute 속성이 도입되어 커스텀 요소 생성이 간소화되었습니다. 이 두 속성은 커스텀 컨트롤과 프로퍼티를 UXML 및 UI 빌더에 직접 표시합니다. 이 새로운 워크플로는 필요한 상용구 코드의 양을 줄이고 UI 요소를 커스터마이즈하는 데 걸리는 시간을 단축합니다.

예시: 커스텀 슬라이드 토글 컨트롤



커스텀 슬라이드 토글 컨트롤은 부울 값을 나타냅니다.

간단한 커스텀 컨트롤의 한 가지 예로 슬라이드 토글을 들 수 있는데, 이는 스위치와 비슷한 요소로 부울 값을 나타냅니다.

슬라이드 토글은 표준 토글에 비해 한층 시선을 사로잡는 경험을 선사할 수 있습니다. 애니메이션 스위치, 컬러 변경, 동적 텍스트와 같은 시각적 피드백을 더하면 더 직관적인 UI를 만들 수 있습니다.

커스텀 컨트롤 정의

QuizU 프로젝트의 CustomControlsDemo 씬에는 이 커스텀 컨트롤의 간단한 구현이 있습니다. SlideToggle.cs 스크립트를 열고 커스텀 컨트롤이 어떻게 작동하는지 살펴보세요(아래 스니핏 참조).

슬라이드 토글 커스텀 컨트롤은 이 경우 가장 적합한 기본 클래스인 BaseField<bool>을 상속합니다. UxmlElement 속성은 UXML과 UI 빌더에 컨트롤을 노출하여 재사용 가능하게 만듭니다.

```
[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;
```

시각적 구조는 배경(m_Input)과 노브(m_Knob), 그리고 외형을 정의하는 USS 클래스로 구성되어 있습니다.

```
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";
```

클릭, 키 누르기 및 내비게이션 이벤트에 반응하도록 이벤트 처리가 구현되어 있어 여러 방법으로 상태를 변경할 수 있습니다.

```
    // …
    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();
    }
}
```

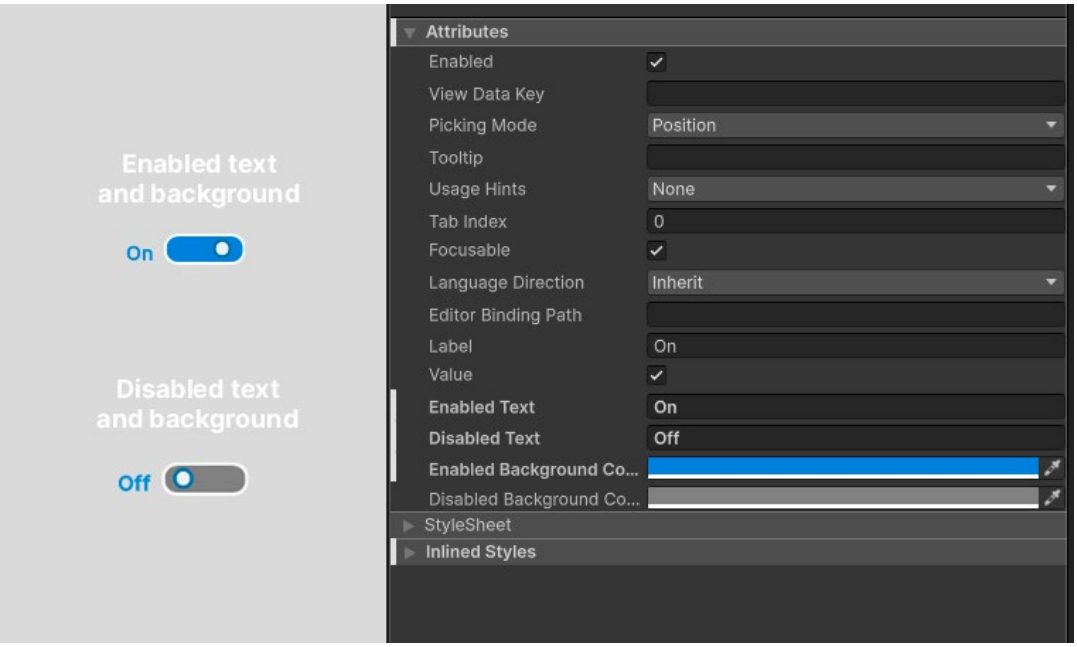
레이블과 배경 컬러는 사용자가 스위치를 토글하면 자동으로 업데이트되어 시각적 피드백을 제공합니다.

아래 코드에서 ChangeEvent를 트리거하지 않고 토글의 시각적 상태를 업데이트하기 위해 SetValueWithoutNotify가 사용된 것을 볼 수 있습니다. 이 메서드는 값이 변경되면 내부적으로 호출되므로 무한 업데이트 루프를 유발하지 않고 UI가 올바르게 업데이트됩니다.

```
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;
}
```

CustomControlsDemo 씬에서 구현된 샘플을 살펴보세요. 요소를 마우스로 클릭하거나 Enter 키 또는 스페이스 키를 눌러 활성 상태를 토글합니다. 이 샘플에서 레이블과 배경 컬러는 사용자가 슬라이드 컨트롤을 토글하면 동적으로 업데이트되며, 간단한 애니메이션으로 시각적 피드백을 제공합니다.

인스펙터를 사용하여 활성 상태와 비활성 상태에 대응하는 문자열 레이블과 배경 컬러를 설정합니다.



슬라이드 토글의 텍스트와 컬러를 커스터마이즈합니다.

슬라이드 토글 사용

컴파일이 완료되면 이제 슬라이드 토글을 UI의 어느 위치에나 통합할 수 있습니다. 커스텀 `SlideToggle` 클래스를 다른 시각적 요소처럼 사용하세요. 다음은 `SlideToggle` 클래스를 사용하여 사운드를 음소거 또는 음소거 해제하는 구현 예시입니다.

```
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);
        }
    }
}
```

여기서 SlideToggle은 기존 UXML 문서의 일부분입니다. MonoBehaviour가 시각적 트리에서 이름으로 SlideToggle을 찾은 다음 `RegisterValueChangedCallback` 메서드를 사용하여 토글 상태를 오디오 설정에 연결합니다.

SlideToggle은 스탠드얼론 커스텀 요소이므로 UI에서 모든 유형의 토글 스위치에 사용할 수 있습니다. 예를 들어 드래곤 크래셔 UI 툴킷 샘플에서는 이와 유사한 SlideToggle이 fps 카운터를 활성화하거나 비활성화합니다.

UI 툴킷 샘플 – 드래곤 크래셔의 스타일이 적용된 토글

애플리케이션에서 필요한 대로 SlideToggle을 커스터마이즈하세요. 비주얼, 사운드, 게임플레이 옵션과 같은 설정에 사용하기에 좋습니다. 한 번 제작해 두고 커스텀 스위치로 사용자 경험을 향상할 수 있는 모든 위치에서 재사용하세요.

전체 구현 코드는 QuizU 프로젝트의 **SlideToggle.cs** 스크립트를 참조하세요.

더 많은 커스텀 컨트롤 생성

표준 UI 툴킷 라이브러리에 원하는 컨트롤이 없다면 직접 생성하면 됩니다. 다음은 게임에서 커스텀 컨트롤을 사용하는 방법에 대한 몇 가지 예시입니다.

— **체력 바/진행 표시줄:** 체력, 마나, 파워 등과 같은 게임 속성은 게임플레이에 따라 크게 달라질 수 있으므로 커스텀 컨트롤을 사용하기에 적합합니다. max value, current value, status color와 같은 UxmlAttribute를 노출하여 컬러 그레디언트 옵션을 추가하세요.

— **별 등급 부여:** 이 컨트롤은 세분화된 진행 표시줄의 기능을 수행하며, 정수 값(예: 레벨 완료 시 부여되는 별)을 나타냅니다. 여러 자식 요소를 가지며 Filled 상태와 Unfilled 상태 간에 전환할 수 있는 시각적 요소를 생성한 다음, 인스펙터에서 최대 값을 갖는 int를 노출하여 사용자가 UxmlAttribute로 스프라이트 이미지를 커스터마이즈할 수 있도록 하세요.

— **탭 뷰 컨트롤:** 탭 인터페이스는 하나의 창 안에서 여러 뷰 또는 섹션 간에 전환하는 용도로 널리 사용되는 UI입니다. 여러 탭으로 구성된 행과 하나의 콘텐츠 영역을 갖는 커스텀 요소를 생성하여 탭 인터페이스를 구현하세요. 각 탭은 동적으로 탭을 추가하거나 제거하는 옵션이 있는 버튼과 같은 시각적 요소로 구현할 수 있습니다.

대부분의 경우 USS 전환을 트리거하여 애니메이션으로 시각적 효과를 추가할 수 있다는 점을 기억하세요. 커스텀 컨트롤을 적용하여, 사용자가 고유의 게임 UI를 통해 핀치, 클릭, 스크롤, 토글 동작을 사용하도록 구현할 수 있습니다.

멋진 결과물을 기대하겠습니다.

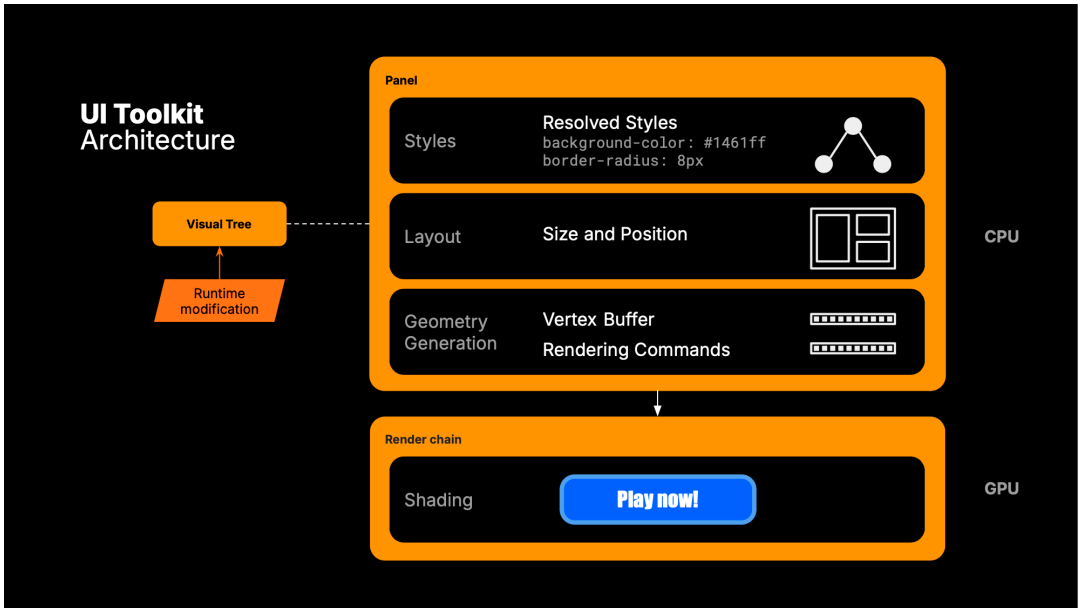
성능 최적화

정교한 게임 UI를 제작하려면 여러 화면 요소로 구성된 대규모 계층 구조를 관리해야 하는 경우가 많습니다. 수백 개의 요소를 관리하다 보면 기술적인 어려움이 발생할 수 있습니다. 조금만 비효율적이어도 런타임에 끊김 등이 발생하는 문제로 이어지고, 플레이어 경험에 부정적인 영향을 줄 수 있습니다.

다행스러운 점은 이러한 문제를 대부분 최적화로 해결할 수 있다는 사실입니다. Unity 6의 향상된 UI 툴킷은 별도의 설정 없이도 뛰어난 성능을 구현하지만, 진정한 의미의 효율적인 사용자 인터페이스는 개발자의 노력 없이는 만들어질 수 없습니다.

많은 경우 불필요한 오버헤드를 없애고 드로우 콜을 줄이는 것으로 문제를 해결할 수 있습니다. Unity 6의 UI 툴킷을 효율적으로 사용할 수 있도록 최적화하는 방법을 살펴보겠습니다.

업데이트 메커니즘



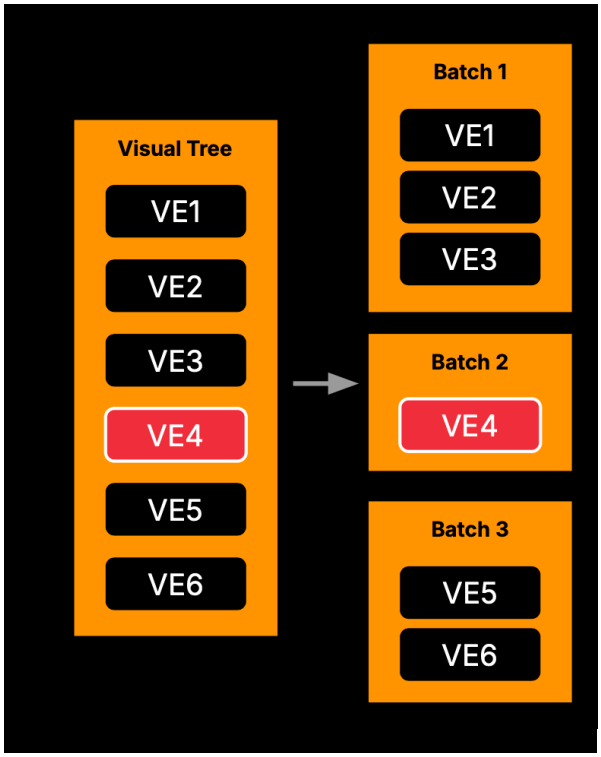
이 시각적 트리에는 여러 업데이트 메커니즘이 있습니다.

이 시각적 트리에는 런타임의 스타일, 레이아웃 또는 콘텐츠 변경에 대응하는 여러 업데이트 메커니즘이 있습니다. 이 모든 업데이트 메커니즘이 성능에 영향을 줄 수 있습니다. 다음 표에는 각 업데이트 메커니즘이 트리거되는 경우와 성능에 미치는 영향이 정리되어 있습니다.

업데이트 메커니즘	설명	트리거되는 경우	성능에 미치는 영향
스타일 지정	USS 선택자와 스타일을 적용하여 요소의 최종 외형을 결정합니다.	클래스 또는 스타일이 변경되면 트리거됩니다(예: 스타일 클래스 추가, 컬러 수정).	계층 구조의 규모가 크거나 중첩된 단계가 많으면 비용이 높아집니다. 잦은 변경을 최소화하세요.
레이아웃 재계산	UI 계층 구조 내에 올바르게 들어맞도록 요소의 크기와 위치를 조정합니다.	요소의 크기, 위치 또는 정렬이 변경되면 트리거됩니다(예: 패널 크기 조정, 요소 이동).	레이아웃을 자주 업데이트하면 리소스를 크게 소모할 수 있습니다. 위치를 직접 변경하는 대신 애니메이션 변환을 사용하세요.
버텍스 버퍼 업데이트	UI 요소를 렌더링하는 데 사용되는 지오메트리 셰이프(직사각형, 둥근 모서리 등)를 업데이트합니다.	요소의 지오메트리가 변경되면 트리거됩니다(예: 둥근 모서리 추가, 테두리 수정 등).	버텍스 버퍼를 업데이트하는 데는 많은 리소스가 소모됩니다. 잦은 지오메트리 변경을 지양하세요.
렌더링 상태 변경	요소를 드로우하는 데 필요한 텍스처와 블렌딩 모드와 같은 렌더링 상태를 변경합니다.	마스킹, 고유 텍스처 등 배칭에 지장을 주는 기능으로 인해 트리거됩니다.	과도한 상태 변경은 CPU 오버헤드를 늘립니다. 고유 텍스처 또는 마스크를 배칭하고 제한하는 방법으로 최적화하세요.

이러한 연산의 비용은 UI 요소를 수정하는 빈도와 범위에 따라 달라집니다.

요소 배칭



배치가 분할되면 성능이 저하됩니다.

사용자 인터페이스를 렌더링할 때는 모든 시각적 요소의 명령이 GPU로 전송되어야 합니다. UI 툴킷은 배칭을 통해 이와 같은 드로우 콜을 최적화합니다. 배칭은 GPU 요구 사항이 동일한 시각적 요소가 함께 처리될 수 있도록 그룹화합니다. 배칭은 게임 오브젝트의 드로우 콜 배칭처럼 GPU의 통신 오버헤드를 크게 줄입니다.

요소를 효율적으로 배칭하려면 요소가 동일한 GPU 상태(동일한 셰이더, 텍스처, 메시 데이터 및 기타 GPU 전용 파라미터)를 공유해야 합니다. 예를 들어 같은 폰트와 스타일을 사용하는 텍스트 요소로 구성된 시퀀스는 함께 배칭될 수 있습니다. 단, 텍스트 사이에 이미지를 삽입하려면 이와 다른 GPU 설정이 요구되므로 새로운 배치가 필요합니다.

이와 같은 배치 분할이 발생할 때마다 약간의 비효율이 발생합니다. 높은 성능을 유지하려면 이러한 분할이 최소화되도록 UI를 구성해야 합니다.

모든 배치는 GPU에 대해 하나 이상의 드로우 콜을 발생시키므로, 배치의 개수가 적을수록 일반적으로 오버헤드가 줄어들고 성능이 향상됩니다.

이어지는 섹션에서는 UI의 배치 개수를 최적화하고 일관된 성능을 유지하는 기법을 살펴보겠습니다.

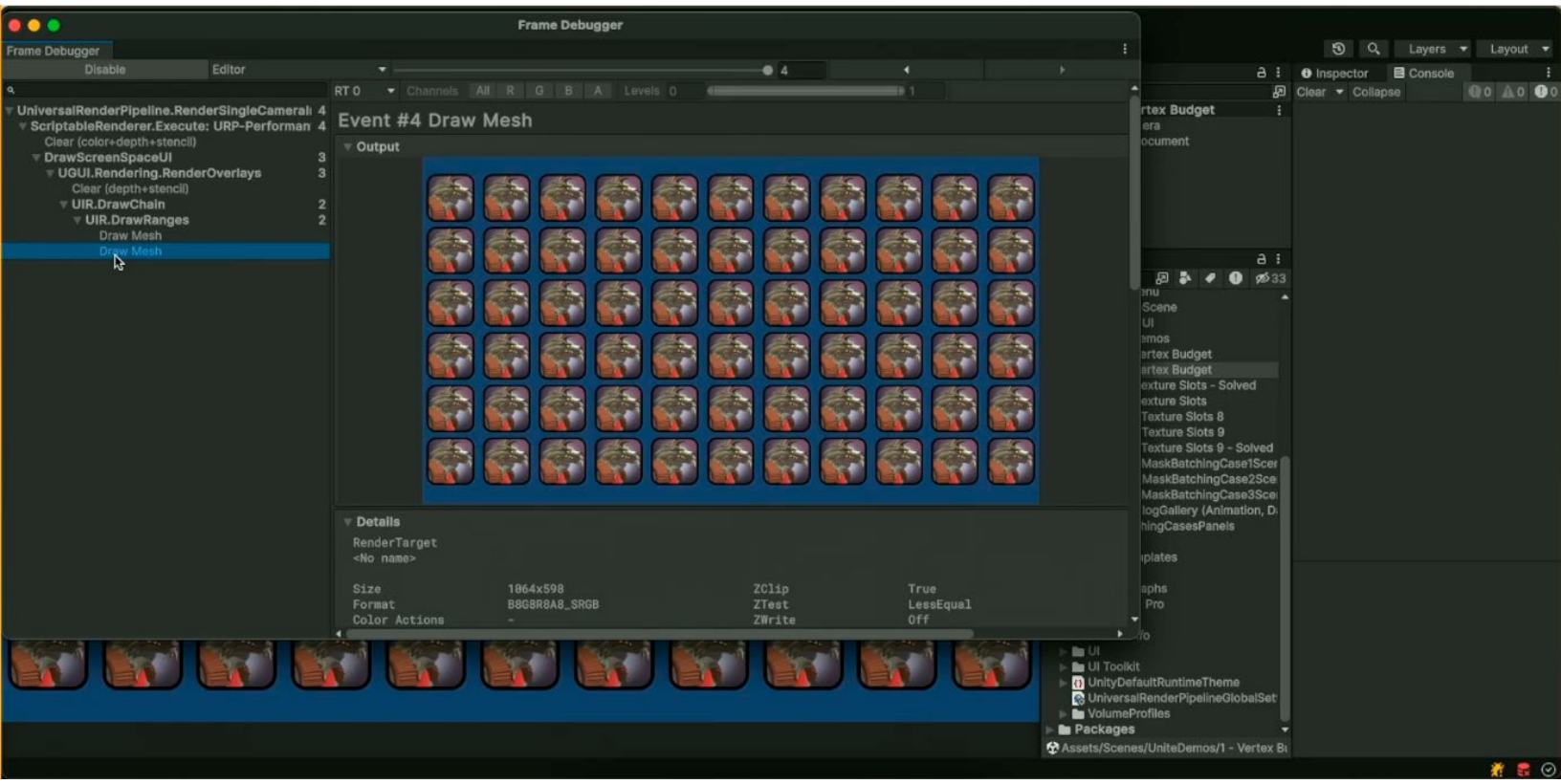
버텍스 버퍼

UI 툴킷에서 버텍스 버퍼에는 UI를 렌더링하는 데 필요한 지오메트리(버텍스)가 저장됩니다. UIDocument는 런타임에 패널을 생성할 때 시각적 요소를 처리할 하나의 버텍스 버퍼를 사전 할당합니다. 버퍼는 시각적 요소를 위한 일종의 '힙 할당자'로, UI에 요소가 추가됨에 따라 동적으로 메모리를 할당합니다.

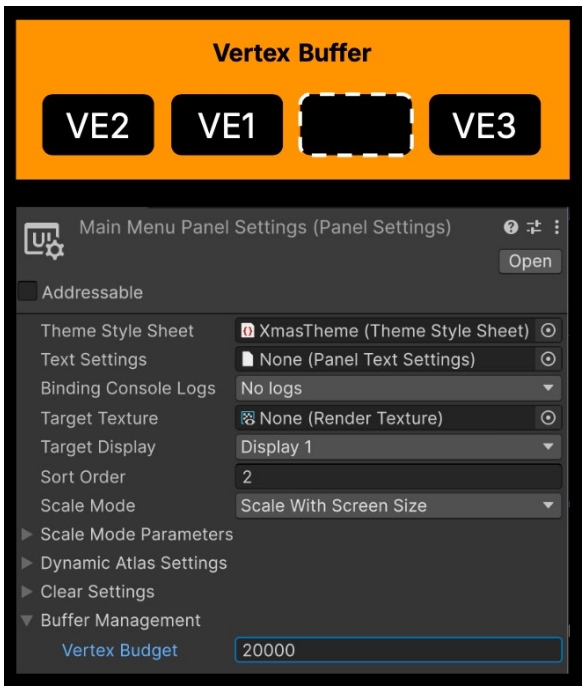
UI가 버텍스 버퍼의 용량을 초과하면 버퍼가 추가로 생성됩니다. 이로 인해 배칭이 분할되어 드로우 콜의 횟수가 늘어나고 성능이 저하될 수 있습니다.

이 문제를 해결하려면 Panel Settings에서 **Vertex Budget**을 조정하여 버텍스 버퍼의 초기 크기를 설정하면 됩니다. 기본값인 0을 사용하면 Unity가 자동으로 크기를 결정합니다. 복잡한 UI의 경우에는 이 값을 수동으로 늘리면 드로우 콜 횟수를 줄여서 성능을 개선할 수 있습니다.

한 가지 예를 들면 다음과 같습니다. 이 UI에 있는 여러 개의 요소는 하나의 버텍스 버퍼에 들어갈 수 없습니다. Frame Debugger에서 드로우 콜이 하나가 아니라 두 개인 것을 볼 수 있습니다.

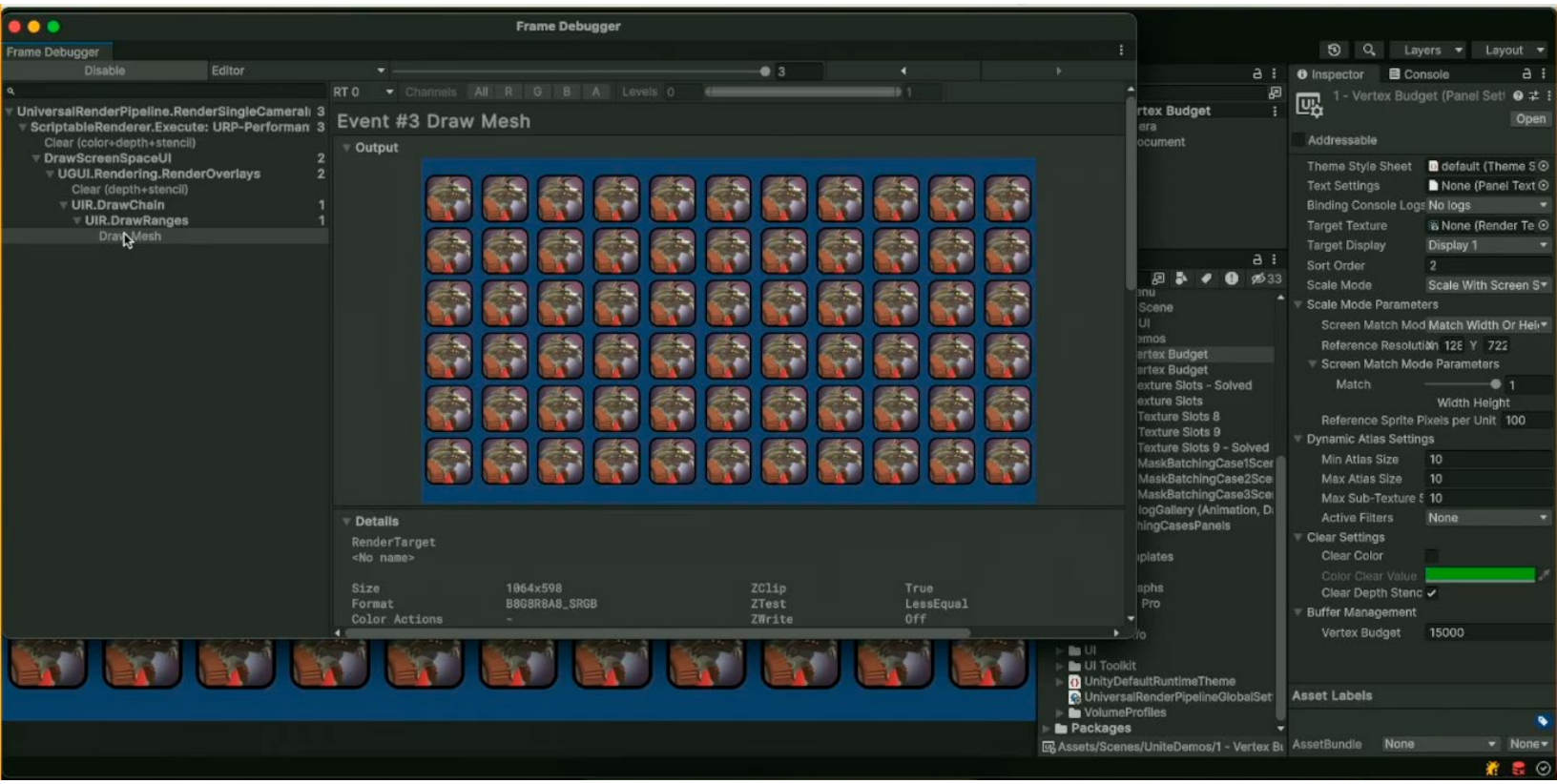


이 UI에는 둘 이상의 드로우 콜이 필요합니다.



예를 들어 Vertex Budget의 값을 버텍스 20,000개로 늘리면 framebuffer가 UI 요소들을 하나의 드로우 콜에 넣을 수 있습니다. 이렇게 하면 설정을 하나만 변경하여 UI의 효율성을 높일 수 있습니다.

Vertex Budget을 늘리면 드로우 콜의 개수를 줄일 수 있습니다.



Vertex Budget을 조정하면 드로우 콜이 한 개로 복원됩니다.

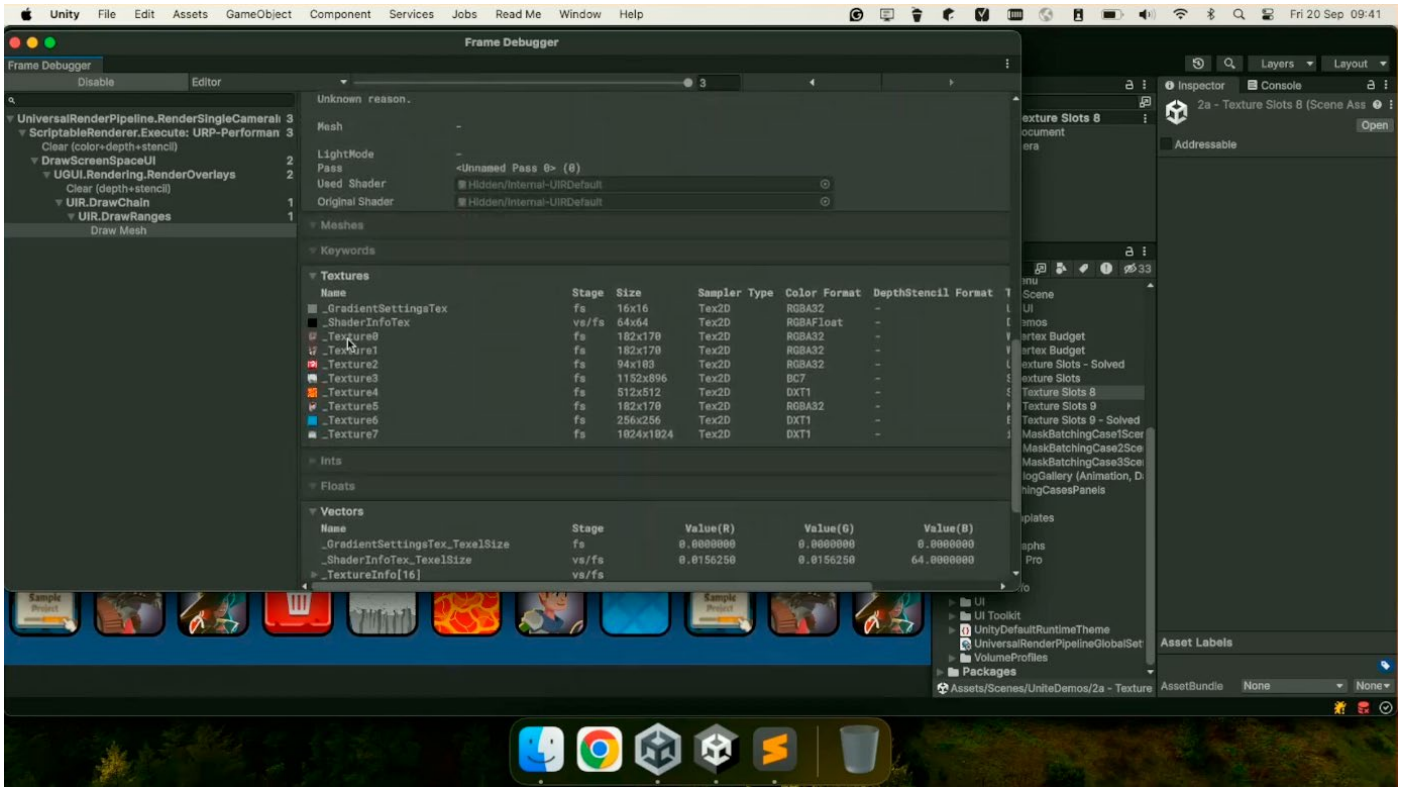
복잡한 UI의 경우에는 이 값을 수동으로 늘리면 드로우 콜의 개수를 줄여서 성능을 개선할 수 있으나, 메모리를 과잉 할당하지 않도록 주의해야 합니다. 프레임 디버거와 Unity 프로파일러를 사용하여 메모리 사용량과 드로우 콜 개수 사이에서 최적의 균형을 찾으세요.

우버 셰이더와 8개 텍스처 한도

UI 툴킷은 모든 UI 렌더링 기능을 하나의 다용도 '우버 셰이더'로 통합합니다. 이 셰이더는 여러 개의 셰이더 배리언트를 사용하는 대신 동적 브랜칭을 사용하여 런타임에 적절한 렌더링 경로를 선택합니다. 이로 인해 셰이더 전환이 최소화되어 CPU 오버헤드가 줄어들지만, 브랜칭 로직으로 인해 얼마간의 GPU 비용이 더해집니다.

이 셰이더는 같은 배치 내에서 최대 8개의 텍스처를 지원하는 강력한 기능을 갖추고 있습니다. 따라서 서로 다른 텍스처를 갖는 요소들을 하나의 드로우 콜로 렌더링할 수 있습니다. 다음 페이지의 이미지에서 서로 다른 8개의 텍스처로 구성된 UI를 볼 수 있습니다.

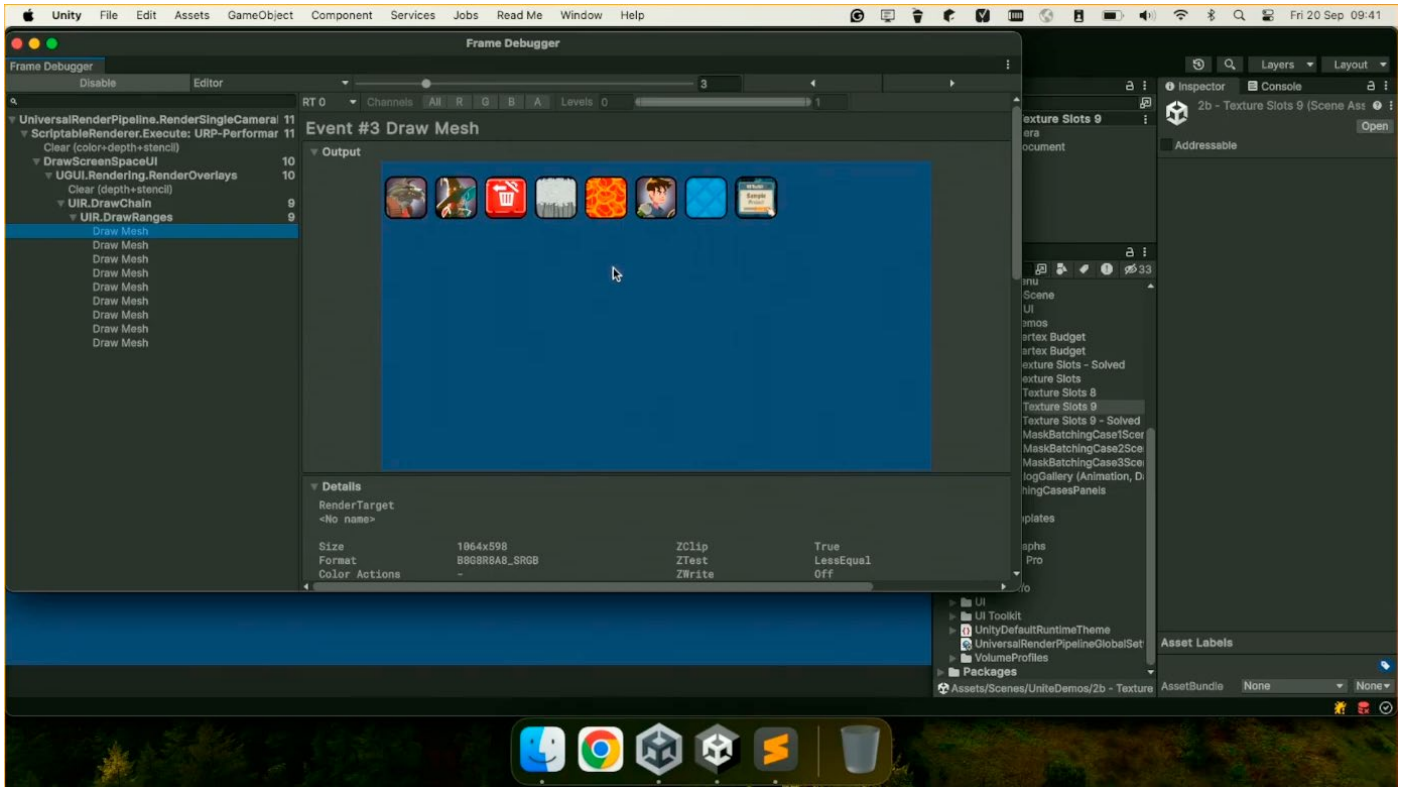
이 예시 UI에는 8개의 텍스처가 있습니다.



'우버 셰이더'가 8개의 텍스처를 하나의 드로우 콜로 렌더링합니다.

Frame Debugger에서 볼 수 있듯이 Unity는 최대 8개의 텍스처에 하나의 드로우 콜을 사용하여 예시 UI를 렌더링합니다. 단, 8개 텍스처 한도를 초과하면 배칭 시스템이 여러 개의 배치로 분할되기 때문에 오버헤드가 늘어납니다.

8개 텍스처 한도를 초과한 경우를 살펴보겠습니다. 드로우 콜이 여러 개로 늘어난 것을 볼 수 있습니다.



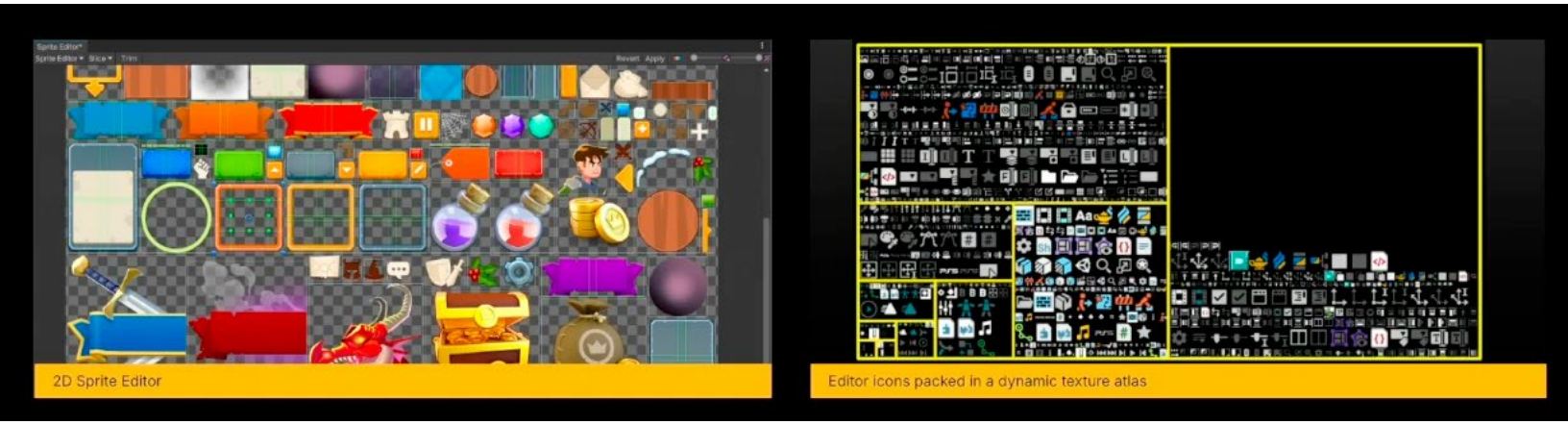
텍스처의 개수가 너무 많으면 배치가 분할됩니다.

이 제한에 대응할 수 있도록 UI 툴킷은 텍스처 사용을 최적화하는 툴을 제공합니다. 예를 들어 다수의 텍스처를 아틀라스로 통합하면 텍스처 수를 지원 한도 내로 유지하는 데 도움이 되며, 배치 효율을 유지하고 드로우 콜을 줄일 수 있습니다.

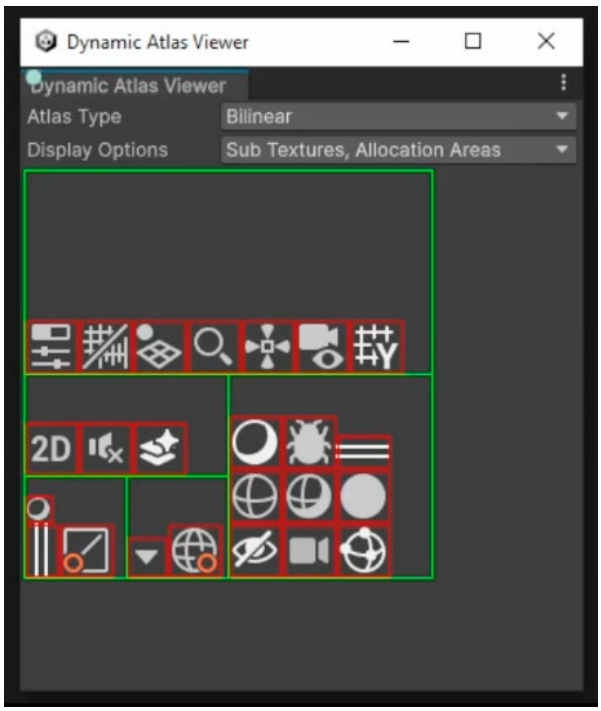
동적 텍스처 아틀라스

여러 텍스처 간에 전환하는 작업은 UI 툴킷이 배치를 분할하도록 강제하고 드로우 콜의 개수를 늘리며 성능을 저하할 수 있습니다. 이 문제에 대한 일반적인 해결 방법은 여러 개의 작은 텍스처를 하나의 큰 텍스처로 결합하는 텍스처 **아틀라스**입니다.

2D Sprite 아틀라스에 익숙하다면 이미 성능을 효과적으로 향상하는 방법을 알고 있는 것입니다. Unity는 여러 개의 스프라이트를 하나의 스프라이트 아틀라스로 패킹하여 하나의 텍스처로 취급하고 배치 분할과 드로우 콜을 줄입니다. 2D Sprite 아틀라스는 UI 툴킷과 매끄럽게 통합되므로 정적 콘텐츠 또는 사전 정의된 콘텐츠에 사용하기에 좋습니다.
단, 2D Sprite 아틀라스에는 런타임에 생성된 텍스처를 처리할 수 없다는 제한이 있습니다. 또한 사전 설정과 스프라이트 레이아웃 구성이 필요하므로 시간이 소요될 수 있습니다.



드래곤 크래셔 샘플은 2D Sprite 아틀라스를 사용합니다.



Frame Debugger의 Dynamic Atlas Viewer를 사용하세요.

UI 툴킷의 동적 텍스처 아틀라스는 여러 개의 이미지를 하나의 텍스처로 효율적으로 병합하여 텍스처 상태 변경을 줄입니다. Panel Settings에서 아틀라스 설정을 구성하고 (UI Toolkit Debugger 창의) Dynamic Atlas Viewer에서 아틀라스 레이아웃을 시각화할 수 있습니다.

UI에 다량의 변경 사항이 적용될 경우(예: 시간의 흐름에 따라 다량의 텍스처가 추가 또는 제거됨) 아틀라스가 분할될 수 있습니다. 이 경우 ResetDynamicAtlas API를 사용하여 아틀라스를 초기 상태로 복원할 수 있습니다.

UI 툴킷에서 2D Sprite 아틀라스와 동적 텍스처 아틀라스를 나란히 두고 사용할 수 있다는 사실을 기억하세요. 스프라이트 아틀라스는 사전 정의된 정적 콘텐츠에 적합하고, 동적 아틀라스는 UI 콘텐츠가 런타임에 변경되는 경우에 적합합니다.

마스킹

UI 툴킷은 스텐실 버퍼를 사용하여 마스크(UI 요소의 일부분을 표시하거나 숨기는 영역)를 생성합니다. 스텐실 버퍼는 GPU 상태의 일부이므로 마스크 설정을 변경하면 UI 툴킷이 배치를 분할하게 될 수 있습니다.

마스크 요소를 계층 구조로 중첩할 경우, 중첩된 뎁스 하나마다 스텐실 버퍼가 추가 상태를 트래킹해야 하므로 복잡성이 증가하여 GPU 워크로드가 늘어납니다.

UI 툴킷은 두 가지 유형의 마스킹을 지원합니다.

— **직사각형 기반 마스킹:** 직사각형 마스크는 셰이더 기반 연산을 사용하여 GPU 상태 변경 없이 배치 일관성을 보존합니다. 이 기법에서는 스텐실 버퍼가 사용되지 않으므로 뎁스 제한 없이 직사각형 마스크를 중첩할 수 있습니다.

— **둥근 모서리와 복잡한 마스크(스텐실 버퍼):** 둥근 모서리와 그 밖의 복잡한 셰이프에는 스텐실 버퍼 연산이 요구되기 때문에 각 마스킹 레벨에서 배치가 분할될 가능성이 있습니다. 이 기법은 최대 7개의 중첩된 마스킹 레벨을 지원합니다.

직사각형 마스크와 둥근 모서리 마스크

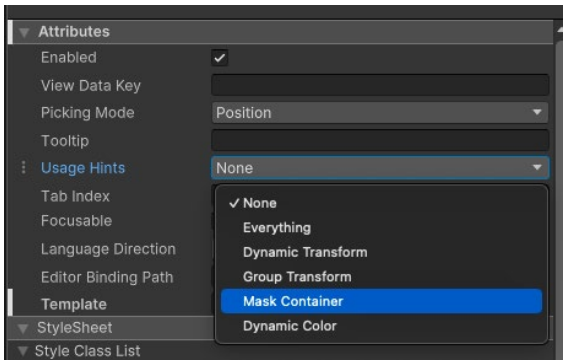
마스킹된 요소의 성능을 최적화하는 방법은 다음과 같습니다.

— 가능한 한 직사각형 마스크를 사용하여 스텐실 연산을 피합니다.

— 마스크의 중첩 뎁스를 최소화합니다. 계층 구조에서 마스크를 평평하게 유지하면 스텐실 재계산 횟수가 줄어듭니다.

— 가능한 한 자식 요소 위에 여러 개의 마스크를 사용하는 대신 부모 요소 위에 하나의 마스크를 사용하세요.

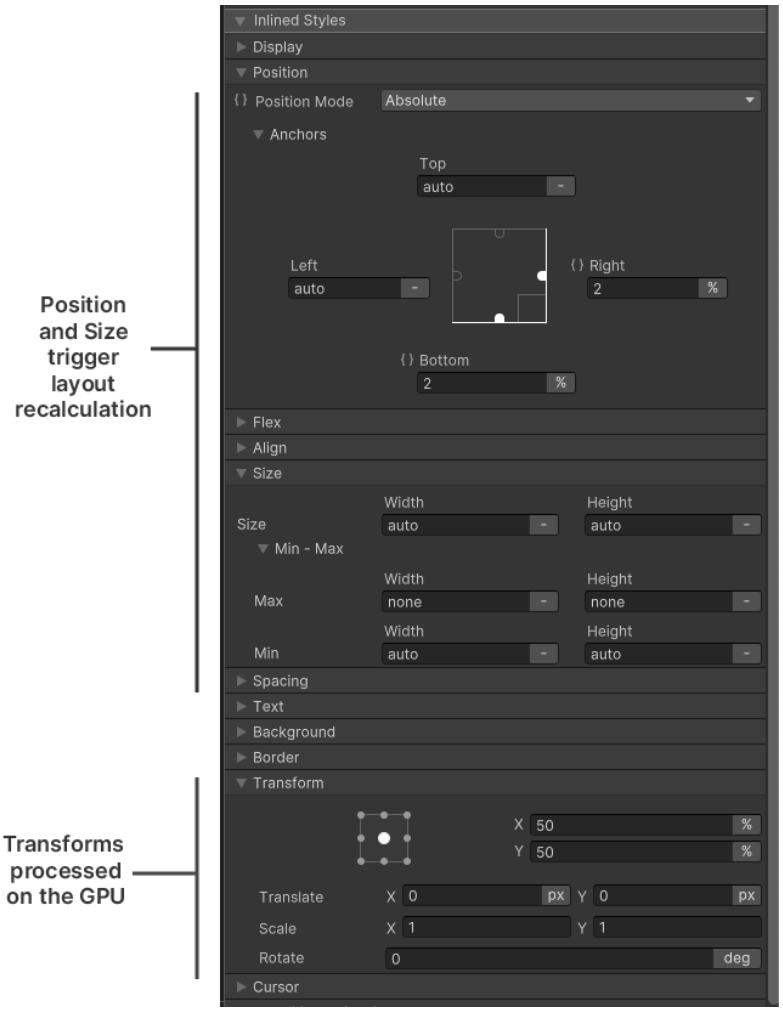
— 여러 개의 마스킹 레이어를 사용해야 한다면 Usage Hints에서 Mask Container를 설정하여 스텐실 상태 설정을 최적화합니다. 단, 배치 분할을 방지하려면 가급적 최소한으로 사용하세요.



Usage Hints에서 Mask Container를 선택합니다.

마지막으로, Frame Debugger에서 최적화의 영향을 검토하여 렌더링과 배칭이 효율적인지 확인하세요.

애니메이션과 전환



레이아웃 프로퍼티 대신 변환 기반 애니메이션을 사용하세요.

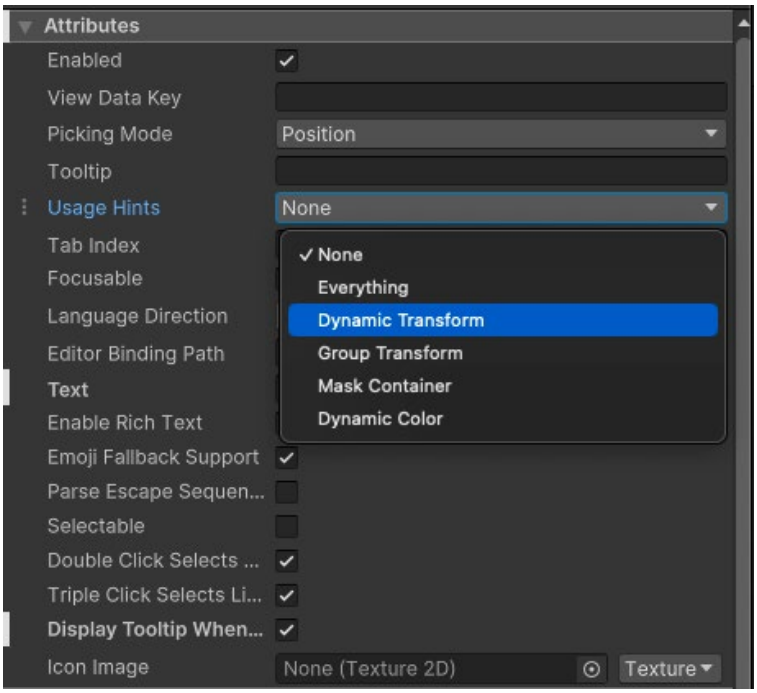
UI 툴킷의 USS 전환은 간단한 프로퍼티 애니메이션을 제공하긴 하나, 크기 또는 위치와 같은 레이아웃 프로퍼티를 변경하면 비용이 많이 드는 레이아웃 재계산이 트리거될 수 있습니다. 애니메이션을 최적화하고 성능 오버헤드를 줄이기 위해 몇 가지 전략을 시도해 볼 수 있습니다.

첫째, 레이아웃 프로퍼티를 변경하는 대신 변환 기반 애니메이션을 사용합니다. width, height, top, left와 같은 프로퍼티에 애니메이션을 적용하는 대신 translate, scale, rotate 변환을 사용하세요. 이들 연산은 GPU에서 직접 처리되므로 레이아웃을 재계산할 필요가 없어 애니메이션이 더 매끄러워집니다.

둘째, 애니메이션을 적용해야 하는 시각적 요소에 대해 **Usage Hints**를 활성화합니다.

DynamicTransform 힌트는 UI 툴킷에 위치 및 변환 업데이트를 GPU에서 처리하라고 지시하여 비용이 많이 드는 버텍스 데이터 재계산을 우회합니다.

애니메이션 자식이 여러 개 있는 부모 컨테이너의 경우, **GroupTransform** 힌트를 사용하면 오버헤드를 크게 줄일 수 있습니다. GroupTransform 힌트는 부모에 하나의 변환을 적용하고 이를 GPU가 모든 자식 요소에 효율적으로 전파하여 대규모 그룹의 애니메이션을 최적화합니다.



Usage Hint는 각 시각적 요소에 대해 제공됩니다.

셋째, 애니메이션 중에는 대규모 계층 구조에서 스타일 변경을 위해 클래스를 전환하지 않습니다. 클래스 변경은 특히 복잡한 UI 구조에서 과도한 스타일 재계산을 트리거할 수 있습니다. 대신 인라인 프로퍼티 변경을 통해 스타일을 직접 업데이트하여 계산 비용을 최소화하세요.

마지막으로, Unity의 프레임 디버거를 사용하여 애니메이션 성능을 모니터링합니다. 이 툴에서 애니메이션이 의도한 대로 작동하고 있는지 확인할 수 있습니다.

런타임 데이터 바인딩

Unity의 런타임 데이터 바인딩은 UI 요소가 기반 데이터의 변경을 자동으로 반영하도록 하여 UI 요소 업데이트를 간소화합니다. 수동으로 업데이트할 필요가 없어 UI 개발의 효율과 관리 용이성이 향상됩니다.

이 프로세스를 최적화하는 몇 가지 기법은 다음과 같습니다.

프로퍼티 백과 소스 생성

프로퍼티 백은 특정 유형 데이터의 효율적인 통과와 조작을 지원하는 컴패니언 오브젝트입니다. Unity는 기본적으로 유형이 처음으로 액세스될 때 리플렉션을 사용하여 프로퍼티 백을 생성합니다. 리플렉션 접근 방식은 편리하긴 하나, 프로퍼티 백이 아직 등록되지 않은 경우에만 작동하기 때문에 약간의 런타임 오버헤드가 추가됩니다.

성능을 개선하려면 프로퍼티 백에 대해 코드 생성을 활성화할 수 있습니다. 유형에 [Unity.Properties.GeneratePropertyBag] 태그를 적용하고, 어셈블리에도 코드 생성 태그를 적용합니다. 그러면 Unity가 컴파일 시 프로퍼티 백을 생성하고 등록하기 때문에 런타임 리플렉션이 필요하지 않습니다. 자세한 내용은 프로퍼티 백 기술 자료를 참조하세요.

GeneratePropertyBag 속성은 유형 전체를 최적화하지만, 개별 프로퍼티에 CreateProperty 속성을 추가하면 Unity가 컴파일 시 바인딩 코드를 생성합니다. 그러면 런타임 리플렉션이 패턴을 발견하고 연결할 필요가 없어 데이터 바인딩의 속도와 효율이 향상됩니다.

많은 경우 [CreateProperty]만 사용하는 것으로도 런타임 데이터 바인딩을 최적화할 수 있습니다. 단, 유형에 효율적인 직렬화나 모든 프로퍼티의 잦은 통과 등 추가적인 최적화가 필요하다면 [CreateProperty]와 [GeneratePropertyBag]을 함께 사용하여 전체적인 성능을 향상할 수 있습니다.

변경 트래킹

런타임 데이터 바인딩에는 데이터 바인딩의 업데이트 빈도를 최적화하는 두 가지 인터페이스가 있습니다.

- IDataSourceViewHashProvider: 이 인터페이스는 해시 기반 동등성 검사를 사용하여 데이터가 유의미하게 변경된 경우에만 바인딩이 업데이트되도록 합니다.
- INotifyBindablePropertyChanged: 이 인터페이스는 특정 프로퍼티 값이 변경된 경우에만 업데이트를 트리거합니다.

위의 두 가지 인터페이스는 데이터가 유의미하게 변경되지 않으면 불필요한 업데이트를 방지하므로 복잡한 UI에 특히 유용합니다.

다음 ScriptableObject에서 이러한 최적화를 구현하는 샘플을 볼 수 있습니다.

```
[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;
}
```

이 예시 클래스는 [GeneratePropertyBag]을 사용하여 컴파일 시 프로퍼티 백을 생성하고 [CreateProperty]를 사용하여 Name 프로퍼티의 런타임 데이터 바인딩을 최적화합니다.

여기서는 변경 트래킹을 위해 `INotifyBindablePropertyChanged`를 구현합니다. `Notify` 메서드는 `Name`이 업데이트될 때마다 `propertyChanged` 이벤트를 트리거합니다. 그러면 UI와 리스너에 변경 사실이 전달됩니다.

이 예시 클래스는 `IDataSourceViewHashProvider`도 구현합니다. `GetViewHashCode` 메서드는 업데이트를 쉽게 감지할 수 있도록 `Name`이 변경될 때마다 하나씩 늘어나는 버전 해시를 반환합니다.

요소 표시 및 숨기기

UI 요소를 숨길 때 불투명도를 변경하거나 화면 밖으로 옮기는 것은 성능 측면에서 좋은 옵션이 아닐 수 있습니다. 요소는 숨겨진 상태에서도 레이아웃 계산, 스타일 업데이트 및 데이터 바인딩 연산에 포함되기 때문에 성능에 영향을 줄 수 있습니다.

UI 툴킷에는 요소를 숨기는 몇 가지 방법이 있으며, 저마다 장단점이 있습니다. 아래 표에 정리된 내용을 살펴보세요.

Method	Bindings update	Layout update	Render cost	Style evaluation	Change cost	Styles memory	Meshes memory
Opacity: 0	Yes	Yes	High	Yes	Low	Yes	Full
Off-screen	Yes	Yes	Medium	Yes	Low	Yes	Full
Visibility: Hidden	Yes	Yes	Medium	Yes	Medium	Yes	Stencil
Display: None	Yes	No	None	No	Medium	Yes	Full
Hierarchy removal	No	No	None	No	High	No	None

요소를 토글하는 여러 방법

UI 요소를 숨길 때 opacity를 0으로 설정하거나 요소를 화면 밖으로 옮길 경우, GPU와 레이아웃 시스템은 여전히 해당 요소를 인식할 수 있으므로 중간~높음 정도의 렌더링 비용이 발생합니다. 이 방법은 전환에는 유용하지만 메모리나 레이아웃 오버헤드를 줄이지는 않습니다.

요소의 `visible` 프로퍼티를 false로 설정하면 렌더링은 방지되지만 여전히 레이아웃의 일부로 유지됩니다. 이는 일시적으로 요소를 숨기면서 계속해서 스텐실 메모리를 사용하는 절충안입니다.

성능 효율을 더 높이려면 `style.display` 속성을 `DisplayStyle.None`으로 설정하여 렌더링과 레이아웃 업데이트를 완전히 중단하는 것이 좋습니다. 단, 이렇게 하면 요소를 다시 표시할 때 레이아웃을 재계산하는 비용이 발생합니다.

다이얼로그 상자나 설정 패널처럼 드물게 표시되는 요소는 `RemoveFromHierarchy`를 사용하여 계층 구조에서 제거하면 오버헤드를 줄일 수 있습니다. 단, 요소를 다시 추가할 때 레이아웃을 처음부터 다시 구축해야 하므로 더 큰 성능 스파이크를 유발할 수 있다는 점에 유의하세요.

요소를 토글해야 하는 빈도에 따라 숨기기 방법을 선택하고, 단기적인 렌더링 요구 사항과 장기적인 성능 사이의 균형을 유지하세요.

오버드로우

UI 툴킷은 투명도를 사용하여 요소를 렌더링합니다. 그렇기 때문에 요소가 중첩될 경우 각 픽셀이 여러 번 처리될 수 있어 다량의 오버드로우가 발생할 수 있습니다. 중첩된 요소의 각 레이어에 복잡도를 더하는 UI 툴킷의 우버 셰이더를 사용할 경우 비용이 크게 늘어날 수 있습니다. 투명 또는 반투명 요소들의 여러 레이어를 중첩하면 성능에 한층 더 영향을 줄 수 있습니다.

오버드로우가 성능에 미치는 영향을 완화할 수 있는 몇 가지 전략이 있습니다.

— `style.opacity = 0`을 사용하면 요소가 여전히 투명하게 렌더링되므로 대신 `style.display = DisplayStyle.None`을 사용하여 요소를 완전히 숨깁니다.

— 여러 개의 요소를 중첩하는 대신, 완전히 가려진 요소를 제거하거나 숨깁니다.

— 스크롤 가능한 콘텐츠를 사용할 때는 ListView를 통해 가상화를 구현합니다. ListView는 화면에 보이는 요소만 효율적으로 렌더링합니다.

— `style.overflow = Overflow.Hidden`을 설정하여 콘텐츠를 특정 영역에 맞게 자르면 눈에 보이는 경계 밖의 불필요한 렌더링을 줄일 수 있습니다.

메모리 관리

USS 파일과 UXML 파일은 폰트, 텍스처 및 기타 에셋을 직접 참조합니다. 이러한 파일을 로드하면 참조된 모든 에셋이 메모리에 로드되어 메모리 사용량이 증가할 수 있습니다. 다음 그림에서 예시 USS에 의해 참조된 에셋을 볼 수 있습니다.

USS는 에셋을 참조합니다.

에셋은 현재 사용 중이 아니어도 임포트되는 즉시 메모리를 소비합니다. 따라서 에셋이 올바르게 관리되지 않을 경우 불필요한 메모리 사용이 발생할 수 있습니다.

에셋 사용을 최적화하려면 다음과 같은 전략을 고려해 보세요.

- **에셋 번들 또는 어드레서블 사용하기:** 가능한 한 특정 씬 또는 컨텍스트에 필요한 UI 문서와 스타일 시트만 로드합니다. 이렇게 하면 메모리 사용량을 제한할 수 있습니다.
- **필요하지 않은 에셋 언로드하기:** UI 요소 또는 문서를 사용하지 않을 때는 `RemoveFromHierarchy`를 사용하여 계층 구조에서 제거합니다. 그런 다음 `Addressables.Release` 또는 `AssetBundle.Unload(true)`를 사용해 언로드하여 다른 연산에서 사용할 수 있도록 메모리를 해제합니다.
- **복잡한 UI는 선택적으로 로드하기:** 대규모 UXML 또는 USS 파일은 작은 모듈식 템플릿(VisualTreeAsset)으로 분할하여 필요에 따라 동적으로 로드합니다. UI에서 보이는 요소에 필요한 리소스만 로드하면 메모리 사용량을 낮은 수준으로 유지할 수 있습니다.

프로파일링 툴

Unity는 애플리케이션에서 발생하는 UI 성능 문제를 식별하고 해결하는 몇 가지 툴을 제공합니다.

Unity 프로파일러, UI 툴킷 디버거, 프레임 디버거는 성능 문제를 진단하는 데 필요한 기본 툴입니다. 이러한 툴을 사용하면 드로우 콜, 배치, 비용이 많이 드는 연산(레이아웃 재계산, 스타일 업데이트, 버텍스 버퍼 변경)을 분석할 수 있습니다.

UI 변경을 보다 세밀하게 살펴보려면 Panel Settings의 SetPanelChangeReceiver 메서드를 사용하세요. 이 메서드로 UI 변경 사항을 수신 대기하고 소스를 트래킹할 수 있습니다. 에디터 및 개발 빌드로 한정된다는 단점은 있지만, 속도 저하의 원인이 되는 UI 동작을 가려내는 데는 유용합니다.

다음은 UI의 모든 변경을 로깅하는 예시 스크립트입니다.

```
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}");
    }
}
```

인스펙터에서 위 코드를 게임 오브젝트에 추가하고 PanelSettings를 설정하세요. `OnVisualElementChange` 메서드는 시각적 요소가 변경(레이아웃, 스타일, 변환 등)될 때마다 트리거되어 콘솔 메시지를 로깅합니다. 콘솔 메시지를 보고 현재 UI의 어느 부분이 수정되고 있는지 파악할 수 있습니다.

Unity 6의 성능 개선 사항

Unity 6에는 에디터 및 런타임 환경에서 매끄럽고 반응성이 뛰어난 경험을 제공하기 위해 다양한 성능 개선 사항이 도입되었습니다.

- **이벤트 디스패치:** 쉽게 이해할 수 있고 속도가 두 배 빨라지도록 이벤트 디스패치 규칙이 간소화되었습니다.
- **메시 생성 개선**: 주요 개선 사항으로는 기존 요소 지오메트리의 생성을 잡 시스템 기반으로 변경한 것과, 벡터 API를 네이티브 구현으로 전환한 것 등이 있습니다. 이에 더해 텍스트 생성이 병렬로 실행됩니다.
- **커스텀 지오메트리 API**: 개발자는 새로운 공용 API를 사용하여 동일한 수준의 성능으로 커스텀 지오메트리를 생성하여 고도로 최적화된 UI 컴포넌트를 구성할 수 있습니다.
- **깊은 계층 구조의 레이아웃 성능**: 레이아웃 계산 캐싱 개선으로 깊게 중첩된 계층 구조의 성능이 크게 향상되고 매끄러운 사용자 경험이 제공됩니다.
- **대규모 데이터 세트에 맞게 최적화된 TreeView**: 기존에는 효율이 낮아 대규모 데이터 세트에서 사용하기에 어려웠던 TreeView 컨트롤이 엔티티에 맞게 새로운 고성능 백엔드로 개선되었습니다.

그 밖의 성능 최적화 리소스

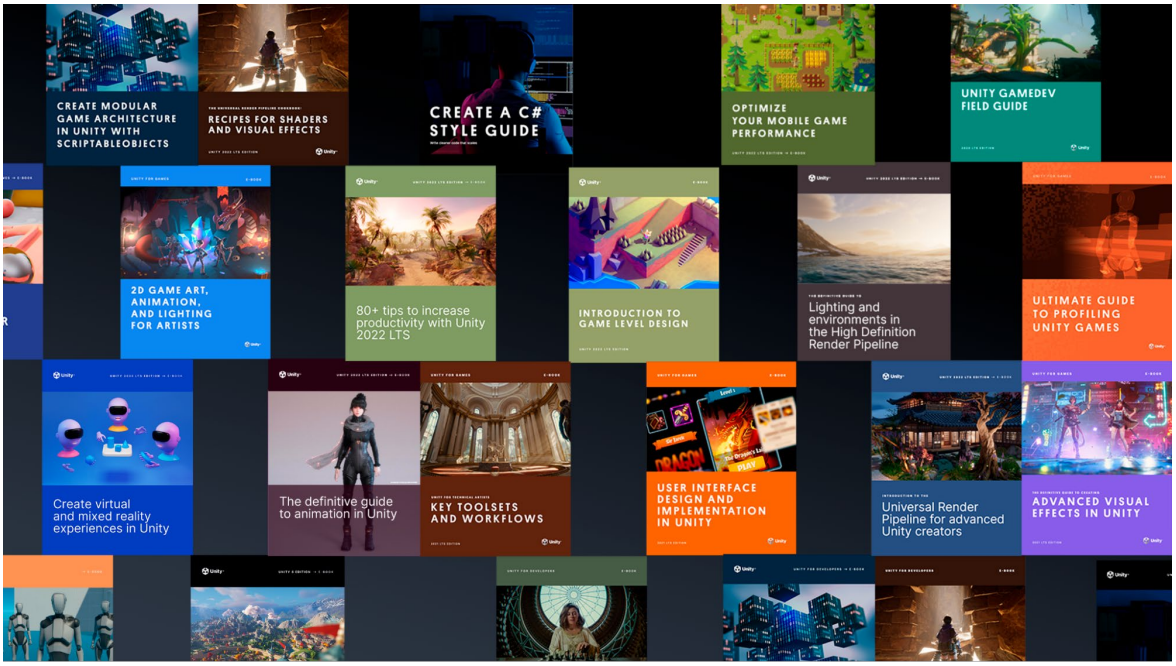
유나이트 2024: UI 툴킷으로 최고의 성능 구현하기(영문)

전자책: Unity 게임 프로파일링 완벽 가이드

전자책: Unity에서 모바일, XR, 웹용 게임 성능 최적화

전자책: Unity에서 콘솔 및 PC용 게임 성능 최적화

숙련된 개발자 및 아티스트를 위한 리소스



Unity 베스트 프랙티스 허브에서 숙련된 Unity 개발자 및 크리에이터를 위한 더 많은 전자책을 다운로드하실 수 있습니다. 업계 전문가, 유니티 엔지니어 및 테크니컬 아티스트가 제작한 30개 이상의 가이드에서 Unity의 툴셋과 시스템을 사용하여 효율적으로 개발하기 위한 베스트 프랙티스를 살펴보세요.

각종 팁, 베스트 프랙티스, 최신 소식은 Unity 블로그, UnityDiscussions, Unity Learn과 **#unitytips** 해시태그를 통해서도 확인할 수 있습니다.

unity.com/kr